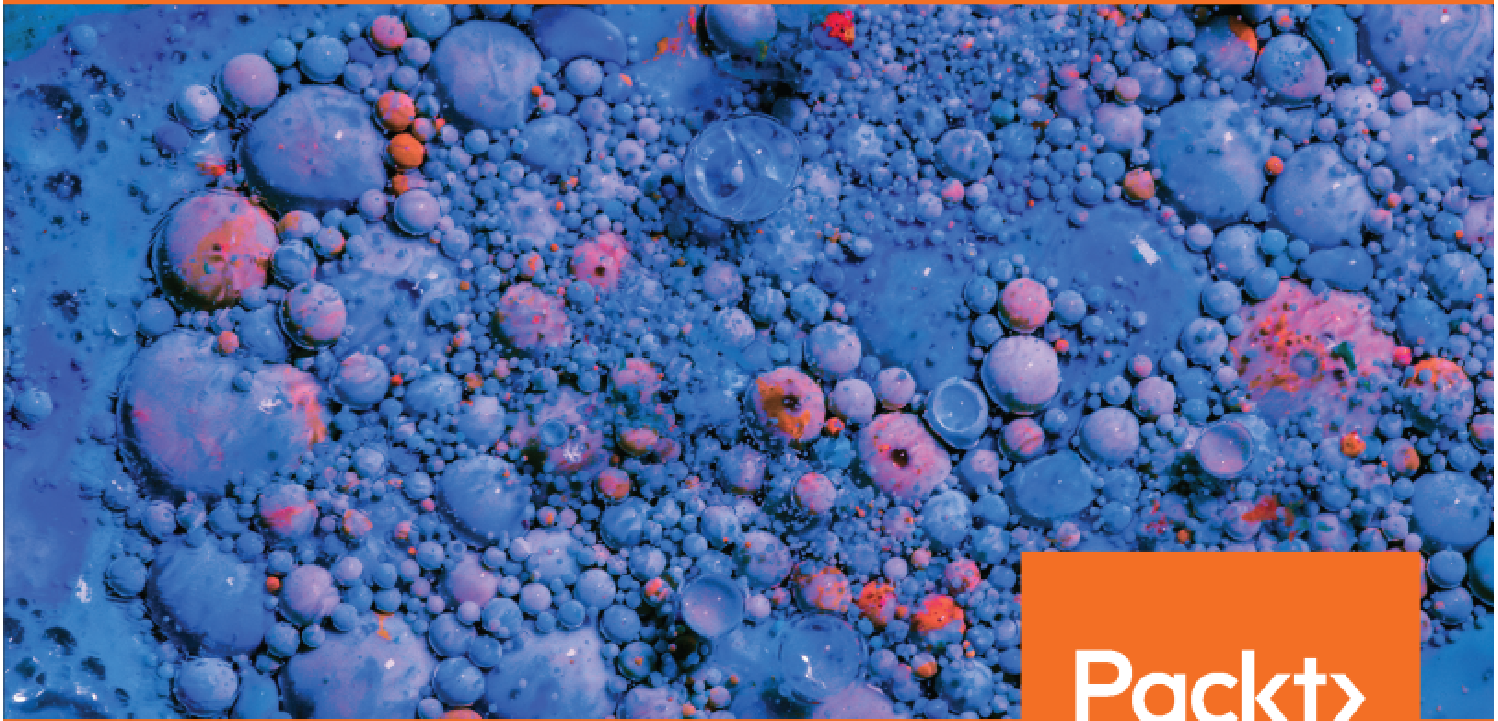
Hands-On Microservices with Rust

Build, test, and deploy scalable and reactive microservices with Rust 2018



Packt

www.packt.com

Denis Kolodin

Hands-On Microservices with Rust

Build, test, and deploy scalable and reactive microservices with Rust 2018

Denis Kolodin

Packt

BIRMINGHAM - MUMBAI

Hands-On Microservices with Rust

Commissioning Editor: Richa Tripathi
Acquisition Editor: Shririam Shekhar
Content Development Editor: Manjusha Mantri
Technical Editor: Mayank Dubey
Copy Editor: Safis Editing
Language Support Editors: Mary McGowan and Safis Editing
Project Coordinator: Prajakta Naik
Proofreader: Safis Editing
Indexer: Priyanka Dhadke
Graphics: Jisha chirayil
Production Coordinator: Tom Scaria

First published: January 2019

Production reference: 1310119

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78934-275-8

`www.packtpub.com`

Contributors

About the author

Denis Kolodin has been developing high-loaded network applications for more than 12 years. He has mastered and used different kinds of programming languages, including C, Java, and Python, for developing a variety of apps, from high-frequency trading robots to video broadcasting servers. Nowadays, he enjoys creating peer-to-peer networking applications and is inspired by distributed systems such as cryptocurrencies.

He has been using Rust since version 1.0 and is delighted with the features it provides, specifically WebAssembly support. He's the author of *Yew Framework*, which collected more than 6,000 stars on GitHub. He writes with Rust continuously as part of his job and believes that Rust will be used for everything in the future, including backend, frontend, operating systems, embedded software, games, and smart contracts.

About the reviewer

Daniel Durante is an avid coffee drinker/roaster, motorcyclist, archer, welder, and carpenter whenever he isn't programming. From the age of 12, he has been involved with web and embedded programming with PHP, Node.js, Golang, Rust, and C.

He has worked on text-based browser games that have reached over 1,000,000 active players, created bin-packing software for CNC machines, embedded programming with cortex-m and PIC circuits, high-frequency trading applications, and helped contribute to one of the oldest ORMs of Node.js (SequelizeJS).

He has also reviewed other books – *PostgresSQL Developer's Guide*, *PostgreSQL 9.6 High Performance*, *Rust Programming By Example*, and *Rust High Performance* – for Packt.

> *I would like to thank my parents, my brother, my mentors, and my friends who've all put up with my insanity of sitting in front of a computer day in and day out. I would not be here today if it wasn't for their patience, guidance, and love.*

Gaurav Aroraa has completed his M.Phil in computer science. He is an MVP, a lifetime member of the **Computer Society of India** (**CSI**), an advisory member of IndiaMentor, and is certified as a scrum trainer/coach, XEN for ITIL-F, and APMG for PRINCE-F and PRINCE-P. Gaurav is an open source developer, and the founder of Ovatic Systems Private Limited. Recently, he was awarded the title "Icon of the year – excellence in mentoring technology startups" for the year 2018-19 by Radio City, A Jagran Initiative, for his extraordinary work during his 20-year career in the industry in the field of technology mentoring. You can tweet Gaurav on his twitter handle: @g_arora.

Table of Contents

Preface 1

Chapter 1: Introduction to Microservices 7
- **Technical requirements** 7
- **What are microservices?** 8
  - Why we need microservices 8
  - How to deploy a microservice 9
    - Docker 10
    - Continuous Integration 10
- **How to split a traditional server into multiple microservices** 11
  - Reasons to avoid monoliths 11
    - Impossible to scale vertically 12
    - Impossible to update and deploy only one feature 12
    - The failure of one server affects all features 12
  - Breaking a monolithic service into pieces 13
    - Definition of a REST API 14
      - User registration microservice 15
      - E-mail notifications microservice 15
      - Product catalog microservice 15
      - Shopping cart microservice 16
      - Payment Integration microservice 16
      - Statistics collecting microservice 16
  - Transformation to microservices 17
    - Reusing existing microservices 17
- **Why Rust is a great tool for creating microservices** 18
  - Explicit versus implicit 18
  - Minimal amount of runtime errors 19
  - Great performance 20
  - Minimal dependencies burden 20
- **Summary** 21
- **Further reading** 22

Chapter 2: Developing a Microservice with the Hyper Crate 23
- **Technical requirements** 23
- **Binding a Tiny Server** 24
  - Adding necessary dependencies 24
  - The main function of the server 24
    - Address of the server 25
    - Server instances 25
    - Setting the requests handler 26
    - Adding the server instance to a runtime 26
  - Building and running 27

Rebuilding on changes 27
Handling incoming requests 28
Adding a service function 28
Implementing a service function 29
Index pages 31
Implementing the REST principles 33
Adding a shared state 33
Accessing a shared state from a service function 35
Parsing paths in a microservice 36
Implementing REST methods 37
Extracting the user's identifier 37
Getting access to the shared data 38
REST methods 39
POST – Creating data 39
GET – Reading data 40
PUT – Updating data 41
DELETE – Deleting data 41
Routing advanced requests 42
Defining paths with regular expressions 42
Adding the necessary dependencies 43
Writing regular expressions 43
Path for index page 44
Path for user management 45
Path for the users list 45
Matching expressions 46
Summary 48
Chapter 3: Logging and Configuring Microservice 49
Technical requirements 49
Adding logging to a microservice 50
Random-value-generating microservices 50
The log crate 51
Loggers 52
Log levels 53
Logging messages 54
Custom level of messages 54
Checking logging is enabled 54
Own target 55
Using logging 55
Configuring a logger with variables 56
RUST_LOG 56
RUST_LOG_STYLE 57
Changing the RUST_LOG variable to your own 57
Reading environment variables 57
Standard library 58
Using the .env file 59
Using the dotenv crate 59
Adding variables to the .env file 60

Parsing command-line arguments 60
Using the clap crate 61
Adding dependencies 61
Building a parser 61
Reading arguments 63
Usage 63
How to add subcommands 64
Reading the configuration from file 65
Adding the TOML config 65
Adding dependencies 66
Declaring a struct for configuration 66
Reading the configuration file 67
Joining all values by a priority 68
Creating and using the configuration file 69
Summary 69
Chapter 4: Data Serialization and Deserialization with the Serde Crate 70
Technical requirements 70
Data formats for interaction with microservices 71
The serde crate 71
Serializing responses 72
Deserializing requests 73
Tweaking serialization 74
Changing the case of names 74
Removing a nesting 75
Using specific names for a tag and content 76
Any Value 76
Using hyper 77
Reading a body from a stream 77
Custom types 80
Custom serialization 80
Custom deserialization 82
Custom error types with the failure crate 83
Binary data 84
Compiling, running, and testing 87
Microservices with multiple formats 89
Different formats 89
Parsing a query 90
Checking different formats 91
Transcoding 92
XML support 92
Summary 93
Chapter 5: Understanding Asynchronous Operations with Futures Crate 94
Technical requirements 95
Basic asynchronous types 95
Basic types of future crate 96
Using the Future trait 96

Using the Stream trait 97
Using Sink to send data back 97
The channel module 98
Channels for sending multiple messages 98
Single-Producer Single-Consumer 98
Multi-Producer Single-Consumer 99
Multi-Producer Multi-Consumer 99
Example of usage 99
One-shot 100
Using channels to use Sink in multiple places 101
Executors 103
Running futures and streams with blocking 103
Using an executor 103
The async/await syntax 104
Creating an image service 105
Uploading images 105
The tokio crate 108
Asynchronous input/output of files 109
Multipart form requests 110
Downloading images 110
sendfile for sending files 112
Testing the service 112
Summary 113

Chapter 6: Reactive Microservices - Increasing Capacity and Performance 114
Technical requirements 114
What is a reactive microservice? 115
Loose coupling 115
Message-driven applications 116
Asynchronous 116
Should a reactive microservice be asynchronous? 117
Reactive microservices with futures and streams 117
Message brokers 118
Remote procedure calls 118
JSON-RPC 118
gRPC 118
Thrift 118
Other RPC frameworks 119
RPC and REST 119
Reactive manifesto 119
Understanding JSON-RPC 119
How JSON-RPC works 120
Creating a microservice 120
Dependencies 121
Client 122
Worker 124
Server 125
Compiling and running 127

Learning about gRPC 128
How gRPC works 128
Creating a microservice 128
Dependencies 129
Protocol 130
Generating interfaces 131
Shared client 132
Client 134
Server implementation 134
Service implementation 135
Handlers 136
The main function 137
Worker 138
Compiling and running 138
Summary 139
Chapter 7: Reliable Integration with Databases 140
Technical requirements 140
PostgreSQL 141
Setting up a test database 141
Simple interaction with a database 143
Adding dependencies 143
Creating a connection 145
Wrapping with a tool 146
Compiling and running 149
Connection pools 150
Creating a pool 151
Parallel imports with the rayon crate 153
Rayon 154
MySQL 154
Database for tests 155
Connecting with the r2d2 adapter 156
Adding dependencies 156
Database interaction functions 157
Creating a connection pool 158
Redis 160
Bootstrap database for tests 160
Creating a connection pool 161
Dependencies 161
Adding commands and interaction functions 162
Data manipulation functions 163
Parsing arguments 164
Connecting to Redis 165
Executing subcommands 165
Testing our Redis example 166
MongoDB 167
Bootstrapping a database for testing 167
Connecting to a database using the r2d2 pool 168

Interaction functions 169
Parsing arguments 170
Creating a connections pool 171
Implementing subcommands 172
Testing 173
DynamoDB 174
Bootstrapping a database for testing 174
Connecting to DynamoDB 176
Adding dependencies 176
Interaction functions 178
Parsing command-line arguments 182
Testing 185
Summary 186
Chapter 8: Interaction to Database with Object-Relational Mapping 187
Technical requirements 187
The diesel crate 188
Adding the necessary dependencies 188
diesel_cli 188
Creating migrations 189
Declaring the data structure 190
Models 191
Connecting to a database 192
Parsing arguments 192
Creating a connection 193
Implementing subcommands using a DSL 194
Adding a user subcommand implementation 194
Listing users subcommand implementation 195
Testing 196
Complex database structure 196
Business logic of the example application 196
API methods 198
Database structure and migrations 198
Diesel initial setup 199
up.sql 199
down.sql 200
Users table 200
up.sql 200
down.sql 201
Channels table 201
up.sql 201
down.sql 202
Memberships table 202
up.sql 202
down.sql 202
Messages table 203
up.sql 203
down.sql 203
Schema 203

Models 205
User 205
Channel 206
Membership 207
Message 207
Database interaction API crate 208
Api 209
Register user 209
Create channel 210
Publish channel 211
Add member 212
Add message 213
Delete message 213
Testing the crate 214
Summary 215

Chapter 9: Simple REST Definition and Request Routing with Frameworks 216
Technical requirements 216
Rouille 217
Creating a microservice 217
Bootstrapping a server 217
Handling requests 218
Root handler 218
Sign-up handler 218
Sign-in handler 220
Default handler 221
Compiling and running 221
Nickel 222
Creating a microservice 222
Bootstrapping a server 222
Handling requests 224
Worker for sending emails 225
Compiling and running 226
Rocket 227
Creating a microservice 227
Bootstrapping a server 227
Handling requests 229
Database schema and models 230
Compiling and running 231
Gotham 232
Creating a microservice 233
Types of framework 233
The main function 235
Handler implementation 238
Running and testing 240
Summary 242

Chapter 10: Background Tasks and Thread Pools in Microservices 243

Technical requirements 243
Interacting with threads 244
Synchronous or asynchronous? 244
IO-bound versus CPU-bound tasks 244
Synchronous tasks inside asynchronous contexts 244
The limitations of using IO operations on files 245
Spawning a thread for image processing 246
Interacting with a thread in an asynchronous way 248
Using thread pools 251
CpuPool 251
The blocking section 252
Actors 254
Basics of the actix framework 255
Implementing actors 255
The count actor 256
Types 256
Actor 256
Message 257
Handler 257
The log actor 258
Types 258
Actor 259
Message 260
Handler 260
The resize actor 260
Types 261
Actor 261
Message 261
Handler 262
Using the server with actors 262
Requests handler 264
Building and running 266
Summary 267
Chapter 11: Involving Concurrency with Actors and the Actix Crate 268
Technical requirements 268
Actor concurrency in microservices 268
Understanding actors 269
Actors in microservices 269
The Actix framework 270
Creating a microservice with actix-web 270
Bootstrapping an actix-web server 270
Starting a server 270
App creation 271
Scope and routes 272
Static files handler 273
HTTP client 273
GET requests 274
POST requests 275

Handlers 276
Signup 276
Signin 277
New comment 278
Comments 280
Counter 280
Middleware 280
Building and running 282
Using databases 282
The database interaction actor 282
Actors 283
Messages 283
Setting a value message 283
Get value message 284
Link to actor 285
Using database actors 286
WebSocket 289
Repeater actor 289
Actors 289
Messages 290
Updating the message 290
Control message 291
The notification actor 292
Actor 293
Handlers 295
Adding WebSocket support to a server 296
Summary 298
Chapter 12: Scalable Microservices Architecture 299
Technical requirements 299
Scalable architecture 300
Basic ideas 300
Message brokers and queues 300
RabbitMQ 301
Kafka 301
Application bottlenecks 301
Scalable application with Rust and RabbitMQ 302
Bootstrap message broker for testing 302
Dependencies 303
Abstract queue interaction actor 305
Dependencies 305
Abstract messages handler 306
Actor 307
Handling an incoming stream 309
Sending a new message 310
Utility methods 311
Crate 313
Request and response 315

Worker 316
Dependencies 316
Handler 316
main function 318
Server 318
Dependencies 318
Shared state 319
Server handler 320
Requests handlers 321
Index handler 321
Tasks handler 322
Upload handler 323
main function 324
Testing 326
How to scale this application 328
Summary 328

Chapter 13: Testing and Debugging Rust Microservices 329
Technical requirements 329
Testing microservices 330
Unit testing 330
Mocking 330
Starting a server for testing 332
Making requests 334
Implementing tests 335
Running our tests 336
Integration testing 337
Starting an application instance 338
Dependencies 338
Utils 339
The integration testing client 340
Types 343
Users 344
Content 345
Mailer 346
Router 347
Debugging microservices 349
curl 349
Postman 350
mitmproxy 352
LLDB 354
Visual Studio Code 356
Structural logging 358
Example 358
Building and testing 360
Distributed tracing 361
Starting Jaeger 362
Generating spans 362

Compile and run 364
Summary 366
Chapter 14: Optimization of Microservices 367
Technical requirements 367
Performance-measuring tools 368
Welle 368
Drill 369
Measuring and optimizing performance 370
Basic example 370
Performance 373
Optimizations 374
State sharing without blocking 375
Reusing values by references 376
Caching 377
Compilation with optimizations 378
Optimization techniques 380
Link-time optimizations 380
Abort instead of panicking 381
Reducing the size of binaries 381
Isolated benchmarks 382
Profiling 383
Summary 385
Chapter 15: Packing Servers to Containers 387
Technical requirements 387
Building a Docker image with a microservice 388
Creating an image with the Rust compiler 388
Users microservice image 391
.dockerignore 391
Dockerfile 392
Building an image 393
Starting a container 395
Content microservice image 395
Email microservice image 396
Router microservice image 397
DBSync worker image 397
Dependencies 398
The main function 398
Hiding microservice source code 400
Composing a microservice set 401
Application definition 402
Database container 402
A container with an email server 403
DBSync worker container 403
Mails microservice container 404

Users microservice container 405
Content microservice container 405
Router microservice container 406
Running the application 406
Adding persistent state to the application 408
Running the application in the background 409
Summary 410

Chapter 16: DevOps of Rust Microservices - Continuous Integration and Delivery 411
Technical requirements 411
Continuous integration and continuous delivery 412
Continuous integration 412
Continuous delivery 413
Container orchestration 414
Rust tools 414
Rustfmt 415
Installation 415
Usage 415
Configuration 416
Clippy 417
Installation 417
Usage 418
Configuration 419
Recommended code attributes 419
Rustfix 420
Installation 420
Usage 420
Cargo test 421
CI and CD tools 421
TravisCI 422
AppVeyor 423
Jenkins 424
Demonstration of continuous integration 425
Docker Compose 426
The SCM server 426
The CI server 426
The CI agent 427
The image 427
Configuring Gogs 428
Configuring TeamCity 431
Authorizing agents 434
Creating a project 435
Building steps for Rust 437
Building with CI 440
Summary 442

Chapter 17: Bounded Microservices with AWS Lambda 443
Technical requirements 443
Serverless architecture 444
AWS Lambda 444
Azure Functions 445
Cloudflare Workers 445
IBM Cloud Functions 445
Google Cloud Functions 446
Minimal Rust microservices for AWS Lambda 446
Dependencies 446
Developing a microservice 447
Building 449
Deployment 450
Serverless Framework 455
Preparation 455
Implementation 456
Dependencies 456
Handler 457
Request and response types 459
Functions 461
Configuration 463
Resources 464
Deployment 468
Permissions 468
Script 469
Running 470
Testing 472
Updating and removing 478
Summary 478
Other Books You May Enjoy 479
Index 482

Preface

This book will introduce you to the development of microservices with Rust. I started using Rust not that long ago, back in 2015. It had only been a couple of months since the release of version 1.0 and, at that time, I didn't think that this tool would usher in a silent revolution that would disrupt the traditions associated with system programming, which, at that time, was tedious and in no way fashionable.

Maybe I'm exaggerating a little, but I have witnessed how companies stopped using the customary tools and began rewriting parts of their products or a number of services in Rust, and they were so happy with the outcome that they continue to do so time and again. Today, Rust is an important part of blockchain initiatives, the flagship for WebAssembly, and is an awesome tool for developing fast and reliable microservices that utilize all available server resources. Consequently, Rust has transformed itself from a hobby tool for curious developers into a strong foundation for modern products.

In this book, we will learn how to create microservices using Rust. We begin with a short introduction to microservices, and discuss why Rust is a good tool for writing them. Then, we will create our first microservice using the `hyper` crate, and learn how to configure microservices and log activities. After that, we will explore how to support different formats of requests and responses using the `serde` crate.

Who this book is for

This book is designed for two categories of reader—experienced Rust developers who are new to microservices, and advanced microservice developers who are new to Rust. I've tried to cover the ecosystem of useful tools and crates available for Rust developers today. This book describes the creation of microservices, from high-level frameworks to constructing low-level asynchronous combinators that produce responses with minimal resource blocking time. This book aims to allow you to find the solution to a specific task.

To be able to understand the topics covered in this book, you need a solid background in the Rust programming language (you should be able to write and compile applications using `cargo`, understand lifetimes and borrowing concepts, know how traits work, and understand how to use reference counters, mutexes, threads, and channels). If you are unfamiliar with Rust, take the time to understand these concepts before reading this book.

You also have to know how to write a minimal backend working on an HTTP protocol. You have to understand what REST is, and how to use it for applications. However, you don't have to understand how HTTP/2 works because we will use crates that provide abstractions agnostic to specific transport.

What this book covers

`Chapter 1`, *Introduction to Microservices*, introduces you to microservices and how they can be created with Rust. In this chapter, we also discuss the benefits of using Rust for creating microservices.

`Chapter 2`, *Developing a Microservice with Hyper Crate*, describes how to create microservices with the `hyper` crate, thereby allowing us to create a compact asynchronous web server with precise control over incoming requests (method, path, query parameters, and so on).

`Chapter 3`, *Logging and Configuring Microservices*, includes information about configuring a microservice using command-line arguments, environment variables, and configuration files. You will also see how to add logging to your projects, since this is the most important feature for the maintenance of microservices in production.

`Chapter 4`, *Data Serialization and Deserialization with Serde Crate*, explains how, in addition to customary HTTP requests, your microservice has to support formal requests and responses in a specific format, such as JSON, and CBOR, which is important for API implementation and in terms of organizing the mutual interaction of microservices.

`Chapter 5`, *Understanding Asynchronous Operations with Futures Crate*, delves into the deeper asynchronous concepts of Rust and how to use asynchronous primitives for writing combinators to process a request and prepare a response for a client. Without a clear understanding of these concepts, you cannot write effective microservices to utilize all available resources of a server, and to avoid the blocking of threads that execute asynchronous activities and require special treatment with execution runtime.

`Chapter 6`, *Reactive Microservices – Increasing Capacity and Performance*, introduces you to a reactive microservice that won't respond immediately to incoming requests, and that takes time to process a request and response when it's done. You will become familiar with remote procedure calls in Rust and how to use the language so that microservices can call one another.

`Chapter 7`, *Reliable Integration with Databases*, shows you how to interact with databases using Rust. You will get to know crates that provide interaction with databases, including MySQL, PostgreSQL, Redis, MongoDB, and DynamoDB.

`Chapter 8`, *Interaction to Database with Object-Relational Mapping*, explains how, in order to interact with SQL databases effectively and map database records to native Rust structs, you can use **object-relational mapping** (**ORM**). This chapter demonstrates how to use diesel crates which require nightly compiler version and whose capabilities are used for generating bindings with tables.

`Chapter 9`, *Simple REST Definition and Request Routing with Frameworks*, explains how, in certain cases, you don't need to write stringent asynchronous code, and that it is sufficient to use frameworks that simplify microservice writing. In this chapter, you will become acquainted with four such frameworks—rouille, nickel, rocket, and gotham.

`Chapter 10`, *Background Tasks and Thread Pools in Microservices*, discusses multithreading in microservices and how to use pools of threads to perform tasks on a background, given that not every task can be performed asynchronously and requires a high CPU load.

`Chapter 11`, *Involving Concurrency with Actors and Actix Crate*, introduces you to the Actix framework, which uses the actor's model to provide you with abstractions that are easily compatible with Rust. This includes the balance of performance, the readability of the code, and task separation.

`Chapter 12`, *Scalable Microservices Architecture*, delves into an explanation of how to design loose-coupling microservices that don't need to know about sibling microservices, and that use message queues and brokers to interact with one another. We will write an example of how to interact with other microservices using RabbitMQ.

`Chapter 13`, *Testing and Debugging Rust Microservices*, explains how testing and debugging is a key component in terms of preparing for the release of microservices. You will learn how to test microservices from unit tests to cover a full application with integration tests. Afterward, we will then discuss how to debug an application using debuggers and logging capabilities. Also, we will create an example that uses distributed tracing based on the OpenTrace API – a modern tool for tracking the activities of complex applications.

`Chapter 14`, *Optimization of Microservices*, describes how to optimize a microservice and extract the maximum performance possible.

`Chapter 15`, *Packing Servers to Containers*, explains how, when a microservice is ready for release, there should be a focus on packing microservices to containers, because at least some microservices require additional data and environments to work, or even just to gain the advantage of fast delivery containers over bare binaries.

`Chapter 16`, *DevOps of Rust Microservices - Continuous Integration and Delivery*, continues with the theme of learning how to build microservices and explains how to use continuous integration to automate building and delivery processes for your product.

`Chapter 17`, *Bounded Microservices with AWS Lambda*, introduces you to serverless architecture, an alternative approach to writing services. You will become acquainted with AWS Lambda and you can use Rust to write fast functions that work as a part of serverless applications. Also, we will use the Serverless Framework to build and deploy the example application to the AWS infrastructure in a fully automated manner.

To get the most out of this book

You will require at least version 1.31 of Rust. Install it using the `rustup` tool: `https://rustup.rs/`. To compile examples from some chapters, you will need to install a nightly version of the compiler. You will also need to install Docker with Docker Compose to run containers with databases and message brokers to simplify the testing of example microservices from this book.

Download the example code files

You can download the example code files for this book from your account at `www.packt.com`. If you purchased this book elsewhere, you can visit `www.packt.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packt.com`.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Hands-On-Microservices-with-Rust`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://www.packtpub.com/sites/default/files/downloads/9781789342758_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Mount the downloaded `WebStorm-10*.dmg` disk image file as another disk in your system."

A block of code is set as follows:

```
let conn = Connection::connect("postgres://postgres@localhost:5432",
TlsMode::None).unwrap();
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
#[derive(Deserialize, Debug)]
struct User {
    name: String,
    email: String,
}
```

Any command-line input or output is written as follows:

```
cargo run -- add user-1 user-1@example.com
cargo run -- add user-2 user-2@example.com
cargo run -- add user-3 user-3@example.com
```

Bold: Indicates a new term, an important word, or words that you see on screen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select **System info** from the **Administration** panel."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packt.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

1
Introduction to Microservices

This chapter will introduce you to the basics of microservices, including what a microservice is and how to break a monolithic server down into microservices. It will be useful if you are not familiar with the concept of microservices or if you have never implemented them using the Rust programming language.

The following topics will be covered in this chapter:

- What are microservices?
- How to transform a traditional server architecture into microservices
- The importance of Rust in microservices development

Technical requirements

This chapter hasn't got any special technical requirements, but now is a good time to install or update your Rust compiler. You can get this from Rust's official website: `https://www.rust-lang.org/` . I recommend that you use the `rustup` tool, which you can download from `https://rustup.rs/`.

If you have previously installed the compiler, you need to update it to the latest version using the following command:

```
rustup update
```

You can get the examples for this book from the GitHub page: `https://github.com/PacktPublishing/Hands-On-Microservices-with-Rust-2018/`.

What are microservices?

Modern users interact with microservices every day; not directly, but by using web applications. Microservices are a flexible software development technique that help to implement applications as a collection of independent services with weak relations.

In this section, we'll learn about why microservices are a good thing and why we need them. Microservices follow the REST architecture, which provides rules about using consistent HTTP methods. We will also look at how microservices can be deployed to the user, which is one of their main advantages.

Why we need microservices

Microservices are a modern software development approach that refers to the splitting of software into a suite of small services that are easier to develop, debug, deploy, and maintain. Microservices are tiny, independent servers that act as single business functions. For example, if you have an e-commerce suite that works as a monolith, you could split it into small servers that have limited responsibility and carry out the same tasks. One microservice could handle user authorization, the other could handle the users' shopping carts, and the remaining services could handle features such as search functionality, social-media integration, or recommendations.

Microservices can either interact with a database or be connected to other microservices. To interact with a database, microservices can use different protocols. These might include HTTP or REST, Thrift, ZMQ, AMQP for the messaging communication style, WebSockets for streaming data, and even the old-fashioned **Simple Object Access Protocol** (**SOAP**) to integrate them with the existing infrastructure. We will use HTTP and REST in this book, because this is the most flexible way to provide and interact with the web API. We'll explain this choice later.

Microservices have the following advantages over monolithic servers:

- You can use different programming languages
- The code base of a single server is smaller
- They have an independent DevOps process to build and deploy activities
- They can be scaled depending on their implementation
- If one microservice fails, the rest will continue to work
- They work well within containers
- Increased isolation between elements leads to better security
- They are suitable for projects involving the Internet of Things

- They are in line with the DevOps philosophy
- They can be outsourced
- They can be orchestrated after development
- They are reusable

There are, however, a few drawbacks of microservices. These include the following:

- Too many microservices overload the development process
- You have to design interaction protocols
- They can be expensive for small teams

A microservices architecture is a modern approach that can help you achieve the goal of having loosely coupling elements. This is where the servers are independent from one another, helping you to release and scale your application faster than a monolithic approach, in which you put all your eggs in one basket.

How to deploy a microservice

Since a microservice is a small but complete web server, you have to deploy it as a complete server. But since it has a narrow scope of features, it's also simpler to configure. Containers can help you pack your binaries into an image of the operating system with the necessary dependencies to simplify deployment.

This differs from the case with monoliths, in which you have a system administrator who installs and configures the server. Microservices need a new role to carry out this function—DevOps. DevOps is not just a job role, but a whole software engineering culture in which developers become system administrators and vice versa. DevOps engineers are responsible for packing and delivering the software to the end user or market. Unlike system administrators, DevOps engineers work with clouds and clusters and often don't touch any hardware except their own laptop.

DevOps uses a lot of automation and carries the application through various stages of the delivery process: building, testing, packaging, releasing, or deployment, and the monitoring of the working system. This helps to reduce the time it takes both to market a particular software and to release new versions of it. It's impossible to use a lot of automation for monolithic servers, because they are too complex and fragile. Even if you want to pack a monolith to a container, you have to deliver it as a large bundle and run the risk that any part of the application could fail. In this section, we'll have a brief look at containers and continuous integration. We will go into detail about these topics in `Chapter 15`, *Packing Servers to Containers*, and `Chapter 16`, *DevOps of Rust Microservices – Continuous Integration and Delivery*.

Docker

When we refer to containers, we almost always mean Docker containers (`https://www.docker.com/`). Docker is the most popular software tool for running programs in containers, which are isolated environments.

Containerization is a kind of virtualization where the scope of the application's resources is limited. This means the application works at its maximum performance level. This is different from full virtualization, where you have to run the full operating system with the corresponding overhead and run your application inside that isolated operating system.

Docker has become popular for a variety of reasons. One of these reasons is that it has a registry—the place where you can upload and download images of containers with applications. The public registry is Docker Hub (`https://hub.docker.com/explore/`), but you can have a private registry for a private or permissioned software.

Continuous Integration

Continuous Integration (**CI**) is the practice of keeping a master copy of the software and using tests and merging processes to expand the features of the application. The process of CI is integrated with the **Source Code Management** (**SCM**) process. When the source code is updated (for example, in Git), the CI tool checks it and starts the tests. If all tests pass, developers can merge the changes to the master branch.

CI doesn't guarantee that the application will work, because tests can be wrong, but it removes the need to run tests from developers on an automated system. This gives you the great benefit of being able to test all your upcoming changes together to detect conflicts between changes. Another advantage is that the CI system can pack your solution in a container, so the only thing that you have to do is deliver the container to a production cloud. The deployment of containers is also simple to automate.

How to split a traditional server into multiple microservices

Around 10 years ago, developers used to use the Apache web server with a scripting programming language to create web applications, rendering the views on the server-side. This meant that there was no need to split applications into pieces and it was simpler to keep the code together. With the emergence of **Single-Page Applications** (**SPAs**), we only needed server-side rendering for special cases and applications were divided into two parts: frontend and backend. Another tendency was that servers changed processing method from synchronous (where every client interaction lives in a separate thread) to asynchronous (where one thread processes many clients simultaneously using non-blocking, input-output operations). This trend promotes the better performance of single server units, meaning they can serve thousands of clients. This means that we don't need special hardware, proprietary software, or a special toolchain or compiler to write a tiny server with great performance.

The invasion of microservices happened when scripting programming languages become popular. By this, we are not only referring to languages for server-side scripting, but general-purpose high-level programming languages such as Python or Ruby. The adoption of JavaScript for backend needs, which had previously always been asynchronous, was particularly influential.

If writing your own server wasn't hard enough, you could create a separate server for special cases and use them directly from the frontend application. This would not require rendering procedures on the server. This section has provided a short description of the evolution from monolithic servers to microservices. We are now going to examine how to break a monolithic server into small pieces.

Reasons to avoid monoliths

If you already have a single server that includes all backend features, you have a monolithic service, even if you start two or more instances of this service. A monolithic service has a few disadvantages—it is impossible to scale vertically, it is impossible to update and deploy one feature without interrupting all the running instances, and if the server fails, it affects all features. Let's discuss these disadvantages a little further. This might help you to convince your manager to break your service down into microservices.

Impossible to scale vertically

There are two common approaches to scaling an application:

- **Horizontally**: Where you start a new instance of application
- **Vertically**: Where you improve an independent application layer that has a bottleneck

The simplest way to scale a backend is to start another instance of the server. This will solve the issue, but in many cases it is a waste of hardware resources. For example, imagine you have a bottleneck in an application that collects or logs statistics. This might only use 15% of your CPU, because logging might include multiple IO operations but no intensive CPU operations. However, to scale this auxiliary function, you will have to pay for the whole instance.

Impossible to update and deploy only one feature

If your backend works as a monolith, you can't update only a small part of it. Every time you add or change a feature, you have to stop, update, and start the service again, which causes interruptions.

When you have a microservice and you have find a bug, you can stop and update only this microservice without affecting the others. As I mentioned before, it can also be useful to split a product into separate development teams.

The failure of one server affects all features

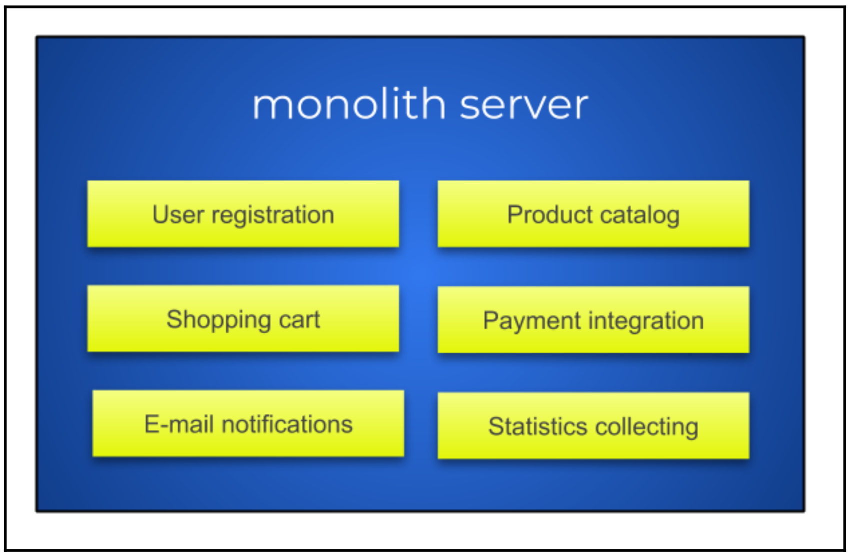
Another reason to avoid monoliths is that every server crash also crashes all of the features, which causes the application to stop working completely, even though not every feature is needed for it to work. If your application can't load new user interface themes, the error is not critical, as long as you don't work in the fashion or design industry, and your application should still be able to provide the vital functions to users. If you split your monolith into independent microservices, you will reduce the impact of crashes.

Breaking a monolithic service into pieces

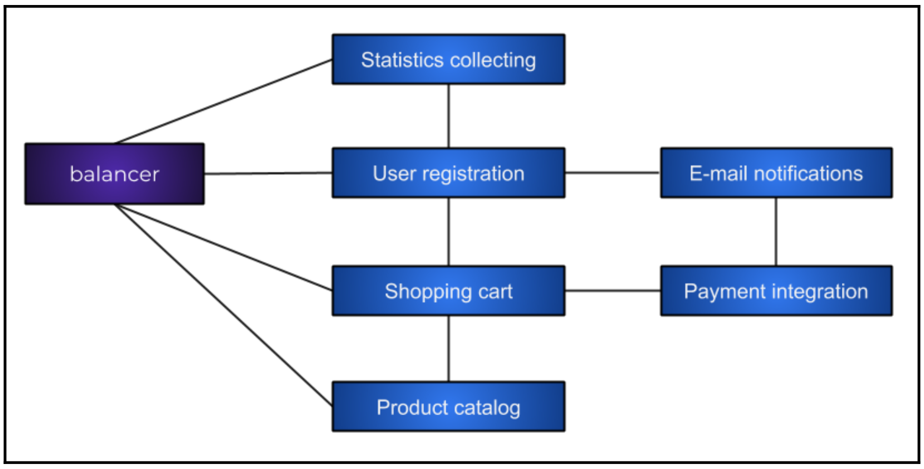
Let's look an example of an e-commerce monolith server that provides the following features:

- **User registration**
- **Product catalog**
- **Shopping cart**
- **Payment integration**
- **E-mail notifications**
- **Statistics collecting**

Old-fashioned servers developed years ago would include all of these features together. Even if you split it into separate application modules, they would still work on the same server. You can see an example structure of a monolithic service here:



In reality, the real server contains more modules than this, but we have separated them into logical groups based on the tasks they perform. This is a good starting point to breaking your monolith into multiple, loosely coupled microservices. In this example, we can break it further into the pieces represented in the following diagram:



As you can see, we use a **balancer** to route requests to microservices. You can actually connect to microservices directly from the frontend application.

Shown in the preceding diagram is the potential communication that occurs between services. For simple cases, you can use direct connections. If the interaction is more complex, you can use message queues. However, you should avoid using a shared state such as a central database and interacting through records, because this can cause a bottleneck for the whole application. We will discuss how to scale microservices in `Chapter 12`, *Scalable Microservices Architecture*. For now, we will explore REST API, which will be partially implemented in a few examples throughout this book. We will also discuss why Rust is a great choice for implementing microservices.

Definition of a REST API

Let's define the APIs that we will use in our microservice infrastructure using the REST methodology. In this example, our microservices will have minimal APIs for demonstration purposes; real microservices might not be quite so "micro". Let's explore the REST specifications of the microservices of our application. We will start by looking at a microservice for user registration and go through every part of the application.

User registration microservice

The first service is responsible for the registration of users. It has to contain methods to add, update, or delete users. We can cover all needs with the standard REST approach. We will use a combination of methods and paths to provide this user registration functionality:

- `POST` request to `/user/` creates a new user and returns its `id`
- `GET` request to `/user/id` returns information related to a user with `id`
- `PUT` request to `/user/id` applies changes to a user with `id`
- `DELETE` request to `/user/id` removes a user with `id`

This service can use the **E-mail notifications** microservice and call its methods to notify the user about registration.

E-mail notifications microservice

The **E-mail notifications** microservice can be extremely simple and contains only a single method:

- The `POST` request to `/send_email/` sends an email to any address

This server can also count the sent emails to prevent spam or check that the email exists in the user's database by requesting it from the **User registration** microservice. This is done to prevent malicious use.

Product catalog microservice

The **Product catalog** microservice tracks the available products and needs only weak relations with other microservices, except for the **Shopping cart**. This microservice can contain the following methods:

- `POST` request to `/product/` creates a new product and returns its `id`
- `GET` request to `/product/id` returns information about the product with `id`
- `PUT` request to `/product/id` updates information about the product with `id`
- `DELETE` request to `/product/id` marks the product with `id` as deleted
- `GET` request to `/products/` returns a list of all products (can be paginated by extra parameters)

Shopping cart microservice

The **Shopping cart** microservice is closely integrated with the **User registration** and **Product catalog** microservices. It holds pending purchases and prepares invoices. It contains the following methods:

- `POST` request to `/user/uid/cart/`, which puts a product in the cart and returns the `id` of item in the user's cart with the `uid`
- `GET` request to `/user/uid/cart/id`, which returns information about the item with `id`
- `PUT` request to `/user/uid/cart/id`, which updates information about the item with `id` (alters the quantity of items)
- `GET` request to `/user/uid/cart/`, which returns a list of all the items in the cart

As you can see, we don't add an extra "s" to the `/cart/` URL and we use the same path for creating items and to get a list, because the first handler reacts to the `POST` method, the second processes requests with the `GET` method, and so on. We also use the user's ID in the path. We can implement the nested REST functions in two ways:

- Use session information to get the user's `id`. In this case, the paths contain a single object, such as `/cart/id` . We can keep the user's `id` in session cookies, but this is not reliable.
- We can add the `id` of a user to a path explicitly.

Payment Integration microservice

In our example, this microservice will be a third-party service, which contains the following methods:

- `POST` request to `/invoices` creates a new invoice and returns its `id`
- `POST` request to `/invoices/id/pay` pays for the invoice

Statistics collecting microservice

This service collects usage statistics and logs a user's actions to later improve the application. This service exports API calls to collect the data and contains some internal APIs to read the data:

- `POST` request to `/log` logs a user's actions (the `id` of a user is set in the body of the request)

- `GET` request to `/log?from=?&to=?` works only from the internal network and returns the collected data for the period specified

This microservice doesn't conform clearly to the REST principles. It's useful for microservices that provide a full set of methods to add, modify, and remove the data, but for other services, it is excessively restrictive. You don't have follow a clear REST structure for all of your services, but it may be useful for some tools that expect it.

Transformation to microservices

If you already have a working application, you might transform it into a set of microservices, but you have to keep the application running at the highest rate and prevent any interruptions.

To do this, you can create microservices step by step, starting from the least important task. In our example, it's better to start from email activities and logging. This practice helps you to create a DevOps process from scratch and join it with the maintenance process of your app.

Reusing existing microservices

If your application is a monolith server, you don't need to turn all modules into microservices, because you can use existing third-party services and shrink the bulk of the code that needs rewriting. These services can help with many things, including storage, payments, logging, and transactional notifications that tell you whether an event has been delivered or not.

I recommend that you create and maintain services that determine your competitive advantage yourself and then use third-party services for other tasks. This can significantly shrink your expenses and the time to market.

In any case, remember the product that you are delivering and don't waste time on unnecessary units of your application. The microservices approach helps you to achieve this simply, unlike the tiresome coding of monoliths, which requires you to deal with numerous secondary tasks. Hopefully, you are now fully aware of the reasons why microservices can be useful. In the next section, we will look at why Rust is a promising tool for creating microservices.

Why Rust is a great tool for creating microservices

If you have chosen to read this book, you probably already know that Rust is an up-to-date, powerful, and reliable language. However, choosing it to implement microservices is not an obvious decision, because Rust is a system programming language that is often assigned to low-level software such as drivers or OS kernels. This is because you tend to have to write a lot of glue code or get into detailed algorithms with low-level concepts, such as pointers in system programming languages. This is not the case with Rust. As a Rust programmer, you've surely already seen how it can be used to create high-level abstractions with flexible language capabilities. In this section, we'll discuss the strengths of Rust: its strict and explicit nature, its high performance, and its great package system.

Explicit versus implicit

Up until recently, there hasn't been a well-established approach to using Rust for writing asynchronous network applications. Previously, developers tended to use two styles: either explicit control structures to handle asynchronous operations or implicit context switching. The explicit nature of Rust meant that the first approach outgrew the second. Implicit context switching is used in concurrent programming languages such as Go, but this model does not suit Rust for a variety of reasons. First of all, it has design limitations and it's hard or even impossible to share implicit contexts between threads. This is because the standard Rust library uses thread-local data for some functions and the program can't change the thread environment safely. Another reason is that an approach with context switching has overheads and therefore doesn't follow the zero-cost abstractions philosophy because you would have a background runtime. Some modern libraries such as `actix` provide a high-level approach similar to automatic context switching, but actually use explicit control structures for handling asynchronous operations.

Network programming in Rust has evolved over time. When Rust was released, developers could only use the standard library. This method was particularly verbose and not suitable for writing high-performance servers. This was because the standard library didn't contain any good asynchronous abstractions. Also, event `hyper`, a good crate for creating HTTP servers and clients, processed requests in separate threads and could therefore only have a certain number of simultaneous connections.

The `mio` crate was introduced to provide a clear asynchronous approach to make high-performance servers. It contained functions to interact with asynchronous features of the operating system, such as epoll or kqueue, but it was still verbose, which made it hard to write modular applications.

The next abstraction layer over `mio` was a `futures` and `tokio` pair of crates. The `futures` crate contained abstractions for implementing delayed operations (like the defers concept in Twisted, if you're familiar with Python). It also contained types for assembling stream processors, which are reactive and work like a finite state machine.

Using the `futures` crate was a powerful way to implement high-performance and high-accuracy network software. However, it was a middleware crate, which made it hard to solve everyday tasks. It was a good base for rewriting crates such as `hyper`, because these can use explicit asynchronous abstractions with full control.

The highest level of abstraction today are crates that use `futures`, `tokio`, and `hyper` crates, such as `rocket` or `actix-web`. Now, `rocket` includes high-level elements to construct a web server with the minimal amount of lines. `actix-web` works as a set of actors when your software is broken down into small entities that interact with one another. There are many other useful crates, but we will start with hyper as a basis for developing web servers from scratch. Using this crate, we will be between low-level crates, such as futures, and high-level crates, such as `rocket`. This will allow us to understand both in detail.

Minimal amount of runtime errors

There are many languages suitable for creating microservices, but not every language has a reliable design to keep you from making mistakes. Most interpreted dynamic languages let you write flexible code that decides on the fly which field of the object to get and which function to call. You can often even override the rules of function calling by adding meta-information to objects. This is vital in meta-programming or in cases where your data drives the behavior of the runtime.

The dynamic approach, however, has significant drawbacks for the software, which requires reliability rather than flexibility. This is because any inaccuracy in the code causes the application to crash. The first time you try to use Rust, you may feel that it lacks flexibility. This is not true, however; the difference is in the approach you use to achieve flexibility. With Rust, all your rules must be strict. If you create enough abstractions to cover all of the cases your application might face, you will get the flexibility you want.

Rust rookies who come from the JavaScript or the Python world might notice that they have to declare every case of serialization/deserialization of data, whereas with dynamic languages, you can simply unpack any input data to the free-form object and explore the content later. You actually have to check all cases of inconsistency during runtime and try and work out what consequences could be caused if you change one field and remove another. With Rust, the compiler checks everything, including the type, the existence, and the corresponding format. The most important thing here is the type, because you can't compile a program that uses incompatible types. With other languages, this sometimes leads to strange compilation errors such as a case where you have two types for the same crate but the types are incompatible because they were declared in different versions of the same crate. Only Rust protects you from shooting yourself in the foot in this way. In fact, different versions can have different rules of serialization/deserialization for a type, even if both declarations have the same data layout.

Great performance

Rust is a system programming language. This means your code is compiled into native binary instructions for the processor and runs without unwanted overhead, unlike interpreters such as JavaScript or Python.

Rust also doesn't use a garbage collector and you can control all allocations of memory and the size of buffers to prevent overflow.

Another reason why Rust is so fast for microservices is that it has zero-cost abstractions, which means that most abstractions in the language weigh nothing. They turn into effective code during compilation without any runtime overhead. For network programming, this means that your code will be effective after compilation, that is, once you have added meaningful constructions in the source code.

Minimal dependencies burden

Rust programs are compiled into a single binary without unwanted dependencies. It needs libc or another dynamic library if you want to use OpenSSL or similar irreplaceable dependencies, but all Rust crates are compiled statically into your code.

You may think that the compiled binaries are quite large to be used as microservices. The word microservice, however, refers to the narrow logic scope, rather than the size. Even so, statically linked programs remain tiny for modern computers.

What benefits does this give you? You will avoid having to worry about dependencies. Each Rust microservice uses its own set of dependencies compiled into a single binary. You can even keep microservices with obsolete features and dependencies besides new microservices. In addition, Rust, in contrast with the Go programming language, has strict rules for dependencies. This means that the project resists breaking, even if someone forces an update of the repository with the dependency you need.

How does Rust compare to Java? Java has microframeworks for building microservices, but you have to carry all dependencies with them. You can put these in a fat **Java ARchive** (**JAR**), which is a kind of compiled code distribution in Java, but you still need **Java Virtual Machine** (**JVM**). Don't forget, too, that Java will load every dependency with a class loader. Also, Java bytecode is interpreted and it takes quite a while for the **Just-In-Time** (**JIT**) compilation to finish to accelerate the code. With Rust, bootstrapping dependencies don't take a long time because they are attached to the code during compilation and your code will work with the highest speed from the start since it was already compiled into native code.

Summary

In this chapter, we have mastered the basics of microservices. Simply put, a microservice is a compact web server that handles specific tasks. For example, microservices can be responsible for user authentication or for email notifications. They make running units reusable. This means you don't need to recompile or restart units if they don't require any updates. This approach is simpler and more reliable in deployment and maintenance.

We have also discussed how to split a monolithic web server that contains all of its business logic in a single unit into smaller pieces and join them together through communication, in line with the ideology of loose coupling. To split a monolithic server, you should separate it into domains that are classified by what tasks the servers carry out.

In the last section of this chapter, we've looked at why Rust is a good choice for developing microservices. We touched on dependencies management, the performance of Rust, its explicit nature, and its toolchain. It's now time to dive deep into coding and write a minimal microservice with Rust.

In the next chapter we will start to writing microservices with Rust using `hyper` crate that provides all necessary features to write compact asynchronous HTTP server.

Further reading

You have learned about the basics of microservices in this chapter, which will serve as a point for you to start writing microservices on Rust throughout this book. If you want to learn more about topics discussed in this chapter, please consult the following list:

- *Microservices - a definition of this new architectural term*, 2014, Martin Fowler, available at `https://martinfowler.com/articles/microservices.html`. This article introduces the concept of microservices.
- `mio`, available at `https://github.com/carllerche/mio`. This is a crate that is widely used by other crates for asynchronous operations in Rust. We won't use it directly, but it is useful to know how it works.
- *Network Programming with Rust, 2018*, Abhishek Chanda, available at `https://www.packtpub.com/application-development/network-programming-rust`. This book explains more about network addresses, protocols and sockets, and how to use them all with Rust.