Backend Developer in 30 Days

Acquire Skills on API Designing, Data Management, Application Testing, Deployment, Security and Performance Optimization

PEDRO MARQUEZ-SOTO

bpb

Backend Developer in 30 Days

Acquire Skills on API Designing, Data Management, Application Testing, Deployment, Security and Performance Optimization

PEDRO MARQUEZ-SOTO

bpb

Backend Developer in 30 Days

Acquire Skills on Api Designing, Data Management, Application Testing, Deployment, Security and Performance Optimization

Pedro Marquez-Soto

www.bpbonline.com

Group Product Manager: Marianne Conor

Publishing Product Manager: Eva Brawn

Senior Editor: Connell

Content Development Editor: Melissa Monroe

Technical Editor: Anne Stokes

Copy Editor: Joe Austin

Language Support Editor: Justin Baldwin

Project Coordinator: Tyler Horan

Proofreader: Khloe Styles

Indexer: V. Krishnamurthy

Production Designer: Malcolm D'Souza

Marketing Coordinator: Kristen Kramer

First published: August 2022

Published by BPB Online

WeWork, 119 Marylebone Road

London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55513-212

www.bpbonline.com

Dedicated to

My beloved wife Alejandra

&

My sons Santiago and Bruno

About the Author

Pedro Marquez Soto is a full-stack software developer with a Master of Science in Computer Science and Machine Learning. He has more than 10 years of professional experience in multiple roles that cover application security, back-end, front-end development, and infrastructure development. He currently works as a full-stack engineer at LinkedIn.

Acknowledgement

There are a few people I want to thank for the continued and ongoing support they have given me during the writing of this book. First and foremost, I thank my wife and friend Alejandra for her unconditional love and support in all my professional ventures. This book was only possible because she believed in me; she had the patience of having a husband who spent hours locked in the office, writing.

I wrote this book in the middle of a pandemic while at the same time I was completing my master's degree. The only thing that kept me on track was having the safe space of a loving family and their support for me to pursue my passions.

I want to thank my mom, Josefina, who has worked tirelessly to provide her kids with an education; without all her hard work and love I would have never had acquired the knowledge needed for writing this book. I also want to thank my father Pedro for teaching me to never give up, especially when life gets hard.

I am grateful to all the official and unofficial mentors I've found during my lifetime. Managers and other smart people I've met in the past decade who gave me the opportunity of working on interesting projects that led to constant professional improvement. These people were honest when my code was not as good as it could be and showed me how world-class software developers work.

My gratitude also goes to the team at BPB Publications for giving me the opportunity to write my first book and providing guidance during this process that was new to me. They gave me the right amount of freedom and support that allowed me to complete this daunting task. BPB Publications is the best home this book could have, and for that, I will be forever thankful.

Even if my name is on the cover of this book, this project is a team effort. All these people are co-authors, and no author should be ungrateful enough to forget that.

Preface

This book covers the process of building large-scale software applications from the point of view of a back-end developer. More than a tutorial on specific tools or frameworks, this book outlines principles and processes shared across the multiple tech stacks. Tools and frameworks change with time: they are replaced by newer stacks; the patterns, however, remain the same. The goal is to show you how things work and why they work the way they do.

This book connects different areas commonly isolated in educational material: API development, database integration, application security, and deployment processes. It gives an integral vision of how the largest software companies build software capable of serving millions of users. This vision benefits people starting their careers as software developers and those who have spent most of their careers working on only one of these areas.

This book has 12 chapters chronologically ordered to follow the phases of a software development project. Ideally, you could take this book with you and read across each chapter in order as your project evolves. Each chapter builds on the previous, so it is advised for you to read from start to end. However, each chapter is also self-contained enough for you to return to any of them when you need to refresh your knowledge.

The first part of the book covers the essential aspects of back-end development. It outlines the process of converting business needs into requirements, defining elegant APIs that are flexible enough to evolve with the application, and choosing the correct type of database. This first part also offers a deep dive into the inner workings of web-based applications, building a robust mental model that will allow you to fully grasp the abstractions built on top of them.

The second part covers patterns and processes needed to build quality into software applications: Testing, application security, error and log management, framework adoption, and continuous integration and deployment. These concepts are the basis for the daily work of software developers across the globe.

The third and last part of the book serves two purposes: Describe how to jump from simple apps to large-scale, distributed systems and concepts that you can use to advance your career and become a senior developer.

Chapter 1 covers the topics of problem-solving and requirements gathering. It provides background on the importance of software applications as tools to solve problems. This chapter gives you useful heuristics to successfully recollect business requirements in an iterative process. Then, the chapter provides a high-level view of large-scale software applications and all the components and tools that address the functional and non-functional requirements for the project.

Chapter 2 is a deep dive into web-based applications and the client-server architecture. It details the inner workings of web servers, guiding you to create your own Java-based web server and compare it with production-ready servers like NodeJS's Express. This detailed view helps you understand the "magic" behind these tools, which is a critical skill for becoming a senior developer.

Chapter 3 disambiguates the term "API". It explains what APIs are, why they are useful to back-end developers, and how they fit in the process of creating an application. Then, the chapter guides you on how to design flexible, data-driven APIs and how to choose the right tools to implement them: REST, GraphQL, and gRPC.

Chapter 4 will cover state and data management, which includes databases. It provides background on why we need databases, how to model real-life data, and how to choose the proper database for your business needs. This chapter covers the differences between SQL and NoSQL, and highlights the differences between the most common products for each, like relational, document, and graph databases.

Chapter 5 explores the area of testing. It details the manual and automated testing process and highlights the differences between concepts like mocks, stubs, and test doubles. This chapter also provides good practices for writing unit and integration tests and their differences. Then, it describes other areas of non-functional testing like performance and security testing.

Chapter 6 covers application security. This chapter defines the concepts used by application security experts that any software developer should know. It describes how to integrate authentication and authorization services into the application, including industry standards like OAuth2, SAML, and OpenID Connect. This chapter also covers some of the most common security vulnerabilities and advises how to prevent them.

Chapter 7 explains the topic of error and log management in software applications. It highlights how some coding languages deal with errors, how to find errors in applications deployed to production environments, and how to centralize and monitor errors in distributed applications using tools like Logstash, Elastisearch, and Kibana.

Chapter 8 covers a deep dive into application frameworks. The chapter explains what frameworks are, what they are used for, and the patterns they use to solve everyday challenges for back-end developers. It then introduces popular frameworks like Java's Spring and Hibernate and Python's Django. It contrasts how patterns like MVC are implemented in these frameworks and provides the groundwork for developers to pick the proper framework for their use case.

Chapter 9 describes how to deploy an application to a production environment. It gives some historical background on how applications have been delivered to users, the challenges faced, and how modern CI/CD (Continuous integration and deployment) flows enable development teams to deliver their applications fast. We also explore the concept of replicable environments through virtualization using Virtual Machines and Docker containers and how they integrate into the CI/CD process.

Chapter 10 presents advanced topics for creating large-scale, distributed applications. The chapter covers how to measure performance to find improvement opportunities. Then, we explore techniques to improve performance like caching, asynchronous architectures, and asynchronous programming through the concepts of "Promises" and "Futures". It presents some tools used to increase performance in large applications like Redis and Kafka.

Chapter 11 takes a step back and integrates every previous chapter into one process for designing a software application. The chapter describes a step-by-step approach to converting requirements into technical specs, estimating server and storage size requirements, and principles used for scaling apps. This process is a blueprint that is especially useful for system design interviews.

Chapter 12 the last chapter in this book, outlines career advice for back-end developers looking to become senior developers. It describes the typical responsibilities of both junior and senior developers, and it provides guidelines for preparing for tech interviews, finding mentors, and finding resources to keep increasing your technical knowledge.

Table of Contents

1. Building Multi-User Apps
- Structure
- Objective
- Digital transformation and a little history
    - *Digital transformation and the Internet*
        - *Software transformation*
- Designing apps to solve real-world problems
    - *Caring for user problems as a back-end developer*
    - *Finding a problem to solve*
- Define a sample use case: The Pizza Place ordering system.
- Defining functional and non-functional requirements
    - *An ineffective way of collecting requirements*
    - *The requirement definition cycle*
        - *Find out what is the problem your client is trying to solve*
        - *Get a detailed picture of the business.*
        - *Request the help of one of your client's domain-experts*
        - *Learn how the existing process works today*
        - *Build prototypes and revise requirements*
        - *Use case: Defining requirements for the Pizza Place*
- The modern system design: a ten thousand-feet view
    - *Getting the front-end out of the way*
        - *Building blocks*
- Conclusion
- Questions

2. The Client-Server Architecture
- Structure
- Objectives
- Architecture details
    - *Abstraction layer: Frontend client and backend client*
    - *Abstraction layer: Data access service client, database server*
- HTTP: The language of the web

Implementing a web server
The main process
Serving a response
Multi-user support with multi-threading
Using a production-ready server
Enabling HTTPS in Express
Layered architecture: Fully splitting the client from the server
Splitting clients
Client versus server computing
Web servers as stateless services
Storing session data
Use case: Applying a client-server architecture to the Pizza Place app
Client server versus peer-to-peer
Conclusion
Questions
References

3. Designing APIs
Structure
Objectives
What is an API?
Functions as contracts
Interfaces and design patterns
Remote APIs: RPC, SOAP, REST, and GraphQL
Building a remote API with RPC/gRPC
Build the gRPC server
Build the gRPC client
When to use gRPC
SOAP and web services
Building REST APIs
Actions versus HTTP request methods
Naming resources
Singular versus plural
Relationships
Versioning
Caching
Effective REST APIs: HATEOAS

Building APIs with GraphQL
Building standalone APIs
Standalone API: Headless CMS
Standalone API: Public APIs
Use case: Designing a remote API for the Pizza Corner
Conclusion
Questions
References

4. End-to-End Data Management
Structure
Objectives
Defining the application state
Hardware storage
Understanding in-memory data storage
In-memory cache
In-memory databases
Simple storage in text and binary files
Understanding complex data storage
SQL or NoSQL?
Document databases
High locality
Few relationships
Unstructured data
Relational databases
Models are related but independent
Multiple relationships
Fixed structure
Normalization
Graph databases
Data with a lot of connections
Data where the relationships are first-class citizens
Scalability
File storage repositories
Beyond technical requirements
Indexing
Reducing time complexity

Example: Benchmark the impact of indexes in SQLite
Backup and recovery
Backup database files
Creating backups with activity logs
Backup through replication
Tackling gaps in backups
Designing data storage in a production system
Choosing a deployment strategy
Database and application share a server
Deploy database in its own server(s)
Embedded databases
Combining databases to approach complex use cases
Use Case: Defining a data model for the Pizza Place application
Requirement: Users should be able to see the menu on their phones or computers
No attributes, fixed size of ingredients
No attributes, dynamic list of ingredients
With attributes, dynamic size
Choosing a data store for storing the pizza menu
The winner
Scaling the database
Conclusion
Questions
References

5. Automating Application Testing
Structure
Objectives
Certainty through testing
Manual testing
Types of manual tests
Building effective manual tests
Creating a test plan
Executing the test plan
Creating detailed reports
Validating fixes
Update test plan

Advantages of manual testing
Explorability
Horizontal validation
User-centered vision
Manual testing and Agile
Let others test your code
Automated testing
Unit testing
Testing in isolation: Doubles, stubs, and mocks
Isolated code is easier to debug
Isolating test dependencies
Test mocks
Stubs versus mocks
Coverage
To use coverage or not to use it
Test-driven development
Integration testing with Selenium
Defining a test environment for integration testing
Simulate a test environment close to production
Testing and CI/CD
Other automated tests: Static code analyzers
Defining effective test cases
Defining a single use case per test
Do not mock everything
On equal conditions, prefer unit tests over integration tests
Non-functional testing
Application security and penetration testing
Load testing
Performance testing
Accessibility testing
Conclusion
References

6. Securing Applications
Structure
Objectives
The CIA triad: Confidentiality, Integrity, and Availability

- *Confidentiality*
  - *Loss of confidentiality*
- *Integrity*
  - *Loss of integrity*
- *Availability*
  - *Loss of availability*

Access Control: Authentication and authorization

- *Authentication*
  - *Identification: Username and password*
  - *Problems with passwords*
  - *Best practices for passwords*
  - *Identification: Multi-factor authentication*
  - *Single Sign-On*
- *Authorization*
  - *Roles and groups*
  - *Least privilege principle*

Use case: Implementing basic authentication and authorization for the Pizza Place

- *Identify user roles*
- *Building an authentication service*
  - *Define user management storage*
  - *Define password-related protections*
  - *Build web forms for signup and login*
  - *Apply authorization controls*
  - *Translate a high-level authorization map to implementation details*
  - *Using scopes to check authorization*
  - *Test access control*

Federated authorization

- *Pros and cons of federated authentication*
- *Security Assertion Markup Language (SAML)*
- *OpenID*
  - *OAuth2*
  - *Terms*
  - *Request an authorization code*
  - *Request an access token*
  - *The implicit flow*

Building OpenID from OAuth2
Building security into the application’s design
Creating a Threat Model
Decompose the application
Determine and rank threats
Determine countermeasures and mitigation
OWASP Top 10: The most common vulnerabilities
Conclusion
Questions
Resources

7. Handling Errors
Structure
Objectives
Why do we need to handle errors?
Understanding common causes of errors
Types of error handling
Exceptions
Using stack traces to debug problems
Defining exception types
Catching exceptions
Errors validated at compile time
Errors as return values
Implementing good exception handling
Preventing all the errors we can
Handling and mitigating errors in production
Defining good error messages
Bubble up!
Providing a fallback
Letting the error propagate
Finding production errors with logging
Anatomy of a log entry
What to log?
Designing good log and error messages
Persisting log entries to file
Handling errors in distributed systems
Using case: Logging errors with the ELK stack

- *Logging errors locally*
- *Configuring Logstash*
  - *Logstash input*
  - *Logstash filter*
  - *Logstash output*
- *Installing and configuring Elasticsearch and Kibana*
- *Creating Kibana dashboards*

A/B testing and gradual deployment
- *Creating a deployment plan*

Conclusion

Resources

8. Adopting Frameworks

Structure

Objectives

What problems do frameworks fix?
- *Solving existing problems*
- *Frameworks and design patterns*
- *Libraries and frameworks*
  - *Pre-building abstractions*
- *Framework's benefits*

Common patterns addressed by frameworks
- *Automation tools and package managers*
  - *Automation tools*
  - *Native package management*
- *Handing web requests (e.g. Spring MVC, Django)*
  - *How frameworks are born: The use of the Spring framework*
  - *About MVC*
  - *Dependency Injection*
  - *Spring's XML configuration*
  - *Code-based configuration*
  - *Annotation-based configuration*
  - *Spring MVC*
  - *More MVC: Express*
  - *The downfall of MVC*
- *Database access with ORMs*
  - *JPA and Hibernate*

Mapping tables to entities
Inserting and querying data
More ORMs: Python’s Django ORM
The downsides of ORM
Choosing a framework
The impact of community
When not to use frameworks
Learning the framework instead of using the basics
Adding debugging complexity
Zero-cost abstractions
Conclusion
References

9. Deploying Applications
Structure
Objectives
Defining a robust deployment process: CI/CD
Before CI/CD
A step forward: Deployment scripts
The advantages of a CI/CD pipeline
Creating reproducible environments
Moving out of shared environments
Advantages of isolated and reproducible environments
Version control
Git
Creating a Git repository
Staging and committing new files
Making changes to existing files
Commit details
Branches
Merge
Remote repositories
Cloning repositories
Merging conflicts
Using Git hooks
Git to enforce reproducible code
SVN and other CVS

Virtual machines
Virtual infrastructure
Containers (Docker)
VMs versus containers
Working with stateless containers
Use case: Creating a reproducible deployment environment for the Pizza Place app using Git and Docker
Setting up the application
The web directory
The nginx directory
The docker-compose.yml and nginx.conf files
Adding Git
Docker in CI/CD
The trade-off
Conclusion
Questions
Resources

10. Creating High-performance Apps
Structure
Objectives
Measuring to improve the performance
Synthetic testing versus RUM
Using percentiles
Improving the performance
Improving the performance with caching
Defining reading and writing load
Cache patterns
Cache-aside and read-through cache
Caching write-heavy applications
Write-through cache
Write-behind cache
Choosing the right caching strategy
Eviction policy
Other caching tools: Proxies and CDNs
Use case: Caching long-running operations with Redis
Using Jedis

Using Redisson
Improving the performance with distributed systems
Keeping data consistency
Data consistency in replicas
Multiple read replicas, single write replica
Eventual consistency versus strong consistency
Data consistency in sharding
Microservices
Improving performance using asynchronous communication (queues)
Improving the performance using asynchronous programming
Promises and futures
Conclusion
Questions
References

11. Designing a System
Structure
Objectives
The system design process
Example: The Pizza Place (at scale)
Defining and clarifying requirements
Defining the system's interface
Defining data models
Calculating the system scale and size
Estimating the storage size
Estimating QPS
Estimating the storage throughput
Calculating the cache size
When to stop estimating the size
Creating high-level and low-level designs
Defining a high-level design
Defining the use case flow
Finding gaps in assumptions
Defining a low-level design
Designing the client
Designing the web server

Designing the service layer
Designing the database layer
Defining a distributed database strategy
Defining read and write replicas
About scaling the file storage
Integrating each layer
Client to server
Web server to services
Services to database
Overall low-level design
Identifying failure points
Failure case: A web server goes down
Failure case: A microservice instance goes down
Failure case: The file storage service goes down
Failure case: A node in the service queue goes down
Failure case: A node in the cache layer goes down
Failure case: A database server goes down
Failure case: The infrastructure for a whole region goes down
Extra considerations about failure management
Conclusion
Questions
Resources

12. Bootstrap Your Career Path
Structure
Objective
Defining the expectations on junior developers
Joining our first development team
Following guidance from other developers
What makes a senior developer?
Characteristics of a senior software developer
Improving hard and soft skills
Improving technical skills
Improving people skills
Practicing conflict resolution
Working on communication skills
Preparing for technical interviews

- *Getting an interview*
    - *Building a good resume*
    - *The importance of networking*
- *Improving interview skills*
- *The technical interview process*
    - *Approaching coding problems*
    - *Using data structures efficiently*
    - *Using algorithms efficiently*
    - *Coding a solution*
    - *Getting better at coding interviews*
    - *Working on system design interviews*
    - *Take-home assignments*
    - *Approaching non-coding modules*
    - *Asking the right questions*

Finding mentors
Finding resources to keep learning
Conclusion
Resources

Index

CHAPTER 1
Building Multi-User Apps

Solving problems is a software developer's primary job; we dedicate our careers to gaining the skills we need to be efficient problem-solvers. We learn computer languages, frameworks, libraries, and design patterns to build a diverse toolset to tackle the great variety of problems this world offers.

In this chapter, we will talk about problem-solving: we will discuss how the problems human beings have found across history have evolved. We will see that technological advances are cyclical, and understanding these patterns will help us predict the problems we may find in the future and design better solutions.

We will also discuss what kind of problems we can fix today and how we can translate problem statements into technical challenges for which we can offer software-based solutions.

Then, we will move on to effectively capture all the requirements for a new project, describing some obstacles we may find in the process and good practices to sort them out.

Ultimately, we will have a high-level view of the variety of tools available to us to build software applications. In later chapters, we will explore these tools in detail, but we will use this chapter first to understand how they all fit together.

Structure

In this chapter, we will learn the following topics:

- Digital transformation and a little history
    - Digital transformation and the Internet
- Designing apps to solve real-world problems

    - Caring for user problems as a back-end developer
    - Finding a problem to solve
- Defining a sample use case: The Pizza Place ordering system.
- Defining functional and non-functional requirements
    - An ineffective way of collecting requirements
    - The requirement definition cycle
- The modern system design: a ten thousand-feet view

Objective

After completing this chapter, you should clearly understand the primary goal of software developers: To solve real-world problems.

By the end of this chapter:

- You will understand how to find good problems to solve as a developer.
- You will realize that we must find a problem to solve *before* trying to build an app.
- You will have a list of heuristics to efficiently define functional and non-functional requirements and know which practices to avoid.
- You will understand the tools available to us for building modern applications.

Digital transformation and a little history

During the second half of the 18th century, a transformation started in Europe: Goods crafted by hand began to be manufactured by machines. Powered by steam and later on by electricity, these machines allowed people to produce goods faster and for less money; with lower production costs, manufacturers could reach more consumers than ever.

This period, known as the Industrial Revolution, changed how the business operated and profoundly transformed society. Rural communities started to grow into urban spaces. Cities that were geographically apart increased communication and trade thanks to advances in transportation (like the creation and improvement of trains and steam-powered boats). Small communities started merging into larger ones.

People moved into cities as many agrarian jobs were replaced by industrial work. Change is never easy, but it was a specifically difficult transition for many people whose current skills were not always valuable for the new job market. Artisans who couldn't compete with the large production machines had to adapt if they wanted to keep providing for their families.

Jobs became more specialized, and workers needed instruction to operate the new machines, so new schools and apprenticeships were created. With people concentrated in urban communities, education also was streamlined. Formal education and professions became ubiquitous.

To speed everything up, money was also concentrated in cities where the manufacturing plant owners could now produce more for less. A concentration on resources lead to inequality by tut also allowed more investment in technological advances, which led to the improvement of the same machines that started this transformation.

A Second Industrial Revolution brought mass-production through electricity, and a Third Industrial Revolution brought automation through electronics and information technology. Each of these new phases was built on the top of the previous one, each one with a more extensive reach. Industrialization became digitalization.

People were forced to adapt in each of these revolutions, not always without resistance.

Note: As it happened then, people in the present have the same kind of fear of change and incertitude. People are afraid some professions can become obsolete due to advances in Artificial Intelligence, resulting in people losing their jobs.

The term "Luddite" was created to describe people who disliked new technology and its threat to their interests like job security and privacy.

Many make fun of people who dislike technological advances, but we seldom try to empathize. The industrial and digital revolutions have not been perfect nor socially fair. There is still much debate on how the industrial revolution improved people's quality of life beyond the

apparent riches it gave to the few owners of fabrics and their social circles.

Some anthropologic studies have found that inequality grew in the United Kingdom during the industrial revolution as "early industrialization disproportionately benefitted capital more than other factors of production":

https://www.econstor.eu/bitstream/10419/201834/1/1671601041.pdf

This same conversation about technology and inequality is happening today when CEOs and founders of technology companies are the wealthiest people in the world.

Feeling anxious about technological progress is expected. Understanding how people were historically able to adapt to new emerging social dynamics gives us an insight into how to prepare society for the future.

As the group that constantly introduces change, it's part of our job to make sure we do everything in our power to guarantee that technology advances take into account issues like inequality or privacy.

As the digital era started and technology became more than just steam machines, people's lives and needs also kept changing.

Digital transformation and the Internet

Suppose charted the amount of technological progress as a function of time. That chart would show that progress has been exponential since the start of the first industrial revolution. As time advances, we take higher technological leaps.

We have seen more technological changes in the last couple of decades than in the previous 200 years, and we're only starting. We now live in a digital world, and the advances in fields like Machine Learning, AI, and quantum computing are only accelerating our adoption of it.

Tip: CPUs are a good reference for how technology advances. Moore's Law, even as we are testing its limits lately, describes how the capacity of processors increases exponentially in time. Technological advancement has progressed like CPUs have: as time passes, the world changes at a faster pace; breakthroughs happen more often.

Since we as humans have lived this intense and accelerated transformation before, we can get answers to a few questions we have today. For instance, many people wonder what will happen when AI starts executing jobs that people have today. Still, as we just saw, people always find a way to adapt, even at the cost of an intense social change and a redistribution of resources.

Just as our needs and how we fulfill them changed during the first phases of the Industrial Revolution, the creation and wide adoption of the Internet have transformed humanity: The Internet is the platform on which the Third and Fourth Industrial Revolutions are built.

People's needs are now more complex and abstract. For instance, social needs go beyond interacting with peers in their direct communities; we are now part of social groups not limited by distance or time. People expect to be connected, even living in a different country or continent.

A whole generation of people now prefers to order food online than order using the phone or just walking into a restaurant. We pay our bills online. New movies and TV shows are being premiered online instead of through movie theaters or cable subscriptions. Podcasts are now as ubiquitous as radio. Music streaming has displaced the consumption of CDs.

Technology is still changing people's roles in society, and as software developers, we need to understand and take advantage of that knowledge. Just as when the farmer had to learn how to operate manufacturing machines, people creating and streaming digital content (e.g., YouTubers) are jobs that are increasingly replacing traditional professions (many of them are destined to be consumed by AI).

Software developers who have understood the new needs of our ever-changing society have built the tools on which these new professionals operate -making millions in the process.

Software transformation

The field of software development is everything but static. Few software developers get to work on the same technology all their careers. Adapting and learning new technologies is an essential requirement for our profession. The best software developers understand change and adapt to it.

The Internet itself has changed. The web stopped being the collection of unidirectional static content it once was; it’s now a highly dynamic platform that enables communication back and forth between the owners of the websites and the users. Modern applications provide value based on the information they receive from their users.

Our users themselves have changed: the Internet is not the place only for the technically savvy, the academics, or the hobbyists that it once was. Thanks to years of hard work from brilliant people and experts in Human-Computer interaction, the Internet is now a democratic space where you don’t need any understanding of how software is built to use it. Users are now grandparents, professionals from any area of specialization, and even toddlers.

The scale of software has increased due to the expansion in the use of the Internet. Web servers now have so many users that a single powerful computer is not enough to serve requests to all of them. The data that modern applications consume don’t fit in a single machine, even considering the high-capacity hard drives we’ve been able to develop in recent years.

The software development process keeps changing, too: We don’t build code only for computers anymore; we build applications that now run on smartphones, TVs, and even toasters or fridges. And you don’t need a Ph.D. or even a college education to build code: Teenagers are making millions of dollars selling apps they create in their spare time.

One thing doesn’t change, though: People still have problems that need solutions. Even with the advances we just described, millions still can’t get the essential services and resources they need to survive. The portion of humanity who can take advantage of technology has solved their basic needs so well that they have time and resources to worry about other, more abstract problems.

Designing apps to solve real-world problems

Many software developers forget a simple but critical principle: Apps are tools. They are a means to an end, even if the end itself is too abstract or frivolous. And most of the time, the software itself isn't that end.

Software developers love to build applications, and that love makes us forget that most of the time, we build applications for someone other than ourselves.

The startup accelerator Y Combinator offered a series of lectures at Stanford University in 2014 titled "*How to Start a Startup*". In those lectures, a question arose: How to choose the topic or idea on which you could base your company. The proposed solution: Find a problem your users may have and find a solution; it doesn't matter how big or small the problem is, as long as it's a real-world problem.

Many startups fail because they focus more on the technology they are using than on creating something that helps people. They spend too much money and time getting the hottest tech stack, the most complex frameworks. For these failed companies, all these layers of complexity hide a single truth: There's no real problem to fix, no user that needs their products.

Other companies might have a real problem to solve but still miss the mark. They will again try to build something complex, too smart for their own good. They could have built a more straightforward solution with less "*cool*" technology for considerably less money and still give solutions to their users.

Your users don't care if you used Java or Python to build the app they need. The only people who care about those things are other developers, who are not your primary target most of the time. Without a real problem to solve, apps are nothing more than fun mental exercises for the developers who create them.

And we are not saying that you can't build code just because you enjoy it. But when you're getting paid by someone else to do it, creating an app that works correctly is a priority.

Tip Search term: creative coding.

There is a type of computer programming that is not based on solving problems: Creative coding. This non-functional coding is focused on creating abstract representations like artistic work, visuals, or audio.

Companies like Google have created events for people to build code as a way of artistic expression, and it's a field we, as software developers, are only starting to explore. Many front-end developers use CSS to build visual works in the browser that push the limits of web technology.

Just like you would paint a picture or write a song to have fun and express yourself, consider creating code for more than starting your own business or making money.

Caring for user problems as a back-end developer

Backend developers work at a layer that is abstracted away from the users: Non-technical users will struggle to see the direct connection between your work and the app they use and love.

This abstraction is normal, as it's common for back-end developers to work in less user-centered tasks: Dealing with servers, APIs, and connecting to databases. Frontend developers, on the other hand, tend to work with more user-centered requirements because their main goal is building the interface with which the users will work directly.

This disconnection challenges back-end developers: It's easy to lose sight of the problems we are trying to fix with our app. You can spend months or even years in your career not knowing how or if the application you're working on addresses the right solutions your clients need.

This is why we put so much emphasis on problem-solving at the beginning of this book because it has to be our guiding principle as software developers. You will be a considerably more successful back-end engineer if you don't forget about your users, even if they don't directly use the code you build.

Finding a problem to solve

The chances are that in some moment of your career as a back-end developer, you will have to work with your team to find a new project to work on. It could be finding a mission for your new startup or starting a new project within your existing company. Focusing on problem-solving will lead to a more optimal assignment of your team resources.

Most of the problems we find revolve around two possibilities: Problems that don’t have a solution yet, and problems that do have a solution, but it is not good enough or only works for a small group of people.

Trying to find problems without a known solution it’s complicated: it’s human nature to try to fix things that don’t work, so either problems get solved relatively soon, or problems remain unsolved because solutions are complex or nonexistent. If you’re lucky to find a solution for a problem that hasn’t been solved before, go ahead and work on it; but feel free to work on problems that can be solved better.

Note: “Smaller” problems tend to be often ignored by developers. We focus on finding big problems because we think it will increase our chances of success; the more people will use the solution we build for them, the more recognition and revenue we get. However, thinking in terms of “big” or “small” problems is not the optimal approach.

In his talk, “Competition is for Losers” Peter Thiel talks about how Startups that focus on niche markets have a bigger chance to succeed just because they have less competition. Narrowing the scope of your project to solve just one simple problem (and solve it well) leads to having more time and resources to explore different solutions.

We should not assume that the existing solutions for known problems are always the best approach. Problems tend to change faster than solutions do, so implementations become outdated. Use this knowledge to keep your mind and eyes open to new, better solutions.

Diversity is a significant point to consider when finding problems to solve. A common problem with existing apps or solutions is that the solution they provide only works for a specific group of users.

For instance, look at video or picture editing apps. Applications like Adobe Premiere or Adobe Photoshop have existed for many years, but their target has always been experienced users who have very specialized knowledge of the field. Nowadays, thanks to smartphones and apps you can get for free, anyone can edit photography and video with professional quality.

Many software developers assume all their users are part of the same demographic group. Historically, software was always built for expert users first. Innovative companies like Apple recognized the importance of building products for non-experts; they owe their success to this democratization of technology.

There is one simple yet powerful fact: You are not your user.

You are the expert on each app you write: It will be evident to you how to use each feature because you wrote them and tested them daily. So, when a user comes to you and tells you they don't know how to use it, it can be a bit frustrating since it's something obvious to you.

It is a common misconception that the solution that works for you will work for everyone. Even when you are part of the group of users your app targets, your opinion about how the app should work is inherently biased. That's why it's essential to talk with your users directly.

The more you understand how diverse your user base is, the easier it will be for you to find their challenges and better tailor solutions for them.

Define a sample use case: The Pizza Place ordering system.

Early in any back-end developer's career, it's normal to feel overwhelmed by the lack of experience in real-world projects. At that point, most of the projects we've worked on are either:

- Projects for school or coding camp, where you are allowed to skip multiple steps from the software development process due time constraints.
- Existing projects to which you joined long after the main technical designs and decisions were made.

It's not until later in your career that you will have the opportunity to work on a project from its inception, finding a problem to solve and making decisions based on your user's needs.

Through this book, we will visit multiple techniques to build the back end for production-level apps, and we will connect them with a narrative that will let you get experience in areas previously unexplored for you. For that, let’s start with the problem to solve.

There’s a small pizza restaurant called “*The Pizza Place*” in your neighborhood. They only have a couple of employees today, but the manager is a brilliant lady with a long-term vision: Build great pizzas with quality ingredients at accessible prices. She’s worried that the pizza local kids are buying is not healthy enough for them, so she wants to offer pizzas cooked with clean, fresh ingredients.

The Pizza Place already offers dine-in but wants to start taking online orders. The manager doesn’t want to lose a hefty commission to an existing online food ordering service, so she and her husband decide the solution is to invest in building their custom ordering app. Yes, it will be expensive at first, but after running the numbers, they will save a lot of money in the long run.

You and two of your colleagues have just founded a new startup focused on helping local businesses to join the new digital era: You’re the lead back-end developer, and the other two are a front-end developer/designer and a salesperson.

The Pizza Place owner sees one of the ads you posted online, and she truly believes you can help her achieve her dream of bringing great pizza to everyone. She reaches out to you and your team to build their ordering system. You all schedule a call to go through the details.

Defining functional and non-functional requirements

It’s well known that one of the first steps in a new project is gathering user requirements. We often split requirements in two:

- **Functional or business requirements**: These are the primary goals of the application, the things your clients care about: the main problem and the abstract solution to it.
- **Non-functional requirements**: This is the definition of the system elements needed to implement the proposed solution to the functional requirements. This covers how many servers you will need, what kind of database, etc.

It is assumed that you have read multiple times about the differences between these requirements, so we will not go further in those details. What is important is that having a detailed list of requirements will help us choose the right pieces to build the best application we can.

Tip: Mental model: Defining requirements works very similarly to the Math concept of linear programming: You have an objective equation you need to optimize by finding variable values that optimize a target function, but there is an infinite combination of values you could use.

By considering a set of constraints (multiple equalities or inequalities), we can discard all the variable values that don't contribute to getting the best results for our objective equation.

In our case, the system requirements are the constraints that allow us to delimit the scope of the problem we're trying to fix. We can find the optimal combination of tools like databases or code libraries by looking at the area delimited by our requirements.

While math is a common skill for software engineers, don't worry if you are unfamiliar with linear programming, as we will not use it in this book. This is only an example to help build a more robust mental model for those who have previously seen this kind of problem.

We often talk about requirements but not how to define them effectively. Many developers expect clients to define their own functional requirements with misguided questions like "*what do you want the app to do?*".

Some clients have a good idea of what they want, but most won't. It's your job as a software developer to hold their hands for this part of the process to find the best solution for them.

An ineffective way of collecting requirements

A good starting point for understanding the process of collecting requirements is a real example. Let us discuss the experience of a developer (who may or may not be the author of this book) and the failures he had early on in his career. There is no better learning opportunity than making tons of mistakes.

A couple of years into this developer's career, he oversaw recollecting the system requirements for a prospective client. He contacted the client's company manager to schedule some meetings to define the requirements. He scheduled three sessions out of "*an abundance of caution*". To make sure he got all the requirements right.

During the meetings, he asked the manager what he wanted the app to do. The manager talked about how the business worked, and the developer and his team asked all the questions they could think of. They took many notes and drew many diagrams. By the end of the three-day business requirement gathering, they felt like a subject matter expert in the client's business.

Fast-forward a couple of weeks later; they start coding the application. As they built the user interface defined in their diagrams, they started having questions—a lot. Their questions ranged from not knowing what the application should do in specific corner cases to having completely misunderstood the contents of a dynamically generated report that the app was supposed to render.

How was this developer supposed to tell his manager they had to schedule more meetings with the client? He had set the expectation that those meetings were all they needed. This developer's manager had to talk with the client to request more meetings to answer their questions, which delayed our project.

The developer learned that he needed to set the right expectations: There will always be more questions.

The requirement definition cycle

The definition of system requirements is an iterative process, often interleaved with the solution design itself.

There is no magic recipe to finding all the requirements for a new project. However, here are a few heuristics you can use to guide your path:

1. Schedule one or two meetings with the following agenda:
    - Find out what the problem your client is trying to solve.
    - Get a detailed picture of the business: Short, mid, and long-term goals and budget constraints.

2. Request the help of one of your client's domain experts.
3. Schedule multiple work sessions with the domain expert with the following goals and actions:
    - Learn how the existing process works today.
    - Understand how your client is currently working around the problem.
    - Build prototypes and revise requirements.
4. With the help of the subject matter expert, iterate multiple times between prototype design and requirements definition and clarification.

Find out what is the problem your client is trying to solve

As we've discussed in this chapter many times, the first thing you need to know is what problem you're trying to fix. If this is a project you defined yourself, you might already know the problem, and this step is mostly done.

However, new clients will have their own business cases, which is why they request your services. Common use cases include:

- Building an application to automatize an existing process. The goal here is to reduce costs by replacing existing manual processes that are error-prone or time-consuming.
- Build a new application to add value to existing business processes. The goal is to create new features whose goal, in turn, is to solve our client's problems; things as offering a new service or product online.
- Refactor an existing application. The goal is to replace (entirely or just parts) a legacy application that is not operating as expected or it's too costly to maintain.

Your prospective client might not have a clear idea of what solution they need (that is why they will pay you), but they will know the challenges they are having. Understanding what led them to you in the first place is the first step.

Get a detailed picture of the business

Once you know what ails your new client, the next step is to understand who they are and their goals in the short, mid and long term.

Why do we need to understand what their long-term goal is? Can't we just focus on the application they want to build today? You can. It is possible to limit your knowledge about your client to this specific problem and still having a successful project.

But, by gaining deeper insight into your client's business, you get two things:

- A better vision of how the application could grow and future use cases.
- The possibility of finding other areas where you can help your clients means more business for you.

The former is especially important from a technical point of view: It lets you better choose the tools that will not become roadblocks when future requirements surface.

For instance, they might be using a specific billing service provider today, but they plan to move to a different provider a few years later. This knowledge will inform you that you must be careful not to tightly couple the system implementation to the existing provider.

Tip: Search term: future-proofing

A term commonly used in software projects is "future-proofing", which means building features into your application that are not needed today but might be used in the future.

Future-proofing is generally considered a bad practice, as it introduces wasted effort and unnecessary complexity in your code base by implementing features that might never be used. Dead code is always tech debt.

However, building applications while understanding what the requirements may be in the future is not future-proofing. We build applications for today's use cases using tools that will not create tech debt for the possible use cases of tomorrow.

One thing about the business which is critical to understand is the limits they have on the budget. It’s not the same as building an app for a billion-user, multi-national company with millionaire budgets as building an app for a local business with only a couple hundred users.

Budget constraints will give you much information about non-functional requirements. If they have a limited budget, your client might be *OK* with doing some trade-off between the number of servers you need and the number of users they can have. Open-source tools may be preferred to licensed products.

Your client may not know how to create a budget for this project; we can guide them in creating one if we fully understand their goals, the functional and non-functional requirements, and how to map them into infrastructure and tools.

Request the help of one of your client’s domain-experts

Before you finish these first meetings with your new client, ask them to assign a domain expert who understands the problem to fix. It can be the person who is expected to use the new app or who works with the existing process.

As you will be working closely with this domain expert, set the right expectations: You need someone you can easily reach out to get answers to your questions. Someone who can effectively train you in how this part of the business works.

This part is essential, as you don’t want to make the same mistakes I did: Wasting too much time on managers or directives in the initial exploration phase. Or believing you have enough knowledge to be the domain expert (99% of the time, you will not be the expert in that domain).

Learn how the existing process works today

The most effective way to collect requirements is to see how clients operate today. With the help of your domain expert, start exploring the details of the business operations. If possible, request permission to shadow them as they do their daily work.

Domain experts are not necessarily specialized users. If you’re building an exercise app, a domain expert can be someone who exercises often or a user who would use it to get in shape.

As you watch your domain expert work, you will learn how the system works today by asking the following questions:

- How is a regular day for the person who is expected to use the app?
- In which context are they expected to use it? In an office? In the gym?
- What does the expert like about the current process? What do they dislike?

As you see the domain-expert work, notice if they do any “*hacks*” or “*tricks*” to make their job easier. These might be hidden requirements directly related to the main problem.

Remember a key fact: You cannot improve a process you know nothing about. You will not become an expert working on this project but relying on a domain expert will give you enough proficiency to find all the requirements.

Build prototypes and revise requirements

As mentioned earlier, the requirements gathering is iterative. Once you know how the new application should work, it’s time to work on the first prototypes.

Tip: Search term: low-fidelity prototypes and high-fidelity prototypes

Building prototypes is part of an exploratory phase in building applications. It’s a task that usually lays in the hands of the team’s designer or front-end engineers, but anyone can build effective prototypes.

Low-fidelity prototypes are roughly built sketches that only highlight the critical functionality. They don’t include details like color, fonts, images, or even text, as these distract from the point you want to discuss.

Low-fidelity prototypes are good tools to use at the beginning of a project to explore ideas and find misunderstandings in requirements.

Low-fidelity prototypes are cheap, as they can be drawn in whiteboards, sticky notes, or even napkins you may have lying around while having coffee.

High-fidelity prototypes are complex and range from high-definition sketches to fully working applications. They are better for later in the project to validate assumptions about user interaction, visual design, and text content.

High-fidelity prototypes can be expensive, as they require time from a designer or a software developer to create. Use them sparsely.

By creating low-fidelity prototypes as you gather requirements, you can confirm if your assumptions about the application are correct. Finding gaps in your mental model early on will save you a lot of effort and money in the long run.

In this step, you want to involve your front-end developers and your designers, as they can collaborate in creating the prototype and get answers to questions they might have on their own.

Don't spend too much time or effort creating these prototypes, as they most likely will change along the way. You only want to include the essential things you need to drive the requirements conversation further.

Figure 1.1 shows an example of a low-fidelity prototype. The low-quality shifts focus to the use case where a user selects an item from the menu and clicks the `Order` button:

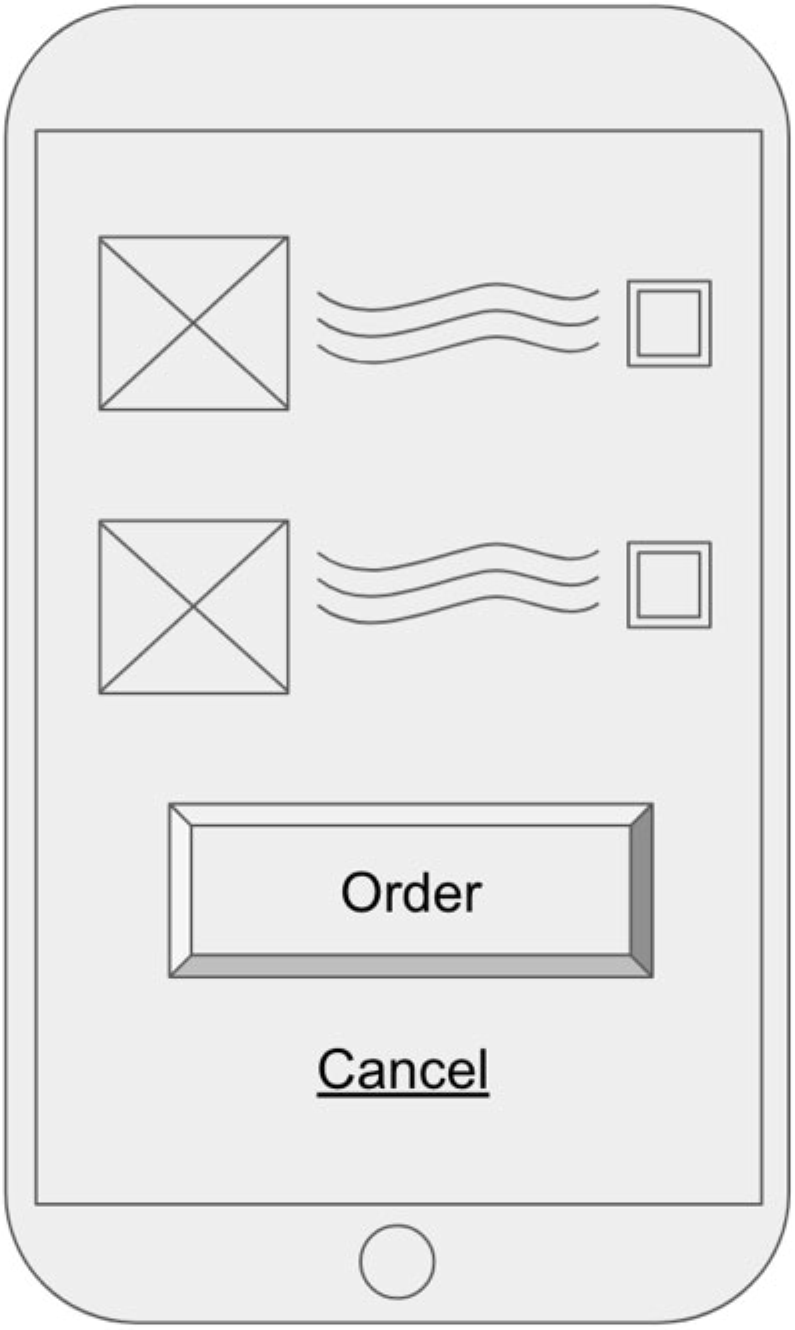


Figure 1.1: *An example of a low fidelity prototype.*

Review your prototypes with the domain expert. They will either validate your assumptions or tell you what you got wrong. The advantage of low-fidelity prototypes is they can be updated relatively quickly. Using this feedback, iterate to see how the domain expert operates and find the correct assumption.

As you get the agreement in the defined requirements, create a central document to include all the project facts.

Use case: Defining requirements for the Pizza Place

After spending all day at The Pizza Place, we find they want a straightforward operation for the first application version: Display a menu to their users on their computers or smartphones and allow them to order food.

They are just starting with this modernization effort, so they want the cheapest option available to service their current client load of an average of 100 orders per day. Before you leave, the manager tells you that if the system works fine, they will want to expand it as they get more daily orders.

This information about their goals and budget gives us a clear picture: The Pizza Place don't need a distributed system with clusters of servers *today*, but they are gaining clients quickly. It will not be long before the app needs to scale, and we will account for that.

After working with the front-end developer and the cooks, you present a collection of prototypes and requirements to the business owner.

She finds that some of your suggestions have features they don't need in the short-term (like allowing users to do too many customizations on their pizza's toppings). The Pizza Place only offers a simple menu of pizzas with pre-selected high-quality toppings, so a customization module is extra work they don't need.

Thanks to working closely with the Pizza Place team for a few days, we can define a document with all the requirements and assumptions. We use that document as the *source of truth* and get sign-off from the client.

Having understood our new client's needs, it's time we see a menu of all the different tools we have to satisfy these requirements.

The modern system design: a ten thousand-feet view

Since we now know that finding solutions to our users' problems takes priority over choosing a tech stack, let's look at the variety of tools currently available for software developers at each layer of the application development process.

The following is a high-level view of the most common elements used in building modern applications:

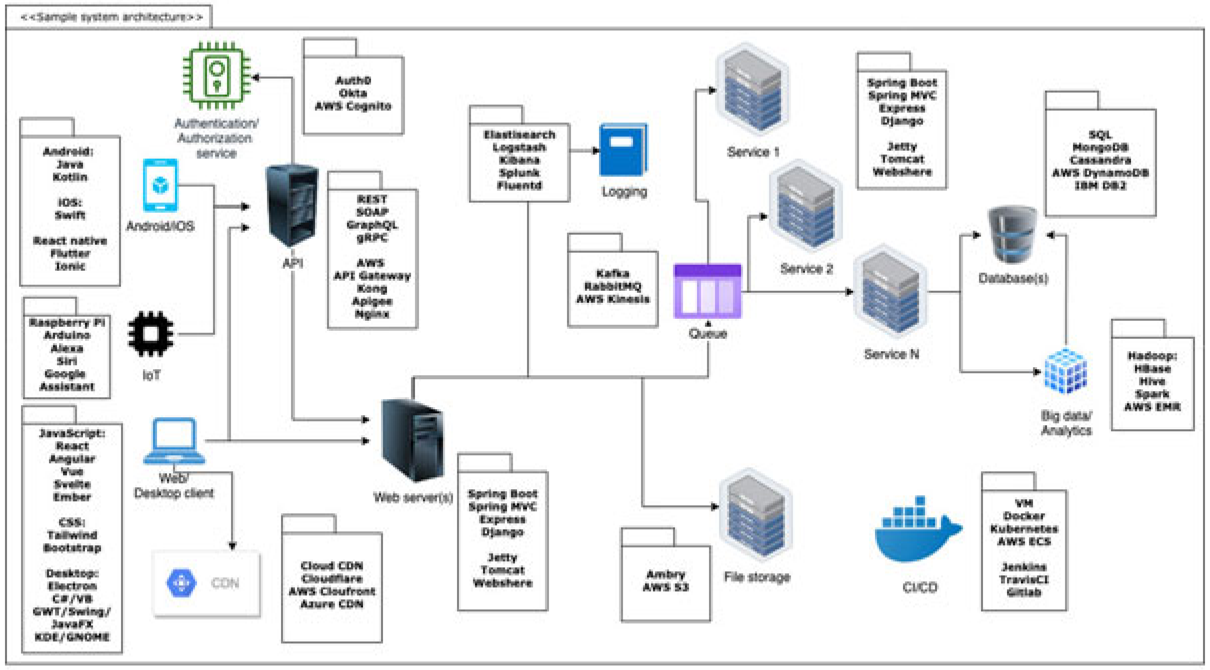

<<Sample system architecture>>
Authentication/ Authorization service
Auth0 Okta AWS Cognito
Android: Java Kotlin iOS: Swift React native Flutter Ionic
Android/iOS
API
REST SOAP GraphQL gRPC AWS API Gateway Kong Apigee Nginx
Elasticsearch Logstash Kibana Splunk Fluentd
Logging
Kafka RabbitMQ AWS Kinesis
Queue
Service 1
Service 2
Service N
Spring Boot Spring MVC Express Django Jetty Tomcat Webshere
Database(s)
SQL MongoDB Cassandra AWS DynamoDB IBM DB2
Big data/ Analytics
Hadoop: HBase Hive Spark AWS EMR
Raspberry Pi Arduino Alexa Siri Google Assistant
IoT
JavaScript: React Angular Vue Svelte Ember CSS: Tailwind Bootstrap Desktop: Electron C#/VB GWT/Swing/ JavaFX KDE/GNOME
Web/ Desktop client
CDN
Cloud CDN Cloudflare AWS Cloufront Azure CDN
Web server(s)
Spring Boot Spring MVC Express Django Jetty Tomcat Webshere
Ambry AWS S3
File storage
CI/CD
VM Docker Kubernetes AWS ECS Jenkins TravisCI Gitlab

Figure 1.2: *Sample system architecture*

Overwhelming, isn't it? We have multiple layers like a front or a back-end, each of which has numerous services like data storage, business logic, and infrastructure management; each has different providers and implementations, with their pros and cons.

Getting the front-end out of the way

Notice that we are not making a hard distinction in the picture between a front-end and the back-end. This soft limit has two reasons:

First, this is a general view of the whole system, end-to-end.

Second, in modern development, the division line between the responsibilities of front-end and back-end developers has become blurry, especially in startups where both budgets and teams are small and where most software engineers end up doing work all over the stack.

It's common for front-end developers to work in the API or for back-end developers to update client code to enable the features they need.

This merge in responsibilities has become so ubiquitous that it led to the emergence of the elusive title of "*full-stack developer*", which is nothing else than a software engineer with experience in technology from both ends of the stack.

Let's focus on the front-end for a second. We can see that this sample architecture supports a diverse set of clients:

- Web clients (browsers like Chrome, Firefox, etc.)
- Native mobile applications (Android, iOS)
- **Internet of Things** (**IoT**) devices with built-in technology like Raspberry Pi or Arduino boards.
- "*Smart*" devices like Amazon Alexa or Google Assistant
- Native desktop clients

Historically, before the general use of smart devices like phones and tablets, traditional web applications had a strong coupling with back-end services. Dynamic web applications would rely on the same back-end server to do multiple things:

- Fetch data from the database
- Execute business logic
- Transform and format data into HTML
- Return the dynamically generated HTML to the browser

However, the need to reuse business logic became critical as a wide range of modern gained popularity. The execution flow of web applications was split in two:

- An API to publish business-specific logic (which also includes fetching data and data transformation)
- A Web server to render the front-end, which calls the business API, if needed

- Mobile apps that make calls to the same API as the web application

This separation of concerns helped decouple the front and back-end development: As long as a contract is fully established at the API level, the back-end and the front-end can be developed in parallel. We'll go more in-depth on these design concerns in the chapter about (*Chapter 4, Designing APIs*).

With this clean separation between the front and back end, let's take out the front end of the picture, and simplify it a little, so we can focus on all the areas we will discuss in this book.

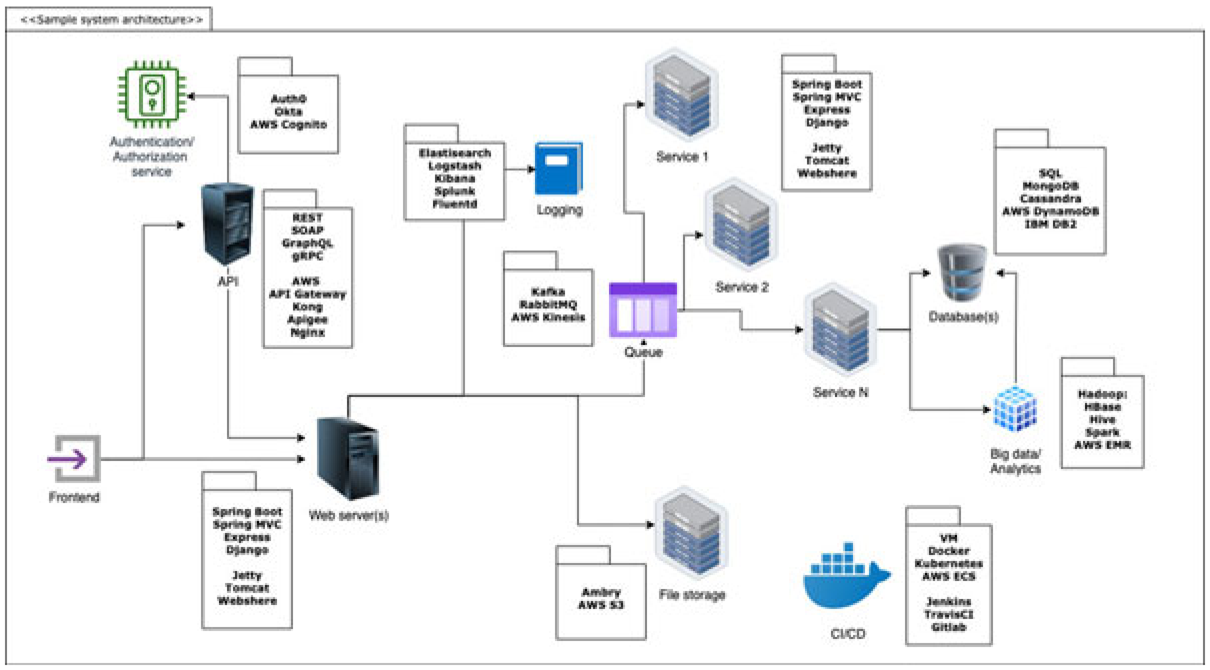

<<Sample system architecture>>
Authentication/ Authorization service
Auth0 Okta AWS Cognito
API
REST SOAP GraphQL gRPC
AWS API Gateway Kong Apigee Nginx
Elastisearch Logstash Kibana Splunk Fluentd
Logging
Kafka RabbitMQ AWS Kinesis
Queue
Service 1
Spring Boot Spring MVC Express Django
Jetty Tomcat Webshere
Service 2
Service N
SQL MongoDB Cassandra AWS DynamoDB IBM DB2
Database(s)
Big data/ Analytics
Hadoop: HBase Hive Spark AWS EMR
Frontend
Web server(s)
Spring Boot Spring MVC Express Django
Jetty Tomcat Webshere
Ambry AWS S3
File storage
CI/CD
VM Docker Kubernetes AWS ECS
Jenkins TravisCI Gitlab

Figure 1.3: *Sample system architecture*

Don't worry if you feel a bit lost. You are not expected to know each part of this diagram, nor should you memorize it. This is just a preview of all the topics we will discuss in this book.

Each component displayed in *Figure 1.3* is a different kind of tool in our belt. They all accomplish different goals, but only a subset of them are required for the average app. For instance, you might not need a file storage service if you're not dealing with binary files in your application.

Building blocks

Web servers are in charge of receiving HTTP requests from either the clients directly or from an external API layer. It could be just one server if there is not much traffic, or it can be a set of multiple servers behind a *Load Balancer*. We will see more details in *Chapter 2, The Client-Server Architecture*.

The **API layer** receives requests from clients or other applications. This can be a logical (services running on the web server) or semi-physical (independent services running in their own servers) layer which takes care of tasks like authorizing client requests using an **authorization service**. The API layer can even limit requests or charge users for using it. We will see more details in *Chapter 4, Designing APIs*.

For distributed systems, it's common to use a **queue**. It allows asynchronous communication between services. Queues provide resiliency to the system in case individual services are down. We will get more information about it in *Chapter 11, Creating High-Performance Apps*.

Logging services provide storage of operation information. Errors, warnings, or event metrics are stored here for later analysis. Logging is one topic that will discuss in *Chapter 8, Handling Errors*.

The application can be split into one or many business services. These can also be logical or semi-physical. The whole second part of this book is dedicated to how to build these services.

Database services provide a place to persist our application's data. Services like **file storage** are specialized types of data storage. *Chapter 5, End-to-end data management*, goes in-depth about how we store and manage data.

Continuous Integration and Continuous Delivery (**CI/CD**) is a methodology focused on building the infrastructure required to compile and deploy our application. In *Chapter 10, Deploying Applications*, we discuss how to leverage concepts like version control, **Virtual Machines** (**VMs**), or containers to deploy our application as we make changes to it.

There are many other building blocks we do not include here. This omission is not due to these components being less critical but because it would blow up the scope of this book.

Also, notice that we're not referring to multiple of these blocks by specific technologies (e.g., we're using "*data storage*" instead of MySQL, MongoDB, etc.). By keeping these concepts at the right level of abstraction, we will see common patterns those specific implementations share, allowing us to master them all with relative ease.

Understanding the listed components will give you enough knowledge to be a proficient back-end developer.

In the following chapters, we will dissect each component in these diagrams along with good practices to use them to create quality applications.

Conclusion

Most of the apps we will build in our careers are tools to fix problems for people. If the app doesn't provide the right solution, users will abandon it.

Having this problem-solving-led vision as a back-end developer will give you an advantage over other developers: It will guide all your technical decisions, making you a more efficient and successful developer; more clients will want to use the code you build, and more companies will want you to be part of their teams because your code gets things done.

Finding problems to solve is not a simple task, but if you are proactive enough and ask the right questions, you will find things to work on that make a difference. Providing the right solutions to simple problems is the path to success for the biggest technology companies.

By following an iterative process, guided by your client's domain expert's feedback, you will be able to identify all the system requirements needed to choose the right components for your application.

Modern applications have an ecosystem of components and tools constantly evolving. More than learning the specifications of each tool, we should focus on learning design patterns that abstract the benefits of the whole toolset.

In the following chapter, we will dive deeply into the most ubiquitous type of application we will build as back-end developers: Web applications.

Questions

- From all the apps installed on your phone, pick one that makes your life easier AND only solves one or two problems. Why does a solution so specific cause an impact on you?

- Have you ever downloaded an application that you immediately uninstalled because it didn’t provide any value for you? If so, what was it missing?
- Think of three companies or startups you think are successful. What problems did they solve when they first started? Did they focus on one area, or did they solve more than one problem?

CHAPTER 2
The Client-Server Architecture

In order to become a proficient backend developer, we need to clearly delimit what our responsibilities are. These responsibilities are strongly defined by the architecture patterns we use while building our application.

The most prominent architecture pattern we use in backend development is the client-server architecture. There are virtually no applications currently being built which don't rely on this pattern, at least in part.

This is a lesson so basic that many books assume you know everything about it. It's common that this assumption can easily lead to misunderstandings on how web applications work; the most common one is to think that HTTP and client-server is a 1:1 mapping. We'll find that client-server is much wider than HTTP.

In this chapter, we will discuss how the client-server architecture is implemented in modern web-based applications, at more than one level of abstraction.

This is not a detailed description of a specific protocol (like HTTP), even if we include details about it. We'll only include enough information for you to have a strong understanding on how things work.

Structure

In this chapter, we will learn the following topics:

- Architecture details
    - **Abstraction layer:** Frontend client and backend client
    - **Abstraction layer:** Data access service client, database server
- **HTTP:** The language of the web
- Implementing a web server
    - The main process

    - Serving a response
    - Multi-user support with multi-threading
- Using a production-ready server
    - Enabling HTTPS in Express
- **Layered architecture:** Fully splitting the client from the server
    - Splitting clients
- Client versus server computing
- Web servers as stateless services
    - Storing session data
- **Use Case:** Applying a client-server architecture to the Pizza Place app
- Client server versus peer-to-peer

Objectives

By the end of this chapter, you will have a strong understanding of how client-server technologies work under the hood and how we can leverage that knowledge to become a better backend developer.

You will also learn about some of the critical configurations in a web server: Response compression and communication encryption through HTTPS.

Architecture details

Having enough computing and storage power to accomplish tasks which today we consider simple was prohibitively expensive a couple of decades ago. Mostly large institutions like universities or governments owned operational computers with high-computing resources (or at least enough resources to provide value to their users).

Historically, the first examples of client-server architectures are based on the need of accessing high-capacity servers remotely. '*Dumb*' terminals were created to allow the operation of these computers without having to be physically present in the same room as them. These were the first clients, and they communicated with their servers through direct connections or private networks.

As technology advanced, clients became less '*dumb*'. Now, modern clients have more computing power and storage capacity than most supercomputers from the past few decades. However, the client-server architecture is going nowhere.

Simply defined, *servers* are high-storage-capacity and high-computing-power equipment that store and process most of the data related to specific functional goals.

A *client* is a smaller device -with less computing resources- which is used to access the server resources. One server can receive requests from N clients, as shown in the following figure:

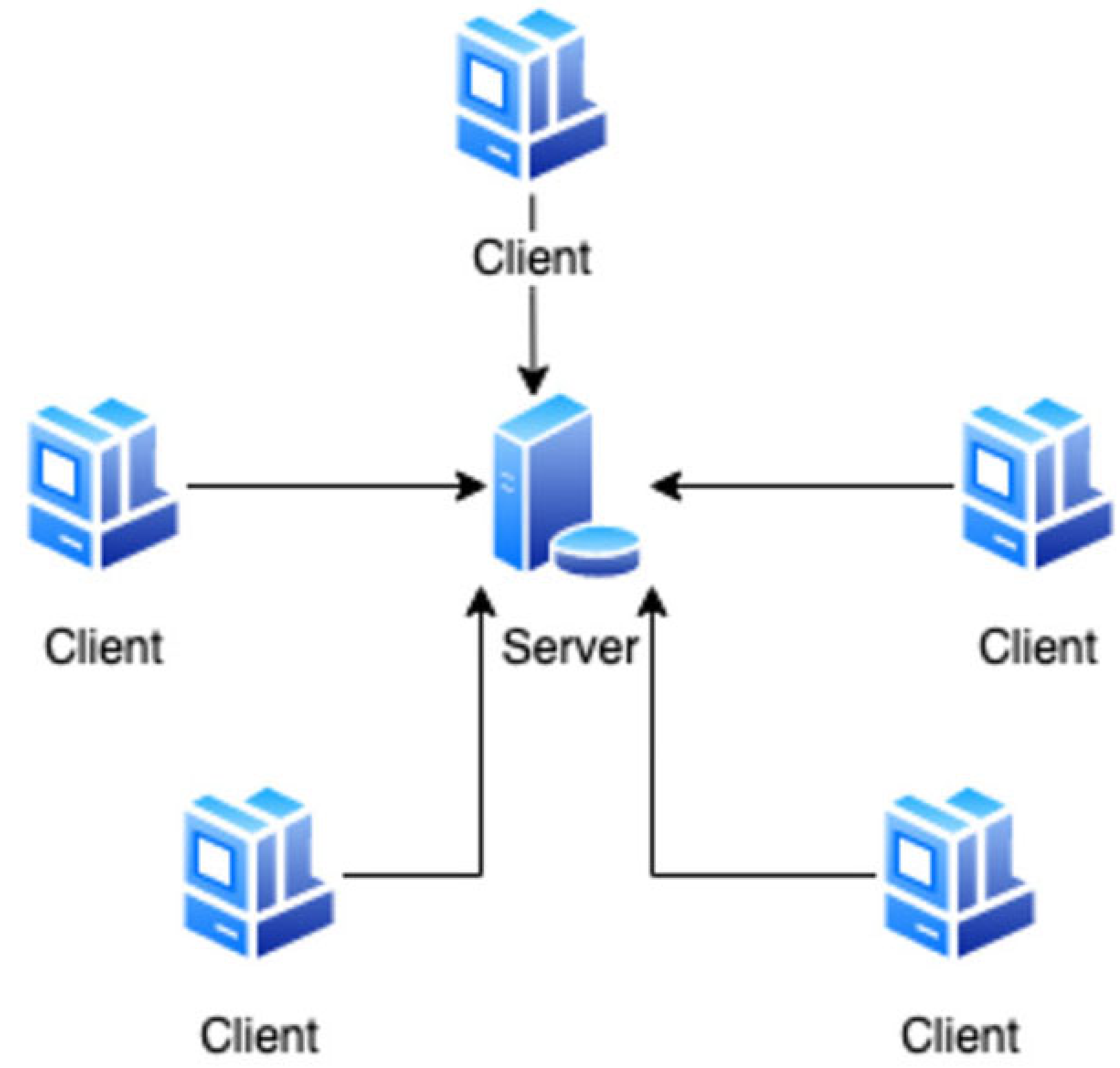


Figure 2.1: *Simple client-server architecture. Multiple clients connect to a single server*

As seen in the previous chapter, we can create clusters with multiple servers, each with different functions. Even if we have multiple servers, the definition of 'client-server' still holds. You can think of the whole interaction between clients and servers as a single, abstract, large scale, computing device. All the parts get together to become a single "*server*" unit.

A common misunderstanding is to believe that the client-server only applies to the web interface, when in reality the client-server is used as a pattern in multiple areas of application development.

The following are a couple of examples of areas which actively implement the client-server architecture:

- FTTP (for file storage)
    - **Client:** A program that allows users to upload, download or modify files hosted in a remote server.
    - **Server:** The server that stores the files.
- Data access service client, database server
    - **Client:** Data processing service, with database adapters like **Open Database Connectivity** (**ODBC**) or **Java Database Connectivity** (**JDBC**)
    - **Server:** Database server

Multiple instances of the client-server architecture can be part of a single, larger application.

Abstraction layer: Frontend client and backend client

The most common instance of the client-server architecture is the web application. The *client* role is assigned to the user interfaces (web apps, mobile applications, IoT devices, and so on) and the *server* role is given to all the services to which the user interfaces make requests to, all grouped in a single layer of abstraction: The backend.

It's important to notice that the role of the client can be given to user interfaces, but not all clients need to involve direct user interaction. There are clients which in turn are used by other servers (like database clients) to make requests to other servers. Thinking about the user interface and client as the same terms is a very common mistake.

Abstraction layer: Data access service client, database server

Relational Database Management System (**RDBMS**) implementations like SQL are based in a client-server architecture: A centralized data storage server which is accessed by lightweight clients.

This model of data storage concentrates all the information in a central service that has the role of a server. Then, multiple clients can send 'read' and 'write' requests to our database server.

Besides the obvious performance gains provided by having a server (or cluster of servers) dedicated entirely to store data, we also get some consistency guarantees if we assume that the database is the source of truth for the application.

Things get a bit complex when some databases implement architectures distinct from the client server. For instance, databases like Cassandra have an internal peer-to-peer node distribution in their clusters. It might seem like the client-server architecture does not apply to these kinds of databases; but from the point of view of external users and services, the cluster of peer nodes is still a server by itself.

The main takeaway here is that the concepts *client* and *server* are abstract: They can have multiple instances each and still be considered a single "*client*" or a single "*server*".

HTTP: The language of the web

We have discussed the client-server architecture from an abstract point of view. Now, let's discuss the most common example of this architecture: HTTP.

Hypertext Transfer Protocol (**HTTP**) enables all the communication between client applications (both desktop and mobile-based) and the web servers. Web Service technologies like REST and SOAP rely internally on HTTP.

In *Figure 2.2*, we can see the activity diagram for the communication between a client requesting a web page `index.html` and then doing a form submission through a POST request:

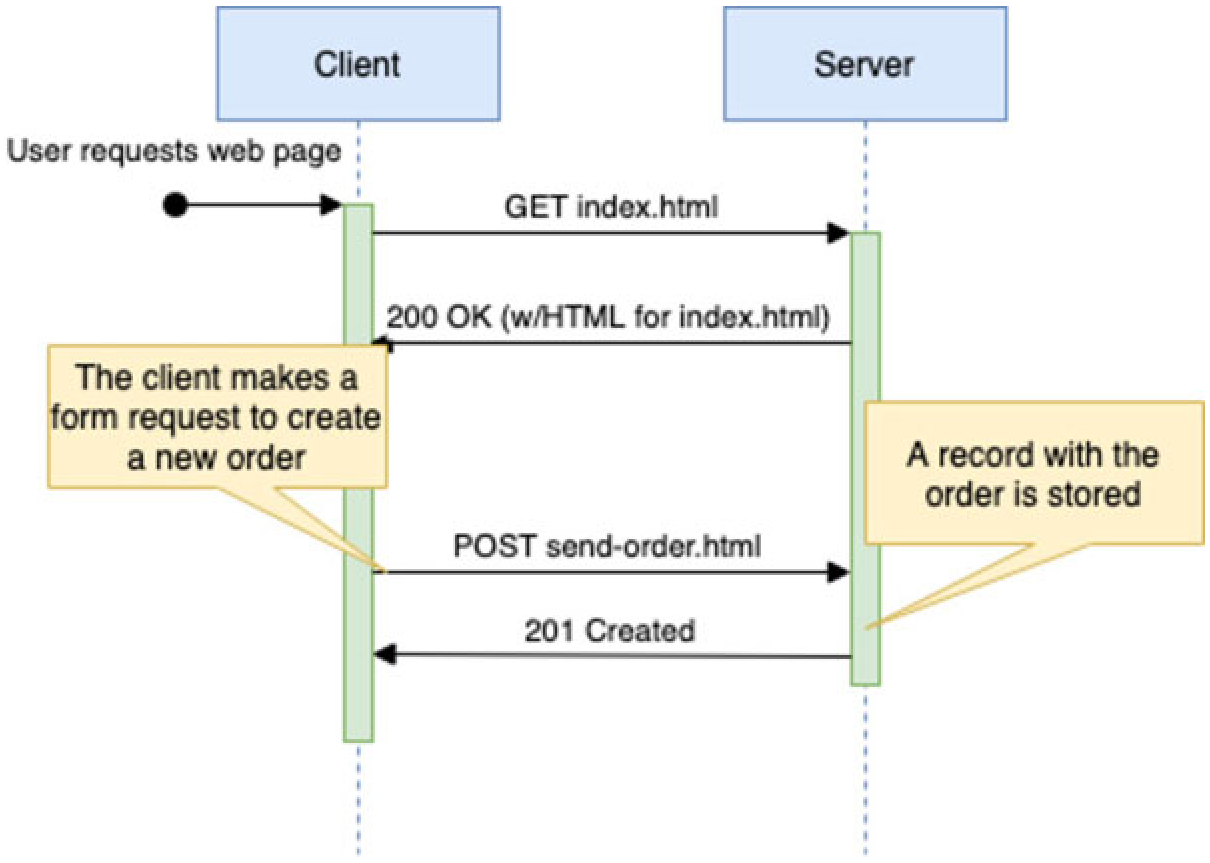

Client
Server
User requests web page
GET index.html
200 OK (w/HTML for index.html)
The client makes a form request to create a new order
A record with the order is stored
POST send-order.html
201 Created

Figure 2.2: *Example of an interaction between a web client and a web server using HTTP*

HTTP is so basic in web development that there are chances that you may already know all the basics. But, at the same time HTTP is composed of so many abstract concepts that it can be difficult to relate it to real-world cases if you don't have enough experience.

When I started my career in backend development, there was a big gap in my understanding of how web servers extended from more basic, Hello-World-like applications. The code we build in languages like C++ or Java while in school or early on in my career followed a *procedural* approach. The code you build for web applications, though, uses an *event-driven* programming paradigm.

How does a procedural-based application become an event-driven web application?

Tip Search terms: procedural programming and event-driven programming

These two are very different approaches of building code. In procedural programming, developers describe the behavior of the program one line at a time, one step at a time, end to end.

In event-driven programming, developers create the code to react to certain events: A user clicks on a button, a file is downloaded, a network response is received, etc.

The glue which is in charge of translating the code between both types of programming paradigms is hidden within the implementation of the web server. This glue looks like magic for inexperienced developers.

If you have experience in web development, you might fully understand how procedural applications work, or how event-driven applications work. But chances are you haven't seen the code that allows both to interoperate. Because of this, in this chapter, we will build a very simple web server.

Building a server has two goals: First is to look at the internals of HTTP as a protocol; which is more helpful than just reciting the HTTP spec. Second, it demystifies the inner workings of web servers; which are nothing else than regular applications which are also compiled and executed (and procedural in their execution). There is nothing really magical inside them.

Implementing a web server

Web servers are complex pieces of software. They deal with requests from multiple users, reading data from sockets and ports, formatting HTTP requests and responses, and so on.

Many developers spend years in their careers not fully understanding how web servers work internally. Chances are you will never have to build your own web server for a production-ready application; and you don't really want to.

Between building your own custom web server and using one that is commercially available, you should always choose the existing server. There are just too many things to consider to make sure a server is production-ready, so it's better to rely on products which have been widely tested before.

Having said that, we will implement a web server just to build a stronger mental model. As soon as you see that the basic operation of a web server is actually really simple, you will feel more confident of your knowledge about building backend code for web applications.

Note: About mental models

Exercises which might seem a bit unnecessary at first (like building your own web server when you will almost never want to use it in a production app) are useful for building a strong mental model.

A mental model is our understanding of how something works. Mental models do not only apply to software, as you can have a mental model of how a car works, how your coffee maker works, or how a bureaucratic process works.

When your mental model about a thing is extremely detailed, you can easily diagnose when something wrong happens within the thing itself. You can understand its strengths and weaknesses, and design improvements.

If you want to become an expert in any field, strive for building a strong mental model of the inner workings of what you want to study. You should be almost capable of imagining each part in detail such as a map or schematic.

We will first build a single-threaded version of the server. Then, with a couple of simple steps, we will extend the example to use a thread pool to allow multiple concurrent requests.

We'll use Java for this example, importing only base libraries. No frameworks or external products. If you're not familiar with Java, rest assured that there is not a lot of Java-specific code and almost every language has network libraries that will allow you to do this.

The main process

From a simplified, high-level view, a web server is an infinite loop which constantly reads a server's socket. For example, we expose the socket at localhost or `127.0.0.1`, and a specific TPC port which is not currently used by the server, in this case, port 8080.

Note: Anatomy of a server: TCP/IP, ports, and sockets

TCP/IP is a network protocol used to interchange information. It's used by HTTP to direct requests and responses to and from a web server.

Web servers can be identified by their IP address (and the domain name which resolves through DNS to that IP, e.g., google.com). Multiple services can be exposed in the same server, each using a specific port. The combination of an IP address and a port is known as a socket.

Ports are numbered and the first 1024 are assigned to well-known services. For instance, HTTP is commonly routed through port 80 and HTTPS through port 443.

Ports above 1024 are not pre-assigned and can be used by your application for exposing custom services. For instance, during development, a common port used for HTTP is 8080, but it can easily be exposed using any other port like 8081, 8082, etc.

Localhost

When a service starts running in a server, it's usually accessed by its public IP address and a specific port.

However, you can call a service from within the same server it's running on by using the "loopback" IP address instead of its public address. The loopback address in a server is always 127.0.0.1, and the DNS alias is 'localhost'.

So, if you're inside a production server running a web application like example.com, you can make a request to localhost:80 and you should see the exact same web page.

As long as the main process keeps looping, the server will be up. Let's take a look at the main process:

```
import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;
public class Server {
```

```java

final private static int PORT = 8080;

public static void main(String[] args) throws Exception {

  try (ServerSocket serverSocket = new ServerSocket(PORT)) {
   while (true) {
    try {
     Socket client = serverSocket.accept();
     handleClient(client);
    }
    catch(Exception err) {
     err.printStackTrace();
    }
   }
  }
}
}
```

In each iteration, the main process will check the socket to see whether a request has been made. Once the client makes a request, **`serverSocket.accept()`** will return an instance of a socket client which will contain the stream of data for a user request.

It's important to mention that **`serverSocket.accept()`** is a blocking operation. The execution will reach that line of code and it will pause until it receives an incoming request. Once the request is received and processed, a new loop will happen and (unless there is an error) the server will pause again at **`serverSocket.accept()`** in the next iteration.

Once a request is received, the **`handleClient`** function will be called:

```java
private static void handleClient(Socket client) throws IOException {
```

```

// Get the input stream
BufferedReader br = new BufferedReader(
  new InputStreamReader(client.getInputStream())
);

List<String> requestsLines = new ArrayList<>();

String line;

// Read all lines in the request until you find a blank line
do {
  line = br.readLine();
  requestsLines.add(line);
} while(!line.isBlank());


// …
}
```

The socket client has a **`getInputStream`** method, which returns the contents of the request. This is a stream of bytes, but since we know this is an HTTP request, we can interpret those bytes as text.

We can read the socket's input stream using a **`BufferedReader`**. The reader has methods that allow us to convert the input stream into an iterator which in turn reads the request one line at the time.

We iterate through the lines of text in the request until we find a blank line which marks the end of the request.

Note: BufferedReader has methods that return a Java Stream instead of an iterator. While this can be useful in certain cases, keep in mind

that the input stream will not close itself. It will stay connected until the client navigates away, closes the browser, or times out the request.

In those cases, the BufferedReader stream methods will hang until the client closes the connection, even if a blank line is returned in the response's stream.

The Java Stream API was introduced in Java 8, and if you're a Java developer and you want to learn more about it, you can check the 'Java Stream API' link in the documentation section for more details.

At this point, you can add a debugger breakpoint inside **`handleClient`**, run the program, open a browser window, and navigate to **`http://localhost:8080/index.html`**.

Once the execution reaches the break-point, you can see the HTTP request in the list of strings we gathered from the input stream:

```
GET /index.html HTTP/1.1
Host: localhost:8080
Connection: keep-alive
Cache-Control: max-age=0
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/92.0.4515.131
Safari/537.36
```

You might see some slightly different values or extra headers not included in this example, which define things like caching, authentication, session management, etc. These might be dependent of your local environment, like the type of browser you used to make the request.

The first line in the request follows a specific format where we indicate the verb or action to perform, the request target or resource to apply that action to, and the HTTP version.

There are 9 different types of verbs which can be used; the most common ones being GET and POST. Different combinations of a verb with the same resource can indicate distinct types of operations.

A POST or PUT request would look like the GET request we just saw, with the addition that the request can include a body below all the headers. This body can be binary data, XML, JSON, etc.

We will visit these verbs in more detail and see how to use them in *Chapter 3, Designing APIs*.

Serving a response

Having received and parsed a request from the client, it’s time to do something with it. Keep in mind that, from the moment the server receives the client’s request, the client will wait for a response; so we better act as fast as we can.

We will support the following two different operations:

- **Return a static HTML file**: This emulates the behavior of servers like Apache or nginx that return static files to the client when requested. Other static files to serve could be JavaScript, CSS, image, or video files.
- **Return a dynamically-generated HTML response**: This is a common way in which web application technologies like Java Servlets or PHP work. They build the response in run-time based on the information given in the request.

In order to support these two cases in our example, we need a condition to indicate the server when to execute each. Just for the sake of this example if the client requests `/dynamic.html`, we will return a dynamically-build response. For any other resources, we will try to fetch the requested static file (e.g. `index.html`). If the file doesn’t exist, we return a `404 Not Found` error:

```
final private static String DATE_FORMAT_NOW = “yyyy-MM-dd HH:mm:ss”;
final private static String DYNAMIC = “/dynamic.html”;
final private static byte[] NOT_FOUND_HTML = “<h1>Not found :(</h1>”.getBytes();


// …continuation of handleClient
    // Parse the requested path from first line in the request
String[] requestLine = requestsLines.get(0).split(" ");
String path = requestLine[1];


Path filePath = Paths.get(".", path);

// if the path requested is dynamic.html, print today's day
if(DYNAMIC.equals(path)) {
  sendResponse(client, "200 OK",
   "text/html",
   getDynamicResponse()
  );
  }

// else, print static file
else if (Files.exists(filePath)) {
  sendResponse(client, "200 OK",
   Files.probeContentType(filePath),
   Files.readAllBytes(filePath)
  );

// else, print error message
} else {
  sendResponse(client, "404 Not Found",
   "text/html",
   NOT_FOUND_HTML
  );
}
}

```

```
private static byte[] getDynamicResponse() {
Calendar cal = Calendar.getInstance();
SimpleDateFormat sdf = new SimpleDateFormat(DATE_FORMAT_NOW);
String response = String.format(
"<h1>Dynamic response</h1> Today is %s", sdf.format(cal.getTime()));

return response.getBytes();
}

private static void sendResponse(Socket client, String status, String contentType, byte[] content) throws IOException {

String LINE_BREAK = "\r\n";
OutputStream output = client.getOutputStream();
output.write(("HTTP/1.1 " + status).getBytes());
output.write(("ContentType: " + contentType + LINE_BREAK).getBytes());
output.write(LINE_BREAK.getBytes());
output.write(content);
output.write((LINE_BREAK + LINE_BREAK).getBytes());
output.flush();
client.close();
}
```

Inside **`handleClient`**, we check the resource to fetch. We rely on the VERB-RESOURCE-HTTP_VERSION structure we mentioned earlier to extract the resource name. If the requested resource is "**`/dynamic.html`**", we will dynamically generate HTML to display the current time and date:

```
String response = String.format(
"<h1>Dynamic      response</h1>      Today      is      %s",
sdf.format(cal.getTime()));
```

This might be a very simple example of dynamically generated HTML, but in practice, you will probably fetch data from a database, convert it to a format which can be understood by your users, and then convert it into HTML.

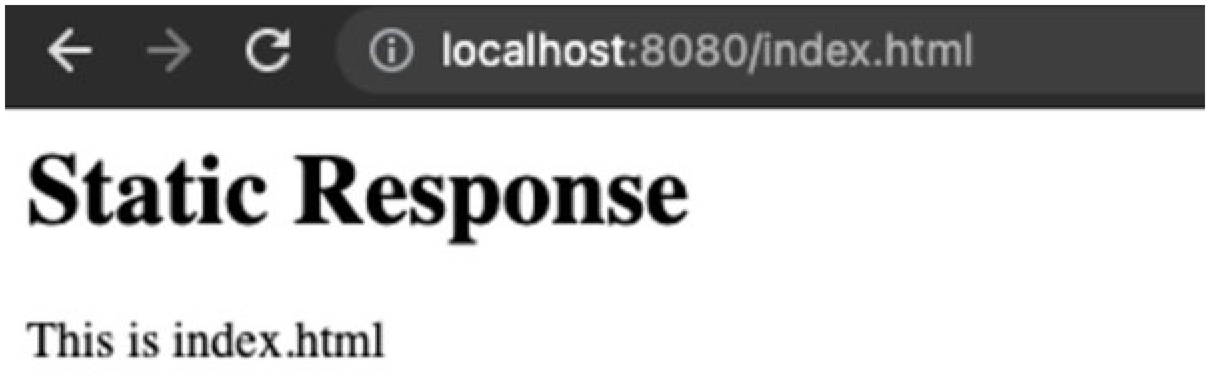
For the static file route, we will create a file with the name **`index.html`**, and put it in the same folder where our server is going to run:

```
<body>
   <h1>Static Response</h1>
   This is index.html
</body>
```

The **`sendResponse`** function takes care of formatting the response headers and rendering the content into the socket's output stream. The stream reads arrays of bytes, so we convert the text we want to return in our response to bytes (using the conveniently-included method **`getBytes`**, part of the **`String`** class).

Having created the two resources, we can access them through a browser.

Open a new browser window and request **`http://localhost:8080/index.html`** to fetch the static response. The browser should display a text message as displayed in *Figure 2.3*:

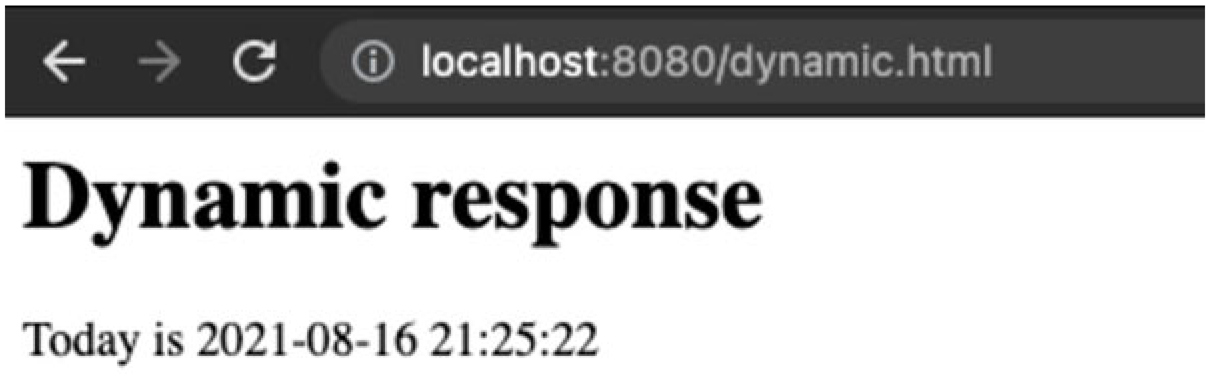


Static Response

This is index.html

Figure 2.3: *localhost:8080/index.html*

In a similar way, we can access the resource that dynamically generates HTML by opening another browser window and navigating to **`http://localhost:8080/dynamic.html`**. You will see the same text as shown in *Figure 2.4*:



Dynamic response

Today is 2021-08-16 21:25:22

Figure 2.4: *localhost:8080/dynamic.html*

You can print the response directly in the server, or you can use your browser's developer tools to see it:

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Connection: Keep-Alive
Content-Encoding: gzip
Content-type: text/html; charset=utf-8
…
<html>
…
```

The format for the HTTP response is as follows: The HTTP version, then the status code, and then a status text. The actual content of the response (in this case, the HTML for **`index.html`** or **`dynamic.html`**) is returned below the response headers.

Notice how the stream of bytes the server returns can be parsed back to a string. There is no magic here: We converted text to bytes in the server, and we can parse them back to text again.

You can keep using your browser to test our server endpoints, but many developers like to use `curl`. You can request both the resources by running the following commands in `terminal/bash`:

```
curl localhost:8080/index.html
curl localhost:8080/dynamic.html
```

Each of these commands makes an HTTP request (using GET by default) to the given URLs.

Note: About curl

The Bash command 'curl' is an open-source command line tool for transferring data with URLs. Developers use it commonly to make requests to remote servers.

If you're using a Unix-based computer like Linux or MacOS, `curl` is already installed in your system. If you're running Windows, there are multiple sources from where you can install curl:

- Git bash, which is included in Git for Windows

- Package managers like Chocolatey, MSYS2, Scoop, Cygwin

Multi-user support with multi-threading

Since our server is running using a single thread, if one of those requests were to take a long time to fulfill (for example, doing a long database query), the next request will have to wait in line for the first to complete before continuing.

Looking at our code, this means that the execution inside our main function will pause at two points: On `serverSocket.accept()` while it waits for a new request to be made, and on `handleClient` while it waits for that function to complete.

Visualizing asynchronous execution can be difficult, but we can directly see how this problem works by adding an artificial delay to our code of 10 seconds and printing the current time in seconds right after the delay finishes:

```
private static void handleClient(Socket client) throws
Exception {
```

```
Thread.sleep(10000);

System.out.println(" Current time in seconds: " +
  Instant.now().getEpochSecond());
//…
```

After adding that code to **handleClient**, call **http://localhost:8080/index.html** two times in a row (it can be in two different browser windows, or two parallel 'curl' calls). If you're using tow browser windows, try to make both requests as fast as you can.

Take a look at the text messages we're printing in the server's console. Notice how each request is exactly 10 seconds from each other.

```
curl localhost:8080/index.html & curl
localhost:8080/index.html
Current time in seconds: 1629241268
Current time in seconds: 1629241278
```

Requests will stand in a queue waiting for the previous one to finish, regardless of how fast one is executed after the other.

As you can probably tell, this is obviously a bad situation. We want users to be able to make requests to our server concurrently. Imagine if this is how sites with millions of users like Facebook or Google operated, where each server can only serve one user at the time.

For enabling support for multiple concurrent users, we can extend our server to use more than one thread.

Adding multi-threading to our server is straightforward. To execute the code in a separate thread in Java, we need to create a new class which extends the **Runnable** interface and implement the **public void run()** method.

We will move all the code which currently sits in **handleClient** into **public void run()**. Since we cannot pass parameters to **run**, we will pass **Socket client** in the constructor for the new class, as follows:

```
class ServerHandler implements Runnable {

private Socket client;
```

```java

public ServerHandler(Socket client) {
  this.client = client;
}

public void run() {
  // the contents of handleClient
```

Then, on our infinite loop, every time we receive a new request, we call **ServerHandler** as a separate thread:

```java
public class HttpServerMultiThread {

static ThreadPoolExecutor executorService =
  (ThreadPoolExecutor) Executors.newCachedThreadPool();

final private static int PORT = 8080;

public static void main(String[] args) throws Exception {
  try (ServerSocket serverSocket = new ServerSocket(PORT))
{
   while (true) {
    try {
     Socket client = serverSocket.accept();
       // Call ServerHandler's run function in a separate
thread
     executorService.submit(new ServerHandler(client));
    }
    catch(Exception err) {
    err.printStackTrace();
    }
   }
```

```
    }
}
}
```

Creating new threads consume memory and resources, so we don't want to generate an infinite number of threads. We use a thread pool (**`ThreadPoolExecutor`**) so Java can reuse threads instead of always creating new ones for each request.

If we don't want to use a thread pool for this example, we can always create the threads manually as follows:

```
new Thread(new ServerHandler(client)).start()
```

This means that our **`main`** function now will only pause in **`serverSocket.accept();`** once the execution reaches the new **`ServerHandler(client) class`**, the main thread will not stop and wait for the request to be processed and the response to be served. The execution will immediately move back to wait for new requests in **`serverSocket.accept()`**.

Now, re-run your server and repeat the experiment where we called **`/index.html`** two times at simultaneously. Notice that now the difference is less than 10 seconds. In this example case, the difference is ~2 seconds, which is the time it took to manually fire the second request after the first, using a browser:

```
Current time in seconds: 1629242176
Current time in seconds: 1629242178
```

If you use **`curl`** to execute both the requests in parallel, you will notice that both the values will be equal, which means the difference between requests is less than a second, even with the 10 second-delay in the code.

All the code required to serve a client's request is being done in a separate thread than the one which is waiting for connections and running the server itself.

This new multi-threading server is a lot more similar to production-ready servers. It can handle requests from multiple users without blocking each other. Yet, we still wouldn't want to use it for a real application because then you would have to manually implement features like encryption (HTTPS/TLS/SSL) or response compression (gzip). While implementing these features yourself is possible, you will need to dedicate extra resources to make sure they follow the standards correctly and they are bug-free.

Tip Search term: gzip

Modern web servers use gzip by compressing the responses they return. As seen, HTTP responses can be long strings of text, with multiple white spaces or repeated words (think of HTML elements like <body>, you will have at least two strings with the value "body" for opening and closing tags). Taking advantage of this repeated information, we can apply encryption to remove repeated data and reduce the size of requests and responses.

Encryption services like gzip compress the responses so they can be transmitted faster to the client. The client then de-compresses the response and parses it.

Servers like Apache provide compression services out of the box. In the case of Apache, you can enable mod_gzip in the configuration file.

Using a production-ready server

Now, let's take a look at an actual server you can use to build production-ready applications. Instead of sticking to Java, let's take a look at the NodeJS Express server.

Note: Traditionally, Java web applications have been deployed by packaging them in WAR files, which then are deployed to independent web servers like Tomcat or Glassfish.

From the past few years, Java web applications started to be built with embedded servers like Jetty using frameworks like Spring Boot or Play. Many applications (especially Docker-based ones) strive for applications with embedded servers.

Both deployment technologies are valid; each with its own set of complexities.

Express is a popular framework used to build web applications in Node.js. Express works as an embedded server, which means that you don't need to start a separate server to execute your application, instead, it all runs in a single process -similarly to our custom-made example web server-.

Assuming you have installed Node.js in your computer, create a new Node project and install Express with the following commands:

```
npm init
npm install --save express
```

Then, create a new **`index.js`** file with the following JavaScript code:

```
const express = require("express");
const app = express();
const port = 8080;

app.use("/static", express.static("public"));

app.get("/", (req, res) => {
  res.send("<h1>Hello World!</h1>");
});

app.post("/", function (req, res) {
  res.send("Got a POST request");
});

app.listen(port, () => {
  console.log(`App listening at http://localhost:${port}`);
});
```

This small application is doing a few things:

- Creating an instance of an Express server, and keeping a reference to it in the variable **`app`**
- Creating a mapping for static files hosted in the folder**`/static`**
- Creating a mapping for both GET and POST requests for the route **`/`**

- Starting the server in the port 8080 and showing a success message once the server completes initialization

Notice that this piece of JavaScript code is very similar to the code for our custom made server in Java. However, notice some important differences:

- **Express follows a fully event-driven programming approach**: We define ‘callback’ functions to execute on different events like getting a GET/POST request, or when the server starts. As mentioned earlier, the procedural part of the server is hidden inside the server’s implementation.
- We don’t have to manually parse the input stream for the request. Express takes care of doing the parsing for us, and passes the resulting object to our handlers (in the function’s first parameter).
- We don’t have to convert strings to bytes in order to return them in the response. Again, Express takes care of that with **`res.send()`**.
- For rendering static files like **`index.html`**, we don’t need to manually read each file from the server. **`app.use(‘/static’, express.static(‘public’))`** takes care of linking a specific sub-route to a folder in the server, mapping URL paths to file names.

We can see one of the advantages right out of the box when we try to enable the response **`compression`** in Express:

```
var compression = require(“compression”);
var express = require(“express”);
var app = express();
app.use(compression());
```

If we wanted to enable compression in our own custom server, we would have to implement it ourselves: check the request headers and see whether the compression should be enabled in the response, in addition to actually compress the response using something like gzip before returning it back to the client.

Enabling HTTPS in Express

HTTP requests and responses are nothing else than text being sent back and forth through the network between the client and the server. Sensitive information like authentication tokens, session tokens, or any information within the requests themselves will be sent in plain text.

Anyone connected to a network can use a *packet sniffer* (a tool to read all traffic in the network) and read the request and responses of the clients connected in the same network.

HTTPS provides an encrypted channel so clients and servers can communicate securely. All traffic is encrypted; and while the encryption itself is not enough to keep malicious users from accessing sensitive data, it's definitely better to use HTTPS than HTTP. Besides, most of the search engines like Google will penalize your site if it's still running in HTTP.

We can definitely add code to our custom server to enable HTTPS. However, the list of the requirements that a server needs to comply with in order to fully support HTTPS is long, and would take longer than we can fit in this chapter. If you're interested in looking at these requirements, take a look to the "*RFC 7230 - Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*".

The advantage of using a production-ready server is that someone else already took care of implementing the list of requirements for HTTPS, so all we need to do is to provide an SSL certificate that will be used to sign all encrypted requests.

SSL certificates provide information to clients about the authenticity of the server. It provides identity information that is validated by a **Certificate Authority** (**CA**).

There are a couple of ways to get an SSL certificate:

- Create a self-signed certificate using OpenSSL.
- Buy a SSL certificate for your domain using one of the many hosting providers out there.
- Get a free SSL certificate for your domain using *letsencrypt.org.*

If you buy your certificate or use `LetsEncrypt`, your vendor would provide you with the steps to get the required certificate and key files for your application.

Note: LetsEncrypt versus paid certificates

Paid certificates typically provide more levels of validation (supported by a Certificate Authority) than LetsEncrypt, and they can give a better sense of protection to your clients and better placement in search engines.

LetsEncrypt is built so that the small projects can protect their data for free; hence, it's great for websites without a lot of sensitive data like blogs or personal sites.

If you plan to get a certificate for your business, consider buying one.

Self-signed certificates are useful for development environments. They are not great for production deployments, though, as the client will get warnings about self-signed certificates not being enough to validate the server's identity.

If you have OpenSSL installed, you can generate a self-signed certificate with the following command:

```
openssl req -nodes -new -x509 -keyout server.key -out
server.cert
```

Once you have a valid certificate and keys, the following code will enable HTTPS in your Express application:

```
var express = require("express");
var fs = require("fs");
var https = require("https");
var app = express();

app.get("/", function (req, res) {
  res.send("hello world");
});

https
  .createServer(
    {
      key: fs.readFileSync("server.key"),
      cert: fs.readFileSync("server.cert"),
    },
    app
  )
  .listen(3000, function () {
    console.log(
      "Example app listening on port 3000! Go to https://localhost:3000/"
    );
  });
```

Notice that the code for the HTTPS-enabled server is only slightly different from the HTTP server. Express is taking care of all the details for encrypting requests and responses.

Web application servers encapsulate all the boilerplate logic required to handle HTTP requests and responses. As we can see with the Express server, they provide a place for you to put your own code. This abstraction allows you to concentrate on building business logic instead of having to take care of implementing HTTPS for every project.

Layered architecture: Fully splitting the client from the server

Traditional web servers had the responsibility of storing/processing data, and rendering the application's HTML. This was the only approach available to build dynamic web applications with a technology like PHP.

The following PHP script queries a list of products from a database and renders an HTML with a table containing the results:

```
<?php
$servername = "localhost";
$user = "johndoe";
$pass = "Admin123";
```

```
$dbname = “products_db”;

// Connect to the database
$conn = new mysqli($servername, $user, $pass, $dbname);

$query = “SELECT id, product_title, product_description FROM Products”;
$query_result = $conn->query($query);

if ($query_result->num_rows > 0) {
    echo “<h1> Products </h1>”
    echo “<table>”
    echo “<tr><th>ID</th> <th>Title</th> <th>Description</th></tr>”;
    // output data of each row
    while($row = $query_result->fetch_assoc()) {
        echo “<tr><td>”.$row[“id”].”</td> <td>”.$row[“product_title”].” ”.$row[“product_description”].”</td></tr>”;
    }
    echo “</table>”;
} else {
   echo “No products to display”;
}
$conn->close();
?>
```

The following is a step-by-step explanation of what this script is doing:

- Make a request to a MySQL database to retrieve the list of names from the table **`MyGuests`**.
- Retrieve the list of results from the query.

- If the list of results is empty, display a `0 results` message. Else, dynamically create an HTML table with the results and render it.
- Return the generated HTML.

This approach is a bit different from the ones we've seen in our custom web server and in the Express server. PHP follows a fully procedural approach.

If you need to hire a developer to maintain an application which is built with a bunch of scripts like the PHP one we just saw, what kind of developer would you hire? A frontend developer? A backend developer? The coupling between front and backend in that application would make it really difficult for teams of front and backend developers to collaborate successfully.

There is a strong coupling in this script between the presentation layer and the business layer. Of course, a production-ready application wouldn't use a single script, but even if you split it into multiple files, nothing enforces a clear separation of concerns.

Tip Search term: Separation of concerns

This design pattern is closely related to the Single Responsibility Principle in Object Oriented programming.

The idea is that each unit of code should only have a defined goal or responsibility. Code with similar goals would be put in the same place.

Separating your code by the concern they address simplifies your project. Finding the correct categories of concerns for grouping your code is something you have to do based on your specific case or domain knowledge.

Also, while these kinds of scripts kind-of worked in the past when we only had browsers as clients, what would you do if you wanted to support native mobile apps? You would have to either rebuild the application or build something completely different.

One approach to solve this thigh-coupling between front and back end is to use a layered architecture, as shown in the following figure:

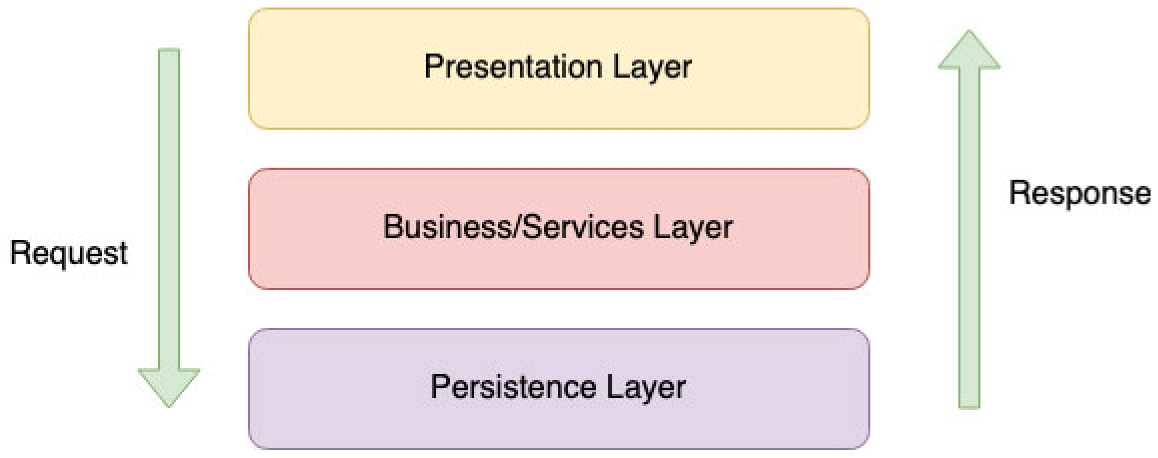


Figure 2.5: *Layered architecture.*

In some literature, the business and services layers are separate, and we have an extra data access layer after the persistence layer. The idea is just to organize your applications by the concerns they address.

In the layered architecture, we organize our applications in layers such as:

- **Presentation layer**: All code related to display and format data for the end user.
- **Business/Services layer**: All code related to transform data to the expected format to accomplish all functional requirements.
- **Persistence layer**: All code related to the interaction with the external data storage: Saving and retrieving data. It translates data from the format in which it is stored into a format that can be used by the business and service layers (e.g. relational tables to Java classes).

Note: The layered architecture as presented in most books and educational resources might not be the best distribution for all projects. The most common use of layered architecture is in monolithic applications, and in such, things like deployment or scalability can be cumbersome.

Actually, many resources even set up the layered architecture as a strict alternative to client-server. This is a very opinionated and pragmatic approach, as we can build client-server applications that also follow a layered architecture..

The main takeaway from there is that the separation of concerns allows our developers to work independently and in parallel.

Be flexible while implementing architecture patterns, and apply the concepts from these architectures which make sense for your specific case.

How do you actually split your code? It depends on the technology, framework, or even language you're using. The idea is to split your code in a way that allows teams work independently without blocking each other.

Splitting clients

One advantage of the separation between the presentation and the business layers is that we can not only have a layer in our server for the presentation code, but we can also extract it completely from the web server.

In modern development of client applications, the presentation code is build and deployed independently from the server code. For instance:

- Web applications projects using tools like React or Angular, deployed to their own web servers.
- Native mobile applications created with tools like Swift for iOS or Kotlin for Android, and deployed to app stores for clients to install them directly in their devices.

This separation of concerns is great! You can hire teams of web, Android, or iOS developers and they can work in parallel to API and server-side developers.

All you need to do in order to guarantee that this independence holds in place is to create well-defined contracts between the web server and these presentation-based clients. We will visit this concept of contracts in *Chapter 3, Designing APIs*.

Client versus server computing

A common interview question is: Considering the power of modern clients and current servers, where should you do most of the processing work for an application?

The answer like in every question that deserves to be asked is *it depends*.

The advantages of shifting processing work towards the client are as follows:

- **Less latency**: Since clients don't need to make requests to the server for any operation, they can save all the time used in network calls and improve the user experience.
- **Less load for the server**: The server is more performant and will be able to support more clients.
- **Increased privacy**: Dealing with sensitive personal data becomes easier as your user's personal information never leaves their devices; you don't have to worry about things like encryption in traffic.

The disadvantages, of course, are as follows:

- **Low consistency**: Certain sensitive operations (like bank transactions) require high consistency requirements. It's not possible to fully perform these kinds of transactions in the client's device, as they need to be consistent with the information of all other users.
- **Access control concerns**: There are certain resources that cannot be sent securely to a user's device. Sensitive information from other users in the system might be required for the operation, and that information can only be safely operated in the server.
- **Performance concerns**: Some operations might be too computationally heavy for a client device.
- **Version fragmentation**: Not all client devices run the same client version. Some browsers are outdated and some Android devices run in previous OS versions. Since we cannot guarantee that all clients run in the latest versions of their software, not all of them will have the same feature set available. A good example is all the browser-compatibility issues which have plagued web developer since decades ago.
- **Variable client performance**: Similar to version fragmentation, not all devices have the computing power to perform memory-intensive operations. Also, the more we shift operations into the client, the more power it will require, causing battery problems in mobile devices.

Moving business logic into a client should be done in an *as-needed* basis. A good rule of thumb is to start by putting all the business logic in the server and slowly move pieces of logic into the client, as it makes sense.

The features that would bring the biggest performance gains are those that are mostly constrained by network latency and that don't require a lot of processing power to be executed.

An iterative approach will give you more control and better insight into which pieces of logic do bring a positive impact to your users.

Web servers as stateless services

By design, HTTP is supposed to be stateless. This means that the HTTP server itself is not expected to keep track of any state information; the server won't remember whether a client has done a request before, or if a request is related to another.

By having HTTP servers being stateless, they can be treated like interchangeable resources. This provides multiple advantages:

- **Better performance**: You can create a cluster of multiple servers running the same web application. Then, put a Load balancer in front of them to redirect client's requests to any of the servers in the cluster, following something like Round-robin.
- **Better availability**: If you have a cluster of stateless servers, if any of the servers fails, you can easily replace it with a new one, without worrying about backups.

If servers are stateless, how do we keep track of user sessions?

We have designed workarounds to make it seem like HTTP stores the state. Some of those tricks are as follows:

- Keeping track of a user session by sending a session token in each request through cookies.
- Calling a web API sending authorization tokens in each request to allow the server to know which user is making the request.

In each request, the HTTP server will validate the token as if it was the first time it received it. It doesn't matter which server in the cluster received the request; all of them perform the same action.

These workarounds are embedded in the way our clients work. Browsers store the session cookies and make sure users don't have to input it in each web page they visit.

Storing session data

Certain applications are required to store data that is either too big or too sensitive to be passed back and forth between the client and the server in each request. In those cases, we want a way to persist the state of a user's session.

Some developers feel tempted to store this session state in the same server; either in an in-memory structure, a local file, etc. And they even work around the way Load Balancers function by creating "*sticky sessions*": Once a load balancer directs a request to a server, it will redirect all further requests from the same client to the same server.

Keeping storage in the web server itself is not optimal, as it prevents load balancers from distributing requests evenly across servers in a cluster, and in case of server failure, you will probably lose that state (unless you add an extra strategy to back it up before the failure happens).

So, where do you keep the session data?

Low-durability caching services like Redis address this kind of problem: they offer a storage service which is external to the HTTP servers themselves. Each server stores the session state in the same Redis cluster. If the HTTP server fails, a new server can be deployed and connected to Redis. Redis is fast enough that it has almost no downsides when compared to the local in-memory local storage.

We will discuss more data storage strategies in *Chapter 5, End-to-end Data Management*.

Use case: Applying a client-server architecture to the Pizza Place app

Let's take a look at two of our *functional requirements* for the Pizza Place app:

- Users should be able to see the menu on their phones or computers.

- The menu should be updated by the managers at the Pizza Place so that the users can only make orders of pizzas for which they have ingredients in stock.

From requirement number one, we can deduce that we want to build both a mobile app and a web app. This means that we will have to support at least three different clients: Web, Android, and iOS.

From the second requirement, we know that the information given by the app to the users' needs to be frequently updated (at least a couple of times a day). This means that the clients will need to fetch the most up to date menu from somewhere, instead of keeping the menu as a static website.

From these two simple requirements, we can see that we need to rely on a client-server architecture, instead of doing something like static, purely client-based apps.

Now, let's review the first **non-functional requirement** we have:

- The application should be as small as possible to correctly serve the small user base the Pizza Place has right now.

We know that we are building a small application that is expected to grow. This means that we have the flexibility of hosting the whole application in a single server; but we also have to structure it so it can grow once it's needed.

For this, let's apply a modified version of the layered architecture. Since we need multiple clients, let's completely split the presentation layer into independent client applications (one project for web, one for native mobile).

Now, for a large application, we would also split the business, service, and persistence layers into their own servers. However, our current requirement points us to put everything in a single server. So, a healthy balance is to create logical layers.

This means that we will create a single project for these three layers, but we will split the code into functions or classes grouped by each of the layers. This will create a healthy separation between code for business functionality and code for data persistence.

As mentioned earlier, don't get caught in rigid definitions of these architectures. They are recipes we can tweak and adapt to our specific use case.

The overall application structure for the Pizza Place application is shown in the following figure:

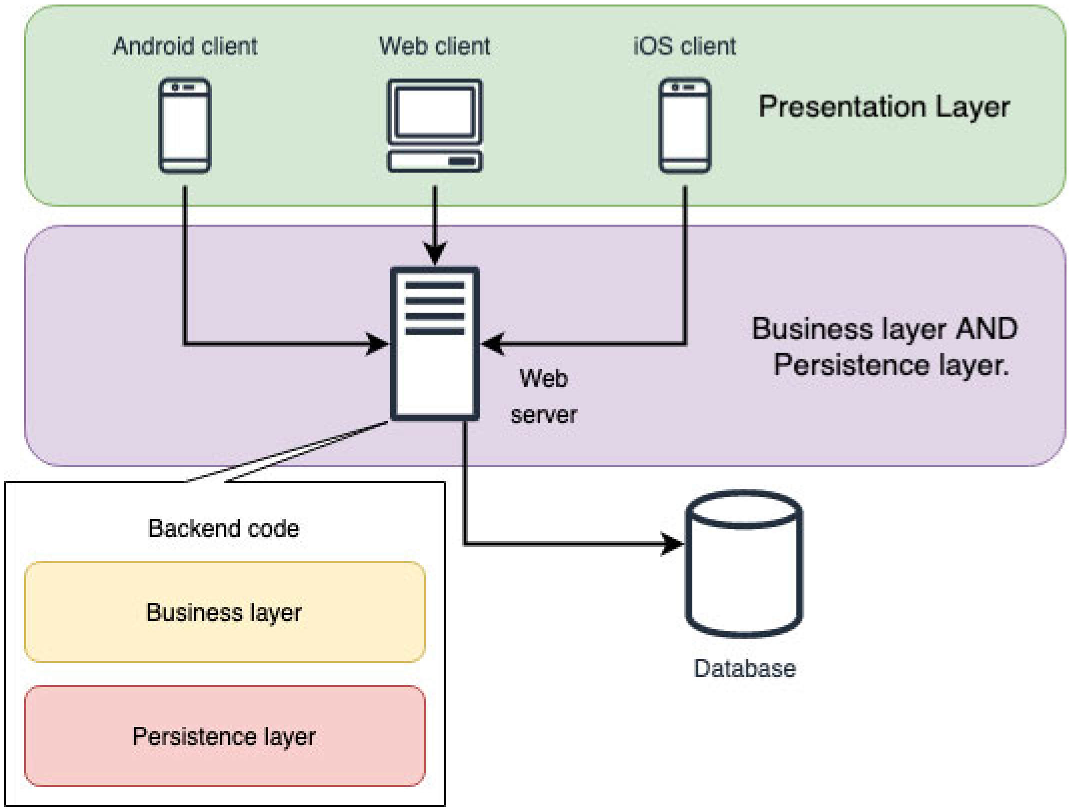


Figure 2.6: *High-level view of the architecture for the Pizza Place application*

We are using this design for the first version of our application because it addresses all our requirements and we are still taking advantage of the benefits of the client-server and layered architectures.

The logical layers approach allows us to iterate faster, as we don't have to worry about managing multiple servers or projects. The logical separation still separate concerns and, as our application grows, these layers can be extracted into their own servers as needed.

Client server versus peer-to-peer

Before we complete reading this chapter, it's important to acknowledge the existence of alternative architectures. **Peer-to-peer** (**P2P**) architectures differ from the client-server architecture in the fact that here each node in the network is both client and server.

In *Figure 2.6*, we can see the topology of a typical P2P network with five peers:

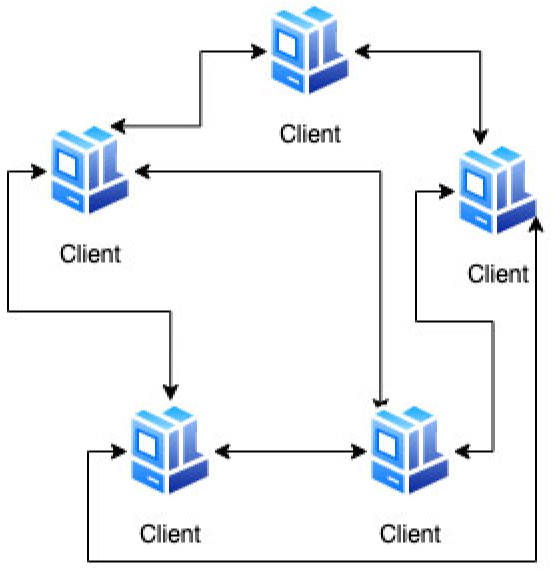


Figure 2.7: *Example of a Peer-to-peer architecture. Peers connect to each other*

Data storage and processing is distributed across multiple clients or peers. A client connects directly to its peers to fetch data they need, and if for some reason the peer goes down, the client can connect to any other peer in the network.

The main benefit of P2P is that it is a decentralized architecture. It means that there is no single server that can be a bottleneck or a single point of failure.

Another benefit is that you don't need servers with tons of memory to host an application, as the computing is distributed among the peers. This could be a very powerful architecture for collaborative work.

The most famous implementations of P2P are BitTorrent and Blockchain. BitTorrent enables file sharing among all the peers in the network, and Blockchain makes copies of the ledger of transactions in each peer.

It's curious to know that the adoption of both of these in the public has proved to be controversial at times, due the potentially gray areas that the use of these protocols brings with them: The lack of centralization also means that no single entity can regulate the application's content.

There has been some exploration about implementing HTTP as a P2P protocol, as well as possible P2P-based replacements. However, there is so much infrastructure in place that already relies on HTTP and the server-client architecture that it’s not easy to find a suitable replacement.

For better or for worse, HTTP is here to stay for the near future.

Conclusion

The client-server architecture is the base on which web applications are built. This architecture model has enabled easily sharing with people the resources or data that is concentrated in a central computer.

The client-server architecture is used in multiple layers within our applications: In HTTP, in database server connections, in mobile apps, FTTP clients, etc. User interfaces are only one type of client, and anything that connects to a server can have the role of client.

We saw the inner workings of a web server implementing HTTP: how to build the main process for a server, how to listen for requests and provide responses. We also reviewed some examples on the kind of resources we can return to clients from a server, either static or dynamic.

We discussed HTTPS, SSL certificates, and the importance of enabling them in a web server. Production-ready servers like Express provide a simple interface to configure horizontal concerns like encryption or data compression in server responses. No modern web application should ever be deployed to production without enabling HTTPS first.

We looked at how legacy web applications can combine multiple concerns in single source code files, like common PHP examples, resulting in highly cohesive teams of developers that cannot work on the application in parallel.

We talked about layered architectures and how they try to address a better separation of concerns. And, taking a step further in separating presentation concerns into its own project, we end up with the architecture followed by modern client applications developed for web, Android, or iOS. Client applications run in their own projects and can be deployed independently from the server which implements business logic.

Balancing the implementation between business logic between clients and servers require fine tuning, as there are trade-offs between putting too much logic in each side. If you want to err on the safe side, most of the time the server should contain the majority of the business logic.

We understood that web servers should be built and deployed as stateless resources. The lack of state at the web server level allows us to discard and spin new instances of the server; or to seamlessly split requests evenly across multiple servers in a cluster.

Lastly, we talked about some alternative architectures like peer-to-peer. P2P architectures can perform better than client-server for cases where data needs to be de-centralized.

Having a strong understanding of how HTTP works; we now have most of the tools required to take a deep dive into the topic of designing and building APIs, which we will discuss in the next chapter.

Questions

- Can a software application be both client and server? Why?
- What is the role of load balancers in clusters of HTTP servers?
- What are the advantages of processing information directly at the browser or mobile applications?
- What are the advantages of using a layered architecture? Can you think of a use case where it might not work great?
- What would happen if we store the user session in the web server and that server crashes? What would the user see?
- Given that HTTP is stateless and we send authentication tokens in each request (usually through cookies), what would happen if we used unencrypted HTTP and someone could read our request information? How would a malicious person use that sensitive information?

References

- Java Stream API: **https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html**

- RFC 7230 - Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing

  https://datatracker.ietf.org/doc/html/rfc7230
- Free SSL certificates at LetsEncrypt

  https://letsencrypt.org/

CHAPTER 3
Designing APIs

Any communication between two entities requires some level of agreement between both. If one person speaks to another, there is an assumption that both persons share a language (and a common context) that would allow them to decode the meaning of the words the other person is saying. Without this agreement, words are just noise. Languages allow us to encode and decode this noise into meaningful concepts.

This agreement also exists between elements in a computational system who try to communicate. At the lowest level, we have protocols like TPC/IP or UDP, which dictate a *contract* on how two computers communicate in a network: The sequence of steps required to establish a connection, the agreed structure for the packets to be sent back and forth, and so on.

Computers receive streams of bits sent by other computers. Network protocols provide a blueprint on how to group these bits into data structures that have a meaning for the computer and its users. These blueprints are the "*language*" computers speak.

At a higher level, communication between two computers (a client and a server, for instance) has protocols that allow sending complex data back and forth, like HTTP. We have seen how client-server architectures rely on a shared contract that describes how HTTP requests and responses should be structured. As long as clients and servers both follow the predefined contracts, they will be able to communicate successfully.

In an attempt to create more complex contracts, techniques like REST and SOAP have been created on top of HTTP. They can be grouped in the concept of web services but in the past few years, they have also been labeled as "*APIs*".

In this chapter, we will discover all you need to know about APIs. Since "*API*" is a very abstract concept, we will disambiguate the most common use cases.

We will also discuss the most currently popular implementation of web services: REST. We will review some best practices for designing clean REST APIs and how REST services contrast against other protocols like SOAP and gRPC.

Then, we will discuss how APIs provide flexibility in distributed applications and how they can be leveraged independently from the front-end clients which consume them.

Structure

In this chapter, we will learn the following topics:

- What is an API?
    - Functions as contracts
- Interfaces and design patterns
- Remote APIs: RPC, SOAP, REST, GraphQL
    - Building a remote API with RPC/gRPC
    - SOAP and web services
- Building REST APIs
    - Actions versus HTTP request methods
    - Relationships
    - Versioning
    - Caching
- Effective REST APIs: HATEOAS
    - Building APIs with GraphQL
- Building standalone APIs
    - Standalone API: Headless CMS
    - Standalone API: Public APIs
- Use case: Designing a remote API for the Pizza Corner

Objectives

After reading this chapter, you will be able to understand the different types of programming interfaces and their uses. You will know the advantages of using interfaces and APIs to abstract out implementation details. Also, you'll understand the use cases and differences between the major tools to build remote APIs: REST, RPC, SOAP, and GraphQL. You will have a basic understanding of how to write and design effective APIs that are flexible enough for multiple clients to consume.

What is an API?

An **Application Programming Interface** (**API**) is a **contract** that defines all the information needed for two software-based entities (functions, applications, servers, etc.) to communicate with each other.

Every API describes the following two basic things:

- A list of models describing the expected format of the data to be transferred in the API's actions.
- The actions that are available to be performed on with those models or entities.

Some specific types of APIs will have other characteristics (for example, REST APIs define an HTTP verb to be used, SOAP defines an XML structure, and so on) but in the end all APIs share at least these two requirements.

Functions as contracts

At the lowest level, you can think of function signatures as a part of an API. For instance, the `std` package in C++ provides a function `to_string`. If you look it up in C++'s documentation, you will find a list of method signatures like the following:

```
string to_string (int val);
string to_string (float val);
string to_string (double val);
string to_string (long double val);
```

Notice that these signatures contain the following two basic requirements we require for an API:

- The expected format of the function input: **`float`**, **`double`**, among others.
- An explicit list of the allowed operations (for example, convert an input into a **`string`**).

How does the **`std`** package implement each one of these functions? We don't know, and most of the time, we don't care. We have the guarantee that, as long as you call the function with the expected parameters, you will receive back a string representation of the input.

Of course, you could go and look for the source code of 'std' and find the implementation, but you don't need to know it in order to use the function. Just think of all the standard library functions you have used in your code for which you have never seen their implementation.

It's safe to assume that the list of function signatures is the API of the **`std::to_string`** package. It might seem a bit redundant that each function is called exactly as the package that exposes them, but this is more a design decision for this specific example than a problem with our definition of API.

Interfaces and design patterns

In some coding languages, the concept of API is distilled as a component of the language itself. In Java and C# we have *interfaces*: A special structure that defines a contract. In general, the interface cannot be used directly and a concrete class needs to implement it.

Here is an example of a simple C# interface:

```
interface ISampleInterface {
    void SampleMethod();
}

class ImplementationClass : ISampleInterface {
    // Explicit interface member implementation:
    void ISampleInterface.SampleMethod() {
        // Method implementation
    }

    static void Main() {
        // Declare an interface instance
        ISampleInterface obj = new ImplementationClass();
        // Call the member
        obj.SampleMethod();
    }
}
```

In the preceding example, the interface **`ISampleInterface`** is defined with a single function **`SampleMethod`**. The function has no parameters and returns no value.

Interfaces are one of the purest examples of an API. Notice how the implementation only defines the method signature, with no body. If you tried to instance **`ISampleInterface`** directly, the compiler would give you an error.

For this reason, we will create a concrete class **`ImplementationClass`** that implements the **`ISampleInterface`** interface. C# (as well as Java) requires any concrete class implementing an interface to also implement each function in the interface. That's the reason why **`ImplementationClass`** defines a body for **void `ISampleInterface.SampleMethod()`**.

Java's example is very similar:

```
interface ISampleInterface {
    void SampleMethod();
}

class ImplementationClass implements ISampleInterface {
    @Override
    void SampleMethod() {
        // … basically the same body
```

When first learning interfaces as part of Java or C#, many people ask themselves: *Why go through the trouble of defining an interface if I will have to define each function again in the concrete class?* It's a natural question, as it seems like a waste of effort.

The answer is that *we can write multiple implementations for the same interface,* and those implementations can be used interchangeably. As long as consumers use the interface directly, they don't care which implementation they are getting. Let's look at an example.

Let's say we want to define a service to store an **`Order`** object in a database. Today, we will use a MySQL database, so our **`OrderService`** class gets an instance of the **`MySQLService`** to save the data.

The following is an example service called **`OrderService`** that shows how the application uses **`MySQLService`** to persist Order objects:

```java
class MySQLService {
    boolean saveOrder(Order order) {
        // …
    }
}

class OrderService {
    MySQLService dbService;

    public OrderService(MySQLService dbService) {
        this.dbService = dbService;
    }

    void processOrder(Order order) {
        // do things to the order, then save it:
        this.dbService.saveOrder(order);
    }
}
```

```

class App {
    void newOrder() {
            OrderService service = new OrderService(new MySQLService());
        // get new order, process it and store it…
        service.processOrder(order);
    }
}
```

Notice how we are passing **`MySQLService`** through the **`OrderService`** constructor, instead of calling the **`MySQLService`** constructor inside **`OrderService'`**. This is called **inversion of control** (**IoC**), and it help us remove a strong dependency between both classes.

Continuing with our example, a few months later after we discover we need to migrate the MySQL database to a MongoDB database. Taking advantage of the IoC pattern, we create a new service called **MongoDBService** and we inject it to the **`OrderService`**.

The following is the class definition of **`MongoDBService`**:

```
class MongoDBService {
    boolean saveOrder(Order order) {
        // …
    }
}
```

Having used IoC made this change a bit easier, but we still need to update both **`App`** and **`OrderService`** to use this new service class. This means that we have to update every line of code where **`MySQLService`** is defined and used.

If we had relied on interfaces, this migration would have been a lot simpler. The following code shows an interface-based implementation of **`DbService`**:

```
interface DBService {
    boolean saveOrder(Order order);

    //… other methods, maybe like getOrder
}
class MySQLService implements DBService{
    boolean saveOrder(Order order) {
        // Do SQL stuff…
    }
}

class MongoDBService implements DBService{
    boolean saveOrder(Order order) {
        // Do Mongo stuff…
    }
}

class OrderService {
    DBService dbService;

    public OrderService(DBService dbService) {
        this.dbService = dbService;
    }

    void processOrder(Order order) {
        // do things to the order, then save it:
        this.dbService.saveOrder(order);
    }
}

class App {
    void newOrder() {
        // Use one implementation…
        OrderService s١ = new OrderService(new
MySQLService());
        // …or the other
        OrderService s٢ = new OrderService(new
MongoDBService());
        s1.processOrder(order);
        s2.processOrder(order);
    }
}
```

The consumer uses our **`DBService`** API directly. Then, the concrete implementation is passed to the constructor of **`OrderService`**. This is a pattern known as **dependency injection**, which is just a more specific version of IoC.

We can swap the implementation any time, or even have two instances of **`OrderService`** using different implementations of **`DBService`**. We can create more implementations without ever needing to update **`OrderService`**.

The assurance here is that, as long as **`OrderService`** follows the contract defined in the interface, any service that implements that same contract will work as expected. These implementations can be local functions, clients for remote APIs exposed by other servers, or even test mock functions. The consumer doesn't need to know which implementation they are using.

The use of APIs at the programming interface level allows us to implement some of the most common *Gang of Four's structural design patterns*:

- **Adapter pattern**: It allows two incompatible classes to work together by wrapping an interface around one of the existing classes.
- **Facade pattern**: It provides a simple interface to a more complex underlying object.
- **Proxy pattern**: It provides a placeholder interface to an underlying object to control access, reduce cost, or reduce complexity.

All these patterns rely on interfaces introducing an abstraction layer that removes complexity or incompatibility of a consumer with other services.

Remote APIs: RPC, SOAP, REST, and GraphQL

When we think of APIs, there's a good chance we are thinking of executing code running in a remote machine. The simplest case we might think of is having a web or mobile client that makes a request to an "*API*", which in turn is hosted in some remote server.

Using our previous definition of API, a remote API is a server that exposes an interface for clients to read and consume, but the communication happens through a network.

In our previous example, `boolean saveOrder(Order order)` was declared and implemented in the same application as the client that consumed it. What if, instead, we wanted to save the data in a database running in a remote database?

Just as we could swap implementations to use MySQL or MongoDB without having to update the client service, we can create an implementation that makes a request to an external server. This is known as **Remote Procedure Call** (**RPC**).

The following figure is a high-level view of the multiple implementations of `DBService`:

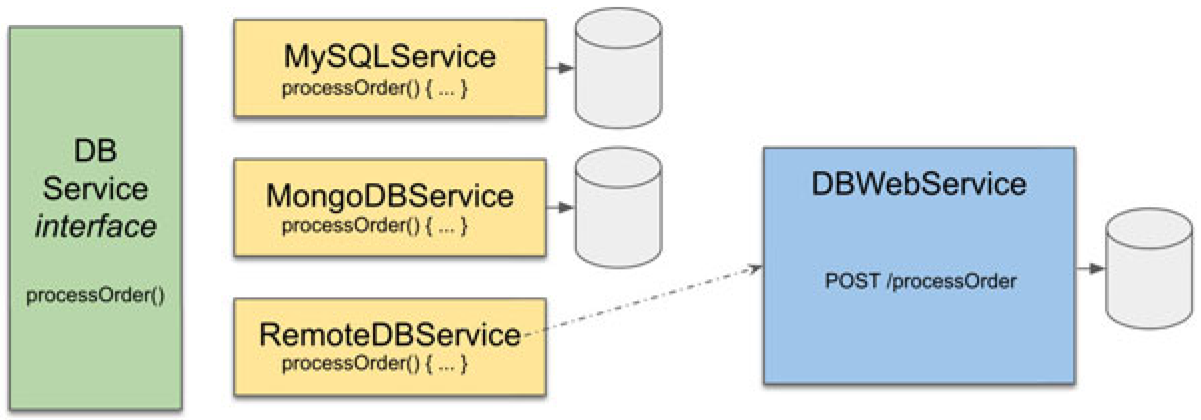


Figure 3.1: *Defining multiple implementations for a programmatic interface*

RPC allows the code to make requests to external servers by calling RPC functions which are no different from regular, local functions (like `boolean saveOrder(Order order)`). RPC implementations abstract away all the logic required to communicate with the external server so the developer can focus on writing business logic.

Building a remote API with RPC/gRPC

There are many implementations of RPC, from Java's **Remote Method Invocation** (**RMI**) or **Common Object Request Broker Architecture** (**CORBA**), all the way to the newer gRPC.

However, at its core, RPC can be done with a simple HTTP implementation. For instance, to support an RPC version of **`saveOrder`** or **`getOrder`** that relies on HTTP, you can create the following endpoints:

```
POST /api/saveOrder
GET /api/getOrder?id=123
```

Now, you can create an implementation for **`DBService`** which makes requests to those endpoints:

```
class RemoteDBService implements DBService{
    HttpClient httpClient;

    boolean saveOrder(Order order) {
        // create http request from order
        // …

        HttpResponse<String> = httpClient.post(
                                                    HttpRequest request = HttpRequest.newBuilder().uri(
                        URI.create("https://pizzaplace.com/api/saveOrder"))
                                                        .header("Content-Type", "application/json")
                    .POST(orderRequest)
                    .build()
        );

        // do something with the response
    }
}
```

This is the power of *programming interfaces*. We can not only change the type of database out application uses almost seamlessly, but also we are able to migrate from using a local implementation to make requests to an external server. All without changing the code that calls **`saveOrder`**.

"*Hold on*", you might say. "*I've seen this before. This is not RPC, this is REST!*"

It is a very common misconception to think that making any HTTP requests to a server using a few distinct HTTP verbs is REST. Actually, the most popular instance of RPC in the recent years, gRPC, relies internally on HTTP/2 for communication. To these web endpoints that look suspiciously close to a REST API we call them '*RESTful services*'. Protocols like OAuth rely on RESTful calls without being fully REST.

Without proper context, it's easy to get confused about the differences between the multiple tools we have to build remote APIs. For example, there is the notion that RPC is an outdated and undesirable way of building APIs. This is far from being true.

In reality, RPC is a pattern very useful for building modern applications. Maybe it wasn't as widespread a few years ago because most implementations of RPC where vendor or language specific (like Java's RMI), while SOAP and REST started to allow developers to integrate applications built in different tech stacks. But the popularity of new protocols like gRPC or Thrift has given a new breath of fresh air to this pattern.

To underscore its advantages, let's leave aside HTTP and REST for a second and focus on a specific example of modern RPC: gRPC.

Figure 3.2 shows the communication flow between a gRPC server and the client making a remote call to the **`SayHello(…)`** method:

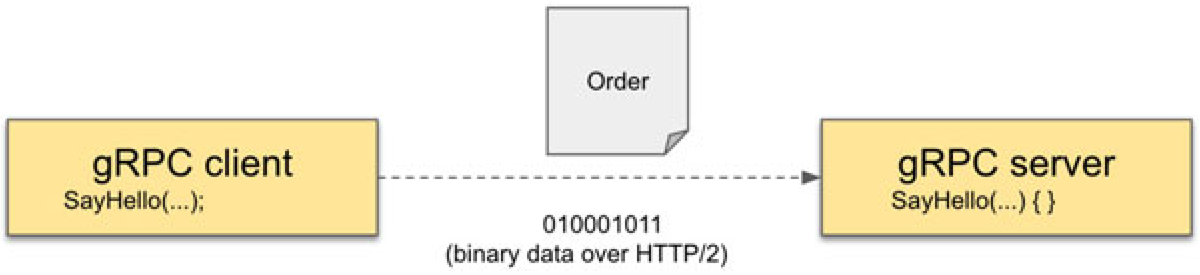


Figure 3.2: *Example of a gRPC request*

Relying on Protocol buffers (protobuf), gRPC creates an interface using the language definition **`proto3`**, which defines models and actions. These simple models are then processed to auto-generate the code to be used by both the server and the client to communicate.

The following is an example of a protobuf interface model:

```
syntax = "proto3";
package helloworld;
option go_package = "models/helloworld";
option java_package = "com.example.grpc"

message Message {
    string body = 1;
}

service HelloService {
    rpc SayHello(Message) returns (Message) {}
}
```

Here, we are defining a **`helloworld.proto`** file that contains one single remote method, **`SayHello`**, and a model called **`Message`**.

The other options described at the top of the file are metadata which protobuf will use when generating code; for example, **`go_package`** is used for creating a Go module while generating the Golang code, and **`java_package`** gives a package name to the generated Java classes.

Tip: Search term: protobuf

A protocol buffer is a library built to serialize data. It relies on the creation of models and actions written with the language "proto3" which is a definition language similar to XML or JSON.

This library can generate code based on the models built in proto3. It also has plugins that allow you to generate code for creating clients and servers using gRPC.

We can build this model using any of the supported languages. For instance, you can generate the gRPC client-server code for Java with the following command:

```
protoc --java_out=api --plugin=protoc-gen-grpc-java proto/helloworld.proto
```

This will generate the **`com/example/grpc/Helloworld.java`** class inside the **`api`** folder. There are plugins that integrate **`protoc`** with common frameworks like Maven or Gradle, which will allow you to generate the API's code as a part of your application compilation workflow.

Alternatively, you can run the following command and generate a Golang version of the same interface:

```
protoc --go_out=plugins=grpc:api proto/helloworld.proto
```

This command will generate a Go file called **`helloworld.pb.go`** inside the **`api/models/helloworld`** directory, just as the **`helloworld.proto`** file described.

This generated code is not meant to be edited directly. If you want to make changes, update the model and regenerate the code. Otherwise, you risk other developers (or yourself) overwriting your changes when they re-generate the API's code.

Build the gRPC server

Let's use Golang to build both a client and a server for a gRPC API. Assuming you have a basic set up of Go installed on your local development machine, create a module **`go.mod`** with the following definition, which includes all the required dependencies for gRPC:

```
module example.com/grpc
```

```

go 1.17

require (
    golang.org/x/net v٢٠٢٠٠٨٢٢١٢٤٣٢٨-٠.٠.٠-c٨٩٠٤٥٨١٤٢٠٢
    google.golang.org/grpc v١.٤٠.٠
    google.golang.org/protobuf v١.٢٧.١
)



require (
    github.com/golang/protobuf v١.٥.٢ // indirect
    golang.org/x/sys v٨٥-٢٠٢٠٠٣٢٣٢٢٢٤١٤-٠.٠.٠ca٧c٥b٩٥cd // indirect
    golang.org/x/text v٠.٣.٠ // indirect
    google.golang.org/genproto v٢٠٢٠٠٥٢٦٢١١٨٥٥-٠.٠.٠-cb٢٧e٣aa٢٠١٣ // indirect
)
```

If you don’t want to create a module, install each dependency manually using the **`go get`** command:

```
go get google.golang.org/grpc@v1.40.0
```

Then, create a file called **`server.go`**. This file will contain the code for the gRPC server Then, **`add`** the following code to server.go:

```
package main

import (
    “fmt”
    “log”
    “net”

    helloApi “example.com/grpc/api/models/helloworld”
    “golang.org/x/net/context”
    “google.golang.org/grpc”
)


type ServerHandler struct {}

func (s *ServerHandler) SayHello(ctx context.Context, in *helloApi.Message) (*helloApi.Message, error) {
      log.Printf(“Receive message body from client: %s”, in.Body)
        return &helloApi.Message{Body: “Hello From the server!”}, nil
}

func main() {
    fmt.Println(“Starting gRPC server!»)

     listener, err := net.Listen(“tcp”, fmt.Sprintf(“:%d”, 9000))
    if err != nil {
        log.Fatalf(“Error with server connection: %v”, err)
    }

    s := ServerHandler{}

    grpcServer := grpc.NewServer()

    helloApi.RegisterHelloServiceServer(grpcServer, &s)

    if err := grpcServer.Serve(listener); err != nil {
        log.Fatalf("Error serving response: %s", err)
    }
}
```

If you're not proficient in Golang, don't worry. This code is simple enough that you can understand the most important parts without a strong knowledge of the language. But, if you want to feel more comfortable with the code, take 20 minutes to check out the website "*A tour of Go*" at **https://tour.golang.org**. That should be more than enough knowledge to fully understand this example.

This sub-module **server.go** is part of the main module **example.com/grpc** (which we defined in go.mod), so all our local code will be imported from that package name.

In the line **helloApi "example.com/grpc/api/models/helloworld"**, we import **models/helloworld** from the main **example.com/grpc** package. Again, **models/helloworld** is the code we generated with the **protoc** command.

We define a structure called **ServerHandler** that will contain all the code used by the server to handle the API requests. In this case, the structure has a single method, **SayHello** which just prints a message in the log and returns the string "**Hello From the server!**" back to the consumer.

The **RegisterHelloServiceServer** function was created by protobuff's gRPC plugin, and all we have to do is give it an instance of ***grpc.Server** and an instance of the **ServerHandler** to create a server.

If we compile and run this Go application, a server will start in the localhost:9000. For example, the following steps compile and run the server in a Unix-based OS (MacOS, Linux, etc):

```
> go build server.go
> ./server
Starting gRPC server!
```

If you're running in a Windows-based system, **go build server.go** will create a **server.exe** file that you can run.

Build the gRPC client

Now, in order to consume our RCP API, we need to create a gRPC client. The code for the client is as simple as the code for the server:

```
package main

import (
    "log"

    "golang.org/x/net/context"
    "google.golang.org/grpc"

    helloApi "example.com/grpc/api/models/helloworld"
)

func main() {
    var conn *grpc.ClientConn
    conn, err := grpc.Dial(":9000", grpc.WithInsecure())

    if err != nil {
        log.Fatalf("did not connect: %s", err)
    }

    defer conn.Close()

    c := helloApi.NewHelloServiceClient(conn)

    response, err := c.SayHello(context.Background(), &helloApi.Message{Body: "Hello From client!"})

    if err != nil {
        log.Fatalf("Error when calling SayHello: %s", err)
    }

    log.Printf("Response from server: %s", response.Body)
}
```

The important parts of the code can be split in two snippets. First, we connect to the remote API server that is running in **`localhost:9000`**:

```
//…
var conn *grpc.ClientConn

conn, err := grpc.Dial(":9000", grpc.WithInsecure())
//…
```

Since this is just a demo, we can use **`grpc.WithInsecure()`** to allow the gRPC client know that we will not use authentication or encryption in this connection.

In our second snippet, we have the code which actually executes the remote procedure **`SayHello`** and prints the response:

```
//…
c := helloApi.NewHelloServiceClient(conn)

response, err := c.SayHello(context.Background(), &helloApi.Message{Body: "Hello From client!"})

log.Printf("Response from server: %s", response.Body)
//…
```

Notice how, for the client, the second code snippet looks just like as if you were executing a local function. The type of response is **`*helloApi.Message`**, which is one of the auto-generated classes we created using the protobuf model.

If you compile and execute the client, you will see a message in the server like `2021/09/06 13:03:30 Receive message body from client: Hello From Client!`, and a message in the client's log similar to `2021/09/06 13:03:30 Response from server: Hello From the server!`.

When to use gRPC

Data transmission in gRPC is extremely efficient, as data is serialized and deserialized using binary data, unlike REST or SOAP which tends to use the text-based formats JSON and XML.

Notice also that the main idea behind RPC is that a local function implementation is moved to *a remote server, "hiding" from consumers that it's actually a remote call*.

RPC creates a tight coupling between the client and the server. Since we are abstracting functions from the client into the server, the API we build with RPC has a low level of abstraction, exposing more implementation details than other types of remote APIs like REST.

The tight coupling between producers and consumers makes RPC ideal for communication in environments of distributed systems -like in microservices architectures- where all the elements in the system interact with each other as if they were making calls to local functions.

One disadvantage of that tight coupling is that RPC has low discoverability, which means it's difficult for clients to figure out what kind of operations they have available to call. We have to rely on the assumption that the API owner will provide compiled clients, or at least consumers will have a list of all the remote functions, and that they fully understand what each function's name means.

A situation which illustrates the discoverability challenges for RPC is that of an API that has two separate functions (for instance, `saveOrder` and `createOrder)` that might behave differently in specific contexts, but they offer no good description about their differences. It's not easy for consumers to figure out which function will work best for their use case. Clear and updated documentation becomes a critical part of RPC; otherwise, people will not read it and fail to adopt your API.

In general, naming conventions (or the lack of thereof, as naming things is hard!) is one of the biggest obstacles for clients to adopt RPC APIs. RPC

APIs end up having increasingly complex methods like **`getOrderForRegularCustomer`** or **`getOrderForPreferredCustomer`** (we have to expose more functions as we need to support new use cases) which require consumers to always have up-to-date knowledge about the implementation details and context of the server to know which function to call.

SOAP and web services

We will not go in depth on the topic of **Simple Object Access Protocol** (**SOAP**), simply because it's not so commonly used anymore for new projects. However, we still need to acknowledge it exists so we can contrast it with other options.

When talking about application design, the concept of API is usually used interchangeably with the idea of web services: A few years ago, web services were a synonymous of SOAP services.

SOAP is a messaging protocol commonly built in top of HTTP (some old implementations relied in SMTP) which defines XML contracts for communication between two remote services.

The consumers discover all the available operations using a **Web Services Description Language** (**WSDL**) file. The WSDL file defines an XML structure for the incoming and outgoing data used by each operation.

The following is just a part of the WSDL definition file for an operation called **`sayHello`** which receives a String as input and returns another String as output:

```
…
<message name=”sayHello”>
    <part name=”parameters” element=”tns:sayHello”/>
</message>

<message name=”sayHelloResponse”>
    <part name=”parameters” element=”tns:sayHelloResponse”/>
</message>
```

```

<portType name="HelloWorld">
    <operation name="sayHello">
        <input wsam:Action="http://example.com/HelloService/sayHelloRequest" message="tns:sayHello"/>
        <output wsam:Action="http://example.com/HelloService/sayHelloResponse" message="tns:sayHelloResponse"/>
    </operation>
</portType>
…
```

In every request, the consumer will create an *XML envelope* (with a similar format to the XML described in the WSDL), fill it with the request information for each parameter in the request, and send it to the provider's endpoint.

SOAP shares many characteristics with gRPC: Both have a definition file which declares all methods, models, and payload types used by the API (proto3 for gRPC, WSDL for SOAP). Both have libraries which can generate code for clients and servers from their definition files.

However, unlike gRPC, SOAP doesn't serialize all requests and responses into binary data. It sends these bloated XML documents, which increase the network traffic size, making the communication "*chatty*".

Also, since WSDL files are difficult to build manually, developers usually build the models using the code in a specific language like Java, and then generate the WSDL from those same code models; then, they that same WSDL model to generate clients in other supported languages.

Due some of its advantages we will discuss next, REST slowly overtook the web service market away from SOAP. But mostly, people saw in REST a less complex, less chatty option for building remote APIs.

When would you use SOAP? Probably in legacy systems which need support. It's rare nowadays to have development teams choosing SOAP for building new projects, as most of the time REST, GraphQL or gRPC have a better performance than SOAP.

Building REST APIs

Representational state transfer (**REST**) is the most ubiquitous type of remote API at the time this book is written, and it has been for some years.

The use of REST is so common that many developers confuse the concept of API as a whole with REST. Many people use these concepts interchangeably when, as we’ve seen, they are not equal.

REST is a pattern built in top of HTTP to allow communication back and forth between servers and their clients; usually using JSON-formatted strings to serialize and deserialize data in traffic.

REST gained popularity in a time where developers were struggling with maintaining their SOAP services. Debugging, monitoring, and inspecting traffic in SOAP services was a headache, and the leaner, human-readable, JSON-based traffic seemed to be a better alternative.

REST exposes a list of resources. These resources are abstractions of real-life elements like orders or users, or concrete things, like products, cars, food, among others. These resources are represented as URLs exposed by the API server.

Then, we use elements of the HTTP specification to act on those resources. We use HTTP methods to describe and perform **actions** on our resources; and HTTP codes to describe the result of such actions.

By abstracting resources, we create a layer that separates the API from both providers and consumers. As long as they follow the REST patterns, consumers and providers can use their own naming conventions on the functions used to make and handle the API requests, reducing the coupling between them.

Resources can be defined as *models* of attributes. We can use JSON to express the model definition for a resource. For instance, a model for `Order` can look as follows:

```
Resource: Order, URI: /order, Model:
{
    id: “123124”,
    items: […],
    created_by: “John Doe”,
    created_on: “2009-06-15T13:45:30”,
    total: 34.12,
    …
}
```

By combining a resource URI with a HTTP method, the client can perform all the available actions for that given resource:

```
OPTIONS /order
GET /order
PUT /order
POST /order
PATCH /order
DELETE /order
```

Just as any HTTP request, each action in our API is defined as the function (**`HTTP_VERB ENTITY_URI) -> HTTP_STATUS`**. For instance, to use the **`Order`** API for creating a new order, a client would make a ‘PUT /order’ request. The API then would return one of a few possible statuses as the response:

- **`HTTP 201 Created`**: The order was created successfully.
- **`HTTP 500 Internal Server Error`**: There was an error creating the order.
- **`HTTP 405 Method not allowed`**: Creating orders is not a supported operation.

The following is a visual representation of the communication flow of the PUT /order request:

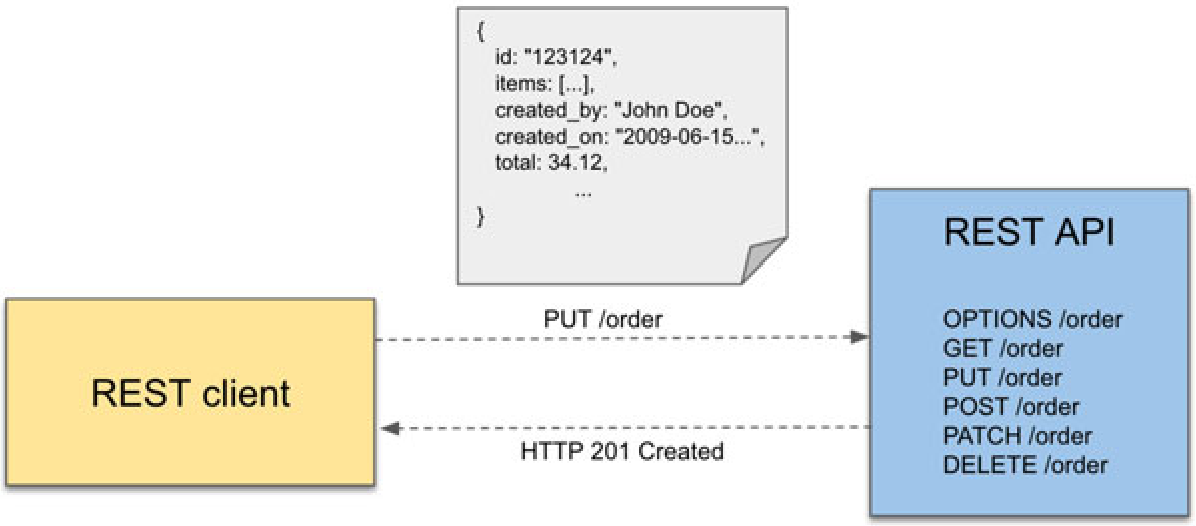


Figure 3.3: *A PUT request which successfully creates a new order*

Notice how we take the existing HTTP features and we repurpose them. We give them a new meaning based on our API's business logic. Response codes like `Created` have different meanings depending on which resources they're paired with.

What is the difference between REST and plain HTTP? The answer is "*not much*". REST is not a framework, nor a protocol. REST is just an architectural pattern that uses a large subset of HTTP to enable the integration between applications; and as such, it needs to follow the same principles of HTTP, like being stateless and cacheable. At the same time, REST can take advantage of the years of improvements we have built for regular HTTP requests.

Note: In the previous chapter, we discussed HTTP as being a stateless protocol, which means each request needs to include all the required information to understand the request, without relying on the web server to store a context.

Another HTTP principle is that requests need to explicitly declare if they can be cached or not -aiming to being cacheable as much as possible- as cacheable resources increase the performance of the HTTP service.

Actions versus HTTP request methods

One of the goals of REST is to express as much as possible using existing HTTP features, as an alternative of having to expose concrete functions.

For instance, instead of making a `POST` request to a `/deleteOrder` endpoint to remove an order, we would send a `DELETE` request to the `/order` endpoint. This simplification allows consumers to execute different actions using just one single endpoint.

The use of a single endpoint for multiple actions reduces the uncertainty for consumers. If they want to cancel an order, they don't have to guess if the function they need to call is named `/deleteOrder` or `/removeOrder`. As long as they know that the endpoint `/order` exists, they can assume that `DELTE` could be available (or use the verb OPTIONS to check what actions are available for the endpoint).

REST relies on HTTP methods to express intent over a resource:

- `OPTIONS`: Retrieve a list of the HTTP methods available for this resource.
- `GET`: Get one or many instances of the resource.
- `PUT/POST`: Create a new instance of the resource.
- `PATCH`: Partially update an existing instance of the resource.
- `DELETE`: Remove an existing instance of the resource.

Tip: Search term: idempotency

Idempotency means that an action will only take effect once, even if called multiple times. Calling an idempotent function multiple times should render the same results after the first call. The implementation of HTTP methods like OPTIONS, GET, and PUT are expected to be idempotent.

POST is a non-idempotent operation, which means that calling a POST method multiple times could potentially end up in multiple side-effects and different results in each call. Some literatures call non-idempotent methods "unsafe".

Idempotency is a critical term for HTTP. Clients assume that idempotent operations will behave as such, and they will allow cases were a request can be sent more than once, simultaneously.

This list of HTTP methods is enough to express most of the actions that can be done on a resource. Some other HTTP methods are less common, but can also be used by REST, if it makes sense for the specific use case.

Note: Both PUT and POST can be used for creating new instances of a resource, POST being most commonly used.

The difference between both is that PUT is assumed to be idempotent, so making the same request multiple times should result in just creating one single instance of the resource. POST doesn't have such limitations.

If the client is relying on the server to generate information such IDs or names, POST is a better option. If all the information is provided by the user in each request, PUT has a clearer intent. If you choose PUT, make sure that your implementation works as expected.

Choose whichever works best for you, just be consistent.

It's important to notice that these are the most common uses for each HTTP verb. Some APIs can choose to implement just a few of them, or do some slight variations. Always implement HTTP methods as defined by the HTTP specification, and return consistent and clear messages, especially for errors.

There are use cases where an API needs to expose an action that cannot be satisfactorily expressed using a combination of HTTP methods and a resource. In these cases, you can rely on **query parameters**.

Let's think of an example. If you want to fetch a list of orders, filtered by year, you can add a query parameter to rely this information to the `GET` action:

```
GET /order?year=2021
```

You can also express things like sorting by a specific attribute of the resource.

```
GET /order?orderBy=price
```

Query parameters offer a lot of flexibility, as you can use as many as you need and they are not tied to specific names or values. Because of this same flexibility, try not to over-use them or you will lose some of the benefits of

REST's standardization. For instance, the following are examples of not-so-optimal implementations:

```
POST /order?action=delete
POST /order?action=getMembershipAndOrder
```

Query parameters should extend the tools given by HTTP, not replace them.

Naming resources

Resources are at the center and front of the REST API design. Naming resources and their endpoints is a critical part of designing APIs.

Resources should ideally be a single noun like "*order*", "*membership*", or "*product*". As we discussed earlier, resource names should not include verbs like "*delete*", "*create*", or "*find*", otherwise you would be defining RESTful endpoints more suitable for web-based RPC.

Following an object-oriented perspective, resources can be seen as classes; blueprints that can have multiple instances. To identify and retrieve specific instances, you can attach an identifier to the URI, as shown here:

```
/membership/23
/order/123124
```

Keep in mind that these identifiers need to be encoded in the URL and clients will use these URLs to access the resources, so exceptionally long and complex identifiers might reduce the usability of your API.

Singular versus plural

A common question when naming resources and their URIs is: *Should you use singular or plural when naming your resources?* Should it be `/order` or `/orders`? The answer is determined directly by the type of actions you will perform with the resource and the level of expressiveness you want to define.

For instance, `GET /orders` has a different intent than `GET /order`. The first retrieves a list of orders while the second makes no sense if the first one exists. However, `DELETE /orders/123` may also make no sense, as `/orders` assumes it works with collections, not specific instances.

Instead of having to support both singular and plural endpoints, many developer teams choose to adopt a single endpoint for most of the actions.

It’s a chosen trade-off that brings simplicity without losing much expressiveness.

You can choose to use singular nouns for naming all your resources, as in the following example:

```
GET /order  ->  fetch a collection of orders
GET /order/123124  ->  fetch a single order
```

Relationships

Resources can be related to each, and we can design our API to express these relationships. Looking back to our *orders* example, multiple orders can be assigned to a *specific membership*.

This relationship should first be expressed in the model of the *order* resource itself:

```
{
    id: “123124”,
    membership: “23”,
    …
}
```

This follows a pattern similar to creating 1:N relationships in SQL entities. The child model contains a reference to a specific instance of the parent resource, which can then be used to retrieve the parent’s data.

We can name our resource’s URIs to also express this relationship by nesting the child entity into the parent’s URI:

```
/membership/23
/membership/23/order/123124
```

Notice how the URI makes explicit the relationship between both entities. You cannot retrieve an order that is not linked to a membership. On the downside, in addition to the order ID, you need to know an order’s membership ID in order to retrieve an order; that might not always be possible or optimal.

Actually, there are cases where relationships can be too complex to express using a single URI; think of resources which are deeply nested. In those cases, it's better to have separate, specific URIs for each resource and express the relationship only using the content of the resource's model:

```
GET /order/123124
Result:
{
    order_id: 123124,
    membership: 23
    …
}
```

While this approach might provide less context to consumers, it's a good trade-off to avoid having really long and complex URIs.

As seen in these past couple sections, REST conventions are there to make your life easier, not the opposite. It's important to be flexible enough to know when a good practice creates more problems than the ones it fixes, and it should be dropped.

Versioning

As flexible as a REST API can be, there will be a moment when it needs to evolve and make changes which can break the contract we have with existing consumers. Since breaking contracts is one of the worst things you can do with an API, we need to introduce breaking changes through *versioning*.

The most common way of versioning REST API is by adding the version information in the URI, like **`/v2/order/123124`**. If a new version of the API needs to be created, it's published to a new URI starting in **`/v3/`**, while keeping deployed an instance of the previous version **`v2`**. In that way, clients which are using the resources in **`/v2/`** will still have access to the previous version of the API, while at the same time clients who migrate to v3 can access the newer version.

The downside of this approach is that you have to create a whole branch of the API to support a new version: there is no way to upgrade only the resources which broke the contract. Most of the time, introducing new versions requires clients to change the URI in all instances of their code base, as there is no easy way to integrate new with older versions. Also, URIs keep getting long and complex.

The second approach is to introduce a custom header for clients to indicate the API which version they can support:

```
curl -H “Accept-Version: 2.0” http://www.example.com/api/order/123124

200 OK
Version: 2.0
```

The advantage of this approach over adding a version to the URI is that our URIs are clean. Clients will always make request to the same URIs.

Once clients upgrade their code to support all breaking changes, they can start sending a different value in the header. This also means that, resources which don’t need to be moved to a new version because weren’t changed don’t need to be exposed in a new URI. All they have to do is return their latest version if the requested version is equal or larger:

```
// Membership didn’t need to be updated, so its latest version is still 1.0, so the API returns that one, even if a larger version was requested

> curl -H “Accept-Version: 2.0” http://www.example.com/api/membership

200 OK
Version: 1.0
```

Requesting specific versions through custom headers allows clients a fine-grained control over each request’s version, and gives API designers the capability of only having to support old versions for resources which have changed, instead of the whole API.

On the downside, using custom headers is a slightly more complex approach and it requires better documentation for clients to know how it works.

Caching

REST takes advantage of HTTP's cache capabilities. Since **`GET`** requests are idempotent and are supposed to only fetch data, they are cacheable by default. **`POST`** requests can be cached but the server has to explicitly tell the client through response headers. Other requests like **`DELETE`** or **`PUT`**, while they are idempotent, they are supposed to mutate data in the server and their responses should not be cached.

All the caching techniques from HTTP apply out-of-the box to REST (as long as we're not doing something weird, like making every request a **`PUT`** request).

The server can indicate clients how to cache REST responses using multiple HTTP response headers:

- **`Expires`**: This indicates the date in which the response will not be valid anymore and the client will have to request it again. This date can be in formatted in seconds, minutes, or days in the future. Until then, the client can use the cached version.
- **`Cache-Control`**: This is a list of keys and values which describe how the resource should be cached. The key max-age indicates how long the resource should be cached. The value of the max-age overrides the value of Expires.
- **`ETag`**: This is a string which identifies a specific version of a response. As long as the content of a response doesn't change, it will have the same ETag value. If a resource expires in the client, it can send a small validation request to the server to check whether the **`ETag`** still has the same value. If the **`ETag`** hasn't changed, the client can still use the cached response.

Finding the right values for these headers depend completely on the business logic and the average time it takes for each resource to update and provide different responses for the same request. For resources whose response doesn't change often, you can set large values for **`max-age`** and future dates for **`Expires`**.

Effective REST APIs: HATEOAS

The greatest challenge of integrating two systems which may evolve at different paces is the lack of clarity in the integration point. Without flexibility baked into the design of an API, it's destined to fail.

From the point of view of API design, flexibility means that the API allows both the provider and consumers to change independently, without ever having to depend on each the other to make changes to keep growing. If a consumer needs to wait for a producer to implement a new specific action or resources, many times that's a sign that the API is not flexible.

The question of how to keep APIs flexible is independent of the technology you use. You can use SOAP services and still find ways to keep the API flexible. Flexibility can be defined with two concepts: *composability* and *discoverability*.

We discussed *discoverability* earlier. It's the capability of consumers to figure out exactly what can be done with the API without having to heavily rely on documentation or an external third party providing usage guidance. APIs should be built so clients can explore them as they use them.

Composability means that the provider exposes enough resources that the consumer can combine and build their own high-level actions without having to wait for the producer to expose that action itself.

Imagine you design an API to create configurations of cars for a carmaker. You can expose resources for **`/truck`**, **`/sedan`** or **`/suv`**; but if a new car category needs to be supported, the consumer would have to wait for the API designer to expose a new resource for that. However, if the API exposes resources at a slightly lower level like **`/wheels`**, **`/chasis`**, or **`/steering-wheel`**, the consumer can build almost any type of car they need to. Just be careful not to go to a level of abstraction that is too low, or the API will get too complex and unusable.

Inside the REST methodology, the ultimate way to reach both discoverability and composability is through HATEOAS.

Hypermedia as the Engine of Application State (**HATEOAS**) is a REST-based architecture in which clients discover actions related to a resource by reading metadata returned along the resource's content.

On websites, users navigate from one web page to another by using links contained in the body of those pages. In a HATEOAS-based API, consumers navigate through resources using URI references included in the response metadata.

Common ways of to implement HATEOAS are HAL, JSON-LD, and Collection+JSON. **Hypertext Application Language (HAL)** adds extra metadata in an attribute **`_links`** to the resource model. That attribute contains links to all the related resources. For instance, the model for our order resource can have the following structure:

```
{
    order_id: 123124,
    … // order data

    _links: {
        self: { href: "/order/123124" },
        parent: { href: "/membership/23" },
        next: { href: "/order/123138 }",
        find: { "href": "/order{?id}", "templated": true }
    }
    …
}
```

Notice how the metadata attribute **`_links`** allows clients to know that, in addition to the resource order which they are currently interacting with, there are other instances of **`orders`** which are related. There is also a reference to the membership of which this **`order`** is part, and a template for clients to find more orders.

Figure 3.4 shows how a graph or network is created by the links added to each resource response:

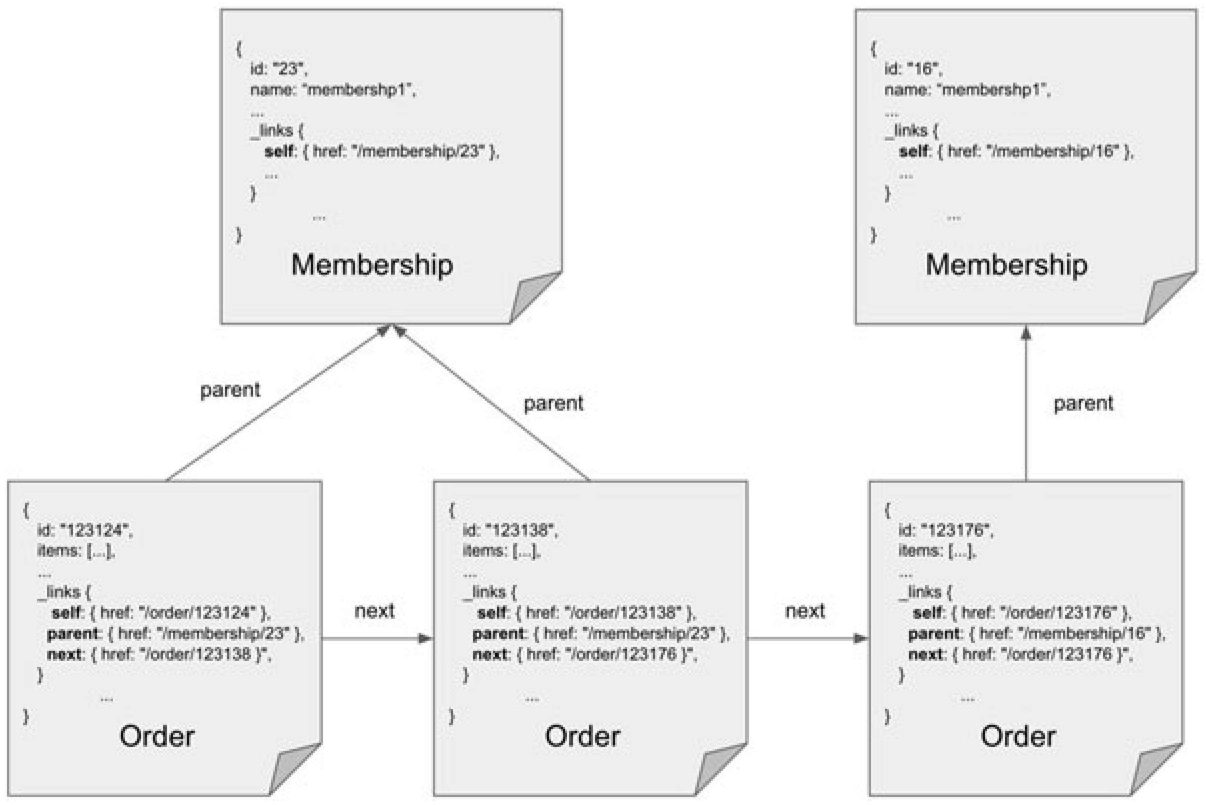

```
{
  id: "23",
  name: "membershp1",
  ...
  _links {
    self: { href: "/membership/23" },
    ...
  }
        ...
}
```
```
{
  id: "16",
  name: "membershp1",
  ...
  _links {
    self: { href: "/membership/16" },
    ...
  }
        ...
}
```
```
{
  id: "123124",
  items: [...],
  ...
  _links {
    self: { href: "/order/123124" },
   parent: { href: "/membership/23" },
   next: { href: "/order/123138 }",
  }
        ...
}
```
```
{
  id: "123138",
  items: [...],
  ...
  _links {
    self: { href: "/order/123138" },
   parent: { href: "/membership/23" },
   next: { href: "/order/123176 }",
  }
        ...
}
```
```
{
  id: "123176",
  items: [...],
  ...
  _links {
    self: { href: "/order/123176" },
   parent: { href: "/membership/16" },
   next: { href: "/order/123176 }",
  }
        ...
}
```

Figure 3.4: *A network of links using HAL*

HATEOAS is the pinnacle of REST APIs, and in practice very few REST APIs fully implement it. Many teams find that the effort required to enable HATEOAS is not worth the benefits it might bring to their specific business cases, and that's fine.

But, there are cases where HATEOAS can become incredibly helpful, like the cases of APIs built to be publicly exposed and directly accessed by external consumers. We will discuss these cases further in this chapter.

Building APIs with GraphQL

The flexibility provided by the composability of REST APIs also has its own challenges. In order to compose a semi-complex action, consumers need to make multiple requests to different endpoints. For instance, if you want to fetch the data of a **`membership`** and all its orders, you might have to do at least two requests; one for fetching the **`membership`** resource and one for batch fetching the list of orders related to it.

Having to make multiple requests for a single action has an impact in performance, especially in low-end mobile devices with expensive data rates and limited Internet bandwidth.

Using REST, you can support retrieving both the `membership` and `order` data in a single request by returning the membership data along with the list of all orders in `GET \membership\23\order`. However, this could force the consumers to always fetch the membership data, even if they don't need it. We can add a query parameter to opt-in to the extra data just in the cases that need it.

Unfortunately, supporting specific cases where it's critical to make as few requests as possible reduces a lot of the flexibility and expressiveness that REST provides.

Developers at Facebook, pressed to find a way to expose APIs that still allowed a high level of flexibility without forcing consumers to make too many requests, came out with **GraphQL**.

GraphQL is a framework built in top of HTTP that allows consumers to send "*queries*" to the API, as you would to a database like SQL. It starts by defining a *schema* which contains the definition of resources like `membership` and `order`. This schema can be written with the GraphQL **Schema Definition Language** (**SDL**) or defined programmatically.

The following is the GraphQL SDL model for both `Order` and `Membership`:

```
type Order {
    id: String!
    created_by: String
    created_on: String
    total: Float!
}

type Membership {
    id: String!
    name: String!
    orders: [Order]
}
```

Then, using the same definition language, you can define operations like **`Query`** to request data or **`Mutation`** to send update/create/delete requests:

```
type Query {  order(id: ID!): Order }


type Mutation {
  createOrder(membershipId: String!, total: Float!): Order!
}
```

Both queries and mutations require the API authors to create resolvers. Resolvers are implementations of the interface actions. They are in charge of fetching the requested data and return it back to the consumers.

Notice how GraphQL SDL follows a similar pattern than protobuf's **`proto3`**. These are languages which help you define the format for the data to be passed back and forth in the API, creating a clear contract.

Unlike gRPC, GraphQL defines its own abstraction layer. All operations are clearly defined and categorized in queries and mutations. In this aspect, GraphQL has a similar flexibility to that of REST.

One advantage of GraphQL over REST is that clients can request all the data they need in a single query request to the API. If they need to request both the membership and order data, a client can send a query as follows:

```
query {
  membership {
    id,
    name,
    orders{
      id,
      created_by
      created_on
      total
    }
  }
}
```

The response from the API could look like this:

```
{
  “data”: {
    “membership”: {
      “id”: 23,
      “name”: “membership1”,
      “orders”: [
        {
          “created_by”: “John Doe”,
          “created_on”: “2009-06-15T13:45:30”,
          “id”: 123124,
          “total”: 34.12
        }
      ]
    }
  }
}
```

We can also choose to fetch only a subset of all fields in each model. This reduces the amount of data that needs to be sent from the server to the client. Less data in traffic means better performance. For instance, you can fetch only the membership information:

```
query {
  membership {
    id,
    name
  }
}
```

Or include only the totals of each order attached to the membership:

```
query {
  membership {
    id,
    name,
    orders{
      total
    }
  }
}
```

We can optimize even further by making multiple queries in a single request:

```
query {
  membership {
    id,
    name
  },
  anotherQuery {
    field1,
    field2
  }
}
```

The response for that request will contain a JSON with the data for both queries. Potentially, you can request all the critical data on a web page by making just one single request.

GraphQL allows us to express multiple use cases without having to change the API implementation. We get the benefits of REST's composability without the performance hit of having to make multiple requests.

The level of expressiveness of GraphQL comes at the cost of complexity. GraphQL's abstraction layer requires more design work than REST. And the learning curve for GraphQL is steep compared with the other remote API methods.

Caching is another complication for GraphQL. REST can rely on HTTP's native caching strategies, while GraphQL's constant use of POST requests for queries makes responses non-cacheable by default. Extra layers of code need to be added on top of GraphQL to get the benefits of caching responses.

When to use GraphQL? Good candidates are applications with many resources with interconnected relationships. It's not a coincidence that GraphQL was built by Facebook developers: The social network works with large graphs of data, and GraphQL was created for those cases. Applications that need to optimize traffic for mobile devices are also good use cases.

Building standalone APIs

The adoption of REST and the continuous improvement of API design as led to API designers to directly expose their APIs to the public. While most of the use cases of remote APIs is to connect two applications built by the same team (or a team close to them), some APIs are built without a specific consumer in mind.

We can find at least two concrete examples of standalone APIs:

- Headless CMS systems
- Public APIs

Standalone API: Headless CMS

Headless **Content Management System** (**CMS**) is a system that directly exposes an API for consumers to manage their content data. CMS is used for managing digital or web content, like news sites or wiki pages.

Traditional CMS like WordPress ship with an integrated user interface. If users need to access the application, the CMS manager needs to provide a responsive web template or a plugin to expose some of the authoring features.

For headless CMS, consumers are expected to provide their own user interface, as the CMS only exposes an API. *Figure 3.5* highlights the differences between traditional CMS and headless CMS:

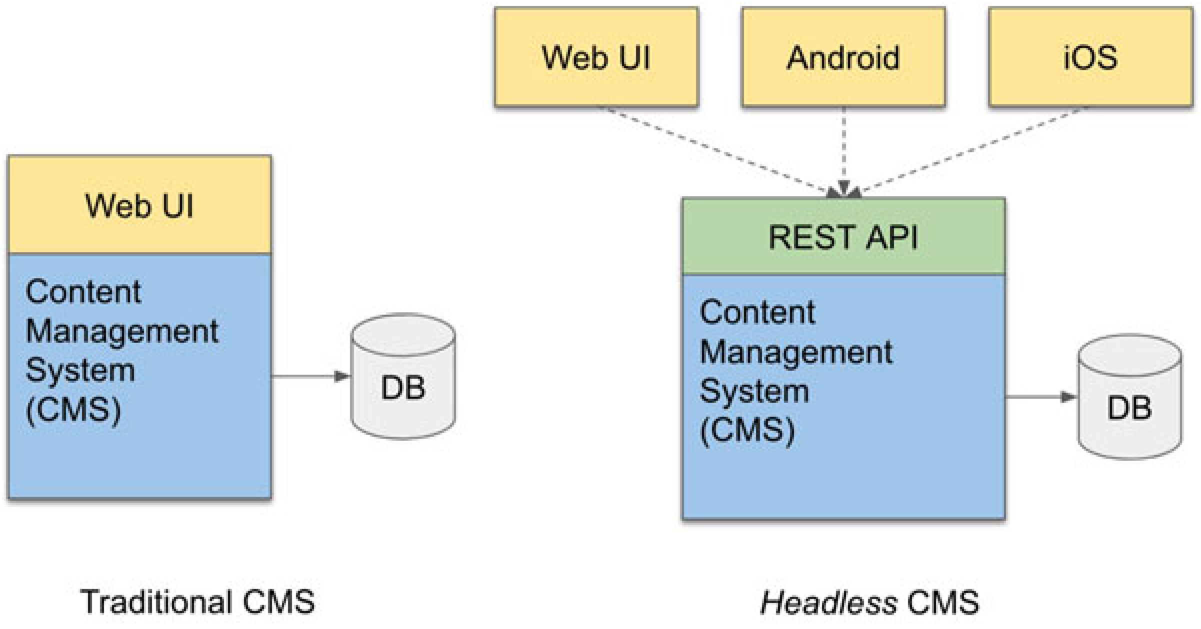


Figure 3.5: *Tradition vs headless CMS*

Some of the most popular headless CMS providers are *Ghost*, *Netflify CMS,* or *Strapi*; all of them open-source platforms.

Headless CMS are flexible. They allow you to use any tech stack you want for your frontend (WordPress users are tied to PHP, unless they do some workarounds). Their APIs even allow you to integrate CMS capabilities into the backend of an existing application.

API designers for CMS products don't have control over the type of applications which will consume their services, so the API needs to be built to provide a good user experience directly.

Standalone API: Public APIs

Some companies expose REST APIs publicly, so they can be used by third parties. These APIs follow a **Software-as-a-Service** (**SaaS**) approach, as in they provide specific functionality where, without having to install any piece of software, consumers can take advantage of the API either for free or by paying a subscription.

Public APIs can be seen as headless applications. Most of them have no user interface, or their authors only provide simple UI libraries focused to make integration with the API easier.

A good example of a public API is *Auth0*. *Auth0* is a SaaS service focused to provide application security services to their clients. They provide user management (or integration with other user management systems like AWS Cognito), and authentication or authorization through protocols like OAuth2.

Note: OAuth is a standard for applications to integrate with identity management systems to authenticate and authorize users.

In OAuth, a user authenticates using the identity server. Then, the server will return a user token, which contains the encoded information for the user. The application then uses this token to verify the user identity in each request.

Companies like Auth0, Google, or GitHub offer OAuth services that allow developers to provide authentication through these third-party sites. OAuth providers return information related to the logged-in user to the application.

Auth0 is a SaaS company which implements flows like log-in, log-out or giving roles to users, so you don't have to implement them yourself. All you have to do is register an account, and use a key provided by them to make requests to their publicly-exposed API. In addition to the API itself, Auth0 provides code libraries for UI frameworks like React or Angular, which help clients to integrate quickly.

In *Figure 3.6*, we can see the home page of Auth0, where you can start setting up an OAuth-based access control flow:

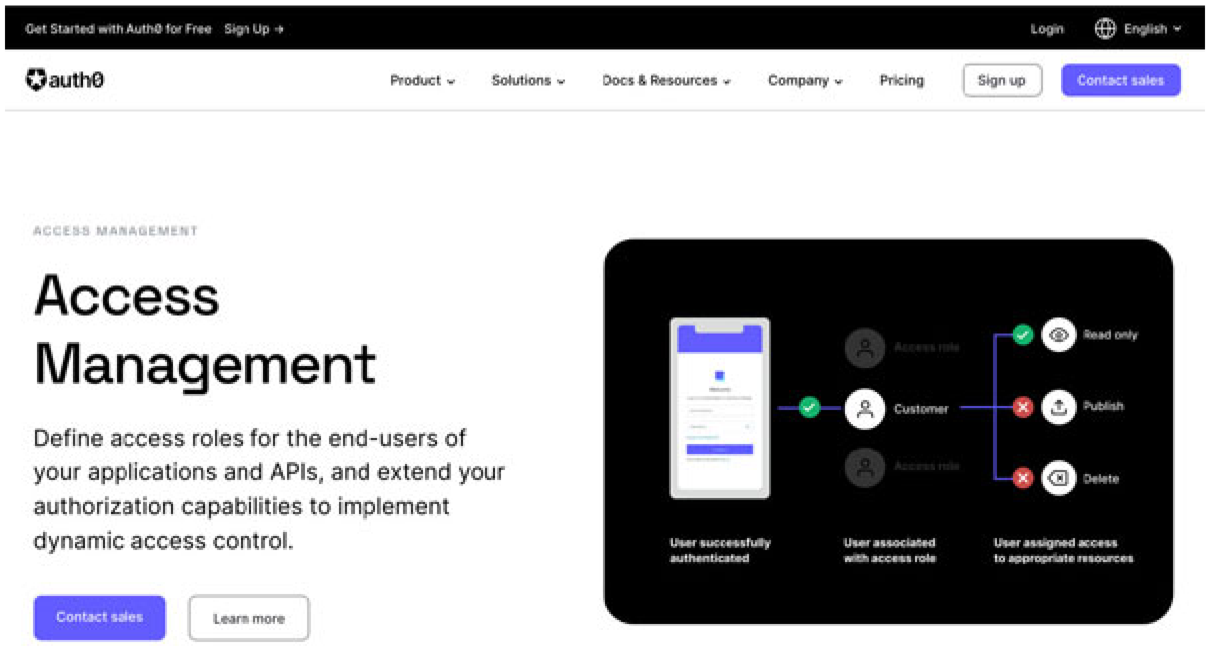

Get Started with Auth0 for Free Sign Up →
Login
English
auth0
Product
Solutions
Docs & Resources
Company
Pricing
Sign up
Contact sales
ACCESS MANAGEMENT
Access Management
Define access roles for the end-users of your applications and APIs, and extend your authorization capabilities to implement dynamic access control.
Contact sales
Learn more
Access role
Customer
Access role
Read only
Publish
Delete
User successfully authenticated
User associated with access role
User assigned access to appropriate resources

Figure 3.6: *Auth0's web page*

Just as Auth0, there is an increasingly large list of companies on the Internet who provides their services through public APIs. It's a business model worth of exploring.

Public APIs highlight the importance of creating good layers of abstraction in REST services. Just as in the case of headless CMS, designers of public APIs have no way to know which clients will consume their services, and have no control of how quickly they change. If the API is not clear, no client will want to use it, resulting in a loss in income.

Use case: Designing a remote API for the Pizza Corner

In this chapter, we discussed multiple ways to create remote APIs. For the Pizza Corner business case, we need to create an API which will allow us to decouple the UI from the backend business logic, so frontend and backend developers can work in parallel.

We will start by defining the contract for our API. This definition can be translated to any of the definition languages we've seen in this chapter, so we have the freedom to define the contract first and then choose the technology used to implement it.

Let’s start with the list of models which will define our resources:

```
// Models:

// model: Member
{
    ID: String,
    first_name: String,
    last_name: String,
    date_of_birth: String,
}

// model: Pizza
{
    ID: String,
    toppings: [String],
    instructions: String,
    cost: Float
}

// model: Order
{
    ID: String,
    pizzas: [Pizza],
    created_date: Date,
    ordered_by: Member,
    total_cost: Float
}
```

Now, let’s define the list of actions we need to support:

```
- See member information: (String memberId) -> Member
```

2. `- See menu: () -> [Pizza]`
3. `- See all orders for a member: (String memberId) -> Member, [Order]`
4. `- Order a pizza: (String memberID (optional), Order order) -> Integer status, Integer orderId`
5. `- Add pizza to menu: (Pizza) -> Integer status`

Looking at our list of actions and models, we can see that we don't have too many resources that need to be connected in each request. The action that requires the most interaction between resources is "*See all orders for a member*", which can involve fetching data for all three resources at most. For this reason, we can use either REST or GraphQL.

There are no strong reasons to choose GraphQL over REST, though. The amount of data that the API can return is small (even if a member has a lot of orders, we can always apply pagination) and we only have three resources. RPC can be discarded as well, as we have to support both web and mobile devices. REST integrates better with both, especially with web, so it's good enough for now.

In the previous chapter, we defined that for now we would host the backed in a single server. We don't have to worry about communication between multiple backend services, so we can save ourselves the effort of implementing gRPC. Once the application grows and we start breaking logical services into their own applications and servers, we can make use of gRPC to enable remote communication between them.

We define a REST API containing the following actions:

1. `– GET /member/:memberId -> HTTP 200: OK, Member`
2. `– GET /menu -> HTTP ٢٠٠: OK, [Pizza]`
3. `– GET /order?memberId=:memberId -> HTTP ٢٠٠: OK, [Order]`
4. `– PUT /order? memberId=:memberId (order info in the body) -> HTTP ٢٠١: Created, Integer orderId`
5. `– PUT /menu (food item info in the body) -> HTTP 201: Created`

Please notice that this is just one way to define our REST API. Other names and approaches can be used, as long as we follow the REST conventions.

Conclusion

The concept of API is agnostic of the technology used to implement it. API, as its acronym indicates, provides an interface that enables the seamless integration between two pieces of software.

An API is a contract that will be followed by the applications being integrated. This contract has to be clear, usable, and discoverable; otherwise, no one will want to consume it.

APIs can be local coding abstractions, as in Java or C# interfaces. They create a contract between classes which can be fulfilled by any implementation that follows the function signatures provided by the interface.

Interfaces allow applications to abstract their implementations, even to move them to remote servers. These implementations can be switched as needed.

The most common patterns used to implement remote APIs are SOAP, RPC, REST, and GraphQL.

RPC abstracts function calls by providing programmatic functions that seamless make requests to a remote server. Consumers call regular functions which internally invoke the remote function. RCP can be implemented using HTTP, like gRPC which relies on HTTP/2.

Consider using RPC if the applications you are integrating are supposed to interact as if they were part of the same application (strong coupling), or if the server and clients are expected to evolve at similar pace (e.g., when both are managed by the same team or close teams within the same company). Integration between microservices in distributed systems is a great place to use RPC instances like gRPC.

REST is currently the default implementation of remote APIs. It provides an abstraction layer that separates client and server, allowing them to evolve at different pace. REST takes full advantage of the HTTP specification, and relies on HTTP's features like caching.

Designing REST APIs involve making full use of HTTP methods and status codes. The combination of these elements with resource-based endpoints allows developers to discover and compose their own semi-transactional actions.

The ultimate implementation of REST is HATEOAS. HATEOAS follows the same web-like structure of the Internet, linking from one resource to another, and returning enough metadata for consumers to discover how to act and navigate through the API.

The majority of new remote APIs can be built with REST. REST provides a good balance between flexibly and performance. It allows you to define an API without knowing the client requirements beforehand.

For cases where network traffic is critical, like mobile applications whose targets are low-end devices, GraphQL can be a better choice than REST of RPC. Other good user cases are applications with a high number of entities or resources, with a lot of interconnections.

GraphQL allows clients to create "*queries*" on the models exposed by the API. Clients can build a request to fetch all the data they need in a single request, saving network time and data usage. On its downside, GraphQL is more complex to implement correctly.

Remote APIs can be published directly for third-party clients to consume them. These APIs enable "*headless*" (without an integrated user interface) products like CMS, or SaaS services like Auth0. Some companies can charge clients for direct use or their API, usually providing usage plans.

If you were to take only one thing out of this chapter is this: The important part of designing a programmatic interface is to create a contract which is clear and explicit to every component following it. It needs to define what are the constants and restrictions, and what kind of data we can expect.

Creating a clear API is the real hard part because once you do this, design tools like REST, gRPC or GraphQL are so mature that they even generate the code for you.

In this chapter, we hinted how interfaces abstract away services that use data stores to persist the data that is passed to them. In the next chapter, we will focus on these data tools, and we will learn some strategies to handle and store all the data our server is receiving through its API.

Questions

- You're planning to publish an API and sell access to its data through a subscription. Which kind of remote API (REST, GraphQL or gRPC) would you use for this? Why?

- Is it possible to create a REST API without using HTTP?
- Another team within your own company asks for your help while redesigning the API which integrates one of their services and a service you maintain. The business logic will remain the same. The service is currently using REST and they want to explore ways to improve the API's performance. Would you recommend them to stick to REST? Or would you recommend them to evaluate a different remote API tool like GraphQL or gRPC?

References

- C++ documentation of to_string
  https://www.cplusplus.com/reference/string/to_string/
- A great explanation of REST
  https://restfulapi.net/
- Golang tutorial
  https://tour.golang.org/
- The official documentation for GraphQL
  https://graphql.org/learn/
- Introduction to gRPC
  https://grpc.io/docs/what-is-grpc/introduction/

CHAPTER 4
End-to-End Data Management

The most significant asset in the modern world is data. The largest companies make millions in revenue by creating data-focused strategies: Finding the needs and characteristics of the current and future market, targeting demographics with their ads, using Machine Learning to predict ways to increase sales, and so on.

Because of its increasing importance, managing data correctly needs to be at the core of any software development. It should be a skill that every backend developer must possess not just specialized roles like **Database Administrators** (**DBAs**).

This chapter describes different ways of storing data, from simple in-memory data structures to specialized databases. We will understand what databases offer and what features help us deal with all the complexities of data handling.

Structure

In this chapter, we will cover the following topics:

- Defining the application state
- Understanding in-memory data storage
- Understanding complex data storage
- Indexing
- Backup and recovery
- Designing data storage in a production system
- Use Case: Defining a data model for the Pizza Place application

Objectives

By the end of this chapter, we should be familiar with the most common ways of storing and querying data. We will understand the differences between the most popular databases (like SQL, NoSQL databases, and all the categories that fall on the latter). We will also know in which cases specific databases work better than others.

We will also get a clear picture of how we can store data in software applications: the challenges and commonly applied solutions.

This chapter aims to make us feel comfortable working with other developers to design a data management strategy that makes sense for our specific use case.

Defining the application state

Every software application uses data. Arguably, there are no exceptions to this rule as the software itself is data we use to describe other types of information.

For instance, in the simplest case of a web application, a static website displays content to everyone who visits it. Even if the website's content never changes, it is still considered data. *Static* data.

As applications become more complex, like mobile or web apps, data becomes less static: It's constantly updated by the users and the application itself. Think of online retailers like Amazon. The list of available products in online stores keeps changing as providers add new items and users buy them.

Since data can change in time, we can think of it as the *application's state*. The state is the exact data that constructs the system at any given moment.

The state can also result from the status of the application itself. If the application is down due to a failure, we can say that at that moment, it's in an *error state* or *failure state*.

As software developers, it is our job to design how this state is structured and stored. Understanding how data is stored allows us to make informed decisions for our applications.

Hardware storage

At a low level, computers don’t store the application state by default. If we’re working in an application like a text processor or a spreadsheet, either you or the program itself needs to explicitly save the state of a document somewhere. If neither you nor the app ‘save’ your work, and for some reason, your computer runs out of power, the application state would be lost.

From a hardware point of view, volatile memory like **Random Access Memory** (**RAM**) will lose all the data it contains if the device loses its power. In general, most data in the RAM will be deleted in a short term unless it’s **persisted** somewhere. The reason **hard disk drives** (**HDDs**) exist is precisely to persist state in the long term, even after turning off the device.

A good question is: Why do we keep using RAM for storing in-memory data? Why not store it directly in HDDs? The answer is that long-term storage devices like HDDs are significantly slower to read and write than RAM. Hypothetically, moving all computation into HDDs would persist all state at all times, but it would also make our applications significantly slower.

So, computing devices need to balance different storage levels to achieve the best results.

Understanding in-memory data storage

The simplest way of storing software is through **in-memory** data structures. Among these are **variables**, **arrays**, **dictionaries** or **hash maps**, **lists**, and **queues**. All these data structures are stored in the RAM’s **heap**, as shown in the following *Figure 4.1*:

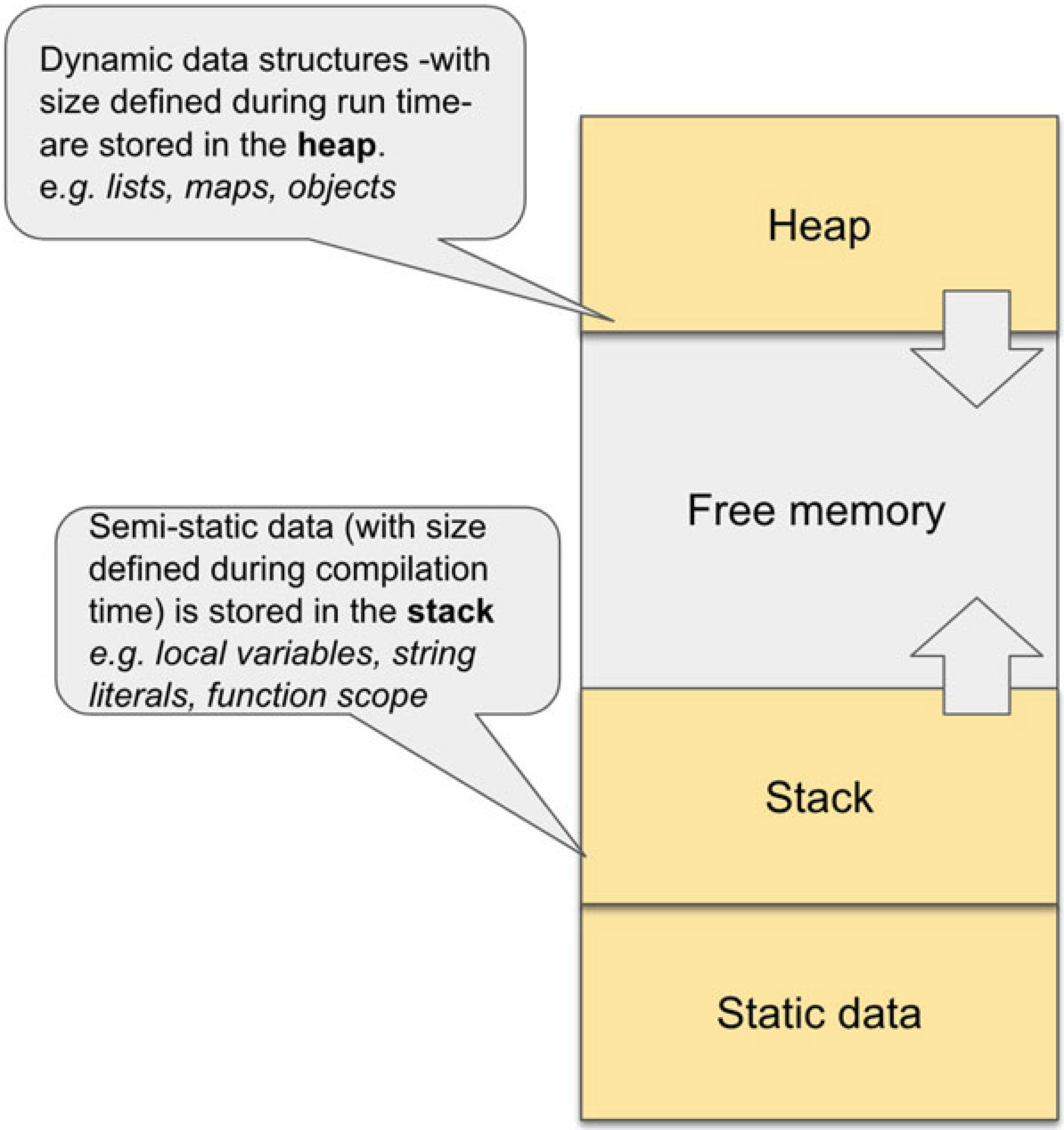

Dynamic data structures -with size defined during run time- are stored in the **heap**.
e.g. lists, maps, objects
Semi-static data (with size defined during compilation time) is stored in the **stack**
e.g. local variables, string literals, function scope
Heap
Free memory
Stack
Static data

Figure 4.1: *Data distribution in the RAM*

The heap is where all dynamically created data structures are stored. When C code calls **`malloc`**, or Java calls **`new Object()`**, it tells the application to reserve enough space in the heap to store our data structure. When you start an application, the program itself will be loaded into memory, into the section labeled with "Static data" (also seen in *Figure 4.1*).

In-memory data structures will only live as long as their context does: If they are declared in the global scope, they will live until the application ends. If they are declared inside a function, they will live until the function context is destroyed (most of the time, that moment is when the function returns or when all references to that context are cleared). Once the context of the data structure is destroyed, the heap memory used by the data structure will be marked as free memory for other processes to use.

Due to their short lifespans, in-memory data structures are not a great place to store things that need to live for a long time. If there is no secondary way to persist data, everything stored inside in-memory data structures will be deleted.

However, as we pointed out when discussing the differences between RAM and HDDs, in-*memory data structures are faster than disk-based solutions like databases*. This makes in-memory data structures great candidates to be used as a caching layer.

In-memory cache

Picture the example of the online store application we mentioned earlier. The application stores the list of all the products for a given vendor in a database. Since each vendor can have hundreds or thousands of products, querying that database can take a long time. Let us assume that each query to the products database takes 3 seconds.

Every time a user opens the online store they will have to wait at least three seconds every time they navigate to the catalog page. This is a bad user experience.

Now, let us also assume that we know vendors update their catalogs at most every hour, so all requests done withing that time window will always return the same results. We can rely on in-memory data structures (maybe using a hash-map) to create a cache with a TTL (Time to live) of one hour, and reduce our user’s page load time from seconds to milliseconds.

This caching layer would allow us to implement a couple of performance improvements:

- After the list of products is queried, we can store the results in the cache, using the query parameters as the key. If that same query is requested again within our TTL, we serve the results directly from the hash-map. Otherwise, we make a request to the database and update the cache, restarting its TTL.

- Select a list of the most popular queries site-wide and pre-store their results in the cache. This will improve page load times for a significant percentage of users without getting the initial performance impact of making a database query.

In this case, storing the application state inside in-memory data structures is fine because if data becomes stale (out-of-sync with the database) or if the application crashes and the data structures get cleared, we can always re-fetch all the data again from the persistent storage. Losing the data in the cache is not a catastrophic failure because it's not the main *source of truth*.

Tip Search term: source of truth

During the lifetime of an application, we can create multiple copies of a specific piece of data. Then, the application can modify each copy as it needs, most of the time due to a user action.

If one copy is modified with changes that conflict with the other copies (e.g., one attribute is updated in one copy, while it's deleted in the other), we can ask ourselves: Which copy is the "real" truth? While choosing the most-recently updated copy may sound like a good approach, it is not always the right solution.

However, when we pick one of those copies to be the primary copy, or the 'source of truth'. When in doubt between conflicting changes, we will often choose the contents of the source of truth above every other instance.

By choosing an instance of data to be the single source of truth, we can concentrate on keeping it updated and safe. If all other copies get corrupted or lost, we will still be in a good position because we can always use the source of truth to make more copies. However, if we lose our source of truth, we may not be able to trust any of the other copies because they might have been tampered in ways out of our control.

In the case of the in-memory cache example, the source of truth is the database. It's OK if the cache misses some updates for a while, but the source of truth should always receive all updates.

When building an in-memory cache, we have to be careful about a few things:

- For simple caches, updates will only be visible after the TTL - If vendors were to update their catalogs two or more times within the TTL (an hour in our example), users will only see the updated data after the TTL of the existing cache expires. Users need to be aware of this trade-off; the actual TTL value needs to be defined for the specific use case.
- We cannot store all the application state in memory - RAM tends to be smaller than HDDs. While RAM operates in gigabytes, HDDs can store terabytes or more. Because of this, we cannot just dump the whole database into the RAM. RAM storage is generally more expensive than HDD storage, so our cache size has to work around those constraints.

In-memory databases

If we build on the idea of the hash-map-based, in-memory cache, we can find *in-memory databases*.

An in-memory database is similar to in-memory data structures in that all data lives in the RAM. To keep operations, fast, each write, read, update, and delete is done to the data kept in memory.

In-memory databases add extra features to data structures to make themselves more useful and resilient. For instance, they can be used as embedded databases or installed as standalone servers. They expose APIs so clients can perform multiple operations on the data, from CRUD actions to advanced querying features.

Many in-memory databases regularly perform back-ups in persistent devices like the HDD to avoid losing information during failures. These backups are commonly one (or a mix) of the two:

- Dump a copy of the data on the disk. If the database crashes or the server goes down, it can recover its contents from the disk-stored snapshot.

- Keep an ordered log of each action performed to the database. When the database needs to recover after failure, it can *replay* the action log and recreate the data as it was until the last action was logged. We will discuss more about these logs further along in this chapter.

Popular in-memory databases like Redis, MemSQL, and VoltDB work as we just described. The high-performance that provides having all data stored in memory has led to a wide adoption of these products. For instance, multiple projects use Redis as:

- **Distributed cache**: This is the same idea as the in-memory cache we just described but in this case, the cache is deployed to its own servers. Even when deployed remotely, in-memory databases are faster than regular, disk-driven databases.
- **Distributed session management**: We discussed how HTTP is stateless. In some specific cases, we need to keep track of data from one request to the following; user's session data is one of those cases. Since user session data will be stored only for the duration of the session itself, a semi-persistent store like Redis is a good place to store this information.
- **Message broker**: Redis can serve as a middle point between multiple applications. It offers APIs that allow consumers to use it as a queue to enable asynchronous communication between two or more applications.

Simple storage in text and binary files

Let's take a step back from databases. What is the simplest way to persist an in-memory data structure without having to install anything or depend on third-party libraries? The simplest persistent storage method is to store data in files.

Files can be categorized into two: **text files** and **binary files**. While both types deal with a set of bytes, text files structure them as text characters. Binary files may contain text and non-text data, but they're not directly encoded to be human-readable.

The process of converting an in-memory data structure into data that can be stored in a file is called **serializing**. The opposite operation, parsing the contents of a file into a data structure, is called **deserializing**. In serialization, we convert in-memory data structures into bytes, text, JSON, or XML (or any other predefined format); then we can store -or even transmit through a network- the serialized data into text or binary files.

How do we serialize data structures? For binary files, it depends on what coding language you're using. For instance, in Java, a class can be serialized if it meets two conditions: it implements the Serializable interface, and all the class attributes are also serializable. Then, a serializable class can be serialized using **`ObjectOutputStream`**, as shown in the following code snippet:

```
Product product = new Product();
try {
    // Create file output stream
    FileOutputStream fileOutputStream = new FileOutputStream("productFile.bin");

    // Convert the stream into an object stream
    ObjectOutputStream objOutputStream = new ObjectOutputStream(fileOutputStream);

    // serialize the "product" object
    out.writeObject(product);

    // close the stream
    out.close();
    fileOut.close();
} catch (IOException i) { //… }
```

The file doesn't need to contain a ".bin" extension; this is just a random extension we chose for this example. The file might not even have a file extension at all.

We can build a complex data structure to store our application's state and create snapshots by serializing it to a file. Later, if we need to recover the state from the latest snapshot, we can read and deserialize that same file.

Data can also be stored in text-based files. CSV, XML, or JSON are, at their core, plain-text files whose contents are structured and formatted following a defined syntax. This syntax defines special rules that allow us to interpret the data these files store correctly.

Using binary or text files as persistent storage can be a good solution for some cases:

- **Text logs**: Logging events or user activity within a system can be done directly to a text file. This is data meant to be read by a human (especially error logs), so storing it in a text file gives us easy access and provides a clear intent of its use.
- **Backup**: Create snapshots of in-memory data structures -both simple and complex- and store them as files. We can retrieve and deserialize the content later, as needed.
- **Configuration data**: Some applications require configuration parameters for their execution: connection details, URLs of external services, business-specific parameters, and so on. If this configuration is meant to be handled by human administrators, serializing them in human-friendly formats allows administrators to modify its contents easily.

Simple file storage has its limitations, though. For starters, it’s difficult to manipulate the contents of a binary file without deserializing it first. If we want our data operations to be performant, we need to store all the information in small files to avoid dumping too much data into memory. We might need to split files as they grow beyond acceptable size.

Querying data stored in files is hard. How many files would you have to deserialize and check to find the list of data for a given user? How many files would we need to store the data of companies like Google or Twitter? As the volume of information we are handling and the frequency it changes grows, files alone become obsolete quickly.

Of course, we could build an application that manages application state using files in a performant way, offering a straightforward API, so users don’t have to worry about the internal implementation of this data storage.

The good thing is that those applications already exist, and they are called **databases**.

Understanding complex data storage

As the size and complexity of our data grows, the tools we use to store data need to be more advanced. As text or binary files become insufficient to contain our application's state, the next level of abstraction is to use databases.

A database is a type of specialized software whose primary goal is to allow users to store, update, and query data. They usually offer a clear querying language and offer multiple features like replication, backup management, and access control, among others. However, underneath its API, databases store data into text and binary files.

This section will discuss the types of databases we can use, and we will also understand how to choose from this list of options. Remember you should choose the database based on the data your application will handle. It is best not to choose the data storage tool first and then try to fit the data to work with it.

SQL or NoSQL?

Relational databases have been around since the 1970s. Since then, they have been the defacto database architecture used in most software development projects. Since its inception, multiple other types of databases have tried to replace relational databases, only to be relegated to niche use cases. To this date, the popularity of relational databases cannot be denied.

It's easy to see why developers adopted relational databases. These data stores abstracted away the implementation details of the data management operations so that developers could think of data as tables and relationships instead of files and bytes.

Relational databases also provided a declarative language to query the database: SQL. These queries allowed developers to describe *what* data they needed without specifying *how to* get it.

This level of abstraction allowed database providers to write multiple optimizations under the hood. As long as the developers kept using the API provided by SQL, the underlying implementation could use improvements like indexes and query optimizers.

Then, came NoSQL. More than a specific type of database, NoSQL is a category used to cover all the databases which don't follow a relational architecture. NoSQL databases were a response to the challenges developers were having while using relational databases. Some of those challenges were the following:

- Relational databases require rigidly defined schemas. While this might have advantages like easy validation and enforcement of a specific data structure, this comes at the cost of flexibility. You need to know your data's structure before you start collecting it, which is not always possible.
- Relational databases are difficult to scale. While database providers have improved in this aspect in recent years, this was one of the strong reasons NoSQL products gained popularity: They allowed developer teams to quickly scale without all the complex considerations that relational databases need.

Under the NoSQL umbrella, the most popular types of databases we have are *document databases*, *graph databases*, and *column databases*, among others. Each of these has very different implementations with specific and interesting solutions to data challenges. If you think about it, it's kind of unfair that they are all clustered in the same bucket.

So, which type of database should we use? In Machine Learning and Statistics exists the "No free lunch" theorem, which implies that there is no single algorithm that fixes all problems. Different problems require different solutions. The same is true for databases.

The main takeaway is that, just like a screwdriver doesn't replace a hammer, NoSQL databases don't replace SQL databases. They extend the catalog of specialized tools available to us.

Document databases

In document databases, we format our data into singular documents encoded with a format like JSON, Avro, or XML. All data related to an instance of a model is stored in the same document.

Similar to other types of databases, documents can be retrieved using keys. These keys allow us to fetch specific documents without searching the whole database for them. Keys can be *indexes*, which we will discuss later in this chapter.

Document databases can have relationships between documents, but database support for joins using foreign keys is not nearly as mature or supported as in relational databases. Related data is expected to be part of the same document, even if that means duplicating records.

Document databases are suitable for:

- Data with high locality
- Data with few relationships
- Unstructured data

High locality

Think of a straightforward model for the `Order` entity, composed of mostly primitive-valued attributes:

```
// Order
{
    id: 123,
    items: [
        {
            type: "PIZZA",
            ingredients: ["CHEESE"],
            quantity: 1,
            size: "XL",
            price: 12.99
        }
    ]
    total: 12.99
    notes: 'Extra crispy'
    date_created: "…"
16. }
```

Looking at the attributes, we could identify another hidden model: Each instance of the `items` list can be considered a separate model we can call `Item`.

By looking at the data, we can see that each `Item` will always be part of only one `Order: "Item"` has very specific values like "`size`" of "`XL`" and a "`quantity`" of `1`, both of which are tightly coupled to its particular order instance.

The relationship between items and order is so close that it makes sense to store them together: If we review the use cases and find that we will almost always retrieve an `Order` and its `items` together, then we know for sure that this model has a *high locality*.

Locality refers to how close data is stored to each other. Document databases offer high locality storage for each document, while databases like graph or relational databases have a reduced locality due to their models being distributed in tables and connected through relationships.

Most of the time, we will find high locality in 1:N relationships where the children entities can be thought of as being part of the parent entity instead of independent entities.

Data with high locality is a great candidate for document databases, but it can also present a challenge when querying data: Each read operation fetches whole documents only. For small documents, this might be OK, but as documents grow in size, you might have to fetch a lot of data you won't use, only because it's local to the data you *do* need. Some document databases have *some* support for querying nested attributes, but it's limited.

Few relationships

To keep a high locality in documents, it's essential to reduce the number of relationships between them. Data with multiple connections between its entities force document databases to do multiple queries to fetch all parts of a relationship.

A high number of relationships between data is one of the things that makes relational databases so difficult to scale horizontally: Related documents should be stored in the same database instance to avoid having to do join queries across multiple servers. Document databases don't have this problem, as most data is already contained in the same record, making it easy to scale horizontally.

Unstructured data

We might not know each attribute that our models should contain until we start collecting it. This is common at the beginning of a project when requirements change quickly and fields have to be iteratively added or removed from the data models.

Another case is where we don't have control of the data's format. The data we need to store might come from an external service that can change the attributes it returns without warning and without us having any control over it.

Document databases are great for all these use cases because we don't need to define a schema before we start saving data. We only need to have a schema to read the data. The schema defines the data format we want to use for the retrieved documents.

Of course, we can also store data that is well defined, but the main takeaway here is that document databases don't strictly enforce schemas, and we can add fields as we need; there is no need to rewrite our whole database or update each existing record like happens with relational databases.

Relational databases

Relational databases require us to break down our data into entities and their relationships. Instead of nesting one entity inside the other (like we did with Order and Item for the document database), we create two different entities and link them through relationships.

Traditionally, relational tables are flat data structures where each attribute or column is a primitive value. *Figure 4.2* shows how the relationship between Order and Item is mapped in a relational approach:

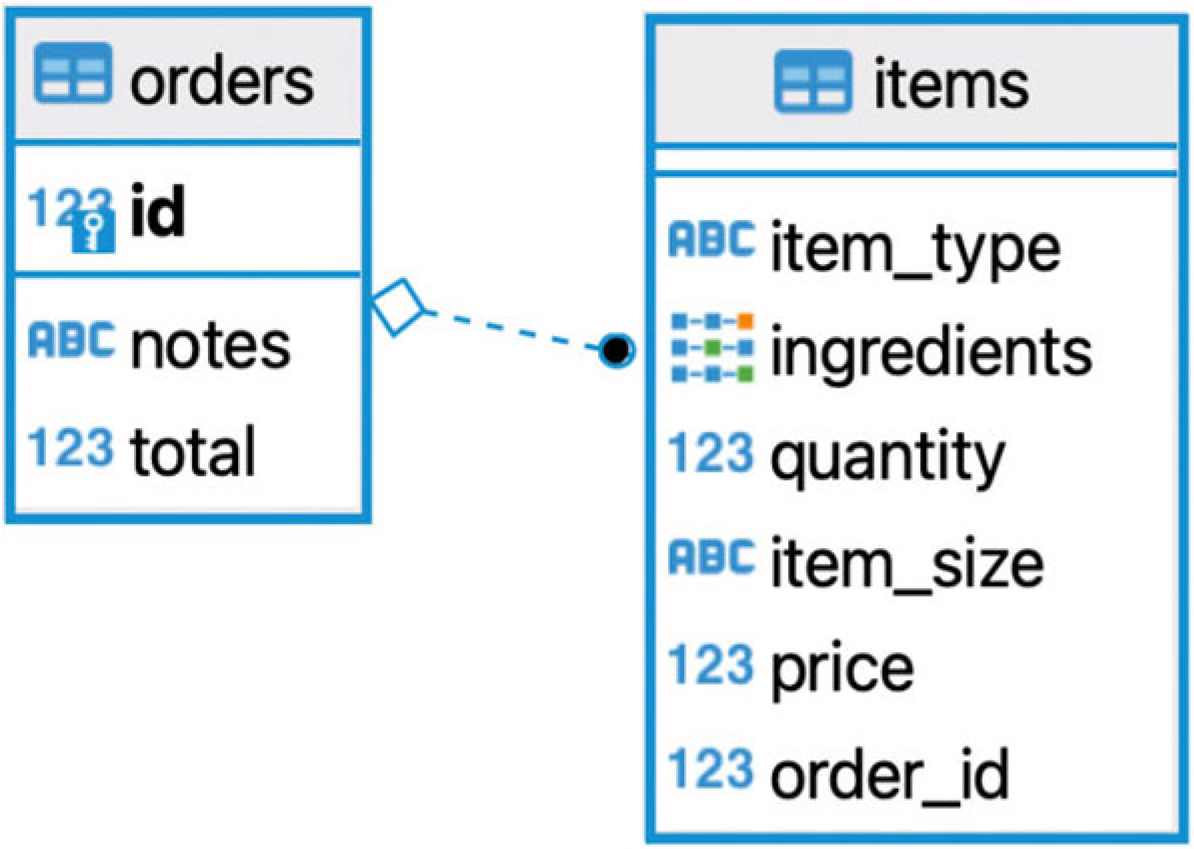


Figure 4.2: *Entity-relationship diagram for the Order and Item models*

If we need to retrieve an order and all their items, we will do a join operation between both tables, where the **`Items`** foreign key **`order_id`** matches the **`Orders`** column ID. The following is an example of such a query:

```
select o.id, i.item_type, i.ingredients, i.quantity,
  i.item_size, i.price, o.notes, o.total
      from Orders o join Items i
      on o.id = i.order_id;
```

Relational databases are a good choice for cases where:

- Models are independent of each other
- Data has multiple relationships between their models
- Data has a fixed structure

Note: While traditional relational databases use only flat tables and relationships, newer versions of some SQL databases allow us to store more complex structures, like arrays of strings, binary data, or JSON documents.

The addition of these features is a way to deal with the changing needs of many developers who want the benefits of high locality -which is characteristic in document databases- while at the same time keeping the query power of join operations in data with a high number of relationships.

Models are related but independent

If there exists a relationship between two or more models, but there are use cases where each model can be queried and used in isolation, we can say that they are independent of each other.

In *Chapter 3, Designing APIs*, we discussed two models, Order, and Membership. Even when all orders are linked to a membership, `Order` by itself has enough meaning to be used in isolation: We can look at a specific order or print lists of orders across multiple memberships. The same applies for `Membership`, as there are cases where we will want to fetch a membership without having to also query its orders.

When models are independent of each other, it’s challenging to store them in nested structures (like JSON documents) because there will be use cases where we want to query one model but not the other. This separation is part of the essence of relational databases.

Multiple relationships

If our data contains multiple entities related to other entities, we have even stronger reasons to choose a relational database.

Think of models that have N:M relationships like **Customer** and `Order`, `Provider`, and `Product`. There is no easy way to encode N:M relationships in document databases. We would be forced to create separate document tables and find a way to represent relationships between them. And that is something at which relational databases do a better job.

Fixed structure

Relational databases require us to define each attribute along with its data type before we can store any records in them. If we need to update our schema to add or remove columns, then we have to update every existing record in that table. This is a stark contrast from document databases were records in the same table can have different attributes.

Because of this characteristic, we prefer data with the following characteristics:

- We know all its attributes in advance.
- The data attributes don't change often.

An example of data with a fixed structure can be historical data, like metrics we took from past events. We can deduce the structure of any data record that has already been collected, and we can assume the rest of the records will follow that same format.

Normalization

In the context of relational databases, normalization is a process through which we structure (or restructure) our tables, so we minimize the amount of repeated data.

Normalization is a concept very close to that of having a *single source of truth*: We need to have just one "main" copy of the data we store. If we need to create variations of that single copy, or if we need to mix it with other entities, we do so when we read them using queries (or concepts like 'views' or 'stored procedures').

For instance, imagine two independent entities **`Videogame`** and **`Person`**, which have a many-to-many relationship with each other (for example, one videogame can be bought by multiple people, and one person can own multiple videogames).

If we need to store this relationship in a document database, we would have lots of repeated data. For instance, the pair of hypothetical gamers, **`Mario`** and **`Luigi`**, can own the same videogame "**`The pipe game`**":

```
[
    {
        personId: 123,
        firstName: “Mario”,
        gamesOwned: {
            [
                gameId: 321,
                gameTitle: “The pipe game”,
                //…
            ]
        }
    },
    {
        personId: 122,
        firstName: “Luigi”,
        gamesOwned: {
            [
                gameId: 321,
                gameTitle: “The pipe game”,
                //…
            ]
        }
    },
]
```

In this case, **`gameTitle`** (along the rest of the attributes for that videogame entity) is repeated for both users. If the game manufacturer decides to change the title to something like “**`The pipe game: Remastered`**”, we would have to find all the people who own this game and change the title attribute in each nested record. The use of repeated data is referred to as being denormalized.

In a relational database, we could normalize this data by creating a separate table for **`Game`**, and store all the people who own games through the table **`owned_by`**:

```sql
create table Person (
    personId serial primary key,
    firstName text
);

create table Game (
    gameId serial primary key,
    gamteTitle text
);

create table owned_by (
    personId serial references Person(personId),
    gameId serial references Game(personId)
);

select p.firstName, g.gameTitle from Person p
    join owned_by ob  on p.personId = ob.personId
    join Game g       on g.gameId = ob.gameId;
```

We represent ownership with a record in **`owned_by`** that references to the primary key of both **`Person`** and **`Game`**. We relate each person and the games they own through a join query.

Since they are normalized, both **`Person`** and **`Game`** can be updated independently and only once per record. If we need to change the title of "**`The pipe game`**", we need to only update its record in **`Game`**. Both **`owned_by`** and **`Person`** remain unchanged, as there is just a single copy of the game attributes, and all relationships are abstracted through the use of primary and foreign keys.

If you have decided to use a document database, one or two denormalized records are not the end of the world. But as the number of denormalized data increases, it might be worth to consider migrating to a relational database.

There is a set of defined rules on how to introduce normalization in a denormalized database. These rules are out of the scope of this book and more suited for a specialized book about SQL.

Graph databases

As the number of connections between the models in our data grows, modeling them in relational databases becomes difficult (and becomes almost impossible for document databases). As the data grows from 1:N or N:M dimensions to something like N:M:O dimensions -multiple links in a single many-to-many relationship-, and the data structure become a set of interconnected entities; *graph databases* become a great way to store it.

Graph databases are used extensively by social networking companies like Facebook, as they are especially good for capturing highly-interconnected data like social graphs. Some of the most popular graph databases are Neo4J, Amazon Neptune, ArangoDB, and RedisGraph.

Graph databases are great for:

- Data with a lot of connections
- Data where the relationships are first-class citizens

Data with a lot of connections

In real life, entities can form relationships at any time, relationships that weren't there before. As our models and our understanding of them evolve, we need the flexibility to design these new relationships after we define our initial schema. Graph databases give us this flexibility.

For instance, think of an entity **`Person`**. If we try to model a system that captures the people working at a movie production, the relationship between **`Person`** and **`Movie`** can be seen as **`ACTED_IN`**, if the person is an actor. However, a **`Person`** can act and produce a movie. In fact, a **`Person`** can be related to a **`movie`** in a multitude of ways, as shown in the following *Figure 4.3*:

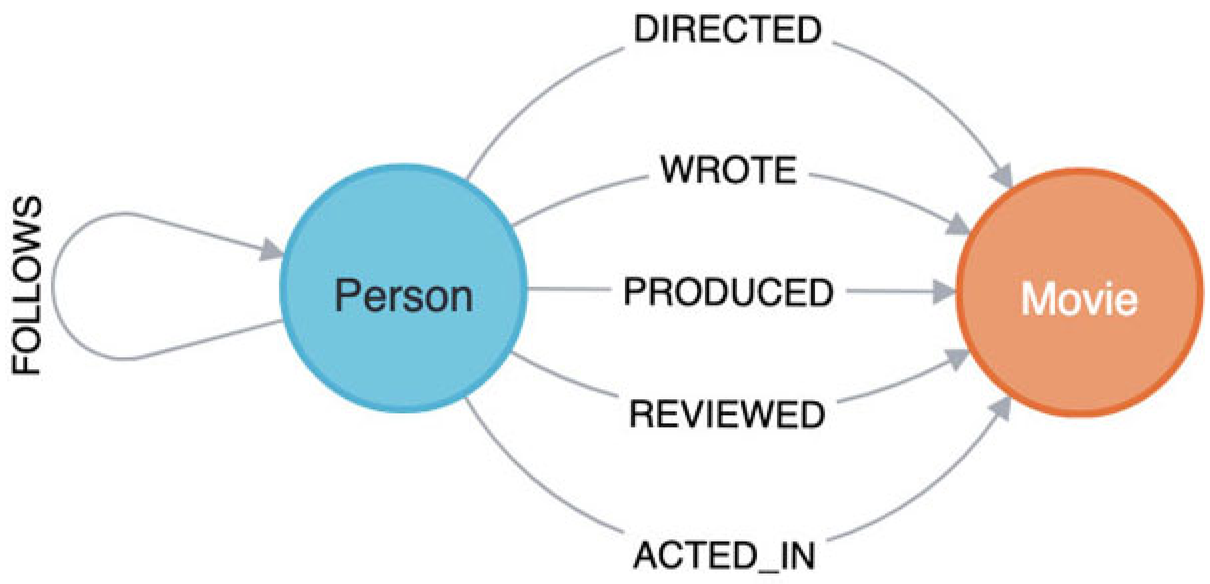


Figure 4.3: *Relationships between the Persona and Movie entities. (Image from Neo4J website)*

As seen from the preceding example, two entities can have more than one type of relationship between them. We could model this in a relational database, but we would have to expand our design beyond just **`Person`** and **`Movie`** to introduce extra entities (like the **`PersonInMovie`** join table) just to try to capture this use case. *Figure 4.4* shows the entity-relationship model needed for this example:

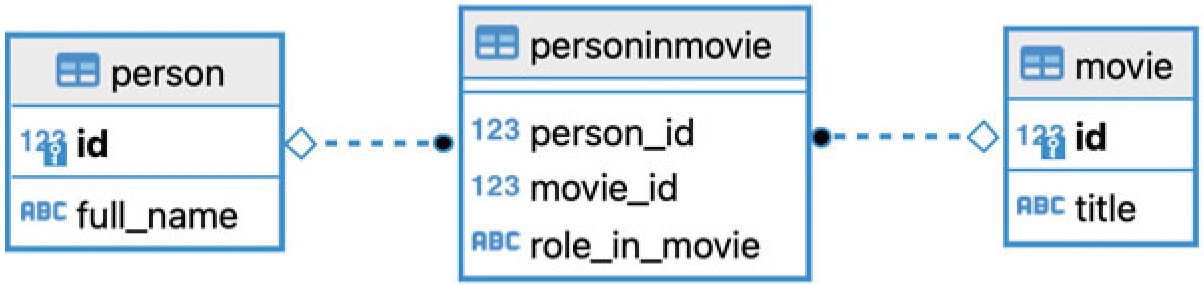


Figure 4.4: *Relational database design for the Person-Movie relationship)*

For this example, the problem is not to have to introduce an extra table to express this complex relationship. The problem is the complexity added by these extra entities grows linearly as our data grows in size and dimension.

Use cases like these fit more naturally in graph databases where complex relationships don't need to be expressed as entities themselves.

Data where the relationships are first-class citizens

In relational databases, the main way to express relationships is by applying the **`join`** operations in foreign keys. These connections are abstract in purpose: Using only foreign keys (and maybe join tables), we can express any kind of relationship. This abstraction comes at the cost of extra complexity which is captured in the length of queries used for this relationship.

For instance, for the simple case of finding all instances of **`Person`** that have a relationship of **`DIRECTED`** to a **`Movie`**, we need to write the following SQL query for relational databases:

```
select p.full_name, m.title, pm.role_in_movie from Movie m
    join PersonInMovie pm on m.id = pm.movie_id
    join Person p on p.id = pm.person_id
    where pm.role_in_movie = 'DIRECTED';
```

The intent of the query is obscured by the hoops we have to jump in order to stitch the data we need together. Now, look at that same query using Neo4J's query language, **Cypher**:

```
MATCH (p:Person)-[d:DIRECTED]-(m:Movie) RETURN p,d,m
```

The main difference between relational databases and graph databases while capturing relationships is that *relational databases encode the relationship as data, while graph databases encode the relationship as part of the database definition*. That is the reason why Neo4J's query language is so expressive: relationships with attributes are first-class citizens to it.

The takeaway here is not to say that relational databases can't store graph data. In some cases, they might do it very efficiently. The point is that we have to do less workarounds storing and querying this data in a graph database than with relational databases.

Scalability

A common point used by system designers to choose a database is scalability. Many people default to using document databases assuming that data will always need to scale horizontally. Many times, this is an overestimation.

Note: Horizontal versus vertical scalability

As the data grows, we also need to increase the database capabilities to meet with this added demand.

Scaling vertically means adding more processing power or memory to the single server hosting the database. This process is almost transparent to any database, and basically no design changes are needed.

Scaling horizontally means distributing the database to multiple low or mid-range servers. As the demand increases, we add more servers to meet it. This process requires more consideration than vertical scaling.

Scaling horizontally is cheaper than scaling vertically. Equipment like high-capacity memory and processors is expensive and covers only a part of the demand which multiple, smaller, servers can cover.

However, for some cases scaling vertically is enough to cover the data demand without having to deal with the complexity of managing a distributed database.

There is a common understanding that relational databases do not scale horizontally well. While there is some truth to this (`join` operations across servers are costly due to network latency, so the database has to be designed to keep most related data in the same server instance), it's a mistake to always choose NoSQL databases for this reason alone.

Relational databases that are scaled vertically still allow some large applications to grow considerably, especially if the database is well designed and fully implements optimizations like indexes.

File storage repositories

Some use cases require us to store binary files like pictures or videos, or text files like resumes, forms, contracts, and so on. Some of these files might be stored as additional data to that which is stored in a separate database (for example, profile pictures for user profiles), or they might need to be processed in order to extract information from them (like image recognition, natural language processing, among others).

Relational databases provide special column types (like BLOB) that allow us to store binary data in them. Just like the JSON-formatted attributes, these kinds of features were introduced as additions to the relational database model.

Relational databases are not the most natural option for storing binary files, as there are some extra downsides:

- The space required by files might complicate some relational database operations like backups or replication.
- File exploration is opaque in relational databases. There is no easy way to preview files stored in a relational database without workarounds.
- Extra processing needs to be done to the files in order to create publicly accessible links.

Again, it's perfectly possible to store large sets of files in relational databases, but we would be missing many of the benefits of *file storage repositories*.

File storage repositories are specialized data stores that focus on storing, organizing, and even querying files. The most common file repositories are AWS S3, Azure Blob, and Ambry, and these are just a few examples.

File repositories offer 'buckets', which are similar to directories in traditional file systems. These buckets are analogous to database tables.

Some of the features which file storage repositories provide on top of storage are:

- Easy exploration
- These products usually have user interfaces that allow developers and users to preview and search the stored files
- Access control
- Limit who can see, edit, or download the files
- Public link generation
- Create links for clients to download the files when they need them
- Query systems can be built on top of file repositories

For instance, Amazon Athena allows you to query and analyze data stored in AWS S3, without requiring developers to build code to parse, store, and query the files.

The heuristics for when and how to use file storage repositories are:

- If files are a central part of the business logic of your application, use a file repository system to save yourself the work of handling file management.
- If processing and/or creating a file (like a report) takes time, instead of building it when the user requests it, create it in advance and store it in the file repository.
- If the file upload rate is high (e.g., many users uploading multiple files at the same time), directly store them in the file repository and process them asynchronously. This will avoid any bottlenecks caused by the throughput limit on your file processing services.
- If you need to "parse" the files to extract information of them, it's a good practice to store the file in the file storage repository, parse it and store the extracted data in your database of choice (SQL, MongoDB, and so on) along with a reference to the original file.

Beyond technical requirements

When we start working on the design for a database, there are other points to consider beyond the technical characteristics of the application itself.

The biggest reason behind why so many applications keep using relational databases instead of more specialized storage is that there is plenty of talent available. Most software developers are trained in SQL, so finding developers for a new project is cheaper than finding someone with experience on, let's say, graph databases.

Sure, you can always hire SQL developers and train them to use MongoDB or Neo4J, but the truth is that you will need someone with experience who can fix a potentially critical bug in production that might affect all of your users.

Another constraint while choosing which database to use is the existing infrastructure. If your clients have already spent millions in creating an infrastructure around relational databases (licenses, optimizations, scaling, and so on), they will hardly want to spend some more money -even if it's a delectable amount- and discard all of their investments.

While "*we have always done it like this*" is not enough excuse to not try to choose the best storage option for your use case, sometimes we have to work with the material we have.

The best we can do is create an analysis to justify a decision that might not be practical at first glance. Many large re-design and re-factoring projects have been born out of finding during analysis that the investment will be worth it in the long term.

Indexing

Indexing is one of the most critical aspects of building data storage. A database without indexes is almost certain to struggle with slow queries. We will dedicate the following section to talk about indexes and how to leverage them to guarantee our database can perform up to our users' expectations.

Reducing time complexity

Let's say we have an unsorted, zero-indexed, list of records that contain a million of **`Product`** elements. Individual elements in the list can be accessed by their index:

```
[
    // record at index 0:
    {
        title: "Product01239",
        price: 499.00,
        …
    },
    // record at index 1:
    {
        title: "Product34464",
        price: 120.00,
            …
    },
    // ١000,000 records in total
]
```

We want to find a product with a specific value, let's say $120.00. Since this is an unsorted collection, we would have to do a linear search in order to find the record that matches the expected price.

Linear searches have a **time complexity** of *O(n)*, which means that the search time increases linearly as the number of elements in the list grow. If checking each record took half a second, then searching for an element in our example would take at worst *~128 hours*. How can we improve this?

We can build an **index**: A list of tuples (**`price`**, **`index`**), where the first element is the price of each record, and index is the index value of the record in the original list. This list is sorted by price and then by index.

What this sorted index allows us to do is to perform a **binary search** on the price attribute. The result of this search would give us the tuple with the expected price and the index pointing to the matched product in the first list.

Binary search has a time complexity of *O(logN)* which means that, for a million records, the search would do at most ~20 look-ups. Using the same time assumptions of half a second per lookup -as in the linear search example- this would take at worst 10 seconds (way less than the hours it may take doing a linear search). Of course, the magnitude used for the average time per look-up in this example is way larger than it would be in practice, but it illustrates the difference in performance that having an index can make.

At a high level, this is what indexes do. *Indexes pre-calculate and store data structures on which queries can operate faster*. Queries that use the indexed attribute will take less time, at the cost of the extra storage used by the index.

In database engines, not all indexes do a binary search. They use complex data structures like **B-trees** which also allow queries to be performed without having to scan for every record in the table.

Example: Benchmark the impact of indexes in SQLite

The best way to evaluate the impact of an index in a database is to run a benchmark against it: Measure the performance of a query searching for a record that matches a specific condition. We will use SQLite, as it doesn't require us to install any database server.

We can create two simple tables in SQLite. They will be identical except for the table name, and the fact that one will have an index on the **`price`** attribute while the other won't:

```
CREATE TABLE products (title text, price real)

CREATE TABLE indexed_products (title text, price real)
CREATE INDEX price_index ON indexed_products(price)
```

Then, we will insert 100,000 records with random values for each attribute. We need to see how the query performance changes as the number of records grow so, every 1000 insertions we will run a query against each table and measure the time they take to complete:

```
select * from products p where p.price = 100;
select * from indexed_products p where p.price = 100;
```

To better visualize their differences, we create a graph with each measurement in milliseconds. *Figure 4.5* shows the results of the comparison:

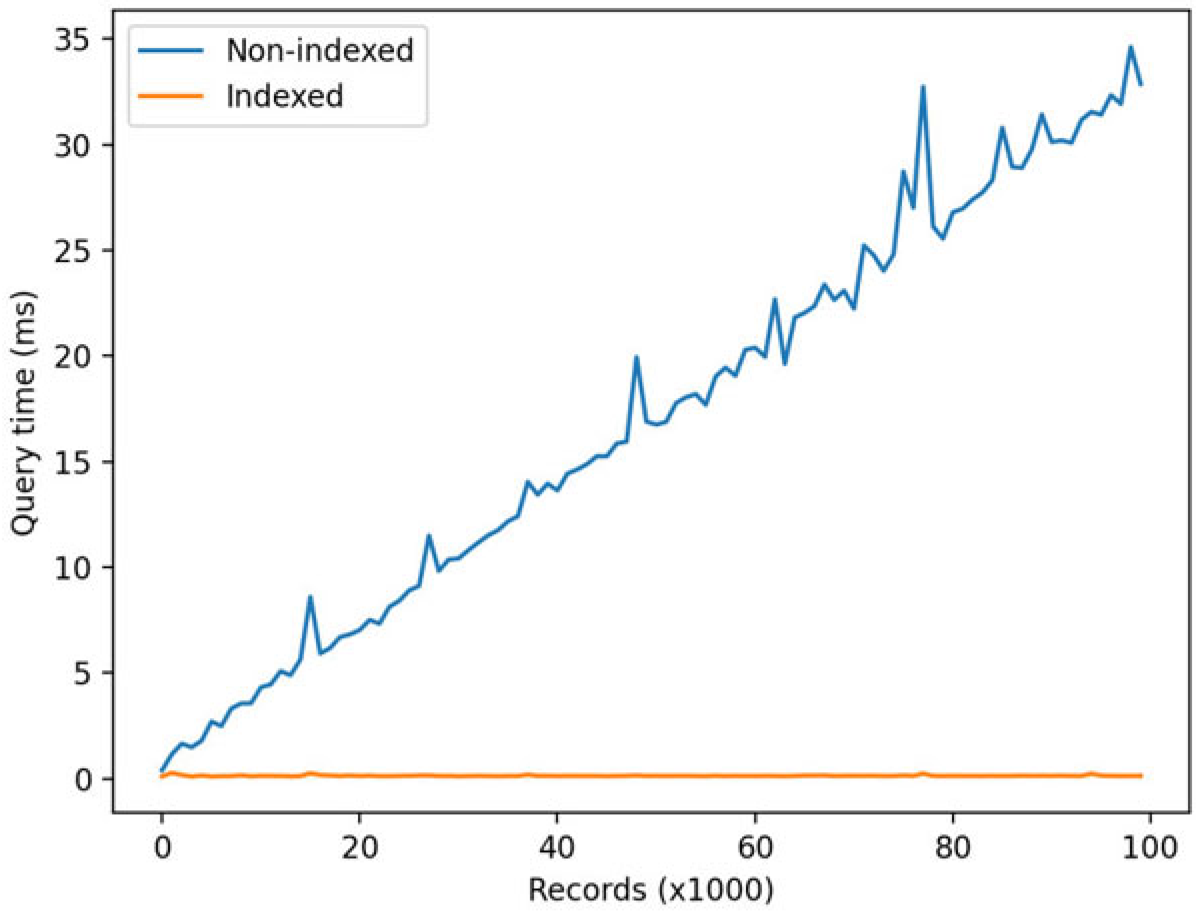


***Figure 4.5:** Relational database design for the Person-Movie relationship)*

We can see that the query time for the non-indexed table kept growing linearly as the number of records increased. On the other side, the graph for the indexed table seems to be flat. It's only after you zoom into it that we can see that the graph for the indexed table is also growing, but at a considerably slower rate.

In the context of modern applications, a million records it's not too much data. This little experiment makes it clear that querying over specific attributes that are not indexed can be extremely un-performant.

Backup and recovery

There are many bad things that can happen to our data: We can delete our database by mistake, hackers can compromise the integrity of our data, or a server itself can break down. Even whole data centers can fail during events like natural disasters. It’s not a matter of if our database will fail, but it’s a matter of when and most importantly, what will we do about it?

Every database instance should periodically create backups. Backups are temporal snapshots of our data: Every certain amount of time we make a full copy of our data and store it somewhere. In case of irreparable database failures, we can fetch the latest snapshot and rebuild the content of the database *up until the point in time when the backup was created.*

How often should we be backing up our database? In a perfect world scenario, we can create a backup every time the database is updated (actually, in a perfect world scenario we wouldn’t need backups at all). In real life, databases can be large enough that if we need to create a full backup for every update we would be getting a newer update before we even completed the previous copy.

The time we configure between backups depends on the characteristics of the data we’re storing. In theory, how much data would you afford to lose? The last day of updates? The last hour? If your data is mostly read-only and you only have one or two updates daily, then maybe it is reasonable to create a backup for every update.

If your application will be getting tens of updates per second, then you need to plan your backup strategy in a better way. There are multiple techniques to create database backups and, as we will see by the end of this section, we can mix a couple of them to fill the gaps.

Backup database files

At the lowest level, databases store data in a file system. We can create a backup by copying these files into a separate location. We must be careful, though. The database might be in the middle of an update or a transaction, and copying its files might end up in a half-baked state. In PostgreSQL, this is method is called “*File system level backup*”, and it’s advised you stop your database before backing up to avoid creating a corrupted copy.

Obviously, turning off a database for backup is not always possible, especially if that’s the only database server you have; it would force your application to be offline for some time.

Even with its downsides, some applications choose to follow this technique. It may be because it’s really simple (just copy the whole data directory to a different server and you’re done), or it might be because they are running an old version of a database that doesn’t support any other kind of backups. Development teams who have to follow this path choose times with low user traffic to pause the application for “*maintenance*”.

Creating backups with activity logs

Most modern databases keep a log of each update operation applied to its contents. These logs can be used to monitor activity or to diagnose the cause of a problem. They can also be used to re-create the contents of the database if needed.

Imagine a list where we append each update operation ever done to the database, in order of execution. We can take this list and re-apply each operation if we need to recover the state the database was at when the activity log was last updated.

Figure 4.6 shows how each update was done to a database is first applied to the activity log and then it is actually applied to the data storage. If the second insertion fails, the database can retry using the activity log:

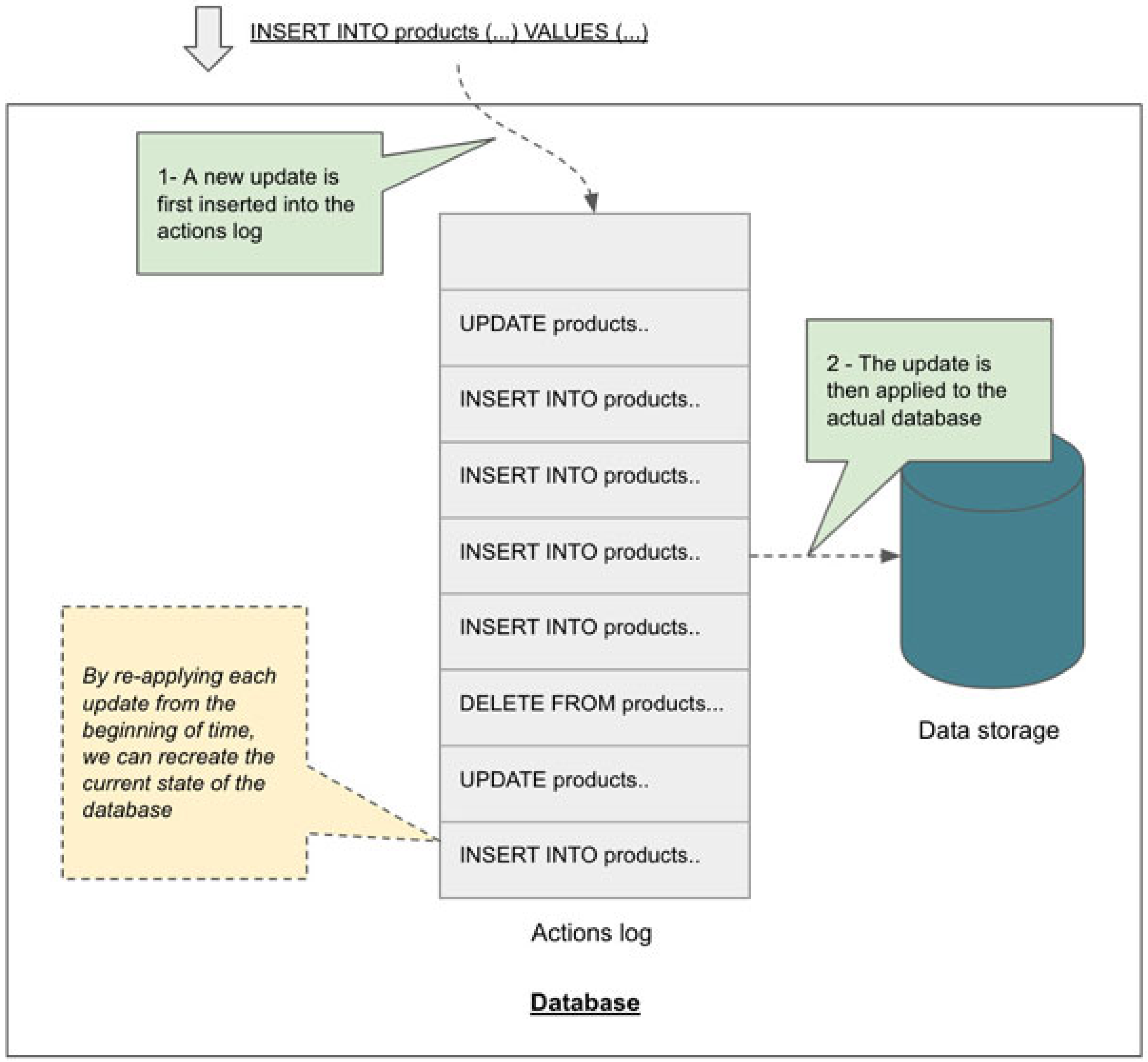

INSERT INTO products (...) VALUES (...)
1- A new update is first inserted into the actions log
UPDATE products..
INSERT INTO products..
INSERT INTO products..
INSERT INTO products..
INSERT INTO products..
DELETE FROM products...
UPDATE products..
INSERT INTO products..
2 - The update is then applied to the actual database
By re-applying each update from the beginning of time, we can recreate the current state of the database
Data storage
Actions log
Database

Figure 4.6: *Activity log within an SQL database*

If this approach sounds familiar it is because we applied the same techniques in this chapter at the *in-memory data storage* section.

One advantage of recovering through activity logs is that, if updates done to the database trigger any side-effects (like running any SQL stored procedures), this approach will guarantee that these will be executed in the same order they originally did.

In MongoDB, the set of activity logs is called *Oplog*. In PostgreSQL, these logs are SQL statements, and creating a backup through them is called creating an *SQL dump*.

Backup through replication

The process of creating one or more instances of a database containing all the data from the original database is called **replication**. Replication has many advantages (which we will discuss later on in this book) and one of them is that it can work as a backup mechanism.

Most database engines have added functionality to replicate their data to other running instances. The original database will receive multiple update statements; these updates are then sent to the running replica so it can apply them as well. Following this procedure, the replica will have the most up-to-date data.

However, if the replica is too behind the original database, (because the original kept getting new records as the replica was being created) they can use the *activity log* from both databases, find the difference of update statements, and apply them as a patch to the replica.

If anything goes wrong with the original database, we can promote the replica to be the main database; then, we create a new replica using the new main database as the source of truth.

Tackling gaps in backups

If we can keep multiple replicas of the database, recovering from failure is relatively simple. But, what happens if we cannot keep a live copy of our database ready to be promoted?

At the beginning of this section, we discussed the case where an application executes multiple database updates in a short period. In that case, it’s almost impossible to create a full copy of the database with the most up-to-date information. Even if we configured our database to create one backup after another (assuming the database allows it without blocking execution), all the information in between backups would be in danger of being lost in case of failure.

For those cases, we can use a hybrid approach:

- Create periodic backups of the database contents, even if there is a gap in operations between each backup.

- Schedule the backup process as often as possible, but without affecting the uptime of your application. This varies between database products.
- For the gaps between the next and latest backups, keep a copy of the *activity log*.
- When we need to recover from a failure, apply the latest backup. Then, on top of it, execute the activity log which includes all the updates captured during the gap in between backups.

Since we're taking smaller snapshots with the activity log, we can back it up more often than we do the full database backups. *Figure 4.7* provides a visual explanation of this approach:

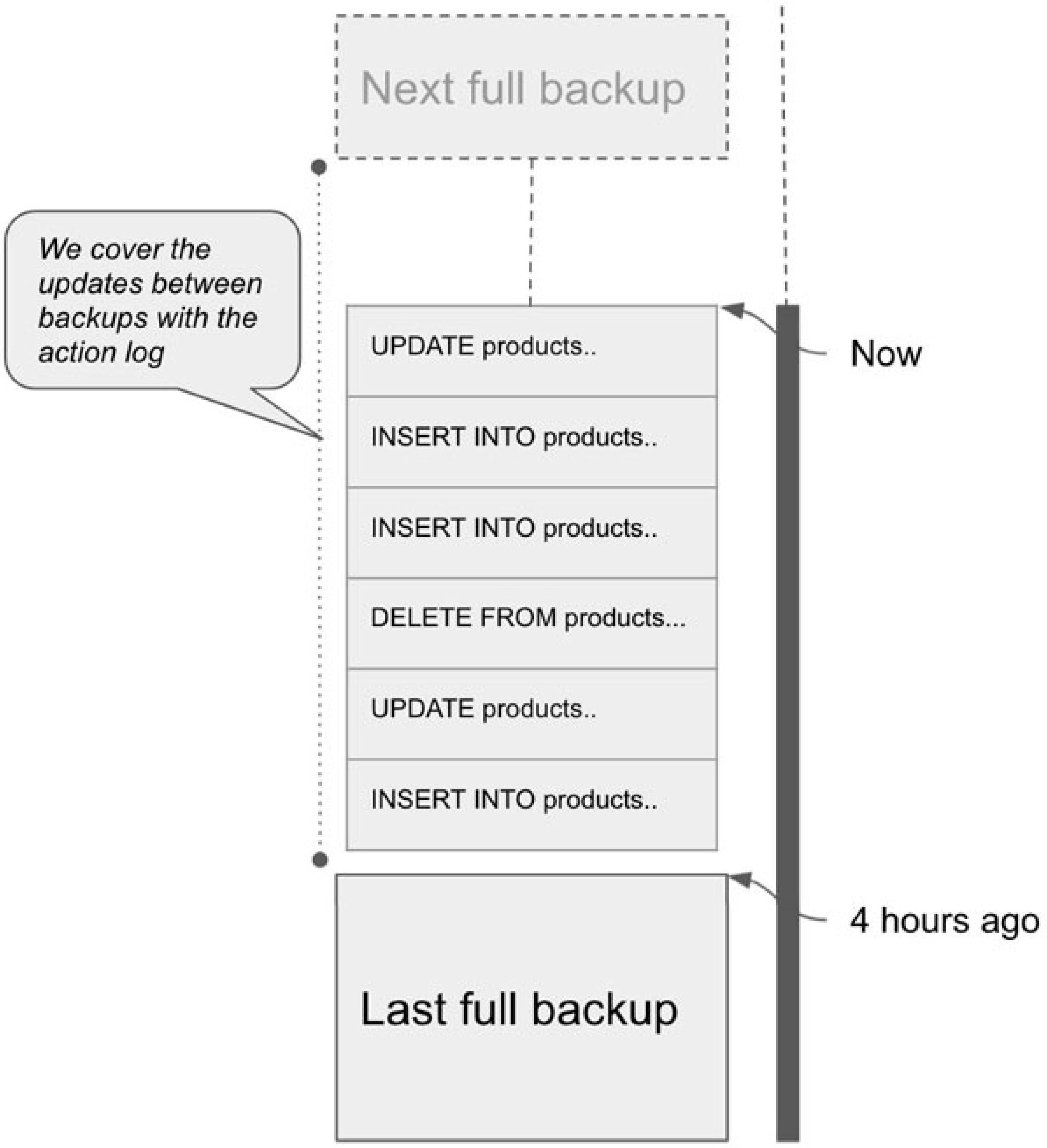


Figure 4.7: *Filling the gaps between full backups with the action log*

While it has a relatively high level of complexity, this hybrid approach can minimize the amount of data we would lose during failure.

Designing data storage in a production system

In this chapter, we explored which kind of data works best for each type of database. By now, you should be able to look at your data models and their relationships and know which kind of database would be better fitted for them.

In addition to the structure of the data models, another factor to be considered while designing a database layer for an application is the *deployment strategy*. When talking about data stores, a deployment strategy is a way we choose to host our database. The most common deployment strategies are as follows:

- Database and application share a server
- Deploy database in its own server(s)
- Embedded databases

Choosing a deployment strategy

In the examples we've seen in this chapter, we've assumed that databases are deployed into their own servers. By exploring different deployment options, we can define architectures better suited for many use cases.

Database and application share a server

For startups or small companies, budget, or infrastructure constraints might limit us to only being able to pay for one single server to host our whole application.

In this scenario, we deploy both the database and the application together on the same server. This approach brings its own set of challenges:

- **Single point of failure**: If the shared server breaks, the whole application will crash. If the disk fails beyond repair, we will lose all our data.
- **Hard to recover**: When an application runs in its own server independent to the database server, if something goes wrong with the application server we can take advantage of the fact that the application itself is stateless and we can just spin a new VM or container with a new instance of the app. If the application shares the server with the database, it's no longer stateless and we need to recover through a backup.

- **Less available storage space and memory**: Sharing a server means sharing its resources. If the database grows too much, it might eat into the space the application needs to correctly execute.

For all its problems, this deployment strategy has some advantages:

- **It simplifies maintenance**: There is just one single place where logs, data, and everything else is hosted, so it is easy to monitor and maintain.
- **It reduces network latency**: The communication between the application and the database is really fast, as there is no need to send a request through a network.
- **It's (kind of) budget-friendly**: We only pay for one server, though it has to be a slightly larger server than if the database or the applications themselves were running alone.

Unfortunately, while this deployment strategy is possible -and for some cases the only option- it's not advisable unless this is a non-critical or small application.

Deploy database in its own server(s)

The most common approach is to deploy the database server into its own server. How many servers you need depends on multiple factors which we will discuss later in this book when we talk about distributed systems.

For now, all you need to know is that for most applications that plan to have multiple users and grow with time, a database will need at the very least one server for its own.

Embedded databases

Embedded databases are particularly common in mobile applications where the client has access to the device's file system. In mobile apps, there is a big chance that the device will be offline as users move into areas without network coverage.

Both Android and iOS have great support for embedded databases like SQLite, and it's common for mobile applications to fully make use of them.

Combining databases to approach complex use cases

Some common architectures using embedded databases showcase how two data stores with different deployment strategies can be integrated. Take, for instance, *PouchDB* and *CouchDB*. Both are document-based databases: *PouchDB* is an embedded database while *CouchDB* is usually deployed to a remote server.

The mobile app can store all data directly in *PouchDB*. If the device is offline, the data will remain safely stored locally in the embedded database. Once the device goes online, the changes will be replicated to an externally deployed *CouchDB*. While the device is online, changes will be constantly synched between both data stores. The following *Figure 4.8* shows this process at a high level:

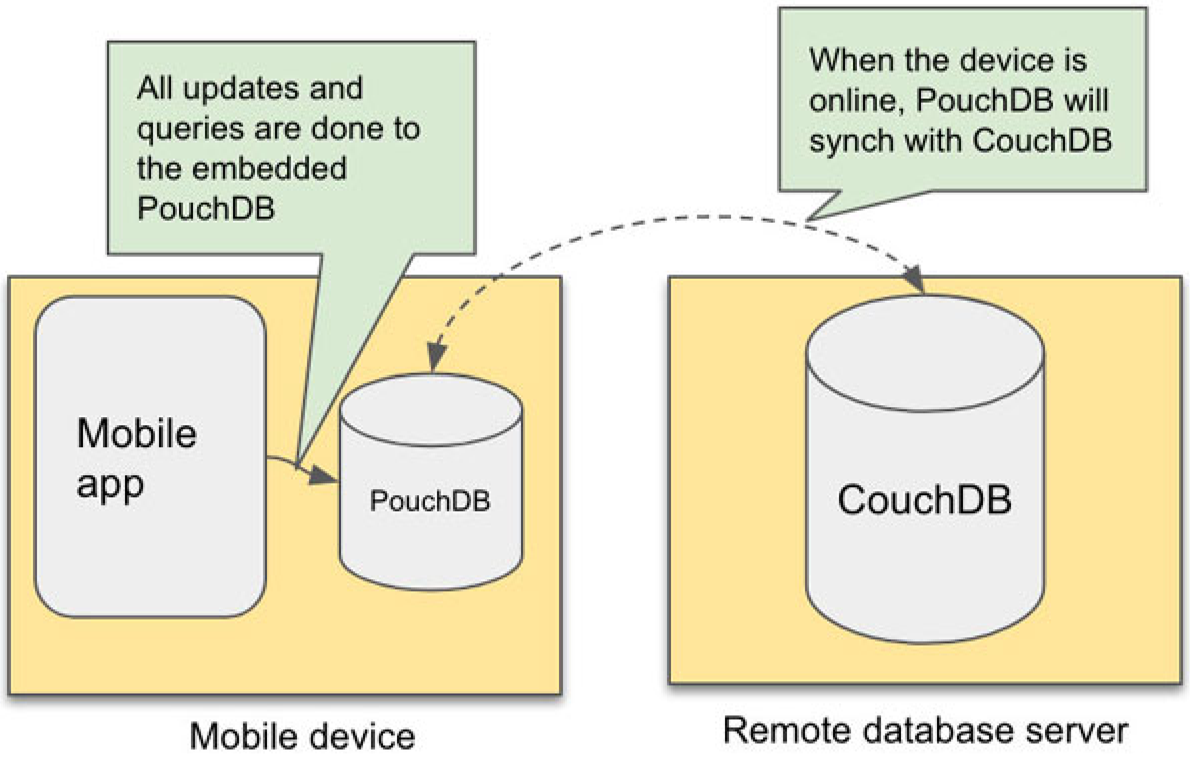

All updates and queries are done to the embedded PouchDB
When the device is online, PouchDB will synch with CouchDB
Mobile app
PouchDB
CouchDB
Mobile device
Remote database server

Figure 4.8: *The flow of an offline-enabled mobile app with an embedded database*

The following piece of code shows how to connect to *PouchDB* as an embedded database in JavaScript and configure it to automatically replicate to a remote *CouchDB* instance:

```
var db = new PouchDB("notes");
db.put({
  _id: "NOTE123",
  title: "TODO for tomorrow",
  content: "Do the laundry",
 });
db.changes().on("change", function () {
  console.log("There was a change in the database");
 });
db.replicate.to("http://couchdb.example.com:5984/notes");
```

As in every asynchronous database, this synchronization is prone to conflicts. What would happen if, while our user is offline, someone deletes the record they were updating?

Each database has its own strategies to resolve conflicts, ranging from requiring a database operator to manually fix the conflict to providing configurable strategies to merge conflicting records. You can find a link with more information about how *PouchDB* handles conflicts in the *References* section of this chapter.

This not-so-complicated setup allows us to tackle a pretty advanced user requirement. As we gain more experience, we find more ways of combining multiple types of databases to build rich systems for modern applications.

Use Case: Defining a data model for the Pizza Place application

To end this chapter, it's time to go back to the Pizza Place example. Let's revisit one specific functional requirement.

Requirement: Users should be able to see the menu on their phones or computers

We know that the menu is a list of all the food items which are offered by the Pizza Place, along with a description of the food items, a list of ingredients, and a set price. We can create our first model from this information:

1. `Model: Menu:`
2. `Attributes:`

```
- Food Items: List(FoodItem)

Model: FoodItem
Attributes:
- Description - Text
- Ingredients - List(?)
- Price - Number with decimals
```

Remember, it’s important that you validate with your client that this list is complete and accurate. Their feedback is critical to find errors early on in the design process.

Notice that we didn’t specify which data type the list **`Ingredients`** contain. Let’s pause for a moment to think about this attribute. How many different ways can we represent it? We need to ask some questions about this possible model:

- Does it have attributes on its own? If it does, do they need to be captured in the application? Do we need to capture a description or its weight?
- How many ingredients can there be? Is it a fixed number? Will we need to add new ingredients often?

The answers to these questions will guide us to find the best implementation for **`Ingredients`**.

No attributes, fixed size of ingredients

If the model doesn’t have attributes on its own that need to be captured, and they always use only a few ingredients -with no plans of adding any more ingredients soon; the best representation would be to create an **`enum`** with a fixed list of ingredients and use its values.

The enum (or enumerated type) is a data type consisting of a set of named values. This list is static, as shown in the following code snippet:

```
enum INGREDIENT_ENUM = {“cheese”, “pepperoni”, “no-glutten-crust”}
…
```

3. `ingredients: list(INGREDIENT_ENUM)`

Only elements that are part of the **`enum`** can be added to the **`ingredients`** list.

Pros: The advantage of using **`enums`** is that you know for sure what values can be contained by that list. No need to add special validations for unknown ingredients.

Cons: In order to add more values to the enums, the application needs to be programmatically updated, compiled, and deployed again. Users cannot add values on their own (unless they have access to the source code and experience with software development).

For this use case: After validating with the client, we can easily see that they need the flexibility of adding more ingredients as they see fit. Using enums is not the best approach for this model.

No attributes, dynamic list of ingredients

If the model has no attributes, but there is no limit in the number of values that can be added to it, the simplest solution is to use **`string`** values:

1. `ingredients: list(String)  // e.g. [“cheese”, “pepperoni”]`

In principle, this looks similar to the enum approach. The difference is that the values in the **`ingredients`** list don’t need to be part of any predefined set.

Pros: Since the only constraint is that the value needs to be a string, users can input any value for the ingredient name, and they can add as many ingredients as they need.

Cons: The lack of a structure makes this field prone to mistakes: One user might write *pepperoni* while another **`peperoni`**. If we need to search for all the ingredients ever used, we could potentially get duplicate values.

Also, with no constraints, users can add any string, even if it’s not really an ingredient (for example, “plane”, “fast”, “garbage”).

For this use case: This approach could work for the Pizza Place. However, after talking with the client, we see that, since one of their selling points is the use of fresh and healthy ingredients, they really need to care to communicate to the public that these ingredients are locally sourced. It’s important we explore additional models and see if we can capture this intent.

With attributes, dynamic size

As soon as we figure out that an attribute has attributes on its own (which also need to be captured), we know that it needs to be a model on its own. In this case, we will capture just one extra attribute for ingredients: **`description`**.

```
Model: Ingredient
Attributes: Name: Text, Description: Text
…
// store either the full ingredient instance object or a reference to it

ingredients: list(Ingredient)
```

This model allows the Pizza Place managers to describe whatever they want to communicate for each one of their ingredients in a better way.

Pros: A full independent model allows us to capture more information about each given attribute. And if more attributes need to be captured later, we can expand this model.

This approach is a good middle point between enums and strings: it provides well-formatted and validated data; it's easier for users to add new values (for example, they will have a user interface to add values to the catalog of ingredients), and they still can choose from a list of correctly named ingredients (the contents of the **`Ingredient`** table) when selecting the contents of the ingredients list.

Cons: This approach requires extra maintenance, as it requires users to first add the ingredients to their own table before they can add them to an order.

Also, as we add more models, the complexity of our database grows. If we create models for everything -even if they are not needed- we'll end up with a bloated database that won't perform well due to all the joins -and the increased chance of un-optimized queries- we will need to fetch even the simplest information.

For this use case: This approach fulfills everything our clients need for their business cases: The menu can include why their pizzas are special and it can express that each ingredient is local and fresh.

It's a good trade-off in maintenance too: The Pizza Place employees are fine with maintaining the list of ingredients. It actually gave them the idea of extending the model to maybe, in the future, include images for each ingredient.

Choosing a data store for storing the pizza menu

Let's revisit some of the databases we discussed in this chapter and see which one would be a good candidate for storing the data needed to render a menu:

- **In-memory data storage**: We need to persist our data, so it's not possible to only store our data inside memory-stored data structures. Once we move to grow this system to make it distributed, we will revisit in-memory data storage for using it as a cache layer, or to store other session-specific data.
- **Document database**: Considering the models of *Menu* and *FoodItem* and *Ingredient*, we defined in this section, a document database could be a good choice: We only have three entities, with two very simple 1:N relationships.

  If we need to use a document database, we have two options:

  - Storing each entity into its own table and connecting them through their keys (e.g. `FoodItem.ingredients = ["IngredientKey1", "IngredientKey2"]`)

  - Denormalize and store a complete copy of each child model into the parent document (e.g. `FoodItem.ingredients = [{id: "IngredientKey1", "name": "Cheese", "Description": "…"},..]`)

  The first approach keeps some level of normalization in our data at the cost of forcing the application to do multiple queries to fetch all data in the menu (remember, it's not common to do join operations in document databases).

The second approach only requires the application to do a single query to retrieve all data, at the cost of having to maintain duplicate data. If a name or description is updated in the catalog, the application will have to iterate through each record in the menu to update it.

- **Relational database**: Relational databases are good choice for this case, as the data structure fits within the expectations of relational data.

  Expanding on the analysis done for the document database case, we can store each entity into its own table, create relationships with foreign keys and fetch all elements in a single join query.

- **Graph database**: There are just not enough entities with relationships between them to fit the case of a graph database. Sure, there is nothing preventing us to use it, (unlike the restrictions given by in-memory data storage), but unless you know the number of entities and relationships will grow, you will end up never using most of the database features.

The winner

As we can see, the database which better fits these models without significant trade-offs is the relational database: We don't need to maintain duplicate data and we can fetch all data with a single query, and the data structure and number of relationships fit really well with the database model.

However, we can choose a document database and our application would be just fine. As long as we know the trade-offs, we need to consider by choosing this type of database, our application will perform well enough to cover all expected use cases.

Scaling the database

Will a relational database scale well enough to fit the application requirements? For this case, it does. A vertically scaled relational database still can serve millions of queries per second, well enough to cover a healthy growth in the number of users fetching a menu for the Pizza Place.

As we add features and the application keeps growing, we will need to reconsider these assumptions. There is no use in expecting every application will need to scale to hundreds of horizontally running servers when most of them could work just fine with a single server.

But don’t worry; we will have more than enough time to discuss distributed systems in the upcoming chapters. For now, let’s focus on creating a strong foundation for the application to grow on.

Conclusion

In this chapter, we covered a lot of information about data: How to represent it and where we can store it. We know that the application state is the data that represents the contents of our application at a given point in time.

Data is the most critical asset for any software application. We saw how it can be stored in very fast but impermanent in-memory data structures. These are perfect to store temporary data that needs to be quickly fetched; caches are an example of such use cases.

We visited the definition of multiple types of databases, the kind of data that is better suited to be stored in them. Remember, the SQL versus NoSQL debate is over-simplistic and unfair to the rich variety of databases we have available for so many different use cases.

Document databases are great for data with high locality and few relationships between its entities. They require us to rethink concepts like normalization, but the low number of relationships in its data allows them to easily scale horizontally if demand increases.

Graph databases are great for data with a lot of relationships between entities. In graph databases, relationships between entities are first-class citizens, which enable them to have attributes on their own. Finally, querying graph databases allows us to expressively explore these relationships between data.

Relational databases have been around for many decades. They are robust and they should not be so easily discarded. They are a middle ground between document databases and graph databases in the context of data and its relationships.

In the end, there is not a single database that is the best choice for all cases. It depends on how your data is structure, how many resources and effort can you put into scaling your database, and what infrastructure and developer talent you have available in your team. There are too many factors to consider in order to choose the right database, and as you gain more experience you'll become better and better at analyzing them all.

The good thing is that you can choose a database that is not the absolute best for your use case and it will still work efficiently. Remember that for years most applications used only relational databases, even if the data didn't make sense. Choosing the wrong database is not the end of the world, but it will definitely make your life harder.

By this point in the book, we should have enough knowledge on how to build a whole backend application for most simple use cases. In the next chapter, we will focus on building *quality* into our application so it can be resilient and easy to maintain.

Questions

Considering the types of databases, we saw in this chapter, which one would you choose for each of the following applications?

- An interactive demo application that will be presented as a proposal for a new project.
- An online bookstore that only handles the title and ISBN of each book. No publishers, nor other entities.
- A personal blog.
- A new social network for pet owners that needs to display their pets' profiles and allow them to befriend other users.

References

- Fixing conflicts in PouchDB: **https://pouchdb.com/**
- MongoDB's Replica Set Oplog: **https://docs.mongodb.com/manual/core/replica-set-oplog/**
- MongoDB Backup methods: **https://docs.mongodb.com/manual/core/backups/**
- PosgreSQL backup methods: **https://www.postgresql.org/docs/9.1/backup.html**

CHAPTER 5
Automating Application Testing

Building software applications is a task that will never be completely finished. Even in non-iterative software development methodologies like Waterfall, it is assumed that the Software Development Life-cycle ends in a maintenance phase, which lasts for an indefinite amount of time. There will be always more work to do, more code to write.

As backend developers, it is our responsibility to build the code that matches with the user's requirements and expectations. At a high level, this is an exceptionally difficult task: The user requirements will always change in time, and if we want to keep complying with them, our application needs to adapt without breaking existing features.

There is a big obstacle to adaptation: Defects. Defects are errors in our application and can be introduced by a long list of factors: Faulty processes, unavailable services, poorly understood requirements, inexperienced developers, and software regressions, among many more. Defects cause our software to not comply with our user requirements, which in turn makes our applications lose value.

Since it is really easy to introduce defects into our application, we need to build processes and mechanisms to prevent these errors from happening in the first place, and to detect them if they end up happening anyways. One of the most critical of these processes is **testing**.

In this chapter, we will discuss all you need to know about testing: Why it is important, who owns that process, and how to build a good testing strategy for our application. We will also talk about tools commonly used in testing like **mocking**.

Testing is such a large and complex topic that there are full roles dedicated to this field: We have roles for performance testing, security testing, and compliance testing, and so on. In this chapter, we will discuss only the topics that backend developers need to know in order to be proficient in their own roles.

Structure

In this chapter, we will learn the following topics:

- Certainty through testing
- Manual testing
- Automated testing
- Unit testing
- Integration testing with Selenium
- Testing and CI/CD
- Other automated tests: Static code analyzers
- Defining effective test cases
- Non-functional testing

Objectives

After reading this chapter, you should be able to understand the importance of testing in the software development life cycle, how to design and write effective tests, and how to write automated tests that can be executed consistently and continuously.

The main goal of this chapter is for you to learn how to protect and manage defects that can lead to economic and reputation losses in your software application.

Certainty through testing

The idea behind testing is simple: Check whether the application works as expected. That is a very wide and subjective criterion, though. We use requirements, both functional and non-functional, to measure how much the application behaves as it should. However, requirements are just reference points.

There are multiple use cases beyond the list of requirements, that also need to work correctly; from implicit requirements which are too obvious to be listed (for example, the application should be available to users a majority of time), all the way to corner cases that might have been missed from the original requirements (for example, abusive user behavior like making too many requests or ordering all products at the same time to block other users from getting them).

This list of explicit and implicit requirements needs to be constantly updated to cover newly found use cases. Using this updated list, we can create a checklist of tests we need to perform on the application to guarantee that it behaves as expected.

In *Chapter 4, End-to-end Data Management*, we discussed how often databases should create backups. We argued that, in a perfect world, we would need to create full backups for every update done to the database. More often than that would be unnecessary as data would remain unchanged from one backup to the next; and less frequently we would risk losing data if failure happened in between the last update and the next backup. The same reasoning can be applied to verify the correctness and integrity of the whole application.

We need to re-execute the tests in our checklist every time we introduce a change to our application to make sure the new changes work and no **regressions** happened to existing features. If we skip one single check or test, we risk introducing errors to the application without us knowing about it. This is, of course, not an easy goal to achieve in practice.

Note: Regressions are defects in features that were previously working correctly. Regressions tend to be caused by changes to the code that affect directly or indirectly other existing features. The easiest way to find regressions in time is to perform as many tests as possible on every feature, even if the updates are unrelated to them.

Manual testing

When we start coding a new application and implement only one or two features, it is easy to test all of them fairly quickly. You can make a new code change, manually test each feature to make sure they are still working, and then deploy the change to your users.

As the application grows, it becomes difficult for a single person to write code *and* test every feature. The added complexity requires that more people get involved; in some teams, it becomes a responsibility shared with the whole team, other teams have people who are specialized in testing software applications whose official role is **software application tester**.

What makes software testers different from regular developers is that they have experience finding hidden corner cases that are usually ignored by software developers and use these omissions to cause errors. The ultimate goal of software testers is to find every defect in the application. While this is practically impossible, a tester will try to find as many errors as possible.

Note: The nature of a software tester's role can put their efforts at odds with other team members.

Some software developers -and even some project managers- see software testers as obstacles that slow down the continuous delivery of new features to clients. This is the wrong approach, as those defects are going to be part of the application, whether the tester finds them or not.

Software testers are not villains who want to see the project fail, nor humiliate developers who forget to add a validation or two. Testers are quality assurance experts who work with both developers and product managers to define a minimal quality standard which guarantees that the application will be up to the user's expectations.

Manual testing is a great way of doing exploratory testing and catching bugs that have never happened before, and finding new use cases.

Types of manual tests

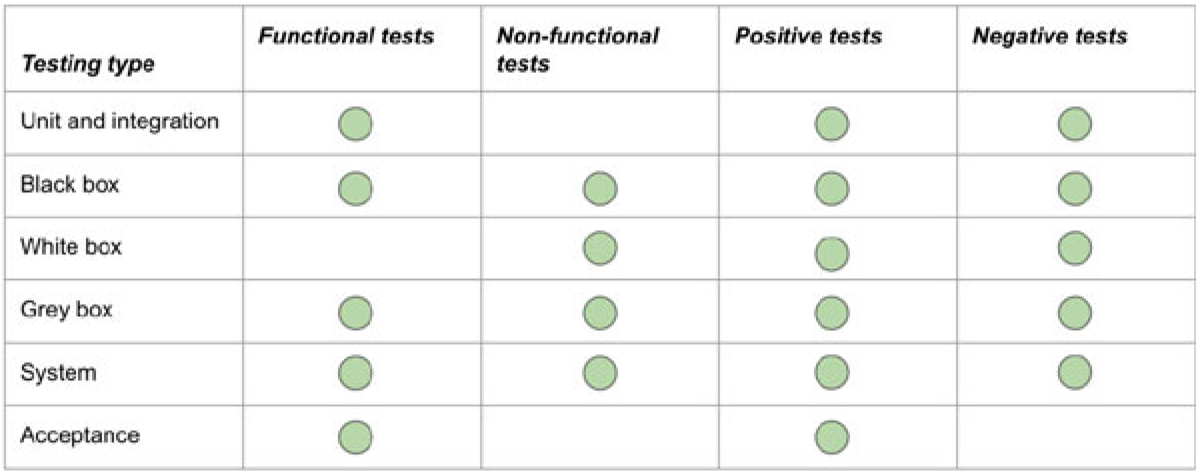
The tests we create to validate the user requirements are called **functional tests**. Tests which assert specific technical characteristics of the application like performance, security, or reliability, are called **non-functional tests**. As you can see, the name of these tests match the same convention we have for the requirements they are covering.

When we assert for conditions that follow the expected requirements, we call these **positive tests**; and when we need to validate the application handles failure cases correctly (for example, the user introduces the wrong input, or an external system fails), we call these **negative tests**.

Using these two classifications, we have different types of manual testing processes, which are as follows:

- **Unit and integration testing**: Test individual pieces of software (like a function) in isolation or in combination with other pieces of software. These tests are more common in automated testing, but they can also be done manually.
- **Black box testing**: This refers to testing a deployed and running instance of the application. The tester only has access to the same features users have, with no priviledged knowledge about the application’s source code.
- **White box testing**: This refers to inspecting the source code of an application to find bugs in it. No running instance of the application is given, so testers need to only analyze the code they are given.
- **Grey box testing**: This is a mix of White and Black box testing. This test uses both a running instance of the application and its source code.
- **System testing**: This is a system-wide testing performed by testers or developers. It performs functional and non-functional tests, but it is slightly more focused in the latter.
- **Acceptance testing**: This phase asserts functional tests only. It is usually divided in *alpha* (pre-release, done by testers and developers) and *beta* testing (post-release, done by users in a real environment). Acceptance testing tends to be the last step of testing we perform in a development cycle.

Figure 5.1 shows how each type of testing is related to each classification:



Testing type	*Functional tests*	*Non-functional tests*	*Positive tests*	*Negative tests*
Unit and integration	●		●	●
Black box	●	●	●	●
White box		●	●	●
Grey box	●	●	●	●
System	●	●	●	●
Acceptance	●		●	

Figure 5.1: *Types of tests executed in each manual test process*

Building effective manual tests

In this section, we will discuss some of simplest steps we can take to perform manual testing like a professional application tester would do. Obviously, application testers have the experience and skills to perform these same steps in a more efficient way than regular developers would usually do.

Just like in the development process, manual testing can be seen as a cycle, as shown in *Figure 5.2*:

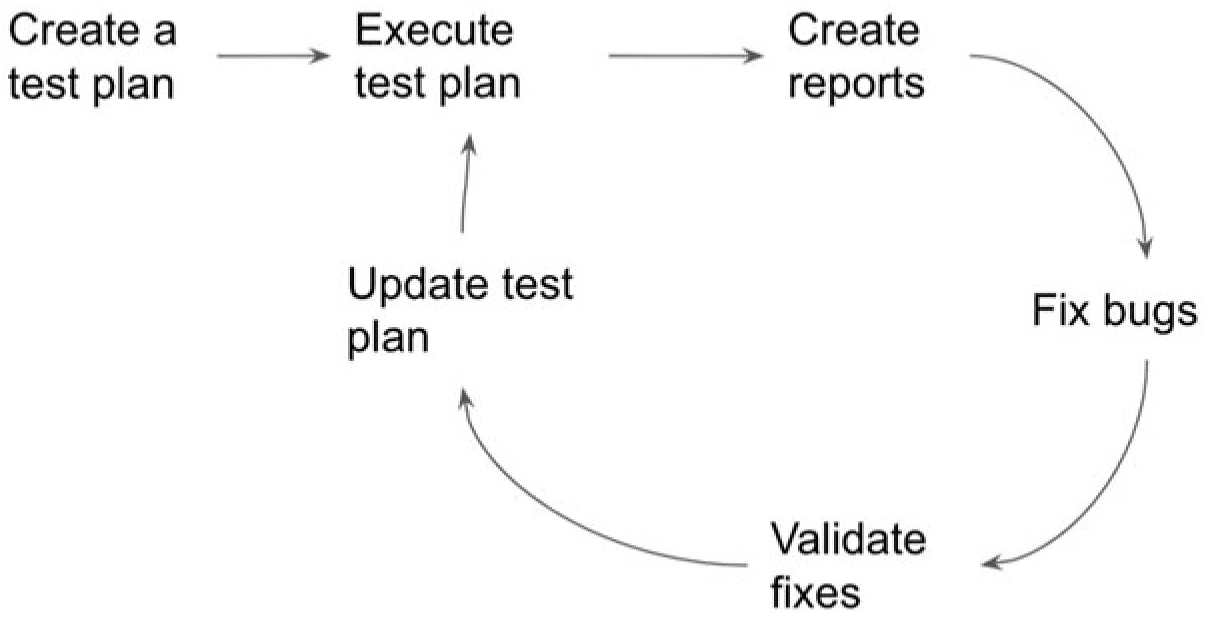


Figure 5.2: *Manual testing cycle*

Creating a test plan

Before we start testing our application, we need to fully define *what* needs to be tested. The starting point is the checklist of tests focused in user requirements.

Checklists are powerful tools used during manual tests. It is not a coincidence that all sensitive jobs make use of them: Pilots, doctors, caregivers, among others; all these professions make active use of checklists. A checklist give us a clear understanding of what needs to be tested, the tests we have already performed, and the tests we are to execute yet.

In the world of application testing, these checklists are known as **test plans** or **test scripts**.

The following is an example of the definition of a test in a test plan:

Use case: "As a [role], I need to [action_to_perform] so I can [correct_action_result]"

Steps:

- As user [role], log into the application at [application_url]

- Select the text field marked with the [input_label] label

- Type the following string: [value_to_search]

- Select and activate the "Search" button

- Expected result: A list of results containing [expected_number] elements should be visible in the browser window.

In addition to test cases, more information should be included in the test plan. Depending on the policies in your team, a test plan can follow pretty strict requirements. Some require testers to define the scope of the testing (what is tested and what is not), the criteria to consider the testing '*done*', success criteria for each test, an estimation of the effort required to execute the test plan, among other details.

For software developers, the heuristic here is to discuss with the rest of the team how strict you want to be with the record-keeping of the test plan. In some cases, being too rigid in the process of creating the test plan leads to a situation where we spend more time filling all the required data for the test plan than actually designing and executing the test itself.

For large applications, we can have many test cases. Since manual testing has hard time constraints, you should *order the list of test cases by priority*. Testing the most critical features should be at the top of the list. Optional, or "*nice to have*" features should be at the bottom. In case, we run out of time while testing, we would have at least tested the most important features.

Executing the test plan

The test plan is useless if it is not followed correctly by the person executing the tests. Make sure that every element in your checklist is verified. If you defined a list of steps to perform for each test case, follow each one of them without skipping any.

By following the test plan, execution should be pretty straightforward. However, if you identify gaps in the plan, or maybe some tests are taking longer than expected, you can adjust the test plan in the next test iteration.

Creating detailed reports

If you plan to fix yourself all the defects you find during your test, you might skip this step. However, it is common to find defects in code that belongs to someone else. Or the defect we found may have already affected users in production. In those cases, you will need to create a report as detailed as possible.

The most important part of finding a defect is to capture the steps to reproduce it. If a defect is not reproducible, the developer in charge of fixing it might think it is not critical enough to be investigated.

Good reports are composed of multiple elements:

- The conditions in which the error happened. Did you test using a specific user role? Did it happen for a specific group the users only? Is it specific to a browser or OS version?
- The detailed list of steps to reproduce the problem. Describe step by step the shortest way to get to the error. Do not skip any step, as obvious as it might seem, as it might confuse the developer trying to replicate the issue (for example, did you log in or not?).
- Description of the expected behavior versus the actual behavior, to make it explicit what the problem is. Include videos and/or screenshots if available. Visual evidence communicates more than long lists of text describing the steps to reproduce. Any extra piece of evidence helps.

Validating fixes

Once the defect is fixed, apply the whole test plan again. If time allows it, it is important to start again from the beginning and not skip any steps even if, in theory, the fix would not affect some of those tests. The whole test suite should be successful before we can call it done.

If defects can still be found, then we send them back to be fixed again.

Update test plan

After validating that all defects were fixed, we might find that the test plan is out of date. We may also have discovered some new corner use cases while executing the test plan. It is critical to update the test plan with this new information so next time we have to run the test suite again we do it from the most informed position.

Advantages of manual testing

Manual testing has multiple strengths that make it relevant, even nowadays with all the tools we have for automated testing.

The strengths of manual testing are: Explorability, horizontal validation, and user-centered vision.

Explorability

Humans are curious by nature. Even if application testers have well-defined test plans and scripts, if they see something different, like a button that was not there before, or a new feature, they will definitely try to use it.

This strength leads to finding defects in a way that automated testing would seldom do: Without following a pre-defined plan.

Horizontal validation

Human beings also have *general intelligence*. This means that we are good in generalizing knowledge and we can perceive and process things that we originally did not mean to.

Let’s say we need a test for checking feature “*A*”. After some code changes, feature A still works as expected for a given use case. However, that same use case is now causing an error in a different feature “*B*” (for example, a concurrency issue like a race condition) in the same page. If this was an automated test focused in feature “*A*”, the test will report success even if feature “*B*” broke.

Automated tests work by validating very specific assertions (more on this later), and these tests will pass as long as those assertions are true, even if other errors happened at the same time.

Let's think of another example: We need to test whether a table in a web application disappears when the user clicks on a button. We can write an automated test to verify that the module is not present once the test itself executes the action in the button. However, we could have a situation where clicking on the button throws an error that hides the content for the whole page. If the test only asserted for the absence of the table without checking for the presence of other parts of the page, the test will pass. A human tester, though, will very likely catch this issue, even if they were focused in the table component.

User-centered vision

One of the most important strengths of manual testing is that testers can put themselves in the position of our users. They can deduce their needs and their intents in ways we might not have considered during design and development.

Our users will not always behave as we expect them to. They will not always be rational and they definitely will not always follow the right path. Testers can understand this and account for this behavior while performing tests.

Testers provide a human factor that automated tests just cannot replicate yet.

Manual testing and Agile

Testing is such a critical part of software development that it has been embedded in traditional **Software Development Life-cycle (SDLC)** methodologies since many years ago. For instance, the Waterfall methodology defines a specific phase for testing in the software development life-cycle.

Figure 5.3 contrasts the testing phases in both **Waterfall** and **Agile**:

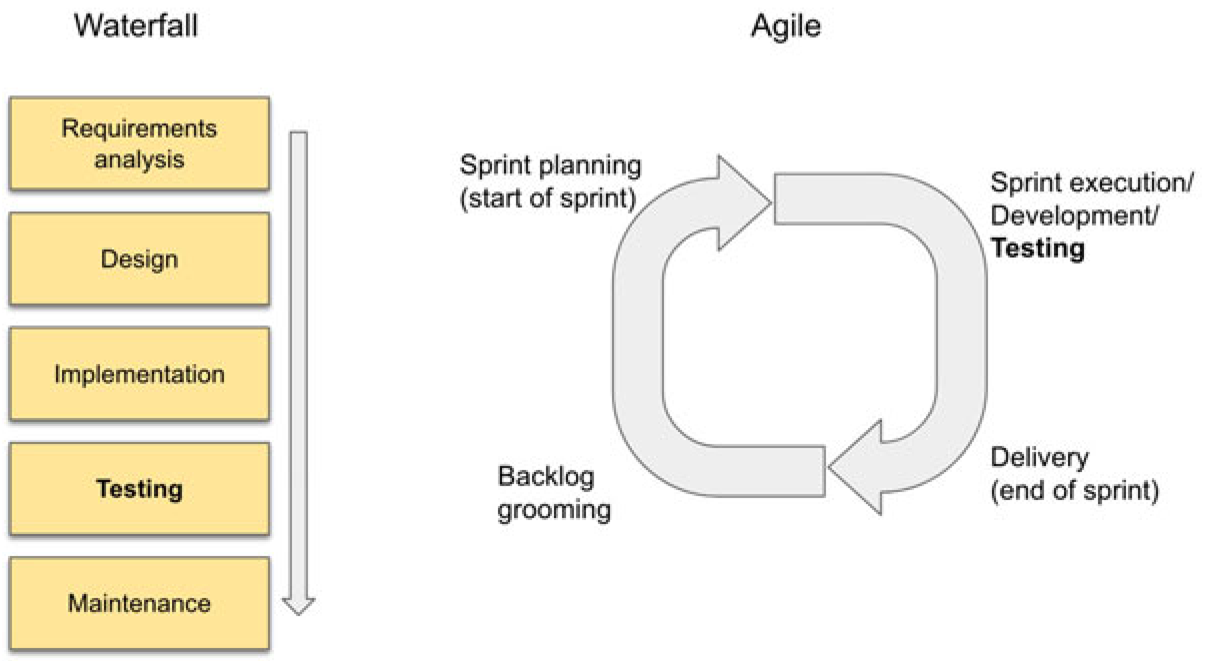


Figure 5.3: *Testing in both Waterfall and Agile methodologies*

In Waterfall, the whole team can dedicate a given amount of time to test and validate each of the features. Only after testing is complete and all found bugs are fixed, the application can be released.

The introduction of Agile methodologies in software development changed the approach we have to do testing. In Agile, teams aim to deliver features as fast as possible; each short iteration should render working, deployable software. As your development cycle accelerates, developers might be able to build and deploy new features in days or even hours.

For short iteration cycles, manual testing does not scale well: Testers might be still working on executing their manual test scripts when a new change needs to be tested.

In addition to the reduced development cycles, Agile as a methodology does not distinguish a specialized role for software testing. It assumes that everyone in the team will collaborate to continuously test the application. This omission has led to many teams to remove the role of an application tester from their teams and projects.

However, who owns the task of testing? Who must verify that the application is thoroughly tested before a deployment? In many development teams, this task went back to the developers. Each developer has to guarantee that their code changes are tested and free of bugs.

In Agile, automated testing is critical as it is the only approach which will allow us to quickly iterate and build new features with enough confidence that the existing application elements are not breaking due these changes.

Let others test your code

The case for having application testers who perform both manual and automated tests goes beyond thoroughness.

People who have to fix defects have a motivation to find as few of them as possible. This is not a critique of software developers but simple human nature: We tend to overlook things that will make our life extra-difficult, even if we do not do it consciously.

A person whose only goal is to find defects, like a software tester, has no such limits in motivation. They will report every single defect they find regardless of how time consuming or difficult to fix they are.

As a software developer, you *have* to test your own code; you do so during and after writing it. You create automated tests for it. However, you cannot be the only person to test your code before delivering it to your users. If your team has no dedicated testers, have some other developers take a look at it.

Automated testing

We call automated testing to the process of building code to execute tests against a software application. Writing code to test code.

We build automated tests because they are *repeatable*. We can execute them multiple times with little effort and with almost no human intervention. We can schedule them to run when we need them to, or we can execute them on demand. Repeatable tests give reliability to the test process itself: we can be rest assured that no tests will be missed from being executed; a guarantee we do not have with manual testing.

In equal conditions, automated tests tend to be faster than manual tests. In the minutes or hours it takes for manual tests to complete we could have executed an automated test suite multiple times.

Speed and repeatability are the two characteristics we want tests to have. They enable these tests to validate our application every time we make an update to it.

The creation of automated tests is a responsibility shared between developers and testers. Developers create low-level tests like unit and integration tests, while testers typically create high-level tests like acceptance testing, or contract testing. This distribution of work is not fixed, of course. As mentioned earlier, some developers are in charge of building all automated tests due the lack of roles dedicated exclusively to testing. In other cases, application testers also build lower level tests.

As mentioned earlier, there are many types of automated tests. Let's explore the ones that backend developers write in our daily jobs, and at the end, we will mention some other types which are out of the scope of this book.

Unit testing

When we build software, we make assumptions about it. If we write a function to calculate the sum of two positive integers, we assume that the result will always be positive. If we write a function to search for a specific product in an online store, we assume that the result will always be an object of the type *product*, instead of something else like a *cat* or *triangle*. These assumptions give us certainty that the program will behave as we expect it to. Just like *axioms* set the basis for Math, these truths are our building blocks for more complex features.

Automated tests check these assumptions. They run the function that adds the two positive integers and check the results; they call the search function and verify that the response is indeed a *product*.

Unit tests focus on the smallest units of code in the application. Most of the time, these units are functions. But, why do we want to spend effort writing tests for such simple code units? In which way do unit tests help us guarantee our high-level requirements are achieved?

The answer of these questions throws a light into the really interesting aspect of unit tests and software itself: Software applications are nothing else but a set of functions that are orchestrated by some other functions, which in turn become building blocks for even higher-level functions, as so on. We can think of this relationship as a sort of directed graph, as shown in *Figure 5.4*:

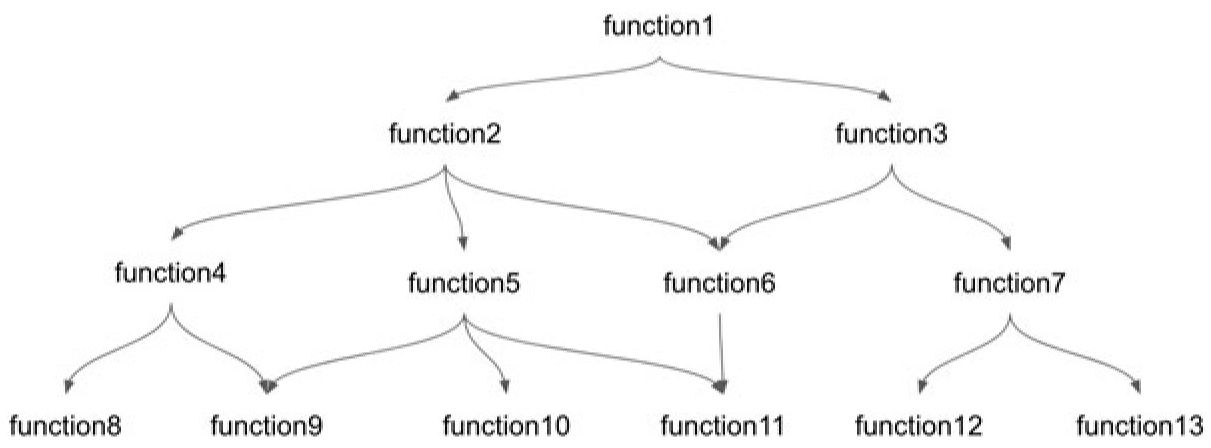


Figure 5.4: *Function execution graph*

The functions at the top are closer to the requirements: High-level functions like “**`Create order`**”, “**`find profile`**”, and so on. The functions at the bottom are reusable, unit functions like “**`toString`**”, “**`sort`**”, “**`toList`**”, and so on.

Any piece of software only works as well as the sum of its parts do. If one single of those functions fails, the error will propagate all the way to the top. If we can guarantee that each function at the bottom will work as expected, we can make the same assumption about the top functions. This is where unit tests shine: They allow us to check each function (or class) in isolation.

The following is an example of a unit test built in Java, using JUnit, to test a function to add in a sample class called **`Calculator`**:

```java
import static org.junit.jupiter.api.Assertions.assertEquals;
import com.example.Calculator;
import org.junit.jupiter.api.Test;

class CalculatorUnitTests {
    private final Calculator calculator = new Calculator();

    @Test
    void additionTest() {
        assertEquals(2, calculator.add(1, 1));
    }
}
```

The major highlights for this Java code are as follows:

- The **`additionTest`** function is marked with the annotation **`@Test`**, which is just a convenient convention used by JUnit to identify tests.
- **`assertEquals`** is one of the many assertions available in JUnit. Other assertions are **`assertNotNull`**, **`assertThrows`**, and **`assertArrayEquals`**. Libraries like **`AssertJ`** exist to provide more human-readable assertions.
- The test executes the function to test, and checks whether the result is equal to the expected value. In this case, the test asserts that the sum of two numbers (1+1) returns the right result.

Just to get a better mental model of unit tests, let's look at the same test, but using Python with **`unittest`**:

```python
import unittest
from Calculator import Calculator

class CalculatorUnitTest(unittest.TestCase):
    def setUp(self):
        self.calculator = Calculator()

    def test_addition(self):
        self.assertEqual(2, self.calculator.add(1,1))

if __name__ == ‘__main__’:
    unittest.main()
```

As you can see, unit tests in Python retain all the key properties of the Java unit tests: A library for testing (**`unittest`**) that provides an identifier for tests (extending the **`unittest.TestCase`** class), a **`test_addition`** function to contain the test logic, and an assertion to verify the result has the expected value. All these are common facts in probably most of the unit tests libraries out there.

Most libraries for unit testing allow you to hook into different phases of the test execution:

- **`setUp`** - Runs before each test. It is used to initialize variables and to create instances of the objects to use during test.
- **`tearDown`** - Runs after each test. All logic to clear and destroy initialized objects can be done here.

Some tools also offer hooks which run just once at the beginning and after all the tests in the test class.

Testing in isolation: Doubles, stubs, and mocks

Testing code in isolation is hard. The function graph we saw in *Figure 5.4* shows how most functions have direct dependencies in others, and even after refactoring to make our code as modular as possible, at least some of your functions and classes will have dependencies in other functions and classes.

Isolated code is easier to debug

Look at the following Python code. We can see two classes: **`ReportService`** and **`PrintService`**. **`ReportService`** is a service used to create a company's report. Once the report is built, it is sent to a remote printer using **`PrintService`**:

```
import time

class PrintService:
  def create_print_job(self, job_to_print):
    # Make a network request to a remote printer
    print("Sending job…")
    # emulate doing a long, expensive call:
    time.sleep(3)
    # emulate a service failing with a probability of ۱۰٪
```

```
        return random.random() <= 0.9

class ReportService:
    def __init__(self, company_name):
        self.company_name = company_name
        self.printService = PrintService()

    def create_report(self):
        print(“Building report… for {}”.format(self.company_name))
        job_to_print = {
            “title”: “Report1”,
            “company”:self.company_name,
            “content”: “This is an example”
        }
        self.printService.create_print_job(job_to_print)
        is_printed = self.printService.create_print_job(job_to_print)

        if is_printed == False:
            raise Exception(“Could not print the report. Try again later”)

        return job_to_print
```

For this preceding example, to simulate that **`PrintService`** is the actual implementation connecting to a real network. For this, we will introduce some emulated behavior:

- Network-based services have added latency. We introduce a wait time of 3 seconds to emulate a long network call.
- Calls to external services can fail without us having any control of it. The **`create_print_job`** returns True 90% of the time, while it returns

False 10% of the time, simulating a service which works correctly most of the time, but it fails unexpectedly some of the time.

If **`create_print_job`** returns False, we need to throw an exception so users know the operation failed and they will need to retry the action. We could have handled it in a different way (like retrying the action) but we will keep it simple for this example.

Now, let's build a test class to assert the report is generated correctly. For now, we will only add one assertion to verify the company name returned in the report is correct:

```
import unittest
from ReportService import ReportService

class ReportServiceUnitTest(unittest.TestCase):
  company = "Company 1 Co."

  def setUp(self):
      self.report_service = ReportService(self.company)

  def test_addition(self):
      report = self.report_service.create_report()

      self.assertEqual(self.company, report["company"])

if __name__ == '__main__':
    unittest.main()
```

After running this test class, we should see one of the following results in the console. If the **`PrintService`** is up and working, your test will pass:

```
python % python -m unittest ReportServiceUnitTests
Building report… for Company 1 Co.
Sending job…
.
```

```
----------------------------------------------------------------------
Ran 1 test in 3.004s
OK
```

Notice the line ran 1 test in 3.004s. We can see our artificial wait reflected in there. For a single example, this does not look too bad, but when your test suite has hundreds of tests, the total time it will take to run will go from many seconds to several minutes.

But if **`PrintService`** is down (again, simulated with a random value), the test will fail with an error similar to the following (some lines were removed for brevity):

```
python -m unittest ReportServiceUnitTests
Building report… for Company 1 Co.
Sending job…
E
======================================================================
ERROR: test_addition
(ReportServiceUnitTests.ReportServiceUnitTest)
----------------------------------------------------------------------
…
Exception: Could not print the report. Try again later
----------------------------------------------------------------------
Ran 1 test in 3.006s
FAILED (errors=1)
```

This instability in the results is caused by not testing the **`ReportService`** in isolation: Any error that affects **`PrintService`** will propagate to **`ReportService`**. For instance, let's simulate a connection error in **`PrintService`** as follows:

```
class PrintService:
  def create_print_job(self, job_to_print):
    raise Exception("Connection error")
```

Run the tests for **`ReportService`** again and you should see the following result:

```
python -m unittest ReportServiceUnitTests
Building report… for Company 1 Co.
E
==============================================================
========
ERROR: test_addition
(ReportServiceUnitTests.ReportServiceUnitTest)
--------------------------------------------------------------
--------
Traceback (most recent call last):
  File "ReportServiceUnitTests.py", line 11, in test_addition
    report = self.report_service.create_report()
  File "ReportService.py", line 19, in create_report
    self.printService.create_print_job(job_to_print)
  File "ReportService.py", line 5, in create_print_job
    raise Exception("Connection error")
Exception: Connection error
--------------------------------------------------------------
--------
Ran 1 test in 0.000s
```

There is technically nothing wrong with **`ReportService`**, but its tests are still failing. In this example, it is really clear where the problem is, but in a real-life application, we would spend a lot of time trying to find what is wrong with **`ReportService`** when in fact there is nothing wrong with it. Again, all this is because we are not testing the functionality in isolation.

We cannot remove external dependencies just for the sake of testing. The best we can do is force these dependencies to behave in a way that they do not interfere with the tests we are running.

Isolating test dependencies

When building unit tests, we need function calls to external dependencies like **`PrintService`** to behave in two ways: They need to be *fast* (in general) and *deterministic*.

A deterministic dependency is that whose behavior does not change unexpectedly: *For a given set of parameters to the dependency functions, it will always return the same result*. In the case of **`ReportService`**, the real implementation of **`PrintService.create_print_job`** is not fully deterministic: The function can take longer to run if the network is slow, or it can fail if the network has connection issues. Non-deterministic dependencies lead to difficult-to-debug issues: If **`ReportService`** has a bug where it does not handle failures from **`PrintService`** correctly, we will only find this issue when, by coincidence, the actual service fails.

In addition to determinism, we need dependencies to run as fast as possible so we can execute our tests repeatedly. Developers often skip long tests, and these tests become a burden in the long run.

The first change we need to do in **`ReportService`** is to move the instance creation of **`PrintService`** out to make our code as modular as possible. In *Chapter 3, Designing APIs*, we did something similar so that we can inject different implementations of the same interface.

```
class ReportService:
    def __init__(self, company_name, printService):
      self.company_name = company_name
      self.printService = printService
#...
```

We now pass the **`printServer`** instance through the class constructor. The tests **`setUp`** hook must be updated too:

```
class ReportServiceUnitTest(unittest.TestCase):
  company = "Company 1 Co."

  def setUp(self):
      print_service = PrintService()
         self.report_service = ReportService(self.company,
print_service)
#...
```

Probably, this talk about injecting different implementations is giving you a hint of what we will do next.

Test stubs

Using **dependency injection** again, we can create a stub implementation of **`PrintService`** called **`FakePrintService`**, and inject it into **`ReportService`** instead of injecting the real service:

```
class FakePrintService:
    def __init__(self, company_name):
        self.company_name = company_name

    def create_print_job(self, job_to_print):
        print(“This is a fake class”)
        return self.company_name == job_to_print[“company”]

class ReportServiceUnitTest(unittest.TestCase):
  company = “Company 1 Co.”

  def setUp(self):
      print_service = FakePrintService(self.company)
      self.report_service = ReportService(self.company, print_service)

#...
```

If we are using an interface for defining the service API, we would need to implement it in the stub implementation. The stub implementation exposes an API identical to the real service: One single function **`create_print_job`** with the same parameters and return type. **`ReportService`** gets the instance of **`FakePrintService`** and uses it without worrying if it is the real implementation or not.

In contrast with the real service **`PrintService`**, the stub will return True if **`create_print_job`** is exactly called with the expected company name.

Otherwise, it will always return False. No network latency, no unexpected errors beyond our control.

You can think of stubs as hardcoded responses for specific parameters. We can make the stub as complex or simple as we want it and control exactly what it returns. We can simulate errors and special conditions that would be very difficult to replicate using the real implementation.

Execute the tests using **`FakePrintService`** as many times as you want to, and you will see the following results confirming our assumptions of determinism and speed.

```
python -m unittest ReportServiceUnitTests
Building report… for Company 1 Co.
This is a fake class
.
--------------------------------------------------------------
--------
Ran 1 test in 0.000s
```

Our test is passing again and **`ReportService`** will execute all its code correctly; except this time we will not see the time delay introduced by the real **`PrintService`**, as we do not depend on network calls anymore or any real business logic unrelated to **`ReportService`**. This test will consistently pass 100% of the time, as long as the stub keeps receiving the expected inputs.

Test mocks

Within the application's code, there are cases where we only call a service in a specific situation. We might only want to generate reports if a given condition is true:

```
#…
 if should_print_report:
     is_printed = self.printService.create_print_job(job_to_print)
#…
```

These use cases are tricky to test. Imagine there is an error in **`create_print_job`**, but execution never reaches to that function because a separate condition causes **`should_print_report`** to always be false.

Existing tests will still report success even if this error exist because a service that is not called cannot fail. To confirm whether the test is successful and **`create_print_job`** is called, we will use mock objects.

For this example, we will use Python's MagicMock, which is defined in the **`unittest.mock`** package, included by default in Python 3. The following tests show how we can create mock objects of **`PrintService`**:

```
import unittest
from unittest.mock import patch
from ReportService import ReportService, PrintService

# …

class ReportServiceUnitTest(unittest.TestCase):
    company = “Company 1 Co.”

    #…

    @patch(‘ReportService.PrintService’)
    def test_create_report_calls_print_service(self, mock_print_service):
        mock_print_service.create_print_job.return_value = True
        self.report_service = ReportService(self.company, mock_print_service)

        report = self.report_service.create_report()

        self.assertEqual(self.company, report[“company”])
        self.assertEqual(mock_print_service.create_print_job.call_count, 1)

    @patch('ReportService.PrintService')
    def test_create_report_not_call_print_service(self, mock_print_service):
        self.report_service = ReportService(self.company, mock_print_service)

        report = self.report_service.create_report(should_print=False)

        self.assertEqual(self.company, report["company"])
        self.assertEqual(mock_print_service.create_print_job.call_count, 0)

#...
```

The highlights of these tests are as follows:

- **`@patch('ReportService.PrintService')`**: This annotation indicates to **`unittest`** that this test should inject an instance of that class into the a parameter of the test function (**`mock_print_service`**).
- **`print_service.create_print_job.return_value = True`**: In this line, we are configuring the mock function **`create_print_job`** to always return True.
- **`self.assertEqual(mock_print_service.create_print_job.call_count, 1)`**: This assertion checks whether **`create_print_job`** was called exactly once. The attribute **`call_count`** in the mock function indicates the number of times the function was called, and the second parameter "1" is the number of expected calls.

Notice how we did not have to create a fake implementation of **`PrintService`**, as the mocking library created it for us. Since we provided the path to the class to be mocked in the **`@patch`** annotation, the library knows exactly what the API for the mocked services should be.

For the sake of completeness, the following code snippet shows us how we can mock a class method in Java using Mockito:

```
// Create mock instance:
PrintService printService = mock(PrintService.class);

// Mock the function createPrintJob to always return true:
when(printService.createPrintJob(anyBoolean())).thenReturn(
true)
```

Stubs versus mocks

In many development teams, you will hear the terms “stub” and “mock” used interchangeably. Most of the time this is not much of a problem, as both concepts have more or less the same goal.

The difference between a stub and a mock is displayed in the examples of the previous section:

- Stubs are implementations of a class or function that provide pre-configured or “canned” responses to specific parameters. In our example, we hard-coded our stub to always return the same result for the exact same set of parameters (the company field in the report).
- Mocks inspect behavior. In addition to the same pre-configured responses stubs return, mocks allow tests to assert information about the mocked method: The number of times it was called, the parameters it received, and so on. All without having to manually implement these conditions in the mock object.

Other common concepts in the field of *test doubles* -everything related to the creation of fake objects for test purposes– are as follows:

- **Fakes**: They are similar to stubs, but they do offer a semi-working implementation (for example, using an in-memory database instead of a remote database server).
- **Spies** They are similar to mocks, as spy’s assert behavior: Number of calls, parameter values, among others.

More than independent categories, fakes, and spies are like profiles of stubs and mocks: Stubs are fakes if they include a reduced version of the real service's logic, and mocks behave like spies when they assert behavior.

Coverage

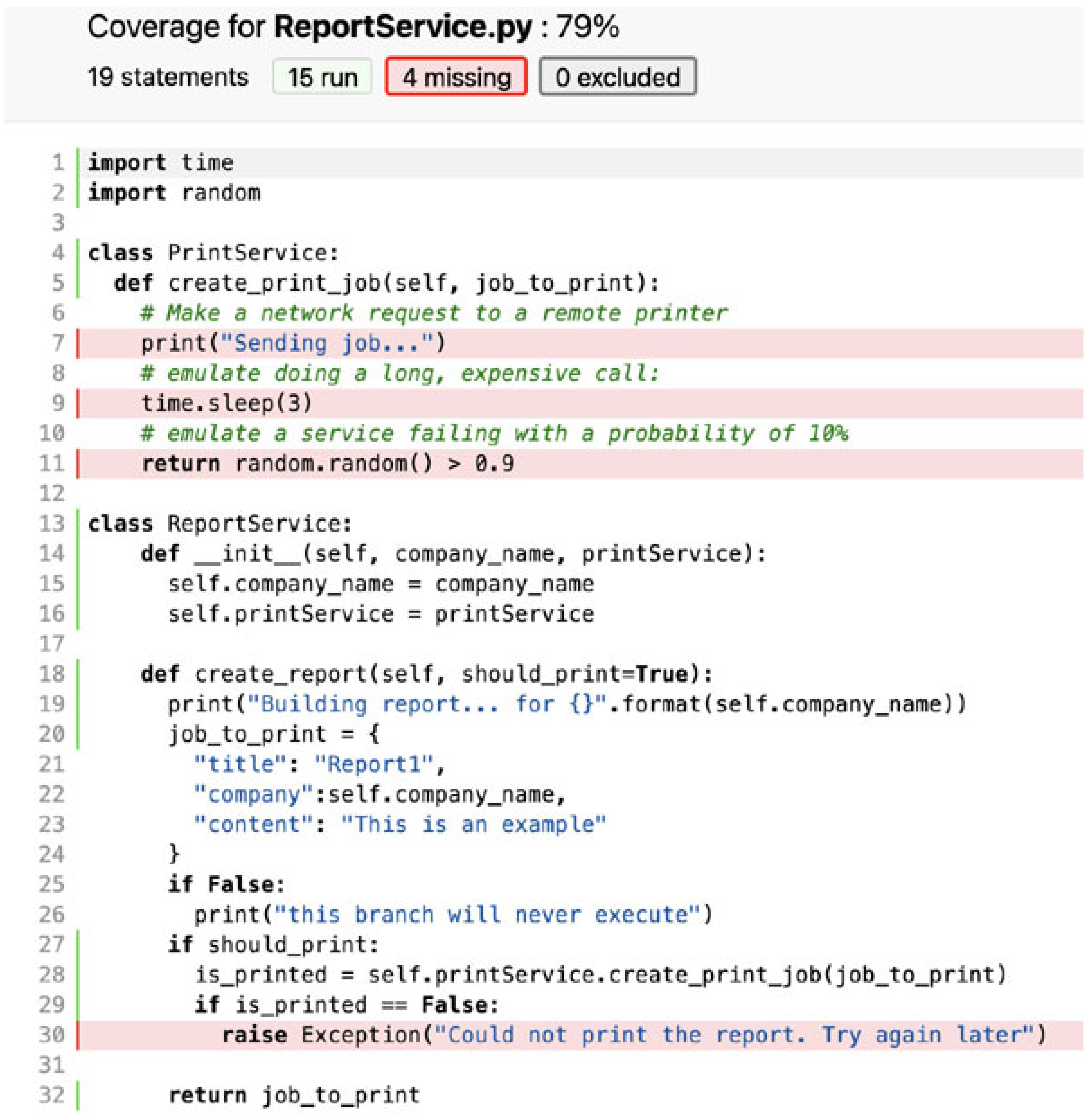
Code execution can be represented as a tree: When the application reaches a condition (IF, ELSE, WHILE, SWITCH, exceptions and so on), execution will be divided in at least two 'branches' or paths: One where the condition is true, and the other were it isn't. This situation leads to developers having to write multiple tests for the same function: one for each execution path.

Coverage is a metric often used by developer teams to measure the quality of their existing automated tests. Each test covers an execution path in the code to be tested; the percentage of lines of code that are in that path is the coverage percentage that the test has.

To understand this concept in a better way, let's run the coverage tool `coverage.py`:

```
coverage run -m unittest ReportServiceUnitTests
coverage html -d coverage_html
```

Figure 5.5 shows the results of the coverage report. Notice the red lines, which is how the report displays lines without coverage:


Coverage for **ReportService.py** : 79%

19 statements | 15 run | 4 missing | 0 excluded

```
 1  import time
 2  import random
 3
 4  class PrintService:
 5    def create_print_job(self, job_to_print):
 6      # Make a network request to a remote printer
 7      print("Sending job...")
 8      # emulate doing a long, expensive call:
 9      time.sleep(3)
10      # emulate a service failing with a probability of 10%
11      return random.random() > 0.9
12
13  class ReportService:
14      def __init__(self, company_name, printService):
15        self.company_name = company_name
16        self.printService = printService
17
18      def create_report(self, should_print=True):
19        print("Building report... for {}".format(self.company_name))
20        job_to_print = {
21          "title": "Report1",
22          "company":self.company_name,
23          "content": "This is an example"
24        }
25        if False:
26          print("this branch will never execute")
27        if should_print:
28          is_printed = self.printService.create_print_job(job_to_print)
29          if is_printed == False:
30            raise Exception("Could not print the report. Try again later")
31
32        return job_to_print
```

Figure 5.5: *Test coverage with non-covered code branches*

Since **`PrintService`** is not being tested (but it is defined in the same module file), all its lines are red. Also, line 30 for **`ReportService`** is marked in red. If we take a look at our existing tests, we will see that none of them is covering the use case where **`should_print=True`** *and* **`create_print_job`** fails (or, what is the same, returns False). All this untested code brings us down to a coverage of 79%.

If we want to expand our coverage by testing the use case where **`create_print_job`** returns False, we can write another test:

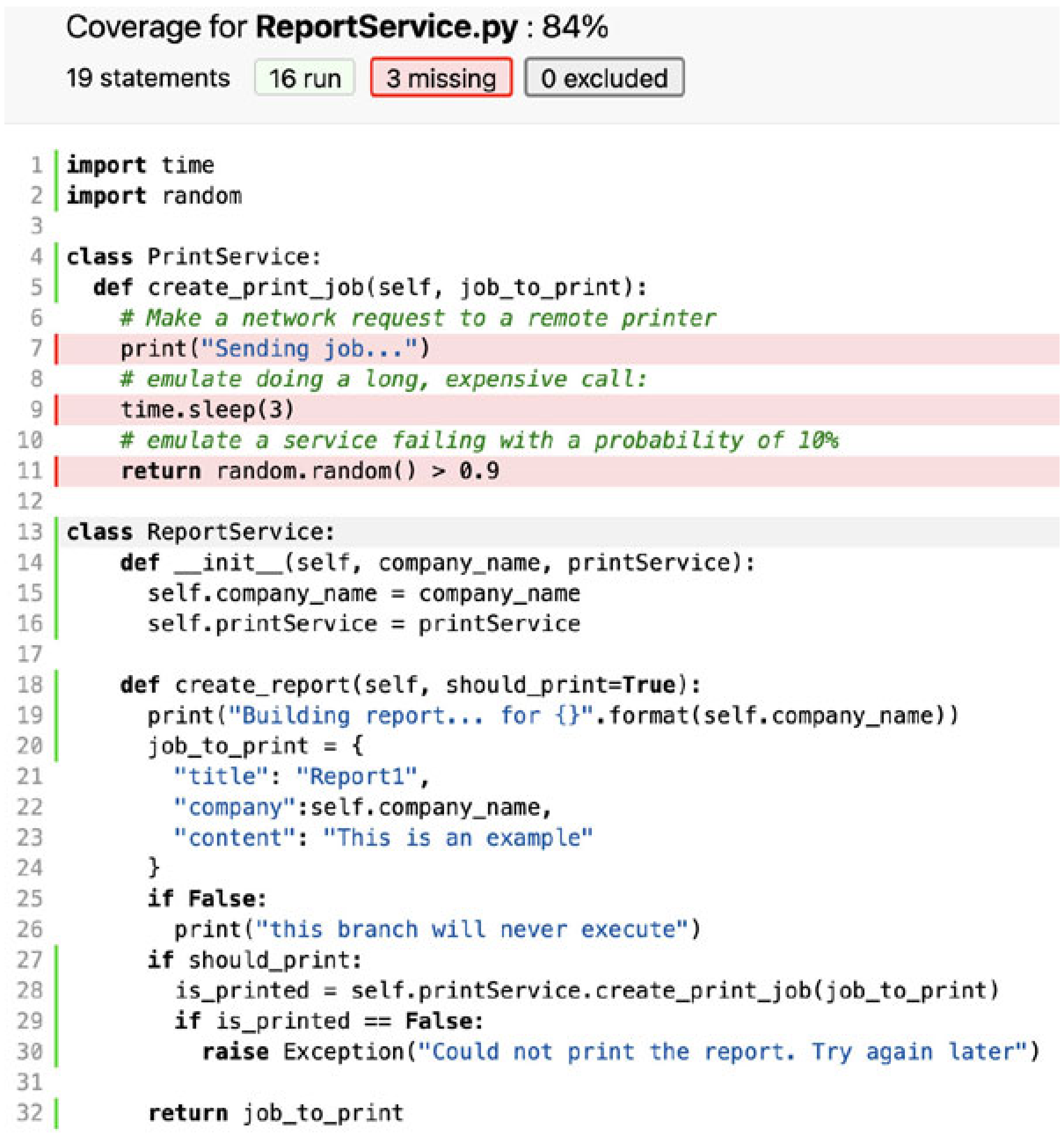
```
@patch(‘ReportService.PrintService’)
def test_create_report_throws_exception(self, mock_print_service):
    mock_print_service.create_print_job.return_value = False
    self.report_service = ReportService(self.company, mock_print_service)

    with self.assertRaises(Exception) as context:
        self.report_service.create_report()

    self.assertEqual(‘Could not print the report. Try again later’, context.exception.args[0])
```

In this test, we use **`self.assertRaises(Exception)`** as context: to tell the test that we are expecting **`create_print_job`** to actually fail. This test then will confirm at the end that the expected message **`-’Could not print the report. Try again later’`** is returned in case of an error within **`PrintService`**.

If we re-run the tests, and check the coverage report in **`./coverage_html/index.html`**, we will see something similar to *Figure 5.6*:



Coverage for **ReportService.py** : 84%

19 statements | 16 run | 3 missing | 0 excluded

```
 1  import time
 2  import random
 3
 4  class PrintService:
 5    def create_print_job(self, job_to_print):
 6      # Make a network request to a remote printer
 7      print("Sending job...")
 8      # emulate doing a long, expensive call:
 9      time.sleep(3)
10      # emulate a service failing with a probability of 10%
11      return random.random() > 0.9
12
13  class ReportService:
14      def __init__(self, company_name, printService):
15        self.company_name = company_name
16        self.printService = printService
17
18      def create_report(self, should_print=True):
19        print("Building report... for {}".format(self.company_name))
20        job_to_print = {
21          "title": "Report1",
22          "company":self.company_name,
23          "content": "This is an example"
24        }
25        if False:
26          print("this branch will never execute")
27        if should_print:
28          is_printed = self.printService.create_print_job(job_to_print)
29          if is_printed == False:
30            raise Exception("Could not print the report. Try again later")
31
32        return job_to_print
```

Figure 5.6: *Test coverage after adding a new test*

Notice that the red mark is gone from line 30, and our test coverage went up to 84%. If our code fails to handle errors in `PrintService`, we now will be able to catch it in tests.

To use coverage or not to use it

As mentioned earlier, many teams use coverage as a measure of code quality. However, as we saw in our example, a score below 100% does not mean we are missing code to test; in our example, the missing percentage is given by code that is clearly not the subject of the test.

There is a high cost to try to achieve 100% coverage, and it brings diminishing returns: Some code has little impact and it is just too hard to test (for example, deeply nested branches), or too straight-forward (for example, getters and setters for classes attributes). Writing tests for this kind of code will increase coverage and maybe catch some very uncommon bugs, but they will rarely find anything critical and they will just bloat your test code base.

While reaching 100% would be amazing, if you plan to use coverage as a quality metric, you will get better results if you aim for a reasonable value between 80% and 90%. Else, developer productivity will be significantly slowed down by having to write too many tests that have little to no value in the task of capturing critical defects.

Test-driven development

Testing is an action that is deeply intertwined with coding. As you build new features into your application, you are continuously testing code you just wrote to confirm it works as expected.

Test-Driven Development (**TDD**) was born out of the idea that all code should be covered by automated tests. The best way to guarantee this test coverage is by first writing tests, then writing code that makes those tests pass.

TDD puts the actual use case before the code itself: By writing the tests first, you make sure you only implement code that will actually be used. When code is not necessary, it becomes really difficult to write tests for it, making TDD a good way to detect unnecessary work.

In a TDD-ideal world, developers will always follow the next three steps when developing new features:

- Write a unittest for the new feature, before you actually write the function's implementation. The test will fail, and that is expected. In fact, TDD experts argue we should always start with failing tests.
- Write just enough code to make the test pass.
- Repeat starting from the first step for the following feature or use case.

In the real world, it is not always possible or optimal to follow these steps in a rigid order. Some features require you to write the code and see how the code interacts with the rest of the application before you can define its interface structure.

The intent of TDD is the correct one, though. It is easier to write unittests at the same time you are building the implementation for a feature, than it is to create "*tech debt*" and try to return weeks or months later -when you barely remember how exactly your code worked- to write those tests. With TDD we introduce quality proactively, instead of reactively (when errors actually happen).

TDD forces us to think about all the different corner cases that a piece of code could encounter. When we think of features from the point of view of a tester, we gain a richer insight into how to make our code more robust and resilient to unexpected input or problems.

Integration testing with Selenium

As we cover most of our code using unit tests, the next logic step is to test how all those units interact with each other.

One way we can test the interaction between functions is to leverage the same unit testing libraries (for example, *JUnit*, *unittest*) and write tests to only cover the integration between a couple of components; if there are more external dependencies than those, we can mock them as we already do in unittests.

Another alternative is to write tests emulating users interacting with a live instance of the application. These tests are called **integration tests**.

Note: Some frameworks have different names for these tests. For instance, to tests that are executed directly in a semi-live version of the user interface, the JavaScript framework Ember.js calls them "acceptance tests".

Regardless of how the tool you are using calls them, integration or acceptance tests are the ultimate tools to verify that the application behaves as a user would expect it to.

In integration tests, we write the code that simulates the steps a real user would execute in the actual application: fill an input field, click on a button, and click on a link to navigate to a different page, among others. Once the test completes all the steps in the use case, it can assert for the existence of certain conditions, visual elements, or text.

For instance, imagine a web application that provides access to the same reporting service we discussed in the previous sections. This web application also provides a page to search for existing reports for a given company. The user interface can be seen in *Figure 5.7*:

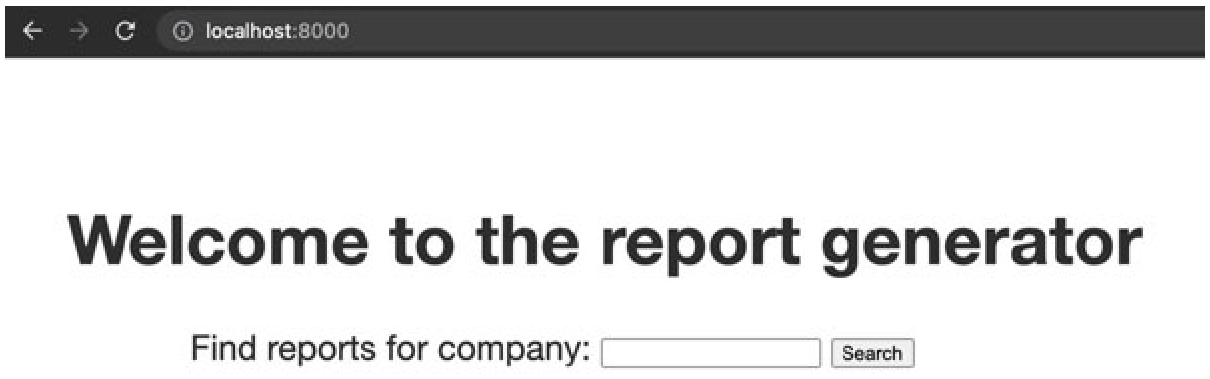
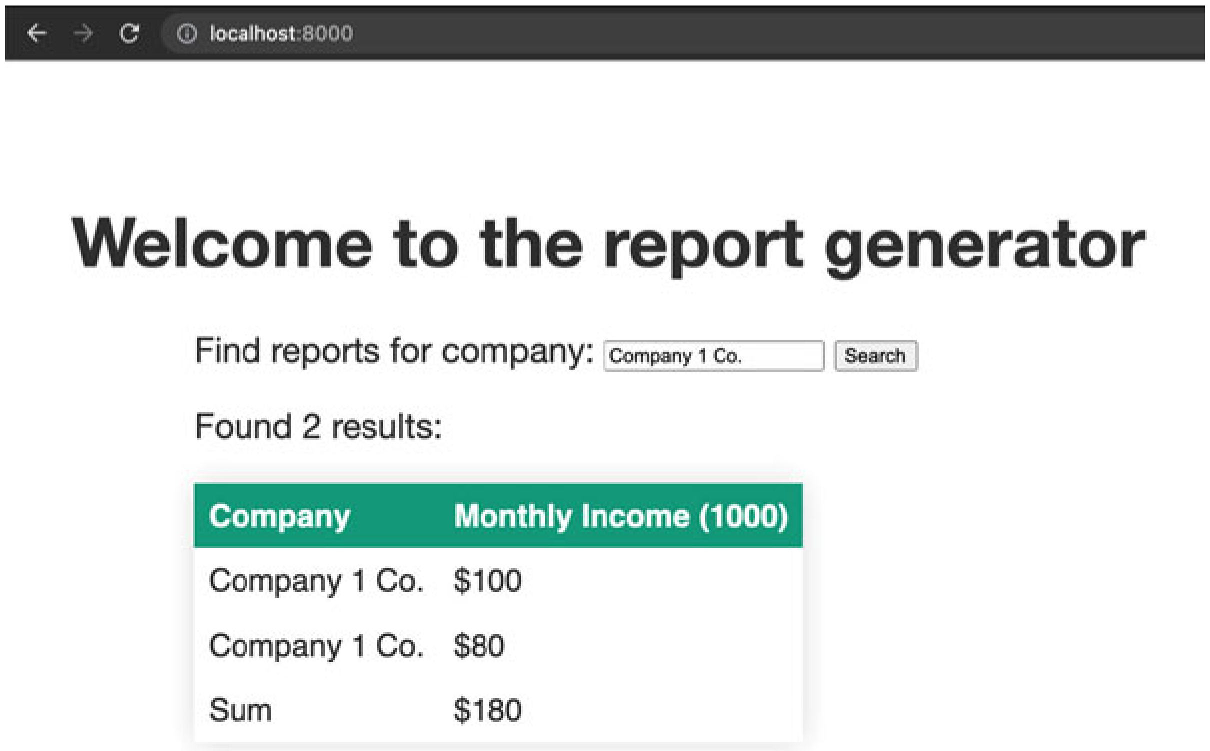


Figure 5.7: *The search page for the Report Generator app*

In order to see the list of reports, a user would have to execute the following steps:

- Open the web application in a browser.
- Fill the search field with a string containing the company's name.
- Click on the `Search` button.

Then, a table containing all available reports for the company will be displayed. *Figure 5.8* shows the end state after executing each step in the use case:



Welcome to the report generator

Find reports for company: Company 1 Co. Search

Found 2 results:

Company	Monthly Income (1000)
Company 1 Co.	$100
Company 1 Co.	$80
Sum	$180

Figure 5.8: *A table containing the results of a search for company results*

In manual testing, a human application tester would execute these steps by hand, and log the end result. We can approach this same action using an automated integration test.

For this section, let us use Selenium with Python. Selenium is a tool that allows developers to write automated tests which run on browsers and simulate the interactions of a real user with a live instance of the web application.

The following code snippet is a Python program using **`unittest`** as the test runner, but using Selenium with a Chrome browser:

```python
import unittest
from selenium import webdriver

class IntegrationTest(unittest.TestCase):
  def setUp(self):
      self.driver = webdriver.Chrome('./chromedriver')

```

```
    def test_integration(self):
        driver = self.driver
        driver.get(“http://localhost:8000”)

        self.assertEqual(“Report Generator”, driver.title)
        elem = driver.find_element_by_id(“company-search-field”)
        button = driver.find_element_by_id(“search-button”)
        elem.clear()
        elem.send_keys(“Company 1”)
        button.click()
        self.assertIn(“Found 2 results”, driver.page_source)

    def tearDown(self):
        self.driver.quit()
```

Again, let’s talk about the highlights in this code. For that, let’s break it up in sections:

```
import unittest
from selenium import webdriver

class IntegrationTest(unittest.TestCase):
    def setUp(self):
        self.driver = webdriver.Chrome(‘./chromedriver’)

```

In the test’s set up, we call **`webdriver.Chrome(‘./chromedriver’)`** to create a browser instance. The string parameter (**`./chromedriver`**) is the path to the browser’s web driver; in this case we use the driver file for Google Chrome, which is local to this test file. Usually, this driver file needs to be downloaded before running the tests.

Then, the test navigates to the application’s URL (*localhost:8080*) and checks whether the page has the correct title, “**`Report Generator`**”:

```
  def test_integration(self):
    driver = self.driver
    driver.get(“http://localhost:8000”)

    self.assertEqual(“Report Generator”, driver.title)
```

The next step is to get a reference to both the input element and the search button. We use the HTML ID attribute to create a reference to them:

```
    elem = driver.find_element_by_id(“company-search-field”)
    button = driver.find_element_by_id(“search-button”)
```

Notice how Selenium’s API is very similar to the DOM API in JavaScript: We have methods to getting elements by ID, by class, or by query selector.

Once we have a reference to these elements in the page, we can delete whatever text might exist in the text input already, add some text to the same input field with the name of a company, and then click on the **`search`** button:

```
    elem.clear()
    elem.send_keys(“Company 1”)
    button.click()
```

As the test executes, you can see the elements in the page within the browser window being updated.

In the end, we assert that the page displays a text message which confirms the search operation was successful. After the test is complete, we close the browser window:

```
    self.assertIn(“Found 2 results”, driver.page_source)

  def tearDown(self):
      self.driver.quit()
```

If we execute this test and all assertions pass, we will see a success message as follows:

```
python -m unittest IntegrationTests
.
----------------------------------------------------------------------
Ran 1 test in 3.051s
OK
```

Once again, notice the time it took to execute the test: Around three seconds. In this case, this is something expected as Selenium tests take longer to run: A new browser window needs to be instantiated, and the test then needs to wait for all resources and network calls to complete at each step of the test. Actually, we can expect these tests to last longer, all things considered. The advantage of these tests, though, is that we get to test the same functionality a user would have access to, end to end.

Defining a test environment for integration testing

Integration tests using real services face the same challenges as unit tests do: Real services can fail, and network latency takes extra time. Just as we used mocks in unit tests to make our tests deterministic and fast, we need to do something similar for integration tests: Create a test environment.

Test environments are live instances of the application, but running with test data. Just as fakes in unit tests, these test environments rely on simpler instances of internal services like databases and calls to external services are stubbed to isolate only services defined within the application.

Some applications are small enough that a new instance of the application can be created before integration tests are executed and teared down once the tests complete. Other teams may decide that they will keep just one instance of the test application running and have integration tests access it remotely.

Test environments have extra advantages: We have full control of the data in the test environment. We can create data for specific test cases which might not be common in production: Reports with missing or wrong information which should trigger validations in certain application modules, reports whose attributes contain text with the maximum allowed character length to help us confirm the UI can deal with handling that condition, and so on.

Simulate a test environment close to production

While test environments give us the flexibility to create data for testing corner cases, it is important that it does not deviate much from the real production environment.

You want developers and testers to work with an environment that is the closest possible to what end users will experience; this is to avoid providing an experience which is significantly different from what developers tested.

Testing and CI/CD

Continuous Integration and **Continuous Deployment** (**CI/CD**) is an automated process that allows developers to deploy each change done to an application with little to no human intervention: A developer commits code changes to a source code versioning system like Git, and the CI/CD pipeline takes care of checking out the code, executing tests, and deploying to a production environment, assuming each step is successful.

Automated tests are a key part of the CI/CD flow: They enable CI/CD pipelines to verify that the code we're about to deploy is mostly free of errors. Without automated tests, the deployment process would have to be paused until an application tester can complete all manual tests. *Figure 5.9* shows a common process for validating each step of a CI/CD deployment:

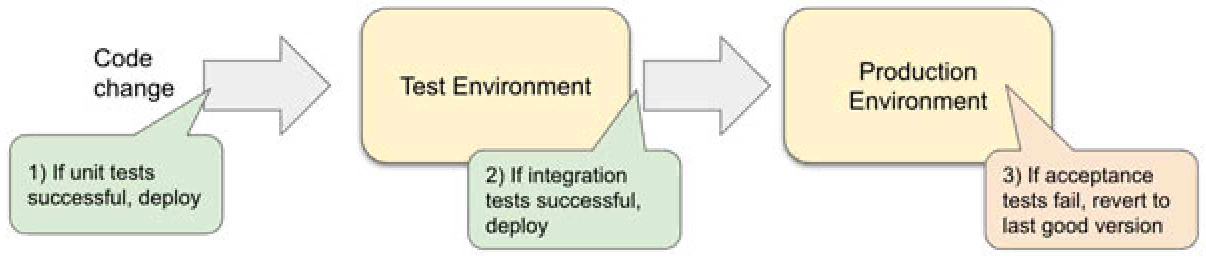

Code change
1) If unit tests successful, deploy
Test Environment
2) If integration tests successful, deploy
Production Environment
3) If acceptance tests fail, revert to last good version

Figure 5.9: *Validation of deployments using automated (green) and manual (orange) testing*

The complete process of CI/CD is a topic we will explore more in depth later in this book.

Other automated tests: Static code analyzers

In addition to unit and integration tests, software developers have other automated tools that help us find quality issues in code. One of such is *static code analyzers*. These tools parse the application’s code and search for specific patterns which that indicate quality issues. Some of those quality issues are as follows:

- Security issues like **Cross-Site Scripting (XSS)**, **Cross-Site Request Forgery (CSRF)**, SQL injection, user/passwords stored in plain text, and so on.
- Unused/dead code
- Deeply nested if blocks
- Calls to deprecated functions
- Bad code practices (lack of validation, unused import statements)

Some popular code analyzers are linting tools like JavaScript’s *eslint* or Java’s *Checkstyle*, *FindBugs*, or *Emma*.

Defining effective test cases

Once we are familiar with all the tools we have for writing automated tests, it is important to think about *what* to test. As we have mentioned repeatedly in this chapter, user requirements are a good starting point to define good test cases. However, we need some extra guidelines to build quality tests which provide value instead of being just an obstacle developers need to bypass to get a deployment.

Defining a single use case per test

Tests should be units of work that are dedicated to verify individual test cases. This rule not only applies to unit tests, but to every type of automated tests..

For instance, if the application provides controls to both create, update, and delete reports, you might feel tempted to run one single test to cover all the three actions. While a test like this could save some time and space, it would also be asserting a very specific use case where the user performs all three actions, always in the same order.

Tests with multiple use cases in a single test make it more difficult to reason about the test results: If there is an error, which action actually failed? Is the error only reproducible after executing each action in the exact same order? Or is the error only reproducible when a subset of the actions is executed?

We should write a single test for each of the actions. Actions like "*update report*" and "*delete report*" might depend on the "*create report*" action, so they cannot be split so easily, but "*update report*" and "*delete report*" can be put into their own tests.

Testing single use cases help us reason about test failures. We know exactly how to replicate the issue so we can fix it later; and by having just the required steps for that very specific use case, we can isolate the exact cause of the defect faster.

Do not mock everything

Mocks and stubs are really useful, but using them in excess can lead to a test set that provides no value at all. Beyond enabling the testing of other functions and classes, mocks and stubs provide no intrinsic value to the tests. Writing too many mocks can lead to tests that, ironically, don't test any real code.

Also, not everything needs to be a mock. Somethings it is easier to write simple stubs with empty implementations than having to configure a test mock library. The simpler your tests are, the easier it will be to maintain them in the long run.

Mocks and stubs should be used *as needed*. Some developers even consider having to use too many mocks to be a *code smell*: It may indicate that the code is not modular enough and it has too many tightly-coupled dependencies.

Note: Search term: code smell

Code smells are red flags in our application that are not problems by themselves but indicate a larger problem lurks nearby

On equal conditions, prefer unit tests over integration tests

Unit tests are faster than integration tests. If a use case can be tested using either of them, we prefer to implement it using unit tests.

If we end up with too many integration tests, the whole test suite might take too long to execute and developers will stop executing it at every code change.

However, if we cannot avoid having integration tests that take too long to run, there are tools we can use to schedule them to run after a deployment: While it is not the best approach, it does help us confirm the deployment went correctly. If an error is found, then we can always revert to an older, more stable version.

Non-functional testing

Some technical areas are critical for software applications: Performance, security, accessibility, among others. Independent teams are created to analyze and test that software applications meet quality standards for these areas. While most backend developers will not perform in-depth testing in most of these areas themselves (and thus are out of the scope of our book), it is important to mention them.

Testing of these non-functional areas are commonly outsourced to external teams, even to other specialized companies. Something that every senior developer understands is that you cannot be expert in every area, so it is important to delegate work to those who are.

Let us briefly discuss some of these areas of non-functional testing to make sure we understand their importance.

Application security and penetration testing

The area of application security includes many sub-areas which experts on the topic assert for software applications. Some of the tests done are as follows:

- **Penetration testing**: Finding and exploiting vulnerabilities in applications to gain access to restricted information or to user accounts with more privileges. Penetration testing can be performed with a mix of manual exploits and automated tools.

- **Thread modeling**: Design a model of all the areas in an application which could be subject of security attacks. Each risk is associated with a criticality of a possible breach and how easy it is to replicate the exploit to assess severity.

Load testing

Load testing is performed to assess how well an application responds to surges in traffic. It also provides development and infrastructure teams of details like:

- In average, how many concurrent users the application can handle.
- What's the maximum number of **queries per second** (**QPS**) our severs support? At what percentage of load do we need to start spinning new servers?
- Which features break first when the load takes the application to its limits?

Understanding how the application behaves when it cannot take any more traffic help us developers to design better architectures, graceful degradation plans, and so on.

Performance testing

Very similar to load testing (some practices even merge both of them under the *performance-testing* umbrella), performance testing allows us to examine how fast and stable the application is.

Performance testing helps us find parts of the application which could work more efficiently and improve metrics like load time.

Accessibility testing

If you want your applications to reach as many users as possible, you want to make sure they are accessible. Accessibility testing checks whether the application's user interface is built in a way that it can be used by any user, regardless of any condition or disability they might have. Many times, for an application to be accessible it means that it must be compatible with adaptive devices like screen readers, special keyboards and mouses, high-contrast mode, zoomed-in, mode, among many more conditions.

Accessibility testing can be performed as a **white box test** to verify that the code is built with all the standards required for the application to be compatible with assistive devices and applications to work correctly.

Black box and manual testing using those assistive devices is also common. An accessibility tester navigates the user interface using these devices and modes, reporting any feature that might be unreachable or unusable.

It is critical to understand what challenges users with specific conditions and backgrounds may face while accessing our application, and the tools they use to overcome these challenges. Even better, you can get users or developers who actually use these assistive technologies in a daily basis to work on your project to better understand their needs.

Conclusion

Testing is a very critical high-level field of software development. As long as we make changes to an application, testing will be required.

Testing helps us find defects in our application. These defects my cause our application to not perform according to our users' requirements and expectations. Testing before and after each deployment is a good process that will help us gain certainty about the application's quality and performance.

As a backend developer, it is your job to write unit and integration tests. In addition to your own testing, you want other developers and specialized testers to review and test your code, as it will increase your code's quality..

Automated tests can check the code at a low, unit-specific level, like unit tests, or it can assert whole features through integration tests. Both types of tests are tools we have to more easily detect and fix errors in our application.

While automated testing is necessary -especially in Agile-based teams, and projects with quick iterations and multiple deployments- it is not a full replacement of manual testing. Manual testing will always provide extra value that is very difficult to find by only using automated tests.

In unit tests, we test the code in isolation, which sometimes is difficult to do due all the dependencies we have between functions, classes, and services. Mocking and stubbing are techniques to isolate specific functions and classes so we can test their business logic without having external dependencies affecting the results of our tests.

As a developer, it is critical you focus in building a robust test set which can be deterministic and run quickly. Concentrate on building tests that bring value to the application, and not just creating tests for meeting metrics goals, like high coverage percentages.

In this chapter, we introduced the topic of application security and how critical it is to assert it is applied correctly for building robust software applications. In the next chapter, we will evaluate how we as backend developers fit in the area of application security .

References

- JUnit's user guide: **https://junit.org/junit5/docs/current/user-guide/**
- Unittest documentation: **https://docs.python.org/3/library/unittest.html**
- Selenium documents: **https://www.selenium.dev/documentation/**
- Mocks aren't stubs, by Martin Fowler: **https://martinfowler.com/articles/mocksArentStubs.html**

CHAPTER 6
Securing Applications

Protecting users' information should be one of the main principles of any software developer. Not only we owe it to our users who trust us with their information; but not taking application security seriously can have serious economic, legal and reputational consequences.

In this chapter, we will review how your application's data might be at risk: What are the ways in which malicious actors can negatively impact your user's data. Then, we will explore simple but effective practices that prevent your user's data -and your own- from being compromised.

Structure

In this chapter, we will learn the following topics:

- **The CIA triad:** Confidentiality, Integrity, and Availability
- **Access Control:** Authentication and authorization
- **Use case:** Implementing basic authentication and authorization for the Pizza Place app
- Federated authorization
- Building security in the application's design
- **OWASP Top 10:** The most common vulnerabilities

Objectives

After reading this chapter, you will have a good understanding of the multiple techniques' backend developers can use to secure their applications.

This is not an in-depth guide to application security or penetration testing. This chapter focuses in the defensive practices any software developer needs to know.

At the end of the chapter, you will have a better knowledge about the following: The intrinsic value of data and how that value could be at risk. How to protect private data from being accessed by unauthorized users? How to protect data from being accessed or modified by users who don't have the right privileges? The best practices used while implementing authentication and authorization checks. How to assess the risks that your application could be exposed to in a better way?

The CIA triad: Confidentiality, Integrity, and Availability

In *Chapter 4, End-to-end Data Management*, we discussed that data is the main asset for any software application. In order for data to keep its latent value, it has to retain certain properties. If any of these is not present, then the data loses its value; it might even become useless.

Among those properties, the most critical are the CIA triad: *confidentiality*, *integrity*, and *availability*.

Confidentiality

All data have an *audience*: A set of actors, human or not, who needs to access or modify the contents of the data. Data should only be available to its intended audience. Confidentiality is the value of enforcing that data is only available to its audience, and not to everyone else.

The audience can be unrestricted or it can include just one person (most probably, the owner or creator of the data). When there are no restrictions on who can read a set of data, we call that level of confidentiality *public*. When the audience includes only a very limited set of actors (only the author, or just a limited set of people), we refer to that confidentiality level as *restricted* or *confidential*.

Publicly facing websites are usually considered to be public, while internal applications or log-in protected applications can be seen as restricted or confidential. The same happens for data like presentations or documents.

Many companies define their own confidentiality levels so they can label all the documents and resources they create. These labels help employees identify when they are dealing with highly-sensitive, restricted data, and act accordingly to protect its confidentiality.

Loss of confidentiality

There are many consequences when a confidentiality level is not respected. Some data’s value is completely based on it being confidential (think of industry secrets, documents describing competitive strategies, and so on) and by losing its confidentiality guarantees it will also lose its value.

Exposing information like personal data to people beyond the expected audience has dire consequences for the owners of the data: From losing a job or having their insurance premiums increase due to discrimination based on an underlying medical condition, all the way to being an easy target for hacking thanks to the close relationship between personal data and access information to multiple services (for example, social security numbers, answers to security questions, and so on).

These kinds of leaks also have huge consequences to us if we fail to protect our users’ information: Loss in revenue, loss of users, reputation damage, legal consequences, and so on.

High profile data leaks and hacks in the past few years have resulted in losses of millions for the businesses who were involved. These events are nothing else but confidentiality being breached and losses have been reported to be as high as $1.6 million dollars.

On a more specific context of software development, certain information depends on being confidential so the application can work as expected: Things like passwords, recovery and encryption keys, security questions, and so on. If any of these restricted things were to be revealed to a wider and unintended audience, the correct behavior and value of the application could be at risk.

Even data that might not seem specially confidential to you, it might be critical for the owner. Confidentiality is defined completely by its context and the relationship it has to its owners.

Integrity

Data has to be *truthful* in order to retain its value. If the data becomes corrupted as a result of being manipulated incorrectly, or due a faulty process, it will lose its value.

For data to retain its integrity, it should only be modified by the audience defined by its confidentiality level, and through the expected processes (for example, if the data was expected to be stored as a binary file, it should be serialized and deserialized with the correct tools).

Data will lose its integrity if it is modified by actors outside of it intended audience; or when it is modified by processes that are not the intended by the owners of the data and the application.

Integrity is not only related to how the data is modified, but to how it *isn't*. Data at rest shouldn't change or be corrupted by external elements like malicious actors (for example, hackers) or faulty infrastructure (for example, bad hard drives with no redundancy).

Loss of integrity

Data you cannot trust is data that has no value. The problem with integrity loss is that it not only affects the corrupted data, but all data related or even just locally close to it.

Think of this: If someone gave you a bag full of candies, but that person told you "*one of these is contaminated, but I don't know which*", would you risk eating any of the candy? Issues with data integrity are not different; you can try to find which part of the data is correct and which isn't, but the overall trust of the whole data set is considerably reduced.

From a business perspective, incorrect data can lead to taking bad business decisions, and even direct customer and financial loss.

Integrity loss also affects applications themselves. If the data that conforms the application database contents, configuration files, or even source code is incorrect, all the data produced by the application will also be incorrect.

Availability

Data needs to be available when its users and owners need to access it. Information that cannot be accessed has no value.

While it’s understandable that data might be temporarily unavailable at specific moments (for example, while the application is under “*maintenance*”), users expect a continuation of service: If the data is unavailable, it has to be for very short periods and not at critical moments. For instance, commercial users expect a bank application to be up at least during office hours, when they conduct business; any downtime beyond that will be unacceptable.

Loss of availability

We don’t tend to relate availability with application security, yet the most common “*cyber-attacks*” are **Distributed Denial-of-Service** (**DDoS**), which consist in saturating the incoming traffic for an application with artificial requests, causing it to reach its limit and break down. Again, each of those incidents ends up in millions in lost revenue simply because users cannot use the application. Users who cannot use our application are users who will take their business somewhere else and never return.

Access Control: Authentication and authorization

All the data within an application needs to be secured according to its confidentiality, integrity, and availability requirements. The most common violations to the CIA triad happen because users perform unauthorized actions on an application’s data; so it’s critical to set an *access control* workflow in place.

Access control can be broken down into *authentication* and *authorization*. When a new user tries to access the application, we need to make sure they have the right authorization level to do so; for that, we need to know *who* this user is.

Authentication

Authentication consists in verifying that whoever tries to access a specific set of data or perform some action is who they say they are. We define users identities within an application context through an *identifier* or *“ID”*.

An identified can be any piece of data that is uniquely related to the user. For instance, an identifier could be the user’s name. However, this identified is not particularly good. The problem is that one person can have the same name as other people. What happens if we have two John Smith? Who is which? So, we move on to define a *unique* identifier. Better identifiers can be a user-defined username, an email, or a numeric ID like a social security number.

However, we cannot validate a user's identity only using their unique ID. The problem is that most of the information used as identifiers is not really private. People who know your email or username can try to *impersonate* you and see all your personal data in the application. We need a *secret* that is shared between the users and the application to better assess the identity of the user.

Identification: Username and password

By defining a secret phrase or word who only the user and the application know, we can perform *authentication*. The unique combination of a username and a password becomes a way of identification by itself.

Problems with passwords

Many factors cause passwords to be not as effective as they should: In average, people use more than one application that requires user authentication. Ideally, people should create a unique password for each application, but in reality, they tend to use very simplistic passwords; or they end up using the same password for all accounts. If one of those systems is compromised, the rest of the applications will be as well. Passwords can be stolen, guessed, or inferred from other personal data. People tend to use easy passwords "*123456*", "*qwerty*", or "*password*"; users also tend to choose passwords based on easy-to-remember (but also easy to guess) information like the dates of birth of themselves or their direct family members. While it is human nature to try to use easy or familiar information for our passwords, malicious users understand these bad habits and use them against regular users.

Malicious users have automated tools that can perform **brute-force attacks** trying each combination of characters possible or **dictionary attacks** trying multiple combinations of words commonly used in passwords, making guessing a password an easy process if there are not enough password protections.

Password policies need to balance enforcing passwords that are not too hard to remember but not too easy to guess or deduce.

Best practices for passwords

The U.S. **National Institute of Standards and Technology** (**NIST**) describes in its "*Digital Identity Guidelines*" document some guidelines for memorized secrets or passwords:

- Enforce a minimum password length of 8 characters for user-chosen passwords, or 6 character length for randomly-generated numeric PINs.
- All printing ASCII and Unicode characters as well as the space character should be acceptable.
- Authentication services may replace multiple consecutive space characters into a single space character, to account for user typing errors.
- Random password generators should be cryptographically strong (a list of advised generators is provided in the NIST document).
- When creating new passwords, the authentication service should compare the password with a deny-list consisting of commonly used or compromised passwords. This list contains dictionary words, or context-specific words like the user's username.
- No hints should be provided to unauthenticated users about the password. This can provide helpful information to malicious users to guess existing users' passwords.
- Limit the number of failed authentication attempts to not more than 100. This is to prevent online brute force or dictionary attacks which aim to guess the password. The number of tries shouldn't be small enough that causes users to be locked out due to human error.
- After reaching a specific number of failed authentication attempts, impose a wait of 30 seconds up to an hour. This is also to discourage online guessing attacks.
- Consider the use of tools like CAPTCHA which discourage automated access to the application.

- Allow copy-paste into the password field. This encourages users to use password managers which increase the likelihood of users creating strong and unique passwords.
- Don’t require other composition rules like requiring special characters or numbers. Don’t enforce periodical password updates. Research has found that these limitations do little to prevent password leaks or guessing, while at the same time, they are a common cause of users having bad password practices (for example, using repeated or easy to guess passwords).
- Prefer “*pass phrases*” consisting of multiple words separated by spaces, over single words. They are harder to guess by brute-force attacks and easier to remember by users.
- Passwords should be encrypted in traffic (for example, while being sent through the network) and in the rest (while being stored in a database).
- Passwords should never be stored in plain-text. A strong hashing algorithm must be used to create a hash derived from the password. The application then can compare the generated hash with the stored hash to confirm the password is correct.

Notice that many of these guidelines seem to contradict common password policies, especially the one about not enforcing special characters or other patterns in the password. These patterns don’t have as big of a positive impact as other factors as the password’s length or implementing *multi-factor authentication*.

There are many considerations for password management. It is not easy to enforce all these rules by ourselves, which is why most of the time *we should prefer using existing authentication providers who are specialized in application security instead of implementing authentication by ourselves*.

Identification: Multi-factor authentication

The strength of user name and password identification can be strengthened by requiring the user to provide more information in addition to the password. This information is usually a *temporary, single-use code* that is sent to the user through a communication channel which *only the user should have access to*.

A simple solution is to associate the user account with either a phone number or an email address. The user will receive the temporary code through an email or through a text and they will have to provide it along with the user name and password to complete the authentication process.

This approach is effective because it requires little extra effort from the user (for example, they don't need to remember a second password), and phone numbers and email addresses tend to be personal and have their own security enforcement measures.

The idea is to create extra layers of security that don't put an undue burden on the users, else they will use workarounds (like having to write the password in a piece of paper) which might compromise the security of the whole application.

Note: Multi-factor authentication is not perfect. Malicious users have found ways to hijack the code sent to emails or mobile phones; they also have used social engineering attacks to fool people into providing these secrets to them.

It's by the combination of many imperfect security measures that we achieve a better, layered solution.

Single Sign-On

One effective way many companies implement to reduce the number of passwords a user needs to remember is to administrate the authentication process to all its internal applications through a **single sign-on** (**SSO**) service.

SSO services redirect users to an authentication form only once: when they first try to access an application that is part of the organization. Once the user is authenticated, a security token is shared with the rest of the applications within the same organization, allowing the user to access them without having to log in again.

SSO requires an extra investment in infrastructure, but it makes users' lives easier while allowing them to maintain secure password policies.

Authorization

Once a user is correctly authenticated, the next security step to take before they can access restricted data is to guarantee that they have the adequate privileges.

Most privilege-based authorization is done as a function of multiple factors: The identity of the user attempting to access the data, the data to be accessed and the type of action to be done on the data (read, edit, write, or delete).

The following table displays an example of how these factors combined define authorization levels for different users:

User	Resource ID	Read	Edit	Delete
User 1	1232	☑		
User 2	1232	☑	☑	☑
User 3	1232	☑		

***Table 6.1:** Example of access permissions for multiple users over the same data resource*

In *Table 6.1*, we can see an example of a resource with ID **1232**, which is owned by **User 2**, but other users have **read** privileges over that same resource.

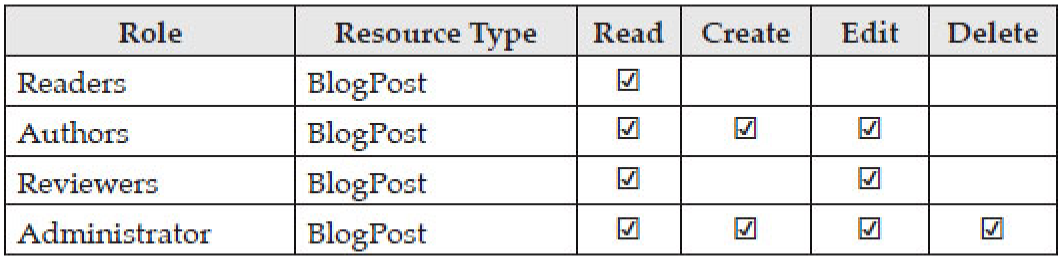
Directly assigning privileges to users can be challenging, as an application can have hundreds or thousands of users. To handle these permissions in a better way, users tend to be assigned to roles and groups which share the same level of access over specific assets.

Roles and groups

Not all users have the same privileges over a specific data record. Typically, a user should only see their own data; but data can also be owned by multiple users.

Users are assigned to *one or more roles* or groups. Each of these roles or groups is linked to a set of privileges. Privileges have a *transitive property*: All users that are part of a user or group have the same privileges as the group does.

For instance, we can create a role-based privilege map:



Role	Resource Type	Read	Create	Edit	Delete
Readers	BlogPost	☑			
Authors	BlogPost	☑	☑	☑	
Reviewers	BlogPost	☑		☑	
Administrator	BlogPost	☑	☑	☑	☑

Table 6.2: *Example of mapping roles to a resource type. In this case, it defines roles for a blog application.*

In *Table 6.2*, we show the privileges for roles in a blog application. Authors can create,

read, and edit, but only administrators can delete. This is to allow moderators to review and control the types of blogs that are created in the application.

Any user assigned to a *reviewers* role or group will be able to read and edit the blog post. We don't need to give read or edit privileges to each user individually, making it easier for administrators to make changes to permissions, if needed.

Remember that a user can be a member of multiple roles or groups. In case a user is assigned groups that have conflicting privileges, the user will gain the union of all their role's prvileges. For instance, take a user who has both the *author* and *reviewer* roles; if they were to try to create a post, that action would be authorized, even if that privilege is not available to *reviewers*. Think of it as applying a Boolean "*OR*" operation on all roles: If the privilege is approved by at least one of the roles, then it should be *authorized* overall. This is not a hard rule, though, and you can choose to have exceptions or deny lists which explicitly forbid a user or role to access specific resources.

There is a chance that, as a backend developer, you will never have to configure these authorization settings yourself. However, it's important to understand how applications typically handle privileges because you will be in charge of enforcing validation that allows or denies access to specific users based on their role or individual permissions.

Least privilege principle

When you're designing the authorization for each role and user, it's easy to look at specific groups or people like internal employees and think "*let's give them access to actions they don't need right now but might need in the future*." Even if we fully trust in their good intentions, when assigning authorization privileges to people, the best practice is to give them the *lowest level of authorization* they need to perform their work.

Think about all the movies about ultra-secret projects. All characters involved in the project always work at a *need-to-know* basis, which means each member in the project only knows the absolute minimal information they need to work in their project; if they are compromised by the enemy or they turn out to be double agents, the potential damage would be mitigated by the least privilege principle..

While designing an authorization plan, give users only the privileges they need to achieve their immediate goals.

Use case: Implementing basic authentication and authorization for the Pizza Place

While many of the features provided by the Pizza Place are intended to be publicly available for anyone online, some specific actions should only be performed by authorized personal working at the Pizza Place. In this section, we will do a step-by-step walk-through to understand how to build application security policies that protect the internal data of the Pizza Place.

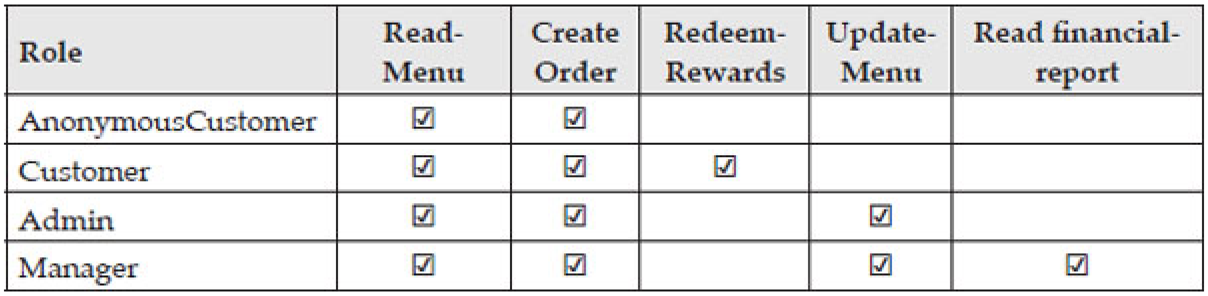
Identify user roles

As we recollected requirements for the Pizza Place application, we recognized different types of users:

- **Anonymous customers**: People who can access the Pizza Place website, see the menu, and even create a one-time order.
- **Registered customers**: People who have created a user account with the Pizza Place who can read the menu, create orders, see their own previous orders, and keep track of loyalty rewards, and so on.
- **Administrative users**: People who work at the Pizza Place and can update the menu and see the orders for all customers. Executive users:

- **The owners of the Pizza Place**: People who can not only see all the data in the application but also get financial reports.

The first step is to identify these users, and create an *access control table*: A table containing all the user roles and the privileges each have. *Table 6.3* shows an example of the access control privileges we're defining for some actions in the Pizza Place app:



Role	Read-Menu	Create Order	Redeem-Rewards	Update-Menu	Read financial-report
AnonymousCustomer	☑	☑			
Customer	☑	☑	☑		
Admin	☑	☑		☑	
Manager	☑	☑		☑	☑

Table 6.3: *Access control table for the Pizza Place app*

This table will help us verify that users assigned to a specific role can only perform the actions they are allowed to.

Building an authentication service

As we've done in previous chapters, we want to find a solution for the Pizza Place app which balances the low traffic the application will get at the beginning and its flexibility of scaling as needed.

In the following sections, we will discuss how to build a user management and authentication module for your application. It's important to notice that this approach has some downsides (which we will discuss later) and it may work only for very simple applications like the one we're building right now. We will go through the process of building the authentications service mostly as an illustrative example for us to understand how these services typically work.

Define user management storage

Users need to be stored in a database. We can store the list of users in the same database we will use to store all business logic like orders, menus, or ingredients. Considering we will have just one server for out database, this approach works well enough for this use case.

In *Figure 6.1* we can see an abstract visual representation of our database infrastructure and the contents of the *Users* table:

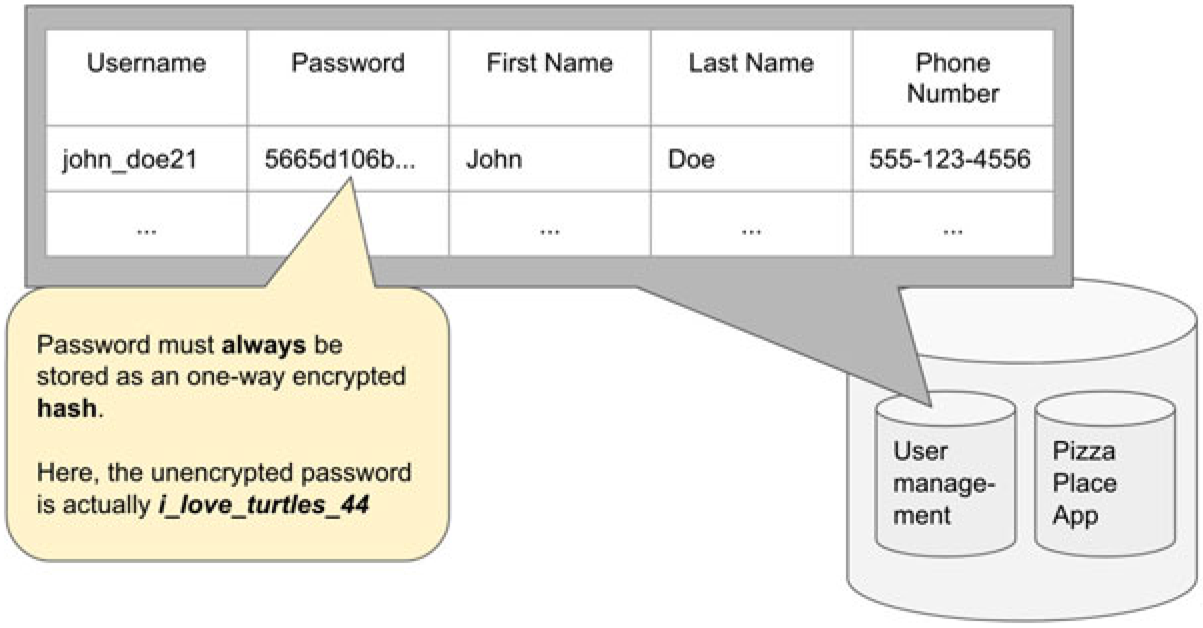

Username
Password
First Name
Last Name
Phone Number
john_doe21
5665d106b...
John
Doe
555-123-4556
...
...
...
...
Password must **always** be stored as an one-way encrypted **hash**.
Here, the unencrypted password is actually ***i_love_turtles_44***
User manage-ment
Pizza Place App

Figure 6.1: *Infrastructure for a simple user management database*

Keep in mind the things we mentioned in previous chapters about concentrating all your data in one single server: Not the best practice, but sometimes it's good enough based on the application's profile and budget.

Notice that *Figure 6.1* shows a table definition where we store all user-related data in the same table. This is only for the purpose of this example, and you should feel comfortable breaking down all the user information in as many tables as you need.

Define password-related protections

Also notice from *Figure 6.1* that we *always* store passwords as *one-way* encrypted hashes. We make emphasis on *always* because this is really, really, really important, and in *one-way* because we don't want passwords to be reversed to their pre-encryption text form.

Figure 6.2 displays a typical sign-up flow where we store the hashed version of the password in the database for the first time:

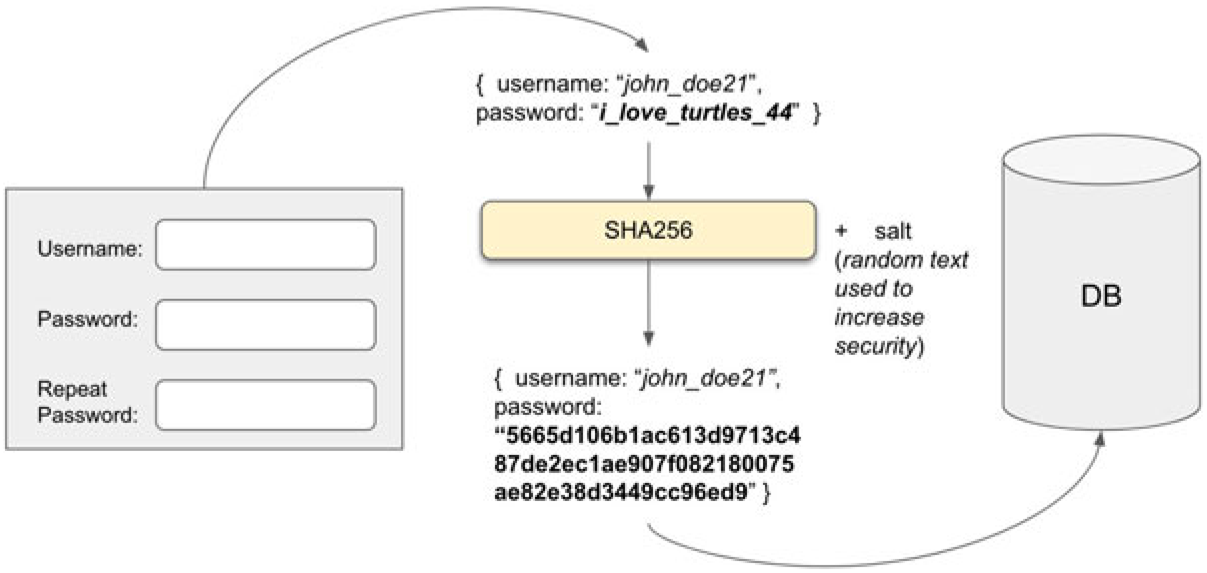

{ username: "john_doe21",
password: "i_love_turtles_44" }
Username:
Password:
Repeat Password:
SHA256
+ salt (random text used to increase security)
{ username: "john_doe21",
password:
"5665d106b1ac613d9713c4
87de2ec1ae907f082180075
ae82e38d3449cc96ed9" }
DB

Figure 6.2: *Sign-up flow: Create a new user*

There are a couple of reasons why we want to store passwords as hashes that cannot be decrypted: If a database is leaked and passwords are compromised, attackers have no way to reverse these hashes back to the real passwords. Even malicious actors *within* our own organization like a disgruntled employee or a malicious provider, who may have authorization to read the database, will not be able to access the real passwords.

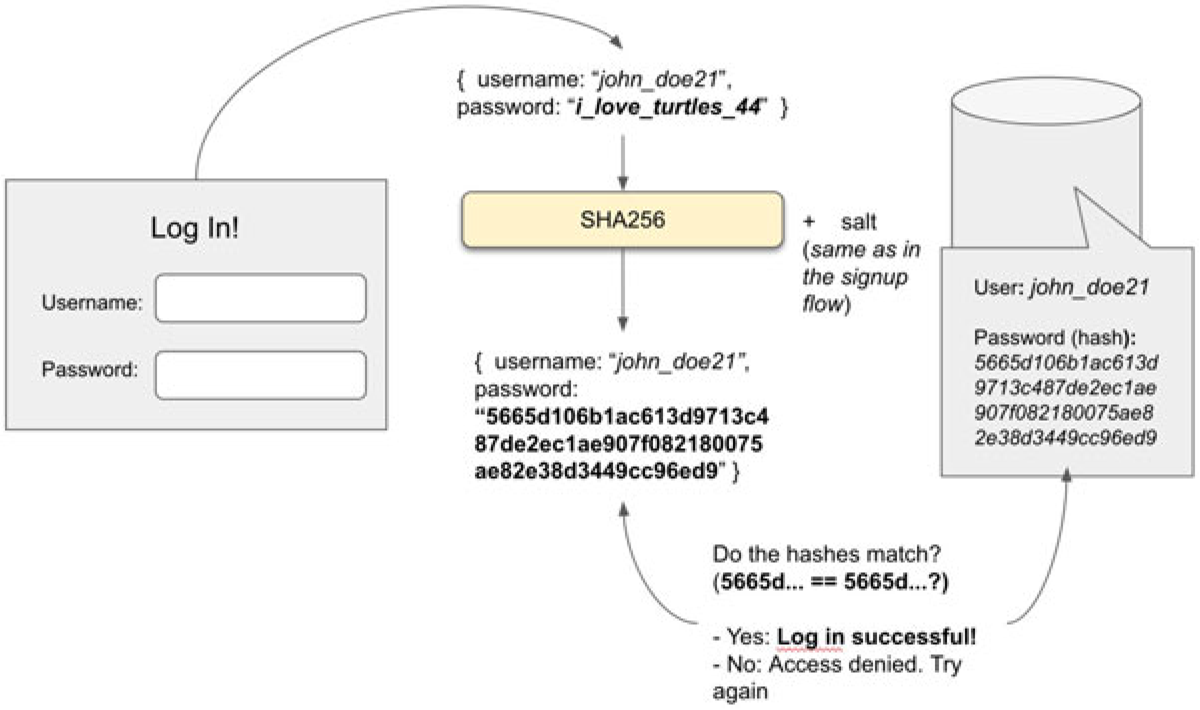
Note: In the past, algorithms like MD5, SHA-1, and SHA-2 have been used to produce password hashes. These have been deprecated due the rapid advancement of computational resources and see how easy they are to decrypt now.

The most common algorithm used for this encryption at the moment is SHA-256, but it's important for you to do your own research to find out what's the best algorithm to use at the moment to build your application.

If we cannot *decrypt* the hash, how are we supposed to authenticate? If the hashes are exposed, are they not the same as the passwords themselves being exposed? Let's evaluate answers for these questions.

We don't need to decrypt the hash for the login flow. When the user submits the login form, we can hash the authentication password. We then take this hash and *compare it with the hash that was stored in the database when the* *user first signed up*. Since we used the same algorithm and the same parameters, if the plain-text password is the same, it should generate the same hash value.

Figure 6.3 shows the flow where we compare the hash for the incoming password with the hash already stored in the table:



Log In!
Username:
Password:
{ username: "john_doe21", password: "i_love_turtles_44" }
SHA256
+ salt (same as in the signup flow)
{ username: "john_doe21", password: "5665d106b1ac613d9713c487de2ec1ae907f082180075ae82e38d3449cc96ed9" }
User: john_doe21
Password (hash): 5665d106b1ac613d9713c487de2ec1ae907f082180075ae82e38d3449cc96ed9
Do the hashes match? (5665d... == 5665d...?)
- Yes: Log in successful!
- No: Access denied. Try again

Figure 6.3: *Login flow: Compare the hashes*

Now, if the hash is compromised, in order for a malicious actor to use it to impersonate the owner of the hash's password, the malicious user would have to bypass the hashing function in the login module. Otherwise, the hashed string would be hashed again, resulting in a third hash value and causing the authentication to fail. For most cases, it's practically impossible to bypass just the encryption function, so having a hash is not the same as having the plain-text version of the password.

The following Java class shows you how to create a *SHA-256* hash from a string, and run some simple tests to assert that the same hashes are generated by using the same passwords:

```
import java.security.MessageDigest;
import java.nio.charset.StandardCharsets;
import java.security.NoSuchAlgorithmException;
```

```

public class Main {
  private static String bytesToHex(byte[] hash) {
      StringBuilder hexString = new StringBuilder(2 * hash.length);
    for (int i = 0; i < hash.length; i++) {
      String hex = Integer.toHexString(0xff & hash[i]);
      if (hex.length() == 1) {
        hexString.append(‘0’);
      }
      hexString.append(hex);
    }
    return hexString.toString();
  }

    private static String hashPassword(String password) throws NoSuchAlgorithmException {
     MessageDigest digest = MessageDigest.getInstance(“SHA-256”);
    byte[] salt = “this is a salt string”.getBytes();
    digest.update(salt);
    digest.update(password.getBytes(StandardCharsets.UTF_8));

    byte[] encodedhash = digest.digest();

    return bytesToHex(encodedhash);
  }

      public static void main(String[] args) throws NoSuchAlgorithmException {
```

```
    String realHash = hashPassword(“Hello world!”);

     boolean isEqual١ = realHash.equals(hashPassword(“Hello world!”));
    boolean isEqual٢ = realHash.equals(hashPassword(“Wrong Password!!”));

    System.out.println(“Using the right password. Success? “ + isEqual١);
    System.out.println(“Using the wrong password. Success? “ + isEqual٢);
  }
}
```

The actual encryption happens in the following snippet:

```
     MessageDigest digest = MessageDigest.getInstance(“SHA-256”);

    byte[] salt = “this is a salt string”.getBytes();
    digest.update(salt);
    digest.update(password.getBytes(StandardCharsets.UTF_8));

    byte[] encodedhash = digest.digest();
```

First, we get an instance of the **`SHA-256`** encryption function. We call the function **`digest.update(byte [])`** twice to pass two arrays of bytes which will be encrypted:

- **The salt**: This is an arbitrary string that adds strength to the encryption. The salt’s bytes are mixed with the bytes from the plain-text password. The combination of password and salt creates the hash, and the hash cannot be reproduced without having both. Think of the salt as a secret owned only by the application, just as the plain-text password is a secret owned only by the user.

- The plain-text password.

Since `digest.digest()` returns an array of bytes, we use the `bytesToHex` function to convert it to a string containing hexadecimal characters.

The `main` function goes ahead and makes it clear that hashing the same string with the same algorithm and salt will result in the same hash.

Note: Password recovery

One downside of not being able to decrypt a password is that we cannot provide real users with their own passwords if they forget them. The best we can do is reset the password by replacing the existing hash with one generated from a new plain-text password, and give the new password to the user so they can log in and update the password again themselves.

Build web forms for signup and login

A large area of authentication's implementation happens in the front-end, so it's necessary to build web forms that will allow the users to submit their username and password to the application. The front-end should also enforce redirecting unauthenticated users to login routes (or any other public route) when they try to access restricted pages. Specifics on building these forms are out of the scope of this book.

As backend developers, we would need to implement *HTTP sessions* where we store all the information related to a successful authentication. Remember that HTTP is stateless, so a session token needs to be passed in every request to identify an authenticated user.

A session data record is created for the user when the user logs in successfully. The session data is stored under a session ID. The *session ID* can be stored in memory or in a database. A cookie or token with the session ID is created for that request.

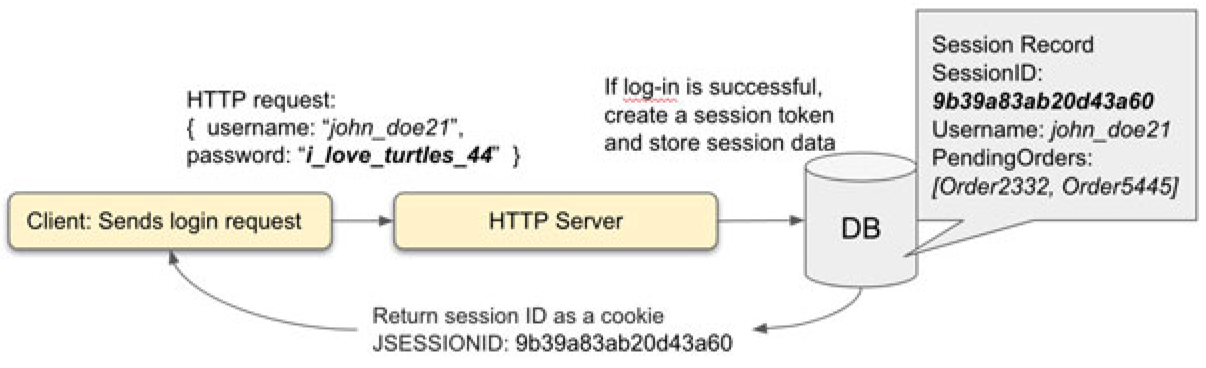


Figure 6.4: *Session ID creation flow on log-in*

In *Figure 6.4*, notice that we use the term JSESSIONID as the session's ID cookie name. This is a name commonly used in Java's HTTP session cookies. Other common cookie names for session tokens are PHPSESSID for PHP or *connect.sid* for Express. However, all servers allow you to configure this cookie's name.

After the client receives the response for a successful authentication, they are in charge of passing the session cookie that contains the session ID in every request. If the cookie is missing from the request, the server will treat it as a non-authenticated request. However, most browsers or web viewers will take care of remembering and passing all set cookies back and forth. *Figure 6.5* displays a visual example of this process:

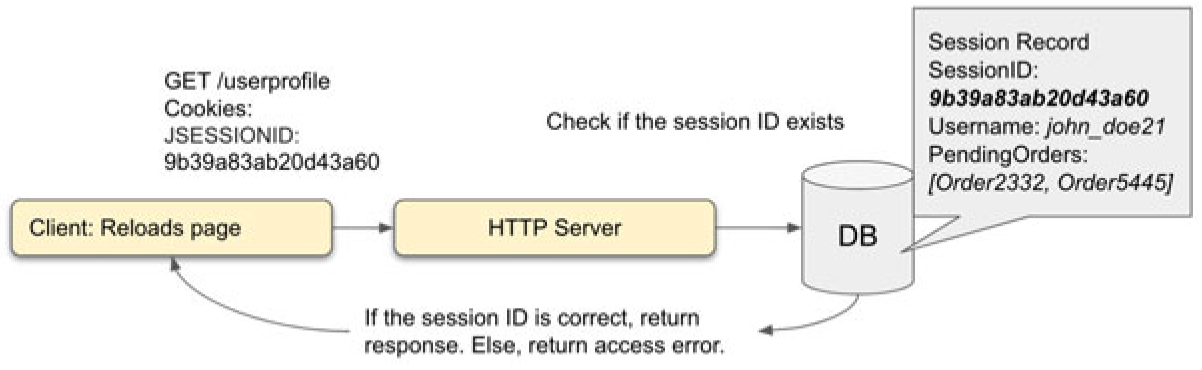


Figure 6.5: *Session ID creation flow on log-in*

Note search term: HTTP cookies

Cookies are small pieces of data which are passed back and forth between a client's browser and the server. Cookies are commonly used

to store information that needs to be sent in each request, like session cookies, authorization tokens, or user preferences like language.

Some servers choose to store the user's data directly in the cookie while others prefer to store the user's data in a database and only use the cookie to store the session ID to retrieve the record. There are multiple trade-offs to these approaches, but most of the time it's harder to store all the session data in the cookie, as it involves cryptographically signing it to make sure a malicious user hasn't tampered with it, and limiting the amount of data it can store, in order to not slow down traffic by adding too much information in each request.

Storing information in a user's session is straightforward in most backend libraries. For instance, we can access a user session in Java servlet-based servers as shown in the following code snippet:

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

 // **

     public void doGet(HttpServletRequest request, HttpServletResponse response){
   // …

   String uname=request.getParameter("userName");

   HttpSession session=request.getSession();
   session.setAttribute("uname", uname);
 }
```

We can do the same using Node's Express library **`express-session`**, as shown in the following code snippet:

```
app.get('/',(req,res) => {
    session=req.session;
    if(session.userid){
        res.send("Welcome! The user is logged in");
    }else
    res.sendFile('views/index.html',{root:__dirname})
});
```

In these examples, the session storage works as a key-value store, allowing us to set and read simple string values.

Apply authorization controls

Regardless of how it will be stored, the user session data can contain as much information as needed to identify the user: Username, email, name, link to a profile picture, and so. Among this session data, authorization-specific data should be included.

Translate a high-level authorization map to implementation details

Until now all the privileges we defined were high-level descriptions of ACTION+RESOURCE like *create order* or *update menu*. We need to translate these abstract privileges into actual implementation details.

We can map privileges into specific things like URLs or routes and functions.

The mapping of privileges to URLs or routes happens in the front-end of the application. For instance, only users who have the role of an admin or a manager should be able to access **`/update-menu.html`** or its equivalent route in a mobile app.

If the front-end has a web server proxying all requests, only authorized users should be able to successfully make requests to them. An example of this can be seen in *Figure 6.6*, in which clients who want to update a menu can send a *POST* ***`/update-menu.html`*** request through an HTML form instead of sending a *PATCH* ***`/api/menu/123321`*** directly to the API. Requests coming from users with unauthorized roles should be denied, as shown in *Figure 6.6*:

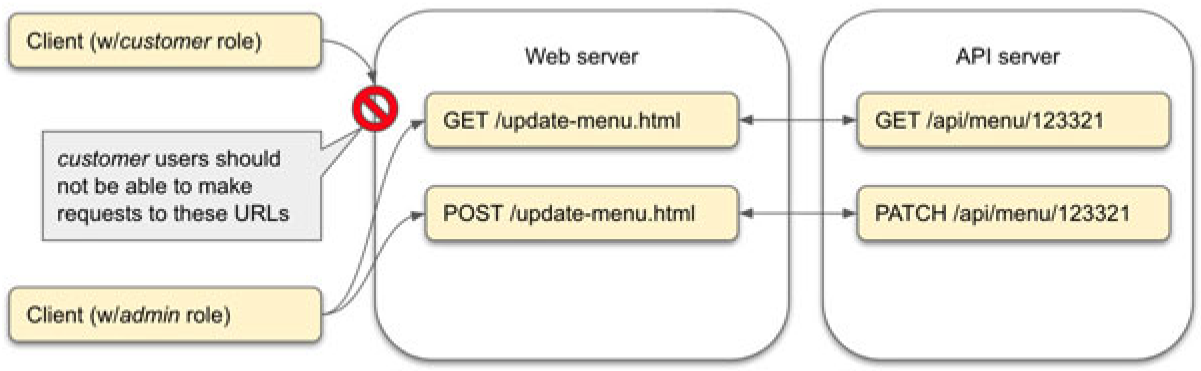


Figure 6.6: *Authorization at the URL level*

Some servers provide configurations that administrators can update to enforce this kind of mapping. For instance, using Java web applications, we can create the following sever configuration to define resources, their URLs, HTTP methods, and the roles that can access them:

```
<security-constraint>
    <web-resource-collection>
           <web-resource-name>Protected Area</web-resource-name>
        <url-pattern>/jsp/update-menu.html</url-pattern>
        <http-method>PUT</http-method>
        <http-method>DELETE</http-method>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>admin</role-name>
    </auth-constraint>
</security-constraint>

<security-role>
    <role-name>customer</role-name>
</security-role>
```

```
<security-role>
    <role-name>admin</role-name>
</security-role>
```

Configurations like these allow us to configure access control for both URLs and for individual HTTP methods: We can allow all users to do **GET** requests to most pages, but no **DELETE**, **POST**, or **PATCH** unless they have the right role and privileges.

However, not all access control checks map directly to a URL or route. If the authorization logic is more complex and requires you to check multiple things beyond roles (for example, actions that the user would have to complete before accessing a feature, like validating their emails), we can always do manual authorization checks in the code. The following code shows how you can manually authorize a function call for Python's web framework Django:

```
from django.core.exceptions import PermissionDenied

def updateMenu(request, pk):
    if not user.has_perm('pizzaplace.change_menu'):
        raise PermissionDenied
#...
```

Access control can also be done at the API level, as shown in *Figure 6.7*. While it's still a good practice to validate access at the front-end level, most API endpoints should perform their own access control checks.

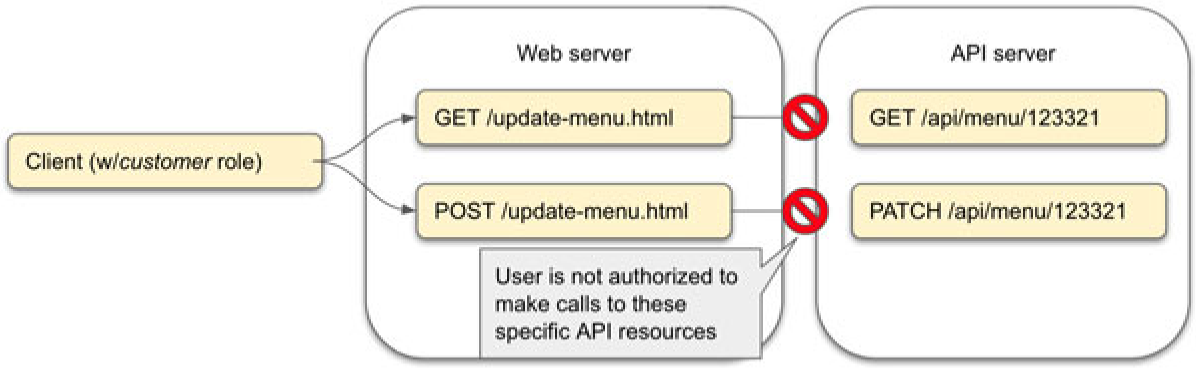


***Figure 6.7:** Authorization at the URL level*

We will discuss authorization to API resources more in depth later in this chapter.

In general, *each application module or server should treat all incoming requests as possibly unauthorized*, even if they are supposed to be coming from a trusted resource like the front-end web server.

Using scopes to check authorization

In addition to the permissions given to a user by their role, we can give that user permissions over specific resources. For instance, a **`manager`** user might want some **`admin`** users to be able to access the financial report for a meeting. In that case, we wouldn't want to upgrade **`admin`** users to **`manager`**, as the role might include other privileges we don't want to give them.

In this case, we can assign scopes or permissions directly to a user:

```
{
  "country": "United States",
  "timezone": "America/Chicago",
  "roles": [
    "admin"
  ],
  //…
  "permissions": [
    "read:finantial_report",
    // …
  ]
}
```

You can configure your authorization as you need to, but commonly these one-off permissions override whichever permission the user has based on their roles.

Test access control

Once you have configured authorization and authentication, all that's left is to use your *access control table* to define tests to ensure each role only has access to the resources they are allowed for, and that the application handles correctly the denial of access to unauthorized and unauthenticated users.

These tests can be unit tests trying to execute specific functions, or integration tests calling API endpoints; both manual and automated, as described in *Chapter 5, Automated Application Testing*.

Federated authorization

The authorization and authentication flow we just discussed is good enough for small, non-critical systems. It has some weaknesses, though:

- Good user, role, and privilege management is difficult to get right. Malicious users are constantly finding ways to bypass security measures and encryption algorithms. If you're implementing your own authorization and authentication administration services, you need to understand the latest trends in app sec and update your application accordingly.
- In the service described in the example, user management is tightly coupled with the application. There are multiple problems with this:
    - If you want your application to integrate with a third-party app, you have to provide them access to your user's database so they can access the session tokens and user information. This assumes you trust those applications enough to read all your data.
    - You can try to build an API for external apps to consume your user's information, but that would be just a juicy goal for malicious users. A lot of effort would have to be put into building and securing your security-sharing services.
    - If you don't want to share your data, you then need to send all user-management related data to the other applications who want to integrate with you. This approach is complex, as it requires you to sync any changes to the user management data across all applications using it.

Now, think of the case where you are building API services but multiple external companies want to integrate with your API as discussed in *Chapter 3, Designing APIs*. Who will own the user information? Will you trust them to access your database directly? Who will be the source-of-truth when conflicts happen?

A user management approach like the one we've been discussing in previous sections is not scalable. It really only works well if your application does not integrate with external APIs/applications or if your application does not need to scale horizontally.

We talked about **Single Sign-On** (**SSO**) which is a solution to the problem of giving access to multiple applications to the same user. SSO is a subtype of **federated authentication**.

In federated authentication, you use an independent user management service or **identity provider** (**IP**) to authenticate your users for you. The IP takes care of storing users and their data. Once the IP validates a user is who they say they are, it will give them a token which the users will use to let your application know that they are indeed authenticated.

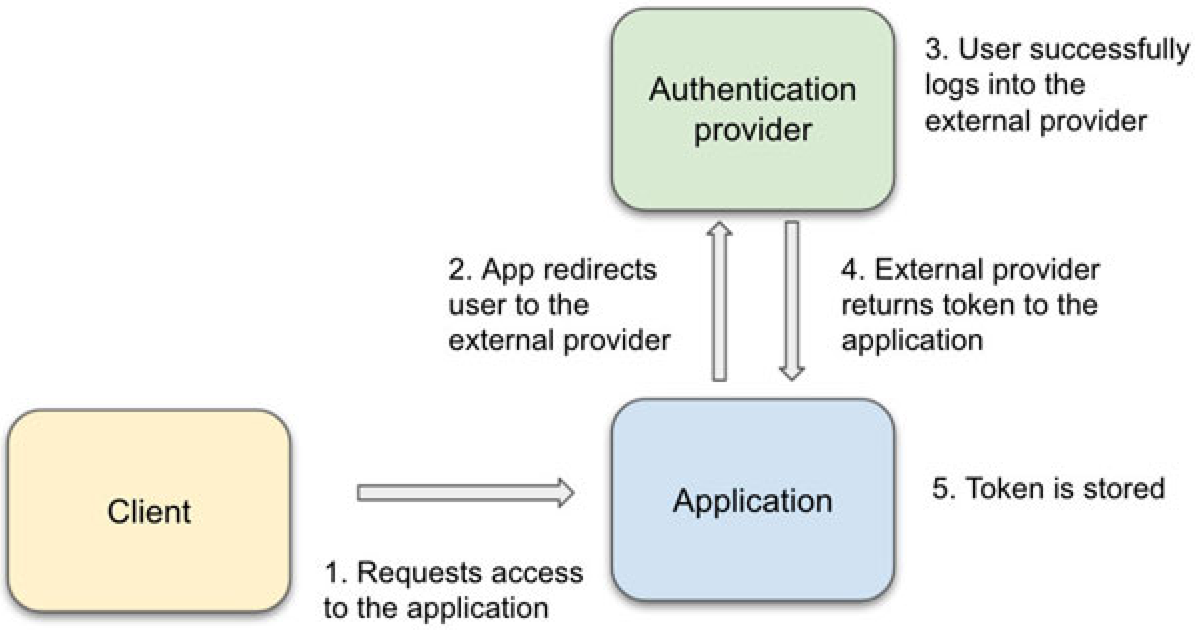

Authentication provider
3. User successfully logs into the external provider
2. App redirects user to the external provider
4. External provider returns token to the application
Client
Application
5. Token is stored
1. Requests access to the application

Figure 6.8: *Authentication flow when integrating with an external provider*

Figure 6.8 displays a high-level view of a user authentication for an application using authentication federation.

This architecture assumes there is an IP collectively trusted by you and all the applications you will integrate with.

You have multiple options when choosing an IP:

- You can build your own IP. Not really advised unless you're a large company or planning on providing IP services yourself. It requires too much work.
- Install an off-the-shelf solution like WSO2 Identity Server or Keycloak. You can buy a license for those products and install them in your private network.
- Use public cloud-based services. Federated authentication is one of the main services of specialized companies like Okta or Auth0; but large companies like Facebook, LinkedIn, or Google offer their own IP services to which you can integrate to.

Pros and cons of federated authentication

The multiple advantages of using federated authentication are:

- It can integrate seamlessly with internal and external applications. As long as all involved applications trust the **identity provider** (**IP**).
- Better separation of concerns. The developers working on the IP can concentrate in building all the latest features in app sec, as well as complying with regulations like HIPAA and GDPR.
- The IP service can evolve and grow independently of all the applications consuming it. It will only consume the resources it actually needs, instead of sharing them with an application.

On the other side, some of the downsides are:

- It increases complexity in the application. This is true for any implementation that requires the application to be distributed among different servers.
- Extra measures are required to confirm all authentications and authorization data is correct and trustful. These measures usually mean extra communication steps between the application and the IP as well as the use of cryptologic keys to make sure all tokens are correct and un-tampered.

Since federated authentication involves the integration and communication of multiple applications and services, using well-defined protocols and standards is important. The most commonly used are *SAML* and *OpenID*.

Security Assertion Markup Language (SAML)

Created in 2001 by the OASIS **Security Services Technical Committee** (**SSTC**), this open standard describes a process to exchange authorization and authentication information between multiple services, like between an IP and applications consuming it.

In *Figure 6.9* we find the typical SAML flow, and the interactions between the involved actors: the *service provider,* the *user agent*, and the *identity provider.* Please take a look at the following figure:

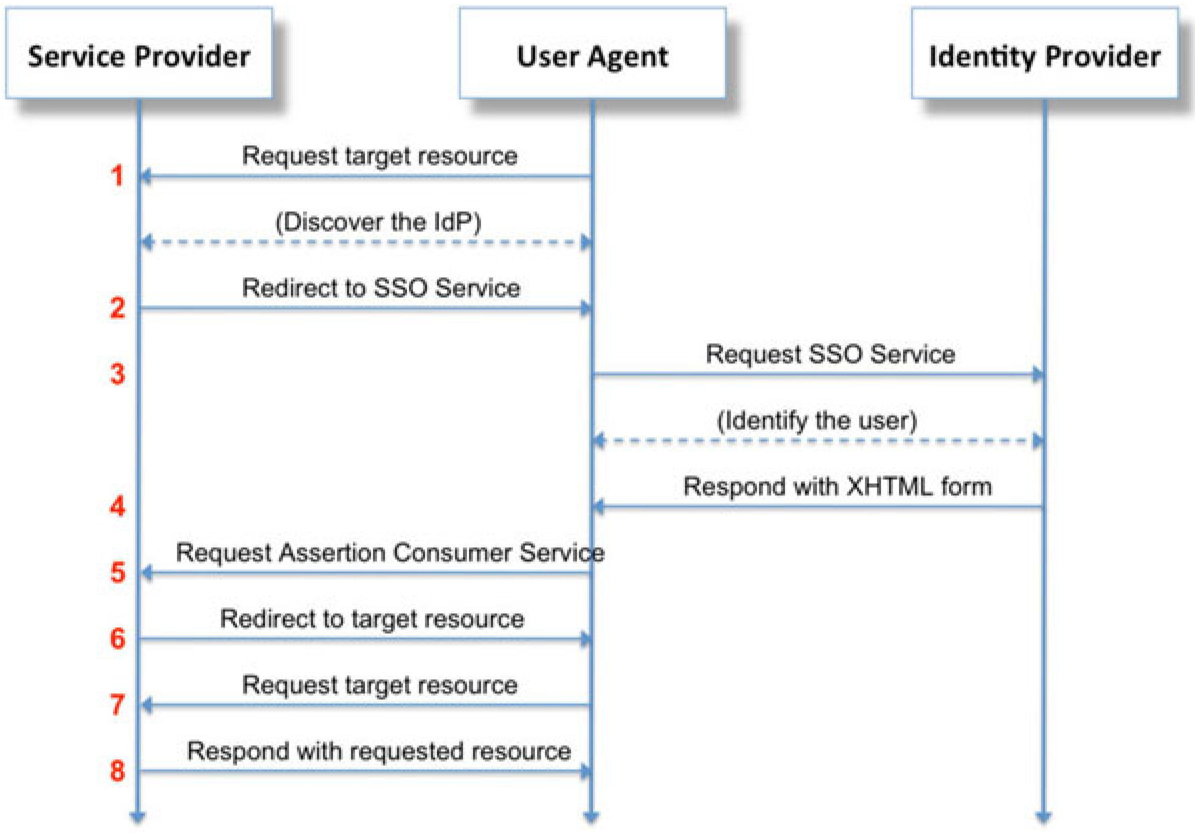


Figure 6.9: *SAML authentication flow*
Image provided by Wikipedia (license is linked in the references section)

SAML relies on XML documents that are digitally signed by both the application and the IP. This sign is then checked by both parties to verify its authenticity. The XML document includes *assertions*, which are just attributes related to the user like name, email, phone number, roles, and privileges.

SAML was the first attempt to standardize federated authentication, and it is widely used to authenticate users among multiple enterprise applications. Thanks to its maturity, it supports features like communicating through proxies. These features are not as well supported by other standards like OpenID.

OpenID

It's difficult to talk about OpenID without talking about OAuth2 first. These two specifications are so coupled to each other that they are constantly confused and used interchangeably.

OAuth2

OAuth2 is an *authorization* protocol. It's based on the idea of *authorization delegation*: An external service which validates the user's privileges to access a specific resource. In the context of OAuth2, these protected resources are commonly actions exposed by an API. The OAuth2 flow generates tokens in behalf of users that the applications send along the requests to the API to gain access to its protected resources.

For instance, imagine you're building an app that monitors a user's progress in a game and then tweets updates to their followers. This app will be used by "*speedrunners*": people who complete video games as fast as they can, sometimes breaking records. They need to concentrate on the game they are playing, but they also need to engage with their followers. To enable engagement of the gamers with their followers, we will use Twitter's API to perform actions in the social network in behalf of our users, all from our app.

In one use case, the speedrunner user opens your application and clicks on a button with the label "*Connect with Twitter*". Since it's the first time the user opens the application, we initiate OAuth2's flow to get an access token.

Once we have the token (or if we already had one), we can attach it to the request to Twitter's API. Twitter's server then checks it to verify that the application's request done on behalf of the user is authorized. If the access token is correct and the user has the right privileges, the API will execute the action (post to the user's timeline) and return the expected result.

Terms

Let’s make sure we use the same terms when discussing OAuth2’s flow.

- **Resource owner**: This is the user who owns the data.
- **Client**: This is another term to identify the application which wants to access data on behalf of the resource owner. While this term is used by OAuth2, this chapter will still refer to the client as an “*application*” to avoid confusing it with the resource owner.
- **Authorization server**: This is the external service which provides the authorization code and access tokens. It also has enough information about the user and the application to let the application know if the user has the right permissions or not.
- **Resource server**: This is the API which the client or application will try to access on behalf of the resource owner.
- **Scope**: This is the level of permissions to be done with the data.

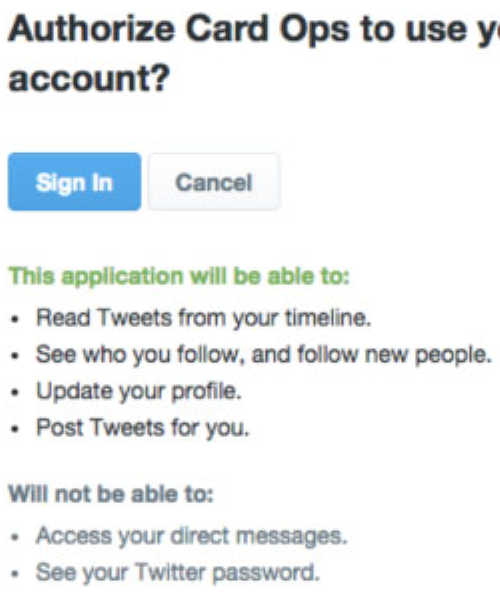
Request an authorization code

The ultimate goal of the OAuth2 flow is to get an *access token*. But, before we can get one, we need to perform a few steps:

First, we request an *authorization code* by redirecting the user to the *authorization server*, which for this example is Twitter’s authorization server.

The authorization server -Twitter- will let the user know what information is being requested (for example, name, email, phone number, profile information, list of tweets, contacts, among others), and the list of actions that the application wants to perform on the user’s behalf (post tweets, read the user’s timeline, and so on).

These are *scopes*, or as we have named them earlier, privileges. The user has to consent and approve the list of information the application is trying to access. *Figure 6.10* shows the screen in which Twitter’s Authorization server requests the user’s approval for authorizing the listed scopes for the application:


Authorize Card Ops to use your account?

This application will be able to:

- Read Tweets from your timeline.
- See who you follow, and follow new people.
- Update your profile.
- Post Tweets for you.

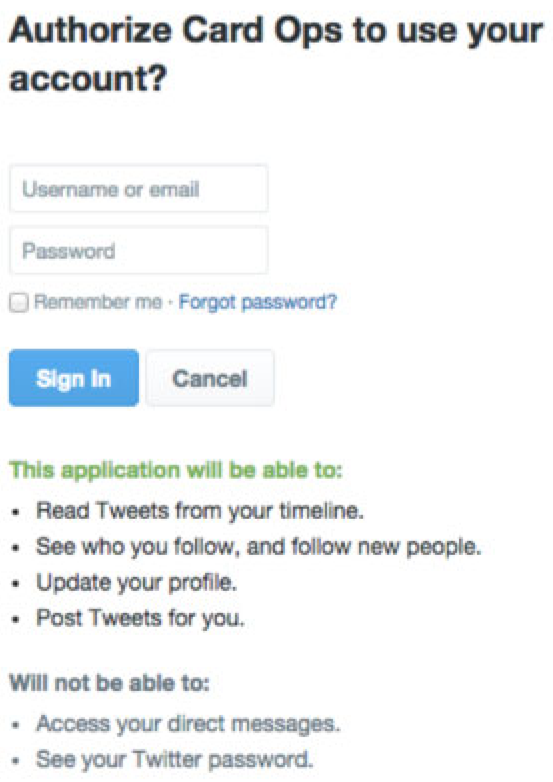
Will not be able to:

- Access your direct messages.
- See your Twitter password.

Figure 6.10: *The authorization server will let the user know exactly what is (and what isn't)*

`approved for the application to do and request in their behalf`

Before being able to provide the application with an authorization code, the authorization server will ask the user to log in to verify they are the owners of the account for which the application is requesting access. *Figure 6.11* shows Twitter's authentication page which is displayed right after accepting the scopes:


Authorize Card Ops to use your account?

Username or email

Password

Remember me · Forgot password?

This application will be able to:

- Read Tweets from your timeline.
- See who you follow, and follow new people.
- Update your profile.
- Post Tweets for you.

Will not be able to:

- Access your direct messages.
- See your Twitter password.

Figure 6.11: *User needs to authenticate with*

`Twitter before allowing the application to request the user's information`

Once the user approves the scopes and gets authenticated with the authorization server, the user's browser will be redirected back to the application, along with a brand new, one-use, authorization code.

Request an access token

Since all exchange until now has been public through redirects in the browser, we risk our authorization code being intercepted or exposed to malicious users.

To mitigate this risk, we need to do one last step: Exchange the authorization code for an *access token* through a *private channel*. In the server's back end, the application sends the authorization token along with a *client ID* and a *client secret* to the authorization server to get the access token. The authorization server will verify that the code is not tampered using a cryptographic signature. If the code and all the other client information are correct, the authorization server will respond with the access token.

Now the OAuth2 flow is complete and the application can send the access token with each request to Twitter's API to prove it has access to the API resource; at least until the token expires and we need to request for a new one. Fortunately, we can skip the log-in and user consent steps when refreshing the token, making the process transparent to the user.

The OAuth2 flow we just described is called a **code flow**. To put it all together, we can see each step in *Figure 6.12*:

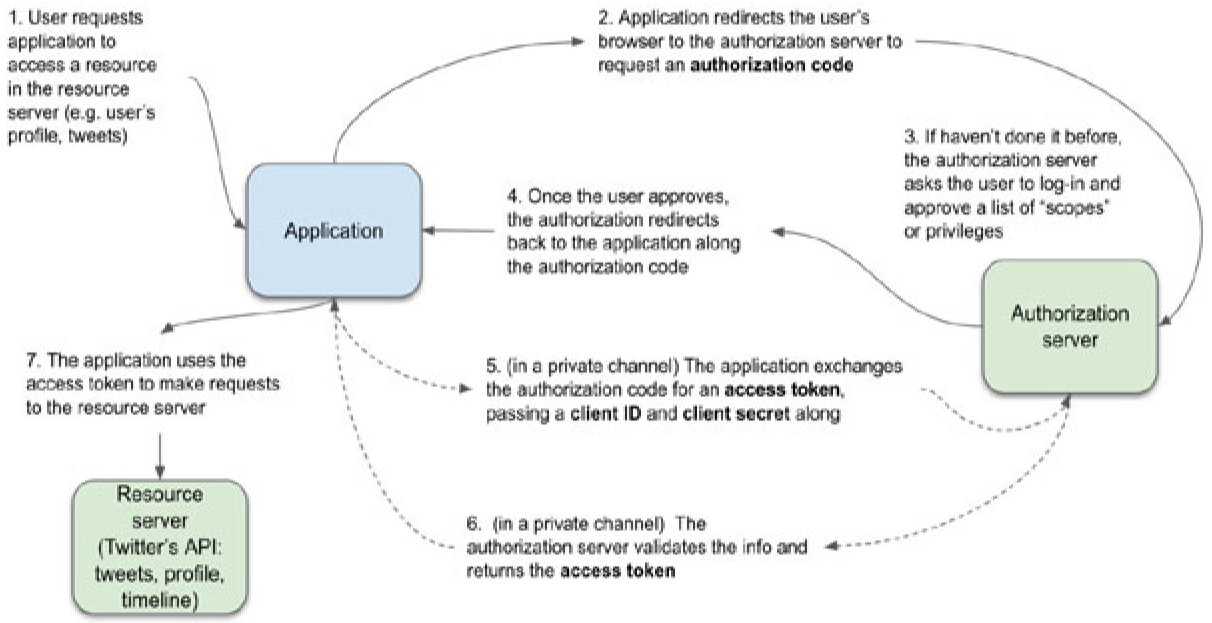

1. User requests application to access a resource in the resource server (e.g. user's profile, tweets)
2. Application redirects the user's browser to the authorization server to request an **authorization code**
3. If haven't done it before, the authorization server asks the user to log-in and approve a list of "scopes" or privileges
4. Once the user approves, the authorization redirects back to the application along the authorization code
5. (in a private channel) The application exchanges the authorization code for an **access token**, passing a **client ID** and **client secret** along
6. (in a private channel) The authorization server validates the info and returns the **access token**
7. The application uses the access token to make requests to the resource server
Application
Authorization server
Resource server (Twitter's API: tweets, profile, timeline)

Figure 6.12: *OAuth2's "code" flow*

The implicit flow

OAuth2 has other flows, to accommodate different application architectures. Another common flow is the **implicit flow**, which is used by applications that might not have a private channel to exchange the authorization code for an access token.

The implicit flow is almost identical to the code flow, except that it requests the access token directly, instead of requesting the authorization code first to later exchange it for the access token. Right after the first redirect, the application is ready to make authorized requests to the API.

The most common example of applications that use the implicit flow is **static web applications**: Front-end applications which are rendered directly from HTML, JavaScript, and CSS files.

Static web applications have no server-side at all: All actions are performed directly in the browser using JavaScript. Modern front-end applications built with frameworks like React or Angular often follow this pattern.

This flow trades-off some amount of security for a simpler flow to support these applications which cannot perform a full code flow. Figure 6.13 shows the steps in the implicit flow so we can contrast it with the code flow:

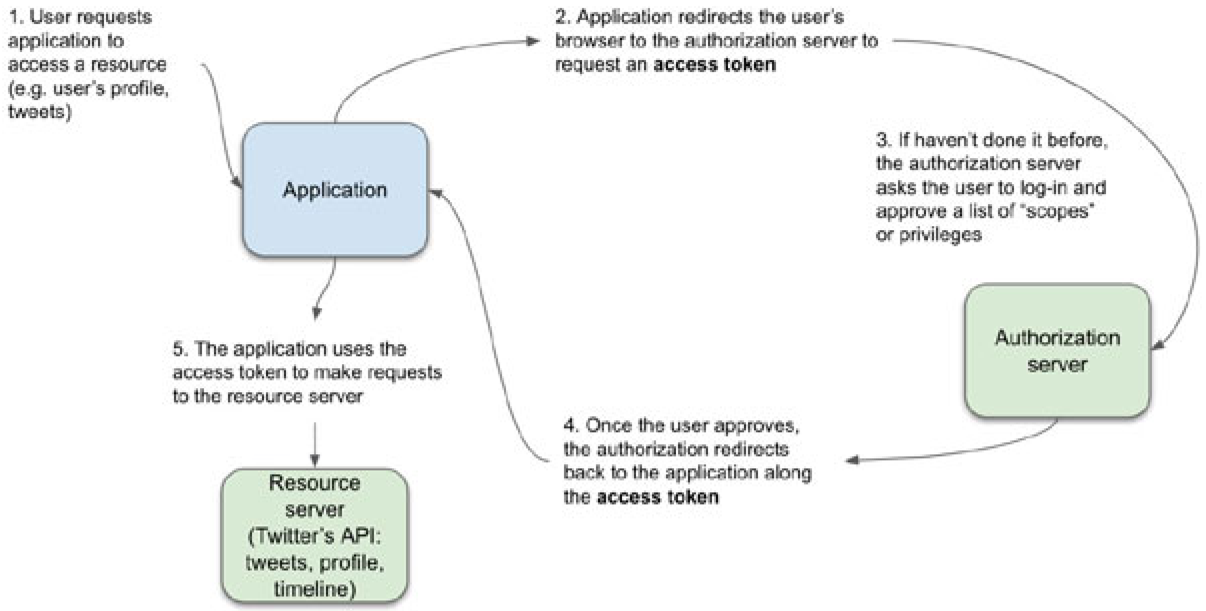

1. User requests application to access a resource (e.g. user's profile, tweets)
2. Application redirects the user's browser to the authorization server to request an **access token**
3. If haven't done it before, the authorization server asks the user to log-in and approve a list of "scopes" or privileges
Application
Authorization server
4. Once the user approves, the authorization redirects back to the application along the **access token**
5. The application uses the access token to make requests to the resource server
Resource server (Twitter's API: tweets, profile, timeline)

Figure 6.13: *OAuth2's "implicit" flow*

Building OpenID from OAuth2

OAuth2 is not an authentication protocol. For a long time -before the introduction of OpenID, OAuth2 was "*abused*" by many companies to perform federated authentication.

Developers started to assume that if a request is authorized to access a resource or an action, then it was safe to say that the user making the request had to be authenticated. Authentication, which was just a step in the authorization process, suddenly became the goal itself.

The problem was that each company started to patch they own authentication flows on top of OAuth2 as they saw fit. For instance, Facebook created *Facebook Connect*, which allowed developers to include authentication through Facebook in their applications.

Since each **Identity Provider** (**IP**) started building their own authentication processes on top of OAuth2, multiple complications arose: Not all providers implemented the same features. If you chose one specific provider, you would end up coupling to it, as migrating to a different provider required code and dependency changes to adapt to their own protocols. Privacy and centralization concerns grew. Developers had to comply with the processes defined unilaterally by the provider they integrated with.

This hole in the OAuth2 specification led to the creation of OpenID. OpenID filled the blanks, thus providing a specific way for providers to implement federated authentication while still supporting OAuth2 flows.

OpenID performs the same flows as OAuth2, but it requests an extra scope: "*openid"*. This scope allows the authentication server to consider this as an OpenID flow and include information about the user's profile in the authorization token.

This simple addition guarantees that the same client can integrate with multiple identity providers, as they all have to follow OpenID implementation rules.

Building security into the application's design

Application security is a horizontal concern in software development. This means that each element of the software development process has at least some level of involvement in the effort of keeping your application secure.

As any horizontal concern, application security needs to be defined before the application is written. Trying to "*add*" security to the application after it was designed and built will end up in a vulnerable system or in broken functionality.

Some business-level use cases will be affected directly by application security concerns. Critical actions like bank transactions typically require users to re-authenticate to improve trust behind each operation; some application owners might not like the added complexity, so these changes in business logic need to be considered from the beginning.

Before we even start building the application, we need to evaluate the security risks our application may face once it's in production. For that, we use Threat Modeling.

Creating a Threat Model

Threat Modeling is a process through which we identify potential security treats and vulnerabilities in a way we can estimate their criticality so that we can plan and prioritize fixes.

This process is usually performed by an application security expert, as it requires some level of experience in order to correctly assess the severity of some of the risks. However, in this section, we will describe the overall steps involved in the threat model so you can get involved in the design of secure applications.

The main steps to perform to create a threat model are as follows:

- *Decompose the Application*
- *Determine and Rank Threats*
- *Determine Countermeasures and Mitigation*

Each part of the process should be correctly documented so the whole team understands the threats which can be identified today; in that way, the Thread Model can be updated as new features are built.

Decompose the application

The first part is to fully understand how the application works and the modules it is composed of. As a backend developer, you might have only a limited view of the application, so it's important to involve other team members which have experience in other areas like front-end, database management, and so on.

In this step, we try to identify:

- Entry points which can be used by malicious users to access -and potentially attack the application.
- Assets which may be of interest for malicious users such as databases, logs containing sensitive information, and so on.

- List the possible vulnerabilities which are common for the type of application we're building (for example, web, mobile, or desktop applications).
- The access privileges for roles and actors.

Determine and rank threats

In this step, we review the information we put together in the previous step and try to find possible vulnerabilities which our application can be a victim of. We can use a model like **STRIDE**, which is a model defined by Microsoft to categorize vulnerabilities.

STRIDE categorizes vulnerabilities in:

- **Spoofing identity**: Illegally impersonate another user. Tampering with data. Any action which negatively impacts the integrity of data at any point in the application.
- **Repudiation**: Repudiation is the capability of malicious users to deny and hide their malicious actions. These vulnerabilities happen when there are no records which can prove a user performed a malicious action.
- **Information disclosure**: Violations to the confidentiality levels of data within the application.
- **Denial of service**: Violations to the availability of data within the application.
- **Elevation of privilege**: Cases where unprivileged users gain access to parts or data of the application which they shouldn't have access to, like admin features.

Not all threats are the same, though. Some can be exploited more easily than others, and some might have a deeper impact than others, depending on the specifics of the application implementation; for this reason, we need to define a way to rank threats. A common model to assess risks is **DREAD**:

- **Damage**: How bad would an attack be?
- **Reproducibility**: How easy is it to reproduce the attack?
- **Exploitability**: How much work is it to launch the attack?

- **Affected users**: How many people will be impacted?
- **Discoverability**: How easy is it to discover the threat?

Each category is given a rating from 1 to 10. Each category should be given a weight to better match with the priorities of the application. An overall score is then calculated by adding the individual scores or by doing a weighted average. This score will guide us in prioritizing which vulnerabilities need to be fixed first.

Remember that some vulnerabilities may not be exploitable in the current implementation but that situation can change as the application evolves. That is why it's important to include these vulnerabilities in the Threat Model but having a low exploitability score.

Determine countermeasures and mitigation

Design a plan to deal with the possible vulnerabilities, taking into consideration their prioritization scores. Most of the time, this plan consists in three actions:

- Fix those vulnerabilities that can be fixed.
- Mitigate vulnerabilities that cannot be fixed.
- Accept vulnerabilities that cannot be mitigated, or remove the features which contain them if they cannot be accepted.

We can see that the Threat Modeling process will end in very concise action items and documented decisions. This is great because this outcome will help us build a more robust application and to react correctly in case of attacks.

OWASP Top 10: The most common vulnerabilities

The best way to know how to protect your application is to understand how malicious users can attack it. This is the root rationale behind testing activities like penetration testing: Look at the application from the point of view of a malicious user.

You will find it difficult to keep track of all the possible attacks and vulnerabilities, especially if you're not an application security expert. This is why we focus in the most common vulnerabilities, which will render better results than trying to cover all or none of them.

The **Open Web Application Security Project** (**OWASP**) is a foundation that researches application security and provides tools and guidance on how to secure applications. They document common vulnerabilities and every year they post a list of the top 10 most common vulnerabilities.

As a backend developer, make a habit out of reading this list at least once a year. It will give you insights into how hackers attack applications and how some applications fail to defend themselves.

The OWASP Top 10 for 2021 is:

Broken Access Control: OWASP: "*Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification, or destruction of all data or performing a business function outside the user's limits.*"

Cryptographic failures: These vulnerabilities cover failures related to cryptography which often leads to sensitive data exposure or system compromise.

Injection: Injection vulnerabilities are those where, due lack of correct validation or sanitization of data incoming into the application, malicious users are able to pass parameters or execute code remotely to retrieve sensitive information or gain privileged access to parts of the application.

Insecure design: These vulnerabilities involve "*missing or incorrect control design*", which is the lack of consideration of application security features, constrains, and analysis like threat modeling while designing the application.

Security misconfiguration: Applications and servers which rely on configuration files to retrieve settings during runtime are vulnerable when those configurations contain factory settings like known or default passwords and keys, or excessively permissive configurations in things like firewalls, open ports, and other network configurations.

Vulnerable and outdated components: During the lifetime of most code libraries and dependencies, users and developers find vulnerabilities which then are fixed in future versions. When applications don't update these components in time, they open themselves to those well-known vulnerabilities.

Identification and authentication failures: Several vulnerabilities exist which cause the incorrect use and implementation of the authentication and authorization flows we discussed in this chapter. Other situations which compromise authentication are those where we expose sensitive data like access tokens or client secrets.

Software and data integrity failures: These are vulnerabilities which fail to guarantee the integrity of the application and the data that is used by it.

Security logging and monitoring failures: One critical aspect of application security is the system's capability of detecting and monitoring intrusions and attacks. Without an adequate logging and monitoring implementation, attacks can be led successfully without us having the chance to try to stop or mitigate them, or at least recover from them.

Server-side request forgery: This specific vulnerability exists when *a web application is fetching a remote resource without validating the user-supplied URL*. This lack of validation can lead to the server making requests to malicious resources through an insecure channel.

The OWASP website includes more complete descriptions of these categories, as well as some examples of the most common vulnerabilities. Feel free to jump in and get a better idea of how hackers take advantage of poorly implemented applications. Then, use this knowledge to build better software.

Conclusion

Securing an application is a large effort. It requires us to guarantee that data will always retain its confidentiality, integrity, and availability qualities. Data should be reliable and our application should create enough validations and constraints to guarantee that it remains like that. We understand that having losses in confidentiality, integrity, and availability has negative consequences both economically and in reputation for ourselves and our users.

In this chapter, we discussed how we protect applications through authentication and authorization. We verify our user's identities and the privileges they have over actions and data so that we can keep protecting the concerns described by the CIA triad.

We went through the most critical concepts in securing an application, as well as why we want to incorporate authentication and authorization checks.

We defined the most up-to-date best practices for password policies, which provide a revised approach to what rules we should enforce among our users.

We discussed how authentication works, why it's important to never store or transmit passwords in plain text, and how to implement a basic authentication flow. We then discussed how to define a model of privileges and roles to authorize (or not) users to perform specific actions or access specific data.

We then analyzed how to better scale authorization and authentication through *federation*: Authentication federation using standards like SAML or OpenID, and authorization delegation using OAuth2. We discussed how OAuth2 flows work, and the reason behind every one of its steps.

And at the end, we took a step back into how to consider possible vulnerabilities in our application to design Threat Models, remediation, and mitigation plans. We also saw some of the most common vulnerabilities for software applications, which gave us an insight into the minds of malicious users; all so we can better protect the data our users put into our applications.

Since we are in the mood of discussing remediation of negative situations, in the next chapter we will talk about application errors in general and ways to track, fix, and mitigate them.

Questions

Pick two different apps you use on a semi-daily basis. Then, for each of those applications, answer the following questions: What part or parts of the application do you think malicious users target the most? How do you think they try to "*hack*" these applications? Is there a part of the application which might be especially vulnerable?

Just remember: Performing malicious activities in an application without previous approval of the application owner is prohibited and it can get you into legal problems, regardless of your intentions. If you find vulnerability in a well-known application, report it. Some companies offer rewards to users who follow the right reporting process.

If you feel like you want to explore the field of penetration testing for defensive purposes only in a legal and safe way, there are resources out there like hackthebox.com, which offer "*hacking challenges*" for you to practice.

Resources

- Study about the impact of data breaches: **https://www.verizon.com/business/resources/reports/dbir/**
- NIST Digital Identity Guidelines: **https://pages.nist.gov/800-63-3/sp800-63b.html**
- National Cyber Security Centre's Password administration for system owners: **https://www.ncsc.gov.uk/collection/passwords/updating-your-approach**
- URL access control in Java web applications: **https://docs.oracle.com/cd/E19226-01/820-7627/bncav/index.html**
- Python's Django: **https://www.djangoproject.com/**
- WSO2 Identity Server IP: **https://wso2.com/identity-server/**
- Okta IP: **https://www.okta.com/**
- Auth0 IP: **https://auth0.com/**
- SAML flow Wikipedia image license: **https://en.wikipedia.org/wiki/File:Saml2-browser-sso-redirect-post.png**
- Threat Modeling process: **https://owasp.org/www-community/Threat_Modeling_Process**
- SAML specification: **http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-tech-overview-2.0.html**
- OAuth2 documentation: **https://oauth.net/2/**

- STRIDE model: **https://docs.microsoft.com/en-us/previous-versions/commerce-server/ee823878(v=cs.20)**

CHAPTER 7
Handling Errors

Even the most advanced application systems are prone to errors. How are these errors introduced into software applications? How can we deal with them? This chapter describes common causes of errors and patterns to help developers find and fix them. We will discuss how logging allows us to inspect errors after they happen.

Structure

In this chapter, we will learn the following topics:

- Why do we need to handle errors?
- Types of error handling
- Implementing good exception handling
- Finding production errors with logging
- Handling errors in distributed systems
- Use Case: Logging errors with the ELK stack
- A/B testing and gradual deployment

Objectives

In this chapter, we will discuss good practices for preventing and dealing with errors within our application. For this, by the end of the chapter, the reader will understand how to programmatically handle errors by implementing effective exception handling.

We will learn how to find and track errors in production environments using logging and to centralize log entries for entire distributed systems. Logging errors allow developers to find, track and fix bugs.

Finally, we will understand how to mitigate the impact of errors introduced by new features to production environments.

Why do we need to handle errors?

In the context of software applications, errors are any undesired or unexpected conditions in the application or its data. Any state or situation that goes against our application's expectations can be considered an error. This definition of an error is essential because many errors are *context-specific*, which in plain language means that an error for one application might not be considered an error for another.

For instance, having a user delete their profile may be an allowed action in many applications. However, other applications might require an administrator to remove the data in the user's behavior. If a user could delete their profile in this second situation, that would be considered an error.

Thankfully, most errors are less ambiguous or context-dependent; and more specific to certain actions or scenarios within the application.

Understanding common causes of errors

A large majority of actions can end up in an error state; and depending on how we can handle them, we can categorize errors into the following two types:

- **Recoverable errors**: When the application can do something to fix the error and return to a normal state.
- **Non-recoverable errors**: When the application cannot handle the error state possibly because the error is too critical or there is no good alternative path and an external actor (for example, a human administrator or a monitoring system) needs to act to recover the system.

An example of a recoverable error is when the application receives the wrong input from the user (for example, the user passes a negative integer for an operation that only accepts positive values). In that case, the application can return an error message to the user asking them to retry their action with the correct parameters.

On the other side, an example of a non-recoverable error is the case where the application's server runs out of memory and the application crashes. In that case, there is nothing the application can do to recover and requires someone else external to it to perform remediation actions like killing other processes to free memory, restarting the application, restarting the whole server, or spinning a new instance of the server altogether. Whether an error is recoverable or not depends on the error itself and the context in which it exists.

Other common causes of errors are as follows:

- A network connection fails.
- A file cannot be opened.
- A user tries to access a resource to which they do not have access.
- The user inputs incorrect data.
- The application's developer introduces a mistake in the code.

Any action that may have a probability of *not* completing successfully within the accepted parameters can be seen as a possible cause of errors.

It is our responsibility as software developers to inspect our code for possible sources of errors and then implement measures to preemptively deal with these errors as follows:

- If the error is recoverable, *log* and *recover*.
- If the error is not recoverable, *log*, *report*, and, if possible, apply mitigation actions such as creating a new instance of the application or dumping all data from the memory into the disk to reduce the negative impact the error will have on the user.

We will discuss *logging* later in this chapter. For now, let us focus on how software applications represent and handle these error states.

Types of error handling

As we will see, each coding language has its own way of dealing with errors. However, most of them can be categorized in specific patterns of error handling Let us talk about the most common strategies to deal with errors and see their pros and cons.

Exceptions

In coding languages like Java, C#, JavaScript, or Python, errors are categorized as *exceptions*. Exceptions are special conditions or states that can happen in an application at *runtime*. These events cause the normal flow of the application to diverge, and unless developers implement specific code logic to handle them, exceptions can potentially terminate the application's execution.

Using stack traces to debug problems

All exceptions have a **stack trace**: A text description of where the exception was thrown. The stack trace is a description of the application execution context at the moment the exception was created. The following is an example of what a stack trace looks like in Java:

```
Exception in thread «main” java.lang.NullPointerException
        at com.example.Item.getIngredients(Item.java:16)
        at com.example.Order.getItems(Order.java:25)
        at com.example.PizzaPlaceApp.main(PizzaPlaceApp.java:14)
```

As we know, when functions are executed, their context is put in a stack; if the ***`getItems`*** function calls the ***`getIngredients`*** function, the context of *getIngredients* will be inserted in the execution stack on top of the context of *getItems*. As the execution reaches more nested functions, the context for these functions will be added to the top of the stack.

To help us diagnose the situation when the application throws an exception, the whole execution stack is converted into its string representation and included as an attribute in the exception object.

Stack traces offer developers a step-by-step path the application took that led to that error. Using this information, we can *put breakpoints in the functions listed in the stack trace* during development to diagnose the cause of the error.

Defining exception types

In object-oriented languages like Java, all exceptions extend from base classes like ***`java.lang.Exception`*** or ***`java.lang.Error`***. More specific errors like ***`FileNotFoundException`***, an exception that the application raises when it tries to access a file that does not exist, extend these generic error classes.

Note: In Java, *java.lang.Throwable* is the base super-class from which both *java.lang.Exception* and *java.lang.Error* extend.

In that context, all classes that are derived from *java.lang.Exception* are supposed to represent recoverable errors, while classes that are derived *from java.lang.Error* classes represent serious, non-recoverable errors.

This subclassing of exceptions allows us to write code to handle different types of errors individually. However, this class hierarchy also leads to the not-so-great practice of developers always using ***`java.lang.Exception`*** to handle *all* types of errors. Always catching instances of the *Exception* class "*works*", but specific exception types provide a richer and easier to debug explanation of errors when these happen. It is easier to find out what happened when we handle a specific exception like ***`FileNotFoundException`*** than ***`java.lang.Exception`***.

Catching exceptions

Languages that support exceptions also have a way to indicate to the application how to deal with them through **`try-catch`** blocks:

```
try {
    // regular code execution
} catch(Exception exception) {
    // code to deal with exception
}
```

Every exception that is thrown or raised within the **`try { … }`** block will stop the regular application's execution and will continue in the **`catch() { … }`** block with the right type of exception, where the application is supposed to recover.

For instance, think of the case where the application tries to access a file that does not exist in the filesystem. If we want the application to handle this error case, a code like the following can be used:

```
try {
    BufferedReader rd = new BufferedReader(
        new FileReader(
            new File(fileName)));
    String line = rd.readLine();
    // keep reading the file

} catch(FileNotFoundException fileException) {
    // insert code here to deal with the exception
}
```

In this preceding code snippet, if any line within the try block raises a ***`FileNotFoundException`***, the execution will jump to that catch block where we can log and recover from that special case. If the code inside the *catch* block does not throw an exception itself, the application will continue as if no errors were ever found.

Also notice that, since we only have one catch block for ***`FileNotFoundException`***, any other exceptions that don't extend this class will not be caught.

Errors validated at compile time

Modern compilers are excellent at **static code** analysis: Parse the code and search for patterns that can lead to common errors: From syntax errors to possible memory leaks.

Some functions have very explicit failure cases. For instance, when opening a file, there is always a chance of the file not existing or being corrupted. We can take advantage of the compiler's static code analysis to *enforce developers to handle errors at compile time*. That is the case of checked exceptions.

Checked exceptions are those errors that the application must handle for their code to compile at all: Developers must catch and handle it in a try-catch block or explicitly re-throw it. For instance, in Java, we can tell the compiler that whoever calls the **`writeList`** function must handle an **`IOException`** by using the keyword ***throws*** in the function's signature:

```java
public void writeList() throws IOException {
    //…
}
```

However, the compiler cannot find all exceptions. Think of Java's ubiquitous **`NullPointerException`**. This exception will be thrown at runtime if we try to access an object's attribute or method referencing a **`null`** object reference. The following code snippet exemplifies the situation:

```java
List nullList = null;
nullList.toString();  // a NullPointerException is thrown
                      //becase nullList is null
```

`NullPointerException` is probably the most common and less informative exception in Java, as the exception itself does little to explain why the variable is null.

In Java, it is almost impossible for the compiler to know at compile time that a reference to an object will cause a **`NullPointerException`**; most of the time, null values are defined at run-time or passed as parameters, as follows:

```java
boolean createOrder(Items itemsParam) {
    Order order = new Order(itemsParam);
    return order.items.size() > 0;
}
createOrder(null); // causes a NullPointerException because
                   // order.items is null
```

This situation is why we always do "*null checks*" on function parameters: To avoid objects getting null references unexpectedly:

```java
boolean createOrder(Items itemsParam) {
    Items items = itemsParam;
    if (items == null) {
        items = new Items();
    }
    Order order = new Order(items);
    return order.items.size() > 0;
}
createOrder(null); // returns false
```

As there is no easy way to assert these exceptions at compile-time, we call them **unchecked exceptions**; and they require that developers use their experience to know how and where to handle them correctly.

Note: Some coding languages like Rust have no ‘null’ values. They use “Optional” objects: A wrapper that makes it explicit that a value may or may not have a value. In order to access their value, optional objects force developers to deal with the case where the value is empty.

Another example is Kotlin, whose variables by default do not allow null values. If we want to make an exception, we must explicitly declare the variable as “nullable”.

These patterns allow compilers and developers to assume that all values by default are always defined and show errors at compilation time for cases where we are not correctly handling possibly-null references.

Errors as return values

Not all coding languages follow the try-catch-exception pattern. Code languages like *Golang* consider errors as values that are part of the regular flow of the application; as such, the code to handle them also belongs to the main application’s flow.

For instance, the same use case where we want to open a file in Golang needs to handle errors as follows:

```
content, err := ioutil.ReadFile(filename)
if err != nil { // if the error is not null, there was an error
    return nil, err
}
```

```
5. return content, nil
```

Functions that can fail would return two parameters: One with the actual result of the operation and a second one that possibly contains a reference to an error object. If the error variable is `nil` -which means null- we can assume the operation was successful.

By making errors a part of the main execution flow, the language specification forces developers to handle them. Unlike exceptions where we can wrap all our code in a try block with the catch-all ***Exception*** class to deal with all errors, code languages that return errors require developers to capture and handle each error returned by a function. While this can be cumbersome, it also makes error handling more explicit and harder to ignore.

Another significant advantage of returning errors over exceptions is that it *increases the visibility of function calls in the application that could fail*. Errors are explicitly displayed in the code. On the other side, when using exceptions, each line in a function could potentially throw an exception.

In fact, the use of checked exceptions was how some coding languages tried to address the lack of visibility of possible errors in the code, with mixed success. However, checked exceptions still lack the clarity of returned errors.

So, at the end, which is better? Is it throwing exceptions or returning errors? We will find advocates for both approaches and, in the end, the code language we are using in our project will constrain our decision. Each approach has its advantages and you will only know which one you prefer until you use them both.

We can debate the pros and cons of each approach, but what is essential for any developer to know is how to deal with errors effectively, regardless of whether they are caught or returned.

Implementing good exception handling

Good exception handling is straightforward in principle but hard in practice. One significant cause of this is that developers do not like to think their code can produce errors, as we associate errors with poorly done work.

The truth is, while we must do everything we can to avoid introducing error conditions in our application, no application will be 100% bug-free. The only application that will always be free of defects is that which has no code at all.

Once we accept that we software developers are fallible humans, we have to create a multi-tiered plan to help us prevent errors from affecting our application and our users negatively:

- Preventing all the errors we can.
- Handling and mitigating errors that have reached a production environment.
- Loging all errors.

Preventing all the errors we can

Before we start handling errors, we need to prevent as many of them as possible. As mentioned earlier, we cannot prevent *all* errors. What we can do, though, is to put in place enough validations and processes to help prevent the most common errors from being introduced into the application in the first place:

- **Testing**: We have dedicated a whole chapter in this book to software testing and how it helps us catch errors before they reach production environments.
- **Linting/static code analyzers**: Just as modern compilers for languages like Rust or Golang will report possible errors at compile time, code analyzers allow developers to search for code patterns or "*smells*" which usually lead to bugs.

Remember the **Pareto principle** (or also called the *80/20* rule): 80 percent of outcomes (in this case, defects) are the result of only 20% of the causes. Focus on that 20% source of errors to maximize the results of your effort.

Note search term: Six Sigma

In the manufacturing world exists the understanding that we will never have zero defects. Some process improvement methodologies aim to reduce these defects as much as possible. For instance, Six

Sigma is a set of techniques oriented to understanding why defects happen, and how to standardize and improve processes to reduce variation and defects.

World-class companies that follow methodologies like Six Sigma focus on achievable and measurable goals. These processes are iterative, and teams constantly search for ways to improve their processes to produce fewer defects.

Among the errors that can be prevented are those related to quality issues: lack of validation, bad coding patterns and so on. Errors out of our control are things like networks, third-party dependencies, conflicting browser versions, among other erratic adverse conditions that can only be *handled*, not prevented.

Handling and mitigating errors in production

Once we have done all we can to find and fix all defects we could during development, what do we do with the rest of the latent errors hiding and lurking deep within our application? We apply the try-catch and null-checking-errors features we have been discussing in this chapter: Catch the error, log it and recover.

Defining good error messages

Our main goal when handling errors is to make them as invisible as possible to users. If we can recover from errors without users knowing they happened in the first place, we would have achieved this goal.

However, most errors cannot be fully handled without the user's intervention. Error messages communicate to users (and developers) what to do to recover from the error state in which they find themselves. Good error messages help users to recover by themselves with little to no intervention.

What makes a good error message? Error messages should use simple and clear language. They should also help users answer the following questions:

- Was the error caused by a user's action? If not, let them know it's not their fault.

- What do they need to do to recover? If nothing, let them know.

Remember that detailed errors are great for developers but not for users. *Error messages should be friendly to regular users, but informative enough for software developers who try to debug them*. The least amount of data you include in the message, the easier it will be for users to act on it.

Bubble up!

Errors are like water or foam: They bubble up across the execution stack, starting from the function that caused the error, all the way up to the main application process, as shown in *Figure 7.1*:

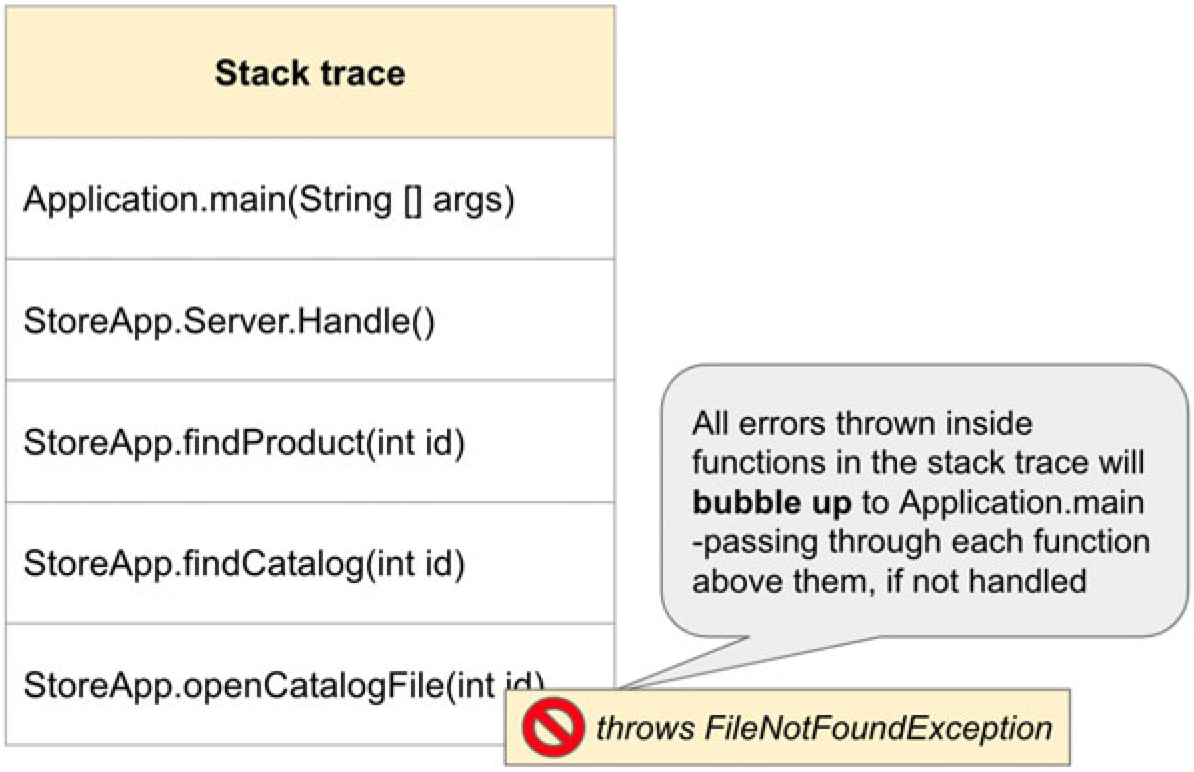

Stack trace
Application.main(String [] args)
StoreApp.Server.Handle()
StoreApp.findProduct(int id)
StoreApp.findCatalog(int id)
StoreApp.openCatalogFile(int id)
All errors thrown inside functions in the stack trace will **bubble up** to Application.main -passing through each function above them, if not handled
throws FileNotFoundException

Figure 7.1: *Visual representation of the stack trace. Errors that are not handled will bubble up*

Developers can catch and handle the error inside any of the functions in the execution stack. If the error is not handled, it will always bubble up to the main process.

To illustrate the idea in a better way, let us translate this into the following JavaScript code. We can simplify a use case to see how an exception thrown at the end of the stack bubbles up to the start:

```
class FileNotFoundError extends Error {
    constructor(message) {
      super(message);
      this.name = “FileNotFoundError”;
    }
  }

function main() {
    findProduct()
}

function findProduct() {
    // do more stuff
    findCatalog()
}

function findCatalog() {
    // do more stuff
    return openCatalogFile()
}

function openCatalogFile() {
    // simulate an error while opening a file
    throw new FileNotFoundError(“catalog.xml”)
}

main()
```

As described in *Figure 7.1*, we will see that the **`findProduct`** function calls the **`findCatalog`** function, which in turn tries to fetch the catalog’s data from a file using the **`openCatalogFile`** function. Notice how we have no try-catch blocks yet.

If **`openCatalogFile`** throws an exception, you will see an error message as follows:

```
FileNotFoundError: catalog.xml
    at openCatalogFile (…/chapter7/exception/handling-errors.js:23:11)
    at findCatalog (…/chapter7/exception/handling-errors.js:19:5)
    at findProduct (…/chapter7/exception/handling-errors.js:14:5)
    at main (…/chapter7/exception/handling-errors.js:10:5)
    at Object.<anonymous> (…/chapter7/exception/handling-errors.js:26:1)
```

While this error is informative, we need to also recover from it. We have aa couple options for recovery:

- Retry the action. For instance, errors that make network requests may fail due to a timeout or any other network issues which may be transient.
- Execute a fallback action: a "plan B" to execute when the original function fails, like a reduced version of the action we originally wanted to take.
- Skip the current action and return an error message. As long as the error is handled before it reaches the main process, the application may be able to continue operating.

Providing a fallback

In the existence of errors, users expect the application to work, even if its functionality is degraded or reduced. It's always preferable to execute a fallback action than just returning an error message.

We will provide a fallback for the previous example: if accessing the catalog file fails, we will choose to serve a simple catalog that is stored in memory. The catalog for this example has only one product:

```
var fallbackCatalog = {
    product1321: {
        name: “Deodorant”
    }
}
```

How well we can handle the errors depends on choosing where to handle the error. *As the error bubbles up, each function in the stack has less flexibility to handle it.*

We can handle the error within the **`findProduct`** function:

```
function findProduct() {
   try {
       findCatalog()
   } catch (error) {
           // the error stops bubbling. It will not reach “main”
   }
}
```

Or we can choose to keep propagating the error manually by skipping the catch block or by re-throwing it:

```
function findProduct() {
    try {
        findCatalog()
    } catch (error) {
           // do something and then bubble the error up to “main:”
        throw error;
    }
}
```

We must *handle the error as close as possible to the function that raised it*, as it gives us more options on alternative actions.

For instance, if we try to bubble the error up to the main function, the best we can do is to retry the whole **`findProduct`** function:

```
function main() {
    try {
        findProduct()
    }
    catch(err) {
        // handle here
    }

}
```

If the error is not *transient* (only happens for a short period) we would need to either call a different implementation of **`findProduct`** or pass extra parameters inside the fallback function to try an alternate action. Either way, this is a bad practice as it exposes implementation details that should remain internal to **`findProduct`**.

Nevertheless, if we capture the error right where it was thrown inside **`findCatalog`**, we have more control to provide a suitable fallback:

```
var simpleCatalog = {
     product1321: {
         name: “Deodorant”
     }
}

function findCatalog() {
     try {
         // do more stuff
         return openCatalogFile();
     } catch (error) {
         // log the error and return fallback catalog
```

```
        return simpleCatalog
    }
}
```

Now, all implementation details are correctly encapsulated within **`findCatalog`**: From the point of view of its parent functions (**`findProduct`** and **`main`**), no error was ever thrown, and the application keeps working correctly for the user, even if it provides a reduced catalog.

Letting the error propagate

We can see something concerning in the code example: While users are not as angry at us because they can still use the application, the error is still there. While handling the error, we *swallowed* it: We hide the error, pretending nothing ever happened. While that may be a semi-positive experience for the users, it obscures the real problem from the developers who can fix it.

Error swallowing is an awful (and widespread) practice. Many developers write try-catch blocks where the **`catch`** block is empty:

```
try {
    // some code
} catch (error) { }
```

These blocks will not fix the error; they barely hide it. Then, weird behavior starts happening in the application with no explanation: The product's catalog is empty but no one knows *why* thus resulting in a bad user experience and no way for developers to fix it.

Between swallowing the error and just letting it bubble up, we prefer to let it propagate as high as possible while keeping the application alive. In that way, we would still get a bad user experience, but we can get some hints on what is wrong with the application.

Is there a use case where we need to propagate exceptions? As our needs for error handling monitoring become complex, some developers would like to *handle errors in more than one place.*

For instance, if the **`openCatalogFile`** function was part of a library maintained by a separate team of developers, they might want to handle the error themselves and return a friendlier error message, but at the same time, they want to allow the application itself to do some more handling such as creating a new catalog file.

In this case, the developers of `openCatalogFile` may choose to catch the error, partially handle it, and then re-throw it so the developers of the main application can catch it themselves. This example might sound contrived, but situations like these are common in large applications where teams of multiple developers work in parallel.

Now, coming back to the fallback we introduced in the code example, we not only want to propagate the error, but we need to log it.

Finding production errors with logging

All errors that cannot be found during the development or testing phases will unavoidably surface for users in production. If we only had one or two users, finding and fixing errors in production would be very easy: We can talk to them to find out if they have experienced any unexpected behavior in the application. However, as we (hopefully) gain more users, finding errors becomes a more significant challenge.

The first step to fixing or mitigating an error is to detect it and diagnose it. Some errors are easy to find, especially critical ones that block all users from using the application. Other errors happen just in particular conditions, which can still apply to a large percentage of our users but are hard to reproduce for developers.

Logging errors allow us to capture a snapshot of an error in a real-world scenario.

Logging is the act of creating a record of an incident or action. *Logging helps us keep track of the application's state in application environments we cannot easily analyze directly,* like in test and production servers. While it is beneficial in the context of diagnosing errors, logging is not only for capturing errors but other information worth keeping track of:

- User log-in attempts. We can use this data to determine if malicious users are trying to access the application through brute force or password guessing.

- Requests to resources. By keeping track of requests done to things like API endpoints, we can extract information like usage metrics, client statistics, latency, and load times, among other useful statistics.
- Warnings. Events that are not necessarily errors but might be helpful while diagnosing other errors. These can be warnings about using deprecated dependencies or tools, insecure passwords, unusual traffic patterns, and other out-of-the-common events.

Logging is such a crucial action that most coding languages offer native libraries to allow applications to create log entries. For instance, we can use Python’s library **`logging`** as follows:

```
import logging

logger = logging.getLogger(‘main.py’)

logger.warn(‘This is a WARNING log entry’)
logger.error(‘This is an ERROR log entry’)
```

We can configure most aspects of the log, such as the format of each log entry and the location where they are stored.

Anatomy of a log entry

In its purest form, log entries are plain text strings. For instance, we can format a log entry as follows:

```
WARNING: 2021-11-05 17:00:12,737 - example_logger.py - This is a Warning
```

There are no hard limits on how log entries should be structured, as long as if follows a pattern previously agreed by the development team; this example follows a typical pattern. Notice the information this log line includes:

- **Log level**: These are usually **`INFO`**, **`DEBUG`**, **`WARNING`**, and **`ERROR`**. Having log levels help us filter log entries by their criticality: If we are trying to diagnose an issue, we can filter out all messages that are not errors. However, if we are running the application locally, we might want to enable all **`DEBUG`** messages to help us better diagnose how the application's internal state is behaving.

- The date and time when the log entry was created.
- The name of the file or class which triggered the log entry (e.g. **`example_logger.py`**).
- The entry's description: A text string describing why the log entry was made.

Log entries tend to be stored in the order they were created. Like other time-series data, once we have enough entries in our log, we can piece together larger events within our application.

For instance, take a look at the following list of hypothetical log entries:

1. `INFO: 2021-11-05 17:05:52,929 - login.py - Log-in successful: User 1312325`
2. `INFO: 2021-11-05 17:05:52,929 - login.py - Log-in successful: User 1331232`
3. `WARNING: 2021-11-05 17:05:52,929 - login.py - Log-in unsuccessful: User 00001`
4. `WARNING: 2021-11-05 17:05:52,930 - login.py - Log-in unsuccessful: User 00002`
5. `WARNING: 2021-11-05 17:05:52,931 - login.py - Log-in unsuccessful: User 00003`
6. `WARNING: 2021-11-05 17:05:52,933 - login.py - Log-in unsuccessful: User 00004`
7. …
8. `WARNING: 2021-11-05 17:05:54,929 - login.py - Log-in unsuccessful: User 00320`
9. `WARNING: 2021-11-05 17:05:54,930 - login.py - Log-in unsuccessful: User 00321`
10. `INFO: 2021-11-05 17:05:54,929 - login.py - Log-in successful: User 00322`
11. `INFO: 2021-11-05 17:05:55,929 - login.py - Log-in successful: User 1331433`

From the contents of this log, we can observe a couple of things:

- We are logging each log-in attempt done by our users.
- At 2021-11-05 17:05, our application had multiple log-in attempts by progressively increasing user IDs, all within a couple of seconds. Then, user 00322 was able to log in successfully.

From these observations, we can deduce that someone was doing an automated brute-force attack against our application, trying to find a valid user. Furthermore, we can see that, with a very high probability, they were successful in their attack at their 322nd attempt.

Note: Logging every log-in attempt in an application may not be the best practice if we have too much user traffic. Remember that log files or databases use disk space, and high traffic rates may lead to hard drives filling quickly.

A better practice may be to only log the log-in activity that may be suspicious, like users trying to log in after a given number of failed attempts.

Notice how none of the log entries were actual errors; unsuccessful log-in attempts are common among real users, so they have to be considered as `WARN` or `INFO`, but we still can deduce there is a problem with the application. This kind of error detection cannot be possible without an effective logging strategy.

What to log?

There is no real limitation in the kind of information we can or cannot log (beyond disk space constraints). However, it is important to be smart about it and recognize what information is actually useful to be logged.

If we log too little data, we risk failing to detect errors happening in production. And if we log too much data, we risk obscuring useful information with "*garbage*" log entries that will rarely be useful.

Logging is an evolving process. Start by defining an initial set of attributes that should be logged for specific actions or events, and add or remove data to be logged as you see fit.

For instance, you can start logging all the details each of web request: OS and browser type, time and date of the request, and so on. However, a couple of months later you may find out that you are never really using the OS information for each request. It would be perfectly fine at this point to stop logging this information. In the same way, you may recognize other attributes that you were not logging that may need to be added.

Designing good log and error messages

For any event that is worth being logged, remember we need to answer the following questions:

- What happened?
- What should have happened?
- Are there any differences between the expected and actual behavior?
- Who was involved in the event?
- When did it happen?
- What was the context in which the event happened?
- Did it involve specific versions of a component, like a browser?
- If this was an error, what was the exception? What is the stack trace?
- If this is log entry is specific to a part of the infrastructure like a server, what is the server address?

Note About sensitive information and logging Even if they are safely stored in a server's file structure, log files still are text-based content. Malicious users know that logs potentially store important information about the application, making these entries a particularly attractive goal.

A good practice is not to log any sensitive information (e.g., passwords, credit card numbers, and social security numbers, among others) that malicious users may take advantage of.

Of course, sometimes logging sensitive information is necessary to diagnose the error; and in those cases, it should be stored in an encrypted format and only for the shortest time possible. The idea is

that compromised log files should not reveal any data that can be used against the application.

For more details about application security concerns in logging, check OWASP's cheat sheets. A link is provided in the references section.

The best application logs clearly and concisely allow developers and system administrators to see the big picture of the application's state and possible causes for the logged error.

Persisting log entries to file

By default, files are directly printed to the application's standard output (for example, the console or terminal). If we want to persist the log entries, we need to redirect the standard output to a text file or configure the logger to use a text file.

Let us revisit the last code example we went through, but this time in Python:

```
import logging
import os
import sys

logfile_name = os.path.dirname(sys.argv[0]) + "/logfile.log"
logentry_format = '%(levelname)s: %(asctime)s - %(name)s - %(message)s'

logging.basicConfig(format=logentry_format, level=logging.DEBUG)

logger = logging.getLogger('login.py')
handler = logging.FileHandler(logfile_name)
logger.addHandler(handler)

```

```python
def openFileCatalog(file_name):
  return open(file_name, ‘r’)

def findCatalog(product_id):
  try:
    openFileCatalog(“catalog.xml”)
  except Exception as e:
      logger.error(‘Could not open the catalog for product {}’.format(product_id), exc_info=True)

findCatalog(123321)
```

It’s the same basic example, but this time we only use two functions **`findCatalog`** and **`openCatalogFile`**. In this example, we are trying to open a real file with “**`open(file_name, ‘r’);`**”. This action should fail as we have no **`catalog.xml`** to read.

Let us focus into the code used for the logging configuration:

```python
logfile_name = os.path.dirname(sys.argv[0]) + “/logfile.log”
logentry_format = ‘%(levelname)s: %(asctime)s - %(name)s - %(message)s’

logging.basicConfig(format=logentry_format, level=logging.DEBUG)

logger = logging.getLogger(‘login.py’)
handler = logging.FileHandler(logfile_name)
logger.addHandler(handler)
```

While this configuration is specific to Python, most coding languages follow the same pattern. Here are the highlights of this code snippet:

1. The **`logfile_name`** variable holds a reference to a **`logfile.log`** file. In this case, the reference points to a file stored in the same folder as the

Python script is currently executing, but typically developers create directories exclusively for log files.

2. The **logentry_format** variable holds a template that defines each log entry format, as discussed in the previous sections of this chapter.
3. The **Logging.basicConfig** function sets the settings which that be used for all logger instances. In addition to the log entry format, in this example are setting the lowest log level which should be stored (from low to high: **DEBUG**, **INFO**, **WARN**, and **ERROR**).
4. The **logging.getLogger('login.py')** function creates a logger instance for this specific file. Instead of '**login.py**', we can pass the name of the class executing the code.
5. The **logging.FileHandler(logfile_name)** function creates a new file handler in the location given by **logfile_name**. **logger.addHandler(handler)** adds the file handler to the current logger instance.

In the **catch** (or **except** as it's called in Python) block, we append the following line of code to create the log entry:

```
except Exception as e:
    logger.error('Could not open the catalog for product {}'.format(product_id), exc_info=True)
```

Since we have already defined the format and location for the log file in the preceding configuration, all we have to include in each log entry creation is the log description and whether we should include the stack trace or not (**exc_info=True** will include the stack trace).

The log entry created in the preceding example should look like this:

```
ERROR: 2021-11-09 08:42:11,461 - login.py - Could not open the catalog for product 123321
Traceback (most recent call last):
  File "…/chapter7/logging/python-logging.py", line 20, in findCatalog
    openFileCatalog("catalog.xml")
  File "…/chapter7/logging/python-logging.py", line 16, in openFileCatalog
    return open(file_name, 'r')
FileNotFoundError: [Errno 2] No such file or directory: 'catalog.xml'
```

If we execute the script, we should see that, in addition to logging into the standard output console, a **`logfile.log`** file was created. All log entries created by the logger will be appended to that file.

Notice that this single log entry follows all the patterns we have discussed: The right log level (**`ERROR`**), date and time, the name of the file that created the log entry, the description, and the stack trace describing the exact error.

Also notice how this log entry answers all the questions we might have about the error: The user-defined message "**`Could not open the catalog for product 123321`**" gives us more context about the action that caused the **`FileNotFoundError`** exception: The application was trying to open the catalog file for the product with ID **`123321`**. We have all the information we need to replicate and debug this issue in our development machine.

Handling errors in distributed systems

Error management in distributed applications is not significantly different from handling errors in single services, which we have discussed in this chapter.

In distributed systems, we have multiple services or applications running in parallel, possibly each deployed into its own server. Following the log configuration we just described will result in each service creating its log file, as shown in *Figure 7.2*:

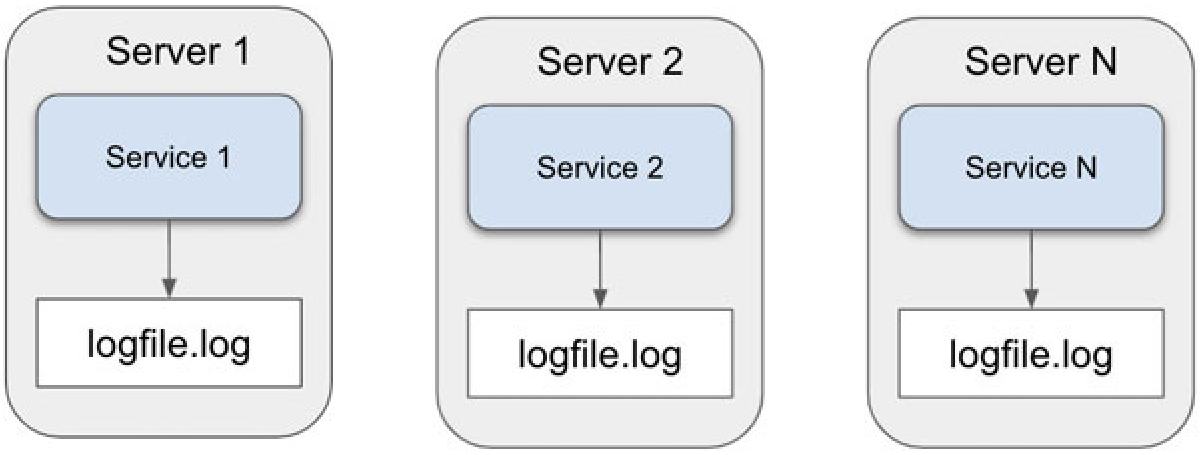


Figure 7.2: *Distributed systems: Each service has its own log file*

The challenge here is to *access* the log files. It is not too hard to access a single server remotely through SSH or any other communication protocol and extract the contents of the log file. Nevertheless, for distributed systems, we have N servers, each creating a high volume of entries in real-time. The distributed services architecture is a good case for log aggregation, which takes each service's log entries and sends them to a central location.

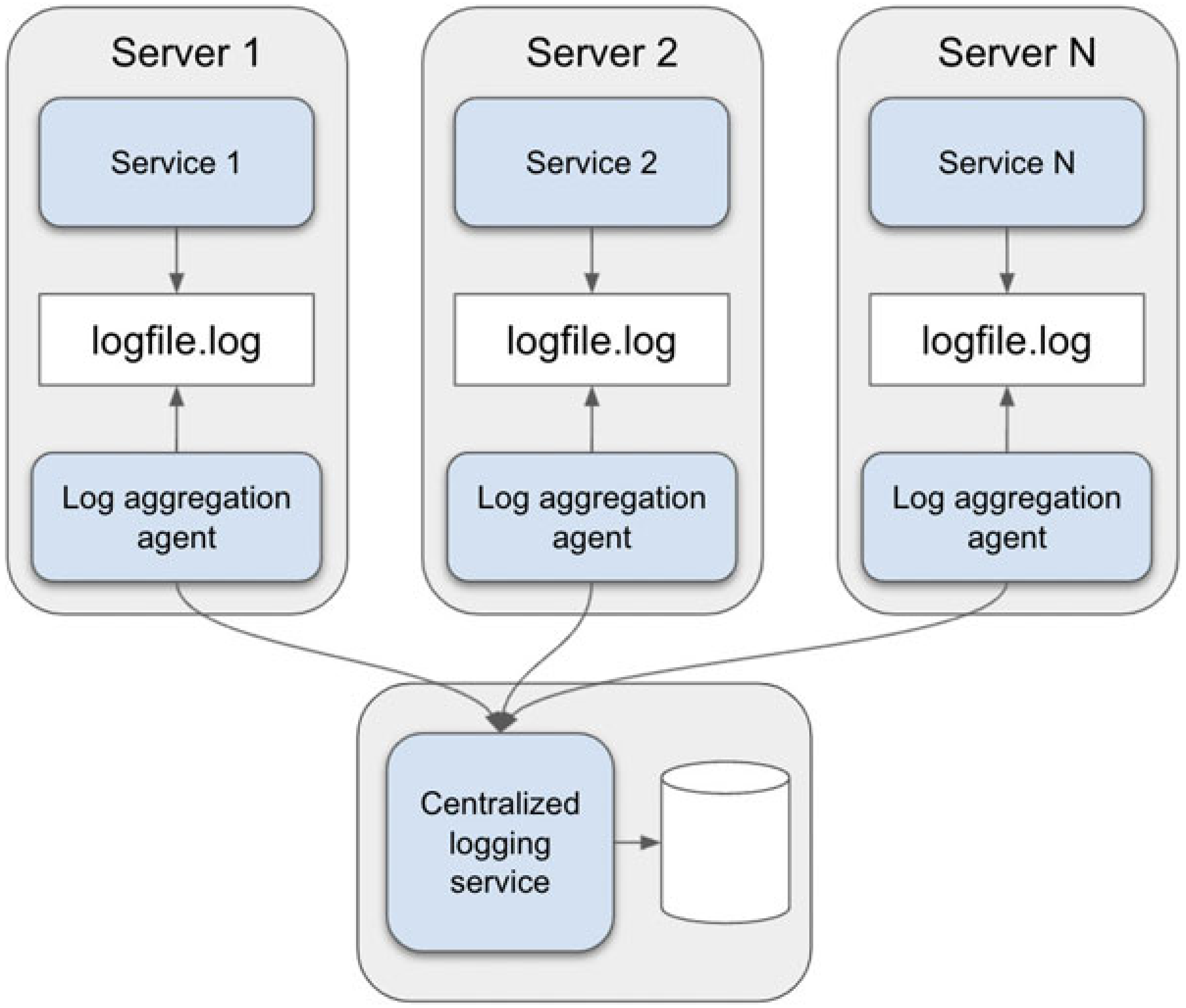


Figure 7.3: *Aggregating logs into a central service*

As seen in [Figure 7.3](), each service is deployed along with a log aggregation agent: A script that is running on the same server as the application. This agent detects changes to the local log files, formats them, and sends them to a centralized logging service. Then, when we need to diagnose an error, we only have to visit the central logging service. Now we can access the logs for the entire system from a single location.

Aggregation agents append an identifier to each aggregated log entry, in case we need to know what server created it.

What is remarkable about these aggregation agents is that they can parse distinct formats of log entries. We can aggregate log files generated by our application or log files created by other services like servers, databases, and any other service which generates log files on their own.

Using case: Logging errors with the ELK stack

The stack **Elasticsearch, Logstash, and Kibana** (**ELK**) is the most popular log management platform. Each tool has a job:

- **Logstash** aggregates, transforms, formats logs, and sends them to other services like Elasticsearch where they are aggregated.
- **Elasticsearch** is a full-text search engine based on Apache Lucene. It stores and indexes data, providing an API to filter and query its contents. Elasticsearch can be considered a NoSQL database.
- **Kibana** is a web application that allows users to interact with Elasticsearch and provides helpful visualization tools.

One significant advantage of the ELK stack is that it is *open-source*, thus making it an affordable solution for big and small companies alike. Whole clusters of Elasticsearch and Kibana can be deployed in private networks, allowing teams to control and monitor all the logs produced by their applications and servers.

The ELK stack fits perfectly in the centralized-logging service we discussed in the previous section. We can swap the names and get an excellent visual representation of how the ELK stack works. *Figure 7.4* shows this representation:

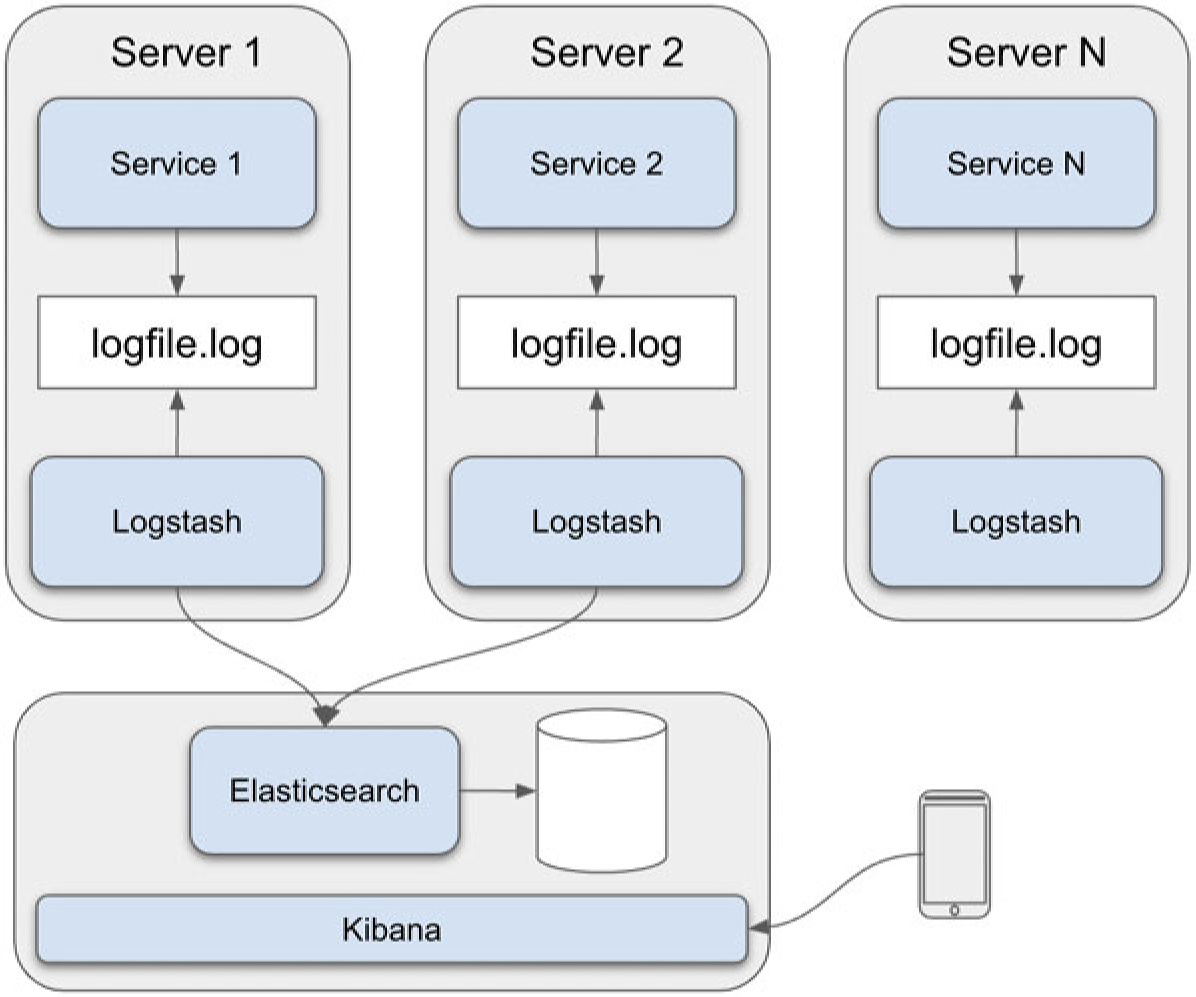


Figure 7.4: *The ELK stack*

With little effort, we can have a complete logging monitoring platform up and running.

In the following section, we will rely on the ELK stack to centralize and monitor the errors for our application.

Logging errors locally

The first step is to configure logging into our application. If we are using Python, we can rely on the **`logging`** library; or if we are using Java, you can use **`log4j`**.

For our example, we will create a script that will create many log entries to simulate an application with a semi-high amount of traffic:

```
import logging
import os
import sys
from time import sleep
import random

logfile_name = os.path.dirname(sys.argv[0]) + “/logfile.log”
logentry_format = ‘%(levelname)s: %(asctime)s - %(message)s’

logger = logging.getLogger(‘logstash-test.py’)
formatter = logging.Formatter(logentry_format)

handler = logging.FileHandler(logfile_name)
handler.setFormatter(formatter)
logger.addHandler(handler)
logger.setLevel(logging.DEBUG)

for i in range(60):
    for i in range(random.randint(0, 200)):
        logger.warning(“This is warn”)

    for i in range(random.randint(0, 200)):
        try:
            raise FileNotFoundError(“config.xml”)
        except Exception as e:
            logger.error(‘Could not open the config file’, exc_info=True)

    for i in range(random.randint(0, 200)):
        try:
            raise Exception("Product does not exist")
        except Exception as e:
                logger.error('Could not create product', exc_info=True)
    sleep(2)
```

We can see the same logger configuration we discussed earlier. The logger stores all entries in a local file named **`logfile.log`**. We also have the following three loops:

- One loop executes warning-level log entries N times, with N being between 0 and 200.
- Two loops execute error-level log entries N times, with N being between 0 and 200.
- One global loop in the top of these other loops, running 60 times and waiting for a second at the end of each iteration.

Essentially, we are simulating a scenario where an application with multiple users is generating a high number of log entries for one minute. This simulation should give us enough log entries to create a good example visualization.

Configuring Logstash

We need to install the Logstash agent in the same server where the logs are stored to monitor changes to the log files and send the new entries to Elasticsearch. In the "*Resources*" section at the end of this chapter, you can find a link to the Logstash installation guide for your chosen OS.

Once we have installed Logstash, we will create a Logstash configuration file with the name **`logstash-simple.conf`**:

```
input{
file{
        path  =>  "/absolute-path-to-code/Code/chapter7/elk-stack/logfile.log"
    start_position => "beginning"
    codec => multiline {
        pattern => "^(WARNING|INFO|DEBUG|ERROR)"
        negate => true
        what => "previous"
        auto_flush_interval => 1
    }
}
}
filter {
    grok {
        match => {
                    "message"  =>  "%{LOGLEVEL:log-level}:  %{TIMESTAMP_ISO8601:timestamp} - %{GREEDYDATA:message}"
        }
    }
    date {
        match => ["timestamp", "ISO8601"]
    }
}
output{
    stdout{codec => rubydebug}
    elasticsearch{
        hosts => ["localhost:9200"]
        index => "my_python_exceptions_index"
    }
}
```

There are three attributes at the root of the file: **`input`**, **`filter`**, and **`output`**. Each one of them provides a configuration for the main tasks Logstash performs: *Read* the log file, *parse* and transform each log entry, and *send* it to the aggregation service.

Logstash input

The **`input`** attribute indicates the location and format used by Logstash to read the log entries. In this case, it will read the **`logfile.log`** file we just created. The **`start_position => “beginning”`** sub-attribute tells Logstash to start reading the file from the first line.

Since, by default, it will try to read the file line by line, the **`coded => multiline`** attribute is configured to consider each line that does not match the regex attribute pattern as part of the previous line. This unifies log entries that have multiple lines (like Python exceptions) into one single entry.

The following log entry will be considered as a single record:

```
ERROR: 2021-11-09 12:49:10,611 - Could not open the config file
Traceback (most recent call last):
  File “/absolute-path-to-code/chapter7/elk-stack/logstash-test.py”, line 22, in <module>
    raise FileNotFoundError(“config.xml”)
FileNotFoundError: config.xml
```

As the log entries are read as defined in the *input* configuration, they will be processed as indicated by the *filter* section.

Logstash filter

The **`filter`** attribute defines how Logstash will parse the log entries. The main tool used for parsing is called *Grok*, which is a regex super-set. Logstash has many predetermined patterns, allowing us to match most of the common log formats. For instance, we use this Grok pattern to match our logs:

```
“%{LOGLEVEL:log-level}: %{TIMESTAMP_ISO8601:timestamp} - %{GREEDYDATA:message}”
```

In comparison, this is what a log entry would look like:

```
WARNING: 2021-11-09 12:37:27,323 - This is warn
```

It is easy to see how the pattern will match and extract the information from this log entry:

- **`WARNING`** will be assigned to the **`log-level`** attribute.
- **`2021-11-09 12:37:27,323`** will be assigned to the **`timestamp`** attribute.
- **`This is warn`** will be assigned to the **`message`** attribute.

We can add as many filtering patterns as we like. For instance, we added a **`date`** filter that matches any dates in the log entry in the example. Having multiple filtering patterns gives us flexibility by allowing us to only extract the information we need from the logs.

After being parsed by the patterns in ***`filter`***, each record will be sent to the locations configured in the output attribute.

Logstash output

The **`output`** attribute defines where we will send each parsed log entry. We can configure multiple locations in the same configuration file. In this example, we are sending it to two locations: **`stdout`** -the application’s console- and Elasticsearch database.

For the Elasticsearch configuration, we provide two attributes: the host where we will host Elasticsearch (**`localhost:9200`** for this example) and a name for the index (**`my_python_exceptions_index`**). An index is the Elasticsearch analog to a database table.

Having created the configuration file, we now can execute Logstash:

```
logstash -f logstash-simple.conf
```

This command will create a new process in the server (or your development machine) that will continuously scan the log file configured in the input, detect when new entries are added, and process them as configured.

Installing and configuring Elasticsearch and Kibana

There are many ways to install and configure Elasticsearch and Kibana, and we need to define which one is the best for us. Since arbitrarily picking one would not fit every use case for everyone reading this book, we have added a link to Elastic's up-to-date installation documentation in the references section at the end of this chapter.

We have included a *Docker Compose* file along with the code that is distributed with this book. If we install Docker (for which we have also included a link to the installation guide in the "*References*" section), all we would have to do is to run the following command in the same folder where the **`docker-compose-yml`** file is on:

```
docker-compose up -d
```

That will create instances of both Elasticsearch and Kibana in our Docker engine. They can be accessed through the *localhost* URL using their default ports **`9200`** and **`5601`**.

Note About describing installation steps in books

Books that describe step-by-step instructions about how to install something do not age well. Most of the time, the tool will evolve, new versions will be released, and the installation process will change.

Instead of focusing on the specific steps used to do something that is time and product-specific, it is better to provide insight into the important concepts that will remain part of the product as it evolves and grows. We will always find an up-to-date installation guide on the Internet, so it is better not to spend time and pages of our book on that.

Creating Kibana dashboards

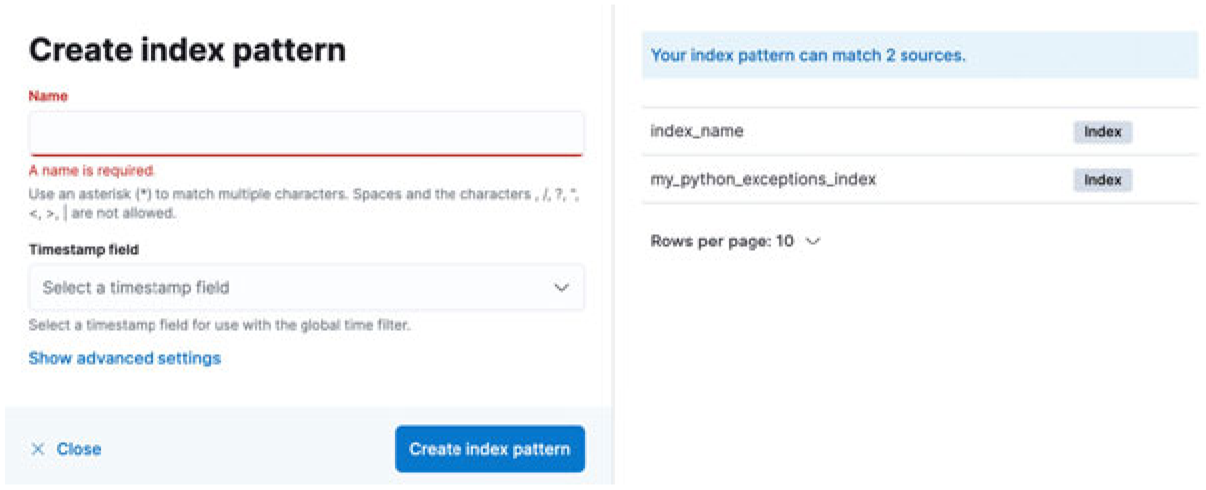
Kibana is based on dashboards: Collections of visual representations of the data stored in Elasticsearch and any other data sources integrated into Kibana. These can contain tables, charts, and graphs, among other types of visualization tools.

Kibana is particularly friendly the first time we use it. Once we execute Logstash and log entries start to be sent to Elasticsearch, Kibana will identify an existing index **`my_python_exceptions_index`**.

The first step is to create an index pattern, which is how Kibana reads and parses Elasticsearch data. This step is automated, as all we have to do is go to the following URL of our locally-deployed Kibana, and click on "**`Create index pattern`**":

```
1. http://localhost:5601/app/management/kibana/indexPatterns
```

In the name input field, type the string "**`my_python_exceptions_index`**" and select the "**`timestamp`**" field from the drop-down menu. Click on the "**`Create index pattern`**" button, and you will be done, as shown in *Figure 7.5*:



A name is required.

Use an asterisk (*) to match multiple characters. Spaces and the characters , /, ?, ", <, >, | are not allowed.

Select a timestamp field for use with the global time filter.

Your index pattern can match 2 sources.

index_name	Index
my_python_exceptions_index	Index

Figure 7.5: *Creating an index pattern in Kibana to parse the log entries we just stored in Elasticsearch*

Having created the index pattern, we can create a new dashboard. To do so, navigate to the following URL:

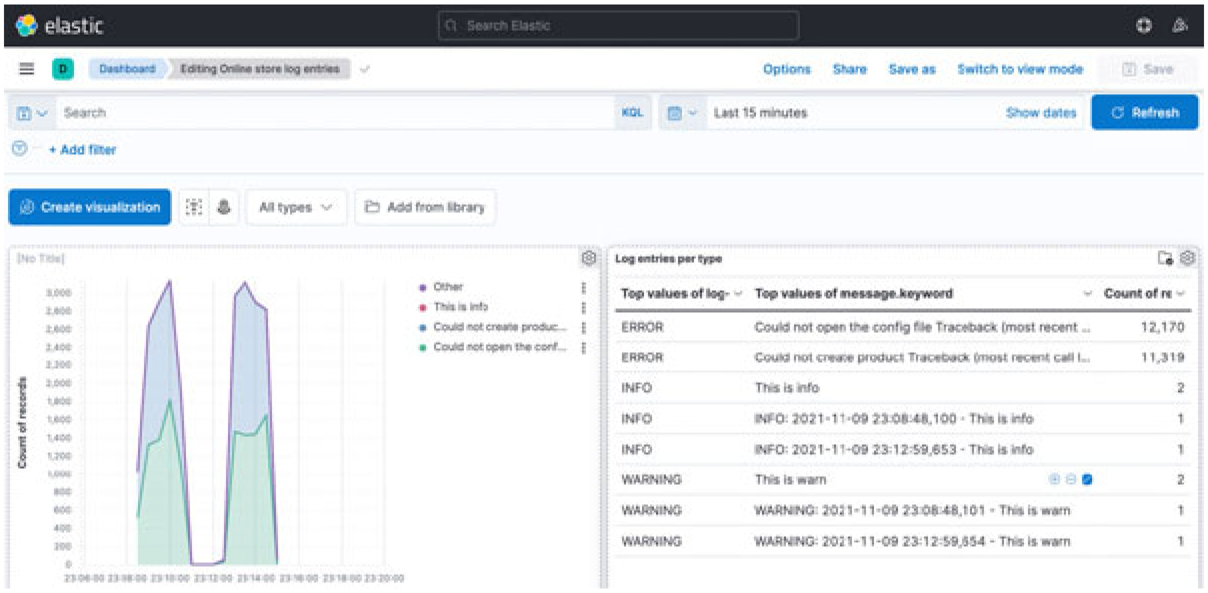
```
1. http://localhost:5601/app/dashboards#/create
```

There are many steps to create a dashboard, but they are pretty intuitive. We will not go much in detail, but, in general, we will create *visualizations*, which are different views of the data. The overall steps to create a visualization are as follows:

1. Click on "**`Create visualization`**".

2. Select the visualization type. For this example, pick any of them.
3. Select the fields we configured in Logstash's regex to parse from the log entries (for example, **`log-level`**, **`message`**, **`timestamp`**, and so on), and drag-and-drop them in the visualization canvas.
4. Save the visualization and return to the dashboard.
5. If it is not included already, add the new visualization to the dashboard.
6. Repeat all steps to add as many visualizations as we wish.

In *Figure 7.6*, we can see how a dashboard created using the log entries we just produced in this example will look like:



Top values of log-	Top values of message.keyword	Count of re
ERROR	Could not open the config file Traceback (most recent ...	12,170
ERROR	Could not create product Traceback (most recent call l...	11,319
INFO	This is info	2
INFO	INFO: 2021-11-09 23:08:48,100 - This is info	1
INFO	INFO: 2021-11-09 23:12:59,653 - This is info	1
WARNING	This is warn	2
WARNING	WARNING: 2021-11-09 23:08:48,101 - This is warn	1
WARNING	WARNING: 2021-11-09 23:12:59,654 - This is warn	1

Figure 7.6: *Kibana dashboard for the index "my_python_exceptions_index"*

Along with the provided source code for this chapter, we've included a file named "**`exported-dashboard.ndjson`**". This file contains all the definitions of the index patterns, dashboard, and visualizations you see in *Figure 7.6*. We can import that definition file in the following URL:

1. `http://localhost:5601/app/management/kibana/objects`

Please consider that by the time you read this book, Kibana's interface may have changed, and the process of creating and importing the dashboard may be different. That is the reason why the steps to create the dashboard are only high-level descriptions.

A/B testing and gradual deployment

The last strategy to mitigate the impact of errors is a direct consequence of *A/B testing*.

In A/B testing, we *deploy two different versions of a feature simultaneously to different segments of our users*. For instance, we can deploy a new variant of the "`Create order`" button, (which is visually larger) to 5% of our users, while the rest still see the existing, smaller version of the button.

By deploying different "*variants*" or versions of the same functionality, we can *compare their adoption and usage statistics* and find out which one is more successful. If the metrics find that a specific variant of the tested feature has more engagement than the others, we can choose that variant to be deployed to all users.

While A/B testing is excellent for comparing variants of existing features, the process of deploying a new feature to a small segment of users has the extra benefit that any *errors introduced by the new feature will only impact users who belong to the tested segment*.

We can embrace this consequence and plan the *gradual* deployment of new features and other significant changes: *Deploy new changes to a small percentage of the users and wait for a while; if no errors are found, then increase the percentage of users who can see the feature*. Please take a look at following figure:

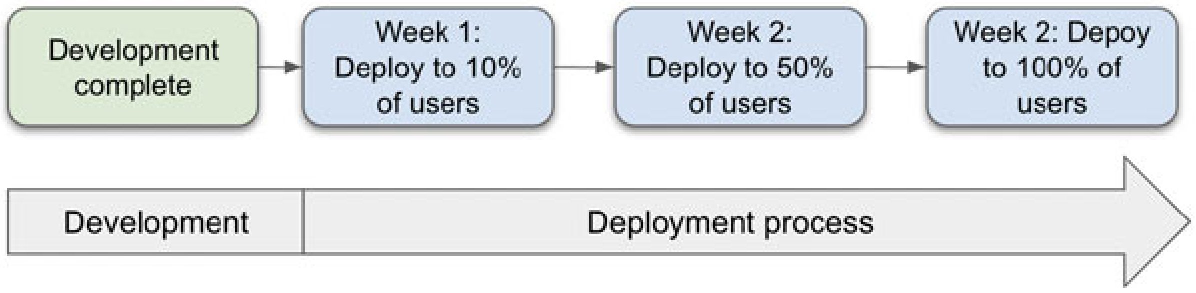

Development complete
Week 1: Deploy to 10% of users
Week 2: Deploy to 50% of users
Week 2: Depoy to 100% of users
Development
Deployment process

Figure 7.7: *Gradually increasing the percentage of users who have access to new features*

By gradually deploying new features or significant changes, new errors will only impact a few users. While the error impact is still negative, it is not the same that 1% of our users see the error as to have 100% of them being a victim of them.

Creating a deployment plan

How should we split the deployment? In *Figure 7.7*, we see an example of ramping to 10% of users for one week, checking for errors, deploying to half the users if no significant problems are found, waiting again, and finally deploying to the rest of the users. These numbers are used to illustrate the process, and you should find a schedule that better suits your users.

Applications with a large user base may want to deploy more slowly than that, considering 10% of users may still be many users. Smaller applications may want to deploy faster, as 10% of users may not be enough to find latent problems.

You can start by creating a simpler plan: Deploy to 30% of users and then all the way to 100% after a couple of days. If you notice that too many errors are getting to 100% of users, then slow down the process. If the process is taking too long, then speed up the deployment process.

Gradual deployment of new features is an important tool to contain the negative impact of potential new errors.

Conclusion

No application is free of errors. One thing that separates junior developers from seniors is knowing what to do to handle and mitigate errors found in production environments.

We have seen how some code languages choose to produce and handle errors by returning them from the functions which produce them. This design choice enforces developers to handle errors instead of just hiding them or ignoring them.

Most popular code languages, though, use exceptions for error handling. Exceptions create separate flows of logic that the application executes when "*special*" conditions like errors are met.

Whether it is by returning or throwing errors, it is our responsibility as software developers to write code that handles or mitigates any error that can be found. Some errors can be easy to find and handle, as they are relatively expected in some operations like file management or network requests. In contrast, others are the product of specific conditions of data in runtime.

When possible, errors should have fallbacks that provide some default behavior in case of errors. A good fallback is that which prevents users from knowing an error even happened. Unfortunately, not all errors can be handled as seamlessly.

All errors, regardless if they can be handled or not, should be logged. Logging helps us keep track of significant events in the application, trace back conditions that led to errors, and even detect malicious activity by users.

Centralizing logs is the collection of log entries that usually are stored in text files along with the application that generates them. Centralization allows us to better analyze logs, especially when dealing with distributed environments where independently accessing each server's logs would take significant time and effort.

In this chapter, we discussed using the stack **Elasticsearch, Logstash, and Kibana** (**ELK**) as a platform to monitor and centralize logs. While this tech stack is one of the most popular, other products fulfill the same responsibilities. All these products follow the same patterns of reading log entries from log files, parsing, transforming, and storing them somewhere, and then using visualization tools to analyze them.

At last, we discussed how to create gradual deployment schedules to help us mitigate the effect and severity of errors introduced by new features or large code changes. The field of A/B testing provides multiple tools and infrastructure to enable applications to be deployed just to a reduced segment of users.

At this point of the book, we have reviewed the majority of basic elements necessary to build the back-end part of software applications. In the next chapter, we will move on to review more advanced topics like the use of frameworks to accelerate the application development.

Resources

- Best application security practices about Logging: **https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sh**

eet.html

- Install ELK stack: **https://www.elastic.co/guide/en/elastic-stack/current/installing-elastic-stack.html**
- Install Logstash: **https://www.elastic.co/guide/en/logstash/current/installing-logstash.html**
- Grok patterns: **https://github.com/logstash-plugins/logstash-patterns-core/tree/main/patterns**
- Docker installation guide: **https://docs.docker.com/get-docker/**
- Docker Compose guide: **https://docs.docker.com/compose/**

CHAPTER 8
Adopting Frameworks

Frameworks take care of code that we would need to write over and over again every time we build a new app. They provide common patterns and enforce best practices. Frameworks allow developers to concentrate on building code that will solve their user's problems and not in technical concerns that show up in software projects repeatedly.

However, frameworks add an extra layer of complexity, requiring developers to learn new interfaces and specifications. Some frameworks are as big as the languages themselves, resulting in a steep learning curve. In this chapter, we will explore the different kinds of problems frameworks address.

Structure

In this chapter, we will learn the following topics:

- What problems do frameworks fix
- Common patterns addressed by frameworks
- Choosing a framework
- When not to use frameworks

Objectives

After completing this chapter, you should have a better picture about:

- What problems do frameworks address?
- Common patterns addressed by frameworks, like MVC, dependency injection, among many other patterns.
- When not to use frameworks.

This chapter is not an extensive guide about how each of the mentioned frameworks works, as an entire book can be written about each. Our goal is to provide heuristics on how and when to choose a framework, so you can then go in-depth on them.

In a broader context, a framework can be any '*recipe*' or a predefined way of doing things. Frameworks are used across industries and even governments to provide guidance to individuals or institutions on how to implement policies or business plans. In this book, when we talk about frameworks, we mean application frameworks.

What problems do frameworks fix?

As seen in *Chapter 1, Building Multi-user Apps*, we can split software requirements into functional and non-functional. The code we write to complete functional requirements is particular to the business case. If we think of building software as ***baking a cake****, functional requirements will define the cake's flavor, shape, colors, and any client-requested customization.*

Non-functional requirements, on the other hand, can be seen as *choosing the right ingredients for the cake*: Choosing the type of frosting, the type of '*sponge*' (the bread part of the cake), and what kind of toppings it should have. In the cake analogy, non-functional requirements are the basic foundations on top of which the cake's functional requirements are built.

If we analyze some of the non-functional requirements (like frosting and bread type) in the cake example, we can see that each of them has its own recipe: A red velvet-based sponge will have a different recipe than a vanilla-based sponge. Also, a cream-cheese frosting will be composed of different ingredients than a classic buttercream frosting. The critical part here is that each of these recipes is independent of the cake's "*functional requirements*": buttercream frosting's recipe does not change whether you bake a wedding or a birthday cake. You can make frosting beforehand knowing that you will be able to use it in both.

Every time we bake a new cake, we can either *prepare our own frosting or buy one already pre-mixed*. Maybe making our frosting will produce a better taste, but it definitely will take longer. Because of this, many bakers prefer to find a good quality, pre-mixed frosting and use it instead of making one every time they have to bake a cake, thus saving time which they can use to work on the customization requested by the clients.

Solving existing problems

For most of the challenges or problems we find while building software applications, there is a big chance that other developers have already designed reasonable solutions. After all, if we think hard enough about any new project, *software applications can be decomposed on simpler components* for which excellent working solutions may already exist.

If the problem is ubiquitous enough, many developers have probably encountered and solved it before. These developers may even have shared their solutions in the form of *open-source libraries*. When working on a budget, these open-source solutions can help us complete our project in time.

At a lower level, when we build software applications, we use an existing coding language. We would rarely build a new coding language exclusively for building a web or mobile application. Since we would rarely build every line of code in an application from the ground-up, bit-by-bit, *the idea of extending and building on top of the existing implementation of common problems is central in software development*.

Note Reinventing the wheel

When talking about adopting common software development patterns, we usually want to use existing implementations of those patterns instead of creating our implementations from scratch.

For instance, instead of implementing custom sorting and searching algorithms, we will use an existing and well-tested library. An example of library like these is the Collections Framework in Java.

There are some exceptions to this heuristic. Teams with very complex requirements, commonly found in large applications, may find that existing frameworks have a scope that limits their capabilities. In these situations, developer teams may want to create their custom implementations, most of the time extending existing frameworks or following the same patterns.

Google Guava is an excellent example of a library created to fill the gaps in existing data structures and collection operations implementations.

Frameworks offer reusable solutions to challenges someone has already encountered and solved.

Frameworks and design patterns

Since frameworks try to encapsulate known good practices, they extensively use **design patterns**.

Much like frameworks, design patterns themselves are reusable solutions to common problems; however, design patterns are abstract concepts. Also, unlike most frameworks, design patterns are not language-specific: Most design patterns can be implemented with any programming language. The most popular design patterns are the "*Gang of four design patterns*", which focuses on the **Object-Oriented Programming** (**OOP**) paradigm. For more details on these patterns, we've included a link in the references section.

Libraries and frameworks

Going back to the cake analogy: Some of the "*recipes*" for non-functional requirements can be "*pre-baked*" ahead of needing them so we can save time when building a new application. These are commonly known as **libraries**: Collections of reusable functions that can be leveraged to build business-specific applications. **Frameworks** usually group one or more libraries, but they are so much more than that.

We can think of taking multiple pre-made ingredients like frosting and sponge and putting them together in a "*cake-baking kit*". This kit produces generic, blank cakes with little effort. These blank cakes then can be "*extended*" with custom-made decorations.

Figure 8.1 shows a high-level view of the cake framework:

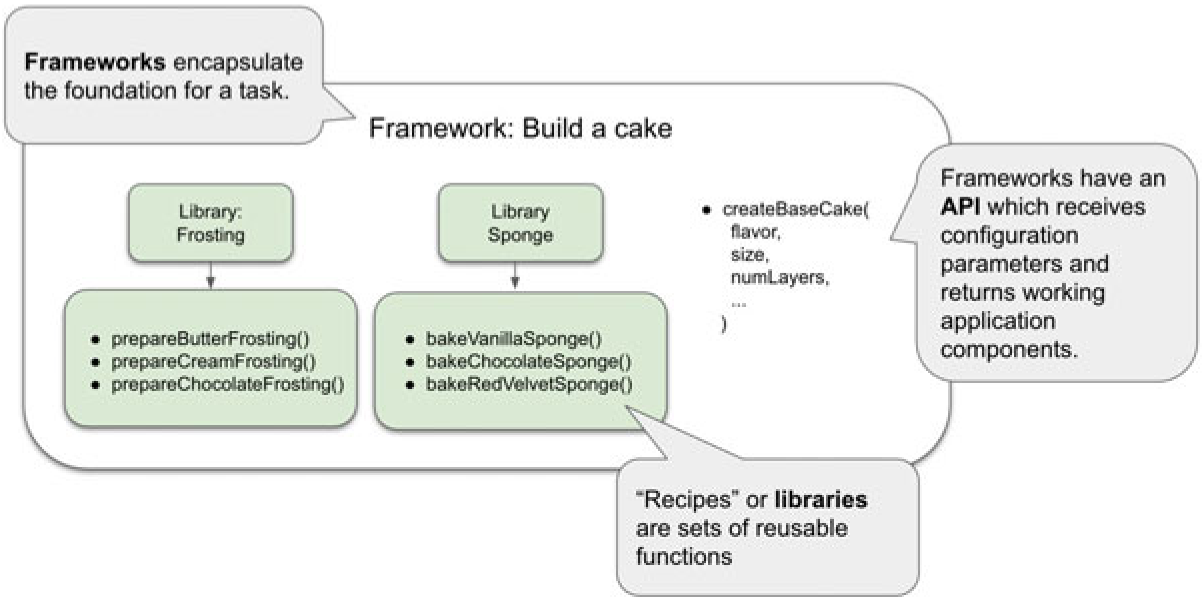

Frameworks encapsulate the foundation for a task.
Framework: Build a cake
Library: Frosting
Library Sponge
prepareButterFrosting()
prepareCreamFrosting()
prepareChocolateFrosting()
bakeVanillaSponge()
bakeChocolateSponge()
bakeRedVelvetSponge()
createBaseCake(flavor, size, numLayers, ...)
Frameworks have an **API** which receives configuration parameters and returns working application components.
"Recipes" or **libraries** are sets of reusable functions

Figure 8.1: *Viewing cake baking as a framework.*

`The framework generates a blank cake using its internal libraries`

These kits would allow bakers to concentrate on the "*functional*" requirements of the cake. In software development, we call these "*kits*" or collections of libraries and patterns *frameworks*.

Frameworks are boxed collections of reusable libraries, patterns, and utilities. *Frameworks abstract-out common patterns and concerns so developers can focus on business requirements*. [Figure 8.2](#) shows how the "*cake framework*" generates a blank cake that allows us to focus on decorations.

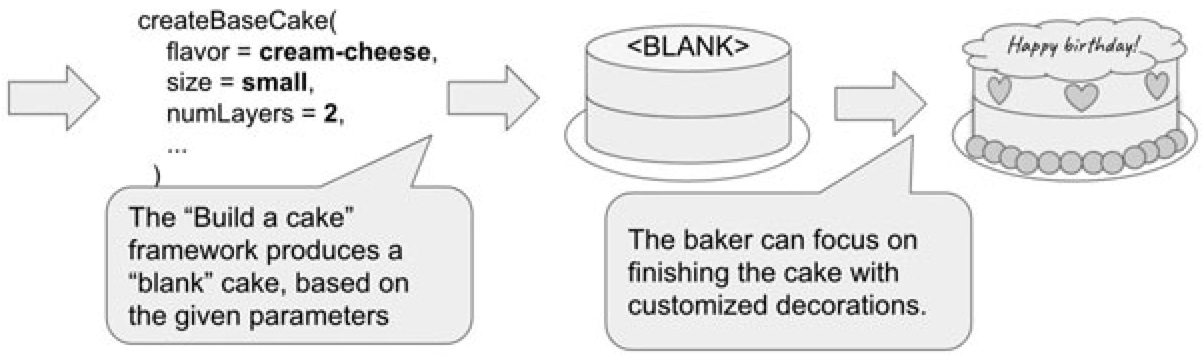

createBaseCake(flavor = **cream-cheese**, size = **small**, numLayers = **2**, ...)
<BLANK>
Happy birthday!
The "Build a cake" framework produces a "blank" cake, based on the given parameters
The baker can focus on finishing the cake with customized decorations.

Figure 8.2: *Frameworks provide the foundation a blank cake so bakers can focus on business requirements*

Let us look at a concrete example to understand the point we are trying to make in a better way.

In *Chapter 2, The Client-Server Architecture*, we discussed how HTTP servers work, and we even went out of our way to build one. We also mentioned that we would not want to use a custom-made server in production because it would require too much work to do it securely and performantly. We used Express to abstract out all the specifics of the HTTP server.

Express is a web framework that helps developers build HTTP-based applications. Here is the code snippet from *Chapter 2, The Client-Server Architecture*:

```
const express = require("express");
const app = express();
const port = 8080;

app.use("/static", express.static("public"));

app.get("/", (req, res) => {
  res.send("<h1>Hello World!</h1>");
});

app.post("/", function (req, res) {
  res.send("Got a POST request");
});

app.listen(port, () => {
  console.log(`App listening at http://localhost:${port}`);
});
```

Notice we did not have to implement **`app.get`** or **`app.post`** ourselves. This pre-baked HTTP server has everything it needs to function correctly while at the same time allows developers to *configure and extend* it. We can take the functions provided by the framework, sprinkle business logic on top, and create a fully working application in no time. The HTTP server inside Express is our blank cake, and the code we use to configure and extend it are the cake decorations.

Pre-building abstractions

Writing software, especially in Object-Oriented Programming, means abstracting features of a real-life object into common attributes shared by all instances of the type of object. For instance, if we build a software model of a car, most of them will have common attributes like a steering instrument (a wheel or a yoke) and a semi-constant number of wheels (usually, four).

Without frameworks, we can build every layer in this abstraction. Using the same example of modeling a car using software, we can create classes like ”Car“,” Wheel“, or” SteeringInstrument“, each with its attributes (e.g., ”Car” can have an attribute to store the size of wheels it uses).

If we assume that modeling cars can be a widespread problem, many people will write their implementations of the ”Car” and “SteeringInstrument” classes.

We can create one instance of the “Car” class and share it with the world. Anyone who wants to model a car in their applications would not have to worry about building these common, abstract classes. They can focus on building the features that distinguish their specific car models.

The idea of pre-building abstractions is that we build the code with the goal of being extended by other developers, instead of building the code to be used by users directly.

Framework’s benefits

In essence, frameworks save time and effort. They encapsulate repetitive tasks, models, and functions and put APIs in front of them so developers can easily access all these features.

Most frameworks have a level of support for *code generation*, which means they can provide a bare-bones version of the project. This code has spaces or placeholders for developers to insert custom-made business logic. Some frameworks call this code generation **scaffolding**.

Frameworks are beneficial to large-scale projects thanks to many of their features:

- Frameworks establish conventions. Teams with many developers tend to struggle to find the best practices that fit their project the best. Frameworks provide clear limits and conventions to allow developers to move away from discussions about things like how to structure a project or what code quality rules to enforce.
- Frameworks tend to be modular, encouraging collaboration. Since frameworks work on the concept of extending software, they are good at separating concerns into multiple components or modules. For instance, the separation enforced by MVC (which we will visit in the next section) allowed front-end developers to work in parallel with back-end developers.

Other advantages of the adoption of frameworks are:

- Large applications can rely on frameworks to reduce their codebase. Frameworks allow them to only include the code needed for fulfilling business requirements.
- Small teams who have starting building a new app can rely on frameworks to speed up their development and release the first version of their app as soon as possible.

Overall, the ultimate goal of a framework is to allow developers to spending time building business-specific features, not spending time building solutions for problems that have been solved already.

Common patterns addressed by frameworks

While theoretically any design pattern can be implemented as a library or framework, some frameworks focus on the most common challenges in software applications.

Automation tools and package managers

Compiling software applications is a multi-step process. For instance, if we need to compile a Java application consisting of three classes, we will run the following command line:

```
javac Order.java Ingredient.java OrderService.java
```

Things start to get complicated when we also need to compile the project using external dependencies that, in Java, are packaged in **Java ARchive** (**JAR**) files:

```
javac -cp lib1.jar;lib2.jar;lib3.jar Order.java Ingredient.java OrderService.java
```

What if each of those JARs needs to import their respective dependencies? How do we enforce that the dependency JARs follow a specific version?

Sure, there are shortcuts, like using the wildcard character ("*") to match multiple files simultaneously, but we still have to manage multiple paths to libraries of dependencies within the project.

The first solution can be to build a script to automatize the compilation process. In a Unix-based environment, applications make heavy use of the '*make*' utility to control the compilation process. The source code for the application will include a file named *Makefile*, which defines the compilation process as a bash script. The following is an example of a *Makefile*:

```
Order.class: Order.java
    javac -cp lib1.jar Order.java
```

We can execute the tasks defined in a *Makefile* by running the command `make`. A *Makefile* can include multiple commands if we need to use multiple compilation targets, or if we need to do other tasks like cleaning existing compiled files.

While '***make***' is a very convenient tool, it has the downside that it is only included in Unix-based systems. Also, all dependency-related tasks (downloading the correct version of the external libraries into a shared directory, managing which version to use, among other concerns) need to be manually configured by developers.

Automation tools

Automation tools like **Maven** and **Gradle** provide both features to run tasks like compilation, packaging projects into JARs or **Web Application Resource** (**WARs**), and manage dependencies to external libraries.

Let us create a simple Java class to showcase how Gradle works. We will create an executable Java class that uses Google's Guava library for storing and printing a list of ingredients.

First, we make sure we have installed Gradle. We have included the link to the official installation guide in this chapter's "*References*" section.

We will create a folder structure of a nested directory with a path of **`src/main/java/hello`** and inside the **`hello`** directory create a class named **`Application.java`** with the following contents:

```java
package hello;

import com.google.common.collect.ImmutableList;
import java.util.List;

public class Application {


    public static void main(String [] args) {
           List<String> items = ImmutableList.of(“cheese”, “tomato sauce”, “sourdough”);

        items.stream()
            .forEach(System.out::println);
    }
}
```

Using Guava is certainly unnecessary, as we can use Java's native classes to achieve the same goal. However, using Guava utilities help us provide an example of an application that has a dependency on an external package.

Note: The directory structure 'src/main/java/' is a convention used by most Java-based projects. Gradle assumes that all the source code for the project is included within that directory.

The directory structure can be changed through the configuration attribute `sourceSets`, within the same `build.gradle` file:

```
sourceSets {
        main {
            java {
                srcDir 'src/java'
            }
            resources {
                srcDir 'src/resources'
            }
        }
    }
```

At the root of our project (the same folder which contains the **`src`** directory), we create a Gradle configuration file named **`build.gradle`** with the following contents:

```
apply plugin: 'java'
apply plugin: 'application'

mainClassName = 'hello.Application'

repositories {
    mavenCentral()
}

dependencies {
    implementation "com.google.guava:guava:19.0"
    testImplementation "junit:junit:4.12"
}
```

Let us evaluate this configuration. The first two lines provide most of the default configuration used for compilation:

```
apply plugin: 'java'
```

The **java** plugging includes all tasks to compile the Java classes inside the **src** directory.

Next, the **application** plugin is applied to the project:

```
apply plugin: 'application'

mainClassName = 'hello.Application'
```

The **application** plugin imports tasks for making this project an executable file. The **mainClassName** attribute allows us to configure which class should be the main executable class.

For package management, we use the **dependencies** attribute, as follows:

```
repositories {
    mavenCentral()
}

dependencies {
    implementation "com.google.guava:guava:19.0"
    testImplementation "junit:junit:4.12"
}
```

In the example, we import the guava JAR version 19.0 from the **com.google.guava** package. We also import the JAR for *Junit*, a framework for unit tests, but only for test classes.

Lastly, the **repositories** attribute indicates which package repository to use. In this case, we rely on Maven's central repository (we have included the URL in the references section). Potentially, we can deploy a private repository to host our libraries and use the **repositories** attribute to point this project to the private repository.

Having created these two files **Application.java** and **build.gradle,** the compilation is simplified to execute the following command from the root of the project:

```
gradle build
```

This command will link dependencies and compile **`Application.java`** (and all other Java classes, if we had more). Variations of the command also allow us to package everything in a JAR.

We can use the following command to execute **`Application.java`** directly:

```
gradle run
```

The output should look like this:

```
% gradle run
> Task :run
cheese
tomato sauce
sourdough
BUILD SUCCESSFUL in 469ms
2 actionable tasks: 2 executed
```

Using Gradle, we have automated the most common tasks for this Java project.

Maven is another popular automation tool. It follows similar patterns to Gradle, except that Maven relies on XML files for configuration. We have included a link to Maven's official tutorial in the *References* section.

Automation tools may not be considered frameworks by some people, but they are part of the toolset most proficient developers are expected to know. These tools abstract out the linking, compilation, and package processes under a simple API that simplifies the whole process for developers.

Native package management

Some code languages are integrated with standard ways to import external dependencies into the project. For instance, *Rust* uses *Cargo* to configure dependencies, configure the compilation process, and add metadata to the project. A **`cargo.toml`** file looks like this:

```
[package]
name = "hello_world"
version = "0.1.0"
edition = "2018"

```

```
[dependencies]
time = "0.1.12"
```

NodeJS is also shipped with a package manager, *NPM*. In *Chapter 2, The Client-Server Architecture*, for the example of a web server using the *Express* framework, we generated the following **`package.json`** file:

```
{
  "name": "express-server-example",
  "version": "1.0.0",
  "description": "",
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.17.1"
  }
}
```

NodeJS may not require compilation, but we still can use **`package.json`** to define a list of installed dependencies and multiple "scripts" that we can use to perform actions like starting the server or pre-processing static resources.

Whether the automation tool is shipped with the code language's compiler or not, we can see that all these tools share some attributes:

- A section where we can configure a set of scripts to execute actions like compiling source code, packaging the project, or executing an application.
- A section to define a list of dependencies this project uses. For each of these dependencies has to be explicitly detailed the dependency version and a namespace to avoid collisions between dependencies.

Another notable mention of a native package manager is Python's ***pip***, which allows Python developers to install external dependencies or libraries in their development environments.

Handing web requests (e.g. Spring MVC, Django)

We have already discussed the idea behind using frameworks to support HTTP-based applications. We have seen how the Express framework allows developers to create fully working back-end code for web applications with just a few lines of code.

Here, we will discuss two other common examples of frameworks for HTTP-based applications, along with a classic pattern: The **Model-View-Controller** (**MVC**) pattern. The two frameworks to discuss are Java's *Spring MVC* and Python's *Django*.

How frameworks are born: The use of the Spring framework

One of the most well-known frameworks in the Java ecosystem is Spring.

Created in 2003, Spring was a response to the complexity of the Java Enterprise specifications. Frameworks like Spring (and Struts before it) offered alternatives to the challenge of building enterprise apps in a more straightforward and open-source way. What started as a single framework later became an umbrella of different projects, focusing on individual concerns for building applications using JVM-based languages.

Spring operates in central patterns like **Inversion of control** (**IoC**) and dependency injection. However, each Spring project implements some domain-specific patterns.

Note search term: Inversion of control (IoC)

IoC is a design pattern based on delegating the creation of class instances to the framework. Developers create configuration files that the framework then uses to create instances of the managed services.

Dependency injection, which we used in *Chapter 3, "Designing APIs"* to inject different implementation types for the same interface, is one type of IoC. Using the dependency injection pattern, we build classes that receive a reference to their external dependencies through a

constructor or a getter. The framework has the job of creating the instances of the dependencies and injecting them into the other classes that need them.

Spring's evolution led to an offer of a diverse set of tools to build cloud-based applications: Spring Boot, Spring Security, and Spring Data, among other projects.

About MVC

One of the most popular parts of Spring was Spring MVC. Spring MVC allowed developers to build "*fat*" web applications (server-side applications that included a user interface) by only implementing some specific APIs provided by the framework.

The **Model-View-Controller** (**MVC**) design pattern is not Spring-specific. It consists of organizing applications (mostly web-based) to encourage separation of concerns. This clean separation was a response to the complexity introduced by web applications where business logic would be mixed with data fetching and presentation logic (like in the PHP example we saw in *Chapter 2, "The Client-Server Architecture"*). Each layer of the MVC pattern has its own goal, as shown in *Figure 8.3*:

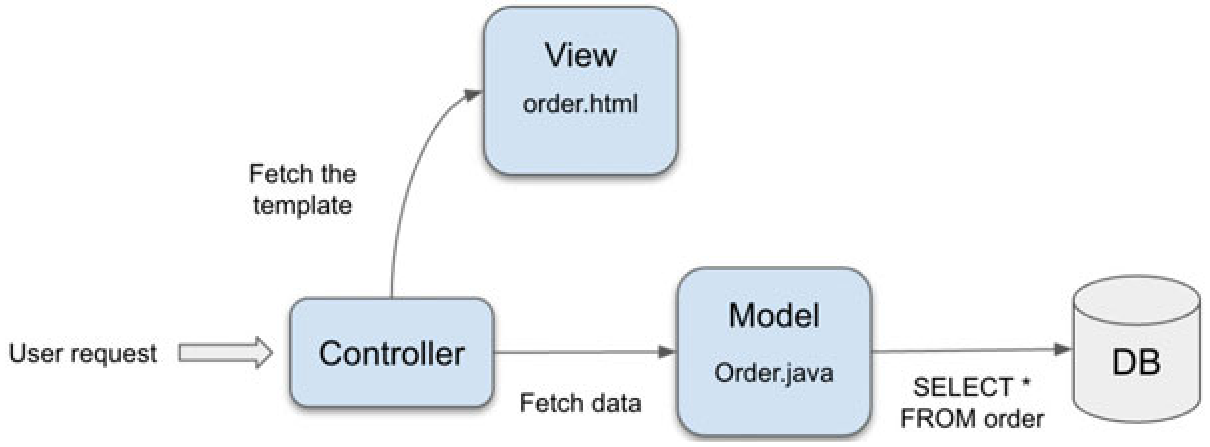


Figure 8.3: *Basic flow of the MVC design pattern*

The concepts of *Model*, *View* and *Controller* have definitions that sometimes vary slightly from developer to developer. However, the following definitions are generally accepted:

- **Model**: The model layer groups all data definitions. In Object-Oriented Programming, models are the classes or "*entities*" in the application: "*Order*", "*Ingredient*" and so on.
- **View**: The view layer collects all files related to the user interface. A typical pattern used in the view layer is using templates, which are predefined views with placeholders used to present the dynamic data. All code related to presentation concerns such as templates, JavaScript, CSS and other static resources is defined in separate files. A well-known heuristic is that the view layer should not contain business logic.
- **Controller**: This is the glue between the model layer and the view layer. The controller defines all user interactions (like HTTP requests), fetches the data the request needs from the data definitions in the Model layer, and then picks the correct template from the View layer, filling it with the fetched data.

By separating the presentation logic (the code required to render and present the application's user interface) and the business logic, the MVC pattern allows front-end developers to work in parallel on the same project as back-end developers, without getting source code conflicts.

Dependency Injection

Before moving into specific uses of Spring, we must describe how its dependency injection capabilities work.

At a first sight of these examples, it might feel like we are over complicating things; and if our application consists of only one service, this feeling may reflect reality. However, these patterns *prevent rework* as applications grow: We can rewrite services in isolation by creating new implementations without refactoring other services that depend on them. Teams can work in parallel to write new services without affecting exciting implementations.

We discussed dependency injection when we talked about using interfaces in *Chapter 3, Designing APIs*. Instead of having a class create instances of its dependencies, an external actor (a factory or a container) creates the dependency instance and passes it to all the classes needing it. In many cases, the injection is done through the class constructor:

```java
class ReportService {
    private PrintService printService;

    public ReportService(PrintService service) {
        this.service = service;
    }

    // …
}

interface PrintService {
    void printDocument(Document doc);
}

class PrintServiceImpl implements PrintService {
    @Override
    void printDocument(Document doc) {
        // create a concrete implementation
    }
}
```

As discussed earlier, we choose to use instances when we may have different implementations for the same services. In this example, we may want to use an implementation for **`PrintService`**, which is *stubbed* for testing.

A typical pattern used for dependency injection is the ***factory pattern***, where specialized classes called "*factories*" create the instances for each class:

```java
class PrintServiceFactory {
    PrintService createInstance() {
        return new PrintServiceImpl();
    }
}


class ReportServiceFactory {
    ReportService createInstance(PrintServiceFactory printServiceFactory) {
        PrintService printService = printServiceFactory
            .getInstance();

        return new ReportService(printService);
    }
}
```

The factory pattern encapsulates the process of creating instances of individual dependencies or services and decouples classes from their dependencies.

Each factory is in charge of choosing the correct implementation to the service (for instance, inject the real implementation if this application is running in the production mode or using the stubbed implementation if we are executing tests).

Then, the application's container takes care of wiring everything together, as follows:

```
class Container {
    public static void main(String[] args) {
        PrintServiceFactory printServiceF = new PrintServiceFactory();
        ReportServiceFactory reportServiceF = new ReportServiceFactory();

        ReportService reportService = reportServiceF
            .getInstance(printServiceF);

        // do something with reportService
    }
}
```

Notice that in this example we are not using Spring. We are building a custom *Continer* class to illustrate the way Spring's dependency injection works.

Spring offers generic factories and containers that are managed by the framework (hence the *inversion of control*). We provide Spring with a configuration to describe how each dependency and service needs to be set up, and Spring takes care of wiring everything together.

The configuration we provide to Spring can be written using the following three different formats:

- XML-based configuration
- Code-based configuration
- Annotation-based configuration

Spring's XML configuration

Spring's XML configuration file details what instances or "*beans*" it should create. The following is an example of this configuration:

```
<bean id="printService" class="com.example.myapplication.PrintService"/>

<bean id="orderService" class="com.example.myapplication.OrderService">
    <constructor-arg index="0" ref="printService"/>
</bean>
```

Using Spring's XML configuration requires almost no changes to our service classes: In the example, we pass an instance of **`PrintService`** through the **`OrderService`** constructor.

In Spring, the container is called **`ApplicationContext`**. Having provided the XML configuration file, we can get a fully configured instance of a dependency through the Application Context:

```java
public static void main(String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationContext("spring-config.xml");
    OrderService orderService = context.getBean(OrderService.class);

    // orderService is ready to be used
}
```

We can see how this makes our lives easier: Spring takes care of instantiating and wiring all services, just as the *Container* class was doing in our example; with Spring we can get rid of the *Container* class.

The advantage of using XML configuration files was that we can compile our application once and then change its behavior by changing the XML files directly; all without having to re-compile and re-package the application.

Code-based configuration

An alternative (and more popular) option is to use Java annotations to configure each class. This approach requires us to create a configuration class similar to the following:

```java
@Configuration
public class Config {

    @Bean
    public PrintService printService() {
        return new PrintServiceImpl();
    }

    @Bean
    public OrderService engine() {
        return new OrderSerivce(printService());
    }
}
```

Again, we can use the Application Context to get instances of the services:

```
public static void main(String[] args) {
    ApplicationContext context = new AnnotationConfigApplicationContext(Config.class);
    OrderService orderService = context.getBean(OrderService.class);
}
```

Annotation-based configuration

The annotation-based configuration is an extension of the code-based configuration.

Spring is smart enough that we can give it a package within our project, and it will scan for any beans that need to be instantiated and wired. The following configuration class uses the "`@ComponentScan`" annotation:

```
@Configuration
@ComponentScan("com.example.myapplication")
public class Config { }
```

Spring will look for classes annotated with `@Component` (or sub-types of that same annotation like `@Service` or `@Repository`) and will create instances of them. Then, it will inject dependencies into attributes annotated with `@Autowired` or `@Inject`:

```
@Component
class PrintServiceImpl implements PrintService {
    // …
}

```

```java
@Component
class ReportService {
    @Autowired
    PrintService printService;

    // …
}
```

Spring’s component scan speeds up development, as now we don’t need to imperatively configure how each class needs to be instantiated.

Spring MVC

Spring MVC was once the cornerstone of the Spring Framework initiative. Spring’s use of dependency injection and wide use of conventions makes it really easy for us to build a fully working web application.

First, we define a model to define the application’s state:

```java
package com.example.pizzaplace.model;

import java.util.LinkedList;
import java.util.List;

public class Order {
    private String orderName;
    private List<String> ingredients;

    // add getters and setters
    // …
}
```

Models are plain Java objects (POJO), but we can extend them using annotations to map their attributes to database columns. We will discuss this case in the ORM section of this chapter.

Having defined a model, let us create a view. Spring MVC uses a templating engine called ***Thymeleaf***, which is a super-set of HTML:

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Getting Started: Serving Web Content</title>
       <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8" />
</head>
<body>
    <h1 th:text="'Order Name: ' + ${orderName}"></h1>
    <ul>
                <li th:each="ingredient: ${ingredients}"
th:text="${ingredient}"></li>
    </ul>
</body>
</html>
```

Unlike HTML, Thymeleaf's templates are dynamic: They offer a placeholder where we can inject data coming from our model. In this example, the template uses the "**`th:each`**" attribute to iterate over a list of ingredients, printing them one at a time inside an unordered HTML list. It also uses "**`th:text`**" to print the name of the order, which is passed through the "**`orderName`**" variable. The template will be compiled into static HTML before it's returned to the client.

Finally, the glue that binds the model and the view: The **`controller`**:

```
package com.example.pizzaplace.controller;

import com.example.pizzaplace.model.Order;
import com.example.pizzaplace.service.OrderService;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class OrderController {

    @Autowired
    OrderService orderService;

    @GetMapping(“/order”)
    public String getOrderHandler(Model model) {
        Order currentOrder = orderService.getOrder();

        model.addAttribute(“orderName”, currentOrder.getOrderName());
        model.addAttribute(“ingredients”, currentOrder.getIngredients());
        return “order”;
    }
}
```

We use Spring’s annotation **`@Controller`** to indicate that this class should be considered a Spring MVC controller. When the Spring context scans all classes in the project and finds **`OrderController`**, it knows what role it will play in the application.

We also use the **`@Autowired`** annotation to inject the **`OrderService`** dependency into the controller. We will look at the definition of this service next, but let us finish discussing the controller.

The **@GetMapping("/order")** annotation tells Spring MVC that all GET requests done to the **/order** path should be handled by this method.

The **getOrderHandler** handler function receives an instance of the class **Model**. This is the view model that we will use to fill the placeholders in the view. Using the function **model.addAttribute** we set values for the **orderName** and **ingredients** variables in the template.

We use the injected service **OrderService** to fetch a single instance of the order, which we then use to set the variable values in the view model.

Finally, the definition of the **OrderService** class consists of one POJO marked with the **@Service** annotation:

```java
package com.example.pizzaplace.service;

import com.example.pizzaplace.model.Order;
import org.springframework.stereotype.Service;

@Service
public class OrderService {
    public Order getOrder() {
        Order newOrder = new Order("Pedro's order");

        newOrder.addIngredient("Cheese");
        newOrder.addIngredient("Tomato sauce");

        return newOrder;
    }
}
```

Just like with the **@Controller** annotation, the **@Service** annotation marks the **OrderService** class so, when Spring initializes its context and scans all classes in the project, it knows that it should create an instance of **OrderService** and inject it any other classes which use the **@Autowired** class on variables of type **OrderService**.

In this example, the service hard-codes values for an instance of the `Order` class, but in a real-world application, the service would query a database or call other external services to fetch real data.

We can now run this application using the following two approaches:

- Package it into a WAR file and deploy it into an application server like Tomcat or Glassfish.
- Include Spring Boot as a dependency and have it instantiate a server; similar to how Express works.

We will discuss deploying applications in the next chapter.

Note Spring Webflux

One hard limitation of Spring MVC has been that it operates in blocking Java's Servlet operations. This means that each thread that is created by the server when it receives requests from the clients will not be released until the response is returned. If a request requires fetching data from a database or any other external service, the thread may be blocked for a long time, reducing the server's capacity to respond to other requests.

While improvements have been made on the Java side to improve the Servlet API, Spring has created a new service called Spring WebFlux, which is fully non-blocking.

Spring WebFlux runs on top of non-blocking servers like Netty or the latest versions of the Servlet API. Its API is fully functional and reactive (the framework does not block thread execution, but it "reacts" to changes like getting a response back from a database).

Spring WebFlux provides reactive support for some databases like PostgreSQL or MongoDB, and it is currently working on providing reactive support for more services.

Spring MVC is still suitable for many projects, but Spring is pushing strongly for development teams to adopt a more reactive style of programming with WebFlux.

More MVC: Express

In previous examples, we used the NodeJS Express framework to build REST APIs. In addition to returning JSON objects, this framework also returns template-based views.

A common templating option for Express is Handlebars, a super-set of HTML that is similar to Thymeleaf. Handlebards can define placeholders for dynamic data in an HTML document. For instance, the following **`home.hbs`** file is the view template for this example:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Example App</title>
</head>
<body>
    <h1>Order name: {{order.orderName}}</h1>

    <ul>
        {{#each order.ingredients as |ingredient|}}
            <li>{{ingredient}}</li>
        {{/each}}
    </ul>
</body>
</html>
```

Handlebars will compile that template along with data returned from the controller, which in this case is defined by the Express method **`get`**:

```
import express from 'express';
import { engine } from 'express-handlebars';

const app = express();

```

```
app.engine('.hbs', engine({extname: '.hbs'}));
app.set('view engine', 'handlebars');
app.set('view engine', '.hbs');
app.set("views", "./views");

app.get('/', (req, res) => {
    res.render('home', {
        order: {
            orderName: "Pedro's order",
            ingredients: ["Cheese", "Tomato sauce"]
        },
        layout:false});
});

const port = 3000
app.listen(port, () => console.log(`App listening to port ${port}`));
```

This is very similar to how Spring MVC operates; we use **`res.render`** to choose the right handlebars template to render, along with the data that should be used to fill the view's placeholders.

We can clearly see the shared patterns between both MVC frameworks, patterns that we can also see in other frameworks like Python's Django:

- A controller that matches the request path and the HTTP method to a function handler.
- Controller handlers that receive request parameters (through a POST body request or through query parameters) and fetch data using the model layer.
- The controller handler retrieves its assigned view, populates the template's dynamic data using the model, producing HTML. That same HTML is then returned back to the client.

The downfall of MVC

During the glory days of the MVC pattern, most dynamic web-based applications were server-side rendered. MVC-based applications would retrieve templates, fill them with data during run-time, and return dynamically created HTML before returning the server response.

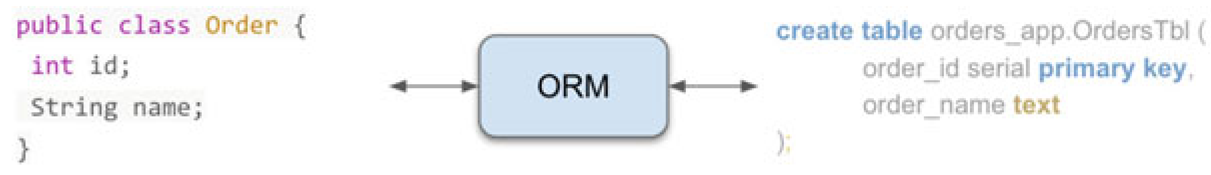
Fast-forward a few years later, and front-end frameworks like Ember, React, and Angular are introduced. As these tools matured and became stable enough, more people started to write front-end applications outside the web server. Suddenly, the MVC framework lost most of its “V” part, as back-end templates were deprecated in favor of JavaScript-based components. The “C” part also became suddenly unfit for the task: Since controllers are the glue that binds models and their views, without views, there were no more things to glue together.

Nowadays, MVC can be considered an outdated pattern. We are talking about it here because many applications exist (especially in enterprise projects) that still rely on this pattern. Also, since it provides a clear translation of a design pattern into a framework, it is an excellent example for this section.

Database access with ORMs

Database operations are one of the most ubiquitous tasks for a software application. As seen in *Chapter 4, End-to-end Data Management*, for many years, the most popular modeling format for databases was the relational model. As OOP became the reigning programming paradigm, an impedance mismatch between the relational data format and the object-relational model happened.

Object-relational mapping (**ORM**) frameworks were born out of the need to convert relational data into objects and back into relational form again. This operation is represented in *Figure 8.4* as follows:



```
public class Order {
 int id;
 String name;
}
```

```
create table orders_app.OrdersTbl (
        order_id serial primary key,
        order_name text
);
```

Figure 8.4: *ORM frameworks map data between objects and relational tables*

One of the key ideas behind ORMs was to separate concerns: to allow application developers to operate on objects without worrying about SQL and allow SQL developers to design databases without worrying about the data representation in an OOP model.

JPA and Hibernate

ORMs became especially popular in the enterprise world. This popularity even led languages like Java to implement an official API for ORM frameworks to implement. In Java, there is JPA (Jakarta Persistence API or, as it was formerly known, Java Persistence API), and the most popular vendor who provided an implementation for JPA was Hibernate.

Note Java and APIs

In an effort to maintain itself independent of implementations provided by specific companies or vendors, the Java specification language provided a set of official APIs: Interfaces that any vendor can implement to provide specific functionality to the language without having to tie the official codebase to that vendor.

The most common example of these APIs is the Jakarta Persistence API (JPA) or the Java Naming and Directory Interface (JNDI). Besides Hibernate, other vendors like Spring Data and Oracle’s TopLink provide JPA implementations.

To achieve its goals, ORMs define an idiom where developers can create configurations to indicate to the ORM how the data should be serialized back and forth. The following XML document is an example of **`persistence.xml`**, the configuration file used by JPA (and by extension by Hibernate) to configure the mapping between a relational database and Java classes:

```
<?xml version=”1.0” encoding=”UTF-8”?>
<persistence xmlns=”http://xmlns.jcp.org/xml/ns/persistence” version=”2.1”>
    <persistence-unit name=”jpa_provider” transaction-type=”RESOURCE_LOCAL”>
```

```
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <class>hibernate.example.model.Order</class>
        <class>hibernate.example.model.Ingredient</class>

        <properties>
            <property name=”hibernate.connection.url” value=”jdbc:postgresql://localhost:5432/postgres”/>
            <property name=”hibernate.connection.driver_class” value=”org.postgresql.Driver”/>
            <property name=”hibernate.connection.username” value=”postgres”/>
            <property name=”hibernate.connection.password” value=”mysecretpassword”/>
            <property name=”hibernate.dialect” value=”org.hibernate.dialect.PostgreSQLDialect”/>
            <property name=”hibernate.show_sql” value = “true” />
        </properties>
    </persistence-unit>
</persistence>
```

This configuration file contains mapping and database connection details:

- The location of the database (PostgreSQL, in this example) is written in the format of a JDBC connection string.
- A username and password are used to access the database.
- The dialect: A class that contains the mappings between Java classes and database types, which allows Hibernate to generate SQL optimized for a kind of relational database (again, PostgreSQL in this example).

Notice these two lines:

```
<class>hibernate.example.model.Order</class>
<class>hibernate.example.model.Ingredient</class>
```

Here, we are indicating to Hibernate the classes it should use for mapping database tables. This pattern is similar to mapping beans in Spring's XML configuration classes. Let us take a look at these classes.

Mapping tables to entities

The following SQL script describes the two hypothetical relational tables we have seen earlier in this book: The model orders (which is defined as **`OrdersTbl`**) to model a restaurant order and food items, which is defined as **`FoodItemsTbl`** to model specific food items included in the order, like a pizza:

```
create table orders_app.OrdersTbl (
    order_id serial primary key,
    order_name text
);

create table orders_app.FoodItemsTbl (
    item_id serial primary key,
    item_name text,
    order_id_fk int references orders_app.OrdersTbl
);
```

Figure 8.5 displays the Entity-Relationship diagram for the preceding SQL tables:

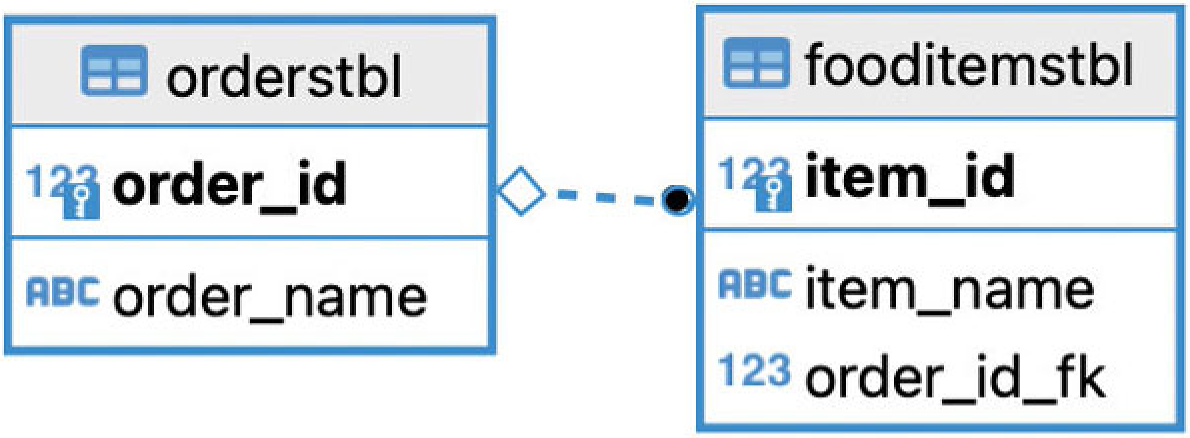


Figure 8.5: *Entity-relation model for the tables OrdersTbl and FoodItemsTbl*

As seen in *Figure 8.5*, the **`FoodItemsTbl`** table is related to the **`OrdersTbl`** through the foreign key **`order_id_fk`**. This relationship is a one-to-many relationship: One order can be related to multiple food items, and a food item is only assigned to one order.

In the JPA model, mapping database tables into classes is done through annotations. Classes that are annotated with the JPA interface are called **entities**:

```
package hibernate.example.model;

import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.SequenceGenerator;
import javax.persistence.Table;

import java.util.List;

```

```
@Entity  // mark this class as an entity
@Table(name = “OrdersTbl”, schema=”orders_app”)
public class Order {

    @Id
    @GeneratedValue(
      generator = “order_id_seq”,
      strategy = GenerationType.SEQUENCE
    )
    @SequenceGenerator(
      name = “order_id_seq”,
      sequenceName = “orders_app.orderstbl_order_id_seq”
    )
    @Column(name = “order_id”)
    int id;

    @Column(name = “order_name”)
    String name;

    @OneToMany(cascade = CascadeType.ALL, mappedBy = “id”)
    List<FoodItem> ingredients;

    // regular getters and setters for each attribute
    // …
}
```

We annotate the class with **`@Entity`** to allow JPA to know that this is a database entity and that Hibernate should manage it.

We map database columns with class attributes using the **`@Column`** annotation. For instance, we map the class attribute **`name`** to the table column **`order_name`**. If both the column and the attribute have the same name, we can even skip the **`name`** attribute in the **`@Column`** annotation.

The special annotation **`@Id`** lets JPA know that the attribute is a primary ID. The **`@GeneratedValue`** and **`@SequenceGenerator`** annotations allowed hibernate to know that the auto-generated values for these keys should be based on PostgreSQL's sequences (which are generated automatically by the database when we use the SQL data type **`serial`** to define the primary key). We can use these PostgreSQL-specific terms thanks to the dialect configured in **`persistence.xml`**.

The modeling mismatch happens when we need to model relationships. While the **`OrderTbl`** table does not have any reference to the **`FoodItemsTbl`**, the **`Order`** class requires an attribute to refer to the list of food items related to that specific order using the **`@OneToMany`** annotation.

In OOP, relationships are modeled as in document-based models: Using nested objects and models. To expressively model the relationship of **`order`** to all its food items, the N:1 relationship has to be explicitly configured.

The **`FoodItem`** class follows a similar pattern but uses the opposite type of relationship to indicate its link to the **`Order`** entity:

```java
package hibernate.example.model;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.SequenceGenerator;
import javax.persistence.Table;

@Entity
```

```
@Table(name = “FoodItemsTbl”, schema=”orders_app”)
public class FoodItem {
    @Id
        @GeneratedValue(generator = “fooditemstbl_id_seq”, strategy = GenerationType.SEQUENCE)
         @SequenceGenerator(name = “fooditemstbl_id_seq”, sequenceName = “orders_app.fooditemstbl_item_id_seq”)
    @Column(name = “item_id”)
    int id;

    @Column(name = “item_name”)
    String name;

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = “order_id_fk”)
    Order assignedOrder;

    //more regular getters and setters
    //…
}
```

We use the same annotations to configure the mapping of the primary key and columns into the class attributes as we did with the **`Order`** entity. For the relationship, however, we use **`@ManyToOne`** to indicate that **`FoodItem`** has an N:1 relationship with **`Orders`**.

The model mismatch gets even more complicated when we think about querying data in multiple scenarios:

- Query the **`FoodItemTbl`** table and fetch the orders for each **`FoodItem`**.
- Query the **`FoodItemTbl`** table without fetching the order table.

If we query the database using this entity, what behavior of these two will we get?

We can configure the fetching behavior with the **`fetch = FetchType.EAGER`** property:

- Eager means that every time we fetch the **`FoodItem`** entities, we will also query **`OrderTbl`** to fetch their corresponding orders.
- Lazy will only query the **`OrderTbl`** until the order is actually retrieved from the entity, thus reducing the amount of data that needs to be fetched if the order is never really accessed.

Notice that, annotations apart, there is nothing special about these classes. They are plain Java objects that are just annotated with the JPA-specific API. This means that these entities can extend other classes, implement interfaces, and so on.

Inserting and querying data

ORM frameworks provide utility classes to perform data operations on the database. In the case of JPA, the **`EntityManager`** class provides an API to persist data contained in objects or to query data using the Hibernate **`dialect`** we configured in **`persistence.xml`**:

```
// create an entity manager using the configuration in persistence.xml
EntityManagerFactory emf = Persistence
    .createEntityManagerFactory("jpa_provider");
EntityManager em = emf.createEntityManager();
```

We use the persistence unit **`jpa_provider`** -which is defined in **`persistence.xml`** to create an instance of the **`EntityManager`**. The entity manager will be configured with all the attributes defined in the XML file.

We can create new records in the database using the **`EntityManager`** instance:

```
EntityTransaction userTransaction = em.getTransaction();

// begin a SQL transaction
userTransaction.begin();

```

```
// Create instances of Order and FoodItem
Order newOrder = new Order();
newOrder.setName("Pedro's order");
FoodItem item = new FoodItem();
item.setName("Lasagna");

// Initialize the relationship between both entities
List<FoodItem> items = new ArrayList<>();
items.add(item);
newOrder.setIngredients(items);
item.setAssignedOrder(newOrder);

// save both entities: the food item is nested in the order entity
em.persist(newOrder);

// commit the transaction to the database
userTransaction.commit();

// close the entity manager
em.close();
emf.close();
```

The following are the highlights of this code snippet:

- We start an SQL transaction. This is done in cases where we need to do multiple database operations in one single action. For this example, a transaction may not be required, but we use it to illustrate JPA/Hibernate’s API in a better way.
- We create instances of the **`Order`** and **`FoodItem`**’ classes and xsave them using the persist method of EntityManager. The entities are marked as persisted by Hibernate, but since we are running them inside a transaction, they are not yet sent to the database.

- We execute the transaction's **`commit()`** method for data to propagate the persisted entities to the database.

After executing this code, we should be able to see new records stored directly in the PostgreSQL database.

Since we set the **`hibernate.show_sql`** attribute to *true* in **`persistence.xml`**, the application's console should show the dynamically generated **`SQL INSERT`** statements:

```
Hibernate: select nextval
('orders_app.orderstbl_order_id_seq')
Hibernate: select nextval
('orders_app.fooditemstbl_item_id_seq')
Hibernate: insert into orders_app.OrdersTbl (order_name,
order_id) values (?, ?)
Hibernate: insert into orders_app.FoodItemsTbl (order_id_fk,
item_name, item_id) values (?, ?, ?)
```

Notice that because we configure the entities to use sequences to generate primary key values, Hibernate automatically queries the database to get the latest value in the sequence and generate the next value.

Next, let us look at the code to query the database:

```
// query and print all food items
List<FoodItem> ingredientQR = em
            .createQuery("select C from FoodItem C», FoodItem.class)
    .getResultList();

ingredientQR.stream().forEach(System.out::println);

em.close();
emf.close();
```

We use the **`createQuery`** method of EntityManager to query the database. This method receives the following two parameters:

- A query that is written in the dialect configured in the **persistence.xml** attribute **hibernate.dialect**. For this example, we use the Java entity name instead of the actual SQL table name **FoodItemTbl** to fetch the list of food item records.
- The entity class that will be used to map the query's response. In this case, we are using the **FoodItem** class to deserialize the response.

The result of the query is a Java List of the entity **FoodItem**. Then, we print the list contents using Java's stream method **forEach**.

More ORMs: Python's Django ORM

Python's Django uses an ORM pattern to fetch data from the database into its models; those same models that live in the **model** layer of its MVC structure.

The following is an example of a Django model. If we look closely at it, we will recognize the same patterns we described for JPA:

```
from django.db import models

class Order(models.Model):
      id = models.IntegerField(primary_key=True)
      name = models.TextField()

   class Meta:
      db_table = “OrderTbl”

class FoodItem(models.Model):
      id = models.IntegerField(primary_key=True)
      name = models.TextField()
      assigned_order = models.ForeignKey(Order, on_delete=models.CASCADE)

   class Meta:
        db_table = "FoodItemTbl"
```

Instead of annotations, this example uses Django's **`IntegerField`**, **`TextField`**, and **`ForeignKey`** functions to map the database columns into this model's attributes.

To persist an instance of **`Order`**, we can use the "*save*" method provided by **`models. Model`**, the parent class our entities extend:

```
order = Order(name="Pedro's order")
order.save()

food_item = FoodItem(name="Pizza", assigned_order=order)
food_item.save()
```

Contrasting JPA with Django's ORM allows one to find the patterns shared by both and to distinguish the characteristics that define ORMs.

The downsides of ORM

One disadvantage of ORMs is that they add an extra abstraction layer to the communication between the database and the application. The ORM framework works so hard to hide the SQL details from the Java developers that the mapping sometimes feels like magic. This abstraction can be problematic when the data access operations do not behave as expected.

The abuse of abstraction layers forces developers to build a mental model that is neither OOP nor SQL. The use of Hibernate dialects is the perfect example of the extra knowledge developers need to acquire to use ORMs, knowledge which would not be necessary if we manually mapped the relational tables.

Another complication of ORMS is the generation of dynamic SQL queries. While Hibernate and other ORMs will try to optimize the queries they generate, they are no match for the highly-performant queries an experienced developer can write.

Choosing a framework

Some problems may have multiple solutions. Web client development has many JavaScript frameworks like React, Angular, Vue, and Ember, among other tools. How do we choose which frameworks to learn and use?

In *Chapter 1, Building Multi-User Apps*, we discussed how building applications without having a problem to solve first could lead to failure. Similarly, choosing a framework without thinking about the problems they solve can lead to wasted effort. *Finding the solution to a problem that does not exist is hard, if not impossible*.

Always guided by our requirements, mainly non-functional requirements, we should shop around and see what existing frameworks offer. All of them will have trade-offs, and we should find the right trade-offs we can live with as we build our project.

A framework may be powerful and easy, but it may only support specific platforms, code languages, or patterns. Do these limitations work well in our application? For instance, a framework that only supports web clients may be good enough for us if we don't plan to build native applications.

The more minor, more straightforward, and more focused the framework is, the easier it will be to adopt it.

Ideally, we should build our applications without using any frameworks. We should only consider frameworks as we find problems that are too hard or take too long to fix ourselves (like building a web server!).

This bottom-up approach (start with no framework and add one as you need) will guarantee that we only add the external dependencies we need. As we gain more experience designing systems, we will identify the cases where we need a framework before our project begins.

The impact of community

One aspect to consider when adopting a framework is *how well the framework is maintained*. Many frameworks are open-source, and they need to be maintained by their community members. Whether their bugs and weaknesses are correctly addressed depends on how active their communities are.

Frameworks that are often maintained and updated have benefits:

- It is easier to find bugs when many eyes point to the framework, not just ours.
- Bug remediation is also faster when an open-source tool has an active community.
- New features will be added as needed, increasing the robustness of these tools.

An active community around a framework often translates to *well-documented features*; documentation is critical when our team tries to learn and use a new framework. A poorly documented framework will lead to a steep learning curve or even project failure.

However, as we will explore in the following section, sometimes the right question is not which framework to use but if we should use a framework at all.

When not to use frameworks

For all their good, frameworks are not always the right solution. As mentioned in the previous section, there is no good reason to find a solution if we have no problem to solve.

Frameworks can be as small as one or two libraries or as large and complex as the specification of many code languages themselves. Varying complexity in frameworks means that in addition to knowing the API for the code language they are using in their project, developers also need to understand the framework's inner workings and API.

Some frameworks implement their architectures on specific patterns, meaning the developer needs to be proficient in these concepts to take advantage of the framework's benefits fully.

Learning the framework instead of using the basics

A current issue in the JavaScript community is that new developers are learning how to use frameworks before learning how the underlying technology (in this case, plain JavaScript) works.

Inexperienced developers try to ramp up their knowledge as fast as possible. Often, they will skip the lessons on building things without frameworks and jump directly into learning the framework's API.

When the use of a framework is prevalent, like in the case of Facebook's React or Java's Spring, these same inexperienced users will often confuse what is part of the framework and what is part of the language specification.

Note Javascript frameworks provide such good examples for this section because it is ubiquitous for people to over-complicate simple projects by building web applications using frameworks.

For instance, primarily static web applications would be easier to implement with pure Javascript. However, as React is popular among web developers, people will blindly include it in their projects, often resulting in increased effort and delayed deadlines if few, if any, advantages.

The simpler our codebase is, the easier and cheaper it is to maintain.

At the beginning of our careers as software developers, it is better to learn how to do everything with no frameworks at all. That knowledge will exponentially speed up the learning process of any framework's specifics (not just Spring or React or the popular framework in turn) later in the future.

Adding debugging complexity

Using a framework when it's not needed increases the overall complexity of most applications. For instance, if our application's database only has one small table, and the query used to fetch data is straightforward, adopting an ORM framework like Hibernate may be overkill.

By adopting a framework, developers not only need to debug issues coming from the code they build but also from the code abstracted by the framework itself.

Not all projects benefit from the use of frameworks. When a project is small or simple enough, introducing a framework will result in negative impacts, like unnecessarily increasing the effort required to build the applic.

Also, frameworks will ship with features that the application does not need and will not use. For instance, a framework that is focused on providing a solution to highly distributed systems will not be a good fit for an application that only uses one or two servers; it ships with many features we do not need and may require extra configuration that could increase the cost of the project.

Zero-cost abstractions

From the point of view of software complexity and performance, frameworks are not free: Even if we do not see their code implementation, code abstractions introduced by frameworks use memory and affect space and time complexity.

The abstraction cost of many frameworks is not large enough to cause a considerable impact on most applications' performance. Still, some specific cases like those of low-level applications need the best performance they can get, and the cost of framework abstractions matters.

Some low-level code languages like C++ or Rust use the "*zero-cost abstractions*" pattern. Code abstractions following this pattern do not perform worse than implementing the same code without the abstractions. In simpler terms, these are abstractions that have no impact on performance.

Considering the impact introduced by frameworks, whether it has a significant impact or not, is one of those skills that developers acquire with experience.

Conclusion

Frameworks and patterns provide correct and reusable solutions to common problems and challenges. Frameworks abstract the essential elements of the problem and provide reusable components that developers can extend to build fully personalized applications.

Frameworks help developers concentrate on business requirements. By using a mix of configuration and conventions, frameworks reduce the amount of code needed to build non-functional requirements or implement helpful design patterns.

Frameworks also promote collaborative work by establishing well-delimited conventions and modular code. For instance, large teams can focus on the layer or component they need to work on without modifying code in other layers.

Frameworks come at a cost, though. They require developers to learn the underlying patterns used by the framework and the API used to configure it. Frameworks increase the knowledge developers need before they can start working on a project, increasing hiring and training costs.

Developers who are just starting their careers should focus on learning the underlying technology before learning the framework's API. If we focus on the basics first, we will exponentially accelerate the learning of any framework, not just those popular today, like React of Spring.

Having seen that there are tools that help us build applications faster, in the next chapter, we will focus on tools and patterns to deploy applications and get them ready to be used.

References

- Gang of four design patterns: **https://www.gofpatterns.com/**
- Gradle installation guide: **https://gradle.org/install/**
- Central Maven repository: **https://repo1.maven.org/maven2/**
- "Maven in 5 minutes": **https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html**
- "Inversion of Control Containers and the Dependency Injection pattern" by Martin Fowler: **https://www.martinfowler.com/articles/injection.html**
- Spring's dependency injection and other core technologies: **https://docs.spring.io/spring-framework/docs/current/reference/html/core.html**
- Spring Webflux: **https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html**
- Java's JPA: **https://jakarta.ee/specifications/persistence/3.0/**
- Hibernate's ORM documentation: **https://hibernate.org/orm/**

- Zero cost abstractions in Rust’s context: **https://rust-lang.github.io/async-book/01_getting_started/02_why_async.html**

CHAPTER 9
Deploying Applications

Once we have built our application, how do we publish it to make it available to our users? How can we guarantee our application will run smoothly on the server as it does on our development computer?

This chapter reviews the process an application has to go through from the source code to the excellent app our users will love. We will also examine how **Virtual Machines** (**VMs**) and containers help make our application environments reproducible, which will allow us to configure our app environment once and run it almost everywhere.

Structure

In this chapter, we will learn the following topics:

- Defining a robust deployment process: CI/CD
- Creating reproducible environments
- Version control
- Virtual Machines
- Containers (A.k.a Docker)
- Working with stateless containers
- Use case: Creating a reproducible deployment environment for the Pizza Place app using Git and Docker

Objectives

In this chapter, we will discuss all things related to the deployment of software applications. By the end of this chapter, we should have a good understanding of the steps of the CI/CD progress, starting from changes made to the codebase repository all the way to having a live instance of the application running in production.

Finally, we will understand how to track changes made to our application's source code, and how to leverage containers to create reproducible and predictable application environments.

The main goal of this chapter is to understand how we can deploy back-end applications to a production environment and what our role as back-end developers is within this process.

Defining a robust deployment process: CI/CD

The software development lifecycle has evolved during the past few years. Years ago, we used to deliver software every few months or even years. Now are expected to deliver code in intervals as short as every few hours. As discussed previously in this book, we would like to deploy our application every time we introduce a new feature or code fix. This demand change has led to the creation of CI/CD pipelines.

Continuous Integration and Continuous Deployment (**CI/CD**) is the process of *building, testing, and deploying an application* as a response to an event like a code commit or a developer manually requesting a re-deployment.

The CI/CD term is composed of two different concepts:

- **Continuous Integration**: The application is repeatedly compiled, tested, and validated end to end. Continuous Integration increases the confidence in the application's quality: The more often we integrate, the faster we can find and fix possible defects and failures.
- **Continuous Deployment**: Deploy the application often. Reducing the number of changes between one deployment and the next makes it easier to identify problems and the code changes that caused them. Also, new features get to users faster, providing more accurate and current feedback.

More than a recipe, CI/CD is a goal of an ideal delivery process that maximizes quality and minimizes defects.

Before CI/CD

Historically, the deployment of software applications has been a largely manual process, from people distributing their applications on floppy disks or CDs to developers publishing software to public repositories where people would download and install them.

As web applications gained popularity, we stopped distributing binary files. Now, developers had to upload their files to a web server using protocols like **File Transfer Protocol** (**FTP**). If they needed to release a new version, developers had to manually replace all files to be updated. *Figure 9.1* illustrates this manual process:

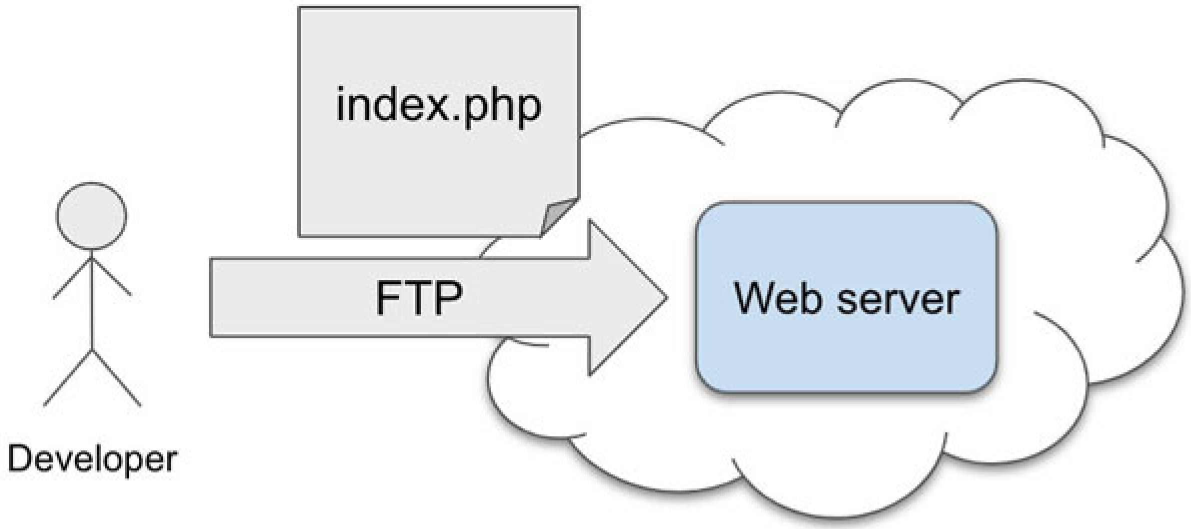


Figure 9.1: *Developers use to update their applications by manually updating files through FTP*

Deployment was slow and error-prone: Developers could miss replacing some files, or they could accidentally and unintendedly modify the server's files, causing errors that required developers to re-deploy the whole application, sometimes multiple times in a single deployment.

With the rise of web-based enterprise applications, web servers became application servers: Large servers that not only hosted static files like HTML, CSS, JavaScript, or even Perl or PHP scripts. Now, servers could run code built using programming languages like Java or C+.

In the Java world, the standard deployment units were WAR or EAR files. These "tick" packages contain the application's compiled source code and files like images, audio, video files, or other resources needed at runtime. These packages forced developers to manually compile and package their software projects directly in their development machines, take the compiled package file, and manually deploy it to the application server. Again, we illustrate this process in Figure 9.2:

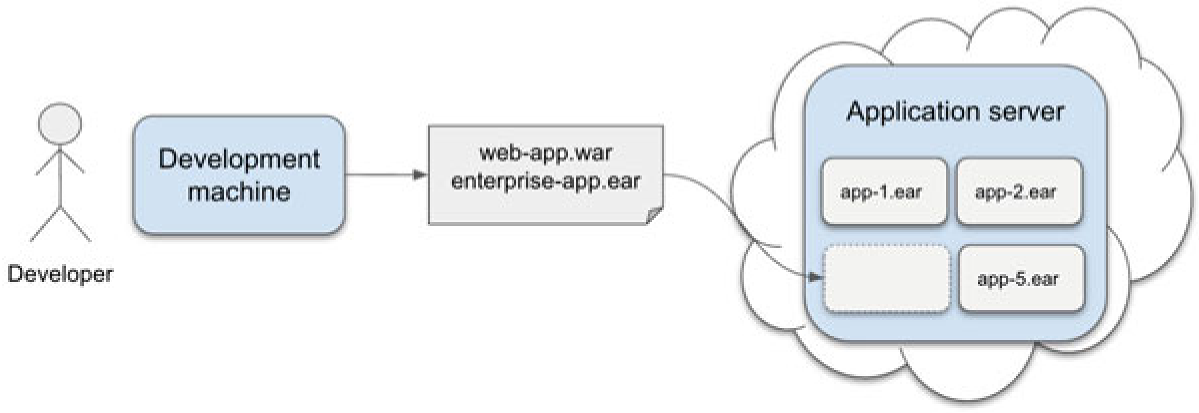


Figure 9.2: *WAR or EAR files are deployed to the application server*

Since *multiple projects were deployed to the same server*, application servers became large and challenging to maintain, full of complex configuration files that we had to update through a user interface manually. The complexity of the application servers resulted in long deployment times and overall instability.

A direct result of this cumbersome deployment process was that developers were afraid of deploying and would delay releasing as much as possible.

The problem with infrequent deployments is that the more changes we include in a deployment, the higher the probability of the deployment failing or introducing errors. This challenge is addressed by the '*CI*' part of '*CI/CD*'. Continuously integrating small changes makes deployments more manageable, as there is less room for errors.

A step forward: Deployment scripts

Some developers tried to improve the deployment process through *automation scripts*. These scripts would build the project, execute the automated test suite, package the application, and copy the package into the application's server's internal file system. These scripts were an improvement, but the process was still difficult to debug when something went wrong.

This setup of "*one developer triggering the deployment script manually*" caused many bottlenecks in the deployment process. Every day, each developer in a team would add changes to the codebase. These changes would add up, and when it was time for deploying, all code changes would have to pass tests and validations. It only took one of these changes not to pass a test to stop the deployment for the whole team.

The developers in charge of deploying became a bottleneck themselves. Every bad code change that broke a deployment had to be managed by the developer running the deployment script, even if that person had no context about the change. In general, deployment scripts lacked visibility for other team members.

To increase the visibility and performance of the deployment process, many teams introduced special servers that would be in charge of executing the deployment process. However, even having a central process still required some amount of human intervention to initiate new deployments. Teams would schedule frequent deployments (for example, daily, at the end of the day), but ad-hoc manual deployments would still be required if critical changes needed to be deployed.

Now, take all these headaches and multiply them as teams introduce new environments for development and test, each requiring independent deployments. All these challenges led to the evolution of CI/CD pipelines.

The advantages of a CI/CD pipeline

CI/CD is *reactive*, which makes it different from previous automated deployment approaches.

Instead of triggering a deployment at an arbitrary time, CI/CD pipelines get triggered when code changes are introduced to the codebase. The pipeline reacts to code changes by automatically compiling, testing, packaging, and deploying the project, all *without direct human intervention*. Furthermore, if any step fails, the pipeline stops and notifies the developer who owns the change that caused the problem.

The overall CI/CD process is described in *Figure 9.3*:

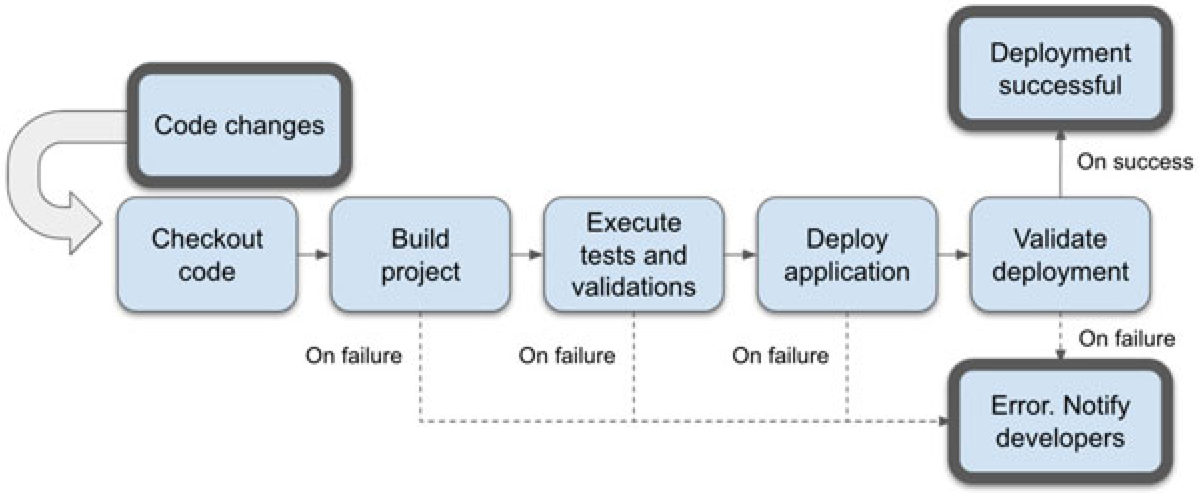


Figure 9.3: *High-level view of the CI/CD process, starting*

`from a change to the codebase and ending in the application's deployment`

As the team now shares the process, no single developer needs to diagnose all broken deployments. Each team member can own their changes and take the lead to fix the deployment if something goes wrong.

The deployment process with CI/CD becomes *asynchronous*. Since each code change triggers a separate deployment, then no single bad change can break the deployment for other developers. Only the developer who introduced the code change will be blocked until someone fixes the issue (or the developer discards the code change).

The main advantage of the CI/CD process is that it *increases the confidence in the codebase*. We know that each change in the codebase is correct because it passed each test and validation before being merged and deployed.

Creating reproducible environments

Whether the deployment is triggered manually or as a response to code changes, one significant bottleneck in the deployment process is the *lack of isolation in the validation and deployment process*. To illustrate this issue, let us take a look at an example.

Imagine we begin working on a new project that uses a library built with Python 3. The version currently installed in the CI/CD server is Python 2, so we upgrade the production server to Python v3. Then, we discovered that another team was still maintaining a Python 2 application, and our Python version update broke their deployment process.

In this example, there are workarounds: For instance, we can create virtual environments or use a tool like Anaconda. However, we would still need to be careful not to break any other deployed applications in the server when changing our team's application environment. The lack of isolation increases the complexity of this task.

Moving out of shared environments

In traditional applications servers, applications rarely ran in isolation: Physical servers had multiple servers installed, each with its own set of applications. For instance, a server could host a WebSphere Application Server for their Java enterprise applications and an Apache server for their static websites. Application servers contained multiple applications, including internal and external facing web applications.

In many cases, *one application would fail due to other applications or servers' actions*, like any changes to shared resources like configuration files. Debugging these problems is difficult, as it would require developers to debug them directly in the production server or try to recreate the whole server in a development machine.

Every new server is configured with clean settings. Servers are configured with optimal settings, and the server operates efficiently. Then, as time goes by, *the state within a server changes as its applications receive and generate data,* possibly due *to* user interactions. After a while, a server shared by many applications will get in such a messy state that it starts to behave erratically. All this randomness led to multiple issues:

- Applications running out of memory unexpectedly.
- Race conditions between two separate applications lead to errors impossible to replicate.
- Code that was working in a developer's machine does not work on the server.

The solution to these issues is *isolated and reproducible environments*. Each of these environments is specifically created for the application they need to host, and they enforce a hard separation from other environments.

Advantages of isolated and reproducible environments

Isolated and deterministic environments have many benefits for software developers and their projects:

- We can upgrade an application’s dependencies (libraries, dependencies, compilers, language specifications) without having to modify other applications’ environments. In the Python application example, each application can use the Python version they need without conflicting with each other.
- Build, test, and deployment becomes deterministic processes. If an application’s state gets messy, it only affects one application. Plus, we can create a new and clean environment instance without affecting other applications.

Creating reproducible environments can be achieved by using a set of tools that enforce these constraints.

There exist multiple tools that allow us to build creating isolated and reproducible environments. Reproducibility can be achieved through *version control*, and isolation through the use of Virtual Machines or containers.

Version control

We cannot create reproducible environments if our source code itself is not reproducible.

From the first day of any project, the source code will change constantly. We build new features, fix bugs, introduce new bugs, and create more bug fixes. With so many changes, we must be confident that the code is correct. Code changes generate value for our application but can also subtract value.

Any software developer has experienced the pain of having accidentally deleted their work at least once in their career. Days, months, or even years of work can be lost, leading to millions in losses; in some cases without a clear way of recovery

In the past, software developers had to create backups of their working source code somewhere safe manually. If something wrong happened, only a tiny amount of work would be lost, as we would have a snapshot of the *last known working version*.

However, any manual process is prone to errors:

- Not backing up as often as needed. The backup process is long and complex for some large applications, so we did not back up as often as we should. The longer we spend without creating a backup of our application's codebase, the more work we will lose in case of an adverse event.
- Missed backups: A developer may forget to back up the codebase when introducing changes to the application. This situation happens a lot for minor changes that had to be done in a hurry to fix critical problems in production, which developers would often forget to back up.
- Incomplete backups: Developers may miss one or two files while creating the backups, unknowingly leading to loss of work.
- Incorrect backups: Developers may accidentally back up files that were not correctly tested or approved in peer reviews.

Any of these previously listed problems can prevent us from deploying a working version of the application to the production environment. **Version control systems** (**VCS**) fix or mitigate most of these issues.

VCS incrementally stores changes done to a file repository. They keep a **file backup**, but they also *hold a* ***historical log*** *of each change to the codebase*. Using these changelogs, we can know precisely how the code has changed in time, who changed it, and when it was changed. This level of meta-information is critical while debugging errors or reverting faulty code.

VCS provides data integrity measures for an application's source code:

- **Observability**: Each change done to the source code needs to be visible and explainable.
- **Recoverability**: Once we have a working version of the application, we should be able to recover it if future changes to the codebase cause errors or break the application.

- **No repudiation:** Each change to the source code is stored along with the person who introduced it.

On top of it all, VCS provides an opportunity for peer reviews. Each change done to the codebase can be inspected and reviewed by other team members. These reviews significantly reduce the probability of defects or poor quality code being added to the application.

The most prevalent version control systems are *Git* and *SVN*.

Git

Git is a *distributed* version control system where files are stored inside a *repository*. Repositories can be replicated across multiple remote servers and clients. At a very high level, a standard Git repository looks like as shown in *Figure 9.4*:

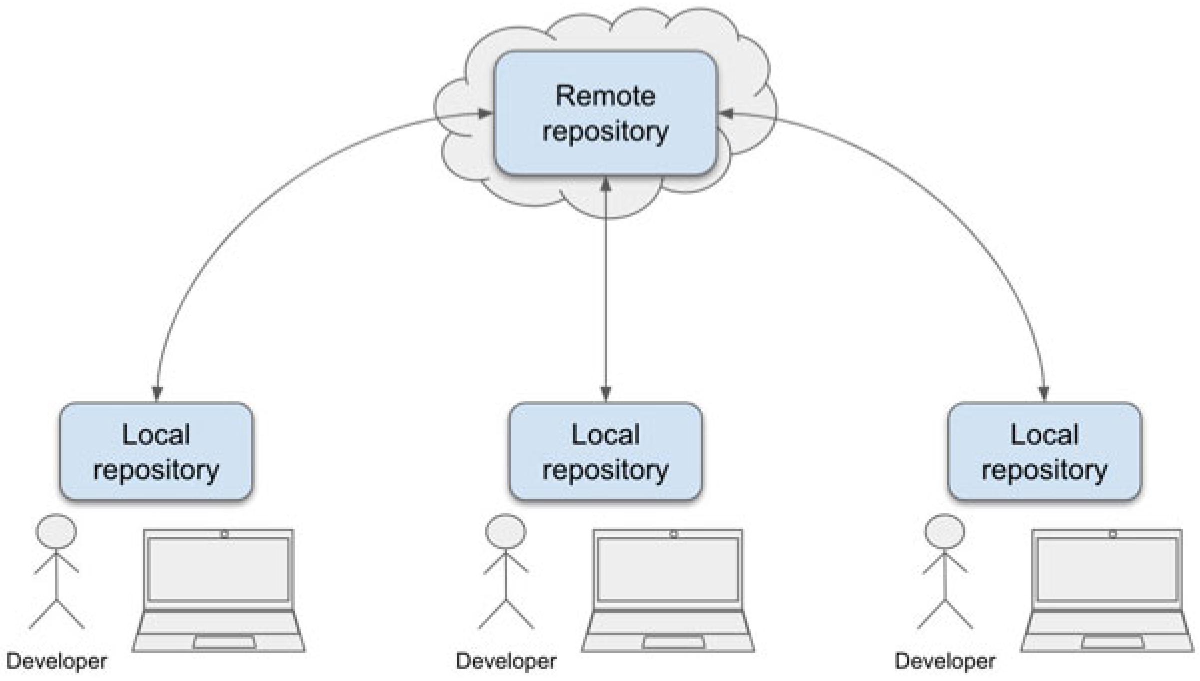


Figure 9.4: *High-level view of a Git project*

Local repositories are copies of a remote repository. The remote repository is the *source of truth* for the codebase, and all changes to the local repositories are synced to the remote repository through Git's API.

Creating a Git repository

Having installed Git on our computer (more details in the "*Resources*" section of this chapter), creating a Git repository is as easy as executing the following command:

```
1. git init
```

This command should return a message similar to the following:

```
Initialized empty Git repository in …
```

We can see the state of the Git repository with the following command:

```
1. git status
```

Assuming the repository is empty, we should see the following message in the console:

```
On branch master
No commits yet
nothing to commit (create/copy files and use "git add" to
track)
```

This message indicates that the repository is empty but ready to work. Git organizes code in *branches* (we will discuss branches in the following sections). By default, all the code in this repository will be stored in a branch called "*master*".

Note Git uses the branch name "master" to identify the main branch (again, we will discuss the concept of branches later in this chapter).

The default branch name "master" has been slowly replaced with "main" to use a more inclusive term. Large Git repository providers like GitHub have changed their naming conventions to follow this pattern.

Staging and committing new files

Each set of changes done to the files in a Git repository is called a **commit**. Commits are applied on top of each other, following their creation order. The list of commits in a repository is the **changelog**.

To continue with our example, let us update our repository by adding a Python script. We will create a file inside our freshly created Git repository:

```
./Application.py
```

Then, we add some Python code to **`Application.py`**:

```
class Application:
    def __init__(self):
        print("App started")

if __name__ == '__main__':
    Application()
```

If we execute **`git status`**, we should see the following text:

```
On branch master
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    Application.py
nothing added to commit but untracked files present (use "git
add" to track)
```

We can see that Git considers our new file as "*untracked*", meaning that the new file is not yet part of the repository. This situation may sound confusing (considering the file is already inside the repository directory), but this distinction has a good reason.

Git will not consider any new files as part of the repository until we explicitly indicate it by staging and committing this file. This step is required because this file may be part of work in progress and not ready to be included in the repository.

The first step to adding any change to the repository is to **stage** it. Staging allows us to selectively choose which files to include in a commit and which not to. To stage this new file, we use the following command:

```
git add Application.py
```

After executing **`git status`**, we will now see the following message:

```
On branch master
No commits yet
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   Application.py
```

By staging the new file to the repository, Git now recognizes this file as part of the project *but it is not yet committed*. Staged and committed files are distinct yet closely related:

- Only staged files can be committed.
- Only committed files can be merged into the remote repository.
- Only committed files are added to the changelog.

We can confirm the changelog is empty by executing the following command:

```
git log
```

Since we have not committed any changes, Git will reply with the following message (confirming that staging is not enough to persist this change into the repository):

```
fatal: your current branch 'master' does not have any commits
yet
```

Having staged the new file, the next step is to *commit it*. We use the following command to create a commit that will include all staged files:

```
git commit -m "Add the file Application.py"
```

The "**-m**" attribute displays the message we will use for the commit. This message is critical since it allows authors to describe why they are making these changes, along with any other information that may be helpful for other collaborators.

After executing the **commit** command, we will see a reply from Git similar to the following text:

```
1 file changed, 6 insertions(+)
create mode 100644 Application.py
```

Now, when we execute **git status**, we will see a slightly more straightforward message:

```
On branch master
nothing to commit, working tree clean
```

The **Application.py** file is now part of the local repository. However, the file is *not part of any remote repository yet*. We can confirm so by executing the following command:

```
git log
```

This command will return the changelog for the local repository. In this case, we will see something similar to the following message:

```
commit 6879f6a9aa21ba4967f36348e8f7644f0dd5b014 (HEAD ->
master)
Author: Pedro Marquez <pmarquez@Pedros-Air.attlocal.net>
Date:   Sat Nov 27 18:45:39 2021 -0600
    Add the file Application.py
```

Before we describe the contents of the commit record in detail, let us modify the file and commit this modification. This extra step will provide a more robust example.

Note Adding files to a repository.

While Git is used extensively to store and backup software's source code, it can store any file type. Most Git features are applied to text-based files, but the repository can also store binary files.

Besides being used for software projects, Git is used to backup and keep track of changes made to books, blogs, and other projects that can take advantage of advanced file management.

Making changes to existing files

Staging and committing new files is just one case of the overall Git flow. More often than not, most of the Git changes we will do to a repository are to update existing files.

Let us update **`Application.py`** to change the text the script prints and add a new Python comment above the exiting **`print`** line:

```
class Application:
    def __init__(self):
        # The following line of text was changed
        print("The application is now started")

if __name__ == '__main__':
    Application()
```

If we execute **`git status`**, we will see the following message:

```
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working
directory)
    modified:   Application.py
no changes added to commit (use "git add" and/or "git commit -
a")
```

We can see the list of files that have been modified but not staged or committed yet.

Before we stage the changes done to the file, we need to know precisely what changed. We can execute the following command to highlight all the changes done to existing files in the repository:

1. `git diff`

The following command will print the list of changes done to each file, which in this case is only **`Application.py`**:

```
diff --git a/Application.py b/Application.py
index 38606cb..46e1977 100644
--- a/Application.py
+++ b/Application.py
@@ -1,6 +1,7 @@
class Application:
     def __init__(self):
-        print("App started")
+               # The following line of text was changed
+        print("The application is now started")
if __name__ == '__main__':
     Application()
\ No newline at end of file
```

Git uses its internal **`diff`** tool to check the difference between the existing file in the repository and the unstaged modifications done to the file we have in our project's folder.

In the preceding example, we can see that the lines marked with a subtraction sign ("-") are lines that belong to the old version, while those lines marked with a plus sign ("+") are the changes we have done in the current version of the file. In this example, we "*removed*" the code at line 8 while adding lines 9 and 10. The rest of the file remains the same.

Now that we have verified that the file has changed as we expected it to, we can stage and commit it:

1. `git add Application.py`
2. `git commit -m "Update the print message and add comment"`

Now, we execute **`git log`** to confirm the new commit we created was successful:

```
commit cb8bef5a4755f4b90eb78a1fc6ff2798ac14b837 (HEAD ->
master)
Author: Pedro Marquez <pmarquez@Pedros-Air.attlocal.net>
Date:   Sat Nov 27 19:03:34 2021 -0600
    Update the print message and dd comment
commit 6879f6a9aa21ba4967f36348e8f7644f0dd5b014
Author: Pedro Marquez <pmarquez@Pedros-Air.attlocal.net>
Date:   Sat Nov 27 18:45:39 2021 -0600
    Add the file Application.py
```

Commit details

From the changes we just did to the Git repository, we can see that each commit has a set of attributes:

- **`commit`**: A cryptographic hash that is generated to identify the changes in this commit. This attribute is an identifier for the commit.
- **`Author`**: The person who created this commit.
- **`Date`**: The date and time this commit was created.
- The commit **message**: The message we passed in the **`-m`** attribute of the **`git commit`** command.

The **`git log`** command has many options that can give us a better view of each commit's details. For instance, the **`git log -p`** command displays the list of commits along with all the code changes that are included in each commit:

```
commit cb8bef5a4755f4b90eb78a1fc6ff2798ac14b837 (HEAD ->
master)
Author: Pedro Marquez <pmarquez@Pedros-Air.attlocal.net>
Date:   Sat Nov 27 19:03:34 2021 -0600
   Update the print message and dd comment
diff --git a/Application.py b/Application.py
index 38606cb..46e1977 100644
--- a/Application.py
+++ b/Application.py
@@ -1,6 +1,7 @@
class Application:
    def __init__(self):
-        print("App started")
+              # The following line of text was changed
+        print("The application is now started")
if __name__ == '__main__':
    Application()
\ No newline at end of file
commit 6879f6a9aa21ba4967f36348e8f7644f0dd5b014
Author: Pedro Marquez <pmarquez@Pedros-Air.attlocal.net>
Date:   Sat Nov 27 18:45:39 2021 -0600
   Add the file Application.py
diff --git a/Application.py b/Application.py
new file mode 100644
index 0000000..38606cb
--- /dev/null
+++ b/Application.py
@@ -0,0 +1,6 @@
+class Application:
+    def __init__(self):
+        print("App started")
+
+if __name__ == '__main__':
+    Application()
\ No newline at end of file
```

Notice that the results of this command can be pretty long, as we are printing every change done to each commit.

Branches

A trendy concept in movies and TV shows is that of multiverses. In these movies, some events create alternate timelines or realities where the protagonists must complete a task before returning to the main timeline. Git branches follow that same pattern (as seen in *Figure 9.5*).

Following the multiverse analogy, the *master* or *main* branch is the main timeline. The one reality from which all other universes are derived. All other branches are alternative realities: A parallel version of the main branch where changes are introduced:

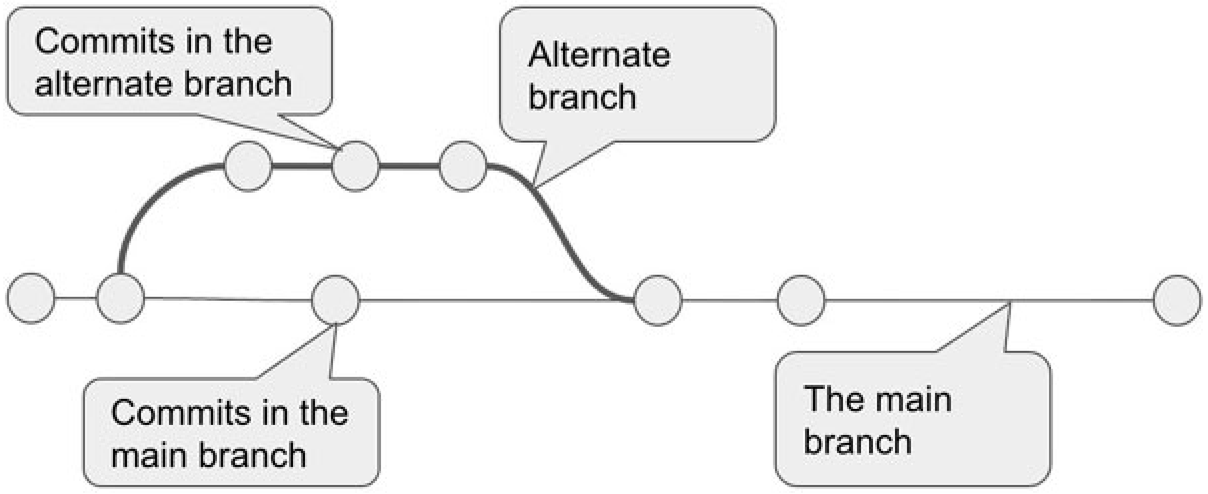


Figure 9.5: *Branches create alternate timelines to introduce changes to the repository*

When developers are working on a new feature, they may need to break all code changes into multiple commits. Multiple, small commits make code reviews, debugging, and overall source code management easier.

Instead of adding work-in-process modifications to the main branch, developers can create a new **branch** to *include all the commits for the new feature*. Once the feature is complete, the alternate branch can be merged back into the main branch.

Branches give us much flexibility to break down our work into multiple pieces.

In our example, we can create a new branch called **`add-print-messages`** which is based on the current ("*master*" in this case) branch:

```
1. git branch "add-print-messages"
```

The following command will create a new branch named **`add-print-messages`**. We can create as many branches as we need:

```
git branch super-cool-feature
```

We can use the following command to see all the branches in the local repository:

```
git branch
```

Along with the list of branches, the star character tells us the currently active branch. Only one branch can be active at a time, and in this example, it is **`master`**:

```
  add-print-messages
* master
  super-cool-feature
```

Since only one branch can be active at the time, we can *switch branches* with the **`git checkout`** command:

```
git checkout add-print-messages
```

To simulate adding a new feature, we can add another **`print`** statement to **`Application.py`**.

```
class Application:
    def __init__(self):
        # The following line of text was changed
        print("The application is now started")

if __name__ == '__main__':
    print("This is a new feature")
    Application()
```

We can stage and commit the changes to our new *feature*:

```
git add Application.py
git commit -m "Add a new feature"
```

Looking at the last two commits in **`git log`**, we can see which commit is the last we did in each branch:

```
commit 9f63c03a91daf1184af281fa00df60ac79078aef (HEAD -> add-print-messages)
Author: Pedro Marquez <pmarquez@Pedros-Air.attlocal.net>
Date:   Sat Nov 27 19:38:24 2021 -0600
    Add a new feature
commit cb8bef5a4755f4b90eb78a1fc6ff2798ac14b837 (super-cool-feature, master)
Author: Pedro Marquez <pmarquez@Pedros-Air.attlocal.net>
Date:   Sat Nov 27 19:03:34 2021 -0600
    Update the print message and dd comment
```

The log shows that the **add-print-messages** branch has one more commit than **super-cool-feature** and **master**. This commit contains our new "feature".

Merge

Let us assume the feature is complete after the commit we just created. We now want to return to the good old main timeline: The **master** branch. We can do this by merging the feature branch into **master**.

We first switch to the **master** branch, and we then execute **git merge**:

1. `git checkout master`
2.
3. `git merge add-print-messages`

All commits in the **add-print-messages** branch will be copied to the **master** branch. We can confirm this by executing **git log** again:

```
commit 9f63c03a91daf1184af281fa00df60ac79078aef (HEAD ->
master, add-print-messages)
Author: Pedro Marquez <pmarquez@Pedros-Air.attlocal.net>
Date:   Sat Nov 27 19:38:24 2021 -0600
    Add a new feature
commit cb8bef5a4755f4b90eb78a1fc6ff2798ac14b837 (super-cool-
feature)
Author: Pedro Marquez <pmarquez@Pedros-Air.attlocal.net>
Date:   Sat Nov 27 19:03:34 2021 -0600
    Update the print message and dd comment
```

Now, the top of both **`master`** and **`add-print-messages`** point to the same commit ID, which means that both branches are now identical. The timeline was merged.

Remote repositories

Until now, all our commits are only included in our local repository. Git promotes collaborative work with the use of *remote repositories*.

Remote repositories need to be kept in sync with local repositories. For that, developers put all commits in their local repository first. Once we are ready to share our work with everyone, we can ***push*** all local changes to a remote repository.

A popular place to create remote repositories for free is GitHub. We can create an account and an empty repository (again, documentation on how to do this is included in the *Resources* section).

Once we have created an empty repository in a service like GitHub, integrating it with our local repository is as easy as adding a new origin:

1. `git remote add origin https://github.com/pfernandom/my-remote-repo.git`

An origin is a remote instance of our repository. We can add multiple origins to the same local repository.

When we created the local repository, the name **`master`** was automatically chosen for our main branch. GitHub specifically uses **`main`** for the primary branch name. To make them match, we can rename our **`master`** branch using the **`git branch -m`** command:

1. `git checkout master`
2. `git branch -m main`

Now that **`master`** is renamed as **`main`**; we can sync our local repository with the GitHub remote repository using the following command:

1. `git push -u origin main`

A response similar to the following will be displayed in the terminal:

```
Enumerating objects: 9, done.
Counting objects: 100% (9/9), done.
```

```
Delta compression using up to 8 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (9/9), 979 bytes | 979.00 KiB/s, done.
Total 9 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), done.
To https://github.com/pfernandom/my-remote-repo.git
* [new branch]      main -> main
Branch 'main' set up to track remote branch 'main' from
'origin'.
```

The remote and local repositories are now in sync.

Cloning repositories

Let us say we wanted to share this new project with other team members. As long as they have access to a public Git repository like **https://github.com/some-username/my-remote-repo.git**, they can clone the repository into their computers with the following command:

```
git clone https://github.com/some-username/my-remote-repo.git
```

A fresh copy of the repository is created after this command completes successfully.

Merging conflicts

Like any distributed system, a common cause of developers' headaches in Git is **merge conflicts**. These happen when two or more developers make conflicting changes to the same file in parallel, and then they all try to merge their modifications into the remote repository. *Figure 9.7* shows the simplest case of a merge conflict:

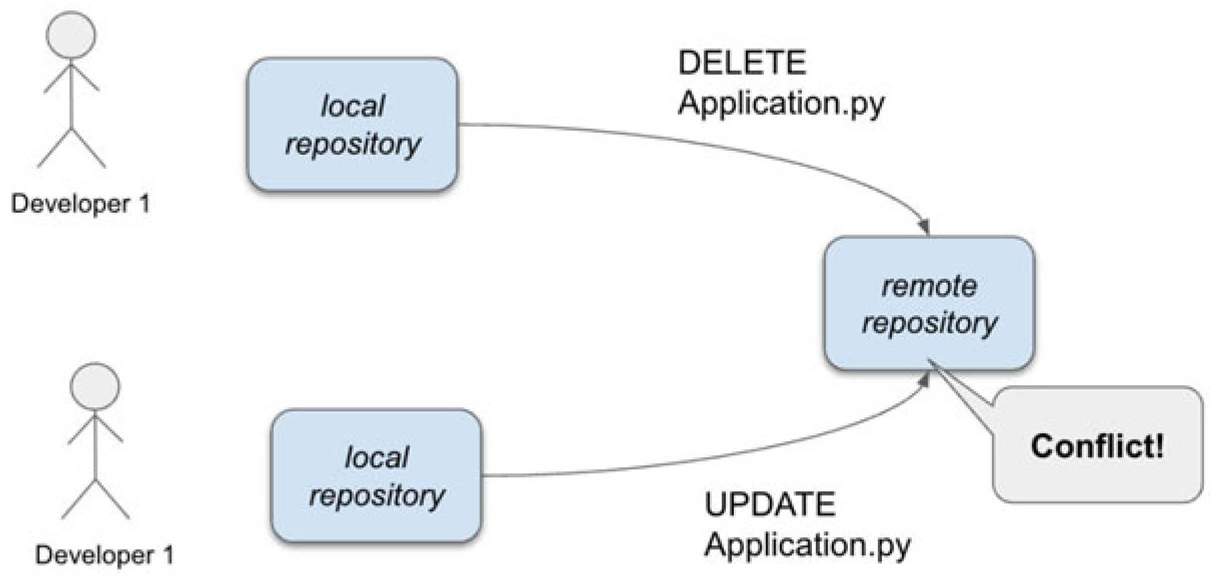


Figure 9.6: *Git merge conflict*

The flow displayed in *Figure 9.6* is described more in-depth in the following steps:

- Developer 1 deletes the **`Application.py`** file from their local repository.
- Developer 1 commits the changes and pushes the commit to the remote repository.
- Developer 2, who has not updated their local repository (and thus does not know Developer 1 deleted the file) updates **`Application.py`** in their local repository.
- Developer 2 commits the changes and tries to push the commit to the remote repository.
- The push operation fails, and Git asks Developer 2 to update their local repository before pushing.
- Developer 2 tries to update their local repository. The update fails, and Git requests Developer 2 to fix the merge conflicts between the code version in the remote repository (the changes done by Developer 1) and the code version in Developer 1's local repository.

Since Developer 1 merged their commit first, they will see no conflicts while pushing changes to the remote server. Every other developer who commits conflicting updates to *Application.py* after Developer 1 pushes their changes will get merge conflicts.

Developers must manually solve merge conflicts. The process usually requires the developer to understand the difference between the version of the file in the remote server and the changes in the conflicting commit and then merge both versions.

We can avoid merge conflicts by updating our local repository often, especially before creating a commit. However, merge conflicts are part of the Git flow, and they help keep the remote repository clean.

Using Git hooks

The concept of hooks in Git is a powerful one. Hooks are custom scripts that run when important actions occur in a Git repository.

We can configure a Git repository to execute a script on the local repository when Git-specific actions are performed. For instance, we can run automated scripts for actions like **`pre-commit`**, which can be used to inspect the staged files that are about to be committed, or **`post-merge`**, which runs after a successful **`merge`** command.

We can hook to actions in the remote repository too. Hooks like **`update`** or **`post-receive`** are executed when we send the code from a local repository to a remote one. *Using Git hooks, we can trigger a new deployment when the code is merged to the remote server*, making the deployment flow reactive. A visual representation of how hooks interact with the different tasks in a Git repository can be seen in *Figure 9.7*:

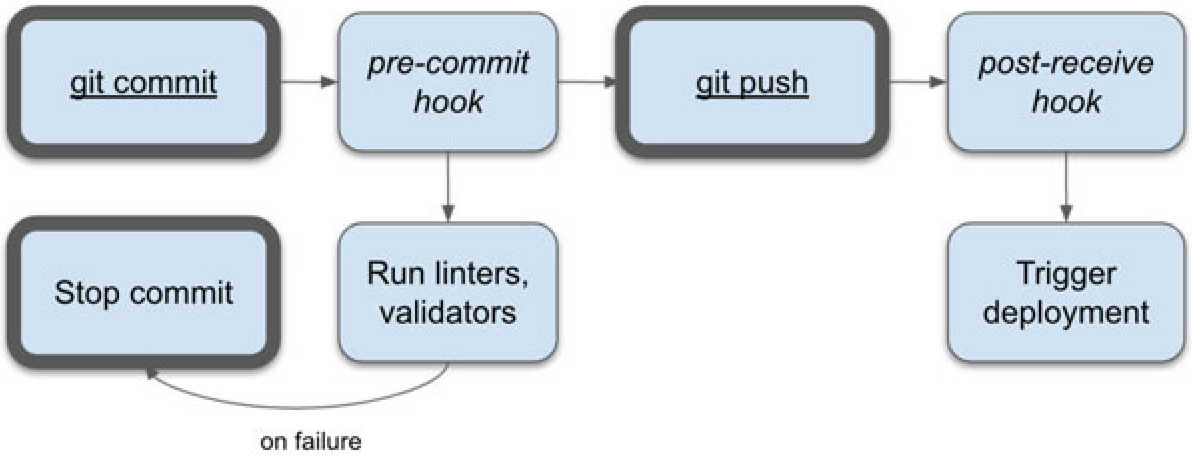


Figure 9.7: *Git hooks run when actions like "commit" or "push" are executed*

We can also use hooks for code quality purposes. We can run linters and validators before deployment to stop changes that introduce code quality issues. Only the code that passes all tests can be merged into the remote repository and deployed to production. These extra validation steps provide defense layers against new defects.

Git to enforce reproducible code

Git repositories allow our source code to become reproducible. If we execute hooks to enforce that *any commit is tested and valid*, we know that every time we clone a repository, the code is correct and "healthy".

If we *clone the repository for each deployment*, we know the code about to be deployed is validated and tested. Each change to be deployed is accounted for. If the deployment fails for causes independent of the application (like the server's hard drive getting full), we can quickly and deterministically retry the deployment later.

Suppose that, for some reason, the pre-commit validations fail to catch a bug before it is merged and deployed. In that case, it is easy to find which commit includes the problem by calculating the difference in commits between the last good deployment and the deployment that caused the problem.

SVN and other CVS

While Git is widely used to keep track of software versioning, other CVS like **Subversion** (**SVN**) are actively used to track open-source projects (for instance, FreeBSD is tracked using SVN). While they have operational and design differences with Git, the same basic concepts are shared by any CVS: Keeping track of the incremental changes done to a project.

As stated before in this book, there is no single tool to do all jobs efficiently. Some projects have adopted SVN instead of Git because they find SVN more adequate for their project. Some people think SVN is simpler than Git, which helps them concentrate on their work.

We will not visit the specifics of SVN or other CVS because there is too much information to cover that is out of the scope of this book. This chapter aims to understand why we use CVS like Git and how they collaborate with the CI/CD pipeline.

Virtual machines

Earlier, we discussed how servers used to get themselves in really complex states because the applications they host would leak the state into the server and into each other, adding, updating, or removing files in time. Clearing these complex states required stopping all the hosted applications for manual maintenance.

It is clear that sharing the server space between multiple applications was a messy challenge, but how could we impose boundaries effectively?

Virtualization was born in the early 70s to allow multiple users to share computational resources. Instead of giving access to users to one shared environment in a server, administrators can create multiple virtual environments, one for each user or *tenant*.

Virtualization helps organizations reduce expenses and increase productivity. Servers can run multiple virtual environments and multiple isolated applications on the same server. This separation was practically impossible to achieve without virtualization.

A **Virtual Machine (VM)** *is a "virtual representation of a physical computer.*" Each virtual machine can have its own OS and file system, and multiple VMs can be hosted in a single physical server, as shown in *Figure 9.8*:

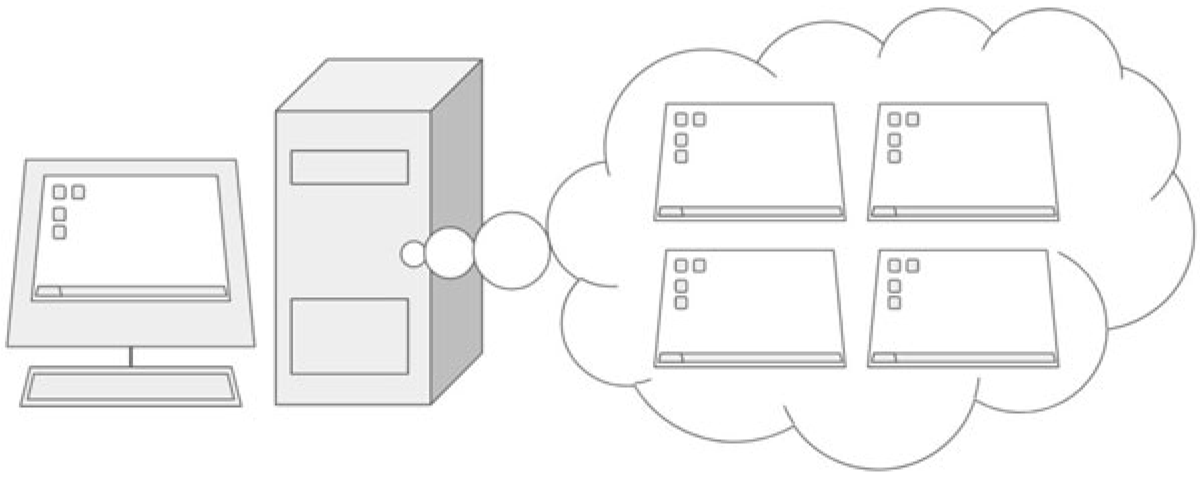


Figure 9.8: *A VM is a computer running inside another computer*

VMs were a game-changer for system administrators. If a VM’s state would get so messy that it needed to be cleaned up, administrators could create a new VM using the snapshot of the last good state of the server and then swap the messy VM with the fresh copy.

VMs reduced the amount of scripted work needed to create a new environment, as whole snapshots could be created with the state of a fully configured server. Scripts add minor customizations on top of the base image.

VMs helped reduce the amount of state leaked from one application into another as independent applications could be deployed to their own VM. If a single VM goes down, the rest of the applications deployed in the other VMs would still operate correctly. Thanks to this level of flexibility, production environments became more robust and resilient.

An excellent benefit for software development teams who adopted virtualization is that they could use identical copies of their production environments to create development or test environments. By reducing the difference between test and production environments, developers can accurately debug new and existing bugs.

Virtual infrastructure

VMs are hosted on top of a special virtualization tool called **hypervisor**. The hypervisor manages server resources like disk space, memory, and input and output devices on behalf of each VM. Each VM can only see the resources the hypervisor assigns to it. The high-level infrastructure can be seen in *Figure 9.9*:

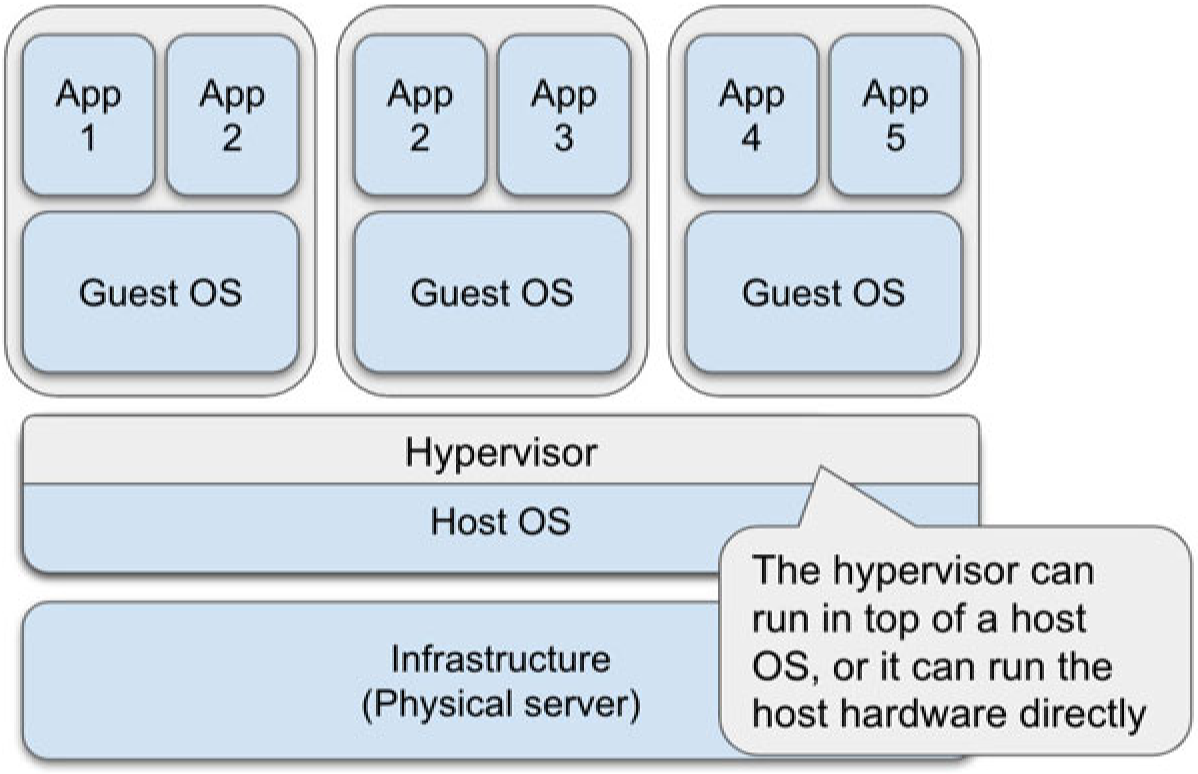

App 1
App 2
Guest OS
App 2
App 3
Guest OS
App 4
App 5
Guest OS
Hypervisor
Host OS
Infrastructure (Physical server)
The hypervisor can run in top of a host OS, or it can run the host hardware directly

Figure 9.9: *The hypervisor manages the access of each VM to the physical server's resources*

Hypervisors can be embedded inside a host OS or run directly on the server's hardware, effectively making the hypervisor an OS by itself. This variety of infrastructure options gives software development teams the flexibility they need to create robust architectures.

VMs are the ultimate step towards creating a reproducible environment, as they share nothing else than low-level computational resources and can be created on-demand. There are practically no downsides to working in a VM instead of directly working on the host server, making them a default choice for most modern infrastructures.

Containers (Docker)

Virtualization through VMs is an essential piece of any infrastructure. The VM acts as a sandbox that isolates its processes from the host OS, the server interfaces, and other VMs. However, each VM contains a full copy of the OS, with all the bundled apps and the memory and storage overhead an OS involves. VMs are definitely not lightweight.

Containers were born out of the need for lightweight, reproducible environments. The goal for containers is to provide the same isolation benefits VMs offer without the overhead, as less overhead means we can host more instances of a container than we would of a VM.

Containers took a different approach to virtualization. Instead of virtualizing server environments containing multiple applications and processes, *containers aim to isolate individual applications* and their dependencies.

The leading platform currently used for containers is **Docker**. Using the industry-standard "*Containerd"*, Docker defines specifications for creating, deploying, and packaging applications inside containers. Docker is considered a **Platform-as-a-Service** (**PaaS**).

Similar to how VMs run on top of a hypervisor, containers run on top of the Docker platform. However, instead of only sharing low-level computational resources as VMs do, containers share *some* OS services. Figure 9.10 shows the high-level view of Docker:

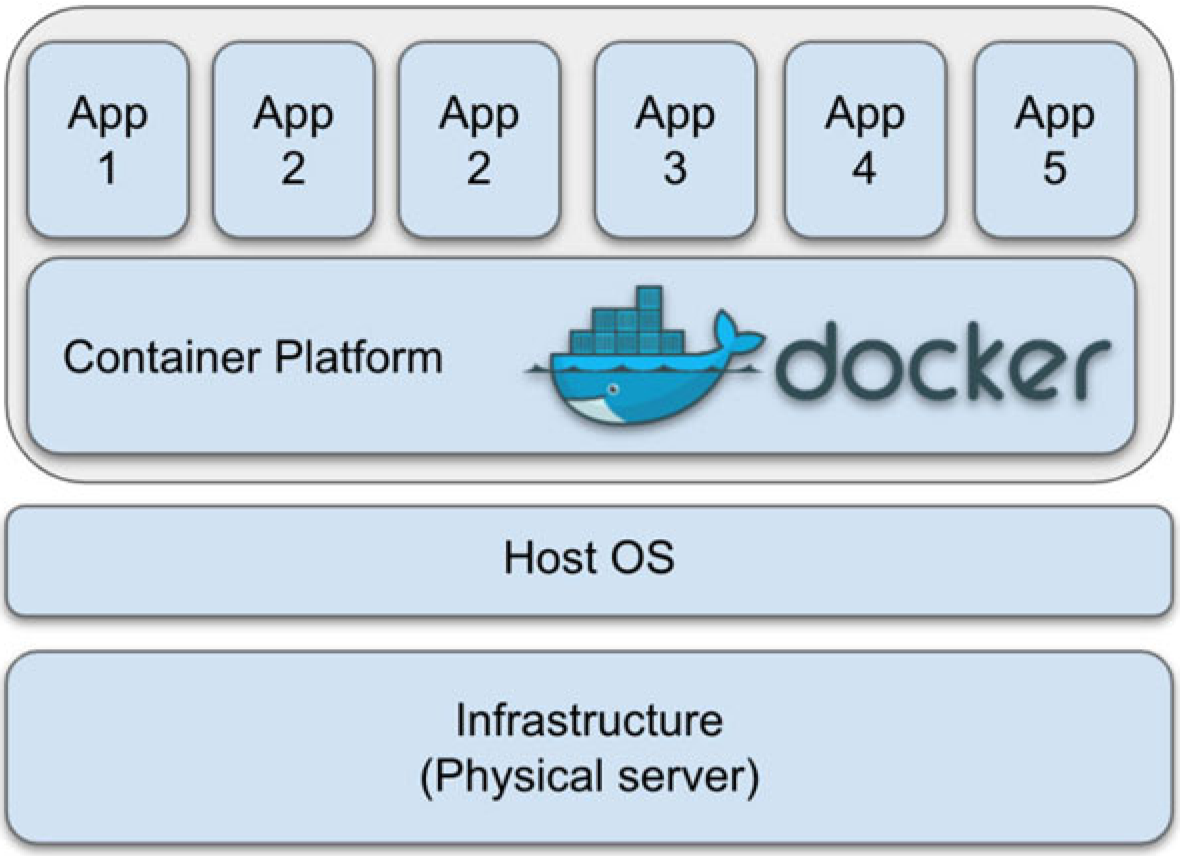


Figure 9.10: *Docker runs on top of the host OS*

The main advantage of containers is that *developers can define one single environment that both runs in their development machine and production server*s. Each container installs the exact version of all the dependencies it needs, no more, no less, removing the differences between development and production environments. Moreover, since containers only install the services and dependencies they need to execute the application, they consume fewer resources than VMs.

While VMs commonly host multiple applications or services, typically we create one container per application, providing a higher level of isolation and independence.

VMs versus containers

Let us be clear. Containers do not replace VMs. Containers address only a subset of use cases traditionally addressed through VMs. There are still many cases where virtualization is preferable to containers. VMs are preferred over containers to host stateful services like database servers. We will discuss more details about containers and storing state in the next section.

Note Containers and databases

Containers can be used to host databases. Volumes are a way to persist container data that can be leveraged to host a database.

However, most orchestration tools (Kubernetes, Openshift, among others) are designed to handle stateless containers. Configuring these tools to run a database in multiple container replicas is a difficult task, one that many database vendors do not support natively.

As a good rule of thumb, avoid using containers to host critical or data-intensive databases. VMs are better suited for hosting these services.

Working with stateless containers

Given the low footprint containers have compared to VMs, we can easily and quickly generate new instances of a container on demand. This makes containers an excellent tool for the following cases:

- Recreate from scratch an application that crashes. Assuming all code changes go through version control into a remote code repository, creating new containers will always contain the most up-to-date version of the application.
- Create replicas of the application on-demand, possibly distributed across multiple servers to increase server capacity. These replicas can make for "elastic" clusters: The number of replicas can scale up and down based on the application's traffic.

Orchestration tools like **Kubernetes** make it easy for teams to create a cluster of containers. Since these tools themselves can be installed as a cluster, we can create a pool of resources that can scale up and down as needed. *Figure 9.11* shows this error handling process as follows:

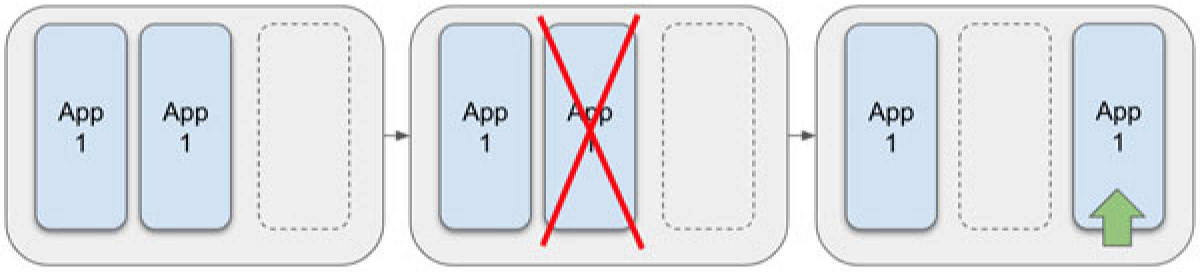


Figure 9.11: *When a container in the cluster crashes, Kubernetes can create a new instance to replace it.*

The orchestration model followed by tools like Kubernetes is designed on the idea of stateless containers. Stateless containers allow us to replace one instance of a container with a new instance without causing disruptions to the users.

Destroying and creating new container instances also means destroying all data inside the existing container. Everything that is not explicitly moved to persistent storage (or a source code repository) will be deleted. This is why, if we want to persist data generated by an application running within a container, *that data must be stored outside the container*.

If we go back to *Chapter 2, The Client-server Architecture*, we remember this is a known pattern: HTTP assumes that the server is stateless. The stateless characteristics of HTTP servers allow us to do load balancing without sticky sessions. In HTTP, all state (like a user's session data) is persisted through cookies, session tokens, and persistent storage in databases.

Use case: Creating a reproducible deployment environment for the Pizza Place app using Git and Docker

To completely understand the benefits of using Docker, let us encapsulate a web application's environment in a container that can be replicated in any server running the Docker platform, including orchestration tools like Kubernetes.

For this, we *containerize* an Express server running an application using the MVC pattern. This Express server will host the Pizza Place's home website. The server uses Node v16 and it will serve as an example for any stateless web application running inside a container.

However, we will not just build one instance of the application. We will build three application instances, which will be load-balanced by an Nginx server (also running inside a container). All containers are managed and connected by **Docker Compose**, a Docker tool that builds the containers and binds them together through an internal network.

Note: Nginx is a web server that can also be used as a load balancer, HTTP cache, and reverse proxy, among other roles. Nginx is a lightweight yet powerful solution to put in front of other HTTP services.

Setting up the application

Our example has the overall project structure:

```
├── docker-compose.yml
├── nginx
│   ├── Dockerfile
│   └── nginx.conf
└── web
    ├── Dockerfile
    ├── package-lock.json
    ├── package.json
    ├── public
    ├── server.js
    └── views
```

At the root directory, we have a **`docker-compose.yml`** file. This file will describe all the Docker wiring needed to create the three instances of the container and the load balancer. We will discuss the details of this file soon. For now, let us focus on the two directories, **`nginx`** and **`web`**.

The web directory

The **`web`** folder contains our Express server (along with its dependency files, templates, and other static resources like images). This folder also includes a **`Dockerfile`**: Docker's most basic configuration file. Dockerfile describes the contents of a Docker **image**. Docker images are used to create containers.

The following is the content of the **`Dockerfile`**:

```
# use the base docker image for Node ١٦
FROM node:16
# create app directory
WORKDIR /usr/src/app
# copy all node related files
COPY package*.json ./
# install application dependencies
RUN npm install
# copy application into container
COPY views/* views/
COPY public/images/*.jpeg public/images/
COPY server.js server.js
# expose HTTP port
EXPOSE 3000
CMD [ "node", "server.js" ]
```

The **`Dockerfile`** has comments that describe what each command does, but in general, the following steps are taken:

- We download a pre-built Docker image called node:16, which includes a fresh install of Node v16.
- We create a folder inside the container, copying all the Express server files: **`server.js`**, **`package.json`**, the views, and public directories.
- Once we copied all the files needed to run the application, we expose the port 3000.
- We execute the server through the **`node server.js`** command.

When this container is instantiated, each step in the **`Dockerfile`** will be executed in order. The last command starts the application's process, whether it is running in an embedded server or if the Docker container already includes a server for the application to run on.

Notice that, since NodeJS is an interpreted language, we copy the source code directly to the container. If this were a Java or C++ application, we would have to deploy the compiled version of the application.

At this point, we can test whether the container is working correctly by manually building the Docker image and running a container. The following commands build the Dockerfile inside the **`web`** directory and create a Docker image named as **`pizza-place-app`**:

```
cd web
docker build . -t pizza-place-app
```

Docker stores compiled images in a local repository. Whether we compiled it or got it from Docker Hub, every image is first downloaded to the local image repository. We can create a container instance of the image we just created and expose it to the **`localhost:3000`** URL with the following command:

```
docker run -p 3000:3000 -d --name pizza-place-app pizza-place-app
```

Now, we can open Docker's **graphic user interface,** and we will see the container running successfully, as shown in *Figure 9.12*:

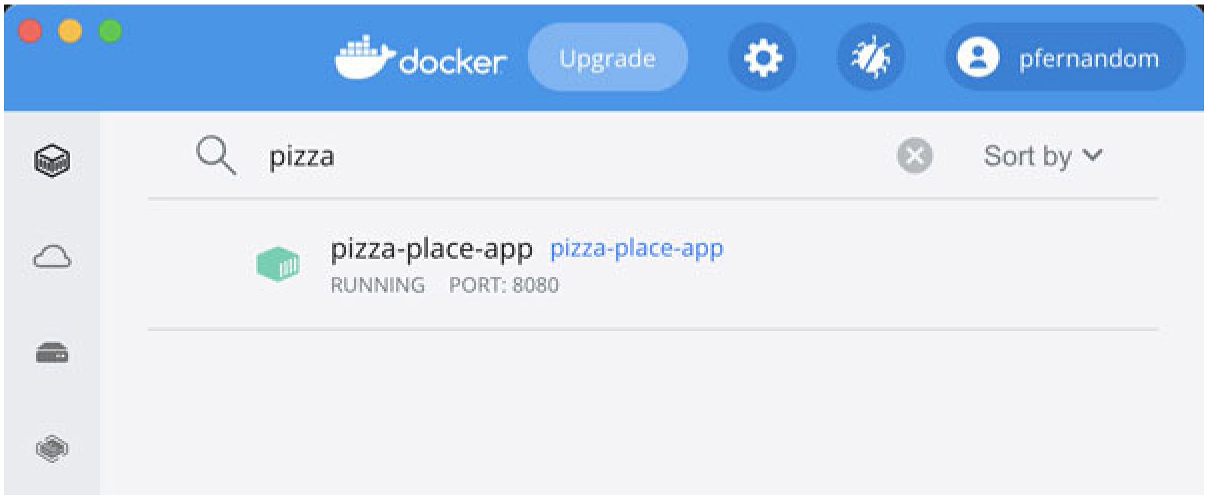
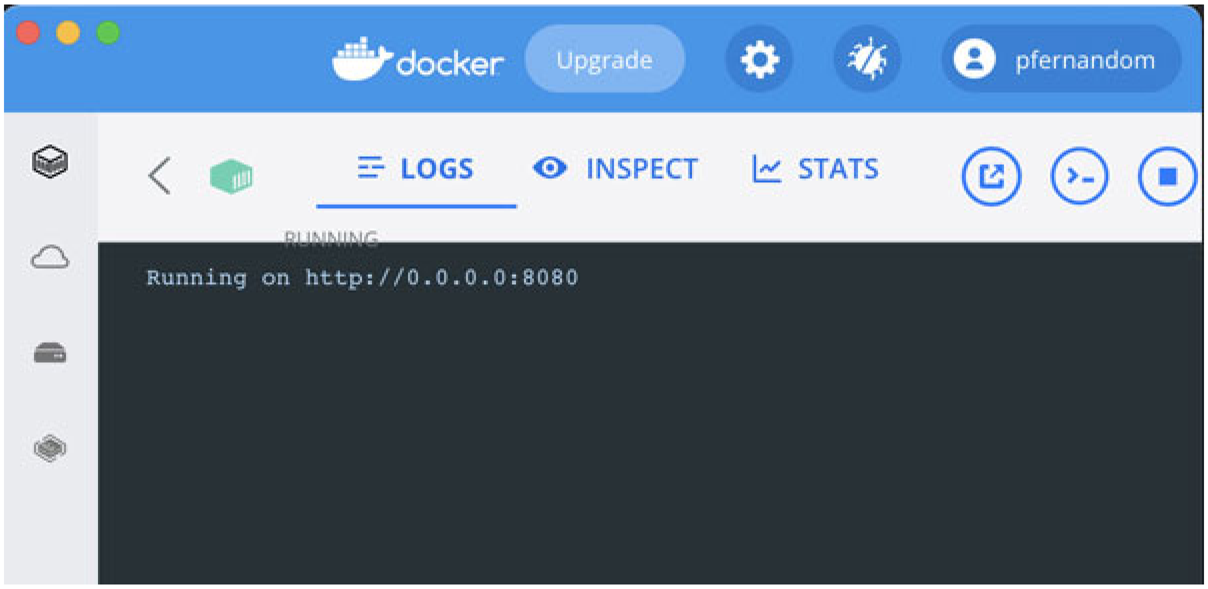


***Figure 9.12:** The new container should be included in the list of running containers*

If we click on the container, we will see its standard output, confirming that the container is up and running. The containers console will be displayed as shown in *Figure 9.13*:



```
Running on http://0.0.0.0:8080
```

***Figure 9.13:** Docker allows us to see the application's standard output*

The nginx directory

The other directory at the root is the **`nginx`** folder. It contains a Dockerfile that describes the steps to instantiate the load balancer container:

```
FROM nginx
```

```
# copy the nginx configuration
COPY nginx.conf /etc/nginx/nginx.conf
# expose load balancer at the following port
EXPOSE 8080
# Start load balancer
CMD ["nginx", "-g", "daemon off;"]
```

The Dockerfile highlights are as follows:

- As we did for the **`web`** folder, we use the base Docker image nginx, which includes an instance of the Nginx server.
- We create a custom **`nginx.conf`** file to replace the default configuration file shipped with Nginx.
- We copy this file inside the Nginx file system to replace the existing configuration file.
- We expose the container's **`8080`** TCP port and start the server.

Note: Docker and TCP

The network adapter used by a Docker container is isolated from the rest of the containers and the host OS. This separation allows us to create multiple containers that publish services to the same (internal to Docker) TCP port. This port is then mapped to a port in the host machine.

In this example, we create three instances of the same Express server, each published in their container's 3000 port, all without conflicts. However, if we need to expose a container's TCP port outside the container (to access the server from the host's browser), we can map the container's port to a TCP port in the host. A specific port in the host can only be mapped to a single container.

If, for example, we needed to access each container's 3000 port from ouside of the Docker internal network, each would have to be mapped to a different host TCP port: "`container1:3000 -> localhost:3000`", "`container2:3000 -> localhost:3001`", and so on for other containers.

The docker-compose.yml and nginx.conf files

Now, to connect everything, these are the contents of the **`docker-compose.yml`** file:

```
version: '3.2'
services:
  webapp1:
      build: ./web
      tty: true
  webapp2:
      build: ./web
      tty: true
  webapp3:
      build: ./web
      tty: true
  lb:
      build: ./nginx
      tty: true
      links:
          - webapp1
          - webapp2
          - webapp3
      ports:
          - '8080:8080'
```

Let us look closely at the services section, which describes all the containers that the Docker Compose stack creates:

```
webapp1:
      build: ./web
```

This part of the configuration means that Docker will create a service called **`webapp1`** by *building the image* whose Dockerfile is contained at the root of the **./web** directory. Docker will then instantiate a container using that image. These are the same actions we took before when we executed the **docker build** command, but in this case, Docker Compose takes care of running them for us.

We are creating three services: three containers with a copy of our Express web server: **webapp1**, **webapp2,** and **webapp3**. We also created a fourth service, **lb**, using the custom Docker image for the Nginx server.

Notice that Docker Compose *uses the service names as domain names* for the Docker's internal network. We can refer to each container through the URLs **webapp1:3000**, **webapp2:3000,** and **webapp3:3000,** respectively. But, again, these domain names only work when used *within the other containers in this docker-compose.yml file*, as the Docker Compose internal network performs the domain name resolution.

The use of these internal domain names makes it easy to configure the load balancer:

```
events {
    worker_connections 1024;
}

http {
upstream localhost {
    # use the domain defined in docker-compose.yml
    server webapp١:3000;
    server webapp٢:3000;
    server webapp٣:3000;
}
server {
    listen 8080;
    server_name localhost;
    location / {
       proxy_pass http://localhost;
       proxy_set_header Host $host;
```

```
        }
    }
}
```

Notice that we only “expose” or map the Nginx port **`8080`** to **`localhost:8080`**, and we do not map each Express container’s **`3000`** port to any host port. We only want users to access the servers through the load balancer and not through each instance directly.

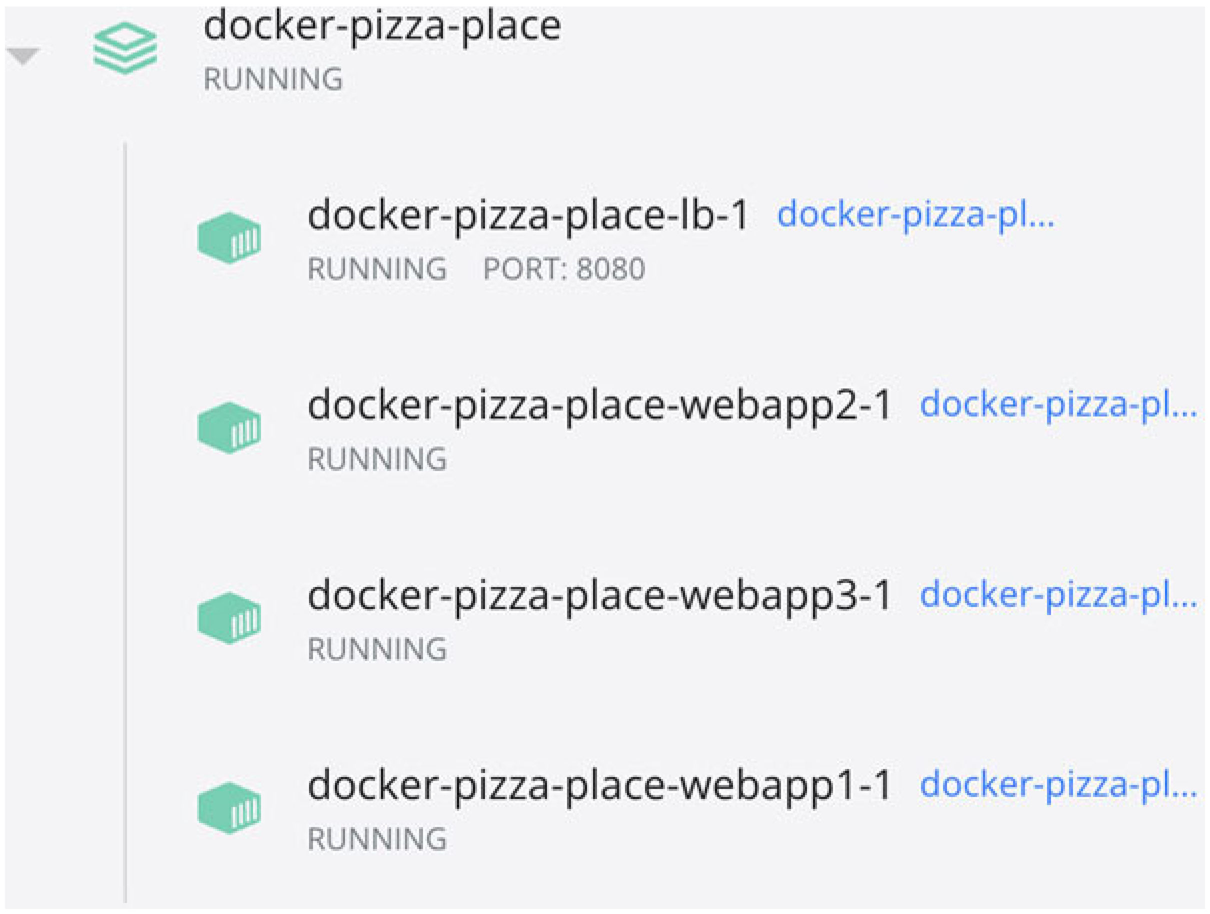
Instead of building each Dockerfile individually, we can rely on Docker Compose to do the job:

```
docker compose build
```

Once all the images for each **`service`** attribute are built, we can start the entire mini-cluster with the following command:

```
docker compose up -d
```

Going back to Docker’s GUI, we will see all three containers for the Express server and the one container for the Nginx server successfully running, close to what we can see in *Figure 9.14* as follows*:*


docker-pizza-place
RUNNING

docker-pizza-place-lb-1 docker-pizza-pl...
RUNNING PORT: 8080

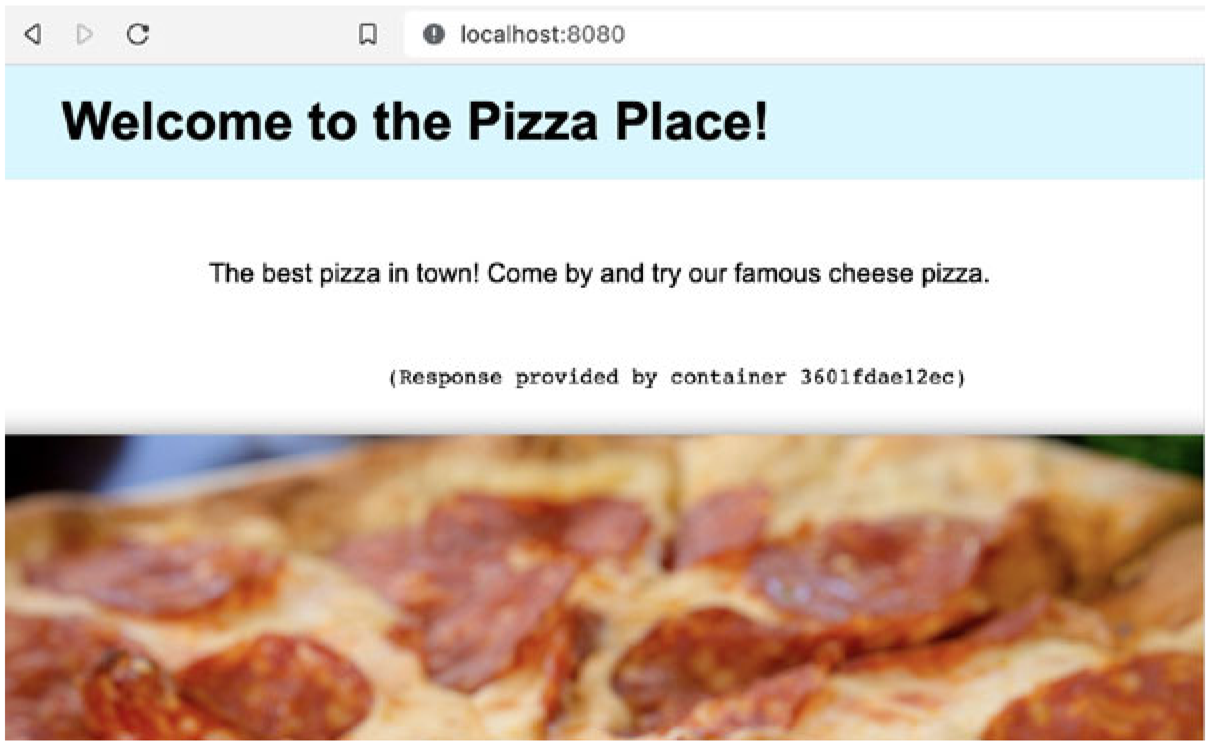
docker-pizza-place-webapp2-1 docker-pizza-pl...
RUNNING

docker-pizza-place-webapp3-1 docker-pizza-pl...
RUNNING

docker-pizza-place-webapp1-1 docker-pizza-pl...
RUNNING

Figure 9.14: *The GUI shows Docker Compose created the containers successfully*

Furthermore, if we navigate through a browser to **`http://localhost:8080`** the Nginx server exposed through the port **`8080`**, we will see the server's home web page, as shown in *Figure 9.15*:


localhost:8080

Welcome to the Pizza Place!

The best pizza in town! Come by and try our famous cheese pizza.

```
(Response provided by container 3601fdae12ec)
```

Figure 9.15: *The Pizza Place app running in a mini cluster*

Each request will hit a different container, as configured by Nginx. Suppose we use the Express application included with this book's source code. In that case, we will find that the web page renders a message that says something like" (`Response provided by container 3601fdae12ec`)", where the alphanumeric string is the ID of the container. As we hit different containers, this string alternates between three values: one for each container.

Adding Git

This Docker Compose setup is excellent for development. We can change the source code, rebuild the images, and re-deploy the updated containers. We can even synchronize the internal container files to the host machine files using *volumes*. However, this setup is not great for a CI/CD pipeline: The files copied to the container may contain changes that are not part of the primary source code repository.

A production-level setup for this example will *checkout the code for the Express server every time Docker creates a container*:

```
FROM node:16
WORKDIR /usr/src/app

# clone a fresh version of the codebase
git clone https://github.com/yourusername/my-remote-repo.git

# navigate to the cloned repository
CD my-remote-repo

RUN npm install

EXPOSE 3000

CMD [ "node", "server.js" ]
```

This small change connects the benefits of VCS with the benefits of containers: provide reproducible environments, end-to-end. Using this Dockerfile, we are confident that each container gets created with tested, reviewed, and validated code. We are not deploying custom, untracked manual changes made to the container.

Docker in CI/CD

For a production CI/CD pipeline, we would like to do some extra steps to increase confidence in the quality of our containers.

First, we would like to decouple creating the container image from creating the container instance. Instead of building an image and immediately deploying it, we will *build it and deploy it to a repository of Docker images*. This repository can be a standalone server or hosted within an orchestration server like Kubernetes.

This separation has several benefits:

- We can create multiple versions of a container's image.

- If we fail to create a new image, we can still use a previous version to keep creating containers.
- Image repositories are historical records we can visit if we need to recreate an instance of an old version of our application version.

A revised CI/CD pipeline for Docker-based applications can be seen in *Figure 9.16*:

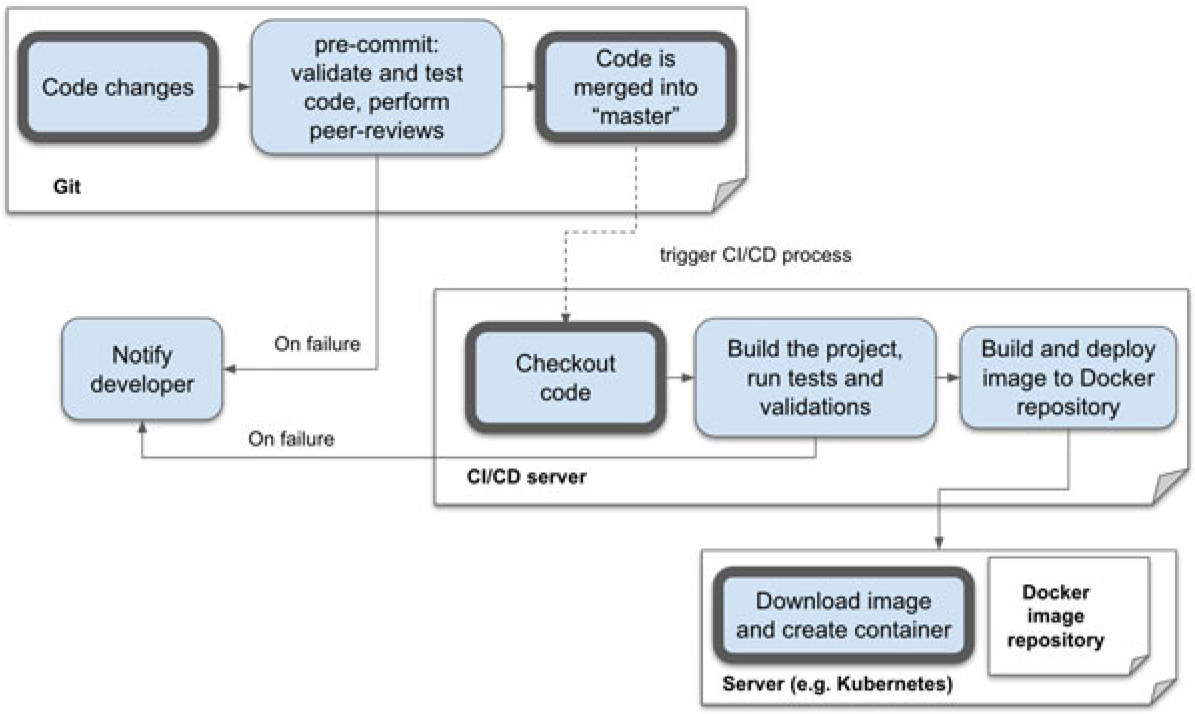

Code changes
pre-commit: validate and test code, perform peer-reviews
Code is merged into "master"
Git
trigger CI/CD process
Notify developer
On failure
On failure
Checkout code
Build the project, run tests and validations
Build and deploy image to Docker repository
CI/CD server
Download image and create container
Docker image repository
Server (e.g. Kubernetes)

Figure 9.16: *CI/CD process for a Docker-based application*

Each component in the pipeline provides visibility to the development team to diagnose problems in the delivery process. If the deployment fails, we can quickly identify the problem by looking at what step in the process was unsuccessful.

The trade-off

The complete CI/CD pipeline we just described is complex. If our application is small, a pipeline like this may be prohibitively expensive in terms of effort and money. Multiple servers and a fair share of maintenance work are required to enable this CI/CD flow.

We have already discussed the problems each component in the pipeline fixes. We need to consult with our teams about how complex our CI/CD flow should be. Small projects may work correctly with only a subset of these tools, but we should not forget the trade-offs this choice brings.

Not all applications are ready to invest in using Docker and all the infrastructure required. Understanding this is good, as it will prevent teams to introduce unnecessary complexity. However, as our application grows in size, containers help save money and effort in the long run.

In the most basic setup, we should always have at least a VCS like Git to keep track of our application's source code and a CI/CD server that builds, tests, validates, and deploys each change. The goal for any experienced developer is to adopt the right toolset necessary to be confident that the application they deliver to their clients is reliable and maintainable.

Conclusion

As applications have grown in complexity, so has the process of delivering them to our users. There is much room for failure during the integration and deployment of an application, which is why we have historically built multiple tools to make this task easier.

We must often integrate and deploy to deliver value to our users faster and find bugs sooner. The only way to reliably do this is by automating the delivery process to CI/CD pipelines.

A CI/CD pipeline relies on reproducibility for development, test, and production environments, even for the source code itself. Being able to easily and quickly create copies of our environments and applications gives us the certainty that we can replace applications in a bad state or scale to meet load requirements.

We rely on **Version Control Systems** (**VCS**) like Git to make the code reproducible. Git keeps a copy of our source code along with all the historical changes that have been done to it. VCS gives us the confidence that we will always have a working copy of the application, even when we unintendedly introduce new defects or a developer accidentally deletes their local copy of the project's source code.

Virtual environments like **Virtual Machines** (**VMs**) and containers encapsulate all the dependencies an application needs, allowing us to configure an environment once and run it everywhere.

Containers offer similar benefits to virtual machines, but they have a lighter memory footprint and operate closer to the application.

Virtualization, in general, provides isolation between applications. Changes like migrating to newer versions of compilers and dependencies can be done safely without adversely affecting other applications running on the same physical server. It also helps to limit the unintended impact one application can have on others deployed close to it.

Overall, CI/CD pipelines provide a well-tested and robust deployment process that allows developers to concentrate on building quality into their applications without losing much time dealing with conflicts and unreproducible issues resulting from the random state in a production server.

This chapter completes the second part of this book. We now have an excellent end-to-end vision of all the moving pieces involved in creating backend applications. From now on, we will focus on building quality in more data-intensive applications and increasing our skills to become more senior developers.

Questions

The following is a list of common interview questions about the deployment of software applications:

- What is the difference between a Docker Image and a Docker Container?
- What is the difference between Continuous Integration and Continuous Deployment?
- What is the difference between a container and a virtual machine? Describe a use case for each.
- What is a hypervisor? What function do Git branches have? What will “git clone” do?

Resources

- Git installation documentation: **https://git-scm.com/book/en/v2/Getting-Started-Installing-Git**
- How to create a new GitHub repository: **https://docs.github.com/en/repositories/creating-and-managing-repositories/creating-a-new-repository**
- Customizing Git – Git Hooks: **https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks**
- Apache Subversion (SVN): **https://subversion.apache.org/**
- “5 benefits of virtualization”: **https://www.ibm.com/cloud/blog/5-benefits-of-virtualization**
- The standard for containers: Containerd: **https://containerd.io/**

CHAPTER 10
Creating High-performance Apps

Application performance is one of the most critical aspects of software development, especially as the application grows and more people adopt it. In this chapter, we describe what factors impact application performance from the point of view of a back-end implementation and what practices can be adopted to guarantee users will not stare at a loading screen and leave our app frustrated.

Structure

In this chapter, we will learn the following topics:

- Measure to improve the performance
    - Synthetic testing versus RUM
    - Using percentiles
- Improving the performance
    - Improving the performance with caching
    - Improving the performance with distributed systems
    - Microservices
    - Improving the performance using asynchronous communication (queues)
        - Improving the performance using asynchronous programming

Objectives

By the end of this chapter, we should have a good idea of what elements impact application performance. We will define effective ways to measure the performance to have a good baseline of our application's metrics. Also, we will explore some techniques to address some of the most common performance bottlenecks.

We will discuss asynchronous programming and its benefits. We will also describe why and how we use distributed architectures to deal with data-intensive applications and how these applications vary from more basic architectures.

Measuring to improve the performance

Ask any optimization expert, and they will tell you that the only way to improve performance in an application is by first measuring the factors that affect it. It is simply impossible to improve the performance of an application when we do not even know what parts of the system need to be improved (if any).

Performance is not a single number we can measure. A large part of performance depends on what users are experiencing. For instance, if a database takes many seconds to execute a query, we may be tempted to consider it a performance problem. Nevertheless, we expect some queries to be relatively slow for some data-intensive applications. If we put appropriate measures like caching in place, the overall user experience may be good enough, and the query may be left as it is. This situation does not mean there is no room for improvement, just that the performance as a whole is a result of multiple factors.

Even if the user experience is subjective, we need objective data to determine if an application is performant. Performance is centered on the response time for live applications like web applications or API services. The response time is the time it takes since a user performs an action until they receive a response from the application.

We can measure response times for many things in an application. We can measure the time it takes since a client sends a request until they get a response from the server; we can measure the time a function takes to complete, a database to return the results of a query, or an external service to return requested data.

From all the things we can measure, how do we know what factors better represent the overall application performance? It depends on the business case. It can be just a subset of critical actions, or it can be all actions. Since each part of the application evolves at a different pace, it is essential to keep track of the response time for each element in the application independently.

For instance, a web application can have multiple pages, and each page can have multiple modules. Each page and module can regress independently from the rest, resulting in a poor user experience even when most of the other modules perform correctly. The main takeaway is *no single number can describe an application's performance.*

Synthetic testing versus RUM

Let us revisit the Pizza Place application we have discussed throughout this book. We can take one module or action in the application (something our users care about) and measure it. An example can be the load of the chronological list of orders for one of our members. We need to ensure a low response time when users request their list of orders.

Since we are diligent developers, we add "*instrumentation*" code to the orders request handler to measure the response time. This response time includes sending the request to the back-end server, querying the database, parsing the results, and returning a JSON with the response.

How do we measure time? In the most straightforward approach:

- We capture the time before we execute the function to be measured
- We capture the time after the measured function completes
- We log the difference.

The following code example shows this process:

```
app.get("/orders", (req, res) => {

// measure time before:
let start = (new Date()).getTime()
const orders = pizzaService.getOrders();

// measure time after:
let end = (new Date()).getTime()

// get the difference:
console.log(end - start, " ms")
res.json(orders);
});
```

We now can start the server, make a request to the **`/orders`** API endpoint, and in the server's standard output, we will see the time in milliseconds it took the **`getMenu()`** function to complete.

Note: The instrumentation code used in the example is the simplest way to measure how long the code takes to complete. However, when we build production-level applications, this code may lack features like capturing and parsing relevant information about the request.

There are more robust ways of measuring response times in tools provided by some programming languages or frameworks. For instance, NodeJS provides a Performance API (more details in the References section) dedicated to recollecting the execution time and memory usage metrics.

When dedicated tools are available, we should prefer those to implement our own instrumentation code.

Pause for a second to consider this question: Is this the real-time it takes to render the menu? There are multiple considerations we need to make to answer this question:

- We only measure the response time for the **`getOrders()`** function. This measurement does not include the time it takes for the request to get to the server, the time it takes to serialize the response, and the time it takes to get the response back to the user. It also does not cover the time other service calls take.
- The response time will heavily depend on the number of records available for **`getOrders()`** to return, and this number may not remain constant in time. In this example, we expect the number of elements in the orders list to remain more or less the same from one request to the next, but other functions may return a significantly different amount of data from one request to the next.
- The response time may also be different from one user to another. Some users may have just one order, while others may have tens or hundreds. Users with high engagement can have magnitudes more data than the average user.

It would be easier to gauge how the response time changes from one version of the application to the next and detect regressions or improvements if we could take out some of this randomness from our measurements.

We call it **synthetic testing** when we run the application in an isolated development or test the environment to collect performance metrics using the instrumentation code. The advantage of synthetic testing is that a controlled environment gives repeatable results, reducing the variability of external factors.

Synthetic testing is helpful for:

- **Regression testing**: Finding if parts of the application have better or worse response times due to code or configuration changes.
- **CI/CD validation**: To stop deployments that significantly increase measure response times, reducing performance.
- Benchmarking different function implementations or versions of libraries.

With all its advantages, synthetic testing cannot escape the fact that the factors we chose to ignore by measuring the response time in a controlled environment matter for the *user-perceived performance*. Things like network latency will always be present in production environments. Synthetic tests are helpful in understanding how individual parts of the application behave. However, they cannot represent the accurate response times users see in our application.

A request's time in network traffic is almost impossible to be accurately measured in a synthetic environment. For instance, the network time will be significantly different for a user near the data center hosting the server than from a user on the other side of the world. A request goes through DNS resolution or proxies redirection, which varies from request to request. Each part of the network can be slow or fail, causing the network to redirect traffic or retry requests. There is a chance that all numbers look good on synthetic testing, and still, users can have a sub-optimal experience due to network latency alone.

In addition to synthetic testing, we need to measure response times *directly at the client* in a production setting, making actual requests. *Figure 10.1* reflects the differences between measuring code and measuring requests on the client-side:

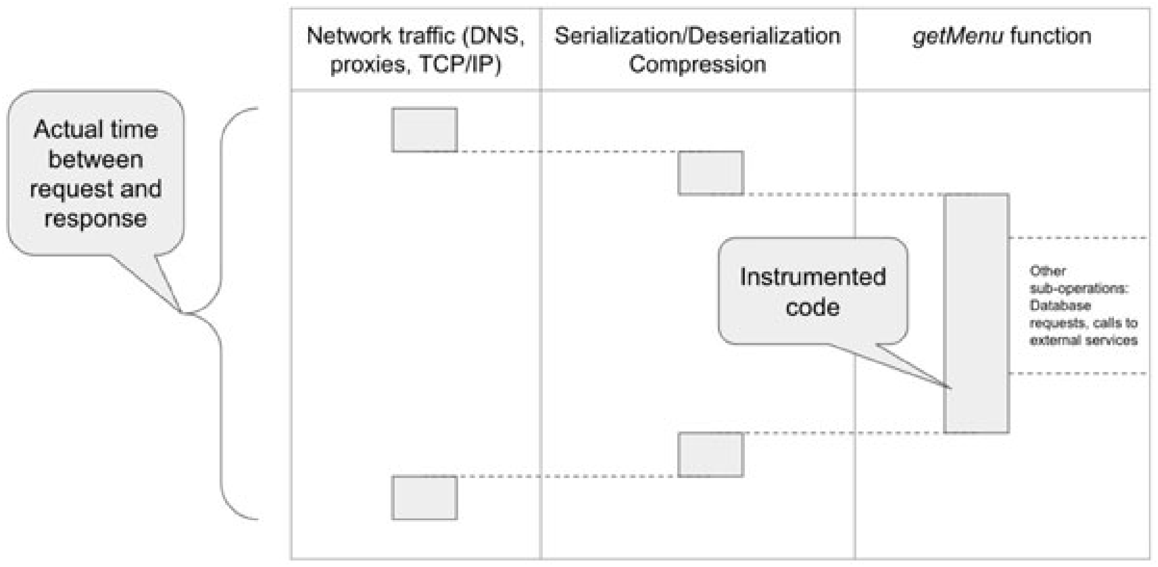

Network traffic (DNS, proxies, TCP/IP)
Serialization/Deserialization Compression
getMenu function
Actual time between request and response
Instrumented code
Other sub-operations: Database requests, calls to external services

Figure 10.1: *Complete flow, starting with a user request and ending with the user receiving a response*

When we measure performance directly in the user's context instead of doing it in a controlled, development, or test environment, we call it **Real User Monitoring** (**RUM**).

RUM metrics are the closest thing we have to capturing the real-user experience. They provide real-time insights into the performance of the measured functions; these insights help us find potential areas of improvement.

Note: Typically, RUM sends real-time performance metrics to a centralized aggregation service. In Chapter 7, "Handling Errors", we used aggregation services to centralize logs. Since we can consider performance metrics as a particular type of log, aggregation of response times follows a similar pattern.

However, unlike text-based log entries, RUM records are aggregated into hourly, daily, weekly, or monthly statistics, providing time-series data about the application performance.

RUM metrics can be captured in multiple places:

- Measure response time directly in the client. We can measure the overall request-response time directly in the browser or mobile app, getting latency numbers on the overall performance for specific actions.
- Measure latency for individual functions or services. We can leverage the instrumentation code we write for synthetic tests to collect real-time

metrics on the application running in production. These metrics will differ considerably from the results of the instrumentation code on synthetic tests; instead of giving consistent numbers from one test execution to the next, RUM data will provide a *statistical distribution* over the latency experienced by real users.

We can correlate these two metrics to find potential areas of improvement: If the response time for a given request is higher than expected, the instrumented code gives us insight into which parts we could fine-tune to reduce the overall request time.

So, which one should we use, synthetic testing or RUM? They are not exclusive from each other. We can use both to get a bigger picture of response times across each part of our application.

This combination gives us an insight into which part of the requests could require improvements: network traffic, serializing requests and responses, database queries, and subsequent requests to external systems, among other operations which contribute to the total response time.

For instance, if requests to a specific endpoint show regression in RUM metrics but not in synthetic testing, there is a big chance the regression is caused by elements external to the application.

Using percentiles

Measuring is comparing. We can only understand an application's performance when we have a baseline to compare our measurements. This baseline is the ideal and realistic response time we aim to achieve. We can affirm that the application is performant if it remains close (or below) to the baseline.

If the response time increases beyond the baseline, we know we have regressed and need to improve the application's performance. If the response time is lower than the baseline, we know we have improved, and it's time to set a new baseline.

To calculate an effective baseline, we need to consider user perception. When we execute an action such as loading a webpage or clicking on a button, humans expect a reaction between **100 and 1000 milliseconds**. We lose the user's attention when an action takes longer than a second to complete. We should aim to complete user requests in under a second to avoid losing user engagement.

As we start collecting response times from our users, we will see an interesting phenomenon: Response times vary for the same user, even between consecutive and identical requests. We can see this in any application: Make the same request multiple times, and we will find that each response time varies, even if just a little. Network latency, which is out of our control, is the cause of most of this variation. Another cause of (positive) variation can be caching, which reduces the response time for consecutive requests after the first one.

Figure 10.2 shows a hypothetical sample of 100 user requests for the same action. We can see that most take under one second, but some take as long as 10 seconds. This sample is typical: Some users are on slow networks or using low-range devices and may experience longer response times.

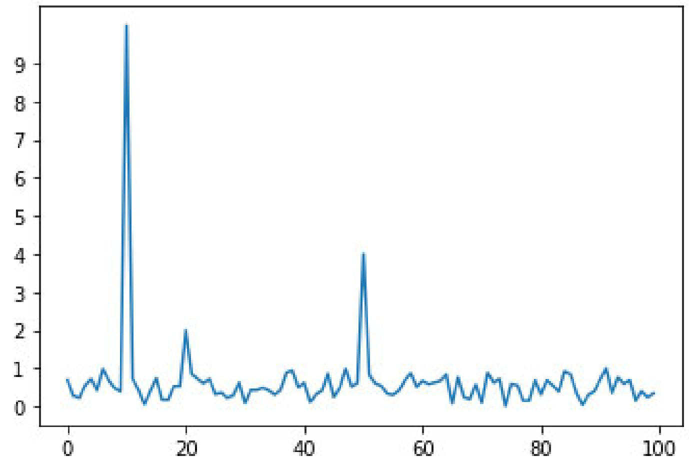


Figure 10.2: *A hypothetical sample of the response time for 100 user requests*

When we get different numbers for the response time, which one should we compare with our baseline? We cannot use the best (0.016 seconds in *Figure 10.2*) or the worst (10 seconds in the example) times because they are likely to be outliers: one-off values caused by particular circumstances; they are not representative of the typical user experience. The average is not helpful either; for example, in *Figure 10.2*, the average is 6.5, which does not represent the typical response time either.

A better approach is to split the requests into **percentiles**. For instance, if we sort all the response times, the lowest 50% of these requests (also known as *p50*) are complete in 0.53 seconds or less. This number means that, at worst, half of the requests take half a second to complete. The other half takes between 0.54 seconds and 10 seconds to complete. The lowest 90% of the requests (or *p90*) take 0.88 seconds or less to complete, and *p99* takes 4 seconds or less.

If we focus on improving the performance for *p90* or *p99* or even *p99.99*, we can safely ignore outliers likely caused by factors out of our control. How many of these outliers we can ignore depend on factors like how many user requests we get on average, how large our application is, and how critical would it be for us if 0.001% of users had a bad user experience.

Percentiles help us find a baseline: For instance, our baseline can keep requests in *p80* under one second and *p99* under two seconds. These numbers are just an example: they can be any other realistic goal that makes sense for our business case.

To accurately calculate our percentiles, we define a sliding window of a given time (a week, for instance) and calculate our percentile metrics using all captured response times inside that window. As time moves and we get new requests, we update the response time for each percentile.

We cannot create an accurate baseline without measuring first. Instrumenting code (adding code to measure performance) and measuring the total response time at the client will provide us with an initial baseline that may not be close to the one-second-per-request goal, but it will help us identify opportunity areas. Let us reiterate this: we cannot start improving the performance until we identify opportunity areas through measurements.

Improving the performance

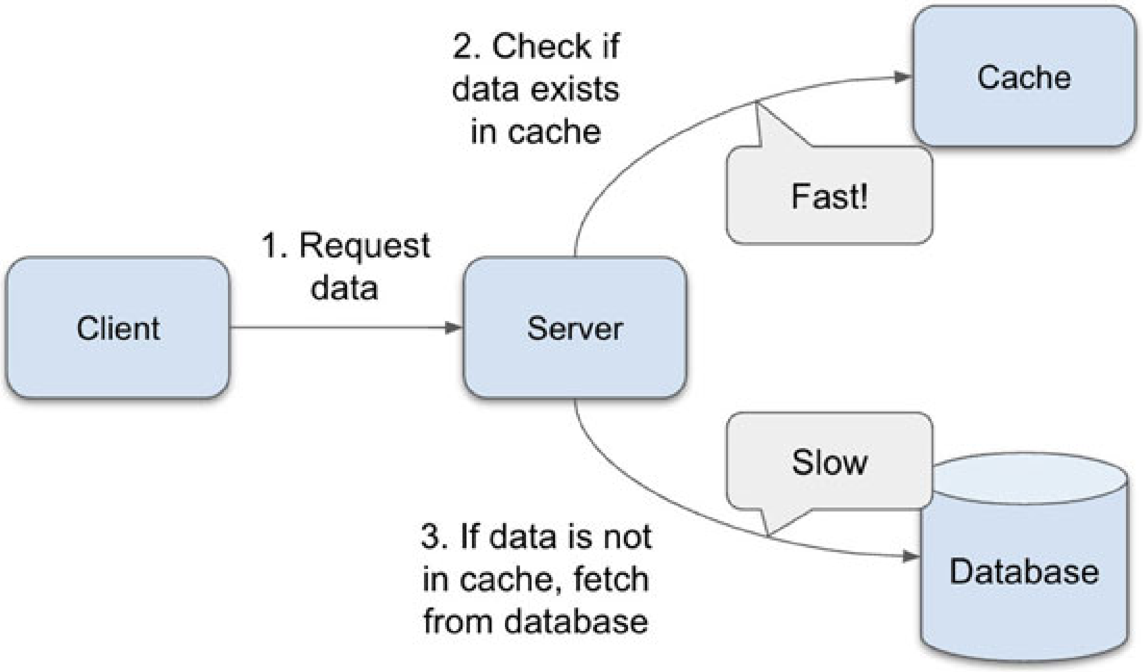
We have multiple approaches to improve the application performance. Some of these approaches are easy to implement preemptively; others require considerable effort and should only be considered once we find opportunity areas through measurement. This section will review multiple practices aimed to improve the application performance.

Improving the performance with caching

Caching is the most common change we can introduce to an application to increase the performance. *Chapter 4, End-to-end Data Management*, briefly discussed using in-memory data structures and distributed in-memory databases as caching layers. Let us expand into the role cache plays in improving the application performance.

A cache is a fast-access storage layer between the server and a data source (a database or an external service). Typically, these data sources take a relatively long time to produce a response. If we were to consume them directly, we would see a negative impact in the response time for the average request, especially as multiple calls to external services are required for completing a single user request.

Since the cache stores data in the local memory, accessing data is faster than fetching it from external data sources. We will reduce the overall response time by retrieving the data from the cache instead of querying the external data source. *Figure 10.3* shows the flow a request follows when using a cache layer:



2. Check if data exists in cache
Cache
Fast!
1. Request data
Client
Server
Slow
Database
3. If data is not in cache, fetch from database

Figure 10.3: *The flow for a read request that has a cache layer in front of the database*

The cache storage has space restrictions due to its in-memory nature. We cannot dump a whole database into the cache, so most work around the cache design consists of working around these limitations.

Note: Memoization

A cache is not always configured as an external product or service. Caching can be done at multiple layers, and the simplest example of this is memoization.

Memoization stores the results of potentially slow functions in a data structure. The function then reuses these results when the function gets called again.

Imagine a slow function called "doSomethingSlow", which receives an integer as a parameter and returns a string. The following is the code definition for that function:

```
Public String doSomethingSlow(int number) { … }
doSomethingSlow(32); // takes 10 seconds
doSomethingSlow(32); // takes 10 seconds
```

Also imagine that, for some unknown reason, each call to this function takes 10 seconds.

This function is pure: It will always return the same string value (with no side effects) for the same input number. Thanks to this property, we can cache the result after the function is called for the first time:

```
Map<Integer, String> cache;
public String doSomethingButFast(Integer number) {
    if (!cache.hasKey(number)) {
        String result = doSomethingSlow(number);
        cache.put(number, result);
    }
    return cache.get(number);
}
doSomenthingButFast(32); // takes 10 seconds
doSomenthingButFast(32); // takes less than a second
```

The first call will be slow because the "doSomethingSlow" function is called but the subsequent function calls using the same parameters will be served from the in-memory hash map, considerably reducing response times. We call this caching pattern "cache-aside".

Memoization is a powerful tool to increase the performance of our algorithms.

Defining reading and writing load

Before we define the most common caching strategies, we need to talk about the specific needs of applications.

We can classify all actions performed in a software application into two categories: **read** and **write**. Fetching a web page, querying a database, and querying an external service, all are examples of *read* operations. Creating or updating a record in a database or uploading a file are examples of *write* operations.

Some applications perform more than one type of operation than the other. A static web application that extracts data from multiple databases (like a complex WordPress site) will be read-heavy. At the same time, a chat application or a log aggregation service will be write-heavy. Most applications balance both types of operations, but one will always be more prevalent. Each of these types of applications will have distinct caching requirements and strategies.

Think of your application. If we log all requests to the application and classify them as either *read* or *write*, which one has a higher count? This question will inform us of effective ways to implement a cache layer.

Cache patterns

A good cache strategy allows the application to "*hit*" the cache as much as possible. The goal would be to have *all* read requests return data from the cache instead of a slower data source, but this is neither easy nor necessary to achieve in some cases.

Depending on the characteristics of our application, we fine-tune the cache layer to balance out getting the most cache hits while keeping the budget and complexity down.

Before we discuss cache patterns, let us get some terminology out of the way. When we talk about a "*data source*", we refer to a database, an external service, or any other source of information that can be queried and may (or may not) be slow to provide a query response.

When we talk about "*data records*" or just records, we talk about data that can be uniquely identified through one or more of its attributes. A data record can be

- A database record that is identified through an ID
- The result of a function that can be identified by its input parameters.

Caches often take the shape of a key-value store (like a HashMap) that allows us to check the existence of records through the unique key.

One last thing! This section will assume that reading from a data source is slow, but only because it is slower than the cache. Reading from a data source can objectively fast (in the range of milliseconds), but it will still be slow relative to reading from the cache.

Remember that accessing data sources can have overhead, especially around network latency: The server may be on a different continent and network traffic alone can contribute for most of performance problems.

Cache-aside and read-through cache

With cache-aside, the application checks the cache directly for the requested record during a read operation. There are two options:

- If the cache contains the record, the application retrieves it and returns it to the client.
- If the cache does not contain the record, the application queries the data source, updates the cache, and returns the record.

The application performs each step in this flow, independently communicating with both the cache and the data source. The flow was displayed in *Figure 10.3*.

A similar strategy is **read-through**, where the application only reads the cache and not the data source; the cache is in charge of fetching the record from the data source if the record does not exist in the cache. *Figure 10.4* shows the requests' flow when using this pattern:

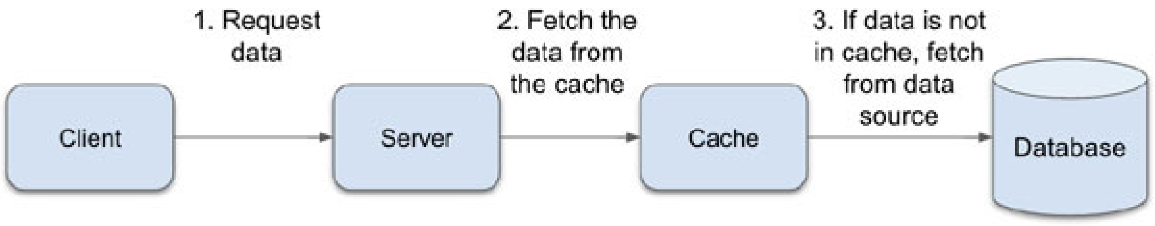

1. Request data
2. Fetch the data from the cache
3. If data is not in cache, fetch from data source
Client
Server
Cache
Database

Figure 10.4: *Request flow for the read-through cache strategy*

Both cache-aside and read-through cache perform the same steps; the difference is that in cache-aside, the application is in charge of updating the cache, while in the read-through strategy, the cache works as a proxy to the data source, and the cache takes care of updating itself when the record is missing.

The advantage of cache-aside over read-through is that *the application can still operate if the cache is down*, while with read-through, the code which calls the cache would have to provide a pass-through call to the data source if the cache service is down. On the other side, cache-aside is slightly more prone to bugs, as the developer is in charge of managing and updating the cache through the application code.

Due to the nature of both these strategies, the first time the application requests a record will always be a cache miss: The cache is empty until the application starts reading data.

We can avoid some of these cache misses by *warming up* the cache: We directly request records so they can be cached before users start making their own requests. Since we cannot warm the cache for all records, we only do so for a set of records that we think are highly likely to be requested by users.

Caching write-heavy applications

Some applications will see little gain from using read-through or read-aside caches. Some examples of such applications are:

- Applications where most records are only read once. For instance, an API that reads hourly data from weather sensors will rarely read the same record more than once, resulting in most read operations being cache misses.
- Applications that mostly do *write* operations.

Now that we talk about *write* operations, it's a good time to ask: What happens when a cached record is updated in the data source? What should we do with the cached version of the updated record? We have multiple options here:

- Use a time-to-live (TTL).
- Update the cache when the record is modified in the data source (update-on-write).

The **time-to-live** (**TTL**) is the longest time a record will be kept in the cache. After a record reaches the end of its TTL, it will be fetched again from the data source on the next read operation (for example, update-on-read). However, if the cached record is modified in the data source, the cache will not update the record until the TTL is done.

Using a TTL works when returning slightly outdated data is not a problem. TTL-based caching works fine for data that rarely changes or whose changes are not time-sensitive. For instance, users of an application that allows people to see videos of birds in distinct geographic locations would be fine if they don't see the latest uploaded video for a few hours.

For the second option (update-on-write) we can introduce two other common caching strategies: *write-through* cache and *write-behind* cache.

Write-through cache

When a "*write*" operation arrives, the application writes directly to the cache. The cache then blocks the execution until it finishes writing in the underlying data source. When the write is complete in both the cache and the data store, the cache successfully returns execution to the application. *Figure 10.5* shows the flow a request follows in a write-through cache:

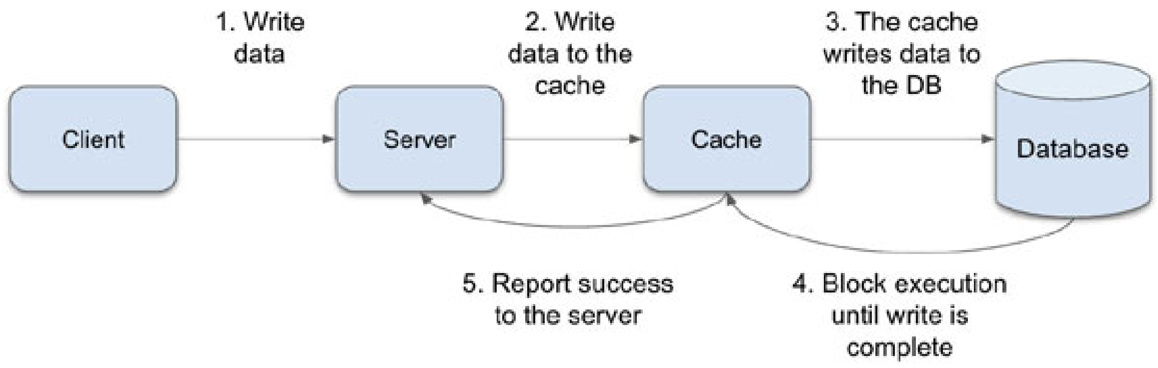

1. Write data
2. Write data to the cache
3. The cache writes data to the DB
Client
Server
Cache
Database
5. Report success to the server
4. Block execution until write is complete

Figure 10.5: *Request flow for the write-through cache strategy*

In most cases, combining the read-through and write-through strategies will keep the cached records up to date. In addition, write-through caches can reduce the amount of cache warming we need to do to populate the cache with data.

One thing to consider is that, just as *read* operations can be slow on external data sources, *write* operations can be time-consuming too (for instance, think of *transactions* in relational databases). With the write-through cache, *write* operations are still slow as the execution pauses until the cache updates the data source. If, in addition to optimizing the read performance, we need to optimize the write performance, we can use *write-behind* caching.

Write-behind cache

In this strategy, the application writes directly to the cache and the cache is still in charge of propagating the "*write*" operation to the data source.

What makes write-behind different is that the data source is updated **asynchronously**. As soon as the data is written in the cache, execution returns to the application immediately. The cache will update the data source in parallel, and the application will not wait for it to finish. *Figure 10.6* shows the flow of this cache strategy:

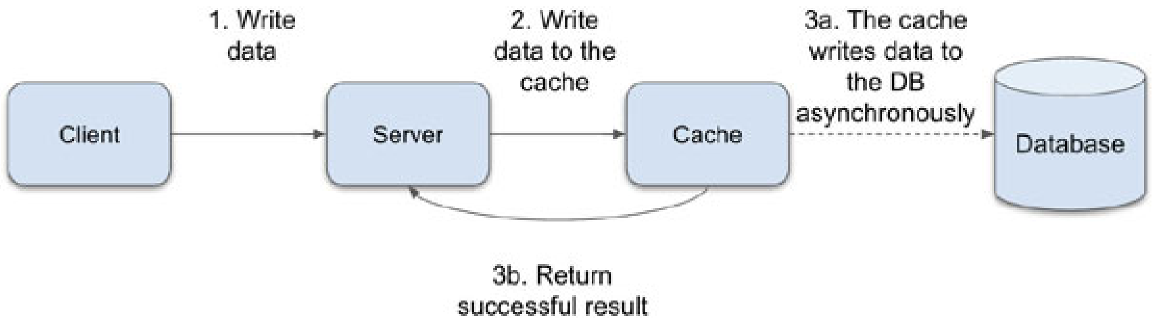

1. Write data
2. Write data to the cache
3a. The cache writes data to the DB asynchronously
Client
Server
Cache
Database
3b. Return successful result

Figure 10.6: *Request flow for the write-through cache strategy*

Internally, the cache keeps a queue of all non-persisted records where the cache inserts new *write* operations. One by one, these writes will be asynchronously propagated to the data source.

Since the application does not have to wait for the slow data source, write operations are sped up considerably. Both reads and writes are performed in the cache directly, keeping the cache updated.

Choosing the right caching strategy

The limited space provided by the caching layer forces us, developers, to analyze our application behavior before choosing a caching strategy. Some caching strategies like *write-through* or *write-behind* will make performance *worst* for some specific use cases.

For instance, imagine an application where most *write* operations are performed on the data source "*A*" and most *read* operations are done on the data source "*B*".

In this case, caching each *write* operation will not improve the performance: The cache will be full of data from "*A*", and each *read* operation will be a cache miss. Then, we will need to delete data from the cache to make space for "*B*" records. *Figure 10.7* captures this scenario:

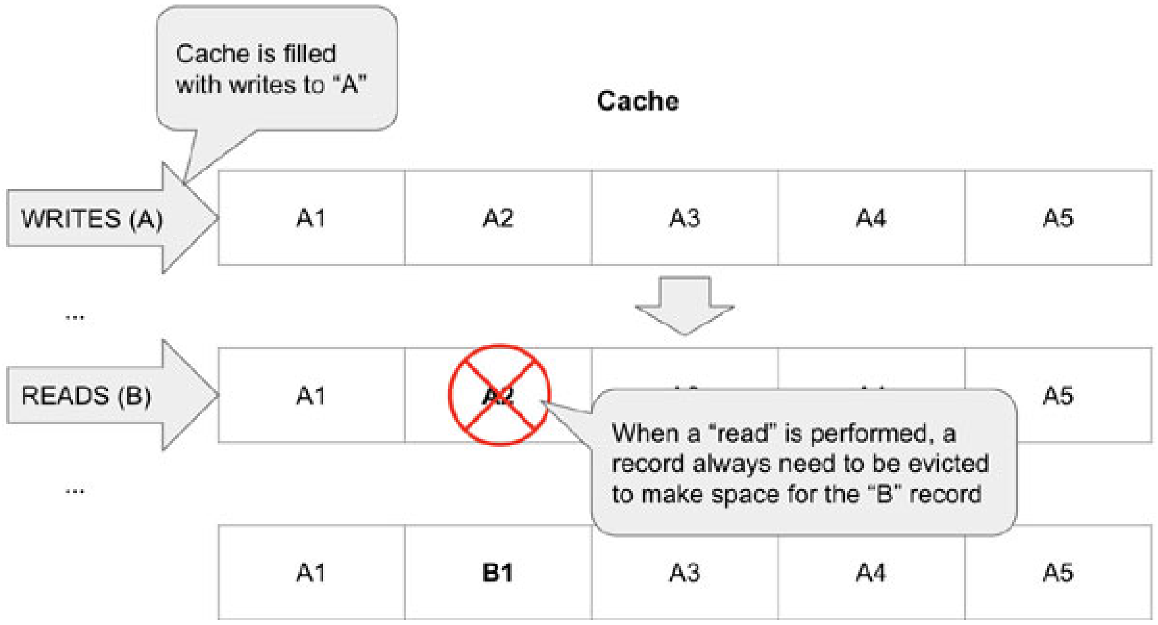

Cache is filled with writes to "A"
Cache
WRITES (A)
A1
A2
A3
A4
A5
...
READS (B)
A1
A2
A5
When a "read" is performed, a record always need to be evicted to make space for the "B" record
...
A1
B1
A3
A4
A5

Figure 10.7: *Poor performance caused by most "read" operations resulting in cache evictions*

Not only the application has to read and write data to the data source directly, but it also has to update the cache at each operation. The application will perform better if we drop the cache layer altogether.

In these complex scenarios, we must design a more nuanced cached strategy. We use the caching strategies discussed in this chapter to design a custom strategy that works well within our application's context.

Eviction policy

Given the limited size of the caching layer, at some point, the cache will run out of space. If we want to keep the cache in sync with the most up-to-date data in the data source, we must delete or *evict* records from the cache.

Ideally the cache should contain data that is likely to be requested in the future. For instance, a record that was just read or written has a high probability of being read again. We must keep this in mind when we decide what records to evict. Evicting data from the cache is critical to keep it performant.

However, we need to reduce the number of cache evictions for two reasons:

- Evictions add application overhead. The cache has to choose a record to evict, delete it, and insert the new record. Each of these operations adds application cycles.

- Evictions often follow cache misses. Cache hits rarely require updating the cache contents.

An eviction policy is a strategy we use to decide what records to remove from the cache when we need to make space for more records.

Simple eviction policies are:

- **First-in-first-out** (**FIFO**): Evict the oldest record in a queue
- **Last-in-First-Out** (**LIFO**): Evict the record at the top of a stack
- **Random Replacement** (**RR**): Randomly pick an element to evict.

The advantage of these policies is that they are simple and perform pretty well. However, these policies do not consider how often a record is accessed. Evicting a "popular" record will guarantee that the record will soon be added back to the cache, resulting in tons of cache misses in the process.

The most popular cache strategy is **least-recently-used** (**LRU**), where, as its name indicates, the least recently used item is evicted first. This strategy prioritizes records that have been recently accessed, keeping them in the cache longer under the assumption that they will probably be used again soon.

Another strategy is **least-frequently used** (**LFU**), where instead of keeping track of when a record was read, we keep track of how many times the cached record has been accessed. LFU prioritizes records that are often read, even if they have not been used in a while.

Variations of these strategies exist. Some keep track of extra attributes like how long the record has been on the cache or how important a record is under business rules (for example, don't evict critical business records). Each approach aims to model different types of data behavior in an application.

Finding the right combination of caching and eviction strategies requires effort, but in the end, it is worth it: caches reduce response times by using faster storage and keeping data close to the user.

Other caching tools: Proxies and CDNs

Not all caching strategies involve in-memory data structures or services. Some caching tools can be remote services outside the application, but they reduce the overhead of common performance problems like network latency.

Most web applications take advantage of two common caching mechanisms: Proxies and CDNs.

Proxies are servers that sit in between the remote server and the client. Network traffic is redirected through the proxy; the proxy can cache requests so, the next time the client makes the same request, the proxy can directly return the cached response.

CDN (Content Delivery Network) are storage servers that keep a copy of static resources like images and text-based files like HTML, CSS, and JavaScript.

CDNs are distributed around different geographic locations; when the client requests static resources, it can choose the CDN that is geographically closest to it, reducing network latency.

Both proxies and CDNs address the problem of network latency: They reduce the distance a network request has to travel to be fulfilled.

A good thing to notice here is that proxies and CDNs don't necessarily store cached data on in-memory storage. Even if they store data on disk (as the remote server would), they can still serve requests faster due to being located closest to the user.

Use case: Caching long-running operations with Redis

A popular caching service is Redis. As mentioned in *Chapter 4, End-to-end Data Management*, Redis is an in-memory database that provides fast *read* and *write* operations.

While it is common for development teams to host their own server instances of Redis in standalone servers, cloud providers like AWS offer services like ElastiCache, which are managed Redis servers that can scale as the demand grows. These managed services help development teams to avoid the work of managing and scaling the servers themselves.

Most modern programming languages offer libraries that can connect with Redis. For instance, Java has the open-source libraries *Jedis* and *Redisson*. In this section, we will see how we can integrate Redis into our Java application to cache the results of potentially slow operations.

Using Jedis

Once Jedis has been added as a dependency to our project, we can connect to a Redis server as follows:

```
JedisPooled jedis = new JedisPooled("localhost", 6379);
```

In this example, we connect to a Redis instance running in our development computer. We use the **`JedisPooled`** object to create a pooled connection to the Redis server. Using a pool of connections is a thread-safe way of integrating with Redis, as the connection can be safely returned to the pool once a client finishes with it.

Once we have a connection object, we can start adding data to the cache. The simplest data structure in Redis is a *string*. The following line of code adds a string with the value **`john.doe`** to Redis under the **`loggedInUser`** key:

```
jedis.sadd(“loggedInUser”, “john.doe”);
```

In our application, we would be dealing with more complex data structures. Instead of storing strings, we can store data structures like hash maps using the **`hmset`** function:

```
public void createUser(User user){
Map<String, String> loggedUser = new HashMap<String, String>();
loggedUser.put(“username”, user.getUsername());
loggedUser.put(“email”, user.getEmail());

jedis.hmset(“user:” + user.getUsername(), loggedUser);
}
```

Retrieving the record from the cache is just as straightforward:

```
public User getUser(String username){
Map<String, String> userMap = jedis.hgetAll(“user:” + username);
User loggedUser = new User();
loggedUser.setUsername(userMap.get(“username”));
loggedUser.setEmail(userMap.get(“email”));
return loggedUser;
}
```

Jedis is a good library for implementing a cache-aside strategy. However, if we want to implement a more advanced caching strategy, we should look for another library.

Using Redisson

Redisson, a Java library that provides a more advanced toolset. We will use it to implement a read-through cache.

For this example, our read-through cache will read user data from an SQL database where the **username** attribute is the primary key. The cache service will only query the database if the user record does not exist in the cache storage.

First, we connect to the Redis server:

```
Config config = new Config();
config.useSingleServer()
.setAddress(“redis://localhost:6379”);
RedissonClient client = Redisson.create(config);
```

Notice that we use the same server URL we configured in Jedis. We use the Redisson -provided **Config** class to provide the connection configuration. For a production setting, Redisson allows us to use configuration files to declare all connection details.

The first step to implementing a read-through cache with Redisson is to define a **MapLoader**. The **MapLoader** is a Redisson interface that defines two functions: **loadAllKeys** and **load**. The **load** function contains the code to execute the SQL query we expect the cache to perform after a cache miss:

```
MapLoader<String, User> mapLoader = new MapLoader<String, String>() {

    @Override
    public Iterable<String> loadAllKeys() {
        List<String> list = new ArrayList<String>();
        Statement statement = conn.createStatement();
        try {
            ResultSet result = statement.executeQuery(“SELECT username FROM users”);
            while (result.next()) {
                list.add(result.getString(1));
```

```java
            }
        } finally {
            statement.close();
        }

        return list;
    }

    @Override
    public User load(String key) {
        PreparedStatement preparedStatement = conn.prepareStatement("SELECT username, email FROM user where username = ?");
        try {
            preparedStatement.setString(1, key);
            ResultSet result = preparedStatement.executeQuery();
            if (result.next()) {
                User user = new User();
                user.setUsername(result.getString(1));
                user.setEmail(result.getString(2));
                return user;
            }
            return null;
        } finally {
            preparedStatement.close();
        }
    }
};
```

Using the **`MapLoader class`**, Redisson will create an instance of a cached hashmap:

```
MapOptions<String, User> options = MapOptions.<String,User>defaults()
                                  .loader(mapLoader);

RMapCache<String, User> cachedMap = redisson.getMapCache("test", options);
```

The application will use the **`cachedMap`** object to access the users' data directly:

```
User currentUser = cachedMap.get("john.doe");
```

If the user with the username **`john.doe`** does not exist in the cache, the **`MapLoader`** function **`load`** will be called, the SQL database will be queried, and a new **`User`** object will be added to the cached hashmap. Next time the application tries to retrieve the same user, the cache will return the instance stored in memory.

As we can see from the example, implementing a read-through cache with Redis and Java's library Redisson is pretty straightforward. We have included Redisson's documentation in this chapter's *Resources* section. It provides code examples for implementing other advanced caching strategies.

Improving the performance with distributed systems

We can tie most common performance problems to two causes: network latency and memory management.

As we have discussed, network issues cannot be avoided, only mitigated. An effective way to reduce the impact of network latency is to put the server as close to your users as possible. Fewer network jumps and redirections are needed when a server is geographically close to its clients. Unfortunately, when everyone in the world has access to your application, there is only so much you can do with a single server to get close to your users.

As our application's traffic surges, so will our server's demand for memory. A single server can be vertically scaled, but it has a limit. After the server hits that memory limit, it will no longer be able to serve all the extra requests clients send to it.

Fortunately, we can address network latency and memory usage challenges with the same approach: Distribute the application to two or more servers.

From the point of view of network latency, using multiple servers enable us to put servers closer to where most of our clients are. On a scale where at one end we have one single server at a considerable distance from most clients, and at the other end, we have one server next to each client, we can find the sweet spot somewhere in between. Cloud providers like Amazon Web Services use the concept of regions and availability zones: geographic areas where we can put multiple servers and be close to users.

Note: Regions and availability zones have multiple benefits. By distributing multiple servers into different regions, our application gains availability assurances if one or multiple regions go offline.

From the point of view of memory management, using multiple servers allows us to distribute our client requests. Since each server receives only a part of all requests, the server's memory usage will be lower than when a single server had to dispatch responses for all requests.

It is easy to get lost when we first start exploring distributed architectures. There are large gaps between the mental model of writing a single-server application and the mental model of designing a multi-server, distributed application.

Distributed applications improve the performance by partitioning data, which means splitting data into multiple servers. The two most common ways in which data is partitioned are:

- Through **replication**: Create multiple copies of our data, one on each server. If we can guarantee that each server holds a fully up-to-date copy of our data, clients can make requests to any server but usually, to the server closest to them distributing all traffic. *Figure 10.8* shows three replicas of the same database instances. Examples of distributed applications that follow this pattern are: Web servers each with an identical copy of the hosted website behind a load balancer. **Content Distribution Network** (**CDN**): Multiple servers that host static resources like CSS, JS, and HTML.

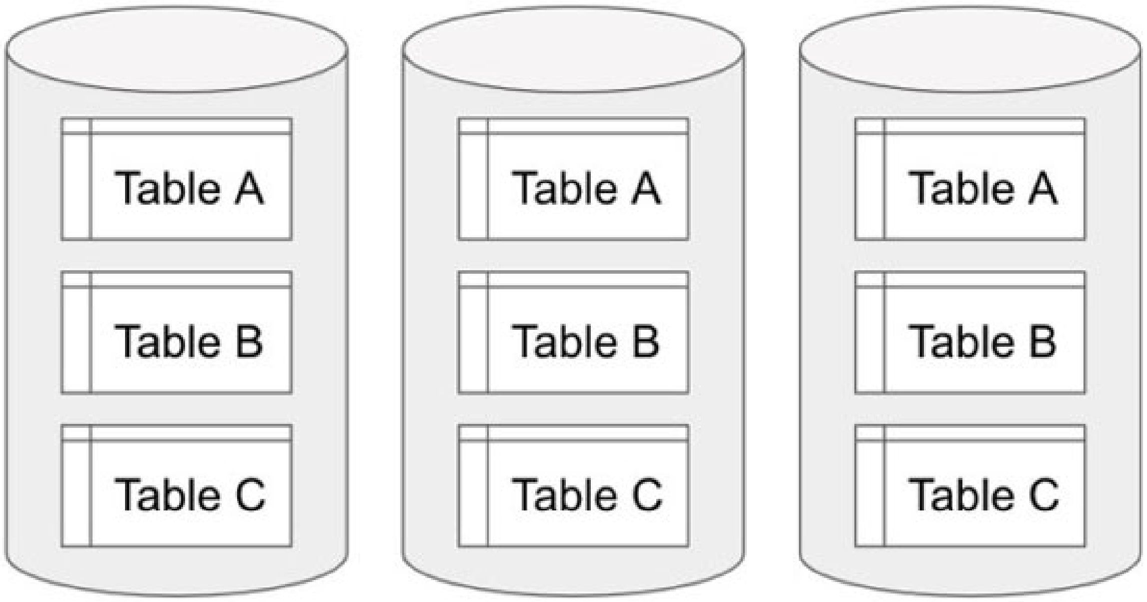


Figure 10.8: *For replication, each server hosts a copy of the data*

- **Through sharding**: Keep a single copy of our data, but distribute it across multiple servers. A shard is a server that only contains a subset of all data. Since each server only holds a part of the data, client requests will be distributed and redirected to the respective servers, reducing each node's load. *Figure 10.9* shows three shards, each hosting one part of the data:

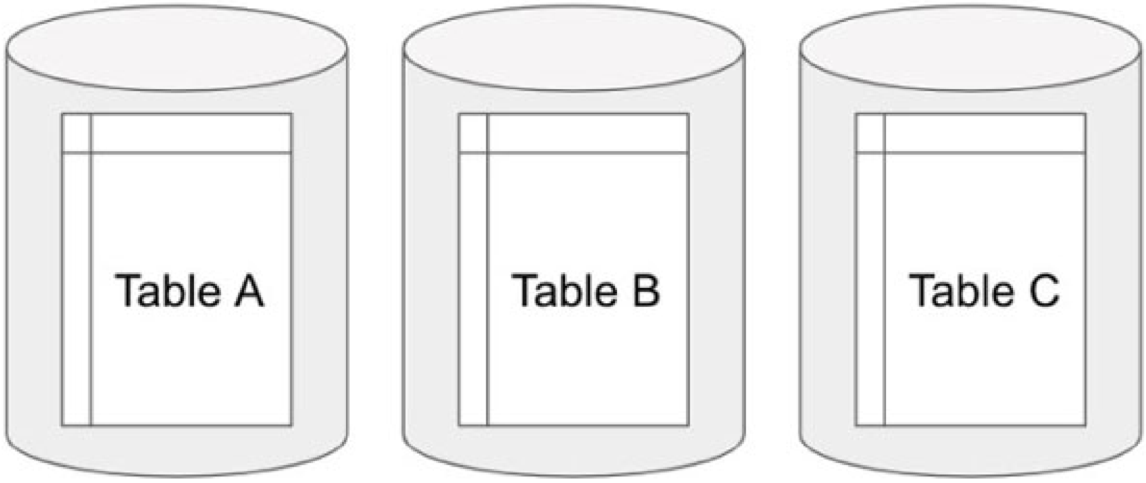


Figure 10.9: *For sharding, each server hosts only a part of the data*

Both strategies are not mutually exclusive. We can have a hybrid approach where we both replicate and shard our data across multiple servers.

The two biggest challenges while designing a distributed application are:

- Keeping data consistent across partitions.
- Enable integration and communication between each server in the application.

If we can solve these two challenges, the remaining design work is not too different from designing single-server applications.

Keeping data consistency

As discussed in *Chapter 4, End-to-end Data Management*, we start having consistency challenges as soon as we keep multiple copies of data. There is no problem when applications are static: We can create as many copies as we want, without any special considerations. But, as soon as we need to support data updates, keeping consistency becomes a challenge.

Data consistency in replicas

Assume we have three replicas of a database. Each database is a perfect copy of the other. We can allow clients to query any of the replicas without a problem because they will all return the same results. No inconsistencies. However, things get interesting when we start allowing clients to create, update, or delete records in the database.

For enabling updates, let us start from what may seem the simplest approach (although, as we will see soon, this is actually the most complex scenario): Allow clients to send *update* requests to *any* replica.

Imagine a client creates a record in "*Replica 1*". Now, all clients querying replicas 2 and 3 will get out-of-date query results. So, the application needs to propagate the update by adding a new record to the other replicas. As soon as replicas 2 and 3 have the new record, queries will be consistent again.

What happens if one client updates record number 12 in "*Replica 1*", but at the same time, another client deletes the same record (number 12) in "*Replica 2*"? We may feel tempted with giving priority to the update that was performed first. This approach has a few challenges:

Figuring out which request finished first is hard. Each server's internal clock may be out of sync.

- How do you determine which request finished first if the clock inconsistencies are in the range of milliseconds?

- One of the two updates will be lost. If this is a critical operation, like a bank transaction, lost updates can have serious consequences, like financial losses.
- Conflicts are detected too late. Each replica will try to propagate its respective updates immediately. The conflict will only be detected when one of the conflicting replicas tries to propagate its update to the other, and by then, other replicas may have been propagated already, making conflict resolution even harder.

Even if we configure our replicas to solve conflicts automatically, the whole process takes time. In the meantime, users will see inconsistent results.

Merging asynchronous conflicts is hard. In *Chapter 9, Deploying Applications*, we saw that Git requires developers to merge conflicts manually. However, applications may not have the luxury of having a human resolve each conflicting update, especially if we start getting hundreds of conflicts per second.

Of course, there are more efficient ways to deal with update conflicts automatically. Conflict resolution is a fairly advanced topic and is out of this book's scope. Many resources (and whole books) are dedicated to this field; if you are interested in the topic, you should check them out.

Since enabling clients to update any replica is difficult, let us explore alternate approaches. For instance, we can send all *update* operations to a single replica.

Multiple read replicas, single write replica

Sending all *update* operations to a single replica is not a new idea. Back in *Chapter 4, End-to-end Data Management*, we discussed having a *source of truth*: A single copy of the data that is always up-to-date and can be used to create other copies. By assigning the role of *write replica* to a single server (and *read replica* to the rest), conflicts are easier to manage. The *write* replica can resolve all conflicts before propagating the changes to the *read* replicas.

Of course, this simplified approach also has its own trade-offs. Since all *update* requests will be redirected to the *write* replica, its load and memory usage will increase. This may not be an issue if the application is read-intensive, but write-intensive applications will struggle to keep acceptable performance levels.

Again, hybrid approaches can help here. Instead of having a single *write* replica, we can have two or three. It is slightly less complicated to implement automatic advanced conflict resolution strategies in two replicas than to do so in all.

Cluster management has more complexities that are out of the scope of this book: What happens when the single *write* replica fails? How does the cluster select what replica to *promote* to replace the failed node? What happens with pending *write* operations? We have added a link titled "*Leader and followers*" in the resources list for this chapter to follow up in all these questions.

In the end, the strategies we have to implement to keep data consistent across multiple replicas depend on the data demands of our application. Most applications can perform well with a single-write-replica, multiple-read-replicas approach.

Eventual consistency versus strong consistency

When data is written or updated in a distributed database, it may take time for all the replicas to be updated. This time can be in the range of milliseconds to minutes. On top of this, the propagation can fail if one or more of the replicas are down.

While *write* operations are propagated, database replicas that are not yet updated could receive *read* and *write* requests, bringing the risk of users operating on outdated data, which in turn can cause conflicts in future *write* operations.

To prevent data conflicts caused by the replication *lag* during *write* operations, we have the following two options:

- **Strong consistency**: Wait until all replicas are updated before confirming to the user that the write was successful. We can lock all *write* operations done for the given record in all replicas until replication finishes. The data introduced by the *write* operation will not be visible in any replica until the replication is complete; meanwhile, we will return the unmodified version of the data for all *read* requests (but *writes* will be blocked until progpagation completes).
- **Eventual consistency:** Confirm to the user that the write was successful as soon as the first replica is updated. *Write* operations are not blocked. All servers will return the data they have (updated or not) for *read* operations and eventually, all nodes will show the up-to-date data when the propagation completes. If new write requests are sent before

replication completes, the previous write operation may be overwritten and lost.

Strong consistency trades off the **performance for consistency**. Having to lock all replicas during the propagation reduces the number of requests the application can serve. An application that does not need strong consistency can gain performance by relaxing consistency constraints.

We can think of examples of cases for each of the consistency models. A bank application needs strong consistency, as it is critical to always show the most up-to-date data about the users' accounts; and updates should only be displayed once the application is certain the *write* operation was successfully propagated to all nodes. On the other side, eventual consistency in a bank application can lead to duplicate money transfers and lost money.

However, a service to post comments on a social network can deal with inconsistent data for a while by showing an outdated list of comments until the replication completes. By using eventual consistency, this application will support more client requests, resulting in increased performance. And if a comment is overwritten, data loss has a minor negative impact due to the nature of the application.

Data consistency in sharding

When using sharding alone, data consistency is not as much of a challenge as it is for maintaining multiple replicas. Since we only keep one single copy of the data, *update* operations don't need to be propagated. The challenge with sharding is to define a good design that allows the application to fully take advantage of having multiple servers.

As discussed in the section about caching, not all data is accessed at the same rate. Some records will be queried more often. This imbalance in operations may lead to some shards having more traffic load than others. If the imbalance gets too bad, some shards will have to deal with the majority of client requests, while others may remain unused.

We can group data in multiple ways. For instance, if we have users from around the globe, we might group data by country. If requests from each country are evenly distributed, a good sharding strategy is to create a server per country.

However, if our assumption is wrong, and we end up in the case where most of our users reside in the same country, one of the shards will get most of the traffic, while the others will be wasted. In this case, we will gain no performance improvements from sharding; we will need to find a better partition strategy, maybe one based on custom regions.

Defining an effective partition strategy requires us to take a good look at our application and our data. Data evolves with time, and we might need to add more partitions to deal with increased traffic. What may be a good partitioning strategy at the beginning of a project may become an outdated approach later on.

Microservices

Just as we partitioned and sharded data into multiple servers to increase performance, we can partition the application itself. Typically, we break the application into multiple, independent services that we can build and deploy independently.

Microservices-based applications rely on composability. Instead of building one single service that performs multiple functions, we build multiple *microservices*, each with a single responsibility. Composability brings flexibility: We can build increasingly complex application flows by integrating multiple services. As the application evolves, we can add or remove individual services.

From a technical point of view, each microservice is a full mini-application with its own server that communicates with other services through well-defined APIs, sending requests back and forth.

With microservices and distributed systems in general, not only the application has room to grow. Services themselves can scale or be split as they become more complex. This flexibility is great for development teams, as they can work and refactor individual services without redeploying the rest of the application.

Microservices architectures have multiple benefits:

- **Enable horizontal scaling**: When we shard data into multiple partitions, we are able to distribute traffic load across multiple servers. When splitting the application into multiple services distributed across multiple servers, applications also increase their traffic load capabilities.
- **Enable parallel collaboration**: Since each service can be deployed independently, development teams can work in parallel without blocking

each other.

- **Encourage the adoption of better tools**: Potentially, we can build each service using a different tech stack. This flexibility gives us the freedom to choose the tool that best fits the service task: Use a robust enterprise stack like Java and Spring for implementing REST endpoints while at the same time using a Python toolset for Machine Learning services.

Microservices applications are distributed applications. As stated earlier, after keeping data consistent in a distributed application, the next big challenge is to enable integration and communication between each microservice in the application. To fully take advantage of distributed applications' performance improvements, we have to support asynchronous communication between services.

Improving performance using asynchronous communication (queues)

The traditional client-server architecture follows a synchronous communication model: The client makes a request and waits for the server to return a response.

Synchronous communication is also possible when integrating multiple services in a distributed application: Service "A" makes a request to services "B" and "C", and waits for them to process the request and return a response. However, it can be challenging to maintain response times low when using synchronous communication.

Actually, it is impossible to provide synchronous response times under a satisfactory baseline for some scenarios:

- An application that allows users to upload large files will take time to fetch, process, and store a new file.
- An application storing data in a large, distributed database may need time to propagate the changes to all its replicas. Some requests can take minutes in the worst-case scenario where the database is not correctly optimized.
- An application that depends on an external service that allows a limited number of concurrent connections will require currently running operations to complete before accepting new requests.

When requests take long, we cannot force users to sit idle waiting for the operation to complete. For those cases, we use **asynchronous communication**.

Asynchronous communication between services does not follow a traditional request-response flow; instead, it provides a communication channel where services produce and consume requests at different rates. Services do not wait for responses to their requests, as they may take a long time to receive a response.

If we pay attention, we can see this communication channel follows a *publish-subscribe* (Pub-sub) architecture. Services can subscribe to a communication channel; then, other services can publish requests to the channel, and any subscribed services will be able to consume those requests. If subscribed services need to produce a response, they can send it through a different channel. The key here is that no service has to wait for others to respond.

In its most common implementation, this channel behaves like a queue. More precisely, this channel is an **application queue**.

An application queue is a transient storage service that sits in between two or more services trying to communicate with each other, as shown in *Figure 10.10*:

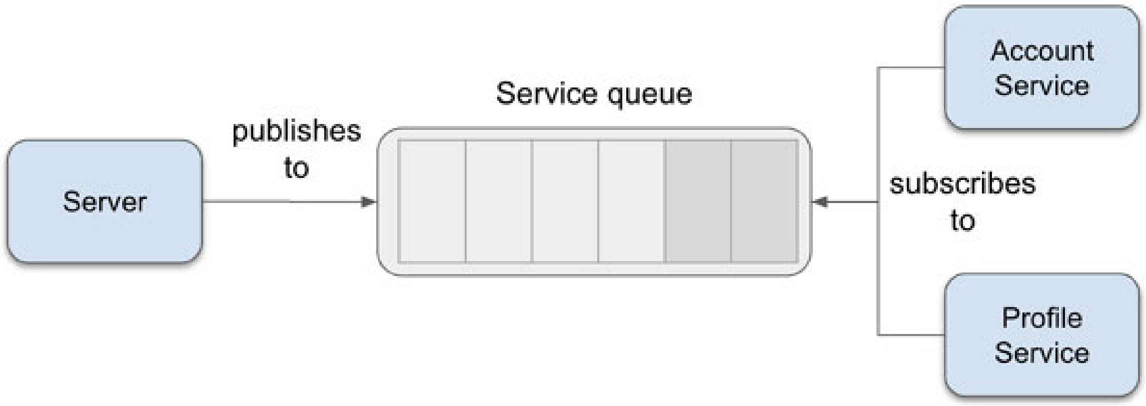


Figure 10.10: *The service queue allow multiple services to communicate*

The storage provided by the queue is what enables asynchronous communication: Services can send a request and, once it is successfully added to the queue, assume that it will be delivered to the adequate services.

Asynchronous services unblock client requests. When a client sends a request, the API server may send a response to the client as soon as it confirms that the request was successfully put in the queue. The response may indicate that the request is being processed and that the user will be notified as soon as a response is available.

Sending an "*in-progress*" message back to the client may not seem the best user experience, but it beats forcing the user to wait for a long time.

Once the subscribed services complete processing the request, they may publish a new request to a separate queue. Then, a service listening to this second queue can notify the user that the request is complete: For instance, a server can listen to a channel where a service publishes responses and then reply a response back to the client through WebSockets or push notifications.

Service queues allow the seamless integration of multiple services into the same application flow. We can create new services to consume an existing channel where other services are already listening.

For instance, we can take an existing service queue and create an application monitoring service that can subscribe to the queue and log relevant information. Or we can integrate an external system to consume our application's information in real-time; all without updating the service that produces the information.

Queues also provide robustness to the application. If any of the subscribed services are down, the queue will store the request until the service recovers and processes it.

In a way, we are cheating to achieve better-perceived performance: We are not completing the request any faster than in a synchronous request-response architecture. Nevertheless, we give almost immediate feedback to the user for a long operation and immediate feedback always makes our users happy.

Some popular examples of service queues are **Kafka** and **RabbitMQ**. These products are used by many high-performant applications to decouple and integrate application services.

Improving the performance using asynchronous programming

Let us take a step back from the high-level view of servers and microservices and back into the low-level view of code.

Some code functions take time to be executed. Functions that make requests to external systems or queries to databases take longer than other functions because they depend on added overhead due to network latency and the availability of external resources.

Since distributed and microservices-based applications split functionality into multiple services, it is common to make multiple requests to external services to fulfill one single user request. This can be a performance problem for a typical sequential code, where we do one operation at a time, as shown in the following example:

```
// Total request time: ~3 seconds
function getOrdersPage(response, userId) {

// each of the following operations is blocking:
let profile = userService.fetchUserProfile(userId); // 1 second
let orders = orderService.fetchUserOrders(userId); // 1 second
let account = accountService.fetchAccount(userId); // 1 second

response.json({profile, orders, account}); // returns a JSON
}
```

In the preceding example, to fetch all the data needed for the application's **`orders`** page, we need to retrieve the complete user profile, the user orders, and account details (payment-related information). The requested data is stored in separate databases fronted by API services.

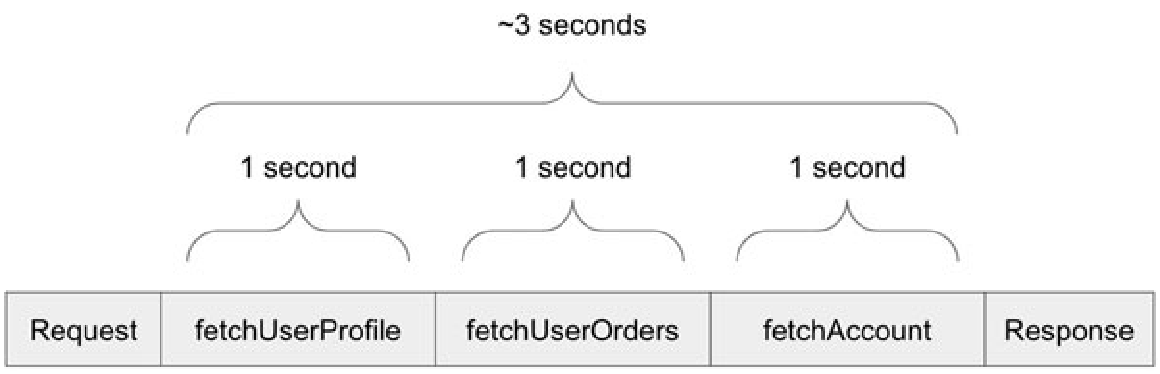


Figure 10.11: *Sequential requests are slow because each request has to wait for the previous to complete*

Figure 10.11 provides a visual representation of the overall execution. When adding up the times to execute each request sequentially, the total response time approaches 3 seconds.

We can see in this example that each call to an external service does not depend on the previous call: We can call each service in any order and still get the same results. Examples like this are perfect candidates for *asynchronous function calls*. Instead of calling one service after the other, we can call them all simultaneously.

Promises and futures

Most modern programming languages have tools to execute code asynchronously. While the most common example of async code is multi-threading, we have other options at a higher level of abstraction.

For instance, Java has the concept of *Futures*, and JavaScript has *Promises*. All these tools share the same underlying principle: they are abstract representations of the result of asynchronous operations.

We can rewrite the previous JavaScript example using Promises as follows:

```
// Total request time: ~1 second
function getOrdersPage(userId) {
let profilePromise = new Promise( // 1 second
    (resolve) =>
        resolve(userService.fetchUserProfile(userId));
);
let ordersPromise = new Promise( // 1 second
    (resolve) =>
        resolve(orderService.fetchUserOrders(userId));
);
let accountPromise = new Promise( // 1 second
    (resolve) =>
        accountService.fetchAccount(userId));
);

Promise.all([
    profilePromise,
    ordersPromise,
    accountPromise
])
.then(([profile, orders, account]) => {
    // once all service calls complete, return a JSON
    response.json({profile, orders, account});
});
}
```

The function passed to each new promise is executed asynchronously. The execution will not pause when it reaches **`fetchUserProfile`** or **`fetchAccount`**. Only after all requests finish, the server will return a single response with the requested data.

Figure 10.12 provides a visual representation of the overall execution. Since we execute all service calls asynchronously, the approximate response time will be as long as *the slowest request*, which is around one second for this example. Using promises, we have achieved a 3x improvement!

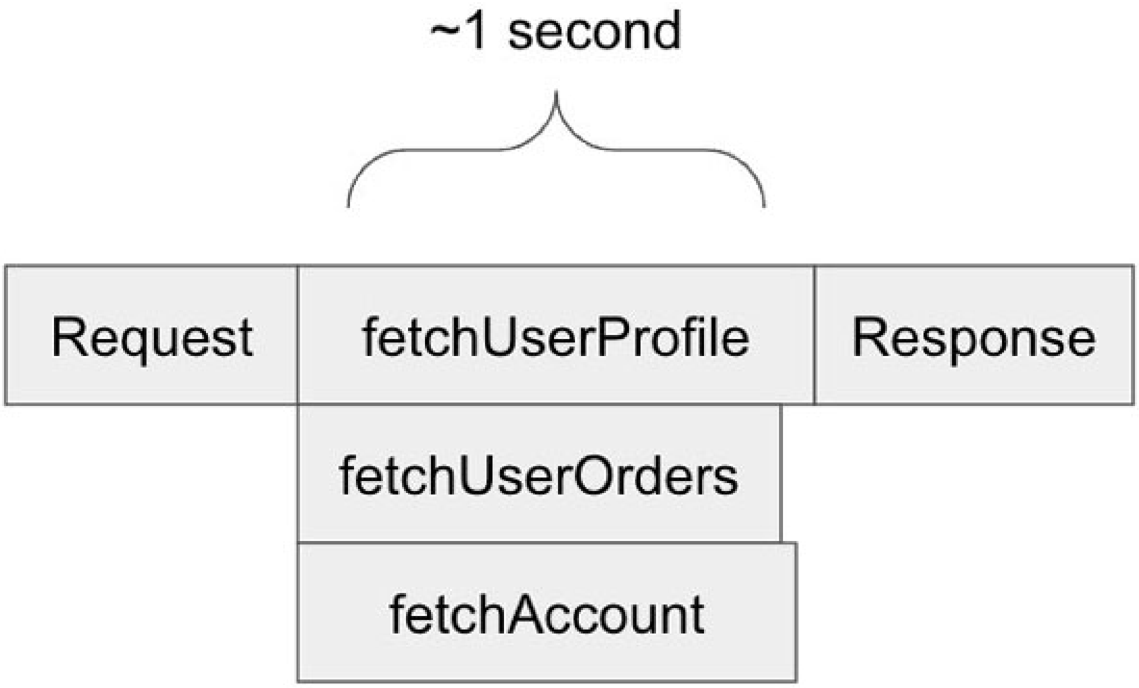


Figure 10.12: *Using parallel requests, the total response time equals the response time of the longest sub-request*

We can use asynchronous code execution as a regular practice, not just when we find performance problems. This technique is widely used in projects where service calls can be parallelized.

What if one of the service calls had a dependency on another? In an asynchronous paradigm, we can chain async calls:

```
// Total time: 2 seconds
let profileAndOrdersPromise = new Promise(
    (resolve) => resolve(
        userService.fetchUserProfile(userId)
    )
)
.then((userProfile) => new Promise(
    (resolve) => resolve(
        orderService.fetchUserOrders(userProfile)
    )
);
);
```

While this example may seem more complex and verbose than executing the sequential code (and it has the same performance), it showcases that asynchronous functions have the flexibility to operate both sequentially and asynchronously. JavaScript even has some syntactic sugar with *async-await* to make it look like sequential code calls, while still keeping its async nature:

```
// Total time: 2 seconds

// userProfile is not a promise, but the actual value
let userProfile = await new Promise(
    (resolve) => resolve(
        userService.fetchUserProfile(userId)
    )
);

```

```
let userOrders = await new Promise(
    (resolve) => resolve(
        orderService.fetchUserOrders(userProfile)
    )
);
);
```

The **`await`** operator indicates to the application that it should pause the execution until the async operation completes and returns its value.

Request parallelization can also be achieved in languages like Java. The following example uses an implementation of the Java Future interface called **`CompletableFuture`**:

```
CompletableFuture<UserProfile>      completableFuture      =
CompletableFuture
  .supplyAsync(() ->
     userService.fetchUserProfile(userId)
  );
```

It is common practice to return Promises or Futures from functions that may take a long time to complete. These utilities allow developers to choose whether they need to execute these operations asynchronously or sequentially, while synchronous function calls always force developers to write sequential code.

As we can see, asynchronous programming has more tools beyond multi-threading functions. This paradigm requires us to shift the mindset, as asynchronous code is harder to debug and think about. However, the performance benefits it brings are well worth it for us to make an effort to design applications that take full advantage of asynchronous communication.

The key idea behind all these practices is that we should build applications to do as much work in parallel as possible, while keeping data consistent.

Conclusion

We cannot improve an application's performance if we don't measure first. Implementing premature optimizations often leads to excessive complexity and multiple bugs.

We cannot measure or represent application performance with a single number. Each part of an application can perform differently, and the response time for each needs to be correctly measured and monitored to find improvement opportunities. Instrumentation code is key to collecting response time metrics in both synthetic tests and real-user contexts.

Once we get a good idea of the typical user experience for different segments or percentiles of our users, we can begin establishing response time baselines. These percentiles are a tangible way to measuring user experience for the variety of users our application serves.

User research tells us that users tend to give up and abandon our application after one second of waiting for a response. This knowledge can guide our performance improvement actions and baseline goals for which we can aim our efforts.

Caching is an effective way of reducing response times in an application. Caches keep data in fast storage that we can leverage to fulfill user requests, instead of always querying data sources that can be potentially slow.

In broader terms, we can evolve our application to use a distributed architecture to cope with increased traffic loads. We balance the application load by increasing the number of servers that can serve responses to user requests.

Distributed applications rely on two concepts to increase performance: Data replication and data partition. Our job as software developers is to design applications that can take advantage of these two concepts while keeping data consistent. This is not a small challenge, but fortunately, proven recipes and strategies exist that we can leverage while designing a distributed system.

Microservices-based applications rely on breaking an application into multiple services that we can build, deploy, update, and scale at an independent pace. While distributing data among different servers help us better cope with increasing traffic demands, microservices help us create robust applications that can grow to maintain and improve performance as needed.

The base for any distributed application is asynchronous communication and integration. Asynchronous services are robust and resilient during failures; and asynchronous coding allows our applications to fully take advantage of the distributed architectures we put in place.

In the end, performance is all about user experience. We use all these tools, strategies, and architectures to provide users with a great experience while using our application. There is no use in building highly complex applications if the user experience does not benefit from them.

This chapter is only an introduction to the world of highly-performant distributed applications. It should be used as a starting point for readers to start their journey into understanding advanced topics.

Questions

- How do we measure performance?
- What does the term “p90” mean?
- What is “eventual consistency” and when should it be used?
- What are examples of distributed systems? Remember that even things like cellular networks are technically distributed systems.
- What is the difference between replication and sharding?

References

- NodeJS Performance API: **https://nodejs.org/api/perf_hooks.html**
- Memoization: **https://www.geeksforgeeks.org/memoization-1d-2d-and-3d/**
- Cache patterns (using AWS ElastiCache): **https://docs.aws.amazon.com/AmazonElastiCache/latest/mem-ug/Strategies.html**
- Kafka (service queue): **https://kafka.apache.org/**
- Java Jedis: **https://github.com/redis/jedis**
- “Leader and followers”: **https://martinfowler.com/articles/patterns-of-distributed-systems/leader-follower.html**
- Microservices (in-depth definition): **https://martinfowler.com/articles/microservices.html**
- Replication and sharding in AWS: **https://www.awsthinkbox.com/blog/replication-and-sharding**
- “Balancing Strong and Eventual Consistency with Datastore” for GCP: **https://cloud.google.com/datastore/docs/articles/balancing-strong-and-eventual-consistency-with-google-cloud-datastore**

- JavaScript Promises: **https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise**
- JavaScript async-await: **https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Async_await**
- Java CompletableFuture: **https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html**

CHAPTER 11
Designing a System

There is not a single recipe for building an app. The architecture of a system depends on multiple elements: User requirements, the project's budget, the expected user growth, and so on. In this chapter, we will learn how to effectively convert our user requirements into a technical design composed of all the modules we have seen in the previous chapters. This system design is the blueprint that guides things like what languages, tools, frameworks, or services we need to choose to build our application.

Structure

At this point in the book, we have broadly described the essential components used to build back-end applications. We have taken each aspect of a modern system design, and layer by layer, we have explored the various tools available to tackle multiple software development challenges.

It is time to take the tools we have discussed, put them in order, and describe the system design process. In this chapter, we will discuss each step involved in the system design:

- The system design process
- Defining and clarifying requirements
- Defining the system's interface.
- Defining data models
- Calculating the system scale and size.
- Creating high-level and low-level designs
- Identifying failure points

Objectives

The goal of this chapter is to connect all the knowledge we have acquired in the previous chapters of this book.

By the end of this chapter, we should have a detailed picture of how all the components, architectures, and methodologies we discussed so far interact with each other to produce an actionable system design.

The system design process

The largest software companies in the world have a step in the interview process for evaluating the candidate's skills in designing systems; increasingly, smaller companies too have adopted this process. This module is especially challenging for inexperienced candidates because the system design process is so vast and involves so many variables that it seems almost impossible to effectively design a large application in only one hour.

One thing that makes system design interviews so confusing is that building an application is an iterative process. Potentially, these iterations have no end, as user requirements will keep evolving and new business opportunities emerge.

System design interviews try to assess a developer's capability to complete the first iteration of a system design, along with the candidate's attention to detail and the candidate's understanding of the multiple trade-offs that choosing a specific architecture or tool brings.

The system design process—that first iteration—involves multiple ordered steps. The goal is for the system designer to understand the system's objectives and create an application blueprint that will cover these objectives as closely as possible under the given constraints (like a limited budget, technical limitations, and compliance with laws and regulations). This blueprint will unavoidably evolve once we start building the application on top of it.

Example: The Pizza Place (at scale)

Across almost every chapter of this book, we have used the Pizza Place application to illustrate each element of back-end development. Now, it is time to grow this example to demonstrate the design of a more complex system.

Going back to the example business case: It’s been three years since we built PizzaPlaceApp v1.0. The application has worked correctly using a simple infrastructure consisting of one server for the application and a second server for the database, as illustrated in *Figure 11.1*:

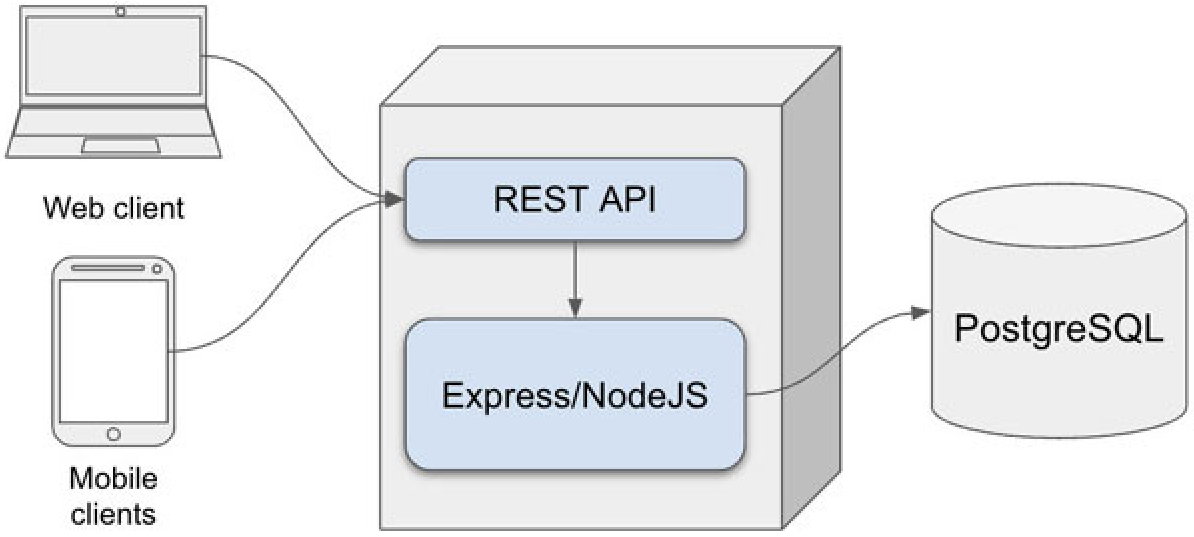


Figure 11.1: *Steps in the system design process*

The Pizza Place has grown beyond anyone’s expectations. New restaurant locations have been opened across the country, serving thousands of people daily.

To cope with the fast expansion, each new location chose to install their own copy of the application: each bought a couple of servers and created a fresh install. While this approach quickly enabled each restaurant to take new orders, there have been some downsides: people who already have an account in one of the location’s application need to create a new account if they want to order in a different location.

Since users need to have one account per location, their rewards are similarly fragmented. There is no easy way for the Pizza Place owners to access order information for all locations. They rely on the managers from each location to generate reports manually and send them daily through emails.

Given the recent business success (and their plans to expand internationally), the owners of Pizza Place reach out to us to re-design their application to centralize the orders from all locations with fewer budget limitations this time. We will use this re-design effort as the primary example to discuss the system design process.

Defining and clarifying requirements

In *Chapter 1, Building Multi-user Apps*, we discussed the process of collecting requirements from our users. Having a clear set of requirements will guide each decision we make during the system design process.

In this step (*Figure 11.2*), we will ensure all requirements and assumptions are clear and approved by all stakeholders.

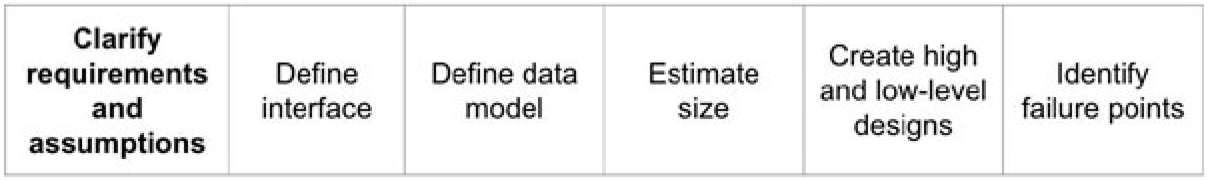

Clarify requirements and assumptions
Define interface
Define data model
Estimate size
Create high and low-level designs
Identify failure points

Figure 11.2: *Step 1*

Requirements allow us to set constraints on the system size and features. Since designing an application may involve a wide variety of tasks, these constraints help us filter out architectures, components, and other patterns that may not be suitable for our use case. If we were to design a house, we need to define constraints to how big it needs to be. These initial characteristics will, in turn, help us determine other, more concrete details.

First, we list all known functional requirements, as these will implicitly (and sometimes explicitly) provide a list of constraints. For Pizza Place v2, we have the following requirements and use cases:

- Users should be able to create one single account in the application and use it to make orders at any location.
- Considering each location can have different food items on its menu, users should be able to see each location’s menu.
- Business owners should be able to fetch daily reports about orders from a single website. Each report should have order statistics for each location, each region (each region having multiple locations), and summarized for all locations.
- The rewards program should allow users to order from any location and accumulate rewards in a single account.

- A new feature needs to be added: The application should automatically order more ingredients from providers when it calculates that the kitchen is about to run out of them.
- Another new feature: Once the order is delivered, we must store a picture of the order at the doorstep for contact-less delivery. This is for quality assurance: The business owners can audit whether the order was delivered correctly, thus improving the user experience.

Given the business's aggressive expansion plan, we may feel free to define a couple of non-functional requirements: The application should support a high traffic load, as it will serve thousands of users across all locations. The application should be failure resilient. Problems with the application in one location should not affect other locations. This requirement will allow users to move locations when they cannot create an order.

If we pay close attention to any list of requirements, we can find the application's constraints between the lines. Some may be obvious and implicit: The application should allow user access from various clients. Since there is no special requirement regarding clients, we can limit our design to the most common ones: web browsers, Android, and iOS. The application should be distributed across multiple servers to allow the system to operate in the face of single server failures.

Other implicit constraints are less obvious: The system needs to scale quickly. We have historical data telling us that the business is fast gaining customers, which means the application needs to grow in time and meet the demand.

Strong consistency is necessary for the system's storage. This is because we do not want users to be able to make orders that cannot be fulfilled due to a lack of ingredients. Strong consistency storage will increase the confidence that the application will request more ingredients from providers before allowing clients to make a new order.

It is normal to have questions about specific sub-requirements and use cases at this point. Not having any questions at all may indicate that we may be missing something or that we are over-simplifying our design.

Note: During a systems design interview, many candidates encounter requirements that are not clearly defined and incomplete use cases.

This is because many interviewers will intentionally leave information out to assess whether a candidate can work with other people to define unclear requirements.

A weak candidate will make a lot of assumptions (some will be correct, but most will be incorrect) and immediately start describing a system with incomplete information. A good candidate will take time to discuss all unclear requirements until reaching an agreement with the interviewer.

The key things to remember are: Don't make assumptions, clarify every requirement, and confirm assumptions with every stakeholder before moving on from this design step.

Notice that we have not defined specific solutions yet. All we have determined is what needs to be done, not how. This distinction is important because it keeps us impartial to specific architectures or patterns. If we immediately say: "*This application needs to use REST APIs*" or "*we have to use MongoDB*", we may be over-constraining ourselves, even if the solution seems obvious.

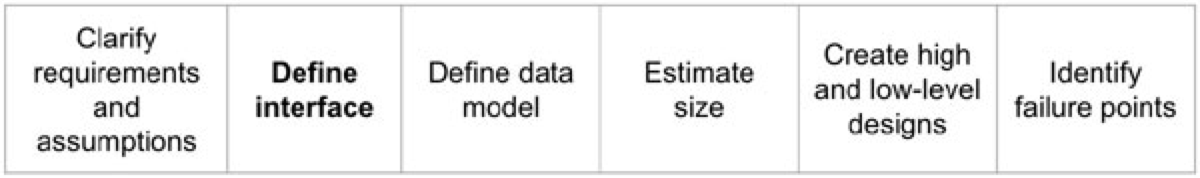
One key thing to remember is that we will use the constraints to reduce the size of the problem. The more constraints we define, the more we can reduce the list of tools available for us to build the system. Choosing frameworks, patterns, and tools will be straightforward if we can reduce the problem.

Once we have reviewed all requirements and put together a list of constraints, we can move to the next step.

Defining the system's interface

Having a defined list of requirements allows us to enumerate the list of operations that the system needs to support. We can think about this step as defining an interface as we did in *Chapter 3, Designing APIs*: A list of operations, parameters, and expected return types.

In this step (*Figure 11.3*), we will define the system operations and user actions in the form of a system interface:



Clarify requirements and assumptions	**Define interface**	Define data model	Estimate size	Create high and low-level designs	Identify failure points

Figure 11.3: *Step 2*

An advantage of defining an interface soon in the design process is that it allows us to create a concrete list of the data we need to complete the interface's actions.

We need to ask the following questions: What actions will the clients and users perform in the system? What data does the application need to fulfill those actions? What data does the action return when the operation succeeds? What data does the action return when the operation fails?

For instance, think of the action of creating an order. The most basic data we need is the order information. However, we need to track who made the order, so we also need to pass something to identify the user.

Also, since the system now supports multiple locations, we need to know the location the user selected to create the order. This parameter can be easy to forget if we have not clearly defined our application's requirements beforehand.

Finally, the action should return a status code to let the clients know whether the order was successfully created or not. Remember that there is a chance the request fails due to technical issues like a server failure or underlying problems with the process, like a location running out of ingredients.

Let us take a look at the interface definition for creating an order:

```
StatusCode create_order(Order, User, Location)
```

Notice that we are not defining a REST API or a Java Interface, as it is too early in the process to know if any of these are the correct tools for this implementation. Maybe GraphQL and Python are better choices. The system design process should remain implementation-details-free for as long as possible to not restrict our creativity and available toolset.

We can repeat this process for each requirement: Querying orders, creating reports, canceling orders, and so on. We will skip them for the sake of brevity, but the process is not different from what we just did: Define the operation, its parameters, and return data.

Defining data models

In this step (*Figure 11.4*), we must define the input and output data attributes for the actions we defined in the systems interface:

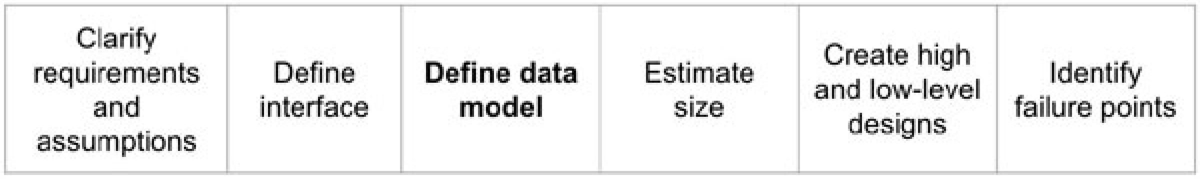

Clarify requirements and assumptions
Define interface
Define data model
Estimate size
Create high and low-level designs
Identify failure points

Figure 11.4: *Step 3*

We can structure the models with any format that clearly defines their contents. For instance, JSON is a good choice as it clearly and concisely describes the relationships between each attribute in the model.

From *Chapter 4, End-to-end Data Management*, we can bring back the example of the data stored in an order model:

```
{
    id: 123,
    items: [
        {
            type: “PIZZA”,
            ingredients: [“CHEESE”],
            quantity: 1,
            size: “XL”,
            price: 12.99
        }
    ]
    total: 12.99
    notes: ‘Extra crispy’
    image: “…”
    date_created: “…”
}
```

Notice that we have added an attribute to store the image we defined in our requirements. What are we storing for that attribute? It could be the file's binary data (in a BLOB), or we can store the file separately and use this attribute to store a reference to the external file storage.

Defining each model in our application will give us insight into what data needs to be stored. This information will inform the next step in the design process.

A vital heuristic here is to avoid defining data not explicitly mentioned in the requirements or by the domain experts. Extra, unnecessary data introduces complexity, extra space, and additional cost that is difficult to remove once we implement the application.

Remember that the model definition will not remain static: We can return and update our models as we need to store more attributes. For instance, if we later find out that we need to keep analytics data about orders, we might later introduce timestamps to store the date and time the record was created. But for now, we will only define data critical for achieving the basic list of requirements we just described.

Calculating the system scale and size

One of our constraints is that the application needs to scale quickly. This constraint involves having an application that can grow as traffic increases to cope with the added load. However, scaling a system can be difficult if we don't define an approximate starting size for the application.

In this step (*Figure 11.5*), we will create an estimation for the number of servers and the amount of storage we need to initially reserve to cope with the expected number of transactions in the system:

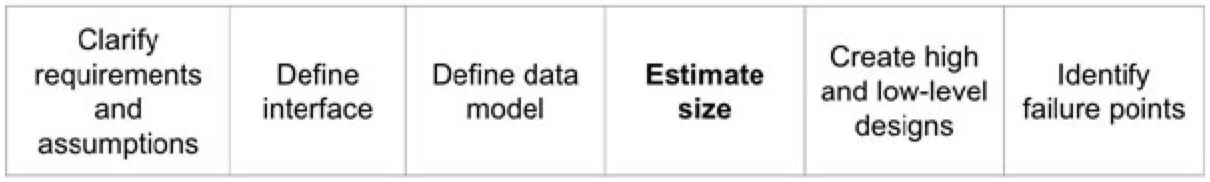

Clarify requirements and assumptions
Define interface
Define data model
Estimate size
Create high and low-level designs
Identify failure points

Figure 11.5: *Step 4*

Calculating an application’s traffic and storage size requirements will reveal constraints around the initial number of servers or the type of databases we need to use for our design.

It isn’t easy to fully define how much traffic an application will get if this is a new system; we might not have concrete data on how many user requests we can expect or how much data we will have to store. At this point, we need to make some informed guesses from the functional requirements.

Let us assume we have no historical information about the existing Pizza Place app. This assumption will force us to demonstrate how to approximate an accurate application size using only our data model, requirements and constraints, and our experience as developers.

Note: These assumptions may be slightly off, and that is fine. We cannot predict the future. However, assumptions should be backed up by as many facts and data as possible; and all stakeholders should agree upon them (other developers, application owners, and infrastructure experts, among others).

If the business owners have historical data about the process we are trying to model, that should inform our design as much as possible. Real data will reduce the number of assumptions we must make while building our system design.

Estimating the storage size

Defining the amount of data storage we will need to host the application requires us to part from fundamental facts. If we build our estimates from actual data, we will be confident that these facts will be as close to reality as possible.

Let’s start with text data. It takes *one byte* of data to store a single character encoded in UTF-8, 2 bytes if we need to use UTF-16 encoding. Unless your application needs to support the whole Unicode character set, it is safe to assume each character will use 1 byte. The average length of English words is between 8 and 9 characters, but languages like German have an average size of 12 characters per word.

These facts allow us to assume that each word we need to store in our application will take, on average, 8 bytes if we are only to support English text. If we want to think in worst-case scenarios, the longest words in English are between 20 and 25 characters, but we will be fine using an average unless our requirements indicate the need to use a wide range of vocabulary.

We can define the same size requirements for numbers: *Integers take 2 or 4 bytes*, depending on the language implementation. *Long-typed variables take 8 bytes*. Since numbers can be contained in single variables, we don't need to calculate average sizes as we did for text. *A timestamp takes 12 bytes* to be stored in most databases. These facts about the most basic data units will provide educated calculations for the next step, so ensure everyone working with you in designing a system agrees on using the same values.

Lastly, for the pictures that need to be stored, we can assume that each *average photograph file has a size of 200kb*. While the image size can vary significantly with the format used to store them, we can enforce this size by processing pictures when they are uploaded to the application.

Now, we define the data entities we plan to store in the system. Let us look again at the **`Order`** entity we described in *Chapter 4, End-to-end Data Management*, but annotated with the data unit sizes we just defined:

```
{
    id: Long (8 bytes)
    items: [
        {
         type: String (8 bytes)
                 ingredients: [String], (8 bytes, times the number of ingredients)
           quantity: Integer, (2 bytes)
           size: String, (8 bytes)
           price: Long (8 bytes)
        }
    ]
    total: Long (8 bytes)
```

```
            notes: String (8 bytes, times the avg. number of
    words in the attribute)
        image: BLOB (200kb) or String (8 bytes), if using a
    string reference
        date_created: Timestamp (12 bytes)
}
```

From this model, we can assume that we will store both "*Orders*" and "*Items*" in the same document. But even if we convert this to a relational model, the size would remain essentially equal.

Notice how the models we defined in the previous step provide the blueprint for the data size requirements. We use well-known basic data type sizes to approximate each attribute size requirement.

At this point, we need to clarify some more constraints: How many items does an order have? How many ingredients does an item have?

After looking at examples of previous orders (or just talking with the domain experts), we all agreed on the following assumptions: Each item has six ingredients on average, making its size 48 bytes. The notes in a regular order can have up to 40 words, so we can assume notes will take 320 bytes to store it. Each order has, on average, five items.

Now, we can reflect on the model in a better way:

```
{
    id: Long (8 bytes)
    items: [ // 5, in average (each 74 bytes)
        {
            type: String (8 bytes)
            ingredients: [String], (48 bytes)
            quantity: Integer, (2 bytes)
            size: String, (8 bytes)
            price: Long (8 bytes)
        }
    ]
    total: Long (8 bytes)
    notes: String (320 bytes)
    image: BLOB (200kb)
    date_created: Timestamp (12 bytes)
}
```

Let us ignore the image attribute to better gauge our storage requirements. After adding up all weights, we have a total of 718 bytes per order just for the data, on average.

This doesn't mean that all orders will be that size or that this will be the max size for a single record. This number will help us measure how much storage we need to reserve to store our records. Some records will be larger than the average; others will be smaller, balancing out the total storage used.

If we need to err on the safe side, we can increase our estimation to 1000 bytes or 1 kb per order. This assumption will give us some space to grow our model slightly if we need to add more attributes in the future. So, in total, we will need 1kb for the order data and 200kb for the picture.

Now, parting from this well-agreed record size, we can calculate the total storage needed for storing all orders in the system. Based on the domain knowledge or historical knowledge, the assumption that we need to define is how many orders we can expect to store in the system.

Estimating QPS

The term **Queries per second** (**QPS**) refers to how many operations, both reads and writes, on average, we expect clients to perform on the system. Estimating QPS can inform multiple assumptions: How much data do we expect the system to generate for a given period. How much cache storage do we need to cope with the number of *read* queries?

Let us assume we currently have 150 thousand clients for all locations. The managers at Pizza Place tell us that, on average, each client makes an order every three days (some users order once a week, while others make one or two orders daily). After some math, we can see that the system would have to store 50 thousand orders daily. Following these numbers we can make the following statement: For a day, we would need 50MB for record data and 9.5GB for storing the pictures. We would need ~18GB of record data and 3.4Pb of storage space for pictures to keep one year of data.

The Pizza Place managers want to keep orders around for three years, resulting in 54GB for the records and 10.1Pb for images alone. The total estimated storage needed for storing three years of order data is around 10.2Pb.

After some discussions with the domain experts, we find that 60% of users tend to order the same food items they have ordered. However, store managers also query existing orders, increasing the number of *read* operations. So, for estimating the number of QPS, it is safe to assume the number of *read* operations will be approximately 80% of the number for *write* operations: Around 40K *read* operations.

This estimation of both *write* and *read* QPS provides us with some information: The application performs more writes than reads, so we should optimize the system for write operations.

Most of the storage needs come from storing the images as binary files. A single database instance can store all the data. For example, PostgreSQL has a theoretical limit of 16Tb per table—a limit imposed by the underlying file system—which is more than enough to store all orders.

While, in theory, we can store all the data in a single vertically scaled server, other constraints tell us that it is not practical to use one instance:

- A single server would be a single point of failure. If the server goes down, so does the whole system.
- The query performance degrades as the data increases. The more data a single server needs to scan to retrieve results, the longer it will take for a query to complete.
- The server throughput is limited. A server can only read a limited quantity of data per second.

Estimating the storage throughput

Throughput is one of the principal roadblocks when using a limited number of servers. Assuming a single database server can read 100MB per second (a pretty decent throughput for a server) and our record size of 1Kb, we would be able to read 105,000 records per second.

Creating database indexes can help with the query performance; indexes allow the database to scan through fewer records before finding the queried result, but the limit of 100k records per second will reduce the overall system performance when the total size of the database is in the range of petabytes.

Note: If we were to keep this data in the "cold storage", which means data that will be stored without being modified for a long time, a single database would be more than enough to hold our data.

However, data in a production database is not at rest: It will grow and change, especially for a write-intensive application. Distributing the load of a database through replication and sharding is an effective way to guarantee that queries will perform at acceptable rates.

Another way to improve the performance under throughput limitations is to store binary files in a dedicated file storage system. Instead of storing the file in the same record as the order information, we will keep a reference to the file. Since we will not be querying orders by their file attachments, we don't need to keep that information in the same database. It is only when the application needs to render the order information that we need to retrieve the file.

Calculating the cache size

The Pareto principle states that for many outcomes, approximately 80% of consequences come from 20% of causes. This distribution is a good starting point for many aspects of software development. For instance, 80% of issues come from 20% of the code. It is also an excellent heuristic to calculate the memory needed for a caching layer.

As discussed in *Chapter 10, Creating High-performance Apps*, caching data is an excellent way to improve the performance of an application. If we use the Pareto principle to define a starting point for our cache, we can see that we will need approximately 11GB of memory to store the cache data for 20% of our total 54GB of order data. Again, that is pretty achievable with a single server, as a typical server can hold between 32GB and 64GB of RAM.

If we use a single dedicated server to host a service like Redis, we would be able to hold 20% of the total data in cache. However, as the application grows, we want to be able to increase its caching capabilities. Fortunately, products like Redis have predefined ways to scale into multiple servers, and hosted services from platforms like AWS, Azure, or GCP will handle scaling for us (at an appropriate cost).

Note: Not all RAM will be available for caching efforts, as the server's OS and its internal processes will consume memory.

Notice that the calculated size for the cache is just a starting point: We should increase or lower it based on business-specific information, budget, and historical data. Also, once we build the application, we should be free to update the cache size as we see fit.

The constraints we defined when we estimated the QPS for the system inform other decisions about the cache: Given that we have defined our application as write-intensive and requiring hard consistency guarantees, we could configure the cache to be synchronously write-through. This will allow users to query past orders efficiently. A good eviction policy would be **least-recently used** (**LRU**), as users will tend to look at the latest orders they created when making a new order.

When to stop estimating the size

Using data models to estimate the required storage size is a tricky process. The more time we put into the task, the more accurate our approximations will be, but it is a task with diminishing returns.

As we put more time and effort into this estimation, there is a higher chance that one of our assumptions will be erroneous. Then, all calculations done on this false assumption will also be wrong. This is why, ironically, putting too much effort may lead to a less accurate design. We should aim for a middle point.

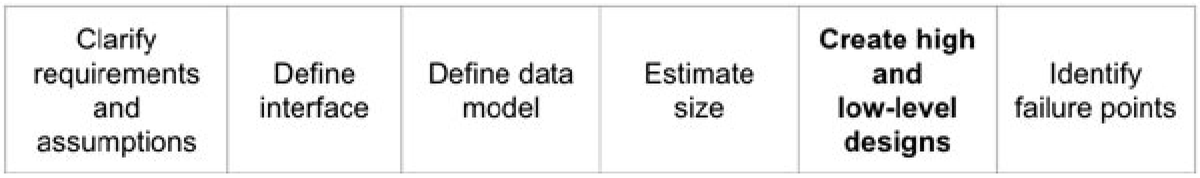
In the end, this is just an estimate that most of the time will be different from the actual space the system will use. Only experience will help you know when to stop estimating the system's storage needs.

Creating high-level and low-level designs

At this point of our design process, we have focused on defining our system's behavior, the data we need to achieve its goals and the overall storage size. Most of our analysis has been abstract: While we have defined the capabilities of specific services or providers, we haven't defined specific technology implementations.

This abstraction is on purpose: We must wait to have all facts in order before deciding what tech stack is better suited for our efforts. Before we pick a stack, let us do one final, high-level design to define how each part of the application should interact with each other.

In this step (*Figure 11.6*), we will create both high-level and low-level designs for our system:



Clarify requirements and assumptions	Define interface	Define data model	Estimate size	**Create high and low-level designs**	Identify failure points

Figure 11.6: *Step 5*

Defining a high-level design

In this step, we will create an end-to-end design of each component needed in the system to fulfill our requirements.

Imagine that we are designing the road infrastructure for a new city. We need to consider all the critical services (like hospitals) the people in that city would need to access. Then, we would put roads in place so people can efficiently reach each service. If two roads need to cross, we know we need to put a light or a roundabout to enable people to navigate safely and quickly through them.

Applying the city analogy to systems design, we need to define all the critical application services client requests will need to reach to achieve each use case. The roads will be the communication and integration protocols and services we will use to allow the requests to flow through the system efficiently.

Let us start by listing the use cases again. The following list only includes the most critical use cases, but you will need to define each use case in an actual design process:

- "As a user, I want to create an order at a given location".
- "As a user/manager, I want to read the list of orders, using a variety of search criteria (orders created by a user or orders created at a given location)".
- "As a manager, I want to query and summarize all order data (for one or multiple locations), so I can create reports".

Defining the use case flow

We will enumerate the steps required to perform each use case. We will only work on the first use case ("*As a user, I want to create an order at a given location.*") for the sake of brevity, but the process for determining each case's flow should be the same.

1. The user accesses the application through the client (mobile or web).
2. This client will then send the new order information to a server.
3. The server will perform multiple separate tasks:
   a. It will format and validate the order information. If it cannot fulfill the order (because there are no ingredients or one of the services is down and no remediation is available), it will return an error message to the client.
   b. It will store the order data in the database, passing through the cache service.
   c. It will upload the file attachment to a file storage service.
   d. It will update the user's loyalty and rewards data in the database.

As seen earlier, any of these tasks can take a long time to complete, or they can fail. To improve the perceived performance, we will perform all these tasks asynchronously. Once the server parses and validates the order data, it will return the client a success message. The client should interpret this message as "*we have received the order, and it is being processed. We will notify you once the order is complete.*"

Now, we can put together a very rough high-level design. We will identify each component and service used in the use case flow, list them, and connect them. The result is visually displayed in *Figure 11.7*:

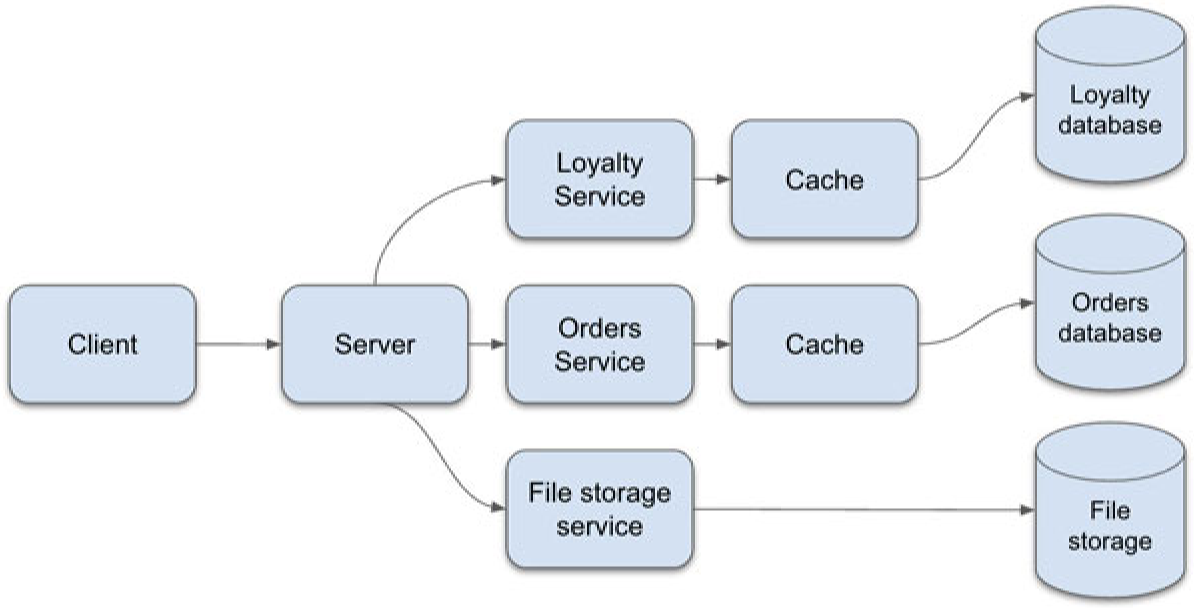


Figure 11.7: *High-level design*

While it may seem like we are missing too much information, the goal of a high-level design is the same as creating a high-level map of a city: We don't need to worry about the materials used to create the roads or the different types of hospitals. We care about the big picture, allowing us to deep-dive into each element later.

Finding gaps in assumptions

At this point, we want to add some conflict to our example. Imagine we completed or high-level design but then we figure out we failed to ask a question: When is an order "*complete*"? Is it when the system successfully has stored all data? Can a location reject an order for any unforeseen reason (like a cook suddenly falling ill)?

In an ideal world, we would have dedicated enough time during the first step of the design process to clarify this use case. In the real world, it is not uncommon to catch these gaps later on, and we must know what to do when that happens.

After figuring out there is a gap in our design, we talk again with the domain experts. We find out that orders have states: They can be "*received*", "*rejected*", "*in process*", "*out for delivery*", and "*fulfilled*".

With this new information, we need to update our "*Orders*" model to include a "*status*" attribute. The value for this attribute will be an ordinal number or an "*enum*" that reflects each of the possible states in the order.

We will also need to create a new "Order Updates" model containing a record per each status change an order goes through. This model will have the following attributes:

- A reference to an order.
- An attribute for storing the time and date the order was updated.
- An attribute for the new order status.
- An optional attribute for adding notes (so the location can add a reason for canceling the order or if there will be a delivery delay).

These model changes force us to re-evaluate our assumptions on storage requirements. Fortunately, the domain experts tell us that we will only keep the "*Order Updates*" data until the order is fulfilled, after which the data will be discarded. The only exception is that we need to keep status updates for canceled orders to evaluate why we couldn't fulfill the order. Since we gave ourselves some margin while defining the storage requirements, we can get away without changing those assumptions for this case. Keep in mind, though, that most of the time we will not be as lucky and we will have to recalculate our estimations.

Some changes are required for our interface, though. We need to define a new operation for clients to check on an order status:

```
OrderStatus check_order_status(Order)
```

After re-evaluating all our assumptions, we can update our high-level design. The updated design can be seen in *Figure 11.8*:

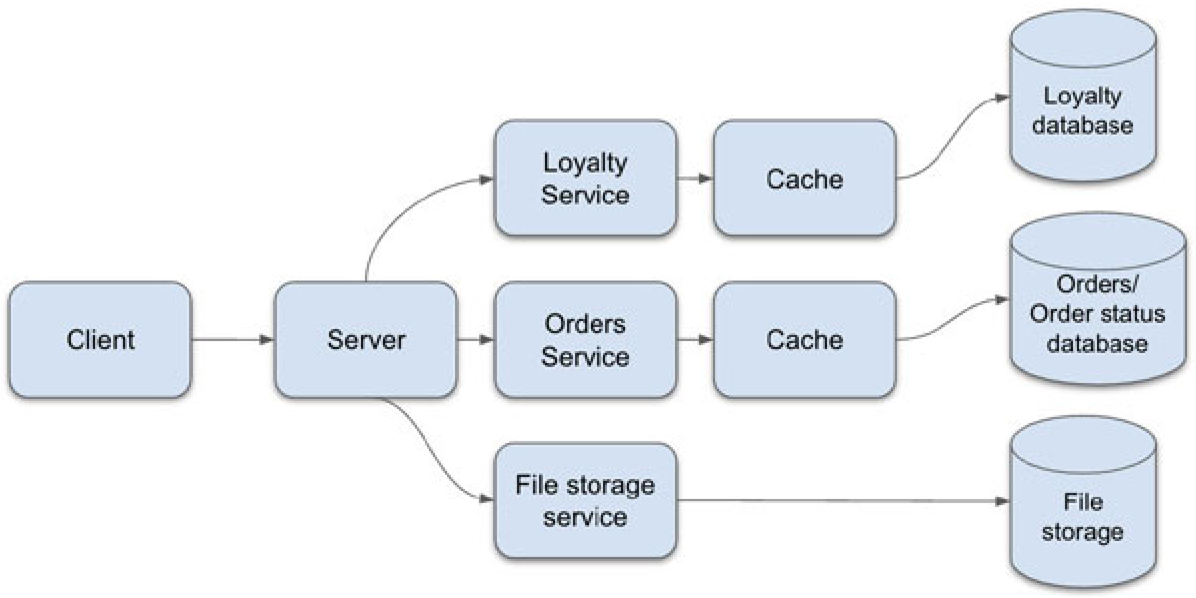


Figure 11.8: *Step 5*

In this example, we have been lucky that our omission did not significantly require re-doing all the design work we have done so far. We have purposely introduced this bump on the road to make a few points:

- Each part of the design process is prone to change.
- It is OK to find gaps, but it is critical to address them on time. The longer we take to catch gaps, the more difficult and expensive it will be to correct them.

Defining a low-level design

In this step, we will move away from the high level of abstraction at which we have been operating. We must stop and validate with each stakeholder that all the assumptions we have made up to this point are correct. Everyone must agree that we are going in the right direction because all our work so far will enable all our infrastructure and development decisions from here on.

In this step, we will revisit the use case flow, but this time we will deep dive into each component.

Designing the client

From our non-functional requirements, we identified that we need to support mobile clients (Android and iOS) and web clients. We can split our high-level entity “*Clients*” into web and mobile components. *Figure 11.9* shows the low-level design for the client component:

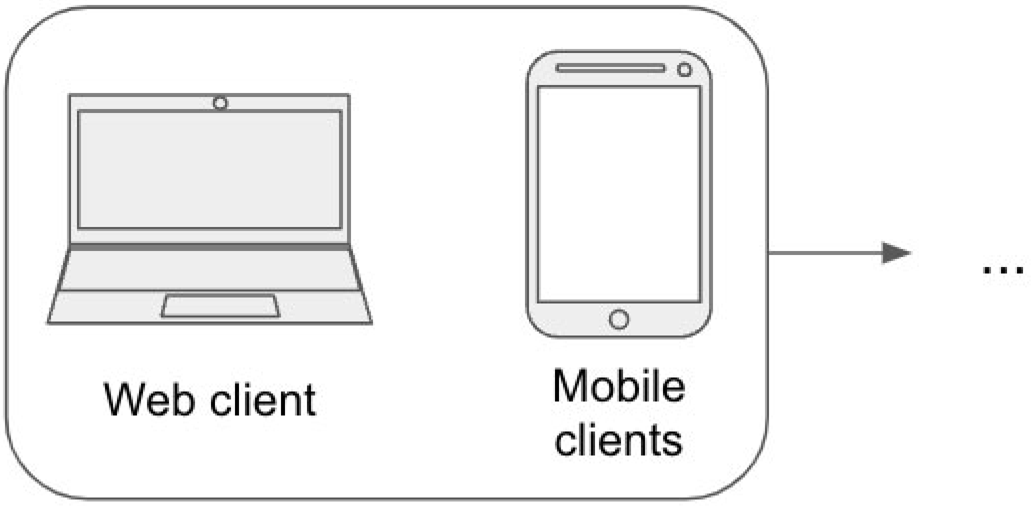


Figure 11.9: *Low-level design for the client*

Designing the web server

Also, from the non-functional requirements, we see that we need to implement a distributed system. To enable this architecture, we will need multiple servers.

- How many servers will we need? We can do a throughput analysis similar to the one we did for our database server. Working with reliability experts, we should define the following points:
- How many QPS can a single web server support? This estimate should consider the latency for downstream services, like database and external services.
- How many locations are there?
- How many QPS can we expect from each location? Remember that some locations can have a significantly larger number of users than others, and those might require extra servers.

The equation we aim to solve is as follows:

```
Number of servers per location = Total QPS per location / QPS
per server
```

Again, this is just a rough estimation. The number of QPS is not static. It varies from low traffic times (during the night) to high traffic times (peak hours, special sales). The estimated number of QPS and the number of servers should also consider the following:

- How many servers do we need to serve average QPS?
- How many servers do we need to serve high-traffic QPS?

Ideally, we would always have as many servers as needed to cope with high traffic, but that number may be cost-prohibitive.

A good strategy is to have enough servers to operate at around 60-70% of server capacity during the average traffic load. This distribution gives us some margin for small spikes in traffic without scaling. Then, assuming the time it takes to scale is fast enough to not result in downtime, we can add more servers once usage for the existing servers reaches a certain threshold (something like 80-90%).

The thresholds we just used are just a baseline. It all comes down to balancing three aspects:

- How expensive it is to have enough servers to keep usage levels at acceptable rates during average traffic.
- How expensive it is to add more servers when traffic increases.
- How expensive it is to have downtime if the gap between the current number of servers and the number of servers needed during high traffic is large enough that we cannot scale fast enough.

Some cloud providers provide elastic computing: The cloud services automatically add more servers as traffic increases, removing the need for development teams to scale manually. However, we should understand what is happening under the hood to gasp the cost implications of operating a fully scalable system.

We will not define specific numbers for our example, as these calculations are better suited for a book about site reliability. This section lists what we need to consider when defining the server infrastructure for a system.

For our example, we will assume we have calculated that we will need "N" servers, which need to be fronted by one or more load balancers. *Figure 11.10* shows this simple low-level design:

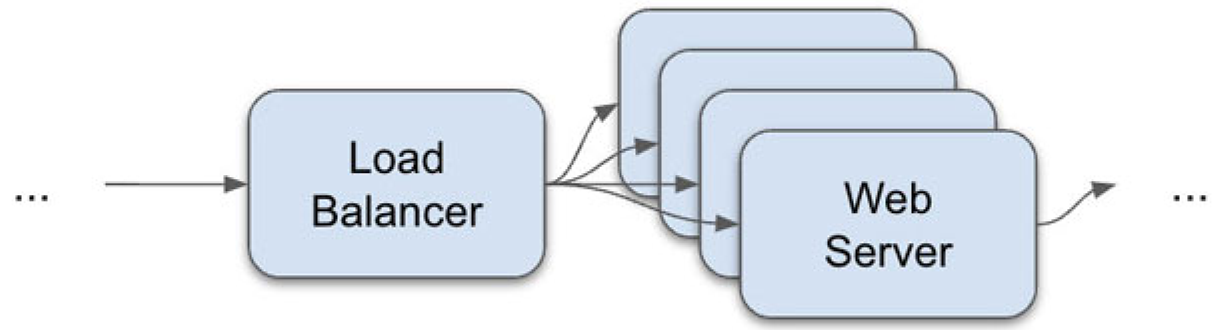


Figure 11.10: *Step 5*

Designing the service layer

Notice we have tacitly included "*services*" in front of each database and cache layer. We do so to isolate the business logic needed to process and store orders, loyalty points, and file attachments from the primary server process that takes care of stitching everything together.

As discussed in *Chapters 3, Designing APIs*, and *Chapter 10, Creating High-performance Apps*, this service separation can be logic -functions or classes hosted in the same application- or infrastructure-based, like a microservices architecture. For our case, we will go fully distributed, and we will create microservices for "*Orders Service*", "*Loyalty Service*", and "*File Management Service*".

Note: Our example is formulated to explore a distributed system example. However, the high-level design would work for a single-server infrastructure too. That is one of the advantages of keeping high-level designs abstract and implementation free.

We will host each of these microservices on separate deployment units. We will deploy each to a Docker container hosted in a Kubernetes cluster. This design choice will allow our system to scale these downstream services quickly if traffic requires it. *Figure 11.11* shows the low-level design for these Kubernetes-managed services:

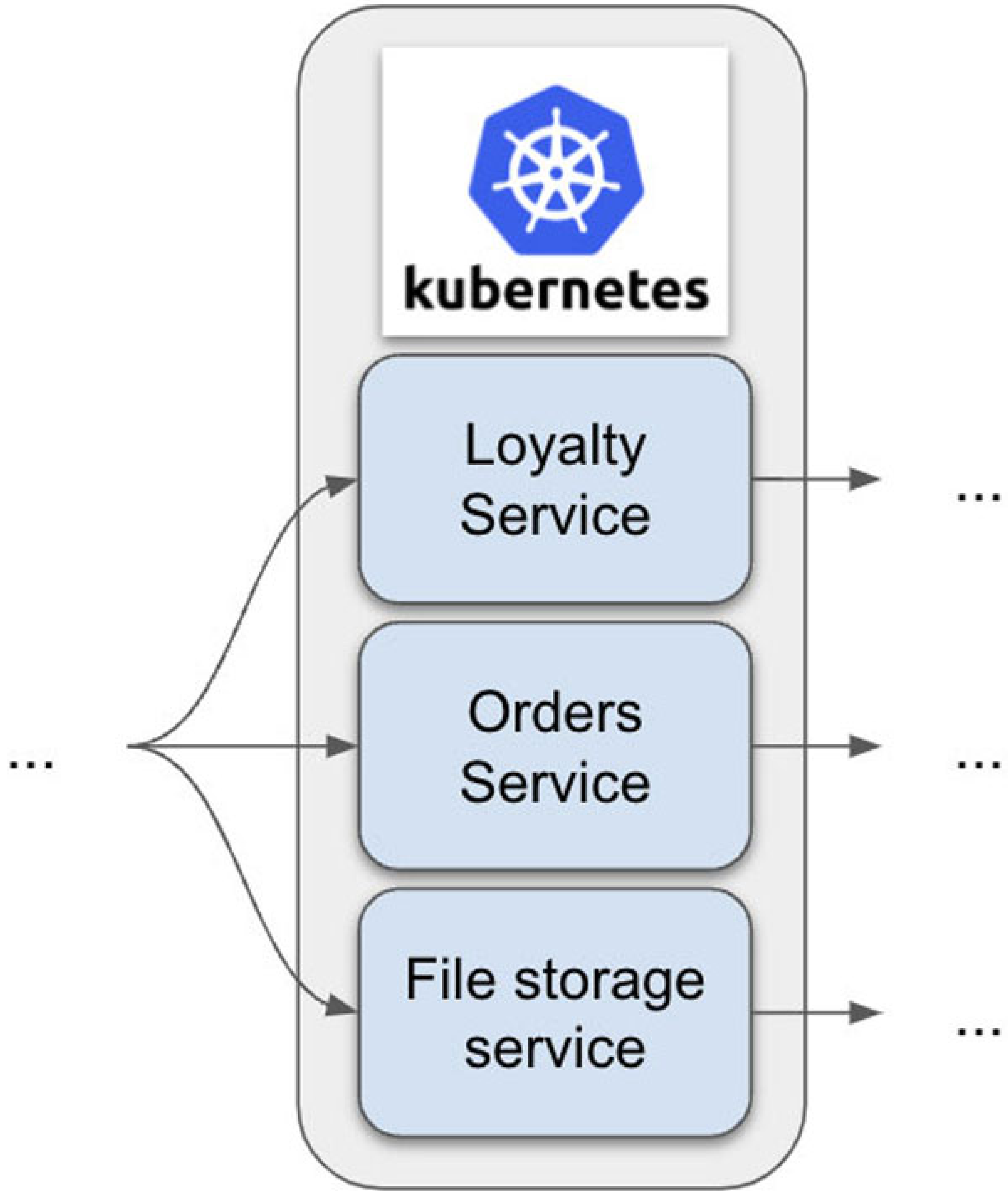


Figure 11.11: *Step 5*

What tech stack should we use to build these services? The services we have defined have no strong constraints for a given stack. If one of these services were to perform more specialized tasks, we might have strong reasons for choosing a specific stack (like choosing Python for training

Machine Learning models, a very popular choice nowadays). However, that is not our case.

Without strong constraints, choosing a tech stack depends on multiple factors:

- The skills available in the development team: It makes no sense to write your services in C# if all the developers in the team are more familiar with Java. It would be more expensive to train the developers or hire external resources to use a technology they are not familiar with.
- The available infrastructure: If our current infrastructure already works with a tech stack, there are a few reasons to choose a stack that will require changing the available resources.

As discussed earlier, one of the advantages of a microservices architecture is that it gives us the flexibility in using the adequate tech stack for the task.

The only exception to the microservices infrastructure will be the file management service. After discussing with the development team, we found that cloud-hosted file storage (like AWS S3 & AWS Cloudfront) is cheaper and easier to manage than hosting our own solution.

Designing the database layer

We will look at our data models for this step in the design process and choose the correct database type.

The data models we have defined lean heavily to a document-based structure. However, some relationships between the data pull us towards a relational model:

- The higher emphasis the new requirements put on ingredients may lead us to split our "*Order*" model into multiple entities: one for orders and one for ingredients.
- The "*Order Status*" is related to the orders in an N-to-1 relationship.

However, we find that both "*Ingredients*" and "*Order Status*" have high locality with Orders, as we often query them together.

The most substantial reason to choose a relational database is the relationship between "*Orders*" and "*Ingredients*". Since the system will need to keep track of ingredients to fulfill the orders, its model will require including attributes that may not be local to "*Order*".

A good trade-off would be the following: Use a document database and do some extra work at the service level to create a relation between "*Order*" and "*Ingredient*".

For this approach, we will have to de-normalize our data model: We will consider "*Ingredient*" as two separate things; one will be the "*Ingredient*" model used to keep track of ingredient inventory. The other will be a string representation of the ingredient:

```
// Order:
{
    type: "PIZZA",
    ingredients: ["CHEESE", "BACON1"],
    …
}

// Ingredient:
{
    key: "BACON1",
    string_representation: "Canadian Bacon",
    stock: 134,
    unit: "SINGLE_PIECE",
    …
}
```

The services will validate the existence of inventory in the "*Ingredient*" database and add the string representation to the order record if the validation is correct. This choice will allow us to take advantage of the high locality in the models and use a document-based database like MongoDB.

Defining a distributed database strategy

Notice that we have defined our high-level design to consider each model as being stored in separate databases. As we did with the services, this separation is logical: The models may be stored in the same database or on independent servers.

We must distribute our data among multiple servers to provide acceptable performance levels. To do so, we will design a region-based sharding strategy.

We can group multiple locations into a single region. We can define the regions' limits using each location's size to distribute the load evenly and keep database servers close to each location.

However, since we assumed we knew nothing about the existing systems, we will group locations based on their geographical location. Regions containing big cities will be smaller, as there is a higher chance they will have many users. Let us assume we have looked at our locations and defined five different regions.

To avoid having a single point of failure, we will apply a few strategies:

- Each region will have multiple database servers.
- All the data for a region will be kept in the same cluster.
- The data for all regions will be replicated locally (multiple replicas in the same region) and globally (multiple replicas across all regions). This will protect us if a whole region cluster goes down: We will still be able to redirect traffic to other regions.

A region will contain multiple locations, but each location can have different traffic demands. Because of this, we will shard the data within a region cluster using consistent hashing. This sharding strategy will guarantee that the traffic load for all locations is spread evenly across multiple servers in the same region.

Search term: consistent hashing

Consistent hashing is a technique used to distribute data across multiple servers. This sharding strategy does not depend on the number of nodes in the cluster directly.

Instead of sharding data based on data attributes, records are distributed in a hash-map pattern: A hash is created for each record, and this hash is used as a key to define which server the record should be stored. That same hash is used later on to retrieve the record.

The cluster defines key ranges for each server. If one of the nodes goes down, the server with the following range takes over. If a node is added to cope with increased traffic, the node with the most records splits its range and re-distributes its data with the new node.

Unlike regular hashing strategies where all data needs to be re-distributed across all nodes when one or more nodes are removed or added, this distribution guarantees that only a tiny percentage of the data will be shuffled around.

Figure 11.12 shows a simple low-level design for the database layer:

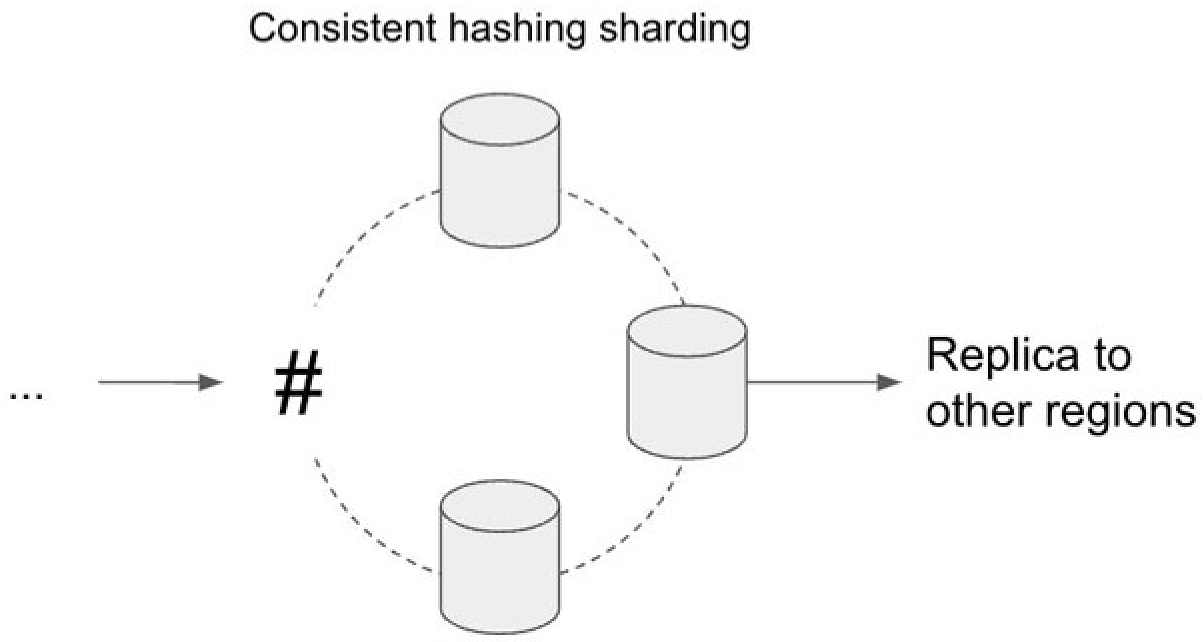


Figure 11.12: *The consistent hashing strategy follows a ring-like pattern*

Again, defining a specific number of servers per region is done based on the calculated throughput of each server and the number of expected QPS per location.

Defining *read* and *write* replicas

Since high consistency is one of the main non-functional requirements we defined initially, we may want to direct all writes through a single node. This approach may conflict with the fact that we labeled our application as "*write-intensive*".

Fortunately, many design choices we have defined reduce the amount of "*write*" operations we need to perform for a single server. Our sharding strategy alone may be enough to allow us to perform all *read* and *write* operations we need without relying on multiple replicas.

We choose to keep replicas for failure management purposes. Secondary replicas may not need to respond to queries unless the primary database servers go down:

- Database servers for one location can hold "*cold*" replicas of other locations.
- Database servers for one region can hold "*cold*" replicas of other regions.

Of course, these replication strategies must be considered when calculating the storage requirement for each database server.

The number of replicas we need to create depends on the expected QPS and throughput calculations, aiming to have enough servers to keep server load at a specific utilization threshold. Once the data storage passes those thresholds, we may consider adding more shards to a region's cluster.

About scaling the file storage

As mentioned in the previous section, we will use a hosted file storage for the file attachments, so we don't need to define a low-level design for this component. We will pay for the cloud provider to scale this service for us. This saved effort is one of the benefits of using cloud-based infrastructures. Sure, it may be a bit pricier, but we will be able to focus on building quality into the other features.

Integrating each layer

We have defined high-level and low-level designs for each component in the system. Now, it is time to put them all together.

Client to server

Clients will communicate with the server through HTTP requests. Since the interface for creating an order is pretty straightforward, the server may expose its API through a REST endpoint. However, querying orders may require different search criteria: Users will search orders created with their user account. Managers will search orders based on time ranges and locations. Owners will search orders based on multiple time ranges and multiple locations.

These search criteria may change with time, considering that the Pizza Place owners are increasingly relying on their system to gain better insights into their business. We may offer ourselves some freedom in the future by exposing our API through GraphQL. If our team has the skills to implement it (and the development team agrees to the added complexity), go ahead and expose the service as a GraphQL API. Otherwise, REST may be a good starting point.

Since we have divided our application into multiple regions, we may want to redirect our users to the load balancer closest to them. We will need a routing service or reverse proxy for this. This service will look at the geographic location of a user and direct the request to the adequate region cluster.

Web server to services

Since up-time is critical for the business, we must add resilience to the communication between the web services and the services hosted in the Kubernetes cluster. An **application queue** service like Kafka may offer temporal storage if one of the services goes down.

Each microservice will listen to the service queue and consume the application events as soon as the web server puts them in the service queue.

Services to database

Services will communicate to the database through the **cache service** (for example, Redis). The cache service will provide resilient communication with the database, retrying queries in case the database is unavailable.

Overall low-level design

Once we have completed a deep dive into each system component, it is time to put together a complete low-level design for the application. We can see the result in *Figure 11.13*:

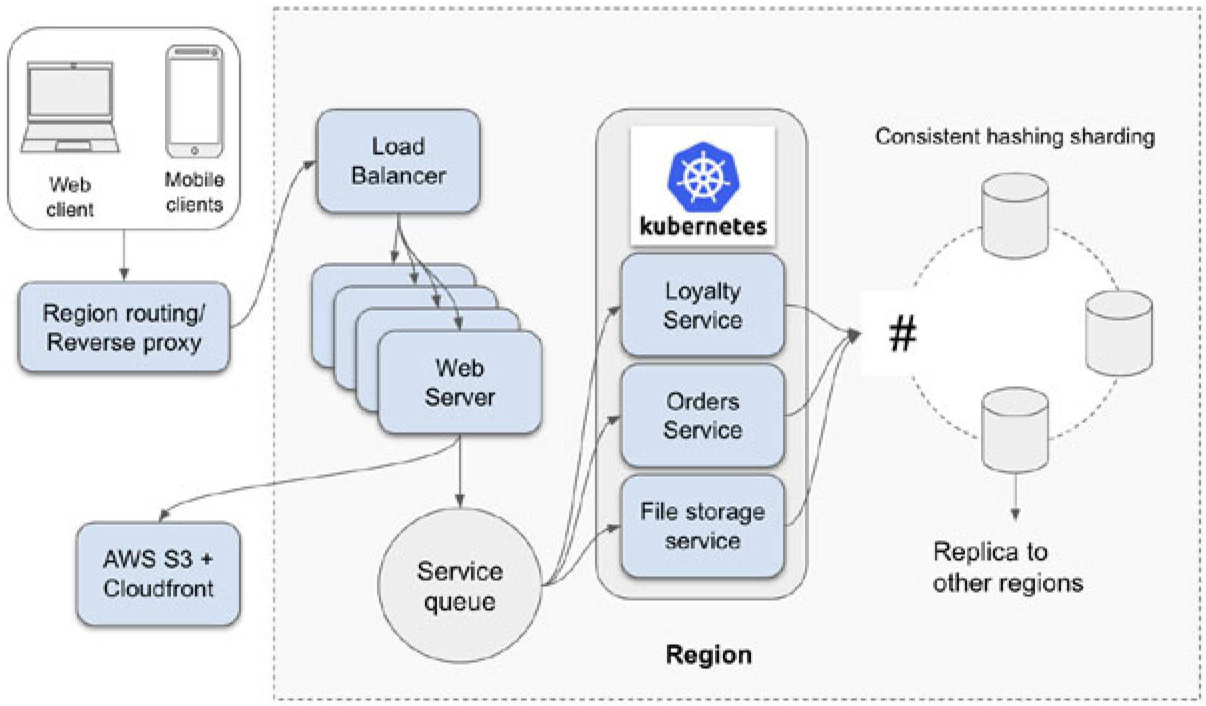


Figure 11.13: *Step 5*

Identifying failure points

One of the main characteristics of a distributed system is its tolerance to failure. Software applications (and the computers they are running on) can fail due to multiple reasons:

- Application errors. Malicious users cause the application to be in an inoperable state.
- The server runs out of memory or disk capacity.
- The server loses its network connection.
- The data center goes offline due to network outages or natural disasters.

When a server becomes inoperable, we have two choices to fix the problem: *try to recover the server or replace it with another server*.

When a server gets into a bad state, it may be because of a networking process or the application itself stopped; a simple reboot is often enough to get it into a working condition again. The advantage of recovering a server is that we can recover all the data state it was holding before the failure.

Recovering a server is particularly useful for database servers, where we risk the data not being correctly backed up before failure.

The main disadvantage of recovering a failed server is that it is a complex, uncertain, and slow process. We must correctly diagnose the problem, fix it, and start the application while the server remains incapable of serving client requests. And there is no guarantee that we will actually be able to fix the problem. Because of these complexities, it is often easier to just replace the server.

Replacing a server is often faster than recovering a failed node. The process can be almost instantaneous if the server we use to replace the failed server is already up and running. We just start redirecting traffic to the healthy node.

The disadvantages of replacing a server are also related to state: The healthy server may not have an up-to-date copy of the server's memory and disk data, and replacing the offline server will result in data loss. This is why we have put so much emphasis across this book on keeping stateless servers and microservices: It simplifies state and error management for servers and containers.

In general, distributed systems deal with failure through redundancy: Every server and service in the application should have at least one copy that can take over in case of failure. In web servers and Docker containers, redundancy means having multiple server copies. In database servers, redundancy is having numerous replicas, even if we already apply a sharding strategy.

In this last step in our design process (*Figure 11.14*), it is time to consider all the things that can go wrong and see if we have addressed all concerns. The goal is to identify single points of failure: Places that can cause the system to be inoperable if something goes wrong.

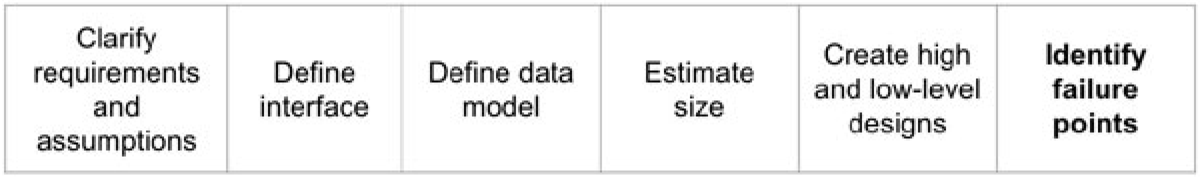

Clarify requirements and assumptions
Define interface
Define data model
Estimate size
Create high and low-level designs
Identify failure points

Figure 11.14: *Step 6*

Let us look at some failure cases to see if our design is robust enough to deal with them so that the system can still operate while we fix the problem.

Failure case: A web server goes down

If a web server goes down, the load balancer will choose a different replica. If the number of web servers goes down a pre-determined threshold (let's say half the servers are offline), we can create new web server instances.

Thanks to the stateless nature of the web server, adding new nodes behind the load balancer should be straightforward.

Failure case: A microservice instance goes down

One of our microservices instances may fail. If this happens, the Kubernetes cluster can create a new container instance and replace the failed service. Again, this is possible only if we keep our microservices stateless: We should keep all the persistent storage in the cache and databases.

Failure case: The file storage service goes down

If AWS S3 goes down, there is little we can do. A simple fallback would be *not to store the file attachment* and log the failure. The upload-picture feature may be unavailable but the rest of the application will work with degraded functionality. However, if the application owners determine this trade-off is not acceptable, our design should expand deeper into other storage alternatives.

For instance, we can use multiple cloud providers, like using both AWS and Azure. If one of the providers is down, we can fall back to store the files in the second provider. Then, we can run an offline process to move the files to the first provider once it goes back online. This approach is more complex and may be better suited for applications where accessing the file storage is critical.

Failure case: A node in the service queue goes down

If a service queue node goes down, all traffic can be redirected to another node. However, any transactions in progress for that node that the microservices have not consumed may be lost.

We can define a time limit for recovering the node. If the node does not come back online (or if it does, but the data is lost for some reason), the application should be able to update the order status to reflect the problem.

This is only possible if the application applies order updates within a single transaction: Only update the order status once all the tasks have been completed successfully.

Failure case: A node in the cache layer goes down

If a node in the Redis cluster goes offline, we can redirect requests to another node. No recovery is needed; the cache writes all data to the database first. The worst-case scenario will be that the application will have to serve requests from the database instead of serving them from the cache storage, resulting in degraded performance.

Failure case: A database server goes down

If one of the database servers goes down, we will lose a shard. However, since we keep replicas both local to the region and outside it, a workaround exists: If the shard is down, redirect the query to one of the "*cold*" replicas in the same region. If the region has no replicas or the replica is also down, redirect queries to a "*cold*" replica in another region.

If we can recover the server, synchronize it with the replica that took over, and once the server is up to date, bring it back online and start accepting queries.

Failure case: The infrastructure for a whole region goes down

In recent years, it has become common to find news like "*Cloud provider goes down, taking thousands of applications with it.*" In these cases, if a single region (a cloud provider-defined region, not one of our Pizza Place regions) or data center goes down, then all the services for the applications hosted in that region also become offline.

Development teams often could have prevented these failures if they had replicated the application across multiple provider-defined regions (or across different cloud providers). It is a more expensive and complex approach, as we need to pay for more servers and synchronize them correctly. Still, it guarantees their applications are resilient during these high-profile events.

Extra considerations about failure management

The system we just designed does allow us to replicate our application across multiple Pizza Place location-defined regions. In worst-case scenarios, one region can take over traffic for another while we work to bring the failed servers up again.

The following are some extra patterns and techniques used to improve the availability and resilience of microservices in a distributed system:

- **Sidecar pattern**: Deploy components of an application into a separate process or container to provide isolation and encapsulation.
- **Circuit breaker pattern**: A proxy service put in front of a service invocation. This proxy allows developers to create error management logic when a service cannot be reached: retrying the service calls, providing reduced functionality, and so on.
- **Chaos testing**: Controlled experiments that simulate infrastructure failures. Chaos testing increases our confidence in the system's ability to operate, possibly at a reduced capacity during the failure of services and servers.

We will not go into further depth on these patterns, as their implementations often depend directly on the underlying tech stack used to build the services and the specific business needs.

After going through all the possible points of failure, we can complete the design process and move forward with the implementation. However, this design must be reviewed by as many peers and stakeholders as possible. After everyone involved understands and approves our plan, we should implement our designed system.

Remember that the system design is a live document: It will keep evolving as we implement the application, and the design needs to reflect any changes. The design is only a blueprint, the ideal state of the application. The system implementation will reveal that many of our assumptions were incorrect and we must be flexible enough to adapt to the ever-changing business landscape.

Conclusion

There are many steps involved in the system design process. It is critical not to skip any of the steps, as each informs our decisions in the next ones. On the other side, the design process is dynamic, so we can revisit each step as we find gaps in our assumptions.

The first steps are focused on listing requirements and clarifying assumptions. We should spend the most time and effort possible at this step, as it will render a solid base for all the decisions we will have to make later. Hard facts and data should back up each assumption we make, so it is essential to have feedback from every stakeholder early on.

Key stakeholders are the business domain experts, other developers, and as many technical and non-technical experts we can involve in the process: site-reliability engineers, application security experts, compliance teams, legal advisors, and so on. A strong team will provide the feedback we need to be confident in our design decisions. And while some of these experts may not be available to us, we should at least look at our design from their point of view and understand their concerns.

Once we have a clear understanding of all the requirements, we must decompose them into multiple use cases for the system. Then, we extract from each use case all the components and actions required for users to fulfill their goals in the application.

We define actions as operations in an interface. The interface should receive data as parameters and return the result of the action. The interface will help us identify what data is needed and what isn’t to achieve the use cases.

Having identified the entities passed back and forth through the interface, we must model their attributes and structure. Models help us understand and define constraints around what data needs to be stored and how much capacity we need to reserve in our servers to store it.

The amount of web servers we need is calculated based on the estimated average and worst expected QPS for each user action. Since each server has limited throughput, estimating how much data will enter our system will help us calculate how many nodes will be needed to cope with traffic requirements at acceptable performance levels.

Data models also inform what type of database we need to represent the system's state. This will lead us, in turn, to design better ways to distribute the application state across multiple shards and replicas. We should choose a sound strategy based on the characteristics of the data, the expected load distribution across the clients, and the consistency and read-write rates we defined while identifying non-functional requirements.

Even when the steps in the process are well defined, there is no single system design that fits all use cases. The whole process is a mix of art and science. We must be creative, collaborative, and fully open to feedback to find the right tools that will allow our application to grow and help our users achieve their goals.

With this chapter, we have almost reached the end of the book. We have used all the knowledge we have acquired across each chapter, so feel confident enough to go out there and start practicing the skills you have received. The only way to become proficient at something is through constant practice.

Remember that this book contains only a tiny fraction of all the knowledge you will acquire during your career as a backend developer. You now have the right tools to start exploring for yourself all the possibilities out there for you to build software applications that will change people's lives for the better.

Questions

1. If we assume we have a single-server infrastructure, what changes can we introduce to handle a sudden increase in traffic successfully? Which components and tools can we introduce? How can we leverage caching strategies?
2. "Over-architecturing" is a serious and common problem were we design a system with more features than needed. What are the

disadvantages of designing a system that is too large or complex for the system use cases?

3. Think of popular, large-scale systems like Google, LinkedIn, or Facebook. What would the result of this design process look for them?

Resources

- "How to get better at 'Back of the envelope' calculations": **https://www.wired.com/story/how-to-get-better-at-back-of-the-envelope-calculations/**
- What is load balancing? **https://www.nginx.com/resources/glossary/load-balancing/**
- "Promoting replicas for regional migration or disaster recovery": **https://cloud.google.com/sql/docs/mysql/replication/cross-region-replicas**
- "How to identify and mitigate single points of failure": **https://scalefaster.com/identify-mitigate-single-points-failure/**
- Consistent hashing: **https://www.toptal.com/big-data/consistent-hashing**—
- Side-car pattern: **https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar**
- Circuit breaker pattern: **https://microservices.io/patterns/reliability/circuit-breaker.html**
- Chaos testing: **https://www.infoq.com/news/2016/03/chaos-testing-microservices/**

CHAPTER 12
Bootstrap Your Career Path

Whether you're only starting your career or you have a few years of experience, a back-end developer never stops learning and progressing. We use this last chapter to define a common path for junior developers to become senior developers, strategies to keep you updated, and advice on preparing for technical interviews.

Structure

In this chapter, we will learn the following topics:

- Defining the expectations on junior developers
- What makes a senior developer
- Improving hard and soft skills
- Preparing for technical interviews
- Finding mentors
- Finding resources to keep learning

Objective

By the end of this chapter, we should understand how to advance further in our careers as back-end software engineers.

Specifically, we should have a good understanding on:

- What is expected of us when we land a software developer role
- How to become a senior developer
- How to tackle coding interviews
- How to set up a path to increase our knowledge

Defining the expectations on junior developers

Every developer is a junior developer at the beginning of their career. Being a junior developer does not mean we are not competent or good at programming, it means we have a lot yet to discover of the world of working in software development projects.

At the beginning of our career, it is challenging to fully grasp our role in a software development project. As discussed in this book, software development has many responsibilities; some require a deeper understanding of specific areas like data theory or algorithms fundamentals. Finding our place in the middle of all these tasks can be scary.

First, rest assured that almost no professional development team will expect junior developers to do it all. More senior developers understand the pressure junior developers face because they have been in their place; good managers will ensure that a junior developer is only assigned tasks they can handle.

So, what is the job of a junior software developer?

Joining our first development team

When we join a development team for the first time, we may find ourselves a little disappointed; yet also relieved because most of the work we will be assigned is simple: Fixing minor defects or making simple changes to the application.

The goal of assigning junior developers simple tasks is to set them up for success. Being able to complete these simple tasks builds the confidence of junior developers and gives them the courage to explore and take more complex work.

However, even senior developers benefit from taking simple tasks when they first join a team. Fixing minor problems allows us to concentrate on crucial aspects of the team-defined development process itself:

- How is the code structured?
- How to deliver code changes through the CI/CD pipeline (if there is one), successfully deploying them to production?
- How the team deals with defect tracking and reporting?
- How the team peer-reviews code changes?

- What are the team’s code quality guidelines?

The first commit a new developer makes is sometimes a good indicator of their experience. Senior developers will always put together the simplest code change possible and focus on taking it all the way to production.

Simple code changes have the advantage of requiring little peer-review work, which, in turn, has two benefits:

- It increases the chance of other developers reviewing your code change. In many teams, developers can choose what code changes to review. People will tend to choose more straightforward, easier to review code changes.
- It decreases the chance of having your commit rejected during peer review. Smaller commits mean fewer places where we could introduce a bug.

The following are a few good practices not only for junior developers making their first commits but also for every code change done to an application repository:

- Make small commits.
- Only include essential code changes.
- Stick to team and industry-defined best coding practices.
- Document the code and add comments where necessary.
- Add all pertinent documentation and change descriptions to the commit.
- Look to get as many reviews as possible.

Most of the time, managers don’t expect junior developers to build complex features within their first weeks of work. They also don’t expect developers to deliver a lot of code changes in a short period. We prefer fewer commits that have high quality built into them.

Following guidance from other developers

Software development is a team effort. Those developers who don’t know how to work well with others are set up for failure, especially when they first join a new team.

The best thing a developer can do is create good relationships with other team members. We don’t have to be best friends with everyone, but *we should build relationships based on trust* with the rest of the team. Otherwise, collaboration becomes impossible, and we cannot achieve common goals.

Other team members are one of the greatest assets a junior developer has when they first join a new team. These developers will know the existing processes, gaps in the documentation, and best practices to help us ramp up faster in the project. They can help us move forward when we get stuck with a task.

Most development teams assign a mentor to new team members. It’s a mentor’s job to guarantee their mentees have the knowledge they need to be productive team members. It is a mentee’s job to trust their mentor and follow the path the mentor sets up for them. Of course, we have input in our career path, and mentors are open to discussion if we are not comfortable with the direction we are following.

Being open and transparent with our new teammates will allow us to quickly transition into more complex tasks. When a mentor and teammates have the visibility into a junior developer’s work and its results, they can start assigning increasingly exciting work.

What makes a senior developer?

The transition from a junior developer to become a senior developer takes time. Being a senior developer takes more than having the necessary technical skills for the job. It requires practical experience in the field.

Many companies have different criteria for considering a developer as ‘*senior*’. There are cases where a developer has the title of a “*senior developer*” in one company only to be designated as a “*developer*” when they transition to another company. The opposite is possible, too. A non-senior developer can become a senior when they move to a new job. These inconsistencies show us that being a senior developer does not necessarily involve having that title assigned to us.

Characteristics of a senior software developer

A senior developer has a set of characteristics that remain semi-constant from team to team. Some teams will have more specific criteria to designate a developer as "*senior*", but this list of features truly makes a senior developer not only in the title:

- Senior developers act independently. They need little to no supervision to complete their work.
- Senior developers own their work. If something goes wrong due to the developer's actions, the developer will openly recognize the situation and act to remediate it.
- Senior developers are team workers. They don't need managers to mediate and find agreement with other developers. They have the communication skills to find a middle ground with other people who have different opinions.
- Senior developers build quality. They produce more value than what is minimally needed to create a feature.
- Senior developers are domain experts. They pick a field in which they specialize and provide guidance to everyone who needs it.
- Senior developers mentor other developers. They usually are mentors to other team members and support their work and careers.
- Senior developers constantly improve themselves. They understand their strengths and weaknesses: They use their powers to help others and find ways to improve on their weaknesses.

How does a junior developer become a senior developer? There is no single path. However, we can use the characteristics we just listed to recognize gaps. We can ask ourselves the following questions:

- Do we have the technical skills to write code that will result in robust applications?
- Can we work on the project without having other people constantly tell us what to do?
- Are we good team members who look up for every person in the team?
- Have we mentored other people?

If the answer to any of these questions is no, we now know our opportunity areas to improve our chances of becoming senior developers. We can create an action plan that will foster our skills.

The following are some actionable items for any junior developer who aims to become a senior developer:

- Improve hard and soft skills. Communication skills are just as valuable as programming skills for senior developers.
- Do the work ourselves, but be open to help. It is valuable that a developer tries to tackle complicated challenges themselves instead of expecting other people to provide solutions for them. However, senior developers recognize when they become stuck and need help from others.
- Aim to become a mentor. Many companies offer developers the opportunity to mentor new team members. This work is usually optional, and beyond our day-to-day responsibilities. Some developers choose to skip mentorship roles, but we should take these chances if we plan to become a senior developer.

Being a senior developer is more than a title. Senior developers build value for themselves, the product, and their team members.

Improving hard and soft skills

The only way to become better at something is through practice, and the best way to become proficient in any skill is by using it. Practice takes time, dedication, and patience.

Professional skills are often categorized into hard and soft skills. Hard skills are technical capabilities like coding and system design skills. Soft skills are more related to the way we collaborate with others.

It is common for people with technical roles to avoid fostering "*soft*" skills: Interpersonal communication, problem-solving through collaboration, or conflict resolution. They believe that technical or "*hard*" skills alone are enough to advance their careers. And while some developers manage to get to management positions with poor soft skills, the truth is that most people will get stuck in their career development if they don't improve skills like communication.

Improving technical skills

For most software developers, improving hard skills is straightforward:

- Take a tech stack and keep building applications with it.
- Follow news feeds and blogs to keep ourselves up to date with new technologies, tools, and frameworks.
- Read books and take online classes.
- Attend tech conferences and meet-ups to learn the latest advances in the industry.
- Find a technical mentor that helps us learn best practices and advanced patterns.

Like in any other technical field, we can find different areas of specialization in software engineering. Each topic covered in this book has an extensive background that overlaps with fields like mathematics or electronics; as professionals, we can choose to go as deep on them as we want.

For instance, some developers choose to build their careers around data management and storage. They become experts in databases and the way they operate internally. They can find work with database providers like Oracle or companies that operate in big data like Google. People who choose to work around data management can find a lot of resources in *Information theory*: a field that covers data and how it is stored and transmitted.

Other developers can choose not to go as deep in any specific technical field and expand their knowledge horizontally into other areas. This work can lead to a technical architect role, which has its own level of complexity. These developers still have successful and meaningful careers solving business problems for other people.

One of the reasons learning software development can feel so overwhelming is the complexity behind each component in a system. It is easy to think that we must become experts at everything to be good software developers. This feeling is justified but a little misguided. We don’t need to be experts in every field thanks to one of the most powerful concepts in software development (and science in general): complexity abstraction.

For instance, we learn basic math during our school years. Many people in the world use math proficiently in their daily lives without a deep understanding of the underlying theory behind arithmetic operations. We don't need to get a Ph.D. in Mathematical Theory to use addition or multiplication. All the facts necessary to make the math work are abstracted out, and we trust that the operations will work as defined.

Thousands of software developers build successful applications daily without ever deep-diving into the inner workings of the tools they use. Just as multiplication is an abstraction layer built on addition and we can multiply numbers without thinking of them as additions, we can use most software development components without knowing each piece of its inner workings.

This doesn't mean that we should not strive to understand the theory behind software development. Understanding the basic principles gives us a better intuition behind the "*best practices*". The more knowledge you acquire about fields like information theory, network theory, hardware specifications, and low-level programming principles, the better developer you will become.

The key takeaway here is that we should not aim to be experts at everything. As long as we understand the big picture of software development (like we have tried to achieve in this book) and specialize in one or two areas, we will become proficient software developers.

Improving people skills

Soft skills are often referred to as "*people skills*": The capability of collaborating with other people to achieve a joint goal.

Software developers often undermine soft skills because software development is a task we can achieve in isolation. Given enough time and resources, a single developer can build a complete application without little to no interactions with other people.

However, resources like time and money are not infinite and are not always readily available. We can achieve a larger goal through collaboration with other professionals while working within these resource constraints.

Soft skills have a less clear path for improvement. They usually require people to introspect on their attitude towards others. This introspection is not an easy process, as it often leads to finding flaws in ourselves and the way we treat other people.

Let us discuss some of the actions we can take to improve our soft skills.

Practicing conflict resolution

In a team composed of people from multiple backgrounds, there will always be a case where team members don't agree on something. In these situations, it is critical to move ourselves to find consensus with others.

Finding a middle ground usually requires compromise. Choosing a solution that will not fully satisfy the expectations of both parties but, in the end, will provide the most viable path forward.

We can practice conflict resolution through empathy. What is the other person trying to achieve? Why is their goal in conflict with ours? Do we fully understand their point of view?

More often than not, conflicts come from people trying to achieve different goals that are not entirely in sync. We can prevent most conflicts by aligning those goals first.

Working on communication skills

The basis of any collaboration is communication. We share our ideas with others to get feedback. We listen to other people's ideas to gain insights from their points of view. Ideas circulate across the team members, improving during the process. Communication is one of the most critical skills for any software developer; yet we treat it as a second-class citizen.

We often believe that having communication skills is being good at reporting to managers, being charismatic, making friends easily, or public speaking. All these, while helpful, are not communication. Worst, some people think that being a good communicator involves using many long words and complex terms. That is the opposite of good communication.

Communication skills mean sharing information concisely and clearly. Simplicity is a key. We should communicate the same ideas with the simplest terms and the least amount of data possible.

Software developers tend to overuse technical terms when communicating with people who may not be as technically proficient. While technical terms are enlightening for those who know them, they obscure information for those who don't. It is critical to understand our audience: Communicating the design for a system should be addressed differently when the receiving end is a team of engineers than when it is a team of business analysts.

Richard Feynman, a famous American physicist, was known for his ability to explain the most complex topics to a broad audience. His lectures at universities were so enlightening and easy to understand that people ranging from undergrad students to experts like Albert Einstein would eagerly gather around to listen. Feynman had a simple yet powerful idea. If you can't explain something in simple terms, you don't understand the topic.

Following Feynman's principles, we can gauge how well we understand software development by measuring how well we can explain it to non-technical people in the simplest terms.

Good communication skills have multiple benefits:

- People will be more open to working with us.
- We will be able to demonstrate the full extent of our knowledge during work interviews.
- Managers will understand the value we are providing to our teams.
- Other team members will understand our expectations of them and their work.
- We will be able to justify each decision done during the software design and development process.

Just like hard skills, we improve soft skills through practice. Join a debate or public speaking group and engage in constructive discussions. But above all, be open to feedback from others; by identifying the cases where we fail to communicate efficiently, we will know when to fine-tune our communication and people skills.

Combining hard and soft skills can lead to software development teams working as one and achieving ambitious goals.

Preparing for technical interviews

Finding a software development job can be a stressful process for many developers. Many talented people fail to find a job because they are not prepared for all the extra steps involved in getting and passing an interview.

This section will break down the typical process a software developer has to undergo when finding a job in a software development team.

Getting an interview

There is a high demand for software developers. The world is in a constant modernization state, moving a lot of processes that historically have been done in person or through more analogical solutions into software systems. Companies in the industry are offering large salaries and excellent benefits because they struggle to find talent.

For many developers, there is a disconnection between this demand and their personal experiences looking for a job. They send tens if not hundreds of resumes every day, only to get one or two interviews. They go through the interview process and end up without an offer. Why is this happening?

Let us put ourselves in the recruiters' shoes. Suppose it is difficult for someone with a technical background to keep up with all the new tools and technologies released every day. In that case, it is more challenging by orders of magnitude for recruiters who typically have little to no technical training.

Typically, recruiters rely on specific keywords to search for candidates. Someone in the development team will provide the recruiter with a list of technologies and an expected proficiency often calculated in years of experience. The recruiters then take that list and search through sites like LinkedIn, where they can find a list of potential candidates and their work history and skills. Only profiles that match these search criteria will pass this first filter.

After defining a list of potential candidates, recruiters may send it back to the development team to choose a few of them for interviews. This second filter will leave out all candidates who don't seem to have the required skills or level of proficiency.

Resumes and profiles in social networks aimed at professionals are the presentation cards for people looking for a job. We may be the best programmers in the market, but we may never get an interview if our profiles are not visible to recruiters.

A good resume or profile needs to achieve multiple goals. It needs to contain all the keywords recruiters use in their queries. It has to showcase our skills and pertinent achievements in just a couple of paragraphs. It has to make other developers want to work with us.

We can achieve those goals by following simple good practices (and avoiding some bad ones).

Building a good resume

A recruiter and the development teams looking to fill a position will read through tens of resumes every day. Unfortunately, due to this high number of potential candidates, employers will only evaluate each resume briefly.

Unclear and long resumes have a disadvantage during these screening processes. Even if the candidate is more than qualified, recruiters will discard them in favor of candidates if their resumes don’t clearly and concisely communicate their skills and capabilities. It is a bit unfair, really, but recruiters operate under limited resources.

Once we understand the recruiters’ point of view, it is easy to understand all those popular pieces of advice on building resumes: Keep them short. Even if we have twenty years of experience, our resumes should not be longer than one or two pages. Following Feynman’s clarity advice, if we cannot state our achievements and skills in a simple way is because we don’t fully understand them.

When we force ourselves to enumerate our achievements and skills in less than one page, we will improve our chances of prospective employers reading all we have to say about ourselves.

Something that can give us the edge over other candidates is to craft individual resumes for potential employers or roles. This is especially useful when you have extensive experience in multiple areas: It puts relevant experience at the top and leaves out irrelevant information.

For instance, we may have experience as full-stack developers. But if we are applying to a back-end development role, there is no use in going in-depth on our front-end skills. A simple mention at the end of the resume will be enough to communicate that it is something we can also do. Or, if we are applying to a team specializing in building REST APIs, we may want to go more in-depth on our experience working in that topic.

People with a lot of experience think that listing all of it will make them look good when instead it is causing the opposite effect.

Another good practice is to list specific, measurable achievements. It is better to write "*working in the database management team, I reduced query time a 34% by implementing an effective indexing strategy, resulting in an increase in profit of X USD" than to write "10 years of experience improving database performance.*" Specific goals give employers a good idea of what you could do for their teams.

If we struggle to state our achievements in the previous projects, those achievements are probably not worth mentioning, as they will not give you any advantages over other candidates.

The importance of networking

Having a good resume is just the first step in getting an interview. While resumes and profiles improve our chances of appearing in recruiters' searches, we are still at the mercy of search algorithms.

We can improve our chances of landing an interview if we know someone already working at the company. Employers take referrals seriously; they even have rewards programs for existing employees to refer to people they know for open positions.

Of course, we will not go around asking random people to refer us. While some people have no issue having people they don't know reach out for a referral, others may consider it rude and may hurt our chances of getting an interview.

Networking is the most effective way of knowing people in companies that can be prospective employers. We can engage in conversations with other developers, recruiters, and managers through social networks like LinkedIn or social events. College alumni programs are also an excellent way to network with other professionals.

Again, we can see the benefits of improving our soft skills. When we have an extensive circle of known people, we may be gaining access to professional opportunities that otherwise would be unavailable for us. People, after all, are social creatures, and we tend to want to help others we already know.

Networking may be complex for introverted people. The expectation here is not to make a lot of friends. The idea is to engage with other people in things we are interested in, as we will gain a lot of insights from their points of view. Fortunately, we live in a world where we don’t have to meet other people face-to-face to create a robust network of people.

An effective way to increase your network is to join clubs and software development communities. Collaborating in open source is an often-used way for developers to showcase their work and meet other people in a comfortable setting, even for introverted people.

Improving interview skills

Once we have built a robust and concise resume or a good profile in a social network, recruiters will start approaching us. Not all opportunities will be as generous as we expect them to be, especially at the beginning of our careers but some will.

Recruiters will reach out to us, asking to set up an exploratory call. This first call has two goals: The recruiter wants to give us a high-level description of the project and the open position. What the team does and skills they are looking for in a good candidate. The recruiter wants to know more about us. Our experience, our background, the kind of work we are interested in, and our skills.

In this first call, they rarely will ask technical questions to us. The idea is to explore if we would be a good candidate to interview for this position. The recruiter will likely go over our work experience as listed in our resume, so we should be able to provide more in-depth details about each project and achievement.

If we like what the recruiter says to us, the next step will be to schedule a technical interview.

The technical interview process

Software development companies, especially large ones, are known for their particular technical interview process. The goal of a technical interview is for prospective employers to understand whether you have the technical skills needed for the open position. However, it is also an opportunity for us to better understand what it's like to work there and whether the development team and position is a good match for us.

Technical interviews can be very simple like a brief discussion about our previous work experience with a potential manager or very complex solving coding problems or building a small software application.

We will describe a typical process for companies like Facebook, Google, or Apple in the following sections. Some other companies may have only a subset of these modules or require even more steps. The idea is to know what we can expect on each part of the interview and prepare for them.

The typical process usually has the same steps for interviewing a junior developer as for a senior developer. However, the degree of complexity and the expectations on each interview varies depending on the seniority level of the position.

The typical interview process contains multiple modules:

- Coding & data structures and algorithm problems.
- System design problems.
- Take-home assignments.
- Manager discussion.

Approaching coding problems

In coding interviews, interviewers try to assess whether the candidates have experience solving problems requiring data structures and standard algorithms. People often refer to coding interviews as "*whiteboard interviews*".

This interviewer presents a problem to the candidate. The problem may sound over-simplistic and, at times, unrelated to the software development process. The expectation is for the candidate to put together a performant code solution for solving the problem.

This interview allows interviewers to assess multiple skills at once:

- Problem-solving: How the candidate deals with a potentially unknown problem?
- Finding the problem's constraints, asking for clarifications in assumptions.
- Knowledge about data structures: Does the candidate use the proper data structure for the problem? Does the candidate understand the trade-offs of using one data structure versus another?
- Knowledge about algorithms: Does the candidate recognize if the problem can be solved with a well-known algorithm?

Let us describe an example question. This will be a simple problem proposition, but it will allow us to demonstrate how to tackle this kind of problem efficiently.

The interviewer asks us: "*Write a function to find a specific number in a list of sorted integers.*"

A weak candidate will immediately jump in and start writing the code. They may choose a poor performant solution like iterating through each element in the list until they find the number they are looking for. Even if they find a performant solution; this shows poor problem-solving skills. Any software developer should understand the problem before trying to solve it.

However, a good candidate will pause and analyze the problem proposition for a second. What exactly is the interviewer asking us to do?

First, we need to analyze the problem the interviewer gave us. For instance, we know we will receive a list of numbers and one number we will look for in the list. We know the numbers in this list are natural numbers, as they are integers.

We also know that the list is sorted. This last piece of information is critical. It will guide us on what algorithm we can choose to solve the problem efficiently.

Let us think of a naive solution. Iterating through each element in the list requires us to search the whole list. As the size of the list increases, so will the amount of work our application would need to perform to find the result. In the algorithm design world, we say that this linear search has a linear time complexity, or *O(n)*. This naive solution is not taking advantage of all the facts provided by the interviewer. For instance, we know that the list is already sorted. How can we use that to our advantage?

Well, problems for searching elements in sorted lists are great candidates for binary search. Since the binary search eliminates half of the elements in the list on each iteration, it needs to execute considerably less work than the linear search. Binary search has a time complexity of *O(logn)*, significantly more performant than *O(n)*.

If we don't have a more formal background in software engineering, we might wonder what these "O"s mean. This is known as **asymptotic complexity**. It is how we measure how performant an algorithm is by looking at how much work it needs to perform as a function of its input, in the worst case.

This is why we assume that a linear search will have to search in every element in the list, even if there are cases where the element we are searching for may be the first element. Asymptotic complexity measures an algorithm performance using the worst-case scenario.

We will not go into more depth about asymptotic complexity, as it is a topic that could cover a whole book by itself. However, since this is a topic any software developer should understand, we have included a link in this chapter's references section.

The key here is to define a solution before we start coding it. Try to solve the problem manually by defining some examples. Try to step through multiple use cases. These examples offer a deeper insight into the obstacles we may find while coding our solution.

Finding a solution before we write its code will save us a lot of valuable time because once we start writing code, it will take a lot of effort to discard a poor solution and rewrite all the code you already have.

Using data structures efficiently

During coding interviews, identifying suitable data structures for specific problems helps us solve the problem. We can use some of the following heuristics while defining the data structure we want to use.

Hash maps are good for retrieving data in semi-constant time (meaning, fast!). If we need to retrieve an element from the data structure over and over again, hash maps allow us to do these repeated searches efficiently.

However, finding ranges of elements (e.g. finding the ten largest elements) is not efficient in normal hashmaps since their elements are not sorted. A binary search tree or a sorted list is better suited for those tasks.

Binary search trees (BSTs) allow us to search efficiently; assuming the tree is balanced because their elements are sorted, and we can apply binary search on them (it is on the name!).

BSTs, like any sorted data structure, are also a good choice to find max or min elements. However, using a heap is even more efficient to keep track of max or min elements, as that is their sole goal.

Linked lists are better than arrays for generating dynamic lists. We can just add more nodes at the end. However, searching elements in a simple linked list require us to search through all nodes. But once we find the location of an element in a linked list, it is faster to remove it (or add more nodes around it) than it is in a regular array where you would have to shift all other elements after a deletion.

As we see, each data structure has its trade-offs, and each is better at some problems than others. Understanding these advantages is what helps us build efficient algorithms.

Using algorithms efficiently

In our example, we have mentioned binary search as an efficient way to search in a sorted list of elements. What if our list is not sorted?

We have a few options here. The most straightforward approach is to sort the list. We can use an algorithm like merge sort, which sorts elements in *O(nlogn)* time. Then, we can apply binary search. However, this approach still takes *O(nlogn)* time, which is consistently worse than just doing a linear search. So, even if it seems like we did the smart thing in our example by sorting the list first, the asymptotic analysis tells us that we are actually better off with a brute force approach.

Knowing the most commonly used algorithms and their expected performance allows us to choose the right one for our problem.

Coding a solution

Once we have identified that our problem can be solved using binary search, we confirm our assumptions with the interviewer. We don't want to start coding until we get feedback from our interviewer and they confirm that our approach is valid.

A perfectly valid answer here is as follows:

```
Given I={1,2,..,N} of integers, and K>=0
result = binary_search(I,K)
```

This response is valid because binary search is a well-known algorithm with an also well-known time complexity of *O(logn)*. It's what we call a black-box algorithm: a well-defined function we can just use and assume its performance.

However, some interviewers will want you to code the actual implementation of the binary search. Unless you are interviewing for a job designing search algorithms, a request like this from an interviewer is a sign of two possibilities: The interviewer is inexperienced. Experienced interviewers recognize that you don't need to memorize black-box algorithms to be a good software developer. Your solution is not the right one. Maybe this problem requires a modified version of binary search. Then, we cannot treat the algorithm as a black box, and we need to work on its implementation.

Notice that the solution we provided is not using any programming language. We call this pseudo-code. It is a step-by-step description of the algorithm using natural language. We could also have used something like Python or Java. We must clarify with the interviewer if it is ok for us to use pseudo-code or if we need to use a specific programming language.

While (or preferably, before) coding our solution, we should identify edge cases. What if the list is empty? Can the list have repeated numbers? What if all the elements in the list have the same value? Pointing out and addressing these edge cases in our code will indicate to our interviewer that we have good analysis skills.

Once we finish writing our function, we need to provide the asymptotic analysis. This analysis will let our interviewer know that we understand the performance implications of our code. At this point, we are done with the problem.

Coding problems are controversial. Some developers believe that they don't accurately reflect a candidate's real-life experience building software. Others believe that they are critical, as they allow interviewers to assess a candidate's proficiency using data structures and algorithms to solve a problem efficiently. Whether we are on one side of the debate or the other, the truth is that coding interviews are a part of the process today.

While some companies do not require whiteboard interviews, most do. It is better to be prepared, even if we don't fully believe in using them to assess a candidate's skills.

Getting better at coding interviews

Being good at coding interviews is itself a skill that we can improve. Actually, being good at software development doesn't necessarily imply that we will be good at coding interviews, and vice versa (which is another argument of the people who are against whiteboard interviews).

There are resources out there that can help us practice solving coding problems. To hone their coding skills, many software developers regularly visit websites like *leetcode.com*, *hackerrank.com*, and *careercup.com*. These websites have many coding problems that companies often use during technical interviews.

Solving practice coding problems do help us develop an intuition on how to approach them. Then, the problem-solving process becomes second nature.

Working on system design interviews

In *Chapter 11, How to Design a System*, we described the overall steps of designing a software application. System design interviews require us to do a reduced version of that process.

In general, we still need to apply the same principles we described in the previous chapter. Define requirements, confirm assumptions, define interfaces and data models, estimate capacity, create high-level and low-level designs, and find bottlenecks in the design.

However, interviewers don't expect our designs to be very detailed. If it is challenging to create a perfect system design in a couple of days, it is almost impossible to do so in one hour. The interviewer will focus on a simplified version of the problem or a specific module of the system.

The expectation for us as candidates here is to show our problem-solving skills. Again, a weak candidate will immediately jump to describe a system design without having defined and confirmed all the requirements and constraints of the system. A good candidate will take their time to make sure they and the interviewer agree on what needs to be designed before describing how.

On system design interviews, we should focus on creating systems that address the problem specifications. We should stay away from designing over-complicated systems just to demonstrate our knowledge. A simple yet elegant design is better than a complex one.

Some examples of common system design interview questions are:

- Design a chat system
- Design a Netflix-like system
- Design an Instagram-like system
- Design a search engine
- Design an elevator
- Design an application for a parking lot

Notice that these questions are intentionally open-ended. They are designed for evaluating how candidates work when they miss information. What kind of questions do they ask? How they collaborate with the interviewer to make valid assumptions. How they clarify requirements.

In the end, system design questions are all about collaboration between the candidate and the interviewer.

Take-home assignments

Some employers consider it unfair to expect a candidate to design and build an application within an hour. Instead, they give the candidate "*homework*": A small assignment that the candidate will take home and solve before a given deadline.

The expectation is that, since we will have more time and resources to work on the application, the result of the assignment should resemble the type of work we would perform for the team.

We must understand what exactly the take-home assignment is asking us to do. Make sure to read many times the problem proposition until we understand all the constraints and requirements.

Focus on code quality, modularity, and reusability. Don't try to cheat by copying a solution we found on the Internet or by having someone else do it for ourselves. Once you return to finish the interview, the interviewer will ask you details about your solution to the assignment and may ask you to make small changes to make sure you fully understand the code.

While going above and beyond by making a beautiful user interface or using a cool new technology may give you some extra points, please don't do it at the expense of ignoring some of the base requirements. Solving the right problem is more important than showing off.

Approaching non-coding modules

A good part of the software development interviews is a simple discussion between a candidate and the interviewer. These discussions can cover a lot of ground:

- Previous experience in our resume. The interviewer wants to understand our involvement in the projects we listed in our resume: our achievements and learned lessons from previous roles.
- Understanding of the software development process. These questions focus on how the candidate collaborates in a software development process. How well they understand things like version control, CI/CD processes, and so on.
- Team dynamics. How good is the candidate in conflict resolution? How do they deal with co-workers who disagree with them?

Most of the time, managers are the ones who perform these interviews, quite possibly our future manager. They are trying to assess how well we would fit in their teams and how easy it would be to work with us.

These modules are our opportunity to shine some light on our accomplishments, so we should be prepared to talk about them. We should also be ready to talk about challenges we have had in the past and how we approached them.

Here is where our soft skills will help us. We will have an ally during the hiring decision process if we can communicate with the interviewer efficiently both technically and non-technically.

Asking the right questions

Most interviews leave some time at the end to allow candidates to ask their own questions. These questions are not there for us to look smart in front of the interviewer. They are for us to get an insight into our potential employer.

This part of the interview is not the right time to ask about compensation or company benefits. Those questions can be better addressed by the recruiter or other HR team members. Besides, the person interviewing us often does not have visibility into how much we would get paid if we get hired.

We should take this time to gain visibility into the point of view of an employee or team member. The interviewer can talk about their personal perspective on how the team and the company operate.

The following are a few examples of good questions to ask here. They shine a light on our future responsibilities and the challenges the team (and the company) are facing.

- *What do you do in a typical workday?*
- *What is the most interesting challenge you have found in this team? How much input do developers' opinions have in product decisions? Can you choose a thing to improve in the team and the company? When was the last time your manager promoted someone?*

Remember that the interview is not just for the employer to assess the candidate. It is also for us to evaluate whether working there is a good choice.

Finding mentors

Improving our careers as software developers is a constant effort. No matter how experienced we are, there will always be room to grow and new things to learn.

However, there is a limit on how fast we can grow professionally by ourselves. We need new problems to challenge us and people around us to widen our perspective. Moving out of our comfort zone is the first step to growing.

Mentors are people who we can look up to. They often are proficient at something we are interested in and are open to sharing their knowledge with others. They inspire us and help us view the world from a new perspective. Finding a good mentor is an excellent way to advance our careers.

Again, people skills are helpful while looking for a mentor. People will only be interested in mentoring us if they consider we are the kind of person they can easily collaborate with. Being a mentor involves giving honest, yet sometimes challenging, feedback to the mentee. This is why a relationship mentor-mentee requires openness and trust; two values that we can only achieve through good communication.

While we get the best out of mentorship when our mentor is actively involved in helping us, we can find a more unilateral mentorship relationship in people whose work we admire. Many developers find inspiration in software developers they follow on social media or whose work can be found online or in books.

Finding a mentor is especially useful when we find ourselves stuck in our careers. We can look at them and know what we could achieve if we follow their same path.

Mentors have the advantage of having walked the road. They may have succeeded, but they also made mistakes that helped them learn, grow, and get where they are. If we can learn from their experience, we can save ourselves from making those same mistakes.

Keep in mind that being a good mentee involves being receptive to feedback. A developer who cannot take constructive criticism or who believes they already know all the answers will gain little to nothing from their mentors.

Finding resources to keep learning

In recent years, there has been a boom in available resources for software developers to improve their skills. From traditional college education through bachelor’s and master’s degrees to commercial boot camps that promise prospective developers to help them become professionals in only a few weeks. Even with all these educational resources, it may be challenging to find the right learning source for us.

The first step is to think about what type of resources work best for us. Not everyone learns in the same way: Some people prefer visual aids like books or blog posts; others prefer more dynamic and auditive resources like YouTube videos or podcasts.

An effective way to keep learning is to immerse ourselves in the world of software development. Find a topic we feel passionate about; it could be database management, REST API design, or overall system design. We will get better results if you focus on a particular field. Then, start exploring all the free resources you can find about the topic: Blog posts, videos, podcasts, social network posts, and so on. Thousands of resources, some better than others, are at our reach with a single Internet search.

Focus on sources that actively engage with their communities and constantly post news on the field’s latest trends. Look at the same content other experts in the field consume, even if we feel they are too advanced for us.

Take a few minutes every day to consult your sources. We don’t have to memorize everything we see or hear. Just let it sink. Little by little, we will begin to interiorize even the most advanced terms. We will start forming our own opinions, our own criteria.

Then, after a few weeks of being surrounded by the material, reflect on it. Do we feel excited about it? Does it make us want to understand it better? If so, we may have found something that makes us feel passionate about it. That is the best state of mind we can have when learning something new. If we are not enjoying the process, it may be a good indicator that we should explore some different topics.

Then, build small personal projects or contribute to an open source project. Talk about it to other developers. Have fun with it. Enjoy the process. Progressively, increase the complexity and depth of the material you consume.

But, above all, be compassionate with yourself. Know that you will not become an expert overnight. Becoming a better software developer is something that takes time and dedication. Don't be too hard on yourself, or it will become a weight more than an advantage.

Being compassionate with ourselves will give us the patience we need to endure the difficult parts of becoming excellent professionals. Like Rocky said to his son in the 2006 movie "*Rocky Balboa*": "*It is not about how hard you can hit but how hard you can get hit and keep moving forward.*"

Conclusion

Advancing in our career is a demanding yet rewarding process. Understanding what other people expect of us at each career level is critical.

Every developer has to start from a junior position. This role helps us get familiar with the software development process in a real-world scenario. At the beginning of our career, we are not expected to build entire distributed systems by ourselves. We must remain open to taking work that may seem unimportant at the time, as it is work that software development teams need to get done.

Transitioning to a senior developer position requires us to operate independently. To be senior developers, we not only have to be able to build quality into software applications, but also must take the role of a mentor for other, more junior developers. The title matters less than actually having the skills that make up a senior developer.

We must work on both hard and soft skills. Hard, technical skills are improved through a mix of practice and keeping ourselves updated on the best practices in the industry. We enhance our soft skills with a combination of self-introspection and openness to communication and collaboration. Both types of skills are required to advance in our careers, and ignoring one in favor of the other will limit the opportunities we will find in our professional future.

Finding our next job requires us to put ourselves in the position of recruiters and managers alike. Once we understand their motivations and practices to screen candidates for interviewers, we can build an attractive profile for recruiters and development teams.

The key to building an effective resume or professional profile is to keep things concise and clear. If we cannot state in just a couple of paragraphs what our achievements have been along our professional career is because we don’t fully understand them ourselves.

The technical interview is a lengthy process that requires preparation. We need to be able to write efficient code using the correct data structures and algorithms, and we should be able to evaluate how performant they are. We must be familiar with the software design process we described in the previous chapter. But above all, the most critical skills for any interview are problem-solving, efficient communication, and collaboration.

The interviews are an opportunity for employers to evaluate how good a fit a candidate is. Still, it is also an opportunity for candidates to gain insight into whether this prospective employer is a good fit for them.

Mentors are people who can help us progress in our careers. They lend us their experience to take a shorter path than the one they had to walk to get where they are. If we choose to have a mentor, we also must be open to their feedback.

While there are thousands of resources out there to keep improving ourselves, it is critical to understand ourselves and the way we learn better. We must be compassionate and patient with ourselves, as we will likely make mistakes and struggle to become better developers.

We have now reached the end of our book. A book’s content is only as valuable as the reader’s effort to put it into practice. The goal of this book is not to make you an expert at backend development, as no single book can achieve that. The goal is to give you enough tools to go into the wild and be able to defend yourself.

Keep yourself humble. The greatest software developer is the one who recognizes that there is always room for improvement, something new to learn.

I wish you well in your adventure ahead. It is going to be a wild yet gratifying ride.

Resources

- Information theory: **https://web.mit.edu/6.933/www/Fall2001/Shannon2.pdf**

- Asymptotic complexity: **https://www.cs.cornell.edu/courses/cs3110/2012sp/lectures/lec19-asymp/review.html**
- LeetCode: *leetcode.com*
- HackerRank: *hackerrank.com*
- Grokking the system design interview: **https://www.educative.io/courses/grokking-the-system-design-interview**
- Richard Feynman and explaining things in simple terms: **https://kottke.org/17/06/if-you-cant-explain-something-in-simple-terms-you-dont-understand-it**

Index

A

abstraction layer, client-server architecture
  data access service client, database server 26, 27
  frontend client and backend client 26
A/B testing 247
acceptance testing
  alpha 143
  beta 143
access control 181
  authentication 181
  authorization 185
  Pizza Place app use case 187
access control, Pizza Place app
  authentication, building 188
  authentication, implementing 187
  authorization controls, implementing 195
  authorization, implementing 187
  high-level authorization map, translating to implementation details 195-197
  password-related protections, defining 189-192
  scopes, using for authorization check 198
  testing 198
  user management storage, defining 188
  user roles, identifying 187
  web forms for signup and login, building 193-195
accessibility testing 174
actions 79
adapter pattern 67
Agile methodologies 148
API layer 20
application deployment
  history 295, 296
  isolated environments 299
  reproducible environments 288, 299
  robust deployment process 294
  scripts 296, 297
  shared environments, moving out of 298, 299
  version control 299-301
application performance
  improving 338
  improving, with asynchronous communication 358-360
  improving, with asynchronous programming 360, 361
  improving, with caching 339
  improving, with distributed systems 351

improving, with microservices 357
measuring 332, 333
measuring, with percentiles 337, 338
synthetic testing, versus RUM 333-336
Application Programming Interface (API) 61
design patterns 66, 67
functions, as contracts 61, 62
interfaces 62-66
uses 67
application queue 359
application security 177
access control 181
building 208
CIA triad 178
threat model, creating 208, 209
application testing
automated testing 149
certainty 140, 141
federated authorization 199
manual testing 141, 142
unit testing 150
asymptotic complexity 415
asynchronous communication
using 358
asynchronous programming 360, 361
Auth0 95
authentication 181
best practices, for passwords 182, 183
identification, username and password 182
multi-factor authentication 184
password issues 182
single sign-on (SSO) 184
authorization 185
groups 185, 186
least privilege principle 186
roles 185, 186
authorization service 20
automated testing 149, 150
and CI/CD 170, 171
coverage 162-164
in isolation 152
integration testing, with Selenium 166-169
static code analyzers 171
test cases 172
Test-Driven Development (TDD) 165
unit testing 150-152
availability 180
loss of 181

B

backup and recovery 125
   backups, creating with activity logs 126, 127
   backup, through replication 127
   database files, backing up 125, 126
   gaps, tackling 127, 128
binary files 107
binary search 123
Binary search trees (BSTs) 416
BitTorrent 55
Black box testing 142
Blockchain 55
brute-force attacks 182
B-trees 123
BufferedReader 32

C

cache 339
cache strategy
   least-frequently used (LFU) 346
   least-recently-used (LRU) 346
caching 339
caching tools
   CDNs 347
   proxies 347
career path
   expectations, from junior developer 402
   mentors, finding 420, 421
   resources, finding 421, 422
   senior developer skills 404
   technical interviews, preparing for 410
CDN (Content Delivery Network) 347
Certificate Authority (CA) 44
challenges, addressed by frameworks 257
   automation tools 258-262
   native package management 262, 263
   package managers 258
changelog 302
chaos testing 397
CIA triad
   availability 180
   confidentiality 178, 179
   integrity 179, 180
CI/CD pipeline
   advantages 297, 298
circuit breaker pattern 396
client 25
   versus, server computing 50
client-server architecture 24, 25
   abstraction layer 26

applying, to Pizza Place app 52-54
examples 26
HTTP 27, 28
versus, peer-to-peer architecture 54
web server, implementing 28
commit 302
Common Object Request Broker Architecture (CORBA) 68
complex data storage 109
document databases 110
file storage repositories 120, 121
graph databases 117
NoSQL 109, 110
relational databases 112, 113
scalability 119
SQL 109
technical characteristics 121, 122
confidentiality 178, 179
loss of 179
consistent hashing 390
containers 317, 318
Continuous Integration and Continuous Deployment (CI/CD) 170, 294
cookies 194
coverage 162
expanding 163
report 164
usage conditions 164, 165
curl 38
Cypher 119

D

database access, with ORMs 277
data, inserting 283
data, querying 283-286
JPA and Hibernate 277-279
tables, mapping to entities 279-283
Database Administrators (DBAs) 101
databases 109
database services 20
data model, for Pizza Place app
database scaling 136
data store, for menu storage 135, 136
defining 132
requirements 132-135
data storage, in production system
deployment strategy, selecting 129
designing 129
dependency injection 66
deployment strategies, data storage in production system
database deployment, in own server 130

databases, integrating 130, 131
embedded databases 130
server, sharing between database and application 129, 130
deployment strategy 129
deserializing 107
design patterns 254
dictionary attacks 182
digital transformation
and Internet 4, 5
history 2-4
Distributed Denial-of-Service (DDoS) 181
Docker 317
Docker Compose 320
document databases 110
high locality 111, 112
relationships 112
unstructured data 112
DREAD
affected users 210
damage 210
discoverability 210
exploitability 210
reproducibility 210

E

Elasticsearch 239
installing 244
ELK stack 238, 239
errors, logging with 239
end-to-end data management
application state, defining 102
backup and recovery 125
complex data storage 109
data storage, designing in production system 129
hardware storage 102, 103
indexing 122
in-memory data storage 103, 104
Pizza Place use case 132
error handling 216
A/B testing 247
deployment plan, creating 248
errors, as return values 222
exceptions 218
gradual deployment 247
types 217
use case 238
error handling, in distributed systems 237, 238
errors
common causes 216, 217

non-recoverable errors 216
recoverable errors 216
errors logging, with ELK stack 239
Elasticsearch, installing 244
Kibana dashboards, creating 245-247
Kibana, installing 244
locally 240, 241
Logstash, installing 241, 242
event-driven programming 28
eventual consistency 356
eviction policy 346
First-in-first-out (FIFO) 346
Last-in-First-Out (LIFO) 346
Random Replacement (RR) 346
exception handling
errors, preventing 223, 224
implementing 223
exceptions 218
catching 219, 220
errors, validated at compile time 220, 221
stack traces, using 218
types, defining 218, 219
expectations, from junior developers
defining 402
development team, joining 402, 403
guidance from other developers, following 404
Express 42
MVC 274-276

F

facade pattern 67
factory pattern 267
failure case, system design
cache layer node failure 395
database server failure 396
file storage service failure 395
infrastructure failure 396
microservice instance failure 395
service queue node failure 395
web server failure 394
failure management
considerations 396
fakes 161
federated authentication
OpenID 202
Security Assertion Markup Language (SAML) 201, 202
federated authorization 199, 200
advantages 201
disadvantages 201

file storage 20
file storage repositories 120
  features 121
  heuristics 121
File Transfer Protocol (FTP) 295
frameworks 251
  advantages 257
  benefits 257
  challenges, resolving 253
  debugging complexit, adding 290
  design patterns 254
  impact of community 288
  issues, fixing 252, 253
  learning 289
  libraries 254-256
  pre-building abstractions 256
  selecting 287, 288
  usage criteria 289
  zero-cost abstractions 290
functional requirements 10
functional tests 142
future-proofing 13
Futures 361

G

getInputStream method 32
Git 301
  branches 308-310
  commit details 307, 308
  conflicts, merging 312, 313
  existing file, modifying 305, 306
  file, committing 302-304
  file, staging 302-304
  hooks 313, 314
  merge feature 310, 311
  remote repositories 311
  repositories, cloning 312
  repository, creating 301, 302
  reproducible code, enforcing 314
Gradle 258
graph databases 117
  data, with connections 117, 118
  data, with relationships as first-class citizens 118, 119
GraphQL 90
  APIs, building with 89-93
GraphQL SDL model 90
Grey box testing 143
gRPC
  using 76, 77

Guava 259
gzip 41

H

hard disk drives (HDDs) 103
hard skills, software developer
  technical skills, improving 406, 407
hash maps 415
headless Content Management System (CMS) 94
high-fidelity prototypes 15
high-level design
  defining 381
  gaps, filtering in assumptions 383, 384
  use case flow, defining 382, 383
HTTP cookies 194
HTTP methods 81
HTTP response headers
  cache-control 87
  ETAG 87
  expires 86
HTTPS
  enabling, in Express 44-46
Hypermedia as the Engine of Application State (HATEOAS) 88, 89
Hypertext Application Language (HAL) 88
Hypertext Transfer Protocol (HTTP) 27, 28
hypervisor 316

I

idempotency 82
identity provider (IP) 199
indexing 122
  example 124
  impact of index, benchmarking in SQLite 124
  time complexity, reducing 122, 123
Industrial Revolution 3
in-memory data storage 103, 104
  in binary files 107, 108
  in-memory cache 104-106
  in-memory databases 106
  in text files 107, 108
integration testing 142, 166
integration testing, with Selenium 166-169
  test environment, defining 170
  test environment, simulating 170
integrity 179, 180
  loss of 180
interfaces 62
Internet 4

Internet of Things (IoT) devices 18
Inversion of control (IoC) 264
IOException 220

J

Java ARchive (JAR) files 258
Java Database Connectivity (JDBC) 26
Java.lang.Error 219
java.lang.Exception 219
java.lang.Throwable 219
Java Stream API 32
Jedis
  using 348, 349
JPA (Jakarta Persistence API) 277

K

Kafka 360
Kibana 239
  dashboards 245
  installing 244
Kubernetes 319

L

layered architecture 49
  business/services layer 48
  client, splitting from server 46-51
  persistence layer 48
  presentation layer 48
LetsEncrypt 44, 45
load testing 174
localhost 30
logging 230
logging services 20
Logstash 239
  filter attribute 243
  input attribute 242, 243
  installing 241, 242
  output attribute 244
low-fidelity prototypes 15
  example 16
low-level design
  client, designing 384, 385
  database layer, designing 388, 389
  defining 384
  distributed database strategy, defining 389, 390
  file storage, scaling 391
  read and write replicas, defining 391

service layer, designing 386-388
web server, designing 385, 386

M

manual testing cycle
detailed reports, creating 145, 146
fixes, validating 146
test plan, creating 144, 145
test plan, executing 145
test plan, updating 146
manual tests
acceptance testing 143
advantages 146
and Agile 147
Black box testing 142
building 143, 144
code testing 149
explorability 146
Grey box testing 143
horizontal validation 147
integration testing 142
performing 141, 142
system testing 143
types 142
unit testing 142
user-centered vision 147
White box testing 143
Maven 258, 262
memoization 340
mental model 29
microservices 357
adoption of better tools, encouraging 357
benefits 357
horizontal scaling, enabling 357
parallel collaboration, enabling 357
mocking 140
mocks 161
Model-View-Controller (MVC) pattern 263, 264
limitations 276
modern system design
building blocks 20
front-end 18, 19
high-level view 17
multi-factor authentication 184
multi-user apps
backend developer 7
designing, for solving real-world problems 6, 7
problems, resolving 8, 9
sample use case 9

N

Nginx 320
NodeJS Express server
  HTTPS, enabling 44-46
  using 41-43
non-functional requirements 10
non-functional testing 142, 173
  accessibility testing 174
  application security and penetration testing 173
  load testing 174
  performance testing 174
non-recoverable error 217
NullPointerException 220, 221

O

OASIS Security Services Technical Committee (SSTC) 201
OAuth 95
OAuth2 203
  access token, requesting 205, 206
  authorization code, requesting 204, 205
  code flow 206
  implicit flow 206, 207
  OpenID, building from 207, 208
  terms 203
Object-Oriented Programming (OOP) 254
object-relational mapping (ORM) frameworks 277
  disadvantages 287
  Python's Django ORM 286, 287
Open Database Connectivity (ODBC) 26
OpenID 202
  OAuth2 203
Open Web Application Security Project (OWASP) 211
  top 10 vulnerabilities 211, 212
orchestration tools 319

P

packet sniffer 44
paid certificates 44
Pareto principle 223
password recovery 193
peer-to-peer (P2P) architectures 54, 55
penetration testing 173
people skills. See soft skills
percentiles 338
performance 332
performance improvement, with caching 339, 340
  cache-aside 342

- cache patterns 341
- caching strategy, selecting 345
- eviction policy 346
- reading and writing load, defining 341
- read-through cache 342, 343
- write-behind cache 344
- write-heavy applications, caching 343
- write-through cache 343, 344

performance improvement, with distributed systems 351-353
- data consistency, in replicas 353, 354
- data consistency, in sharding 356, 357
- data consistency, maintaining 353
- eventual consistency, versus strong consistency 355, 356
- multiple read replicas 355
- single write replica 355
- through replication 352
- through sharding 353

performance testing 174

Pizza Place app 9, 10
- client-server architecture, applying 52-54
- data model, defining 132
- defining 9, 10
- remote API, generating for 96-98
- reproducible deployment environment, creating with Git and Docker 319, 320
- requirements, defining 16, 17

Platform-as-a-Service (PaaS) 317

ports 30

procedural programming 28

production errors, handling
- anatomy, of log entry 231, 232
- error, propagating 229
- errors, bubbling up 224-226
- errors, identifying with logging 230, 231
- errors, mitigating 224
- fallback, providing 227-229
- good error messages, defining 224
- good log and error messages, designing 233, 234
- log entries, persisting to file 234-237
- logging 233

production-ready server
- using 41-43

professional skills, software developer
- hard skills, improving 406
- improving 406
- people skills, improving 408
- soft skills, improving 406

programming interfaces 69

Promises
- using 361-364

proto3 71

protobuf 71
proxies 347
proxy pattern 67
public APIs 95, 96
Python's Django ORM 286, 287

Q

queries per second (QPS) 174
queue 20

R

RabbitMQ 360
Random Access Memory (RAM) 103
Real User Monitoring (RUM) metrics 336
recoverable error 216
Redis 347
  as distributed cache 106
  as distributed session management 106
  as message broker 107
  long-running operations, caching with 347, 348
Redisson 349
  using 349-351
regressions 141
Relational Database Management System (RDBMS) 26
relational databases 112, 113
  fixed structure 114
  independent models 114
  multiple relationships 114
  normalization 115-117
  versus, graph databases 119
remote API 67
  building, with GraphQL 89-93
  building, with RPC/gRPC 68-71
  gRPC client, building 74-76
  gRPC server, building 71-74
  Remote Procedure Call (RPC) 67, 68
Remote Method Invocation (RMI) 68
Remote Procedure Call (RPC) 67, 68
remote repositories 311
replication 127
representational state transfer (REST) 79
reproducible deployment environment, Pizza Place app
  application, setting up 320
  creating, with Git and Docker 319, 320
  docker-compose.yml 324-326
  Docker in CI/CD 327, 328
  Git, adding 327
  nginx.conf 325

nginx directory 323
trade-off 328, 329
web directory 320-322
requirement definition cycle
client's domain-experts help, requesting 14
detailed picture of business, obtaining 13, 14
domain-expert work 14, 15
heuristics 12
problem, identifying 12, 13
prototypes, building 15, 16
requirements, defining for Pizza Place use case 16, 17
requirements
collecting 11
functional requirements 10
non-functional requirements 10
REST APIs
actions, versus HTTP request methods 81-83
building 79-81
caching 86, 87
effective REST APIs 87-89
naming resources 83
relationships 84, 85
singular, versus plural 83, 84
versioning 85, 86
robust deployment process
defining 294

S

scalability 119
horizontal, versus vertical scalability 119, 120
Second Industrial Revolution 3
Security Assertion Markup Language (SAML) 201, 202
self-signed certificates 45
sendResponse function 36
senior software developer
characteristics 404-406
skills 404
separation of concerns pattern 48
serializing 107
servers 25
shard 353
sidecar pattern 396
Simple Object Access Protocol (SOAP) 77, 78
single sign-on (SSO) 184
Six Sigma 223
socket 30
soft skills, software developer
communication skills, working on 408, 409
conflict resolution, practicing 408

improving 408
software application tester 141
Software-as-a-Service (SaaS) approach 95
software developer
professional skills, improving 406
technical interviews, preparing 410
Software Development Life-cycle (SDLC) methodologies 147
software transformation 5, 6
source of truth 105
spies 161
Spring 264
annotation-based configuration 270
code-based configuration 269
dependency injection 265-268
MVC 264-274
XML-based configuration 268, 269
Spring WebFlux 274
SQLite
impact of indexes, benchmarking 124
stack trace 218
staging 303
standalone APIs
building 93
headless CMS 94
public APIs 95, 96
stateless containers
working with 318, 319
static code analysis 220
static code analyzers 171
static web applications 206
STRIDE
denial of service 210
elevation of privilege 210
information disclosure 209
repudiation 209
spoofing identity 209
strong consistency 355
stubs 161
Subversion (SVN) 314, 315
synthetic testing 334
for CI/CD validation 335
for regression testing 335
system design process 368
cache size, calculating 379, 380
client to server 391, 392
data models, defining 373, 374
failure points, identifying 393, 394
high-level and low-level designs, integrating 391
high-level design, defining 381
interface, defining 372, 373

low-level designs, creating 381
low-level designs, defining 384
overall low-level design 392
Pizza Place app example 369
queries per second (QPS), estimating 378, 379
requirements, defining 370-372
scale and size, calculating 374, 375
services to database 392
size estimation, stopping 380
storage size, estimating 375-378
storage throughput, estimating 379
web server to services 392
system testing 143

T

TCP/IP 30
technical interview process 413
algorithms, using 416
coding interviews 417
coding problems, approaching 414, 415
data structures, using 415, 416
non-coding modules, approaching 419, 420
questions to ask 420
solution, coding 416, 417
system design interviews 418
take-home assignments 419
technical interviews
interview skills, improving 412, 413
networking, importance of 412
preparing 410
resume, building 411
scheduling 410
test cases
conditions 173
mock test, as needed 172
single use case per test, defining 172
test doubles 161
Test-Driven Development (TDD) 165
testing 139
functional tests 142
negative tests 142
non-functional tests 142
positive tests 142
testing in isolation 152
isolated code testing 152-156
mock tests 159-161
stub, versus mock 161
test dependencies, isolating 156-158
test plans 144

test scripts 144
text files 107
threat modeling 174
Threat Modeling process 208, 209
   application, decomposing 209
   countermeasures, determining 210
   mitigation, determining 210
   threats, determining 209
   threats, ranking 209, 210
time complexity 123
time-to-live (TTL) 343
traditional CMS
   versus, headless CMS 94

U

unchecked exceptions 221
unit testing 142

V

version control systems (VCS) 300
   Git 301
   SVN 314
Virtual Machines (VMs) 20, 315, 316
   versus, containers 318
   virtual infrastructure 316, 317

W

Waterfall methodology 147, 148
Web Application Resource (WARs) 258
web requests, handling 263
   Spring framework 264
web server
   as stateless services 51
   implementing 28, 29
   main process 29-33
   multi-user support, with multi-threading 38-41
   response, serving 33-38
   session data, storing 52
Web Services Description Language (WSDL) file 77
White box testing 143