Rust for the IoT

Building Internet of Things Apps with Rust and Raspberry Pi

Joseph Faisal Nusairat

Apress®

Rust for the IoT

Building Internet of Things Apps with Rust and Raspberry Pi

Joseph Faisal Nusairat

Apress®

Rust for the IoT: Building Internet of Things Apps with Rust and Raspberry Pi

Joseph Faisal Nusairat
Scottsdale, AZ, USA

ISBN-13 (pbk): 978-1-4842-5859-0 ISBN-13 (electronic): 978-1-4842-5860-6
https://doi.org/10.1007/978-1-4842-5860-6

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar

Cover image by Maarten van den Heuvel on Unsplash (www.unsplash.com)

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484258590. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

To my beautiful, wonderful, wife Heba Fayed, your patience and support is what got this book finished. Thank you for everything you do.

Table of Contents

About the Author

Joseph Faisal Nusairat, author of three Apress books, is currently a Senior Staff Engineer at Tesla, developing the next generation of products for the Platform Engineering team. He has experience in a full range of the development life cycle from requirements gathering, to full stack development, to production support of applications, in addition to speaking, coaching, and training of software. Joseph started his career in 1997 doing primarily Java/JVM full stack applications. In the Java realm, he became proficient and gave talks on Java, Groovy, Scala, Kotlin, and Clojure. In the last few years, other languages like Rust, Go, and Elixir have caught not only his interest but his dedication. Over the years, he's learned to create code that not only is readable but maintainable all while trying to minimize its memory footprint and maximizing performance. His career has led through a variety of industries from banking, insurance, fraud, retailers, defense, and now electric cars. Joseph is a graduate of Ohio University with dual degrees in Computer Science and Microbiology with a minor in Chemistry.

Joseph is a published author, speaker, and trainer. He can be found on twitter, github, and gitlab as `@nusairat`.

About the Technical Reviewer

Kan-Ru Chen is a Software Engineer at Amazon who builds cloud services for millions of customers. Before that, Kan-Ru worked six years at Mozilla, tuning Firefox performance. He was exposed to the Rust programming language while at Mozilla and fell in love with it.

At his free time, he enjoys contributing to free and open source projects, like Rust. He is also a long-time Debian Developer. His main areas of interest include programming languages, parser, compiler, performance, and security.

You can reach him at kanru.rust [at] kanru.info or on Twitter at `@kanru`.

Acknowledgments

Technical books are one of the biggest labors of love for those who write. They most often don't make much money for the amount of time we spent working on them and are mostly written to fulfill a dream of writing something we are passionate about and a desire to communicate with you that information. This dream is not fulfilled by just the author, there are many people along the way that help both directly and indirectly, and I would be remiss if I did not thank them for getting me there.

First, the love of my life, Heba. You met me when I first started writing this book and have supported me every step of the way. This included many weekends we couldn't go out because I was working on the book, many vacations I had my laptop open, and many late nights I didn't go to bed till late because of it. Not only did you support me but you helped me with some of the diagrams, as well as proofreading the book to tell me when my explanations made no sense. You are amazing!

Mac Liaw, you've been not only a great friend but an awesome mentor as well. You've given me opportunities for not only new jobs, but you were there responding to texts at 1 a.m. when I was getting stuck on topics for the book. Your help was greatly appreciated. Also thank you for the excellent advice on where to propose to Heba; it made that day unforgettable.

Next my technical reviewer Kan-Ru Chen, I can't thank you enough; you did cause my author reviews to run longer, but you saved my butt by pointing out in detail sections that were convoluted, incorrect, or could have been written better. Often we are pushed to get things done in a timely fashion and one ends up rushing. I'm grateful to have had you help fix those errors and provide great feedback. And Mark Powers, my editor, who I initially told this book will be done in September or late fall, thank you for your patience in letting me put together an enormous set of information even though it meant missing quite a few deadlines.

Joseph Swager, the man who dragged me out to the bay and gave me a shot in some new directions in my work life and also whose idea it was to write this book in the first place and was initially my coauthor, you unfortunately had to bail a few chapters in due to work, but hopefully our shared vision is what this book ended up being. Next book we'll do!

Additionally, my past work experience has been littered with many people who've helped not only make me a better developer but helped me in my career and world advice. You all have been there to help guide and give feedback and support when needed. Brian Sam-Bodden, founder of Integrallis Software, you got me into writing my first book, giving my first presentation, never would I have gotten this far without you. Jason Warner, you gave me my first job in Arizona, and continue to give great advice to this day. (Incidentally, he has a GitHub page where you can get great advice/perspective too; I say we force him to keep answering questions for years to come (github.com/jasoncwarner/ama).)

And a final few folks: Jeff Hart, thanks for all your technical wisdom on cloud systems and for helping me to debug my Rust code at random times. Aaron Kark, we worked together 18 years ago, and 1 year ago, hopefully it doesn't take 17 more to work together again. Kelli Alexander and Veronica Martinez, thanks for being very supportive friends. I also want to give a final praise to my current team: Sheen, Ilia, Kouri, Konstantin, Clark, Ross, Issac, Crystal, and Nick, you are of the best groups of people I've ever had the pleasure of working with.

Lastly, thank you, the reader, for picking up and selecting this book. I really hope it helps you despite what errors may have snuck into the code. This book has taken more time than I initially had planned and has been one of the biggest labors of love in my professional career. I hope I've delivered something you can really learn from.

Very last, thank you mum, you're the best mother, grandma, person I know.

Preface

This book is for anyone with programming experience who wants to jump into the Internet of Things space. This book covers the set from cloud application building and deploying to creation of the Raspberry Pi application and communication between. While there are many crates in the Rust world to write applications, this book shows you which crates you will need and how to combine them together to create a working IoT cloud and device application. This book while not an advanced Rust language book does cover a few more advanced features. It is best to have at least some understanding before starting.

Chapter Listing

The book is composed of the following chapters:

- Chapter 1 – This covers what this book is going to solve; we tackle the issues and problems surrounding IoT applications and their architecture. We also go over the hardware that is needed for this book, and the chapter ends with some simple Rust examples.
- Chapter 2 – This starts with setting up and creating our first microservice, the `retrieval_svc`; this will set up simple calls to it and integrate and set up the database for it.
- Chapter 3 – This chapter is more heavily focused on the `upload_svc`, and in here we learn how to upload images and video files to store locally. We then parse the metadata out of the files and call the `retrieval_svc` to store their metadata.
- Chapter 4 – Back to the `retrieval_svc`, we add GraphQL to use on top of the web tier instead of pure RESTful endpoints. We also create the `mqtt_service` that will serve as our bridge to communicate between the back end and the Pi using MQTT.

- Chapter 5 – Enhancing both the `retrieval_svc` and the `mqtt_service` by using serialized binary data via Cap'n Proto to talk, instead of having the communication between the two be REST calls. Also on the `retrieval_svc` side, we add CQRS and eventual consistency to our graph mutations for comments.
- Chapter 6 – This adds using Auth0 to authenticate the user so that our database can identify a device to a user. We also add self-signed certificates to secure the communication of the MQTT.
- Chapter 7 – In this chapter, we learn how to create Docker images of all our microservices, combining them with Kubernetes and deploying to a cloud provider with Helm charts.
- Chapter 8 – This is our first hands-on chapter with the Raspberry Pi in which we will set up the heartbeat to communicate to the MQTT backend we created earlier.
- Chapter 9 – This incorporates the Sense HAT device to gather data about our environment to the Pi. The Sense HAT provides us a visual LED display, temperature sensors, and a joystick for interactions.
- Chapter 10 – In this chapter, we add a camera to the device which will allow us to do facial tracking and recording.
- Chapter 11 – This is one of the last chapters in which we incorporate the video camera to send data back to the cloud as well as allowing the Pi to receive recording commands from the cloud, and finally we allow the Pi to be used as a HomeKit device to show temperature and motion.
- Chapter 12 – This final short chapter discusses how we would build an ISO image for our given application and other bundling issues.

CHAPTER 1

Introduction

The Internet of Things (IoT) is a highly encompassing term that covers everything from your home network-connected camera to the oven that is Wi-Fi connected, all the way to your modern electric cars like the Tesla that are always connected to the network and almost always on. The most basic premise of IoT is a hardware device that is a connected network appliance. In modern days, that usually means Internet and almost always connected to a cloud service, but it can just as easily be a local area network.

Only in the last 10 years have we truly embraced the IoT model for not only offices and factories but for everyday living. The most common consumer IoT systems are the ones from or supported by Apple, Google, and Amazon that provide cameras, thermostats, doorbell, and lights. All of those devices can then be used in conjunction with each other and for home automation and control. While many of these devices are used for fun in a home, they have beneficial application for elderly care and for medical monitoring and even can be used in industrial and manufacturing components. Devices in factories can report on the status of how many components are rolling off the assembly line, if there is a failure at a point, or even throughput of a factory. Used in conjunction with machine learning, the possibilities are endless.

And while IoT as a term didn’t make our way officially in the lexicon till 1999 by Kevin Ashton of Procter & Gamble, the concept has been around since well before that. What gave birth to IoT dates back to 1959 with the Egyptian-born Inventor Mohamed M. Atalla and Korean-born Dawon Kahng while working at Bell Labs in 1959. They created the MOSFET (metal-oxide-semiconductor field-effect transmitter) which is the basis for the semiconductor, which revolutionized electronics from huge tubes to the microchip components we have in our smart watches, phones, cameras, cars, and even your ovens. It would still take another 23 years though till someone at Carnegie Mellon decided to hook up monitoring a Coca-Cola machine for its inventory that would mark the first true IoT device, before anyone even thought of what IoT was, and then another 10 years before companies like Microsoft and Novell really proposed usable solutions. However, even then chips were expensive and relatively big. Today Raspberry Pi packs way more punch than the desktops of the 1990s, especially in the GPU department.

© Joseph Faisal Nusairat 2020
J. F. Nusairat, *Rust for the IoT*, https://doi.org/10.1007/978-1-4842-5860-6_1

Who Is This Book For?

This book is for anyone from your hobbyist to someone trying to create their own commercial IoT products. I guess for your hobbyist, there is the question of why. Purchasing IoT applications has become inexpensive, fairly customizable, and routine; why bother to create your own? And while one answer is simply for fun, another is that you want to create a fully customizable solution. And finally another answer that became even more apparent this year was ownership, that you are the sole owner. This importance became obvious to me in two cases this year. This first was with the Amazon-owned company Ring. They had had to let go of four employees due to privacy concerns that they had spied on customers snooping in on their feeds. And while this is likely the exception and not the rule, it still lends to the idea of wanting to create pipes that are 100% solely owned by you.[1] The second was Sonos, who after customers spent years buying components found out the older products will no longer be backward compatible, leaving many in the dark to use new software updates.[2] And while it would be hard to replicate the amount of code they write, the open source community that integrates with custom Pi components is growing and will help to live on even if it means you have to code it yourself.

This book is titled *Rust for the IoT*. Before we discuss what we are going to build, let's break out those two words further.

What Is IoT?

Internet of Things, or IoT, which will be used to reference it for the rest of the book, has become a new and ever-growing marketplace in the last few years, even though it's been around for decades. At its core, it's a network of devices that communicate with each other and often with the cloud.

The most common IoT systems are the ones from or supported by Apple, Google, and Amazon that provide cameras, thermostats, doorbell, and lights that all interact with each other. These devices in conjunction can be used in home automation and control.

[1]www.usatoday.com/story/tech/2020/01/10/amazons-ring-fired-employees-snooping-customers-camera-feeds/4429399002/

[2]https://nakedsecurity.sophos.com/2020/01/23/sonoss-tone-deaf-legacy-product-policy-angers-customers/

While many of these devices are used for fun in a home, they have beneficial application for elderly care and for medical monitoring and even can be used in industrial and manufacturing components.

In addition, IoT does not stop there; it's gained dominance in all realms of device use. Car companies have started adopting IoT to have a more fully connected car. Tesla started the trend, and others have really picked it up full speed and use the same concepts and features as your smart device. Incidentally, this is something I know quite a bit about because I was in charge of architecting and coding Over-The-Air (OTA) updates for one such car company.

For this book though, I am sticking to personal use in the home, since most people into IoT are home enthusiasts and because creating one for a car is a tad more expensive since you would need a car. But the same principles can be applied everywhere.

I have had an interest in IoT since the rudimentary RF devices you could purchase in the 1980s from RadioShack. Quite a bit has changed since then. We are now in an age where home automation is actually pretty good. We have cameras, devices, cloud, and voice integration, but there are still many improvements to be made. We feel this book could start you on your way as a hobbyist or even in a professional setting. Why Rust? When reviewing what languages we could use for both embedded board development and was extremely fast cloud throughput computing at a low cost, Rust kept coming up.

IoT 10K Foot Picture

IoT at its core is the concept of connectivity, having everything interconnected with each other; executing that is often not the most simple concept. In Figure 1-1, I have diagramed your basic IoT interactive diagram.

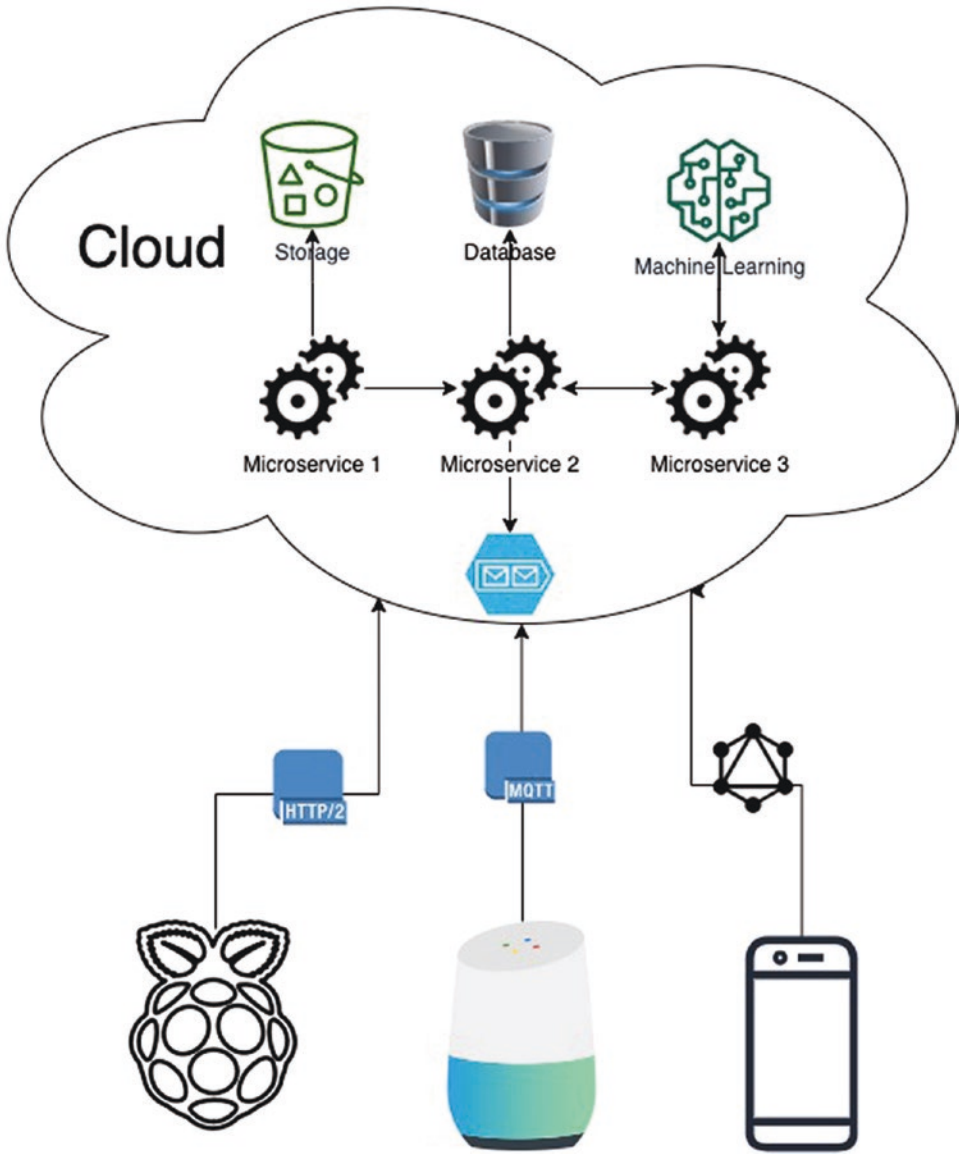

Figure 1-1. *Showing your standard IoT diagram*

Let's get a few takeaways from this diagram. You will notice at the bottom there are a few hardware devices and a mobile application. Hardware devices in our case will be a Raspberry Pi, but they could just as easily be your Google Home Hub, Alexa, or a car. These are all devices you are familiar with. The Raspberry Pi and Google Home Hub in the picture serve as endpoints that can play music, capture video, or record other information about their surroundings. The mobile devices then serve a role in communicating with those devices (in the case of the Google Home Hub, it serves a dual role, one in communicating and the other in capturing the world around it).

The end goal as we said is to have a fully connected system, so not only do these devices communicate with the cloud, but they receive communications back from the cloud. The communication back from the cloud can be due to input from your mobile application or could be a scheduled call. The pipes between represent this communication, but you will notice we have a variety of communication paths listed.

HTTPS

This is your standard HTTPS path. These paths exist often from the device to the cloud. Remember the endpoint in your cloud will have a static domain name like `rustfortheiot.xyz`. This domain name allows a constant path for the IoT device to talk to. The device can upload video or other large data and can download video, music, and other media content. And it's also available for anything that would require an immediate request/response, for example, if we wanted to know what the forecast was for today.

The downside to HTTPS connections is that if the server endpoints are down, if they are overloaded, they may be slow or not responsive at all. In addition, there is data that the device will send back that doesn't require a response.

The hardware is the core feature the reason we even have the rest of the diagram. These will give life to our commands. A car every time you drive is generating data on your speed, distance, and so on. Your home devices know when you turn on the lights and when you walk by a camera even if it's not recording; it's detecting the motion. For that data, HTTP may not even work or is overkill.

Message Queue

Message queues (MQs) you have often used with any publication/subscriber system and that in many ways are a few of the use cases we just described. If you are sending health data of your device, periodical temperature readings, this is all pub/sub type systems. The device wants to send the data to the cloud, but it doesn't care where the data eventually ends. MQs are battle tested to handle high loads and are not as often updated as your microservice updates. This means you can easily update your microservices without worrying about downtime of the application. In addition, if you need to take the microservice down for an extended time, you won't lose the data; it will receive it once it reconnects to the message queue.

We will use the message queue as the intermediary for sending messages back and then the HTTPS call from the hardware for retrieving videos. Also remember that the calls you will be making for HTTPS will be secure connections, and the MQ calls should be via Transport Layer Security (TLS). Now let's jump to the cloud. You will notice a fairly standard application layer with microservices, a database, and a bucket to store files in. In our case, we will used a local store for saving image and video files. Two other interesting items are the message queue (MQ) and machine learning. Machine learning (ML) is growing and really helps with IoT devices since often they generate so much data. We just mentioned all the data that the MQ can retrieve. This data is invaluable in being able to use ML to generate guides, suggestions, and adaptive feedback. We won't dive into machine learning in this book, that will be a book in of itself. If you are interested, you can read *Practical Machine Learning with Rust* (`www.apress.com/us/book/9781484251201`). The microservice architecture in the backend allows you to create a variety of small services you can scale independently of each other but can also communicate as if they are on one endpoint (we will discuss how to do this when we get to Chapter 7). These microservices can then talk to database, bucket stores (like S3), or the message queue. All of that backend will process data, serve as endpoints to route data from mobile to the device, and even send notifications either to the device or the mobile application.

Why Rust?

The next question that may come to mind is why did we pick Rust? If you look at most web applications, they aren't written in Rust; if you look at most board development, it isn't in Rust either. So why Rust? Rust is a multi-paradigm programming language that focuses on performance and safety. Rust, by what it allows you to do, has quite a bit more performance and safety implemented than other languages. The biggest way this is shown is in Rust's borrowing and ownership checks. Rust makes it so that there are specific rules around when a variable is borrowed, who owns it, and for how long they own it for. This has been the main attraction of Rust for many. The code becomes faster, less memory intensive, and less like to have two variables access each other at the same time. We will get into this more in the borrowing section. Stylistically, Rust is similar to languages like Go and has C-like syntax with pointers and references. And while some of the Rust crates lack the maturity of other languages, the language itself is continuously enhancing and added to.

Pre-requisites

While we are covering some Rust syntax at the end of this chapter, this is not an introduction to Rust. And while I don't think you need an advanced understanding of Rust, if you are familiar with other functional or imperative languages, then you should be able to get away with a basic understanding. However, if you don't have that background and have already purchased this book or are thinking of doing so not to fret, I'd suggest one of two options:

1. *Learn Rust* (`www.rust-lang.org/learn`) – Between reading the online book and the examples, you can gain quite a good understanding of book. The book is often updated and usually up to date. It's how I initially learned Rust.

2. *Beginning Rust* book (`www.apress.com/gp/book/9781484234679`) – Often it's easier to learn through longer books that will go into greater detail, and if that's what you need, *Beginning Rust* is for you.

In addition, throughout the book we are going to cover a number of topics from microservices, GraphQL, CQRS, Kubernetes, Docker, and more. And while I will provide some explanations and backgrounds for each technology we introduce, there are entire books devoted to each of those tools. If you ever feel you need to learn more, I would suggest looking online; we will give resource links during those chapters.

Components to Purchase

In the first half of this book, we will create a cloud application, and while we will be deploying that application to DigitalOcean cloud services, that isn't actually a requirement in building everything. Even then, we are picking DigitalOcean over services like AWS to mitigate the cost.

However, the second half of the book does deal with creating a Raspberry Pi–based application with a few add-ons. And while you will be able to follow along with this book without any cloud or hardware, to make the most of it, we will recommend a few cloud pieces and hardware that is designed to integrate with the software in this book. In the following section, I've given a list of hardware that you will need to purchase to fully follow along with the book. I've also provided the links on Amazon, but you can get the actual hardware from anywhere, and after some time the links may change:

1. Raspberry Pi 4 4 GB Starter Kit (`https://amzn.com/B07V5JTMV9`) – This kit will run about $100 but will include everything we need to get the basic Raspberry Pi up and running: from cables to connect it to your monitor, to power cables, and even a 32 GB SD card to be used for application and video storage. There are cheaper kits you can buy, but the all-in-ones will be a great starting point. Note: I selected and used the 4 for development, but if you used a 3, it should work as well; you will just have to adjust some endpoints when downloading OS software. The full Pi 4 kit will cost roughly $100.

2. Raspberry Pi Debug Cable (`https://amzn.com/B00QT7LQ88`) – This is a less than $10 cable that you can use to serially connect your Pi to your laptop without having to have a monitor, keyboard, or SSH ready. We will use this for initial setup, but if you are willing to hook your keyboard and monitor directly, it's not necessary.

3. Sense HAT (`https://amzn.com/B014HDG74S`) – The Sense HAT is an all-in-one unit that sits on all the Pi's GPIO pins that provides an 8 x 8 LED matrix display as well as numerous accelerometer, gyroscope, pressure, humidity, temperature sensors, and a joystick. We will be making use of the temperature, LED, and joystick later in this book. But this HAT provides quite a bit for $35.

4. Raspberry Pi Camera Module with 15 Pin Ribbon (`https://amzn.com/B07JPLV5K1`) – The camera we will be using is a $10 simple camera that is connected by ribbon to the Raspberry Pi. Since we are using this for simple video and face detection, the camera can be fairly basic, but it's up to you how much you want to spend on it.

While I have given you Amazon links to purchase everything, you are free to purchase from any supplier; it's all the same. This was just for ease of use.

We will cover and use all these components throughout the book.

Goals

The main goal of this book is to create a complete IoT application from the device all the way to the backend and all the parts in between. Without taking too many shortcuts, we will be using practices and techniques that are used for larger-scale applications. The goal is to give you all the lessons needed should you wish to expand your IoT application.

What we will actually be building is a HomeKit-enabled video recording device that stores and parses for metadata video and image files to the cloud and allow downloading and commenting on those videos. Here are the details:

Raspberry Pi – Allow a user to authenticate so that we know which Pi the files originate from. Allow the user to click a button on the Pi to see the temperature. Record video with facial recognition storing the video and image captures and sending the data to the cloud.

Cloud – Allow downloading and uploading of video and image files. Parse video and image files for metadata content. Create endpoints in the backend for users to create comments and query comments for the video files.

To perform all these features, we will use dozens of rust crates all working together to create a seamless system. We will create an application using a variety of tools like GraphQL, OpenCV, and eventual consistency (EC), all words that will become more clear as we go on. I will say what I am writing is not anything you couldn't figure out yourself if you know what to look for. Most of this information is available online, if you dig far enough, but it's sometimes hard to pick the right crates and get them to work together, and we've spent countless hours researching for our own work and for the book to bring it together. And in many instances, I've forked crates to either upgrade them for our use or to provide customizations we need. The code for the book will cover in more detail the following techniques:

1. Server side
   a. Creating a deployable set of microservices
   b. Server application that exposes GraphQL endpoints
   c. Server application that uploads and downloads files
   d. Communicating with hardware securely via MQTT
   e. Creating and using certificates
   f. Creating Docker, Helm, and Kubernetes scripts to deploy the application

2. Hardware side
    a. Setting up a Raspberry Pi
    b. Adding peripherals to the development board/Raspberry Pi
    c. Interacting with HomeKit
    d. Capturing video data
        i. Performing OpenCV on the video
        ii. Recording and uploading video content
        iii. Using SQLite to have a resilient store of data

Before we start coding, we are going to discuss the server and hardware side more.

Source Code

All of the source code for this book will be located on my GitHub page at `http://github.com/nusairat/rustfortheiot`.

This will include the services for the cloud, the applications for the Raspberry Pi, and the necessary build-and-deploy docker files. While I do step you through most of the code in the book, some of the more repetitive items like arguments for variables I only show you once to apply to your other services/applications. If you have any issues, please create an issue or you can tweet me at @nusairat. Now of course be aware that as the years go on, there could be compilation issues due to the version of Rust that is the current version. As of the time of finishing this book, the version of Rust is 1.43.1.

Web Application

The first half of the book will be on the server side. On that side, we will create a multitude of endpoints and tools that work with each other. The following is what we are going to make:

1. Microservices
    a. Upload/download service
    b. Retrieval service
    c. MQ service

2. Postgres database
3. Eventstore database
4. Message queue

All of those services will become more clear later. The first backend server we are going to design is going to serve a multitude of purposes, but essentially act as a bridge between your IoT device and the cloud. This will allow us a multiple of flexible options that you may not get with having a stand-alone IoT application and certainly is the way that most home devices work these days:

1. Act as a remote storage. When recording video or images from your IoT device, we will store it locally on the IoT device initially. However, this is not ideal if we want to retrieve the data from a remote client device; the round trip would be extremely slow. In that case, the application would have to call a server, and the server would then have to call the IoT device and start the transfer of data. While this is fine for real-time live video, if you are trying to view lots of historical archives, the slow download speed would become uncomfortably noticeable. In addition, it's great to have offsite storage of the data to serve as backup for your application. Most cloud providers will provide fairly cheap storage for large data; they just charge you for access to the data. As long as you aren't constantly accessing the data, you are fine moneywise. Our cloud application will be able to store to the local file store of the server it's on as well as to cloud storage services like S3. The reason for this will become apparent later, but this will allow us to run one of the upload services locally from a Raspberry Pi (or other server) co-located in your house and to the cloud. This can help lower costs for storage and servers and is common in any do-it-yourself system.
2. Another issue we want to tackle is sending commands to the IoT device. Mobile devices allow you to send commands to your home units through the backend. In our application, we are going to allow recording start and stop commands to be sent via a RESTful endpoint to the backend that will control whether the Pi records or not.

3. Querying of data. As you store more and more files, images, and video, you are going to want to add tags to these uploads but also search for them, not only custom tags but the metadata associated with the files. Images and video often have metadata created and stored with them. These can include things like location, time, aperture and other settings (for video/camera), quality rate, and so on. These are all services the user will want to search for. We will parse the video and image files and store their metadata for use later.

Board Application

When we first started thinking of what we wanted to use with Rust, the board is what attracted us the most. With the board, there are many options from your Raspberry Pi to a more advanced iMX.8 board, which we initially thought of going for, but then the Raspberry Pi 4 came out. The 4 is an extremely powerful and advanced board, and it is not only a hobbyist board of choice but is often used in the real world. In the real world, before you've created your custom chipset, design, and others for your hardware piece, the Pi can serve a short-term prototyping tool that your engineers can work on while waiting for revisions of the production board. Raspberry Pis are the hobbyist choice because of their cost, size, and ability. There are a few ways of creating IoT solutions, and companies employ a variety of solutions. You have anywhere that range from relatively dumb devices that do one thing like take record temperature and send it back to a common device (think of Ecobee's thermostat sensors) to a more encompassing device that has speakers and cameras like a a smart doorbell, or even more advance that has monitors, sensors, and so on. With the Pi, our options are a bit more limitless since we can attach whatever sensors we want to the Pi. In addition for something that is just recording temperature, you could go down to a cheaper Pi Zero. All of this will be future options for your components; for now, we are sticking with one Pi that has all the components attached to itself.

With this setup, we are going to be able to record video via attached camera, have display interactions, and show the temperature. The purpose is to give you a powerful starting point for creating your own applications. You will interact with the GPIO and the camera port and learn how to build and deploy applications to the board. One of the biggest hurdles will be how to run multiple processes at once in Rust on the board to perform heartbeats, face monitoring, and receive input.

The set of applications we will build for the board are as follows:

1. Face recognition video recording
    a. Background uploading of video
2. Communicate with MQTT
    a. Send heartbeat
    b. Receive recording commands
3. LED display
    a. Display pictures for holidays
    b. Display device code for authentication
    c. Display temperature
4. Homekit integration
    a. Display temperature
    b. Display motion detection

Basic Rust

While I mentioned a few other resources for learning Rust, I feel I'd be remiss if I did not at least cover a basic introduction and touched on topics and language syntax that you will see throughout the book. As software developers, and especially modern-day software developers, switching languages is part of our everyday job; as a community, we keep evolving to solve new problems. In this final section of Chapter 1, we are going to discuss the Rust language, its syntax, and its features and go over some code samples which will help you understand the language. If you already are comfortable with Rust, you can skip this section and start Chapter 2; if not, read on.

Rust Origins

Rust is not a new language but rather has been around since 2008 but until recently got popular in the main stream. It was started by and still the biggest contributors to it are Mozilla. It was mainly used as a language for the Mozilla browser engine. Rust syntactically is like C/C++ with the standard curly brackets and language syntax.