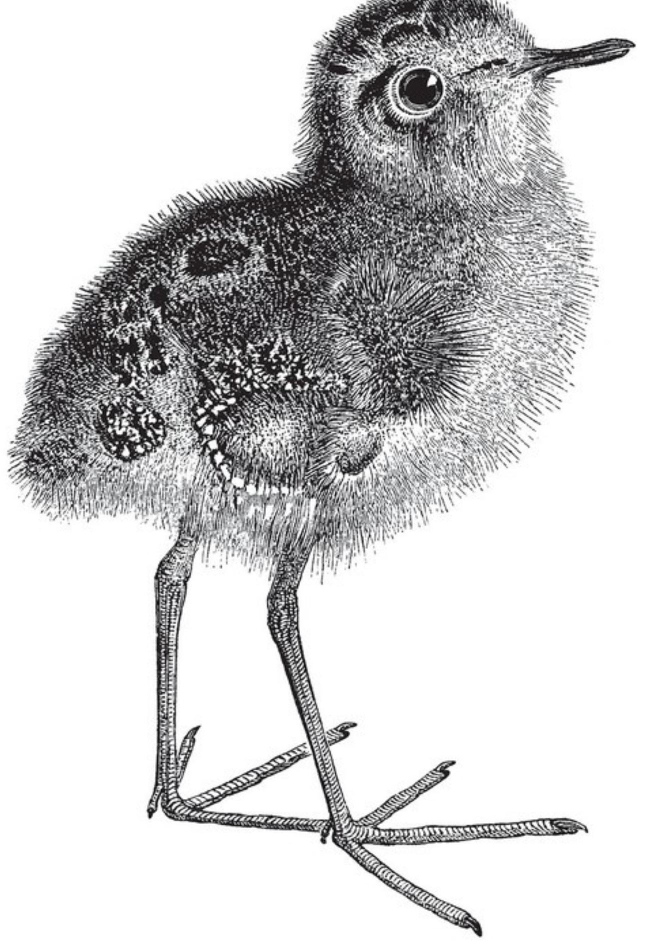
O'REILLY®

Programming the Internet of Things

An Introduction to Building Integrated, Device to Cloud IoT Solutions



Early Release
RAW & UNEDITED

Andrew King

Programming the Internet of Things

An Introduction to Building Integrated, Device to Cloud IoT Solutions

Andrew King

Programming the Internet of Things

by Andrew King

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Editors: Melissa Duffield and Sarah Grey

Production Editor: Christopher Faucher

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: O’Reilly Media, Inc

July 2021: First Edition

Revision History for the Early Release

- 2020-08-10: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492081418 for release details.

978-1-492-08134-0

Preface

The Internet of Things (IoT) is a complex, interconnected system-of-systems composed of different hardware devices, software applications, data standards, communications paradigms, and cloud services. Knowing where to start with your own IoT project can be a bit daunting. *Programming the Internet of Things* is designed to get you started on your IoT development journey, and shows you, the developer, how to make the IoT work.

If you decide to stay with me through this book, you'll learn how to write, test and deploy the software needed to build your own basic, end-to-end IoT capability.

Who Is This Book For?

Programming the Internet of Things is written for programmers by a programmer. It’s also designed for any technologist (programmer or not) interested in learning about, and building, end-to-end IoT solutions, as well as college-level instructors teaching their own connected devices course.

This is not designed as a reference book, nor does it provide an exhaustive deep dive into and analysis of the various protocols commonly used for communicating amongst IoT devices and cloud services. There are extremely well-written specifications for this purpose (although we’ll certainly dive into some of these details to better understand these protocols).

Those who are interested in practicing what is learned in each chapter can move straight into the exercises at the end of each chapter, while those interested more in the concepts and general knowledge can simply move on to the next section. I’ve organized this book so that each of these three groups - Programmers, Instructors, and Information Technology (IT) Executives - can benefit.

To the Programmer

If you’re embarking on your own IoT learning journey as a practitioner, I’m assuming you’re mostly interested in expanding your skillset, as you’ve witnessed the growth of IoT opportunities and want to be part of this important technology evolution. Over the years, I found the step-by-step, “build from the ground up" approach to implementing integrated and open IoT solutions to be most helpful in understanding the complexities of this area, and so I follow this model throughout the book.

The programming examples you'll encounter throughout this book are constructed from my own journey in learning about the IoT, and have evolved to represent many of the lab module assignments from the graduate-level Connected Devices course I teach as part of Northeastern University's Cyber Physical Systems program.

Each chapter and exercise builds upon the previous, so if you're starting out with the IoT, I'd recommend walking through each as-is, in the order given. If you're experienced in this space and are using this book as a reference guide, you may consider incorporating specific lessons of interest into your own application.

If you're new to the IoT and unfamiliar with how an end-to-end IoT system comes together through software, I'd recommend walking through the book in order, working through each exercise, and consider application customizations only after you've mastered each chapter.

To the Instructor

The contents that form the underpinnings of this book have been used successfully in my graduate-level Connected

Devices course for the past three years. This book is the formalization of these lecture notes, presentations, examples, and lab exercises, structured in much the same way as my course.

The original class was designed as an introduction to the IoT, but with student input and suggestions from my Teaching Assistants, it quickly morphed into an advanced, project-oriented software development course designed as one of the final required classes for students wrapping up their Master's in the Cyber Physical Systems department of Northeastern University. The goal of the course is to establish a strong baseline of IoT knowledge to help students move from academia into industry, bringing foundational IoT skills and knowledge into their respective organizations.

This book is structured so it can be used as a reference guide or even as a major component of a complete curriculum to help you with your own IoT-related course. The content is focused on constructing an end-to-end, open and integrated IoT solution, from device to cloud, using a 'learn as you build' approach to teaching. Since each chapter builds upon the previous, you can use this book to guide your students in building their own platform from the ground up.

You can remain up to date on exercises and other relevant content useful for teaching and explaining concepts on the book's website (https://programmingtheiot.com/programming-the-iot-book/).

To the Technology Manager and Executive

The contents of this book should help you better understand the integration challenges inherent with any IoT project and provide insight into the skillsets required to help your technology team(s) succeed across your IoT initiatives.

If you’re part of this group, my assumption is that you’re mostly concerned with understanding this technology area as a whole – its integration challenges, dev team setup requirements and needs, team skillsets, business user / stakeholder concerns around the IoT, and the potential change management challenges you may encounter as you embark on your organization’s IoT journey.

What Do I Need To Know?

Although the exercises in this book assume that you have experience as a programmer, most do not require sophisticated programming skills or a formal computer science background. However, if you intend to complete the exercises at the end of most chapters, you’ll need to possess a basic level of comfort working with both Python and Java as coding languages to build simple applications comfortably; working in an Integrated Development Environment (IDE); reading, writing, and executing Unit Tests; and configuring Linux-based systems via a shell-based command line.

All exercises are preceded by a to-be design diagram for the specific task at hand, which provides specific detail on the way any new logical components you build are to work together along with the existing components you've already developed. Most of these diagrams will follow the guidelines specified by the Unified Modeling Language (UML)[1], and incorporate other non-UML design constructs in order to make the design as clear as possible. A legend is included within those diagrams where I deemed it will add sufficient value, and excluded where I felt it just adds noise.

For technology executives and managers, you don't need to implement the exercises yourself, but it will be helpful to read through the entire book so you understand the challenges your team will likely encounter.

For business stakeholders interested mostly in understanding what the IoT entails, I recommend reading – at a minimum – the Overview section of each chapter, then focusing your efforts on the final chapter, which discusses a handful of practical cases, scenarios, and implementation suggestions.

How Is This Book Arranged?

This book will take you through building an end-to-end, full-stack, and integrated IoT solution using various open source libraries and software components that you’ll build – step-by-step. Each of these components will be part of the larger system, and you’ll see how they interconnect with and map into an end-state architecture as you work through each chapter’s exercises.

If you’re not planning on implementing the exercises and just want to get your feet wet with various IoT technologies, feel free to skip around to those sections that are most interesting to you and relevant for your needs.

I’ve grouped like chapters together and used this scheme to establish the four parts of the book: Part I - Getting Started, Part II - Connecting to the Physical World, Part III - Connecting to Other Devices, and Part IV - Connecting to the Cloud. Let’s review each in a bit more detail.

Part I – Getting Started

In this section, we build our initial foundation for IoT development. You'll start by creating a development and testing environment, and wrap-up the section by writing two simple applications to validate our environment is working properly.

- Chapter 1 is the longest chapter in the book as it lays the groundwork for how you'll build your end-to-end solution. It will help you establish a baseline of IoT knowledge and set up your workstation and development environment so you can be productive as quickly as possible. I'll cover some basic IoT terms, create a simple problem statement, define core architectural concepts, and establish an initial design approach that I'll use as the framework for each subsequent exercise.

NOTE

The IoT consists of a plethora of heterogeneous devices and systems, and there are many tools and utilities available to support development and system automation. The tools and utilities I use throughout the book fall into the open source category, and represent a small subset of those available to you. These shouldn't be taken as carte blanche recommendations, of course, as you may have your own that you prefer to work with and are best suited for your specific needs. My goal is only to keep the content well-bounded, and to help inform a generalized development and automation approach to help you be successful implementing the exercises in this book.

- Chapter 2 covers your development environment setup and requirements capture approach, and then moves into coding. In this chapter, you’ll create your first two IoT applications - one in Python and the other in Java. These will be quite simple, but set the stage for subsequent chapters. Even if you’re an experienced developer, it’s important to work through the exercises as given so we’re working from the same baseline going forward.

Part II – Connecting to the Physical World

This section covers a fundamental characteristic of the IoT – integration with the physical world. You'll learn how to read sensor data and trigger an actuator both virtually using emulators and on an actual IoT device.

- Chapter 3 explores ways you can collect data from the physical world (sensing), and trigger actions based on that data (actuation). You'll start by building a set of simple emulators which you may continue using throughout each subsequent exercise. These simple emulators will help you simulate real sensors and permit you to trigger actuation events.
- Chapter 4 builds upon Chapter 3 by discussing ways to connect to a small sampling of real sensors and actuators if you choose to build and run your applications on one or more physical devices. This chapter focuses on the Raspberry Pi, and assumes the hardware is running the Raspbian operating system.
- Chapter 5 discusses telemetry and data formatting. I'll discuss ways to structure your data so it's easy to store, transmit, and understand by both humans and machines. This will serve as the foundation for your interoperability with other 'things'.

Part III – Connecting to Other Things

This is where the rubber meets the road, so to speak. This section focuses on integration across devices, as to be truly integrated, you'll need a way to get your telemetry and other information from one place to another. The chapters in this section focus on learning about and utilizing application layer protocols designed for use within IoT ecosystems. I'll assume your networking layer is already in place and working, although I'll discuss a few wireless protocols along the way.

- Chapter 6 introduces Publish / Subscribe protocols - specifically Message Queuing Telemetry Transport (MQTT) and other related standards often used in IoT applications. I'll walk through a select set of specifications and explain how you can begin building out a simple abstraction layer that allows you to easily interface with common open source libraries.
- Chapter 7 is nearly identical to Chapter 5, except it focuses on Request / Response protocols - specifically the Constrained Application Protocol (CoAP) and other standards, also commonly used in IoT applications.
- Chapter 8 explores the use of both Publish / Subscribe and Request / Response protocols together. You'll integrate MQTT and CoAP as part of your solution, and see how these two protocols complement one another.
- Chapter 9 introduces yet another protocol - the OPC UA (Open Platform Communications Unified Architecture) - giving you the ability to use both the publish / subscribe and request / response

communications paradigms through a single protocol standard.

Part IV – Connecting to the Cloud

Finally, at the ‘top’ of the integration stack, you’ll learn how to connect all your IoT device infrastructure to the cloud by using your gateway application to serve as the go-between for your cloud functionality and all your devices.

This section covers basic cloud connectivity principles, and lightly covers the various cloud services that can store, analyze, and manage your IoT environment. In each case, you’ll build the same simple cloud application across each platform.

- Chapter 10 considers various IoT-centric cloud services that are built on open source platforms that can either be re-hosted in your own environment or accessed via a free or paid tier.
- Chapter 11 provides an overview of various commercial cloud services that have IoT-specific capabilities. As there are many books and tutorials available for these platforms, I won’t go into detail on building anything specific; this chapter will simply serve as a review of these capabilities.
- Chapter 12 examines a few simple IoT use cases that I’ve found particularly helpful in preparing my Connected Devices course at Northeastern University. Specifically, I’ll cover the overall problem statement, expected outcome, design approach, and target solution for a home environmental monitoring system, pet monitoring system, and hydroponic garden management system.

In each chapter, I'll provide a brief introduction to the topic along with some helpful background material, which will include some pertinent definitions that should be useful in understanding the concepts I'll discuss. I'll also summarize why each of these topics is important and what you can expect to learn. Each chapter ends with some hands-on implementation exercises to help you take the information presented and apply it to your own application.

My hope is that this approach will allow you to understand, and create, an integrated IoT system - end to end. It's also designed so you can skip to the chapters that are most relevant for you as more of a reference guide, so you can skip the exercises if you're not interested in coding a solution, although I do recommend you give them a try! If you do decide to delve into each lab module, please keep in mind that they are cumulative in that the work you do for each chapter will generally be used – or referenced – in a later chapter.

Some Background on the IoT

It may be helpful to look at just what enabled these ecosystems, and so here's a brief summary of how we got to this point. Computing took a big step forward with the invention of the transistor in the 1950's, followed in the 1960's by Gordon Moore's paper describing the doubling of transistors packed in the same physical space (later updated in the 1970's) [2].

With modern computing came modern networking, and the beginnings of the Internet with the invention of the ARPAnet in 1969 [3]. This led to new ways to chunk, or packetize, data using the Network Control Protocol (NCP) and Transmission Control Protocol (TCP) via the Internet Protocol (IP) and leveraging existing wired infrastructure in the 1970's. This was useful for industry, allowing electrical industrial automation to move down the path of centralized management of distributed, connected systems. Supervisory Control and Data Acquisition (SCADA) systems emerged from their proprietary roots, and Programmable Logic Controllers (PLC's) were developed to take advantage of TCP/IP networking and related equipment standards.[4]

Eventually, we arrived at the 1980's with the introduction of User Datagram Protocol (UDP) [5] and the birth of what many of us experienced as the early modern Internet - the World Wide Web (WWW) - invented in the late 1980's by Tim Berners-Lee [6].

I'm sure you've noticed a common theme: A problem is followed by a technology innovation (often proprietary in nature) to address the challenge, which then becomes standardized (or is superseded by one or more standards), leading to wide adoption and further innovation.

This brings us to the era of the IoT, where in the late 1980's and early 1990's we begin to see the first connected devices emerge, including the demonstration of an Internet-connected toaster, demonstrated by John Romkey and Simon Hackett at 1990's Interop show [7]. Shortly thereafter, a web camera was set up to monitor the coffee pot near the Trojan Room at the Computer Lab at the University of Cambridge in 1991 [8]. Because who wants to make a trip to the coffee machine and find the coffee pot empty? I

More devices followed, of course, and I'd guess even more were built and connected as experiments in college labs, dorms, homes, apartments, and businesses. All the while, computing and networking continued to become more inexpensive, powerful, and - of course - smaller. In 1999, Kevin Ashton presented the "Internet of Things" at Proctor & Gamble [9], which is widely believed to be the first coining of the phrase.

Fast forward to 2005, and the Interactive Design Institute in Italy gives us the inexpensive and designed-for-novices Arduino Single Board Computer (SBC) [10], opening the door for more people to start building their own sensing and automation systems. Add easily accessible storage and the ability to analyze data through services reachable from anywhere on the Internet, and we have the underpinnings of an IoT ecosystem; that is, a lot of different things that may be individually unique, but can be connected to each other in such a way as to serve a larger purpose.

In this sense, I believe it's best not to view the IoT as a bunch of things that connect the physical world and the Internet to do useful work. The essence of the IoT is *heterogeneity*: lots of dissimilarity in terms of device types, features and capabilities, and purposes as well as implementation approaches, supported protocols, security, and management techniques.

Complexity Redefined

So, what *exactly* is the ‘Internet of Things’? It’s a complex set of technology ecosystems that connect the physical world to the Internet using a variety of *edge computing devices* and *cloud computing services*.

For the purposes of this book, edge computing devices represent the embedded electronics, small computers and software applications that either interact directly with the physical world through sensors and actuators or provide a gateway, or bridge, for those devices and applications to connect to the Internet. Cloud computing services represent the computing systems, software applications, data storage, and other computing services that live within one or more data centers and are always accessible via the Internet.

To be sure, this is an oversimplification of two complex, and often overloaded, technical terms. My intent is to provide a relatively simple categorization of the key computing resources for which to discuss the IoT and the integrated system you’ll build as you work through the exercises in this book.

With that said, and for the remainder of the book, I’ll focus on the IoT in terms of architecture and purpose; that is, how an

IoT system is designed, and the outcome the design is expected to achieve.

The first step is to map these two technical terms of edge computing devices and cloud computing services into *architectural tiers*, where edge computing devices are part of the *edge tier*, and cloud computing services are part of the *cloud tier*. This provides both physical and logical separation of the key functionality of an IoT system, meaning, for example, that all sensing and actuation will take place in the edge tier, and all long-term storage and complex analytics will take place in the cloud tier.

This separation of tiers represents a high-level depiction of the systems architecture, or plan, that I'll use going forward to describe the composition of the various physical and logical components comprising an integrated IoT system.

NOTE

The Industrial Internet Consortium (IIC) has published a variety of useful documents on a subset of the Internet of Things, called the Industrial Internet of Things (IIoT). One such publication, the Industrial Internet Reference Architecture (located here https://www.iiconsortium.org/IIRA.htm), discusses a framework and common vocabulary for IIoT systems, and has heavily influenced my thinking on the topic of IoT architecture.[11]

The real value in any IoT system is its ability to provide measurement and analysis. When sufficient measurements are collected, and with sufficient granularity and samples, an analysis can be performed to determine how a system is performing. The insight this analysis provides helps determine if the system is achieving its desired outcomes, or if a correction needs to be made.

If, for example, I can measure the inside temperature of a refrigerator every minute, I can determine how long the items stored in the refrigerator have been exposed to a given temperature. If I sample the inside temperature only once a day, I don't have sufficient detail to make that determination.

Of course, the data that I collect needs to be understandable by other systems, otherwise it's pretty much useless. Building integratable IoT solutions requires the developers of each part of an IoT system to think carefully about how their design (and data) might interact with other systems. This is why, although individually these parts may be unique and NOT co-dependent on another, it's important for you as the developer to consider how other systems (and other developers) may need to interact with your software and data to serve this larger purpose we've been discussing.

The nuances you as the developer will need to deal with at each step along the way of building an integrated IoT system are indeed significant – we can't expect plug 'n play, or even consistent behaviors from the system you receive data from.

And, to make matters worse, your code may not always work across every platform even with the best intentions of writing it generically enough to function the same way from one hardware device to another.

It’s not possible to cover all specialized platforms, of course, nor is it easy to write consistent, semi low-level code at the device level that doesn’t need to be optimized for every device. As such, I’m going to work with the understanding that, while every device may be a bit different, and have differences that we need to account for when we create (and test) our solutions, the code samples will be largely portable (with some minor exceptions) and usable across most systems that can run a Java Virtual Machine or Python 3 Interpreter.

Living on the Edge

Recall we have two architectural tiers: edge tier and cloud tier. In terms of outcome analysis, the keys to the kingdom lie in the cloud tier. *Scalability* is what gives the IoT its true power, and is the ability for a system to handle as much (or as little) as we want. For example, a scalable cloud system that supports the IoT is one where I can have a single gateway device sending it data, or thousands (or millions, or billions) without it failing.

You’ve likely read numerous reports describing the IoT as being gazillions of connected devices which will drive some exorbitant amount of business value by {pick your favorite year post-2020}. In the cloud, I (mostly) don’t care about where my services run, as long as they’re always available and can handle any workload I throw at them, no matter how many devices I have within my edge tier.

With all the scalability the cloud tier gives us, an IoT solution generally exhibits the greatest complexity at the edge, which is where most, and often all, of the heterogeneity lives. The two categories of devices this book will focus on are *constrained devices* and *gateway devices*. I’m leaving out the nomenclature of “smart devices” on purpose, since it’s becoming less clear how to best define “smart” versus “not so smart” devices. To grossly over-simplify, constrained devices have some limitations from a power, communications, and processing perspective, whereas gateway devices do not.

An example of a constrained device is a low-power (sometimes battery-operated), single-board computer (SBC), that either reads data from the environment (such as temperature, pressure, humidity, etc.), or triggers a valve to open or close when instructed to do so. An example of a gateway device is also an SBC, but much more powerful and likely requires a direct-connect power supply, has the ability to communicate with many different constrained devices, and also has enough processing power to aggregate the data from these other devices, perform some simple analytics, and then determine when any relevant data is to be sent to the cloud for further storage and processing.

Figure P-1 envisions a notional IoT systems architecture that represents the relationships between these device types within the edge tier and the services and other functionality that live within the cloud tier.

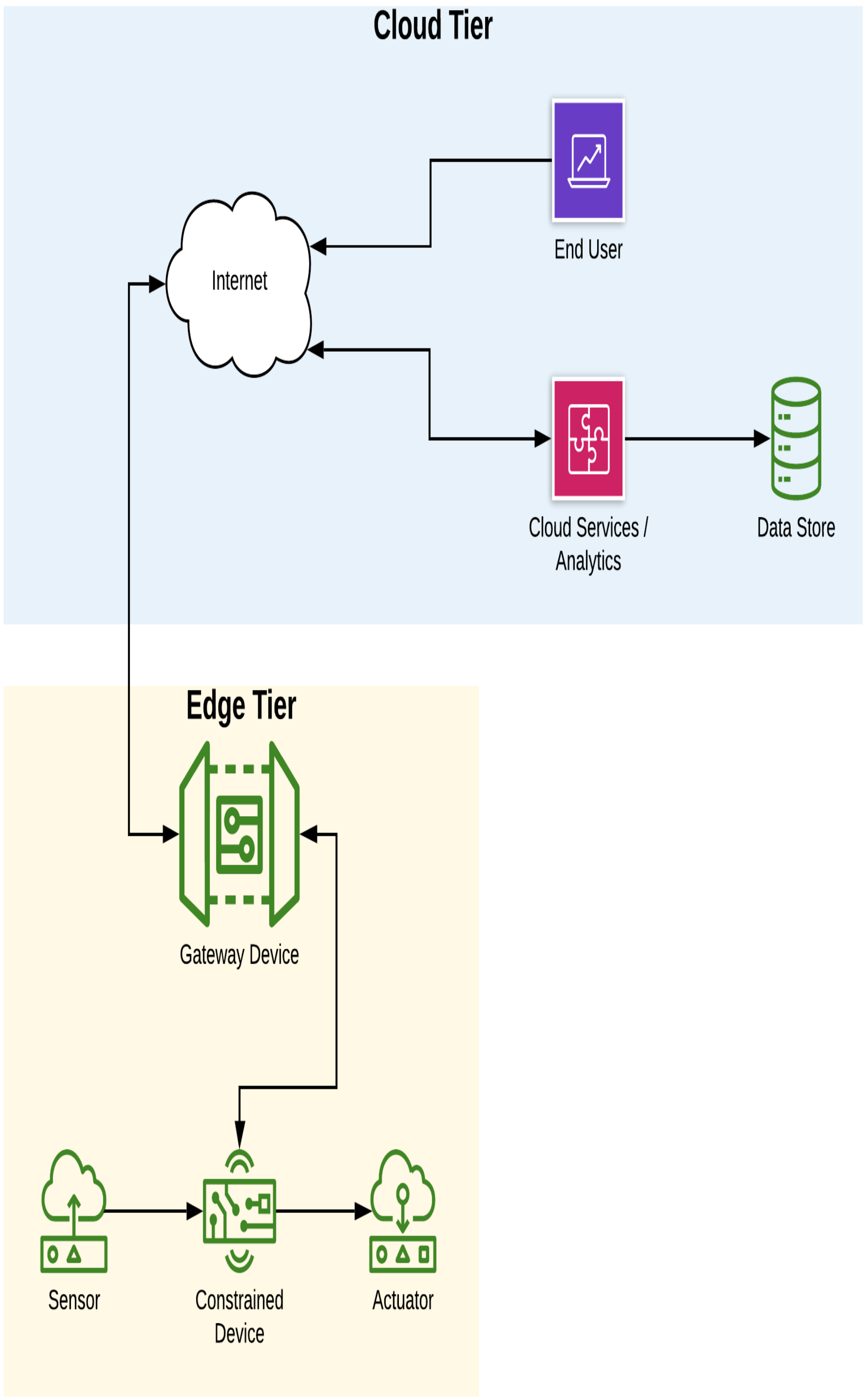

Cloud Tier
End User
Internet
Cloud Services / Analytics
Data Store
Edge Tier
Gateway Device
Sensor
Constrained Device
Actuator

Figure P-1. - Notional IoT System Architecture

I find it easiest to qualify these devices as follows:

- *Constrained devices* only handle sensing, actuation, or perhaps both, while only processing messages for themselves, passing messages along if the right communications protocol is implemented, and sending messages to a gateway device. In short, they're limited in their abilities, and don't connect directly to the Internet – only via a gateway device.
- *Gateway devices* are a super set – they can potentially do the same work as a constrained device (in that they may have sensor(s) or actuator(s) attached), but also have the ability to perform 'at the edge' analytics, determine how messages should be routed (if at all), and of course, connect directly to the Internet and the various cloud services that make the IoT useful for business stakeholders.

Can a constrained device connect directly to the Internet? Sure, if it contains a TCP/IP stack, has a routable IP address that's accessible to and from the public Internet, and of course has the appropriate communications hardware to talk to, and is connected with, the Internet. For the purposes of this book, however, I'll narrow the category of constrained devices to these two limitations:

- They do not support packet routing directly to or from the public Internet (although we'll assume they support both TCP/IP and UDP/IP) and must interact with a gateway device in order to be part of any IoT ecosystem.

- They do not contain adequate computing resources to intelligently determine complex courses of action based on the data they collect.

What it all means is this: As we make better computing devices that are smaller, faster, and cheaper; use them to interact with the physical world, and connect them (or their data) to the Internet for processing using cloud services, we can derive insights that help deliver better business outcomes.

At the end of this book, I'll explore some use cases like the one just discussed, but in more detail. Each use case will include an implementation exercise that will allow you to explore the concept using the knowledge you've gleaned from previous chapters and the code you'll develop as part of the exercises.

One last point before delving into our first section: Each chapter begins with a haiku that attempts to capture the essence of what you'll learn and some of the challenges you'll likely encounter. How did this come to be? From my early days as a software developer, one of the team's I worked with had a policy that if you committed code that caused the nightly build to break, you'd have to write a haiku related to the issue and e-mail it to everyone on the team. You may not encounter 'broken nightly builds' much by working through the exercises in this book, but the haiku's should at least provide some levity before you dig into each chapter.

Thanks for reading!

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

`Constant width italic`

Shows text that should be replaced with user-supplied values or by values determined by context.

TIP

This element signifies a tip or suggestion.

NOTE

This element signifies a general note.

WARNING

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at https://github.com/oreillymedia/title_title.

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and

ISBN. For example: “*Programming the Internet of Things* by Andrew King (O’Reilly). Copyright 2021 Andrew King, 978-1-492-08141-8.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O’Reilly Online Learning

> **NOTE**
>
> For more than 40 years, *O’Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O’Reilly’s online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O’Reilly and 200+ other publishers. For more information, visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O’Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and information about our books and courses, visit http://oreilly.com.

Find us on Facebook: http://facebook.com/oreilly

Follow us on Twitter: http://twitter.com/oreillymedia

Watch us on YouTube: http://www.youtube.com/oreillymedia

---

1 The latest UML specification can be found on the Object Management Group’s (OMG) website (https://www.omg.org/spec/UML/).

2 For further reading on Gordon Moore and Moore’s Law, see https://www.britannica.com/biography/Gordon-Moore

3 https://en.wikipedia.org/wiki/ARPANET

4 S. D. Antón, D. Fraunholz, C. Lipps, F. Pohl, M. Zimmermann and H. D. Schotten, “Two decades of SCADA exploitation: A brief history,”

2017 IEEE Conference on Application, Information and Network Security (AINS), Miri, 2017, pp. 98-104, doi: 10.1109/AINS.2017.8270432.

5 Details on the IETF's RFC 768 can be found at https://tools.ietf.org/html/rfc768

6 For further reading on Tim Berners-Lee's WWW proposal, please see https://www.w3.org/History/1989/proposal.html

7 See an explanation at https://www.livinginternet.com/i/ia_myths_toast.htm

8 See an explanation at https://en.wikipedia.org/wiki/Trojan_Room_coffee_pot

9 You can read more about Kevin Ashton's "Internet of Things" presentation at https://www.rfidjournal.com/that-internet-of-things-thing

10 You can read a brief summary of the Arduino's birth at https://www.computerhistory.org/timeline/2005/

11 More information about the Industrial Internet Consortium can be found on their website (https://www.iiconsortium.org/index.htm).

Part I. Getting Started

Chapter 1. Setting Up Your Environment

A path lies ahead,

Brambles and thorns, then it clears.

Almost there. Patience.

Fundamental concepts: Identify a problem area to tackle and define an architecture as the baseline for your IoT solution; Setup an IoT-centric development environment that supports multiple deployment options.

You’ve likely gathered by now that the Internet of Things can be vast, unwieldy, and very difficult to tame. To plan a way forward, we’ll first want to identify a problem area to tackle and then create an architecture from which to design and build our IoT solution.

Let’s start with a few key questions to establish a baseline: What problem are you trying to solve? Where does it start and end? Why does it require an IoT ecosystem? How will all these pieces work together to solve this problem? What outcome can you expect if everything works as designed? We’ll explore each of these in detail, and along the way construct an end-to-end, integrated IoT solution that meets our needs.

What you’ll learn in this chapter

To help you really understand how an IoT system can and should be constructed, I’ll dig into some basic architectural concepts based on the questions above, and use this as the basis for each programming activity. From there, you’ll build a solution that addresses the problem layer by layer, adding more functionality as you work through each subsequent chapter.

It goes without saying, of course, that the right development tools will likely save you time and frustration, not to mention help you with testing, validation, and deployment. There are many excellent open source and commercial development tools and frameworks available to support you.

If you’ve been a developer for any length of time, I expect you have your own specific development environment preferences that best suit your programming style and approach. I certainly have mine, and while the examples I present will be based on my preferred set of tools, my goal in this chapter is not to specify those you must use, but to help you ramp up on IoT development in a way that enables you to move out quickly and eventually choose your own for future development projects.

The concepts I present will be what matter most – the programming languages, tools (and their respective versions), and methods can be changed. These concepts represent some of the fundamentals of consistent software development: system design, coding, and testing. Let’s dig into each and get moving towards building your integrated IoT system.

Designing Your System

Creating a problem statement is probably the most important part of this puzzle. Let’s try it by drafting something reasonably straight-forward, but sufficient to encompass a variety of interesting IoT challenges:

I want to understand the environment in my home and how it changes over time, and make adjustments to enhance comfort while saving money.

Seems simple enough, but this is a very broad goal. We can narrow it down by defining the key actions and objects in our problem statement. Our goal is to isolate the ‘what’, ‘why’, and ‘how’. Let’s first look at the ‘what’ and the ‘why’, and identify any action(s) that the design should consider as part of this process.

Breaking Down The Problem

The exercises in this book will focus on building an IoT solution that can help you understand your home environment and respond appropriately. The assumption is that you'll want to know what's going on within your house (within reason), and take some sort of action if it's warranted (that is, if the temperature is too hot, turn on the air conditioning). Simple, right?

This part of your design approach requires three steps:

1. Collect Data

   Let's define this in terms of what can be sensed, like temperature, humidity, etc. This is centered on the capture and transmission of *telemetry* (measurement data). The action, or rather, action category, will simply be named ***data collection***

   - Temperature
   - Relative Humidity
   - Barometric Pressure
   - System Performance (utilization metrics for CPU, memory, storage)

2. Determine Relevant Changes

   To decide which data is relevant, and whether or not a change in value is important, we need to not only collect data, but to store, and trend, time-series data on the items we can sense (like temperature, humidity, etc, as indicated above). Let’s refer to this action category as ***data management***.

3. Take Action

   We’ll establish some basic rules to determine if we’ve crossed any important thresholds, which simply means we’ll send a signal to something if a threshold is crossed that requires some type of action (e.g. turn up / down a thermostat). We’ll refer to this action category as ***system triggers***.

Defining Relevant Outcomes

Now that we know what steps we need to take, let’s explore the ‘why’ portion of our problem statement. We can summarize this using the following two points:

- Increase Comfort

  Ideally, we’d like to maintain a consistent temperature and humidity in our living environment. Things get a bit more complicated when we consider the number of rooms, how they’re used, etc. We’ll refer to this action category as ***configuration management***, and it goes hand-in-hand with both data management and system triggers.

- Save Money

  This gets a bit tricky. The most obvious way to save money is to not spend it! Since we’ll likely need to allocate financial resources to heat, cool, or humidify a given area, we want to optimize - not too much (wasteful), and not too little (we could end up with frozen water pipes in the winter). Since we might have some complexity to deal with here - including utility costs, seasonal changes, etc., as well as anything related to configuration management - we'll probably need some more advanced analytics to handle these concerns. We'll call this action category ***analytics***.

You've likely noticed that each of the steps in the 'what' and 'why' sections above have an action category name that will help with the solution design once we move onto the 'how'. As a reminder, these categories are: data collection, data management, system triggers, configuration management, and analytics. We'll dig further into each of these as part of our implementation approach.

Although the problem statement seems rather simple on the surface, it turns out that the things you'll need to do to address the problem are actually quite common within many IoT systems. There's a need to collect data at its source, store and analyze that data, and take action if some indicator suggests that would be beneficial. Once you define your IoT architecture and start building the components that implement it - even though it will be specific to this problem - you'll see how it can be applied to many other problem areas.

Let's take a quick look at a simple data flow that represents this decision process in Figure 1-1. You'll notice each action category is highlighted in the data flow diagram depicted in Figure 1-1:

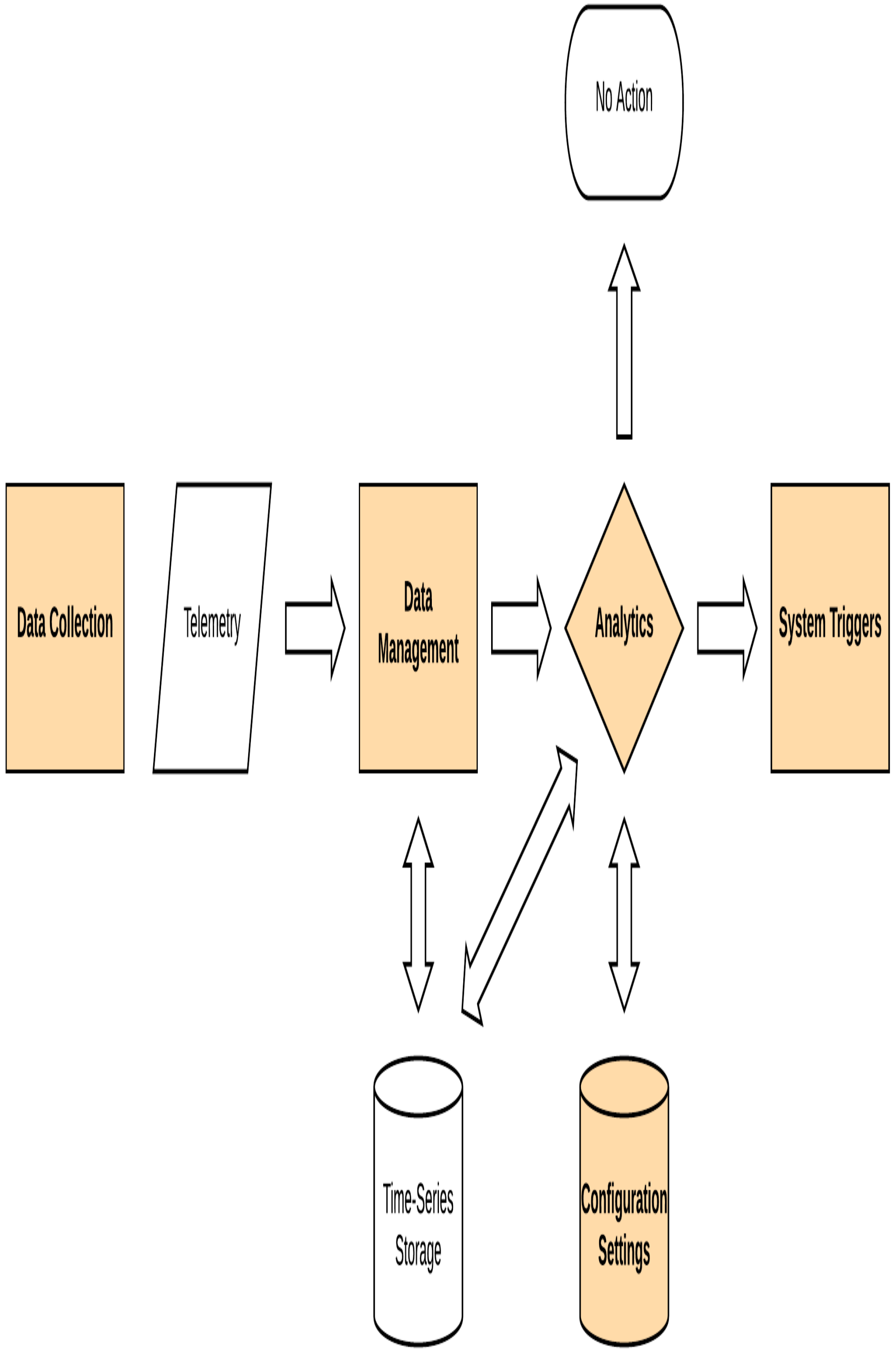

No Action
Data Collection
Telemetry
Data Management
Analytics
System Triggers
Time-Series Storage
Configuration Settings

Figure 1-1. Simple IoT data flow

Most IoT systems will require at least some of the five action categories I’ve called out: data collection, data management, analytics, configuration management, and system triggers. This means we can define an architecture that maps these into a systems diagram, and then start creating software components that implement part of the system.

This is where the fun starts for us engineers, so let’s get going with an architecture definition that can support our problem statement (and will, in fact, be reusable for others).

Architecting a Solution

Organization, structure, and clarity are hallmarks of a good architecture, but too much can make for a rigid system that doesn’t scale well for future needs. And if we try to establish an architecture that will meet all our plausible needs, we’ll never finish (or perhaps even never get started)! It’s about balance, so let’s define the architecture with future flexibility in mind, but also keep things relatively well bounded. This will allow you to focus on getting to a solution quickly, while still permitting updates in the future. But first, there are a few key terms that need to be defined to help establish a baseline architectural construct to build your solution upon.

As you may recall from Figure 0-1 in the Preface, IoT systems are generally designed with at least two (sometimes three or more) architectural tiers in mind. This allows for the separation of functionality both physically and logically, which permits for flexible deployment schemes. All of this is to simply say that the cloud services running within the cloud tier can - technically speaking - be anywhere in the world, while the devices running within the edge tier must be in the same location as the physical systems that are to be measured. Just as Figure 0-1 implies, an example of this tiering may include a constrained device with sensors or actuators talking to a gateway device, which in turn talks to a cloud-based service, and vice-versa.

Since we need a place for these five categories of functionality to be implemented, it’s important to identify their location within the architecture so we can have some things running close to where the action is, and others running in the cloud where you and I can access (and even tweak) the functionality easily. Recalling the edge tier and cloud tier architecture from the Preface, let’s see how to map each of the action categories from the ‘what’ and ‘why’ into each:

- Edge Tier (constrained devices and gateway devices): Data collection, data management, device triggers, configuration management, and analytics.
- Cloud Tier (cloud services): Data management, configuration management and system analytics.

Why do the Edge Tier and Cloud Tier include similar functionality? First, let’s take a look at how our flow chart fits within a tiered architecture:

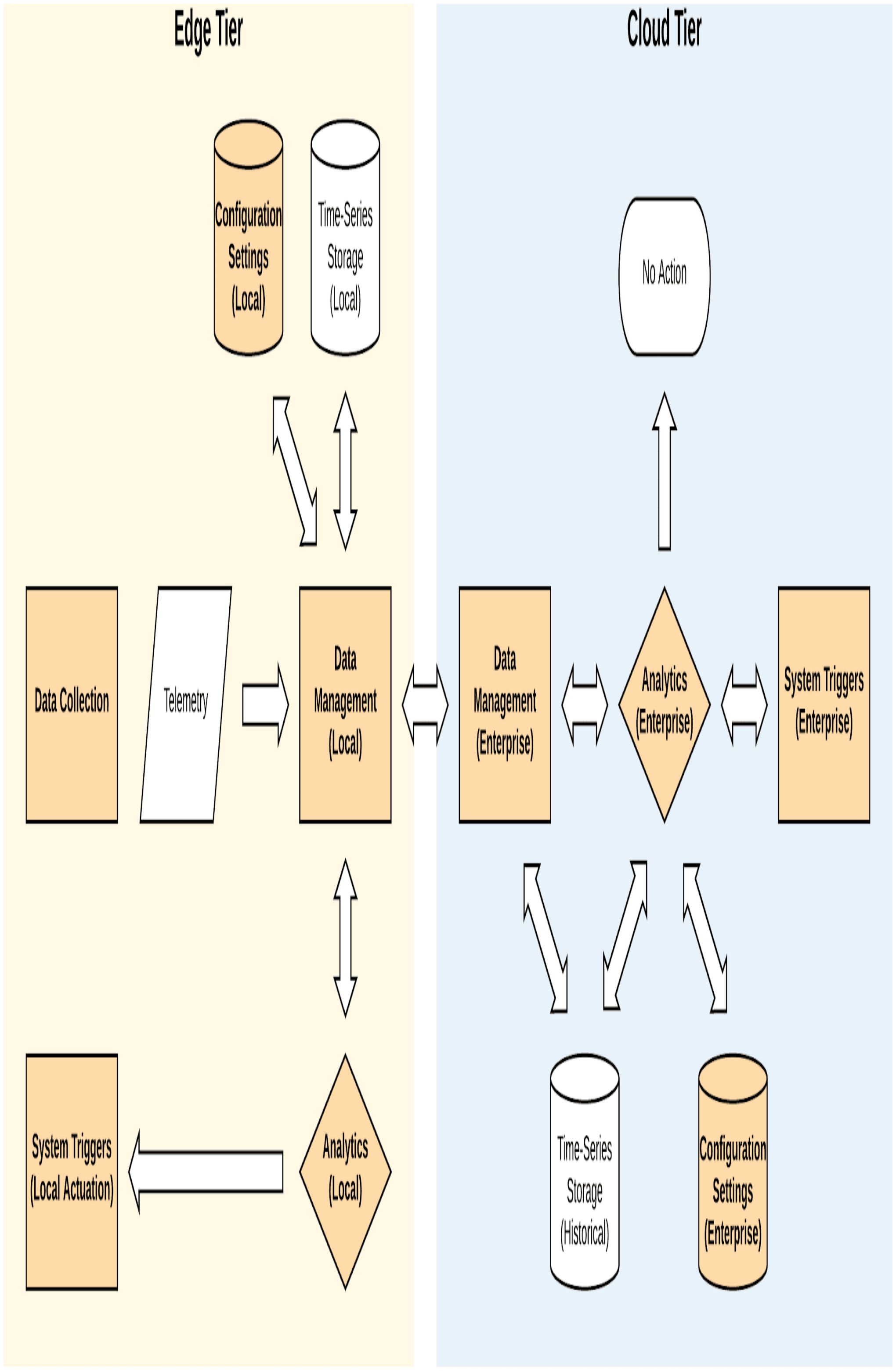

Edge Tier
Cloud Tier
Configuration Settings (Local)
Time-Series Storage (Local)
No Action
Data Collection
Telemetry
Data Management (Local)
Data Management (Enterprise)
Analytics (Enterprise)
System Triggers (Enterprise)
System Triggers (Local Actuation)
Analytics (Local)
Time-Series Storage (Historical)
Configuration Settings (Enterprise)

Figure 1-2. - Notional IoT data flows between the Edge and Cloud Tiers

Again, notice that we have some shared responsibility where some of the action categories are implemented within both tiers. Normally, duplication of effort is a bad thing - but in this case, it can be an advantage! Simple analytics can be used to trigger a device based on some simple settings if warranted - for example, if the temperature in your home exceeds 80o F, you'll probably want to trigger the HVAC straight away and start cooling things down to, say, 72o F. There's no need to depend on a remote cloud-based service in the Cloud Tier to do this, although it would be useful to notify the Cloud Tier that this is happening, and perhaps store some historical data for later analysis.

Our architecture is starting to take shape. Now we just need a way to map it to a systems diagram so we can interact with the physical world (using sensors and actuators). It would also be good to structure things within our Edge Tier to avoid exposing components to the Internet unnecessarily. This can be managed using a constrained device with a specialized software application that can either run directly on the device, or on a laptop or other generic computing system with simulation logic that can emulate sensor and actuator behavior.

Since you'll want to access the Internet eventually, your design should include a gateway to handle this and other needs. This functionality can be implemented as part of a software application that runs on a gateway device (or, again, a laptop or other generic computing system). Your gateway device and constrained device will comprise the ‘edge’ of your IoT design, which I’ll refer to as the edge tier of your architecture going forward.

We’ll also want to deploy analytics services, storage capabilities, and event managers in a way that’s secure, but accessible from our gateway device and also by human beings. There are many ways to do this, but we’ll focus on one (or more) cloud services for this functionality.

Figure 1-3 provides a new view that will provide further insight into what you’re going to build and how you can begin incorporating the five action categories I’ve already mentioned. It represents, in grey boxes, cloud services within the Cloud Tier, and two applications within the Edge Tier - CDA and GDA - that will be implemented to contain the functionality of your constrained device and gateway device, respectively.

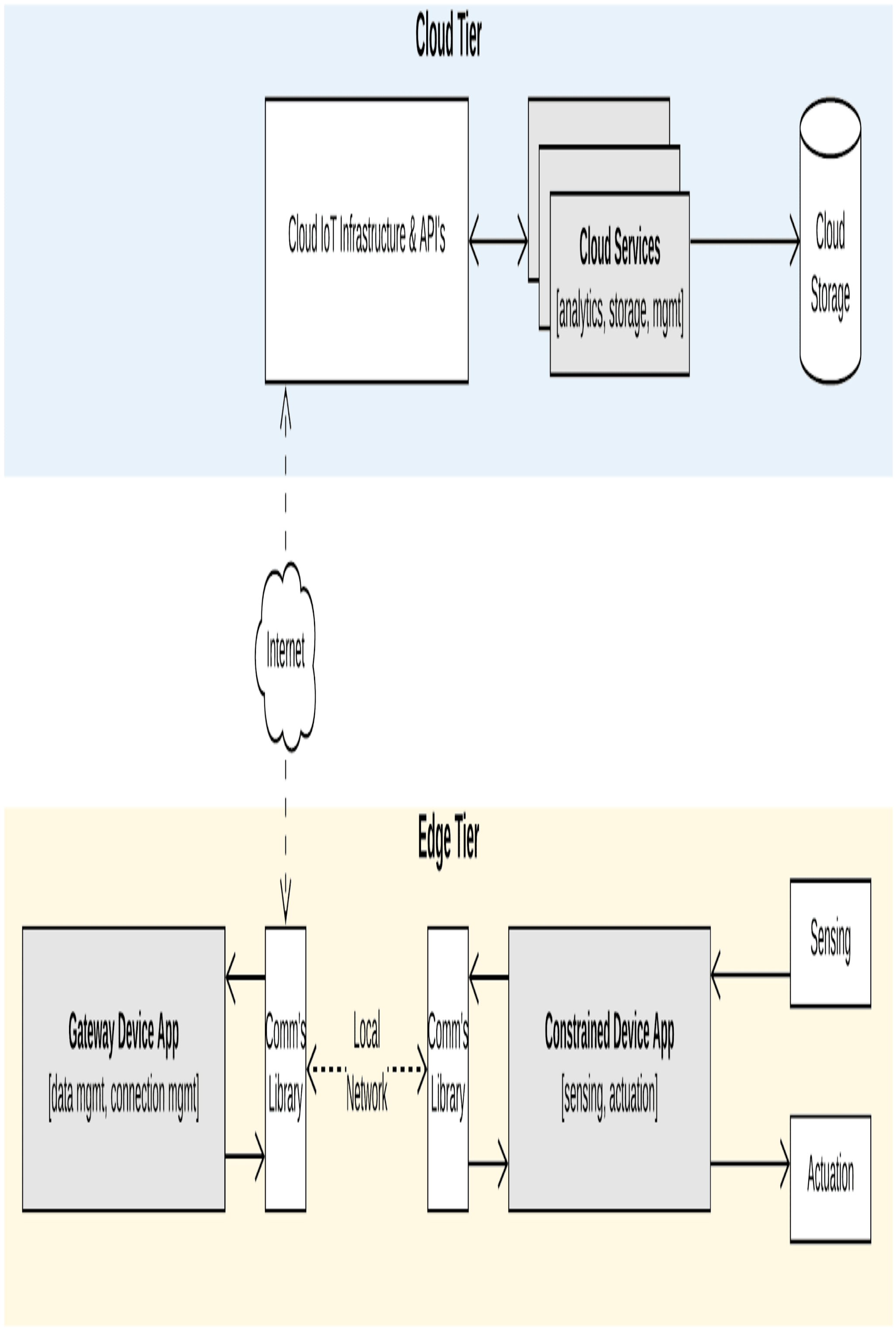

Cloud Tier
Cloud IoT Infrastructure & API's
Cloud Services
[analytics, storage, mgmt]
Cloud Storage
Internet
Edge Tier
Gateway Device App
[data mgmt, connection mgmt]
Comm's Library
Local Network
Comm's Library
Constrained Device App
[sensing, actuation]
Sensing
Actuation

Figure 1-3. - Notional IoT simplified logical architecture with Edge and Cloud Tiers

Let’s dig into each a bit further:

- Constrained Device App (CDA)

  You’ll build this software application to run within the constrained device (emulated or actual), and it will provide data collection and system triggers functionality. It will handle the interface between the device’s sensors (which read data from the environment) and actuators (which trigger actions, such as turning the thermostat on or off). It will also play a role in taking action when an actuation is needed. Eventually, it will be connected to a communications library to send messages to, and receive messages from, the Gateway Device App.

- Gateway Device App (GDA)

  You’ll build this software application to run within the gateway device (emulated or actual), and it will provide data management, analytics, and configuration management functionality. It’s primary role is to manage data and the connections between the CDA and cloud services that exist within the Cloud Tier. It will manage data locally as appropriate, and - sometimes - take action by sending a command to the constrained device that triggers an actuation. It will also manage some of the configuration settings - that is, those that represent nominal for your environment, and perform some simple analytics when new telemetry is received.

- Cloud Services

All cloud services applications and functionality do the heavy data processing and storage work, as they can theoretically scale it ad infinitum. This simply means that, if designed well, you can add as many devices as you want, store as much data as you want, and do in-depth analysis of that data - trends, highs, lows, configuration values, etc., all while passing any relevant insights along to a human end user, and perhaps even generate Edge Tier actions based on any defined threshold crossing(s). Technically, cloud services within an IoT environment can handle all the action categories previously mentioned, with the exception of data collection (meaning, they don’t perform sensing or actuation actions directly). You’ll build some cloud services to handle this functionality, but mostly utilize those generic services already available from some cloud service providers.

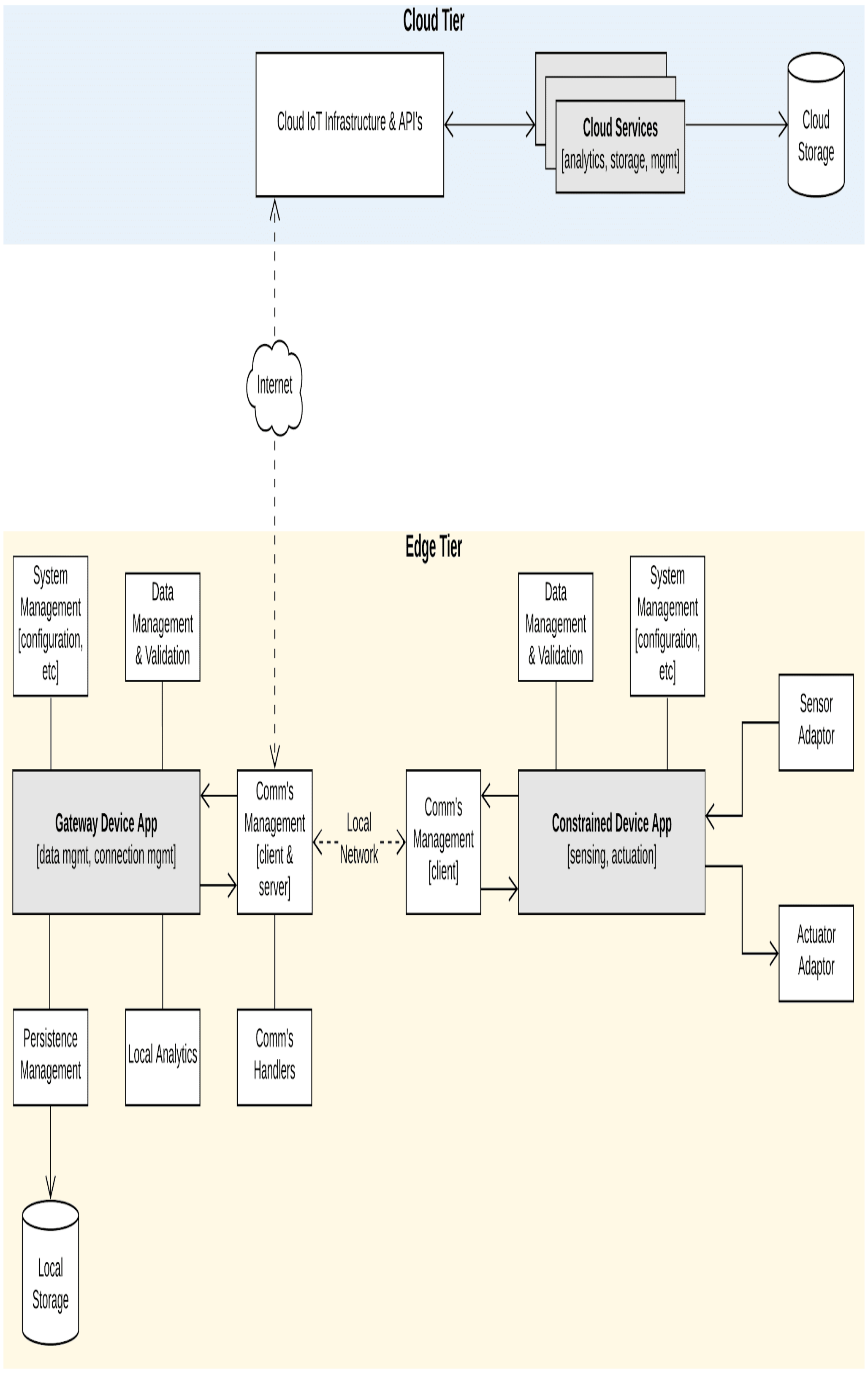

Cloud Tier
Cloud IoT Infrastructure & API's
Cloud Services
[analytics, storage, mgmt]
Cloud Storage
Internet
Edge Tier
System Management [configuration, etc]
Data Management & Validation
Data Management & Validation
System Management [configuration, etc]
Sensor Adaptor
Gateway Device App
[data mgmt, connection mgmt]
Comm's Management [client & server]
Local Network
Comm's Management [client]
Constrained Device App
[sensing, actuation]
Actuator Adaptor
Persistence Management
Local Analytics
Comm's Handlers
Local Storage

Figure 1-4. - Notional IoT detailed logical architecture with Edge and Cloud Tiers

Putting it all together into a detailed logical architecture, Figure 1-4, shows how each major logical component within our two architectural tiers interacts with the others. We'll use these two diagrams (Figure 1-3 and Figure 1-4) as our baseline architecture for all the exercises in this book.

Now that we have a handle on what we're up against, let's get our development environment setup so we can start slinging code.

Building, Testing and Deploying Software for the IoT

Building and deploying code across different operating systems, hardware configurations, and configuration systems is no walk in the park. With typical IoT projects, we not only have to deal with different hardware components, but the myriad ways to develop and deploy across these platforms, not to mention the various Continuous Integration / Continuous Deployment (CI/CD) idiosyncrasies with the various cloud service provider environments we often work within.

With all these challenges, how do we even get started? First things first - what problem are we trying to solve? As a developer, you want to implement your IoT design in code, test it, package it into something that can be easily distributed to one or more systems, and deploy it safely. We can think of our development challenges in two build, test and deploy phases, that just so happen to map to our architectural tiering: Edge Tier Environment and Cloud Tier Environment. We’ll dig into the functionality within the Cloud Tier beginning in Chapter 10. For now, we’ll focus just on our Edge Tier.

Although our Edge Tier will eventually have specialized hardware to deal with, we can simulate some aspects of these hardware components within our local development environment. This will make deployment much easier and is perfectly fine for all the exercises in this book except for Chapter 4, which I’ll touch on later in this chapter and is optional.

There are many ways to get up and running with IoT development, but we’ll focus on three specific paths: One is purely simulated, and as I mentioned previously is sufficient for all but Chapter 4. The other two require IoT-specific hardware and will be discussed in more depth in Chapter 4.

Integrated Simulated Deployment

> This approach doesn’t require any specialized device, and allows you to use your development workstation (laptop) as both gateway device and constrained device. This means you’ll run your GDA and CDA in your local computing environment. You’ll emulate your sensing and actuation hardware by building simple software simulators to capture this functionality within your CDA. All exercises, with the exception of those in Chapter 4, will work using this deployment approach.

Separated Physical Deployment

This requires a hardware device, such as a Raspberry Pi, that meets the hardware criteria listed in the below, giving you the ability to connect to and interact with real sensors and actuators. Although many off-the-shelf single-board computing (SBC) devices can be used as full-blown computing workstations, I'll refer to this as your constrained device, and it will run your CDA directly on the device. As with the Simulated approach, you'll run the GDA in your local computing environment.

NOTE

The Internet Engineering Task Force's (IETF) Request For Comments document RFC 7228 defines various classes of constrained devices (also referred to as constrained nodes). These classes include Class 0 (very constrained), Class 1 (constrained), and Class 2 (somewhat constrained). [1] For our purposes, we'll assume our CDA can run on Class 2 or even more powerful devices, which typically support full IP-based networking stacks, meaning the protocols we'll deal with in this book will generally work on these types of devices. Although technically feasible to connect Class 2 devices directly to the Internet, all of our examples and exercises will interact indirectly with the Internet via the GDA.

Blended Physical Deployment

This approach is nearly identical to the Separated Deployment approach, but will run both your CDA and GDA on the SBC device (e.g. Raspberry Pi or other, similar system). This technically means you can choose to deploy each app to a different SBC, although it isn't necessary for any of the exercises listed.

If you choose either Path B or C for your deployment, there are a wide range of inexpensive SBC's that could work for you. Like choosing our device should have general purpose input/output (GPIO) functionality, at least one available Inter-Integrated Circuit (I2C) bus, support TCP/IP and UDP/IP networking via Wi-Fi or Ethernet, run a general purpose Linux-based operating system such as Debian (or a derivative such as Raspbian), and support Python 3 and Java 11 (or higher).

Although you may choose any SBC that works for your needs, the exercises in the book will focus on the Integrated Simulated Deployment Path, excepting Chapter 4 of course. Part II introduces the concept of integration with the physical world, and I'll devote Chapter 4 to this type of integration using actual hardware. While the principles will apply to many SBC's, any hardware-specific integration will focus on the Raspberry Pi platform - specifically, the Raspberry Pi 3 Model B+, Model 4 Model B, and the latest version of the Raspberry Pi Zero W.

Irrespective of the selected Deployment Path, all exercises and examples assume you'll do your development and deployment on a single workstation. This involves a three-step process that includes preparation of your development environment, defining your testing strategy, and identifying a build and deployment automation approach. I'll cover the basics of these steps to get you started in this chapter, but also add to each as you dig into later exercises that have additional dependencies, testing, and automation needs.

> **NOTE**
>
> As I mentioned previously in this section, the exceptions to the ‘develop on a single workstation’ assumption are the exercises in Chapter 4, where I’ll discuss implementation and deployment on real hardware. Again, this only applies if you are planning on following deployment path B and C, otherwise, all your Edge Tier code will deploy and run on your local development workstation.

Step I: Prepare Your Development Environment

Recall that your CDA will be written in Python, and your GDA will be written in Java. Let’s make sure your workstation has the right stuff installed to support these languages and their associated dependencies, by following the steps below.

1. Install Python 3.7 or higher on your workstation (the latest version as of this writing is 3.8.2). You can check if it’s already installed, or install it if not, using the following steps:

    - Open a terminal or console window, and type:

      ```
      $ python3 -version
      ```

    - It should return something similar to the following:

      ```
      Python 3.7.6
      ```

- If you get an error (e.g. ‘not found’), you’ll need to install Python 3.7 or higher. Follow the instructions for your operating system (Windows, Mac, Linux) at https://www.python.org/downloads/.

2. Install pip by downloading the script located at https://bootstrap.pypa.io/get-pip.py.

    a. Use Python to execute the pip installation. Open a terminal or console window, and type:

    ```
    $ python3 get-pip.py
    ```

3. Ensure Java 11 or higher is installed on your workstation (the latest version of OpenJDK as of this writing is Java 14). You can check if it’s already installed, or install it if not, using the following steps:

    a. Open a terminal or console window, and type:

    ```
    $ java -version
    ```

    b. It should return something like the following (make sure it’s at least Java 8)

    ```
    java 13.0.1 2019-10-15

    Java(TM) SE Runtime Environment (build
    13.0.1+9)

    Java HotSpot(TM) 64-Bit Server VM (build
    13.0.1+9, mixed mode, sharing)
    ```

    c. If you get an error (e.g. ‘not found’), you’ll need to install Java 11 or higher. Follow the instructions for your platform (Windows, MacOS, or Linux) here: https://openjdk.java.net/install/

4. Install Git. Go to https://git-scm.com/book/en/v2/Getting-Started-Installing-Git and review the instructions for your specific operating system.

> **NOTE**
>
> A prerequisite for any of the exercises in this book, and for setting up your development environment, is a basic understanding of Git – a source code management and versioning tool. Many IDE’s (including Eclipse) come with source code management already enabled via an embedded Git client. In a previous step, you installed Git via the command line so you can run Git commands independently of your IDE. For more information on using Git from the command line, please see https://git-scm.com/docs/gittutorial.

> **TIP**
>
> You can use Git as a standalone source code management tool on your local development workstation, and also manage your source code in the cloud using a variety of free and commercial services. GitHub [2] is the service I use, and will leverage some of the features this platform provides to support CI/CD pipelines, or workflow steps, that allow you to automate the build, test, package, and deployment process.

5. Create a working development directory, and download the source code and unit tests for this book:

a. Open a terminal or console window, create a new working development directory, and then change to that directory. Type the following:

    i. Linux/macOS:

    ```
    mkdir $HOME/programmingtheiot
    ```

    ```
    cd $HOME/programmingtheiot
    ```

    ii. Windows:

    ```
    mkdir C:\programmingtheiot
    ```

    ```
    cd C:\programmingtheiot
    ```

b. Clone the following two source code repositories for this book by typing the following:

```
$ git clone
https://github.com/programmingtheiot/pytho
n-components.git
```

```
$ git clone
https://github.com/programmingtheiot/java-
components.git
```

6. Install and configure virtualenv using pip [3]

a. Open a terminal or console window. Change your directory to that of the constraineddeviceapp code you just cloned from GitHub, and install virtualenv by typing the following:

    i. Linux/macOS:

    ```
    $ pip install virtualenv
    ```

    ii. Windows:

```
C:\programmingtheiot> pip install
virtualenv
```

b. Configure virtualenv for your environment. Change your directory to that of the constraineddeviceapp code you just cloned from GitHub, and type the following:

   i. Linux/macOS

```
$ cd
$HOME/programmingtheiot/python-
components
```

```
$ virtualenv -p python3 .venv
```

```
$ . .venv/bin/activate
```

```
(venv) $ pip install -r
basic_imports.txt
```

   ii. Windows

```
cd C:\programmingtheiot\python-
components
```

```
C:\programmingtheiot\python-
components> virtualenv -p
C:\PathToYourPythonExecutable\pytho
n.exe .venv
```

```
C:\programmingtheiot\python-
components>
.venv\Scripts\activate.bat
```

```
(.venv)
C:\programmingtheiot\python-
components> pip install -r
basic_imports.txt
```

c. Your virtualenv environment is now set up, and your terminal is running within virtualenv. You can activate (using the `activate` script) and then deactivate virtualenv (using the `deactivate` command) from your command line easily enough:

   i. Linux/macOS

   ```
   $ . .venv/bin/activate
   ```

   ```
   (venv) $ deactivate
   ```

   ii. Windows

   ```
   C:\programmingtheiot\python-
   components>
   .venv\Scripts\activate.bat
   ```

   ```
   (.venv)
   C:\programmingtheiot\python-
   components> deactivate
   ```

At this point, your development workstation is mostly configured, so let’s dig into the next section and install some additional tools to make the development process more streamlined.

Configuring an Integrated Development Environment (IDE)

There are many excellent tools and IDE’s that help you, the developer, write, test, and deploy applications written in both Java and Python. There are tools that I’m very familiar with and work well for my development needs. My guess is you’re much the same and have your own tool preferences. It doesn’t really matter which toolset you use, provided they meet some basic requirements. For me, these include code highlighting and completion, code formatting and refactoring, debugging, compiling and packaging, unit and other testing, and source code control.

There are many fantastic IDE's on the market - both commercial and open source, and the choice is certainly yours. I developed the examples in this book using the Eclipse IDE [4] with PyDev [5] installed, as it meets the requirements I've specified and provides a bunch of other convenient features that I regularly use in my development projects.

If you're already familiar with writing, testing, and managing software applications using a different IDE, most of this section will be old hat. I do recommend you read through it, however, as this section sets the stage for the development of your GDA and CDA.

NOTE

I maintain detailed instructions for setting up Eclipse with PyDev for use with Programming the IoT on my website for students of my Connected Devices course. You can find more details on this books website at https://programmingtheiot.com/programming-the-iot-book.html.

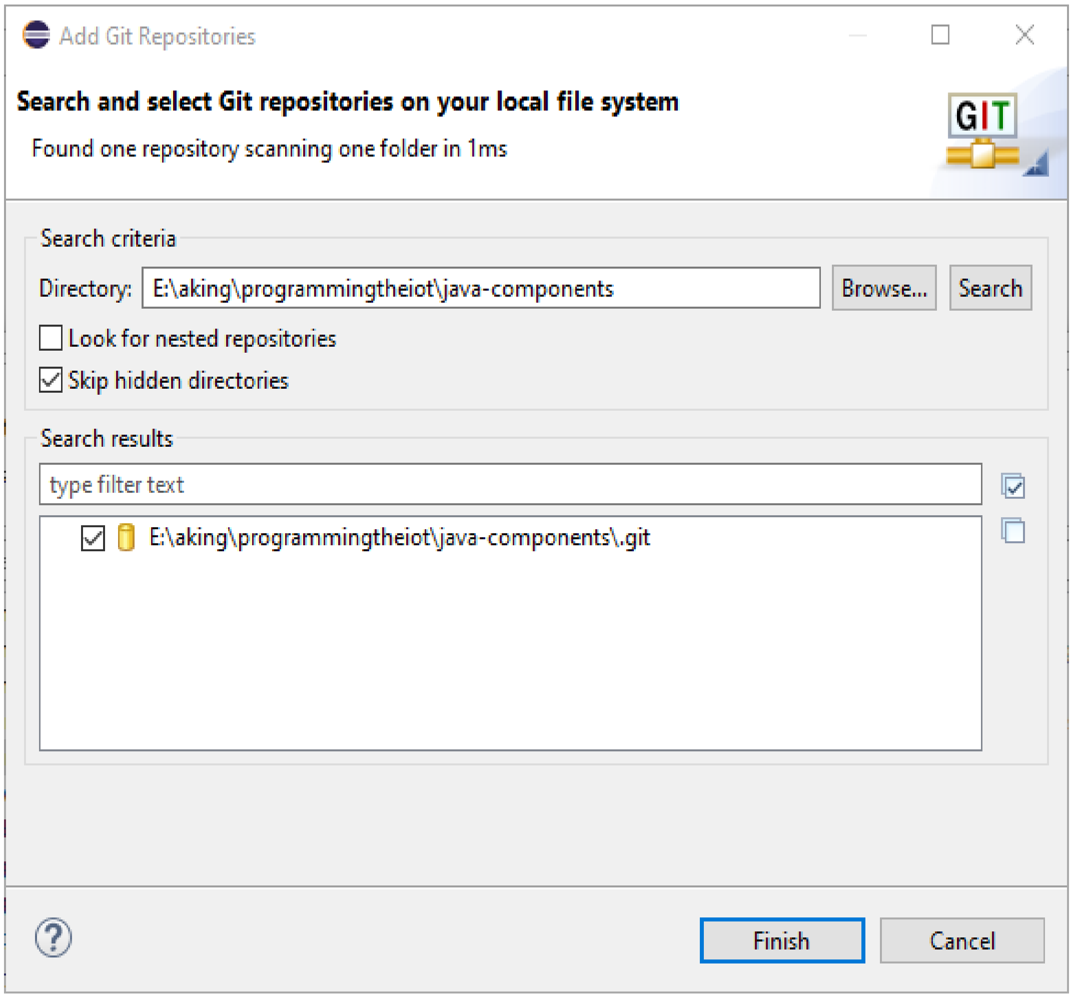
Setup Your GDA Project

The first step in this process is to install the latest Eclipse IDE for Java development. You can find the latest download links for Eclipse at https://www.eclipse.org/downloads/. You'll notice that there are many different flavors of the IDE available. For our purposes, you can simply choose 'Eclipse IDE for Java Developers'. Then, follow the instructions for installing the IDE onto your local system.

Once installed, launch Eclipse, select 'File -> Import', and find 'Git -> Projects from Git', and click 'Next'.

Select 'Existing local repository' and click 'Next'. If you already have some Git repositories in your home path, Eclipse will probably pick them up and present them as options to import in the next dialog (not shown). To pull in the newly cloned repository, simply click 'Add…', which will take you to the next dialog, shown in Figure 1-5. From here, you can add your new Git repository.

On my workstation, the repository I want to import is located at E:\aking\programmingtheiot\java-components. Yours will most likely have a different name, so be sure to enter it correctly!


Add Git Repositories

Search and select Git repositories on your local file system

Found one repository scanning one folder in 1ms

Search criteria

Directory: E:\aking\programmingtheiot\java-components

[ ] Look for nested repositories

[✓] Skip hidden directories

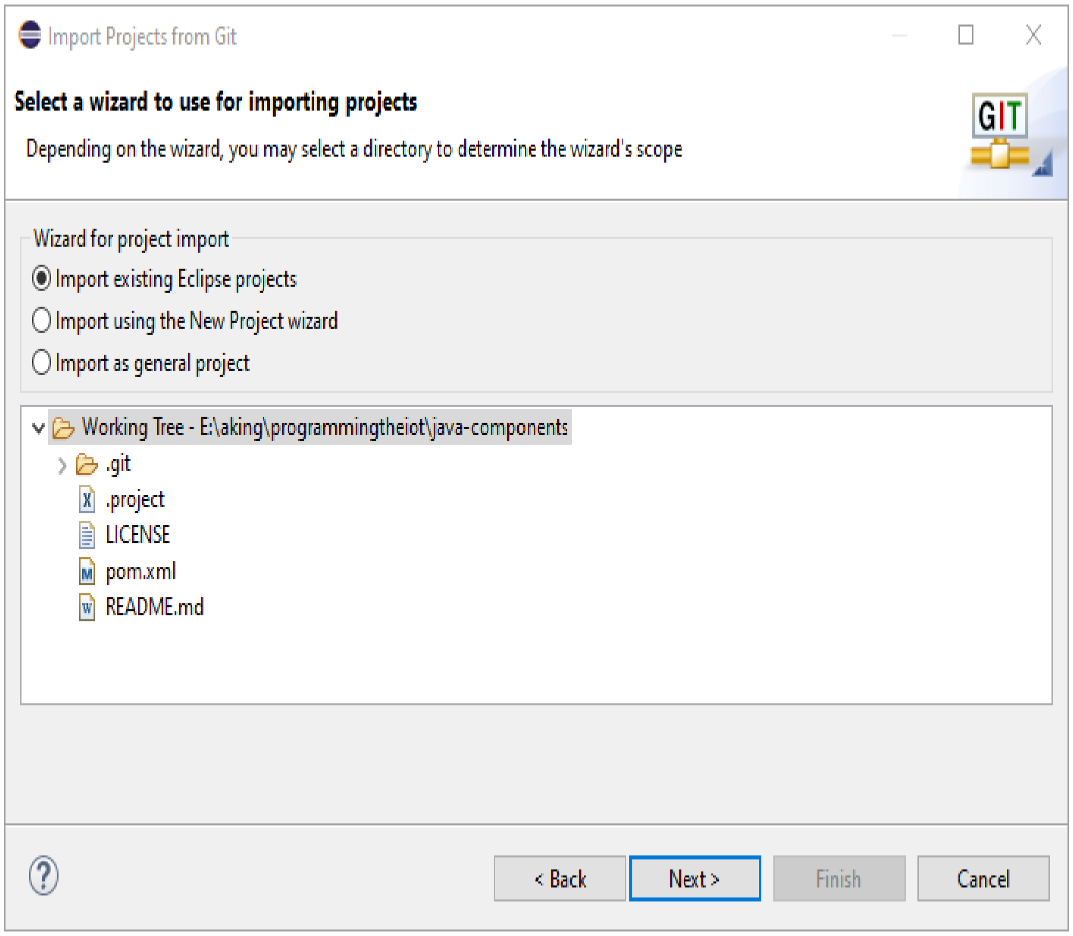
Search results

type filter text

[✓] E:\aking\programmingtheiot\java-components\.git

Figure 1-5. - Import java-components from your local Git repository

Click ‘Finish’, and you’ll see your new repository added to the list of repositories you can import. Highlight this new repository and click ‘Next…’. Eclipse will then present you with another dialog, asking you to import the project, using one of the options as shown in Figure 1-6.


Import Projects from Git

Select a wizard to use for importing projects

Depending on the wizard, you may select a directory to determine the wizard's scope

Wizard for project import

● Import existing Eclipse projects

○ Import using the New Project wizard

○ Import as general project

```
Working Tree - E:\aking\programmingtheiot\java-components
  .git
  .project
  LICENSE
  pom.xml
  README.md
```

Figure 1-6. - Import java-components as an existing Eclipse project

You now have a choice to make: you can import the java components as an existing Eclipse project, using the new project wizard, or simply as a general project. Unless you want to fully customize your project environment, I'd recommend using the first option - import an existing Eclipse project. This process will look for a .project file in the working directory (which I've included in each of the repositories you've already cloned), resulting in a new Java project named 'piot-java-components'.

Click 'Finish', and you'll see your new project added to the list of projects in the Eclipse Package Explorer, which - by default -

should be on the left side of your IDE screen.

Your GDA project is now set up in Eclipse, so let's explore the files inside. Navigate to this project in Eclipse and click on the '>' to expand it further, as shown in Figure 1-7.

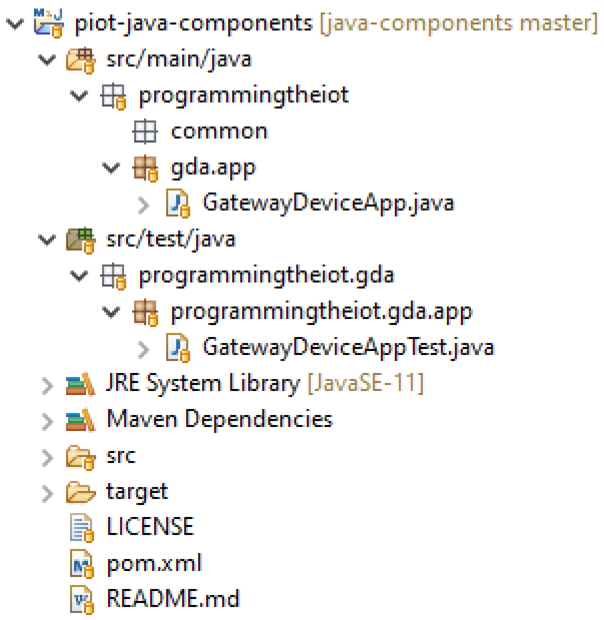


Figure 1-7. - GDA project now setup and ready to use

NOTE

What if you don't like the project name? Easy. You can simply right click the 'piot-java-components' name. select 'Rename…', type the new name, and click OK. Done. Just know that I'll continue to refer to the project by the original name throughout the book :).

You'll notice that there's already two files in the project: One is GatewayDeviceApp in the programmingtheiot.gda.app package, and the other is at the top level called pom.xml. The GatewayDeviceApp is a placeholder to get you started, although you may replace it with your own. I'd recommend you keep the naming convention the same, however, as the pom.xml depends on this to compile, test, and package the code. If you know your way around Maven already, feel free to make any changes you'd like.

NOTE

For those of you new to Maven, the pom.xml is simply Maven's primary configuration file and contains instructions for loading dependencies, their respective versions, naming conventions for your application, build instructions, and of course packaging instructions. Most of these dependencies are already included, although you may want to add your own if you find others to be useful. You'll also notice that Maven has its own default directory structure, which I've kept in place for the Java repository. To learn more about these and other Maven features, I'd recommend you walk through the 5-minute Maven tutorial located at https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html.

Now, to make sure everything is in place and you can build, package, and run the GDA, do the following:

1. Make sure your workstation is connected to the Internet.
2. Build your project and create an executable package.

a. Right click on the project ‘piot-java-components’ in Project Workspace, scroll down to ‘Run As’, and click ‘Maven install’.

    i. Since Maven will have to install any missing dependencies specified in the pom.xml, this may take a little bit to run, depending on your Internet connection speed and other factors.

b. Check the output in the Console at the bottom of the Eclipse IDE screen. There should be no errors, with the last few lines similar to the following:

```
[INFO] --- maven-install-plugin:2.4:install (default-install) @ gateway-device-app ---

[INFO] Installing E:\aking\workspace\gda\programmingtheiot-java\target\gateway-device-app-0.0.1.jar to C:\Users\aking\.m2\repository\programmingtheiot\gda\gateway-device-app\0.0.1\gateway-device-app-0.0.1.jar

[INFO] Installing E:\aking\workspace\gda\programmingtheiot-java\pom.xml to C:\Users\aking\.m2\repository\programmingtheiot\gda\gateway-device-app\0.0.1\gateway-device-app-0.0.1.pom

[INFO] ------------------------------------------------------------------------------

[INFO] BUILD SUCCESS
```

```
[INFO] ----------------------------------------
-------------------------------------

[INFO] Total time: 4.525 s

[INFO] Finished at: 2020-07-04T14:31:45-
04:00

[INFO] ----------------------------------------
-------------------------------------
```

3. Run your GDA application within Eclipse.
    a. Right click on the project ‘programmingtheiot-java’ again, scroll down to ‘Run As’, and this time click ‘Java application’.
    b. Check the output in the Console at the bottom of the Eclipse IDE screen. As with your Maven build, there should be no errors, with the output similar to the following:

    ```
    Jul 04, 2020 3:10:49 PM
    programmingtheiot.gda.app.GatewayDeviceApp
    initConfig INFO: Attempting to load
    configuration.

    Jul 04, 2020 3:10:49 PM
    programmingtheiot.gda.app.GatewayDeviceApp
    startApp INFO: Starting GDA...

    Jul 04, 2020 3:10:49 PM
    programmingtheiot.gda.app.GatewayDeviceApp
    startApp INFO: GDA ran successfully.
    ```

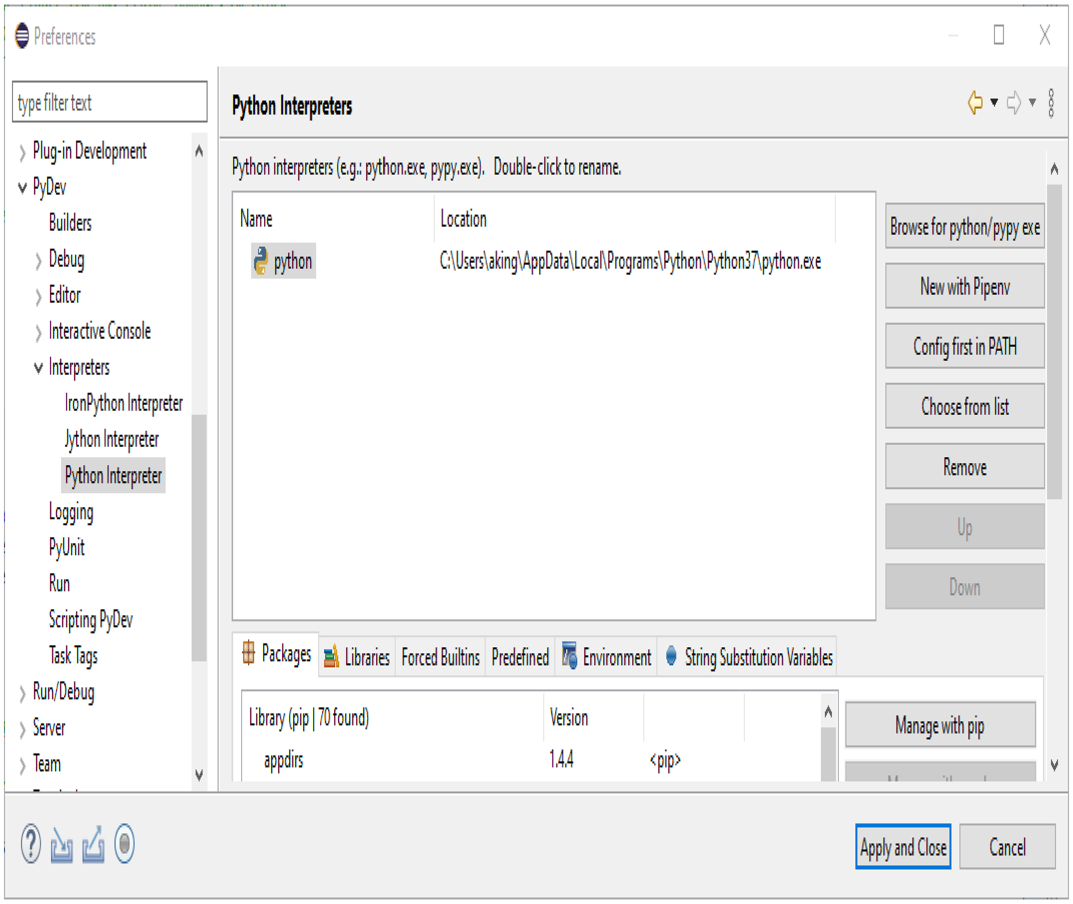
At this point, you’re ready to start writing your own code for the GDA. Now let’s get your development workstation setup for the CDA.

Setup Your CDA Project

This process will mimic the GDA setup process, but requires the addition of PyDev to Eclipse. Here’s a summary of activities to get you started.

If not already running, launch the Eclipse IDE. In a separate window or screen, open your web browser and navigate to https://marketplace.eclipse.org/content/pydev-python-ide-eclipse. Drag the PyDev “Install” icon from the web page and drop it near the top of the Eclipse IDE (you’ll see a green ‘plus’ icon show up, which is the indicator you can drop it into the IDE). Eclipse will then automatically install PyDev and its dependencies for you.

Once PyDev is installed, you can switch the Python interpreter to use the virtualenv environment you created in the previous section. Select ‘Preferences -> PyDev -> Interpreters -> Python Interpreter’. Eclipse will present a dialog similar to that shown in Figure 1-8.



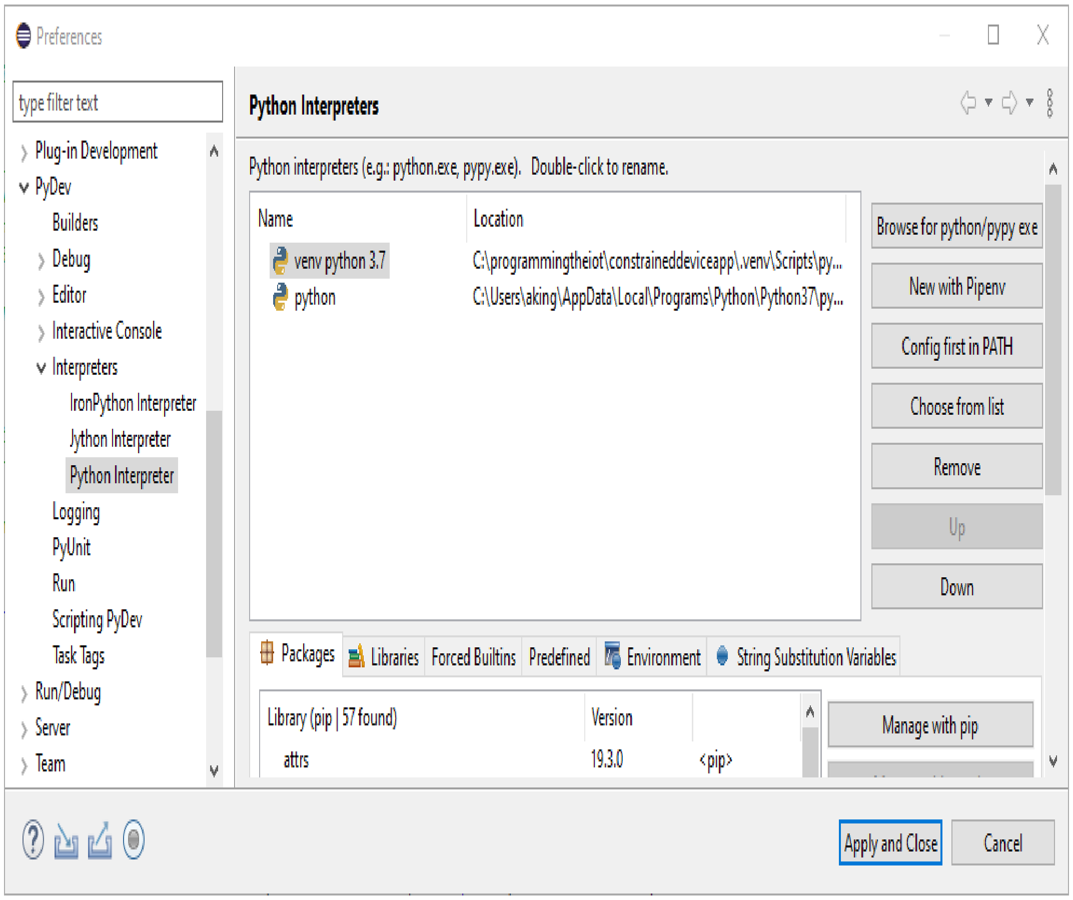
Python interpreters (e.g.: python.exe, pypy.exe). Double-click to rename.

Name	Location
python	C:\Users\aking\AppData\Local\Programs\Python\Python37\python.exe

Library (pip | 70 found)	Version	
appdirs	1.4.4	<pip>

Figure 1-8. - Add a new Python interpreter

Then, simply add a new interpreter using the ‘Browse for python/pypy.exe’ selection and provide the relevant information in the next popup window. Once complete, select the virtualenv interpreter and click ‘Up’ until it’s at the top of the list. At this point virtualenv will be your default Python Interpreter, as Figure 1-9 indicates. Click ‘Apply and Close’.


Preferences

type filter text

- Plug-in Development
- PyDev
  - Builders
  - Debug
  - Editor
  - Interactive Console
  - Interpreters
    - IronPython Interpreter
    - Jython Interpreter
    - Python Interpreter
  - Logging
  - PyUnit
  - Run
  - Scripting PyDev
  - Task Tags
- Run/Debug
- Server
- Team

Python Interpreters

Python interpreters (e.g.: python.exe, pypy.exe). Double-click to rename.

Name	Location
venv python 3.7	C:\programmingtheiot\constraineddeviceapp\.venv\Scripts\py...
python	C:\Users\aking\AppData\Local\Programs\Python\Python37\py...

Library (pip | 57 found)	Version	
attrs	19.3.0	<pip>

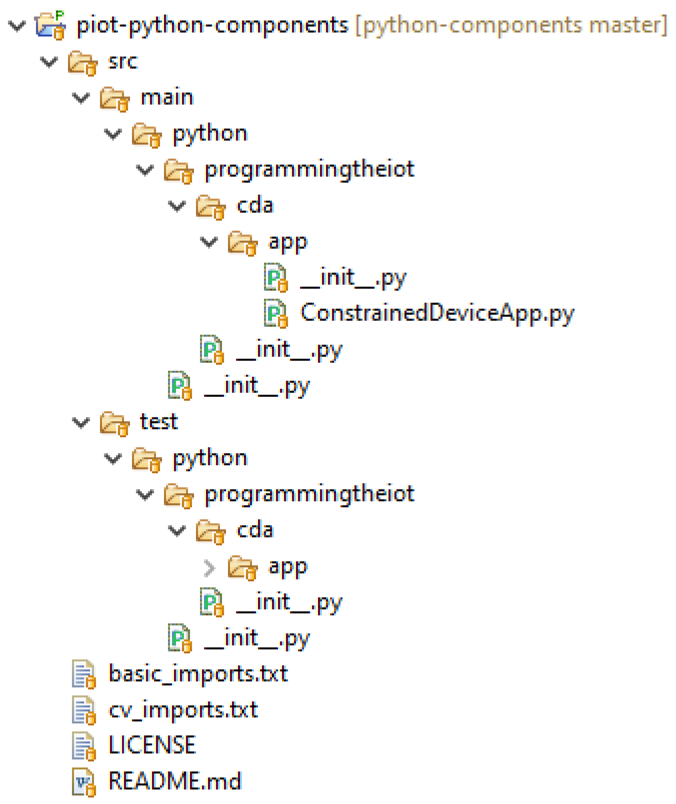
Figure 1-9. - Virtualenv Python interpreter now set as default

Once these steps are complete, select 'File -> Import', and import the python-components Git repository you've already cloned from GitHub. Again, this is nearly identical to the previous steps shown in Figures 1-5, 1-6, and 1-7, except you'll import the python-components Git repository you cloned from GitHub.

On my workstation, the repository I want to import is located at E:\aking\programmingtheiot\python-components. As with the GDA, your repository name will likely be different, so be sure to use the correct path. I've also included the Eclipse .project file within this repository, so you can simply import it as an Eclipse project. This one will default to Python, so will use PyDev as the project template. Again, you can import any way you'd like, but my recommendation is to use import it as you did with the GDA.

Once you complete the import process, you'll notice a new project in your Package Explorer named 'piot-python-components'. You now have the CDA components set up in your Eclipse IDE.

To view the files inside, navigate to 'piot-python-components' and click on the '>' to expand it further, as shown in Figure 1-10.



```
piot-python-components [python-components master]
  src
    main
      python
        programmingtheiot
          cda
            app
              __init__.py
              ConstrainedDeviceApp.py
            __init__.py
          __init__.py
    test
      python
        programmingtheiot
          cda
            app
            __init__.py
          __init__.py
  basic_imports.txt
  cv_imports.txt
  LICENSE
  README.md
```

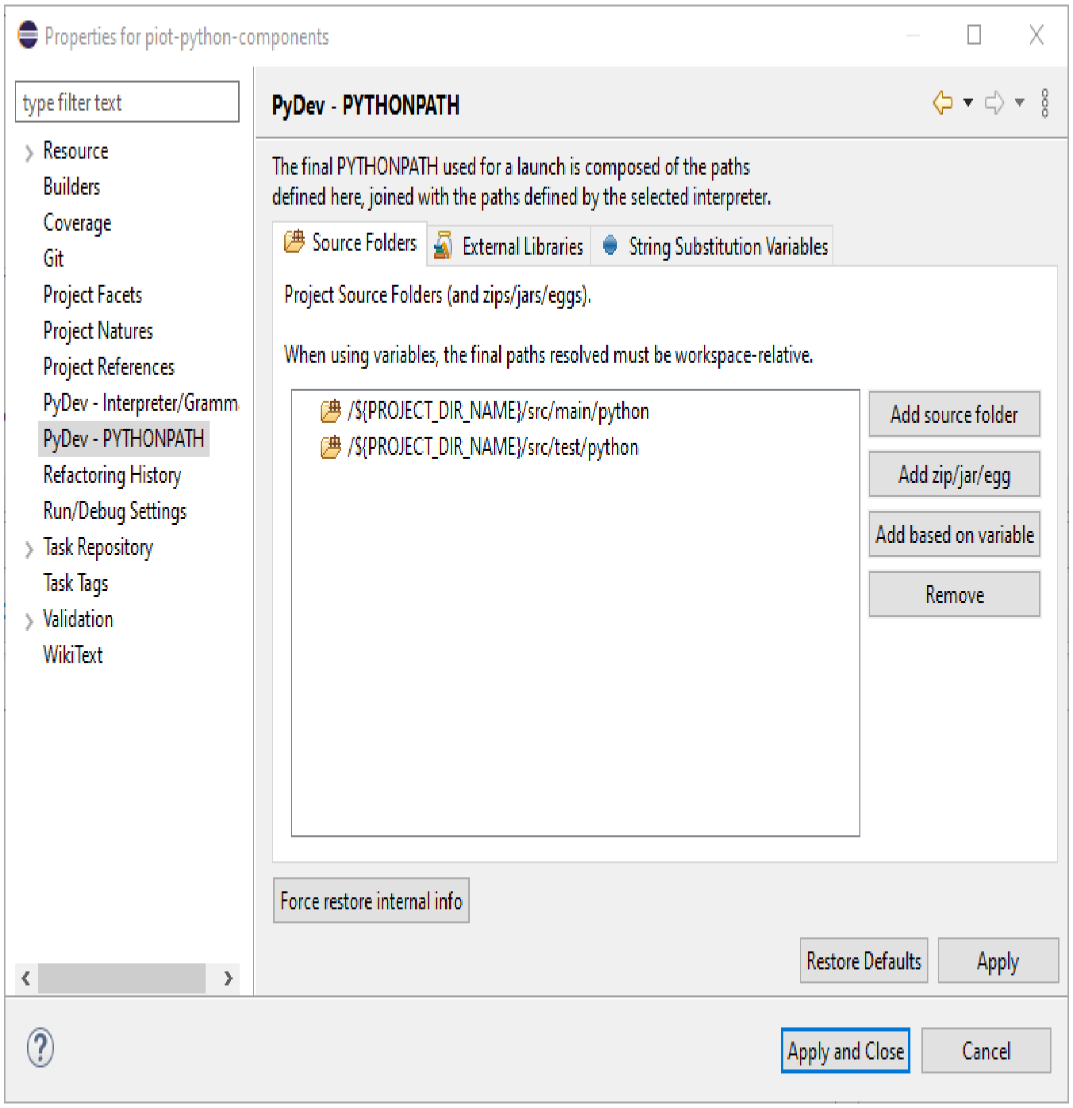
Figure 1-10. - CDA project now setup and ready to use

You'll notice that there's already six files in the project: One is ConstrainedDeviceApp in the programmingtheiot.cda.app package, and the other is at the top level called pom.xml. There are also four __init__.py files, which are empty files the Python interpreter uses to determine which directories to search for Python files (you can ignore these for now). Much like the GDA example previously given (and written in Java), the ConstrainedDeviceApp is simply a placeholder to get you started.

There's also a Maven pom.xml provided, which may seem odd for those familiar with Maven, as this is a Python project, not Java, and Maven is often associated with building, testing and packaging Java applications. So, why Maven? It's rather flexible and doesn't really care about the type of files it manages, so will be useful for testing and packaging Python modules. You won't need this capability until you start working on the exercises in the next chapter, so for now, you only need to verify your CDA runs by doing the following:

NOTE

If you’ve worked extensively with Python, you’re likely familiar with the PYTHONPATH environment variable. Since I’ve attempted to keep the GDA and CDA packaging scheme similar, you may need to tell PyDev (and your virtualenv environment) how to navigate this directory structure to run your application. Make sure ‘python’ is set for both ‘main’ and ‘test’ in PYTHONPATH by doing the following: Right click ‘piot-python-components’, select ‘PyDev - PYTHONPATH’, then click ‘Add source folder’, as shown in Figure 1-11. Select the ‘python’ folder under ‘main’, and click ‘Apply’. Do the same for the ‘python’ folder under ‘test’. Click ‘Apply and Close’ to finish.


Properties for piot-python-components

PyDev - PYTHONPATH

The final PYTHONPATH used for a launch is composed of the paths defined here, joined with the paths defined by the selected interpreter.

Project Source Folders (and zips/jars/eggs).

When using variables, the final paths resolved must be workspace-relative.

/${PROJECT_DIR_NAME}/src/main/python
/${PROJECT_DIR_NAME}/src/test/python

Figure 1-11. - Updating the PYTHONPATH environment variable within PyDev and Eclipse

1. Run your CDA application within Eclipse.

    a. Right click on the project ‘piot-python-components’ again, scroll down to ‘Run As’, and this time click Python Run’.

    b. Check the output in the Console at the bottom of the Eclipse IDE screen. As with your GDA test run, there should be no errors, with the output similar to the following:

```
2020-07-06 17:15:39,654:INFO:Attempting to
load configuration...

2020-07-06 17:15:39,655:INFO:Starting
CDA...

2020-07-06 17:15:39,655:INFO:CDA ran
successfully.
```

Congratulations! Both your GDA and CDA are set up and working within your IDE. At this point, you're technically ready to start writing your own code for both applications. But before we jump into the exercises in Chapter 2, we need to get our heads wrapped around the next two steps: testing and automation.

Step II: Define Your Testing Strategy

Now that your development environment is established for your GDA and CDA, we can discuss how you'll test the code you're about to develop. Obviously, good testing is a critically important part of any engineering effort, and programming is no exception to this. Every application you build should be thoroughly tested, whether it works completely independently of other applications or is tightly integrated with other systems. Further, every unit of code you write should be tested to ensure it behaves as expected. What exactly is a unit? For our purposes, a unit is always going to be represented as a function or method that you want to test.

NOTE

What's the difference between a function and a method? To grossly oversimplify, a function is a named grouping of code that performs a task (such as adding two numbers together) and returns a result. If the function accepts any input, it will be passed as one or more parameters. A method is almost identical to a function, but is attached to an object. In object oriented parlance, an object is simply a class that's been instantiated, and a class is the formal definition of a component - its methods, parameters, construction, and deconstruction logic. All of the Java examples in this book will be represented in class form with methods defined as part of each class. Python can be written in script form with functions or as classes with methods, but I prefer to write Python classes with methods and will do so for each Python example shown in this book, with only a few exceptions to this.

Unit, Integration, and Performance Testing

There are many ways to test software applications and systems, and some excellent books, articles and blogs on the subject. Developing a working IoT solution requires careful attention to testing - within an application and between different applications and systems. For the purposes of the solution you'll develop, I'll focus on just three: Unit tests, integration tests, and performance tests.

Unit tests are code modules written to test the smallest possible *unit of code* that's accessible to the test, such as a function or method. These tests are written to verify a set of inputs to a given function or method returns an expected result. Boundary conditions are often tested as well, to ensure the function or method can handle these types of conditions appropriately.

NOTE

A unit of code can technically be a single line, multiple lines of code, or even an entire code library. For our purposes, a unit refers to one or more lines of code, or an entire code library, that can be accessed through a single interface that is available on the local system - that is, a function or a method which encapsulates the unit’s functionality and can be called from your test application. This functionality can be, for example, a sorting algorithm, a calculation, or even an entry point to one or more additional functions or methods.

I use JUnit for unit testing Java code (included with Eclipse), and PyUnit for unit testing Python code (part of the standard Python interpreter, and available within PyDev). You don’t have to install any additional components to write and execute unit tests within your IDE if you’re using Eclipse and PyDev.

NOTE

In your GDA project, you've likely noticed that Maven creates two directory structures for your source code: one for Java source code located in ./src/main/java, and another Java unit test code located in ./src/test/java. Since this is the default convention for Maven projects, to leave it as is.

In your CDA project, the code path structure is a bit different than the GDA project: the ./src/main/java and ./src/test/java are missing. That probably seems quite sensible, since this is a Python project and not Java! All the source code is contained within ./src/programmingtheiot, while all the tests are contained within ./tests/programmingtheiot.

Although I provide a bunch of unit tests for you to use, we'll dig into creating some of your own in Chapter 2. If you're new to this area of development, I've provided a couple simple code snippets below in both Java and Python to help you become more familiar with the code structure. Here's a simple unit test in Java using JUnit [6] that verifies the method `addTwoIntegers()` behaves as expected:

```
@Test

public int testAddTwoIntegers(int a, int b)

{

MyClass mc = new MyClass();
// baseline test

assertTrue(mc.addTwoIntegers(0, 0) == 0);

assertTrue(mc.addTwoIntegers(1, 2) == 3);

assertTrue(mc.addTwoIntegers(-1, 1) == 0);

assertTrue(mc.addTwoIntegers(-1, -2) == -3);

assertFalse(mc.addTwoIntegers(1, 2) == 4);

assertFalse(mc.addTwoIntegers(-1, -2) == -4);

}
```

What if you have a single test class with two individual unit tests, but you only want to run one? Simply add `@Ignore` before the `@Test` annotation, and JUnit will skip that particular test. Remove the annotation to re-enable the test.

Let’s look at the same example in Python, using Python 3’s built-in PyUnit framework and the `unittest` library[7]:

```
def testAddTwoIntegers(self, a, b):

MyClass mc = MyClass()

# baseline test
```

```
self.assertTrue(mc.addTwoIntegers(0, 0) == 0)

self.assertTrue(mc.addTwoIntegers(1, 2) == 3)

self.assertTrue(mc.addTwoIntegers(-1, 1) == 0)

self.assertTrue(mc.addTwoIntegers(-1, -2) == -3)

self.assertFalse(mc.addTwoIntegers(1, 2) == 4)

self.assertFalse(mc.addTwoIntegers(-1, -2) == -4)
```

PyUnit, much like JUnit, allows you to disable specific tests if you wish. Simply add `@unittest.skip("Put your reason here.`”), or just `@unittest.skip` as the annotation before the method declaration, and the framework will skip over that specific test.

Integration tests are super important for the IoT, as they can be used to verify that the connections and interactions between systems and applications work as expected. Let’s say you want to test a simple sorting algorithm using a basic data set embedded within the testing class - you’ll typically write one or more unit tests, execute each one, and verify all is well. Simple, right?

What if, however, the sorting algorithm needs to pull data from a data repository accessible via your local network or even the Internet? So what, you might ask? Well, now you have another dependency just to run your sort test. You’ll need an integration test to verify that data repository connection is both available and working properly before exercising the sorting unit test.

These kinds of dependencies can make integration testing pretty challenging with any environment, and even more so with the IoT, since we sometimes have to set up servers to run specialized protocols to test our stuff. Fortunately, Maven can help with this, too, since it supports plugins which make some of this much easier. For all the code that needs to integrate only within the Edge Tier, we’ll use Maven plus some specialized Unit Tests as part of our integration testing strategy.

Finally, performance tests are useful for testing how quickly, or efficiently, a system handles a variety of conditions. They can be used with both unit and integration tests when, for instance, response time or the number of supported *concurrent*, or simultaneous connections needs to be measured.

Let’s say there are many different systems that need to retrieve a list of data from your data repository, and each one wants that list of data sorted before your application returns it to them. Ignoring system design and database schema optimization for a moment, a series of performance tests can be used to time the responsiveness of each system’s request (from the initial request to the response), as well as the number of concurrent systems that can access your application before it no longer responds adequately.

Another aspect of performance testing is to test the load of the system your application is running on, which can be quite useful for IoT applications. IoT devices are generally constrained in some way - memory, CPU, storage, etc - whereas cloud services can scale as much as we need them to. It stands to reason then, that our first IoT applications - coming up in Chapter 2 - will set the stage for monitoring each device's performance individually.

Since performance testing often goes hand-in-hand with both integration and unit testing, we'll continue to use Maven and specialized Unit Tests for this as well, along with open source tools where needed.

NOTE

There are many performance testing tools available, and you can also write your own. While system-to-system and communications protocol performance testing is completely optional for the purposes of this book, I'll cover it in more detail beginning in Part III, Chapter 5, where I'll introduce Locust [8], a tool you can use in your local environment to test scalability for your solution. Locust allows for extensions that enable you to test many different protocols and connection paradigms, which will be important for device-to-device communications testing between your GDA and CDA.

Testing Tips for the Exercises in this Book

The sample code provided for each exercise in this book includes unit tests, which you can use to test the code you'll write. These unit tests are provided as part of the connected-devices-java and connected-devices-python repositories you've already pulled into your new iot-gateway and iot-device (respectively) projects are key to ensuring your implementation works properly.

Some exercises also have integration tests that you can use as-is, or modify to suit your specific needs. I've also included some sample performance tests you can use to test how well some of your code performs when under load.

Your implementation of each exercise should pass each provided unit test with 100% success. You're welcome to add more unit tests if you feel they'll be helpful to verify the functionality you develop. The integration tests and performance tests provided will be helpful, but not required as part of each exercise.

Remember, tests are your friend. And, like a friend, they shouldn't be ignored. They can surely be time consuming to write and maintain, but any good friendship takes investment. These tests - whether unit, integration, or performance - will help you validate your design and verify your functionality is working properly.

Step III: Managing Your Workflow - Requirements, Source Code, and CI/CD

So you’ve figured out how you want to write your code and test it - great! But wouldn’t it be great if you could manage all your requirements, source code, and CI/CD pipelines. Let’s tackle this in our last step, which is all about managing your overall development process workflow. This includes requirements tracking, source code management, and CI/CD automation.

You’re probably sick of me saying that building IoT systems is hard, and that’s largely because of the nature of the Edge Tier (since we often have to deal with different types of devices, communication paradigms, operating environments, security constraints, and so on). Fortunately, there are many modern CI/CD tools that can be used to help navigate these troubled waters. Let’s look at some selection requirements for these tools, and then explore how to build out a CI/CD pipeline that will work for our needs.

Your IoT CI/CD pipeline should support secure authentication and authorization, scriptability from a Linux-like command line, integration with Git and containerization infrastructure, and the ability to run pipelines within my local environment as well as a cloud-hosted environment.

There are many online services that provide these features, some of which provide both free and paid service tiers. When you downloaded the source code for this book, you pulled it from my GitHub [9] repositories using Git’s clone feature. GitHub is an online service that supports overall developer workflow management, including source code control (using Git), CI/CD automation, and planning.

Each exercise will build, test, and deploy locally, but also assume your code is committed to an online repository using Git for source code management. You’re welcome to use the online service of your choice, of course. For this book, all examples and exercises will assume GitHub is being used. You can learn more about GitHub’s features and create a free account on their website (https://github.com/).

> **NOTE**
>
> There are lots of great resources, tools and online services available that let you manage your development work and set up automated CI/CD pipelines. Read through this section, try things out, and then as you gain more experience, choose the tools and service that works best for you.

Managing Requirements

Ah yes - requirements. What are we building, who cares, and how are we going to build it? Plans are good, are they not? And since they’re good, we should have a tool that embraces goodness, which includes: task prioritization, task tracking, team collaboration, and (maybe) integration with other tools.

Remember the code you cloned from my GitHub repository? Choose one - for example, constraineddeviceapp, and open its

URL in a web browser. You'll notice that it has some Issues (like, literally - there's a column called 'Issues' on the main page). If you click on 'Issues', you'll see all the tasks (requirements) that need to be implemented.

Each task contains the instructions and other details a developer (you) will need to write the code and make sure it works correctly.

So far, so good, right? These tasks - at least in GitHub - can be organized into a board, so that you can see all of the things that need to be implemented for a project. You've probably heard of Agile [10] project management processes such as Scrum and Kanban.

One of the workflow organizational approaches GitHub let's you select for a project is a Kanban-based electronic board. Kanban organizes tasks as cards, which can then be placed into various columns within the board's workflow depending on their status. This is the approach we'll use in the book to track things that need to be done for each code repository.

NOTE

I'm managing all of the activities for this book within a Kanban board, too. Each card on the board represents a task I (or one of my team members) needs to complete. Cards only move to 'Done' after Sarah approves (thanks Sarah!)

This approach is really powerful, and I think you'll see how it will help you organize the work you need to do and track your progress as you go through this book.

Setting up a Cloud Project and Repositories

This book is actually organized as a large IoT project using a single Kanban board across the repositories. Each repository will have a set of tasks that are aligned to chapter-based milestones, which map back into the Kanban board.

The beauty of this approach is that, once your project and repositories are properly set up, all you really need to do is update your task status as you work through each chapter writing, testing and committing your code. The workflow engine will automatically parse your task updates, and represent the task's card on the board for you. Neat, huh?

If you're interested in using GitHub to set all of this up, here's a simple 3-step process you can use to get started:

1. Create a GitHub account
    a. If desired, create an organization associated with the GitHub account
2. Within the your account (or organization), create the following:
    a. A new private project named 'Programming the IoT - Exercises'

   b. A new private Git repository named ‘piot-java-components’

   c. A new private Git repository named ‘piot-python-components’

3. Update the remote repository for both ‘piot-java-components’ and ‘piot-python-components’

   a. From the command line, execute the following commands:

      ```
      git remote set-url origin {your new URL}
      
      git commit -m “Initial commit.”
      
      git push
      ```

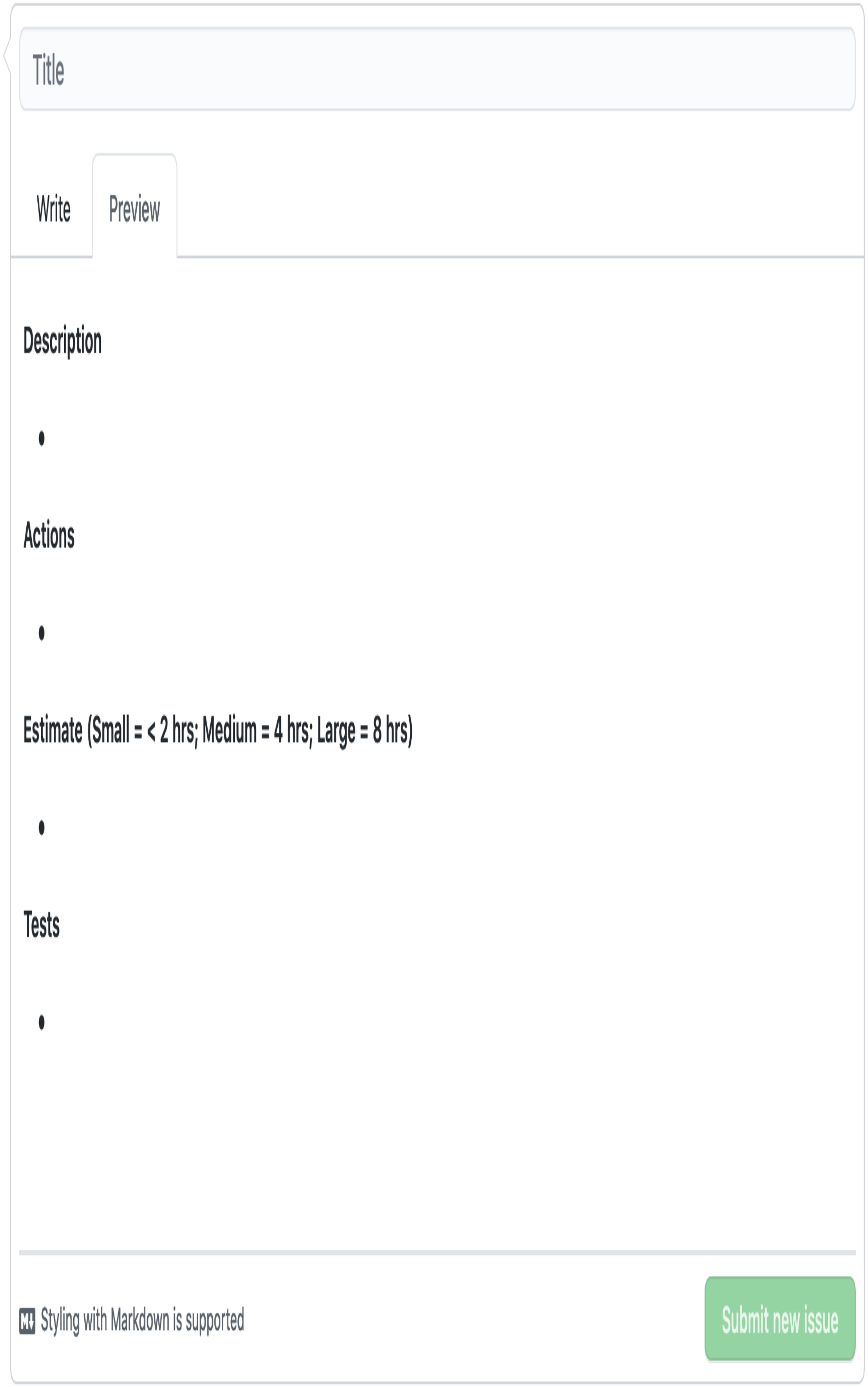
   b. IMPORTANT: Be sure to do this for both ‘piot-java-components’ and ‘piot-python-components’, using the appropriate Git repository URL for each!

NOTE

I’ve written a guide to help you configure all of this, located at the book’s homepage (https://programmingtheiot.com/programming-the-iot-book.html).

Once you complete the tasks above, your cloud-based workflow environment is now set up and ready to use. Take a look at Figure 1-12. It shows the task template I’ll use throughout the book (GitHub calls this an ‘Issue’). This is the stuff that goes into each task.

It’s rather simple, and you only have to enter five items: Name, Description, Actions, Estimate, and Validation. You can update this, but I’d recommend sticking with one template for all your cards. If you create a card that doesn’t have any validation step (for some reason), just enter ‘N/A’.



Description

-

Actions

-

Estimate (Small = < 2 hrs; Medium = 4 hrs; Large = 8 hrs)

-

Tests

-

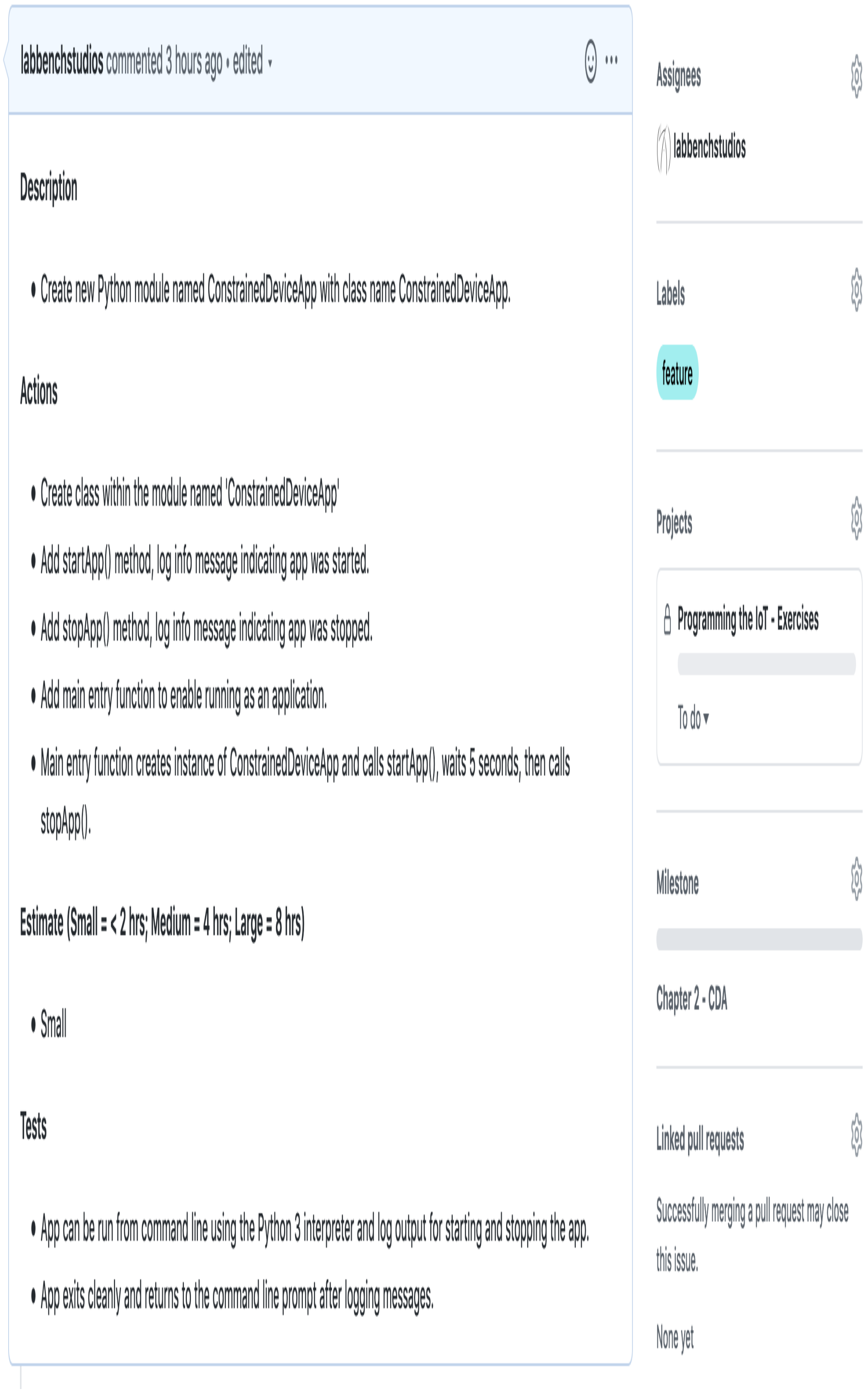
Figure 1-12. - Task template

Most of these categories are self-explanatory. But, why only three levels-of-effort for Estimate? In this book, most of the activities should take ~2 hours or less (Small), about half a day (Medium), or about one day (Large). If you come across an activity that you're sure will take more than a day, it should be broken down into smaller tasks that fit one of these three levels-of-effort.

For example, a 'task' with the name *Integrate IoT solution with three cloud services* certainly represents work that needs to be done, but judging by the name only, it's clearly way too big and complicated to be a single work activity.

My suggestion is to write tasks that align to a specific code module, or even a single method, as much as possible. I'm sure you've heard this before: Keep it simple.

Figure 1-13 shows an example of the template filled in with the first coding task you'll have - creating the Constrained Device Application (CDA).


labbenchstudios commented 3 hours ago • edited

Description

- Create new Python module named ConstrainedDeviceApp with class name ConstrainedDeviceApp.

Actions

- Create class within the module named 'ConstrainedDeviceApp'
- Add startApp() method, log info message indicating app was started.
- Add stopApp() method, log info message indicating app was stopped.
- Add main entry function to enable running as an application.
- Main entry function creates instance of ConstrainedDeviceApp and calls startApp(), waits 5 seconds, then calls stopApp().

Estimate (Small = < 2 hrs; Medium = 4 hrs; Large = 8 hrs)

- Small

Tests

- App can be run from command line using the Python 3 interpreter and log output for starting and stopping the app.
- App exits cleanly and returns to the command line prompt after logging messages.

Assignees

labbenchstudios

Labels

feature

Projects

Programming the IoT - Exercises

To do

Milestone

Chapter 2 - CDA

Linked pull requests

Successfully merging a pull request may close this issue.

None yet

Figure 1-13. - Example of a typical development task

And Figure 1-14 shows the result of adding the task as a Kanban card. This card was generated automatically after aligning the task to a project. Notice it’s been added into the “To do” column on the board, since it’s new and there’s no status as of yet. Once you start working on the task and change its status, it will move to ‘In progress’.

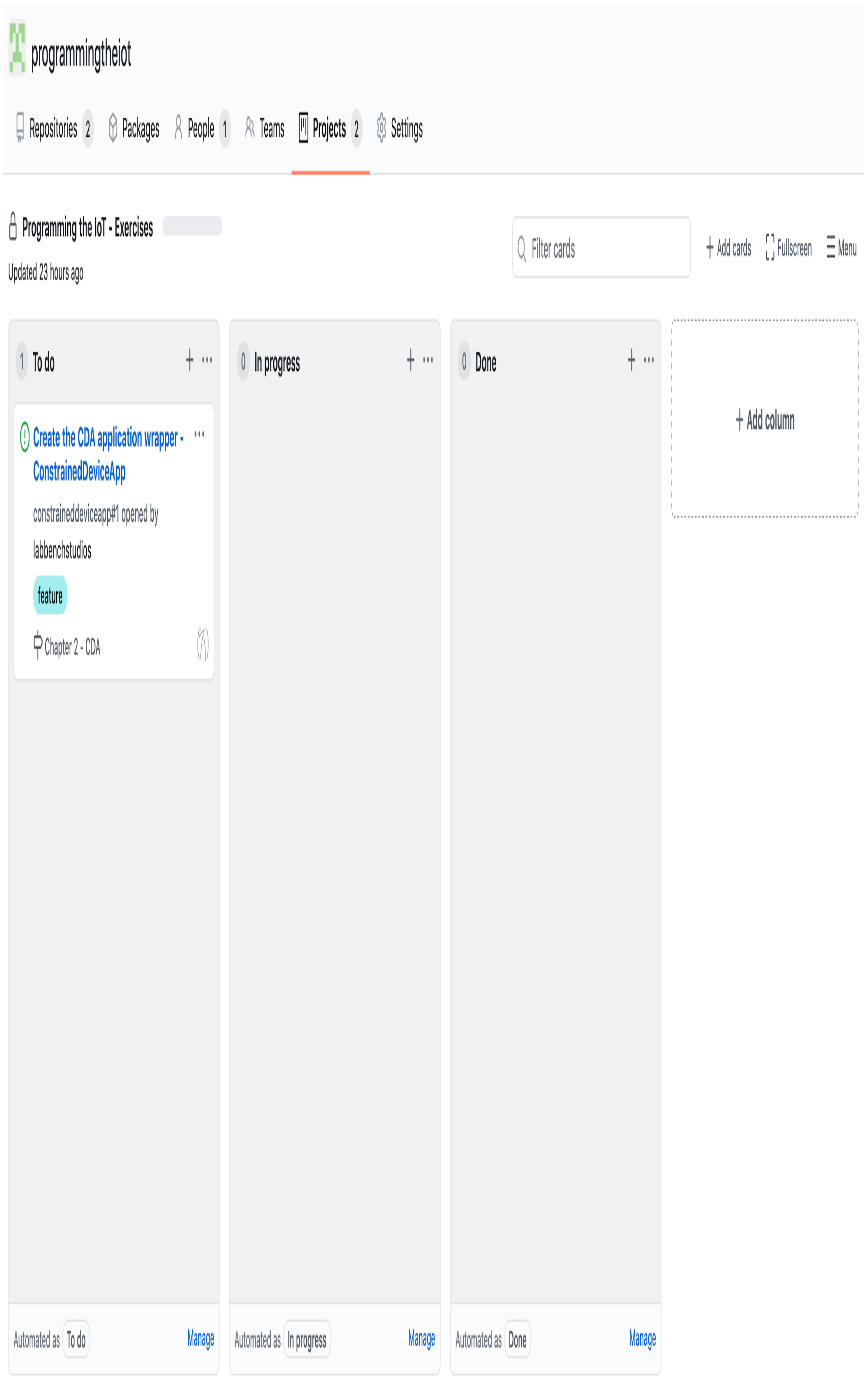

programmingtheiot
Repositories 2
Packages
People 1
Teams
Projects 2
Settings
Programming the IoT - Exercises
Updated 23 hours ago
Filter cards
Add cards
Fullscreen
Menu
1 To do
0 In progress
0 Done
Add column
Create the CDA application wrapper - ConstrainedDeviceApp
constraineddeviceapp#1 opened by labbenchstudios
feature
Chapter 2 - CDA
Automated as To do
Manage
Automated as In progress
Manage
Automated as Done
Manage

Figure 1-14. - The new example task automatically added into the Kanban board

Source Code Control Using Git Remotes and Branching

One of the key benefits of using Git is the ability to collaborate with others and synchronize your local code repository with a remote repository stored in the cloud. If you’ve worked with Git before, you’re already familiar with remotes and branching. I’m not going to go into significant detail here, but they’re important concepts to grasp as part of your automation environment.

Branching is simply a way of enabling each developer, or team, to segment their work without negatively impacting the main code base. In Git, this default main branch is currently called ‘master’, and is typically used to contain the code that has been completed, tested, verified, and - usually - placed into production. This is the default branch for both ‘programmingtheiot-java’ and ‘programmingtheiot-python’, and while you can leave it as is and simply work off this default branch, it’s not generally recommended for the reasons I mentioned.

Branching strategies can differ from company to company and team to team, although the one I like to use has each chapter in a new branch, then once all is working correctly and properly tested, the chapter branch gets merged into the master. From there, a new branch is created from the merged master for the next chapter, and so on.

This approach allows you to easily track changes between chapters, and even go back to the historical record of an earlier chapter if you want to see what changed between, say, Chapter 2 and Chapter 5. In Eclipse, you can right-click on the project (either ‘programmingtheiot-java’ or ‘programmingtheiot-python’), and choose ‘Team > Switch To > New Branch…’ to establish a new branch for your code.

I’d suggest you use the naming convention of ‘chapter*nn*’ as each branch name, where ‘*nn*’ is the two-digit chapter number. For instance, the branch for Chapter 1 will be named ‘chapter01’, Chapter 2 will be named ‘chapter02’, and so on.

NOTE

All the gory details on Git branching and merging are out of scope for this book, so I’d recommend reading the following guide if you’d like to dig in https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging.

Automated CI/CD in the Cloud

Within Eclipse, you can write your CDA and GDA code, execute unit tests, and build and package both applications. This isn’t actually automated, since you have to start the process yourself by executing a command like mvn install from the command line, or invoking the Maven install process from within the IDE. This is great for getting both applications to a point where you can run them, but doesn’t actually *run* them -

you still need to manually start the applications up and then run your integration and / or performance tests.

As a developer, part of your job is writing and testing code to meet the requirements that have been captured (in cards on a Kanban board, for example), so there’s always some manual work involved. Once you know your code units function correctly, it would be pretty slick to have everything else run automatically - say, after committing and pushing your code to the remote dev branch (such as ‘chapter02’, for example).

GitHub supports this automation through GitHub actions [11]. I’ll talk more about this in Chapter 2 and help you set up your own automation for the applications you’re going to build.

Automated CI/CD in your Local Development Environment

There are lots of ways to manage CI/CD within your local environment. GitHub actions can be run locally using self-hosted runners [12], for example. There’s also a workflow automation tool called Jenkins [13] that can be run locally, integrates nicely with Git local and remote repositories, and has a plugin architecture that allows you expand its capabilities seemingly ad infinitum.

WARNING

There are lots of great 3rd party Jenkins plugins, but some are not well maintained and may introduce security vulnerabilities. If it’s a plugin you’re certain you need, be sure to track how often it’s updated and patched. If it’s not regularly maintained, it could introduce compatibility and security issues to your environment.

Once installed and secured, you can configure Jenkins to automatically monitor your Git repository locally or remotely, and run a build / test / deploy / run workflow on your local system, checking the success at each step. If, for example, the build fails because of a compile error in your code, Jenkins will report on this and stop the process. The same is true if the build succeeds, but the tests fail - the process stops at the first failure point. This ensures your local deployment won’t get overwritten with an update that doesn’t compile or fails to successfully execute the configured tests.

Setting up any local automation tool can be a complicated and time consuming endeavor. It’s super helpful, however, as it basically automates all the stuff you’re going to do to build, test and deploy your software. That said, it’s not required for any of the exercises in this book.

I’ve detailed my own approach using GitHub actions and self-hosted runners as well as Jenkins in a document on my website at https://programmingtheiot.com/setup/ConnectedDevices_DevE nvironmentSetup.html. See the Table of Contents to navigate to the section most relevant for your environment.

A Few Thoughts on Containerization

You’ve likely heard of *containerization*, which is simply a way to package your application and all of its dependencies into a single image, or *container*, that can be deployed to many different operating environments. This approach is very convenient, since it allows you to build your software and deploy it in such a way as to make the hosting environment no longer a concern, provided the target environment supports the container infrastructure you’re using.

Docker [14] is essentially an application engine that runs on a variety of operating systems, such as Windows, Mac, and Linux, and serves as a host for your container instance(s). Your GDA and CDA, for example, can each be containerized and then deployed to any hardware device that supports the underlying container infrastructure and runtime.

It’s worth pointing out that containerizing any application that has hardware-specific code may be problematic as it will not be portable to another, different hardware platform (even if the container engine is supported). If you want your hardware-specific application to run on any platform that supports Docker, that platform would require a hardware-specific emulator compatible with the code developed for the application.

For example, if your CDA has code that depends on Raspberry Pi-specific hardware, This is less of a concern for us at the moment, since you'll be simply emulating sensors and actuators and won't have any hardware-specific code to worry about until Chapter 4 (which, again, is optional). I'll discuss this more in Chapter 4, along with strategies to overcome hardware-specificity in your CDA.

When using CI/CD pipelines in a remote, or cloud, environment, you'll notice that these services will likely deploy to virtual machines and run your code within a container that includes the required dependencies, all configured as part of the pipeline. For many cases, this makes perfect sense and can be an effective strategy to ensure consistency and ease of deployment.

To keep things a bit more simple, I won't walk through containerization in this book for use within your development environment and as part of your workstation, even though there are many benefits to doing so. The primary reason is that it adds another layer of complexity to manage initially and I want to get you up and running with your own applications as soon as possible.

If you're interested in learning more about containerizing your IoT workstation and local environment, I've detailed my approach for the GDA, CDA, and other related applications within https://programmingtheiot.com/setup/ConnectedDevices_DevE nvironmentSetup.html. See the section labeled ‘Using Containers’ for more information.

Conclusion

Congratulations! You’ve just completed the longest - and perhaps most tedious - chapter in the book. You learned about some basic IoT principles, created a problem statement to drive your IoT solution, and established a baseline IoT systems architecture that includes the cloud tier and edge tier.

Perhaps most importantly, I introduced two applications - the GDA and CDA - which will serve as the foundation for much of your IoT software development throughout this book, and that you’ll start building in Chapter 2. Finally, you set up your development environment and workflow, learned about requirements management, explored unit, integration and performance testing, and considered some basic CI/CD concepts to help automate your builds and deployment.

You’re now ready to start building your first two IoT applications using Python and Java. If you’re ready to move on, I’d suggest you grab a good cup of coffee or tea. Let’s dig in.

---

1 Details on the various classes of constrained devices can be found within the IETF’s RFC 7228 (https://tools.ietf.org/html/rfc7228).

2 More information on GitHub can be found on their website (https://github.com/).

3 Virtualenv provides a way to contain your Python 3 environment and all the necessary library dependencies within a single, lightweight, virtual environment.

4 The Eclipse IDE version used for the exercises and examples listed in this book is 2020-06 and can be downloaded from the Eclipse website (https://www.eclipse.org/downloads/).

5 More information on PyDev can be found on the project's website (http://www.pydev.org/).

6 Detailed information on JUnit and the latest version, JUnit 5, can be found on the JUnit website (https://junit.org/junit5/).

7 Detailed information on Python 3's unittest library can be found in the unittest documentation (https://docs.python.org/3/library/unittest.html).

8 For more information on Locust, see the documentation from their website (https://docs.locust.io/en/stable/what-is-locust.html).

9 You can read more about GitHub and its Git hosting features on their website (https://github.com/).

10 Read more about the Agile Manifesto at https://agilemanifesto.org/.

11 GitHub actions is a feature available within GitHub that allows customized workflows to be created for those who have an account within GitHub. You can read more about GitHub actions on their website (https://docs.github.com/en/actions).

12 Self-hosted runners, part of GitHub actions, allow you to run your action workflows locally. There are caveats, of course, and security considerations. You can read more about self-hosted runners on the GitHub actions documentation website (https://docs.github.com/en/actions/hosting-your-own-runners/about-self-hosted-runners).

13 You can read more about Jenkins and its automation features at their website (https://www.jenkins.io/).

14 You can read more about containerization concepts and Docker containerization products at their website (https://www.docker.com/resources/what-container).

Chapter 2. Building Two Simple IoT Monitoring Applications

I must have data!

How much memory remains?

I dare not look. Sigh.

Fundamental concepts: Build two IoT performance monitoring applications – one as an IoT gateway, and the other as an IoT constrained device.

This chapter focuses on the initial steps to getting your IoT solution up and running. You'll build upon these in the upcoming chapters, so setting the basic design in place now is very important. The overall architecture presented in Chapter 1 provides the initial guidance you'll need to start coding your IoT solution, and we’ll keep building upon it as we blaze ahead in this chapter and beyond. I’m sure you’re anxious to start building out your own solution based on what you’ll learn, but it’s important to take things one step at a time.

We’ll start with two simple applications that collect some simple telemetry about the devices (real or virtual): a Gateway Device App (GDA) that will run on your “gateway device”, and a Constrained Device App (CDA) that will run on your “constrained device”. The design of each application provides the foundation for all of the code you’ll develop for the edge tier, so you may want to keep the design from Chapter 1 handy for reference.

What you’ll learn in this chapter

This is the beginning of your coding journey with the IoT. You’ll learn how to define a detailed design for both your GDA and CDA, separate the logical components of your design, and implement the framework for these two applications in Java (GDA) and Python (CDA).

These will be very simple applications, but they’ll serve as important foundation layers for your overall solution. Each will implement an application framework and exercise that framework by incorporating external libraries used to track the performance of a computing system. You’ll use the data they generate for your initial foray into creating telemetry using your CDA and GDA.

Designing Your IoT Applications

Remember the problem statement from Chapter 1? Let’s briefly review it now:

> *I want to understand the environment in my home and how it changes over time, and make adjustments to enhance comfort while saving money.*

If you think about all of the things your applications will need to do to address this problem, and also consider the importance of testing system behavior and performance, there are some important capabilities you’ll want to ‘bake in’ from the start. One is the ability to easily add new features as we go, and the other is to track each application’s performance so we know if it’s working adequately.

The first step in designing both the GDA and CDA is to create an application wrapper that can determine what features need to be loaded and then launch those features consistently. I’ll introduce application configuration using a configuration file in Chapter 3, so no need to worry about that right now.

Your initial GDA and CDA designs will look very similar to one another. They’ll collect some basic telemetry, which for now will include just CPU utilization and memory utilization, and simply log the data to the console. It won’t be long before you’ll have both applications talking to one another.

Let’s get started, shall we?

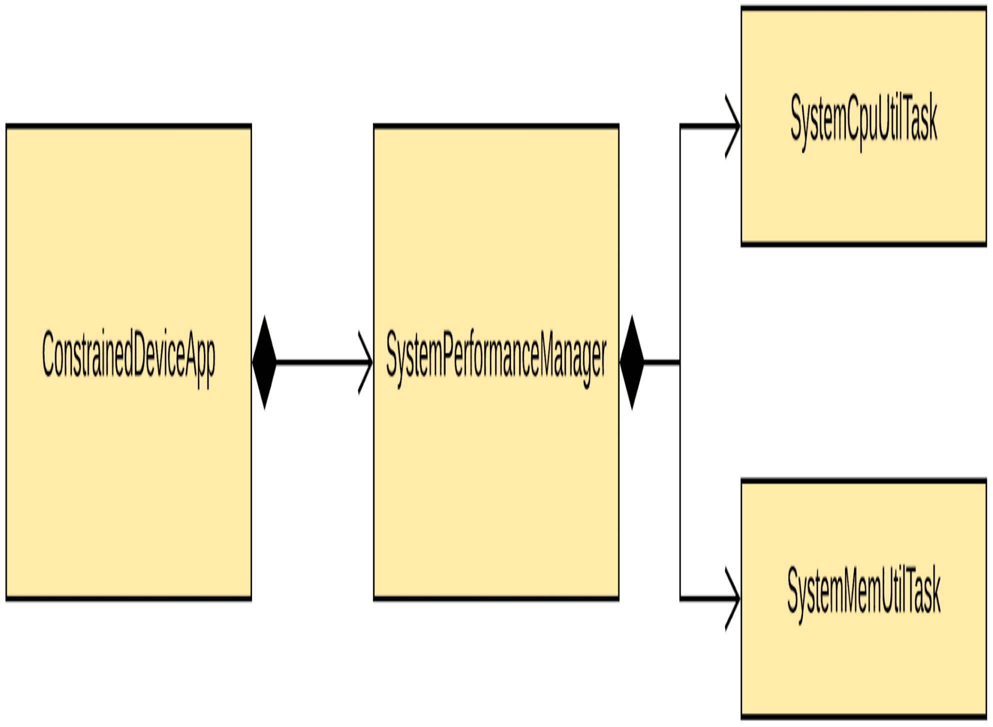
NOTE

Most of the design diagrams throughout the book will follow Unified Modeling Language (UML) notation. I'll explain them as we go, but you'll want to become familiar with the UML's class diagram and relationship constructs. Check out the links at https://www.uml.org/resource-hub.htm for tutorials and other helpful UML resources.

The CDA

Your CDA needs four components: an application wrapper, a manager to run the show, and two components to read the basic system performance data we want to collect as part of our telemetry: CPU utilization and memory utilization.

Let's use Figure 2-1 to represent this design:



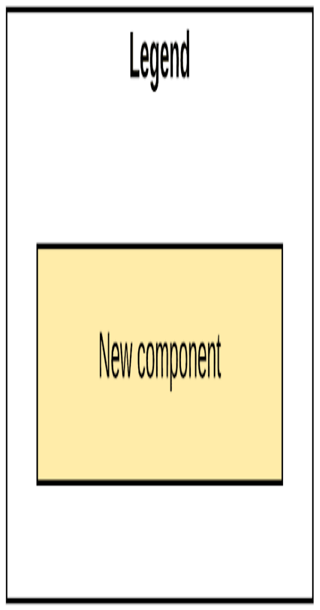


Figure 2-1. - CDA system performance design diagram

The class diagram in Figure 2-1 shows the four primary components and their relationship to one another: ConstrainedDeviceApp, which you'll recall from Chapter 1 as being part of the code base you've already downloaded, is the application wrapper.

ConstrainedDeviceApp is the entry point for the application and creates the instance of, and manages, SystemPerformanceManager. There are also two other components - SystemCpuUtilTask and SystemMemUtilTask. As their names imply, these are components that will collect - you guessed it - system CPU utilization and system memory utilization. These components will be managed by SystemPerformanceManager and run as asynchronous threads that update a method you'll define within SystemPerformanceManager.

NOTE

If you're already familiar with Python development and comfortable implementing your solution from the UML design above, feel free to skim the next section as a reference and write your code. I'd recommend you do walk through CDA Testing Details, however, as it will provide some insights into the unit test framework that's part of your codebase.

CDA Implementation Details

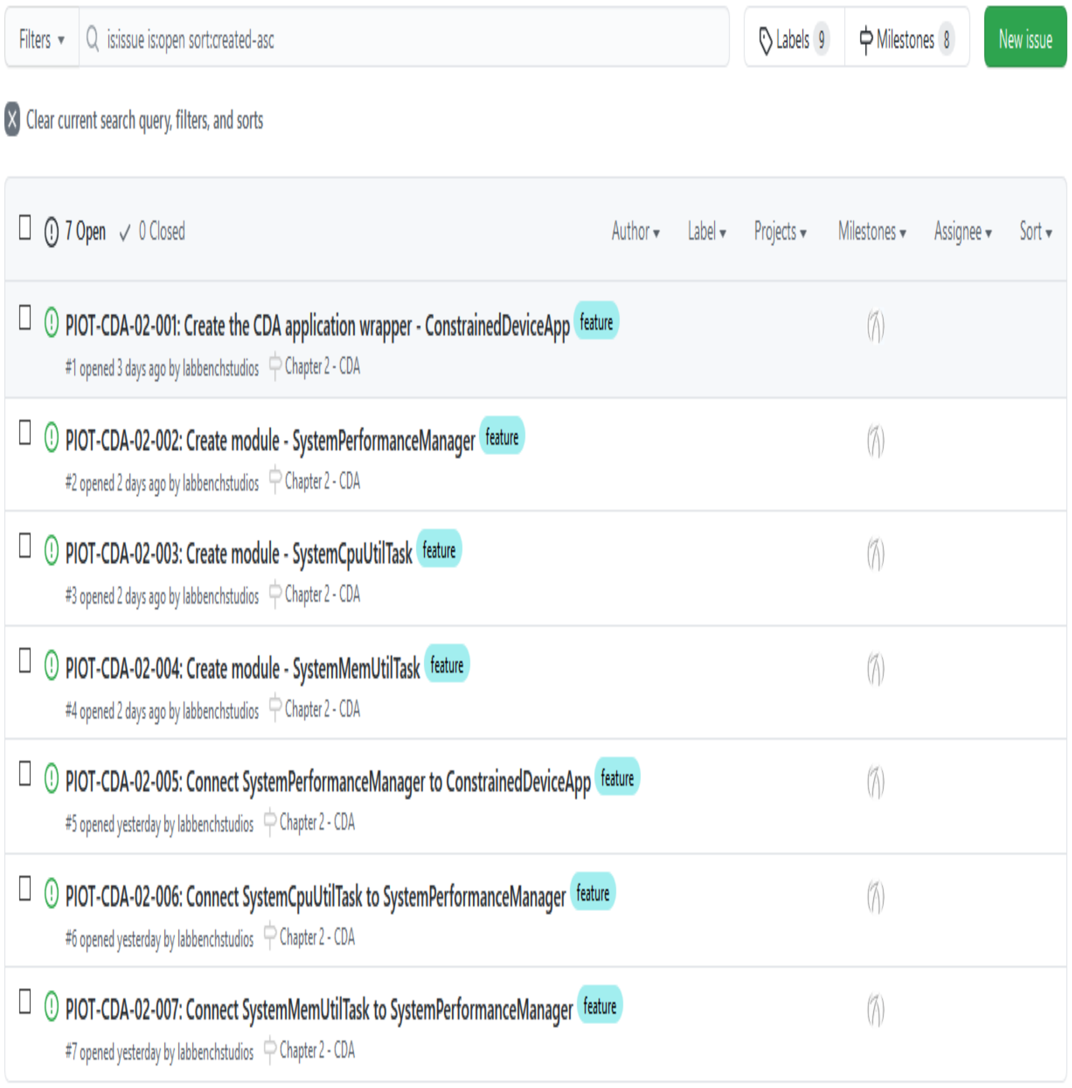
Before you start writing code, there's some administrative steps you'll need to take to get your CDA code repository and Issues tracker setup. This will let you easily track your work within your project’s Kanban board. Let’s do this administrative work now.

NOTE

You may find this section to be a bit pedantic. Don’t worry! Once you get through some of the drudgery that comes next, the following code examples will be easier to digest and move more quickly. There are two reasons for this:

1. The setup work will be done after this section; you’ll know how to add tasks, write meaningful descriptions, and map them to your Kanban board.
2. I’ll move along a bit faster and only explain the code that’s somewhat specialized or unique to IoT environments. My expectation is that you’ll ramp up quickly and won’t need a line-by-line description of everything you need to write.

As I mentioned in Chapter 1, the tasks you write are called ‘Issues’. Figure 2-2 is a screenshot from my GitHub page displaying all the CDA tasks that need to be implemented for this chapter.


Filters ▾ is:issue is:open sort:created-asc
Labels 9 Milestones 8 New issue
Clear current search query, filters, and sorts
7 Open ✓ 0 Closed Author ▾ Label ▾ Projects ▾ Milestones ▾ Assignee ▾ Sort ▾

PIOT-CDA-02-001: Create the CDA application wrapper - ConstrainedDeviceApp feature
#1 opened 3 days ago by labbenchstudios Chapter 2 - CDA

PIOT-CDA-02-002: Create module - SystemPerformanceManager feature
#2 opened 2 days ago by labbenchstudios Chapter 2 - CDA

PIOT-CDA-02-003: Create module - SystemCpuUtilTask feature
#3 opened 2 days ago by labbenchstudios Chapter 2 - CDA

PIOT-CDA-02-004: Create module - SystemMemUtilTask feature
#4 opened 2 days ago by labbenchstudios Chapter 2 - CDA

PIOT-CDA-02-005: Connect SystemPerformanceManager to ConstrainedDeviceApp feature
#5 opened yesterday by labbenchstudios Chapter 2 - CDA

PIOT-CDA-02-006: Connect SystemCpuUtilTask to SystemPerformanceManager feature
#6 opened yesterday by labbenchstudios Chapter 2 - CDA

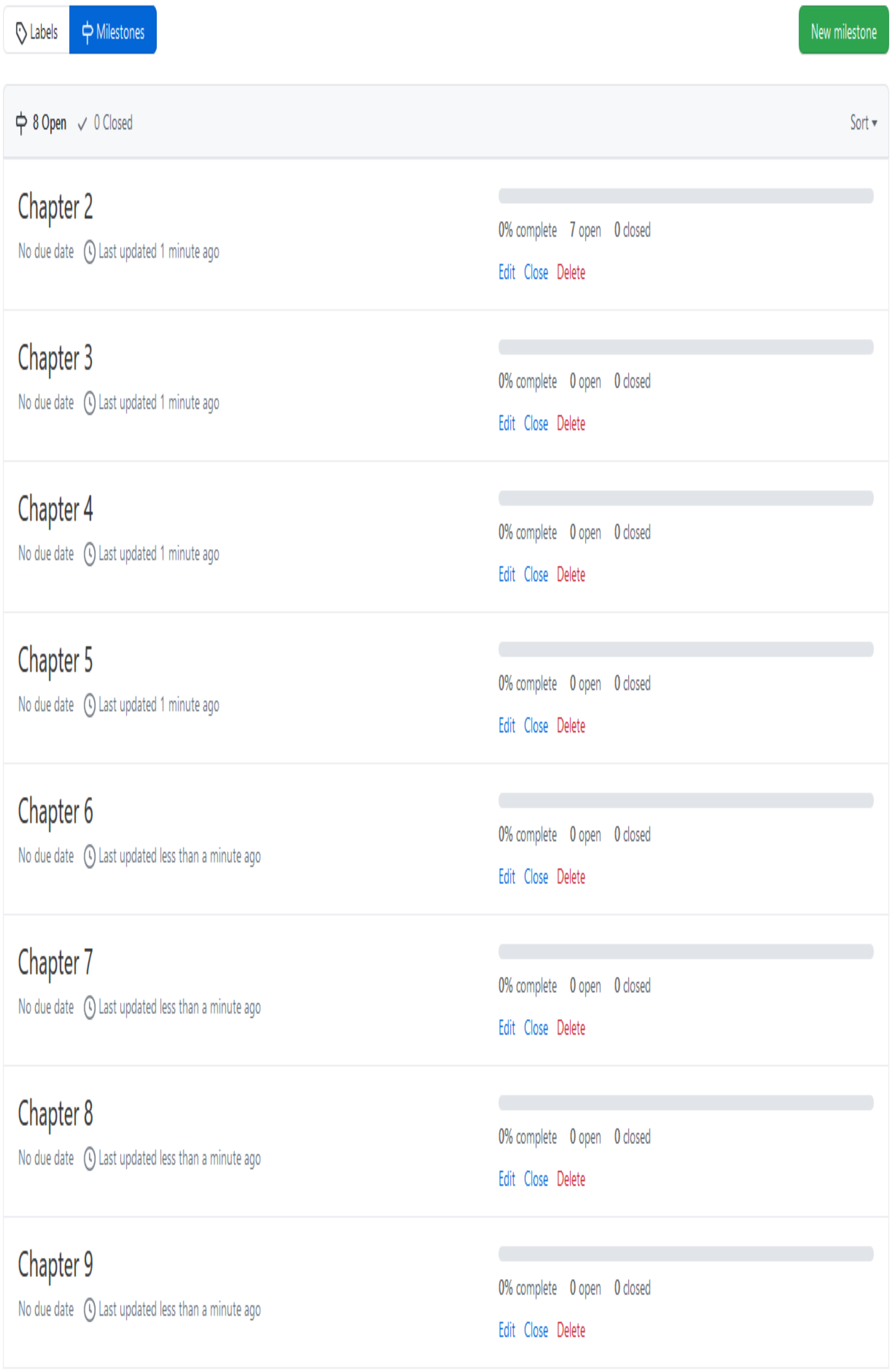
PIOT-CDA-02-007: Connect SystemMemUtilTask to SystemPerformanceManager feature
#7 opened yesterday by labbenchstudios Chapter 2 - CDA

Figure 2-2. - CDA system performance design tasks

If you're using GitHub, use your web browser to navigate to your CDA's Git repository page now. You'll see there's an 'Issues' tab - click that, which will take you to a page similar to Figure 2-2, but without any Issues.

Let's change one of the labels first. Click 'Labels', and look for 'Enhancements'. You can leave it as is or rename it to 'Feature', which I recommend.

Go back to the Issues page, and now click ‘Milestones’. Let’s add in one milestone for each chapter, as shown in Figure 2-3.


Labels Milestones New milestone

8 Open ✓ 0 Closed Sort ▾

Chapter 2
No due date Last updated 1 minute ago
0% complete 7 open 0 closed
Edit Close Delete

Chapter 3
No due date Last updated 1 minute ago
0% complete 0 open 0 closed
Edit Close Delete

Chapter 4
No due date Last updated 1 minute ago
0% complete 0 open 0 closed
Edit Close Delete

Chapter 5
No due date Last updated 1 minute ago
0% complete 0 open 0 closed
Edit Close Delete

Chapter 6
No due date Last updated less than a minute ago
0% complete 0 open 0 closed
Edit Close Delete

Chapter 7
No due date Last updated less than a minute ago
0% complete 0 open 0 closed
Edit Close Delete

Chapter 8
No due date Last updated less than a minute ago
0% complete 0 open 0 closed
Edit Close Delete

Chapter 9
No due date Last updated less than a minute ago
0% complete 0 open 0 closed
Edit Close Delete

Figure 2-3. - Milestone listing for all CDA activities

See the green button? Simply click on that to add your milestone. I'd recommend using the names provided in Figure 2-3, as it will help to track all your work on a chapter-by-chapter basis in your project's Kanban board. You'll see how as we add each task next.

Create the CDA application module

Let's create an Issue with the requirements for ConstrainedDeviceApp. Navigate back to your main CDA repository page and click 'Issues'. There's still a green button in the upper right corner, but it's called 'New issue'. Click that now.

NOTE

Issue tracking tools typically number your issues for you, often in the order created. To make searching and sorting easier, however, it helps to have your own naming convention embedded within the issue title. Here's what I recommend and am using myself: PIOT-{app}-{chapter}-{order}, for example, PIOT-CDA-02-001. A quick look at the name tells me this issue is part of Programming the Internet of Things (PIOT), the CDA, Chapter 02, and task #001.

Name the issue *PIOT-CDA-02-001: Create the CDA application wrapper module*, and add the following contents to the description (feel free to customize if you'd like, but keep the template the same):

Description

- Create the ConstrainedDeviceApp application in Python.

Actions

- Create a new Python package in the programmingtheiot\cda source folder named app and navigate to that folder.
- Import the Python logging module: import logging
- Create a new Python module named ConstrainedDeviceApp. Define a class within the module by the same name of ConstrainedDeviceApp.
- Add the startApp() method, log info message indicating app was started.
- Add the stopApp() method, log info message indicating app was stopped.
- Add the main entry function to enable running as an application. It will create an instance of ConstrainedDeviceApp, call startApp(), wait 60 seconds, then call stopApp(), as follows:

```
def main():
    cda = ConstrainedDeviceApp()
    cda.startApp()

    while True:
        sleep(60)
    cda.stopApp()
if __name__ == '__main__':
    main()
```

Estimate (Small = < 2 hrs ; Medium = 4 hrs ; Large = 8 hrs)

- Small

Tests

- Run the ConstrainedDeviceAppTest unit tests. The log output should look similar to the following (you can ignore the non-INFO messages - these are generated by the PyUnit framework within Eclipse):

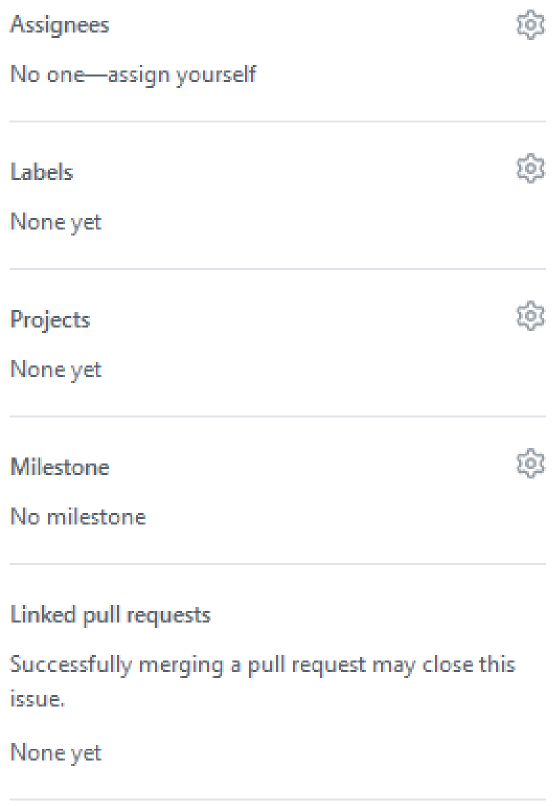
```
Finding files... done.
Importing test modules ... done.
2020-07-20 10:08:20,169:INFO:Initializing CDA...
2020-07-20 10:08:20,169:INFO:Loading
configuration...
2020-07-20 10:08:20,169:INFO:Starting CDA...
2020-07-20 10:08:20,169:INFO:CDA started.
2020-07-20 10:08:20,169:INFO:CDA stopping...
2020-07-20 10:08:20,170:INFO:CDA stopped with exit
code 0.
---------------------------------------------------
-------------------
Ran 1 test in 0.001s
OK
```

Notice the template includes sections for Description, Actions, Estimate, and Tests. This should be consistently applied to all your tasks - this way, when you're reviewing what needs to be done, and what's been accomplished, you have a consistent set of requirements and testing criteria for each one.

NOTE

As you begin implementation, you can add separate notes to chronicle your work. It's often useful to add specifics such as algorithm idiosyncrasies, testing setup requirements, static design images, etc.

Save the issue, then navigate to the far right column. It will look similar to Figure 2-4. This is where you can assign the issue (to yourself), set the label to 'Feature', add a milestone, and align the issue to a project (the one you created in the previous chapter - this will ensure it shows up in your Kanban project board.


Assignees
No one—assign yourself

Labels
None yet

Projects
None yet

Milestone
No milestone

Linked pull requests
Successfully merging a pull request may close this issue.
None yet

Figure 2-4. - You can set properties for each issue using the selections shown

OK - all the current requirements for the CDA application wrapper are known, so it's time to start writing code!

1. Open your IDE and navigate to piot-python-components.
2. Navigate to 'src -> main -> python -> programmingtheiot -> cda' and create a new folder named 'app'.
3. Add the remaining code as indicated in the task, including the logging.

NOTE

Since there's a sleep call in the main function, you'll need to import the time module using the following code:

```
from time import sleep
```

Follow the Test instructions to execute the ConstrainedDeviceAppTest unit test (there's only one). Check the output in your IDE console - it will look similar to what I've included above.

Your ConstrainedDeviceApp is beginning to come together, but you still need a way to collect data and generate your first telemetry data. We'll work on that next.

CREATE THE SYSTEMPERFORMANCEMANAGER MODULE

The next issue to create is for the SystemPerformanceManager. Let's name it *PIOT-CDA-02-002 - Create module SystemPerformanceManager*. Here's the description content:

Description

- Create a new Python module named SystemPerformanceManager with class name SystemPerformanceManager.

Actions

- Create a class within the module named SystemPerformanceManager.
- Add the startManager() method, log an info message indicating manager was started.
- Add the stopManager() method, log an info message indicating manager was stopped.

Estimate (Small = < 2 hrs ; Medium = 4 hrs ; Large = 8 hrs)

- Small

Tests

- Run the SystemPerformanceManagerTest unit test. The log output should look similar to the following:

  ```
  2020-07-06 21:03:03,654:INFO:Initializing
  SystemPerformanceManager...
  2020-07-06 21:03:03,655:INFO:Started
  SystemPerformanceManager.
  2020-07-06 21:03:03,656:INFO:Stopped
  SystemPerformanceManager.
  ```

This component will actually live in a different folder than the application, so let’s create that now. In your IDE, navigate to ‘src -> main -> python -> programmingtheiot -> cda’, and create a new folder named ‘system’.

The requirements for SystemPerformanceManager are pretty straight forward, so see if you can implement these on your own. You can always review the solutions located in my python-solutions GitHub page if you need a hint (https://github.com/programmingtheiot/python-solutions)!

CONNECT SYSTEMPERFORMANCEMANAGER TO CONSTRAINEDDEVICEAPP

This task is really straight forward and should only take a few minutes. You'll simply connect the SystemPerformanceManager start and stop methods into the ConstrainedDeviceApp start and stop methods, then run a quick test to ensure everything is working correctly.

You can name this task *PIOT-CDA-02-003 - Connect SystemPerformanceManager to ConstrainedDeviceApp*. Here's the task description:

Description

- Create an instance of SystemPerformanceManager within ConstrainedDeviceApp and invoke the manager's start / stop methods within the app's start / stop methods.

Actions

- Create a class-scoped instance of SystemPerformanceManager within the ConstrainedDeviceApp constructor called sysPerfManager using the following:
  self.sysPerfManager = SystemPerformanceManager()
- Edit the startApp() method to include a call to self.sysPerfManager.startManager().
- Edit the stopApp() method to include a call to self.sysPerfManager.stopManager().

Estimate (Small = < 2 hrs ; Medium = 4 hrs ; Large = 8 hrs)

- Small

Tests

- Run the ConstrainedDeviceAppTest again. It will still call the startApp() and stopApp() methods on ConstrainedDeviceApp, but this time generate output that will look similar to the following:

```
Finding files... done.
Importing test modules ... done.
2020-07-20 10:23:00,146:INFO:Initializing CDA...
2020-07-20 10:23:00,147:INFO:Loading
configuration...
2020-07-20 10:23:00,260:INFO:Starting CDA...
2020-07-20 10:23:00,260:INFO:Started
SystemPerformanceManager.
2020-07-20 10:23:00,260:INFO:CDA started.
2020-07-20 10:23:00,260:INFO:CDA stopping...
2020-07-20 10:23:00,260:INFO:Stopped
SystemPerformanceManager.
2020-07-20 10:23:00,260:INFO:CDA stopped with
exit code 0.
--------------------------------------------------
---------------------
Ran 1 test in 0.115s
OK
```

Pretty simple, right? OK - almost there. Let’s create the different modules that will actually do the system monitoring next, and get them connected to the SystemPerformanceManager. You can see where I’m going with this, right?

CREATE THE SYSTEMCPUUTILTASK MODULE

This is where the rubber starts to hit the road for our system performance application. The SystemCpuUtilTask will retrieve the current CPU utilization across all cores, average them together, and return the result as a float.

Let's name it *PIOT-CDA-02-003 - Create module SystemCpuUtilTask.* Here's the description content:

Description

- Create a new Python module named SystemCpuUtilTask with class name SystemCpuUtilTask.

Actions

- Import the psutil library.
- Create a class within the module named SystemCpuUtilTask.
- In the constructor, add the following:
  self.perfMgr = psutil()
- Add the getTelemetry() method, and add the following:
  cpuUtilPct = self.perfMgr.cpu_percent()
- Within getTelemetry(), log an info message indicating data was collected along with the value of cpuUtilPct.

Estimate (Small = < 2 hrs ; Medium = 4 hrs ; Large = 8 hrs)

- Small

Tests

- Run SystemCpuUtilTaskTest as a PyUnit test. All tests should pass.

NOTE

You may recall psutil was one of the libraries you imported when setting up your virtualenv environment and used pip to install basic_imports.txt. This library gives you the ability to monitor system metrics, such as CPU utilization and memory utilization. You can read all about its features online (https://psutil.readthedocs.io/en/latest/#).

Look carefully at the action specifying the following line of code:

```
puUtilPct = self.perfMgr.cpu_percent()
```

The library will aggregate all cores into a single CPU percentage, which makes life rather easy for you as a developer. This class essentially boils down to returning the value of this single line of code.

OK, so before we go any further, why the fancy wrapper around the code? Isn't this overkill? Simply put, yes it is. But you'll see a pattern emerge in how you'll write similar code to represent other, more complicated sensors. The point of this particular exercise is certainly to obtain the CPU utilization, but also to establish a pattern of *separation of control*, or separation of key functions so they can be managed and updated separately from the rest of the application's logic.

NOTE

As with any software design, there's a balance between complexity, clever coding, and just getting it done. I'll attempt to strike that balance throughout the book. You may have different ideas - that's great! I encourage you to consider how else you might implement each exercise to meet your specific needs. This is part of what makes programming creative and fun.

CREATE THE SYSTEMMEMUTILTASK MODULE

Assuming your tests for SystemCpuUtilTask run and all pass, you can move onto the next module, which will be the creation of SystemMemUtilTask. This will retrieve the current virtual memory utilization and return the result as a float.

Let's name it *PIOT-CDA-02-004 - Create module SystemMemUtilTask.* Here's the description content:

Description

- Create a new Python module named SystemMemUtilTask with class name SystemMemUtilTask.

Actions

- Import the psutil library.
- Create a class within the module named SystemMemUtilTask.
- In the constructor, add the following:
  self.perfMgr = psutil()

- Add the getTelemetry() method, and add the following: memUtilPct = self.perfMgr.virtual_memory().percent
- Within getTelemetry(), log an info message indicating data was collected along with the value of memUtilPct.

Estimate (Small = < 2 hrs ; Medium = 4 hrs ; Large = 8 hrs)

- Small

Tests

- Run SystemMemUtilTask as a PyUnit test. All tests should pass.

The key functionality is encapsulated in the following line of code:

```
memUtilPct = self.perfMgr.virtual_memory().percent
```

There are other properties you can extract from the call to virtual_memory(), and you’re welcome to experiment. For now, just return the percent utilization.

Make sure your SystemMemUtilTaskTest PyUnit tests all pass before moving on. Now, let’s connect both SystemCpuUtilTask and SystemMemUtilTask to SystemPerformanceManager.

You can repeat these same steps for SystemMemUtilTask, but use the correct name of course! I’d suggest doing that now, and then run the SystemMemUtilTaskTest unit test to ensure it’s returning valid data.

How to Poll for Sensor Updates

Reading a single value from a sensor (emulated or real) is good, but not very useful for our purposes, since we'll want to monitor these values over time to see if they change. Even after connecting SystemCpuUtilTask and SystemMemUtilTask into SystemPerformanceManager, you'll want to process their data on a recurring basis.

There are many ways to do this in Python: You can build your own scheduling mechanism using Python's concurrency library, or leverage one of many open source libraries to do this for you. I've included apscheduler[1] in basic_imports.txt, which provides a scheduling mechanism that will suit our purposes rather well.

NOTE

Python provides two mechanisms for running code in a way that appears to execute simultaneously with other code. One is using concurrency, and the other is using multiprocessing. The former is handled using threads, whereas the latter is handled using separate child processes. One key difference is that Python threads actually get run in sequence using the same processor core as the main application, but they happen in such a way as to appear to be running simultaneously. Multiprocessing allows for true parallelism, where the code written using the multiprocessor library can be distributed to run on a separate processor core, which can execute in parallel to other code in a different processor core. The CDA-specific exercises and samples within this book will assume that threaded concurrency in Python is sufficient for our needs, and so I won't discuss multiprocessing for any CDA development.

INTEGRATING SYSTEMCPUUTILTASK AND SYSTEMMEMUTILTASK WITH

SYSTEMPERFORMANCEMANAGER

In this section, you'll create two very similar tasks: *PIOT-CDA-02-006 - Connect SystemCpuUtilTask to SystemPerformanceManager*, and *PIOT-CDA-02-007 - Connect SystemMemUtilTask to SystemPerformanceManager*.

Here's the description content for *PIOT-CDA-02-006 - Connect SystemCpuUtilTask and SystemMemUtilTask to SystemPerformanceManager*:

Description

- Create an instance of SystemCpuUtilTask and SystemMemUtilTask within SystemPerformanceManager and use the apscheduler library to run each task at a regular interval.

Actions

- Add the following import statement:
  from apscheduler.schedulers.background import BackgroundScheduler
- Update the SystemPerformanceManager constructor to accept a parameter named pollRate and set the default to 30:
  def __init__(self, pollRate = 30):
- Create a class-scoped instance of SystemCpuUtilTask within the SystemPerformanceManager constructor named cpuUtilTask
- Create a class-scoped instance of SystemMemUtilTask within the SystemPerformanceManager constructor named memUtilTask

- Create a public method named handleTelemetry() and add the following lines of code:

  ```
  cpuUtilPct = self.cpuUtilTask.getTelemetry()
  memUtilPct = self.memUtilTask.getTelemetry()
  ```

- Within handleTelemetry() add an informational log message that logs the values of cpuUtilPct and memUtilPct
- Create a class-scoped instance of BackgroundScheduler within the SystemPerformanceManager constructor named scheduler
- Add a job to the scheduler within the SystemPerformanceManager using the following line of code:
  self.scheduler.add_job('self.handleTelemetry', 'interval', seconds = pollRate)

Estimate (Small = < 2 hrs ; Medium = 4 hrs ; Large = 8 hrs)

- Small

Tests

- Add a unit test to SystemPerformanceManagerTest that will create the SystemPerformanceManager instance with pollRate = 10, call the startManager() method, wait for 65 seconds, and then call stopManager().
- The test should display the application and SystemPerformanceManager start log messages, followed by a number of interspersed CPU utilization and memory utilization messages representing six (mostly) different values each, and then finally the SystemPerformanceManager and application stop log messages.

NOTE

Notice the use of apscheduler - it’s relatively straight-forward. This is partly because, in Python, you’re able to pass function pointers (yes, we’re calling them methods). This allows the callback you’ve already created in SystemPerformanceManager called handleTelemetry() to be easily invoked by apscheduler using concurrency.

You’re now ready to execute the full functional unit tests test of ConstrainedDeviceApp. Run it now, and examine the log output. It should look similar to the following (again, your log messages may be different than mine):

```
2020-07-06 21:03:03,654:INFO:Attempting to load configuration...
2020-07-06 21:03:03,655:INFO:Starting CDA...
2020-07-06 21:03:03,655:INFO:Initialized SystemPerformanceManager.
2020-07-06 21:03:03,656:INFO:Starting SystemPerformanceManager...
2020-07-06 21:03:03,656:INFO:CDA ran successfully.
```

Congratulations! You’ve just completed the first iteration of the CDA. It’s a fully standalone Python app, with a full suite of unit tests for you to build upon over the next chapters. Now let’s move onto the GDA and start writing some Java code.

The GDA

Your GDA will need the same three components as your CDA: an application wrapper, a system performance manager, and components to read the system performance data that will comprise your GDA’s telemetry.

NOTE

In IoT systems, the gateway device may or may not generate its own telemetry. The example you'll build within this book only generates its own system performance telemetry. The constrained device, implemented as the CDA, will be responsible for not only generating its specific system performance telemetry, but any sensor-specific telemetry as well.

Let's use the class diagram in Figure 2-5 to represent the GDA's design. You'll notice it looks surprisingly similar to our CDA Design (Figure 2-1), with the main differences being the implementation language, the application wrapper name, and - of course - the number of tasks!

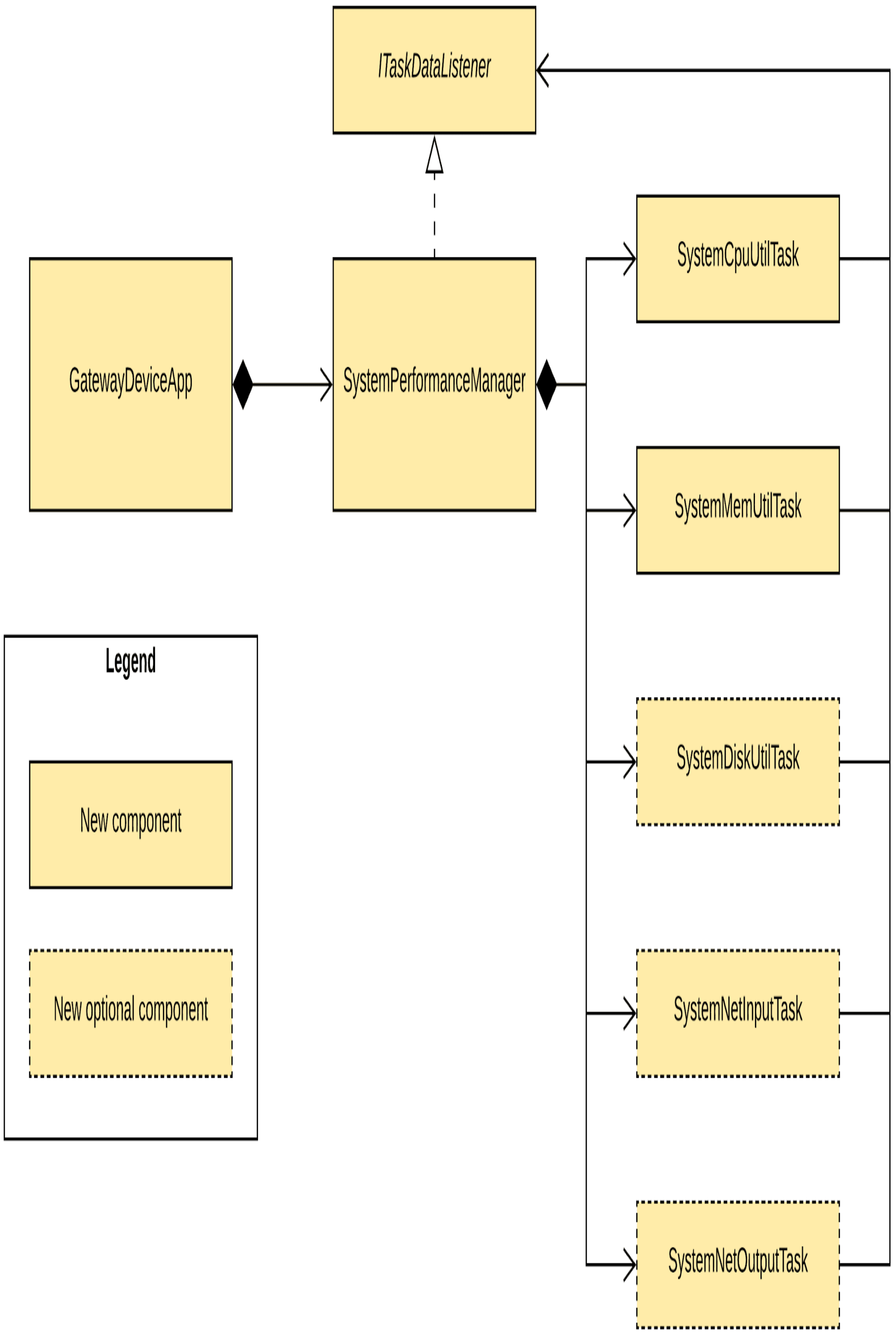


Figure 2-5. - GDA system performance design diagram

Why is the gateway device doing all of this work? Tracking CPU and memory utilization makes sense, since you'll want to track the device's overall load and memory consumption.

But disk and network utilization? In Chapter 5, I'll discuss local caching of messages, so this will be an important metric to track. In Chapter 6 and beyond, I'll dig into passing messages between devices (and eventually the cloud in Chapter 10), so tracking network utilization will be pretty useful as well.

Aren't there tools for this? Sure, and you should use them when they make sense. Devices that are part of the IoT Edge Tier aren't always easy to manage, nor are they always able to participate in network monitoring environments. This doesn't mean we need to build everything from scratch! But, we will anyway. Because we can. And it's fun :).

Back to the diagram. Notice that GatewayDeviceApp is the entry point for the application and creates the instance of, and manages, SystemPerformanceManager, which is rather similar to the CDA design. We also have a SystemCpuUtilTask and SystemMemUtilTask, along with the other tasks (SystemDiskUtilTask, SystemNetInputTask, and SystemNetOutputTask) that will help us later.

NOTE

It goes without saying, but if you're already familiar with Java development and comfortable implementing your solution from the UML design in Figure 2-5, feel free to do so. But be sure to look carefully at the requirements in the next section first.

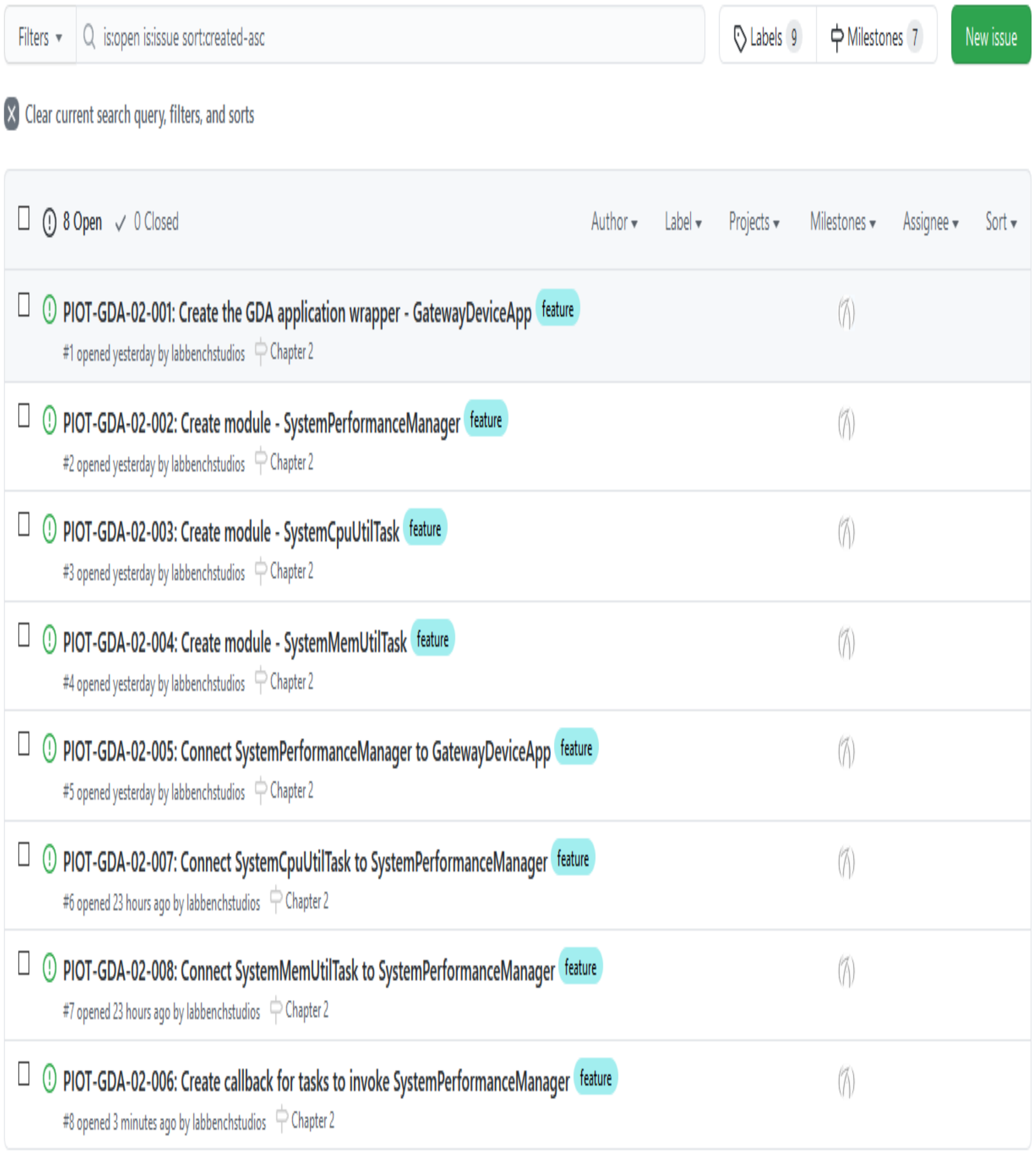
GDA IMPLEMENTATION DETAILS

You'll recognize these next steps, of course, because you already did something similar for the CDA. Since the GDA repository is different from the CDA repository, you'll need to set up your issue tracker separately.

Figure 2-6 gives you the lay of the land regarding the requirements you need to implement for the GDA in this chapter, and I'll walk through each in turn.

Figure 2-7 lists the milestones you should create, but take note that they're a bit different from the CDA milestones. The names are the same (by design), but there are no milestones for Chapter 3 and Chapter 4, and yet there is a milestone for Chapter 10 (the CDA milestones end at Chapter 9).

Now would be a good time to create those milestones, and while you're at it, edit the 'Enhancement' label and rename it to 'Feature'.


Filters ▾ is:open is:issue sort:created-asc

Labels 9 Milestones 7 New issue

Clear current search query, filters, and sorts

8 Open ✓ 0 Closed Author ▾ Label ▾ Projects ▾ Milestones ▾ Assignee ▾ Sort ▾

PIOT-GDA-02-001: Create the GDA application wrapper - GatewayDeviceApp feature
#1 opened yesterday by labbenchstudios Chapter 2

PIOT-GDA-02-002: Create module - SystemPerformanceManager feature
#2 opened yesterday by labbenchstudios Chapter 2

PIOT-GDA-02-003: Create module - SystemCpuUtilTask feature
#3 opened yesterday by labbenchstudios Chapter 2

PIOT-GDA-02-004: Create module - SystemMemUtilTask feature
#4 opened yesterday by labbenchstudios Chapter 2

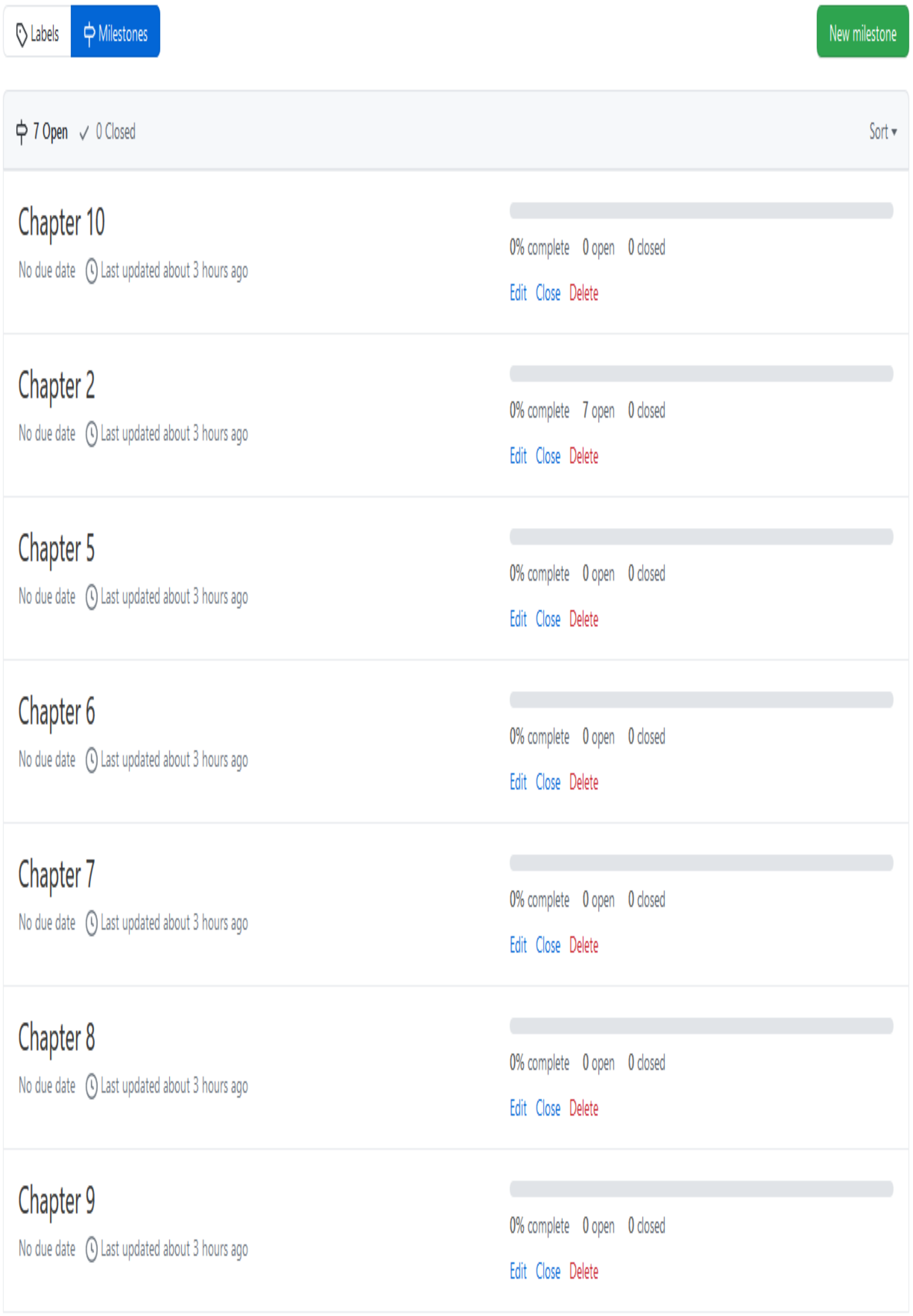
PIOT-GDA-02-005: Connect SystemPerformanceManager to GatewayDeviceApp feature
#5 opened yesterday by labbenchstudios Chapter 2

PIOT-GDA-02-007: Connect SystemCpuUtilTask to SystemPerformanceManager feature
#6 opened 23 hours ago by labbenchstudios Chapter 2

PIOT-GDA-02-008: Connect SystemMemUtilTask to SystemPerformanceManager feature
#7 opened 23 hours ago by labbenchstudios Chapter 2

PIOT-GDA-02-006: Create callback for tasks to invoke SystemPerformanceManager feature
#8 opened 3 minutes ago by labbenchstudios Chapter 2

Figure 2-6. - GDA system performance design tasks



Chapter 10
No due date Last updated about 3 hours ago
0% complete 0 open 0 closed

Chapter 2
No due date Last updated about 3 hours ago
0% complete 7 open 0 closed

Chapter 5
No due date Last updated about 3 hours ago
0% complete 0 open 0 closed

Chapter 6
No due date Last updated about 3 hours ago
0% complete 0 open 0 closed

Chapter 7
No due date Last updated about 3 hours ago
0% complete 0 open 0 closed

Chapter 8
No due date Last updated about 3 hours ago
0% complete 0 open 0 closed

Chapter 9
No due date Last updated about 3 hours ago
0% complete 0 open 0 closed

Figure 2-7. - Milestone listing for all GDA activities

Ready to create your first GDA issue? The process is the same one you followed for your first CDA issue. Navigate back to your main GDA repository page, click ‘Issues’, and then the green ‘New issue’ button. This is the process you’ll follow for each new issue within either repository.

Let’s start with the application wrapper for the GDA.

CREATE THE GDA APPLICATION MODULE

Much like with the CDA, the GDA application module issue will be named *PIOT-GDA-02-001: Create the GDA application wrapper module*. Here are the details:

Description

- Create a new Java class named GatewayDeviceApp.

Actions

- Create a new Java package in the programmingtheiot\gda source folder named app and navigate to that folder.
- Import the java.util.logging logging framework. You can import all, or just Level and Logger.
- Create a package-scoped constructor that accepts a single parameter as follows: private GatewayDeviceApp(String[] args)
- Add the public stopApp(int code) method, and log an info message indicating the app was stopped. Include a try / catch block to handle the stop code. On exception, log an error message along with the stack trace. Outside of the try / catch block, and as the last line of code, log an informational message with the code included.

- Add the public startApp() method, and log an info message indicating the app was started. Include a try / catch block to handle the start code. On exception, log an error message along with the stack trace, then call stopApp(-1).
- Add the private initConfig(String fileName) method, and log an info message indicating the method was called. It will mostly remain empty for now.
- Add the private parseArgs(String[] args) method, and log an info message indicating the method was called. For now, the args can be ignored. Before the method exits, call initConfig(null).
- Update the constructor to include a call to parseArgs(args).
- Add the public static void main(String[] args) method to enable running as an application. It will create an instance of GatewayDeviceApp, call startApp(), wait 60 seconds, then call stopApp(0), as follows:

```
public static void main(String[] args)
{
        GatewayDeviceApp gwApp = new
GatewayDeviceApp(args);

        gwApp.startApp();

        try {
                Thread.sleep(60000L);
        } catch (InterruptedException e) {
                // ignore
        }

        gwApp.stopApp(0);
}
```

Estimate (Small = < 2 hrs ; Medium = 4 hrs ; Large = 8 hrs)

- Small

Tests

- Run the GatewayDeviceAppTest unit test named testRunGatewayApp(). The log output should look similar to the following:

```
Jul 19, 2020 12:53:45 PM
programmingtheiot.gda.app.GatewayDeviceApp <init>
INFO: Initializing GDA...
Jul 19, 2020 12:53:45 PM
programmingtheiot.gda.app.GatewayDeviceApp
parseArgs
INFO: No command line args to parse.
Jul 19, 2020 12:53:45 PM
programmingtheiot.gda.app.GatewayDeviceApp
initConfig
INFO: Attempting to load configuration: Default.
Jul 19, 2020 12:53:45 PM
programmingtheiot.gda.app.GatewayDeviceApp
startApp
INFO: Starting GDA...
Jul 19, 2020 12:53:45 PM
programmingtheiot.gda.app.GatewayDeviceApp
startApp
INFO: GDA started successfully.
Jul 19, 2020 12:53:45 PM
programmingtheiot.gda.app.GatewayDeviceApp
stopApp
INFO: Stopping GDA...
Jul 19, 2020 12:53:45 PM
programmingtheiot.gda.app.GatewayDeviceApp
```

```
stopApp
INFO: GDA stopped successfully with exit code 0.
```

If you've downloaded the sample code, you'll notice that this module is already created for you. Embedded within the module are a few commented 'TODO' lines of code as placeholders for you to eventually add more functionality.

If you're starting from scratch, however, just follow the instructions above to implement your own version of the application wrapper.

Either way, run the unit test as specified - testRunGatewayApp() - by using the JUnit 4 Runner. If you're using Eclipse as your IDE, simply right click on GatewayDeviceAppTest, and select "Run As -> JUnit Test". You should get a green bar along with sample output as indicated under the "Tests" section above.

Your GatewayDeviceApp has some neat stuff in it, but it's not very useful as an IoT gateway application. Let's start working on the other components to bring this app to life.

CREATE THE SYSTEMPERFORMANCEMANAGER MODULE

The next issue to create is for the SystemPerformanceManager. Let's name it *PIOT-GDA-02-002 - Create module SystemPerformanceManager*. Here's the description content:

Description

- Create the SystemPerformanceManager module.

Actions

- Create a new Java package in the programmingtheiot\gda source folder named system and navigate to that folder.
- Import the java.util.logging logging framework. You can import all, or just Level and Logger.
- Create a private variable as follows: private int pollSecs = 60;
- Create a public constructor that accepts a single parameter as follows: public SystemPerformanceManager(int pollSecs)
- Set this.pollSecs to pollSecs only after validating that pollSecs is more than 1 and less than Integer.MAX. You may choose other constraints if you wish.
- Add the startManager() method, and log an info message indicating manager was started.
- Add the stopManager() method, and log an info message indicating the manager was stopped.

Estimate (Small = < 2 hrs ; Medium = 4 hrs ; Large = 8 hrs)

- Small

Tests

- Run the SystemPerformanceManagerTest unit test named testRunManager(). The log output should look similar to the following:

```
Jul 19, 2020 1:01:38 PM
programmingtheiot.gda.system.SystemPerformanceManager
 startManager
INFO: SystemPerformanceManager is starting...
Jul 19, 2020 1:01:38 PM
programmingtheiot.gda.system.SystemPerformanceManager
```

```
 stopManager
INFO: SystemPerformanceManager is stopped.
```

The one currently active unit test for SystemPerformanceManager is rather simple and not *really* automated, in that the only validation is via a log file review. As you hook all the pieces together, you'll see how your unit tests for other classes can be automatically validated, and then used to automate the entire test flow of your system.

CONNECT SYSTEMPERFORMANCEMANAGER TO GATEWAYDEVICEAPP

This is a simple task designed to hook the SystemPerformanceManager to GatewayDeviceApp, and the only objective with this is to ensure the app can properly delegate work to the SystemPerformanceManager component.

You can name this task *PIOT-GDA-02-003: Connect SystemPerformanceManager to GatewayDeviceApp*. Here are the task details:

Description

- Connect SystemPerformanceManager to GatewayDeviceApp so it can be started and stopped with the application. This work should be implemented within the GatewayDeviceApp class.

Actions

- Create a class-scoped variable named sysPerfManager.

- Create an instance of SystemPerformanceManager within the GatewayDeviceApp constructor called this.sysPerfManager. Use ’10’ as the parameter to the constructor.
- Edit the startApp() method: Add a call to sysPerfManager.startManager().
- Edit the stopApp() method: Add a call to sysPerfManager.stopManager().

Estimate (Small = < 2 hrs ; Medium = 4 hrs ; Large = 8 hrs)

- Small

Tests

- Run the GatewayDeviceAppTest unit test named testRunGatewayApp(). The log output should look similar to the following:

```
Jul 19, 2020 1:01:38 PM
programmingtheiot.gda.app.GatewayDeviceApp <init>
INFO: Initializing GDA...
Jul 19, 2020 1:01:38 PM
programmingtheiot.gda.app.GatewayDeviceApp
parseArgs
INFO: No command line args to parse.
Jul 19, 2020 1:01:38 PM
programmingtheiot.gda.app.GatewayDeviceApp
initConfig
INFO: Attempting to load configuration: Default.
Jul 19, 2020 1:01:38 PM
programmingtheiot.gda.app.GatewayDeviceApp
startApp
INFO: Starting GDA...
Jul 19, 2020 1:01:38 PM
```

```
programmingtheiot.gda.system.SystemPerformanceManager
 startManager
INFO: SystemPerformanceManager is starting...
Jul 19, 2020 1:01:38 PM
programmingtheiot.gda.app.GatewayDeviceApp
startApp
INFO: GDA started successfully.
Jul 19, 2020 1:01:38 PM
programmingtheiot.gda.app.GatewayDeviceApp
stopApp
INFO: Stopping GDA...
Jul 19, 2020 1:01:38 PM
programmingtheiot.gda.system.SystemPerformanceManager
 stopManager
INFO: SystemPerformanceManager is stopped.
Jul 19, 2020 1:01:38 PM
programmingtheiot.gda.app.GatewayDeviceApp
stopApp
INFO: GDA stopped successfully with exit code 0.
```

Simple, right? Notice the testing requirement - it’s not on the SystemPerformanceManager; you’re simply rerunning the GatewayDeviceAppTest unit test called testRunGatewayApp(). Neither this test nor the SystemPerformanceManager test has changed - the connection between the two is now enabled, so the output is a bit different.

With both components integrated, let’s build out the actual system performance telemetry collection logic.

CREATE THE SYSTEMCPUUTILTASK MODULE

This is where the rubber starts to hit the road for our system performance application. The SystemCpuUtilTask will retrieve the current CPU utilization across all cores, average them together, and return the result as a float.

Let’s name it *PIOT-CDA-02-004 - Create module SystemCpuUtilTask.* Here’s the description content:

Description

- Create the SystemCpuUtilTask module and implement the functionality to retrieve CPU utilization.

Actions

- Within the programmingtheiot.gda.system package, create a new Java class named SystemCpuUtilTask.
- Add the following import statements:
- import java.lang.management.ManagementFactory;
- Add the public getTelemetry() method. It will retrieve CPU utilization (averaged across any / all cores) and return the value as a float. Use the following code for the value: ManagementFactory.getOperatingSystemMXBean().getSystemLoadAverage()

Estimate (Small = < 2 hrs ; Medium = 4 hrs ; Large = 8 hrs)

- Small

Tests

- Run the SystemCpuUtilTaskTest unit test (there’s only one). If your Operating System supports retrieval of CPU load, each test should pass while displaying values greater than 0.0% and (likely) less than 100.0%. If your

Operating System doesn't support this, each test will result in negative value, as follows:

```
Test 1: CPU Util not supported on this OS: -1.0
Test 2: CPU Util not supported on this OS: -1.0
Test 3: CPU Util not supported on this OS: -1.0
Test 4: CPU Util not supported on this OS: -1.0
Test 5: CPU Util not supported on this OS: -1.0
```

NOTE

You may recall psutil was one of the libraries you imported when setting up your virtualenv environment and used pip to install basic_imports.txt. This library gives you the ability to monitor system metrics, such as CPU utilization and memory utilization. You can read all about its features online (https://psutil.readthedocs.io/en/latest/#).

Look carefully at the action specifying the following line of code:

```
puUtilPct = self.perfMgr.cpu_percent()
```

The library will aggregate all cores into a single CPU percentage, which makes life rather easy for you as a developer. This class essentially boils down to returning the value of this single line of code.

OK, so before we go any further, why the fancy wrapper around the code? Isn't this overkill? Simply put, yes it is. But you'll see a pattern emerge in how you'll write similar code to represent other, more complicated sensors.

One object with each exercise is to establish a pattern of *separation of control*, or the separation of key functions so they can be managed and updated separately from the rest of the application's logic.

> **NOTE**
>
> As with any software design, there's a balance between complexity, clever coding, and just getting it done. I'll attempt to strike that balance throughout the book. You may have different ideas - that's great! I encourage you to consider how else you might implement each exercise to meet your specific needs. This is part of what makes programming creative and fun.

CREATE THE SYSTEMMEMUTILTASK MODULE

I'm sure you've noticed a pattern here. We're replicating some of the functionality within the CDA here within the GDA. This is due to the nature of distributed IoT devices and various operating or runtime environments, as I've mentioned previously in the Preface.

This next task is to create the memory utilization collection logic. Let's name it *PIOT-GDA-02-005 - Create module SystemMemUtilTask*, and use the following as the description content:

Description

- Create the SystemMemUtilTask module and implement the functionality to retrieve JVM memory utilization.

Actions

- Within the programmingtheiot.gda.system package, create a new Java class named SystemMemUtilTask.
- Add the following import statements:
- import java.lang.management.MemoryUsage;
- Add the getTelemetry() method. It will retrieve JVM memory utilization and return the value as a float. Use the following code for the value: ManagementFactory.getMemoryMXBean().getHeapMemoryUsage().getUsed()

Estimate (Small = < 2 hrs ; Medium = 4 hrs ; Large = 8 hrs)

- Small

Tests

- Run the SystemMemUtilTaskTest unit test. It should pass while logging values between 0.0% and 100.0%.

The key functionality is encapsulated in the following line of code:

```
memUtilPct = self.perfMgr.virtual_memory().percent
```

There are other properties you can extract from the call to virtual_memory(), and you’re welcome to experiment. For now, just return the percent utilization.

Make sure your SystemMemUtilTaskTest JUnit tests all pass before moving on. Now, let’s connect both SystemCpuUtilTask and SystemMemUtilTask to SystemPerformanceManager.

RUNNING REPEATABLE TASKS IN JAVA

Although your GDA isn’t collecting sensor data, it clearly needs to gather and assess its own system performance (such as CPU and memory utilization, of course). This type of functionality is typically collected in the background at regular intervals, and as with Python, Java provides options for creating polling systems.

Fortunately, there’s no need to import any separate libraries, however, because this is built-in to the core Java SDK.

NOTE

Java’s concurrency library is quite powerful, and allows you to use a basic Timer functionality as well as a ScheduledExecutorService (you can also create your own threaded polling system if you really want to, of course). We’ll use ScheduledExecutorService, as it provides a semi-guaranteed way to poll at regular intervals, handling most of the complexity for us. Modern Java virtual machines will handle the load distribution across the CPU architecture, meaning it will utilize multiple cores if at all possible.

INTEGRATING SYSTEMCPUUTILTASK AND SYSTEMMEMUTILTASK WITH SYSTEMPERFORMANCEMANAGER

In this section, you’ll create two very similar tasks: *PIOT-GDA-02-006 - Connect SystemCpuUtilTask and SystemMemUtilTask to SystemPerformanceManager*.

This particular exercise is a bit more involved, even though it doesn’t take many lines of code to complete. This is because you’ll be using both concurrency and a *Runnable* implementation, the latter of which is just an interface definition for a method that can be invoked one or more times by a Java thread.

Let’s document this work using the following description:

Description

- Connect SystemCpuUtilTask and SystemMemUtilTask into SystemPerformanceManager, and call each instance’s handleTelemetry() method from within a thread that starts when the manager is started, and stops when the manager is stopped. This work should be implemented within the SystemPerformanceManager class.

Actions

- Add the following import statements:

  ```
  import java.util.concurrent.Executors;
  import
  java.util.concurrent.ScheduledExecutorService;
  import java.util.concurrent.ScheduledFuture;
  import java.util.concurrent.TimeUnit;
  ```

- Add the following members to the class:

  ```
  private ScheduledExecutorService schedExecSvc =
  null;
  private SystemCpuUtilTask sysCpuUtilTask = null;
  private SystemMemUtilTask sysMemUtilTask = null;
  private Runnable taskRunner = null;
  private boolean isStarted = false;
  ```

- Create a public method named handleTelemetry() that includes the following:
  - cpuUtilPct = this.cpuUtilTask.getTelemetry()

        - memUtilPct = this.memUtilTask.getTelemetry()
        - Log an info message that includes the values of cpuUtilPct and memUtilPct

- Within the constructor, add the following:

```
this.schedExecSvc =
Executors.newScheduledThreadPool(1);
this.sysCpuUtilTask = new SystemCpuUtilTask();
this.sysMemUtilTask = new SystemMemUtilTask();
this.taskRunner = () -> {
    this.handleTelemetry();
};
```

- Within the startManager() method, add the following:

```
if (! this.isStarted) {
    ScheduledFuture<?> futureTask =
this.schedExecSvc.scheduleAtFixedRate(this.taskRunner,
 0L, this.pollSecs, TimeUnit.SECONDS);
    this.isStarted = true;
}
```

- Within the stopManager() method, add the following: this.schedExecSvc.shutdown();

  Estimate (Small = < 2 hrs ; Medium = 4 hrs ; Large = 8 hrs)

- Medium

Tests

- In the GatewayDeviceAppTest test case, comment out the @Test annotation before the testRunGatewayApp() unit test, and uncomment the @Test annotation before the testRunTimedGatewayApp unit test. Run the latter test - it should yield output similar to the following:

```
Jul 19, 2020 1:53:19 PM
programmingtheiot.gda.app.GatewayDeviceApp <init>
INFO: Initializing GDA...
Jul 19, 2020 1:53:19 PM
programmingtheiot.gda.app.GatewayDeviceApp
parseArgs
INFO: No command line args to parse.
Jul 19, 2020 1:53:19 PM
programmingtheiot.gda.app.GatewayDeviceApp
initConfig
INFO: Attempting to load configuration: Default.
Jul 19, 2020 1:53:19 PM
programmingtheiot.gda.app.GatewayDeviceApp
startApp
INFO: Starting GDA...
Jul 19, 2020 1:53:19 PM
programmingtheiot.gda.system.SystemPerformanceManager
 startManager
INFO: SystemPerformanceManager is starting...
Jul 19, 2020 1:53:19 PM
programmingtheiot.gda.app.GatewayDeviceApp
startApp
INFO: GDA started successfully.
Jul 19, 2020 1:53:20 PM
programmingtheiot.gda.system.SystemPerformanceManager
 handleTelemetry
INFO: Handle telemetry results: cpuUtil=-1.0,
memUtil=6291456.0
Jul 19, 2020 1:53:50 PM
programmingtheiot.gda.system.SystemPerformanceManager
 handleTelemetry
INFO: Handle telemetry results: cpuUtil=-1.0,
memUtil=6291456.0
Jul 19, 2020 1:54:20 PM
programmingtheiot.gda.system.SystemPerformanceManager
 handleTelemetry
```

```
INFO: Handle telemetry results: cpuUtil=-1.0,
memUtil=6291456.0
Jul 19, 2020 1:54:24 PM
programmingtheiot.gda.app.GatewayDeviceApp
stopApp
INFO: Stopping GDA...
Jul 19, 2020 1:54:24 PM
programmingtheiot.gda.system.SystemPerformanceManager
 stopManager
INFO: SystemPerformanceManager is stopped.
Jul 19, 2020 1:54:24 PM
programmingtheiot.gda.app.GatewayDeviceApp
stopApp
INFO: GDA stopped successfully with exit code 0.
```

To test all of this new goodness, you only need to run the GatewayDeviceAppTest unit test named testRunTimedGatewayApp(). Follow the instructions listed above under the Test section.

NOTE

JUnit unit tests can be included or excluded in a test run by using the @Test annotation before the unit test method. You can simply comment it out / uncomment it as desired. Note also that unit tests are not designed to run in any particular order - you should expect any random order and write your tests as standalone.

The sample output is provided above - notice that it’s quite extensive! This is because you’re not only doing a bunch of cool stuff, you’re also running the app for over a minute.

If your test run yields similar output, fantastic! Now you can really celebrate. You’ve just completed the first iteration of both IoT Edge Tier applications - the GDA and the CDA. The rest of this book is about adding functionality to these applications, connecting them together, and eventually hooking everything up to a cloud service. Buckle up!

Additional Exercises

Figure 2-5 indicates the GDA actually has more than two telemetry collection tasks. These are optional, but still important. See if you can add the SystemDiskUtilTask, SystemNetInTask and SystemNetOutTask components using the patterns for SystemCpuUtilTask and SystemMemUtilTask.

In fact, it’s not a bad idea to implement these for the CDA as well, although it’s less relevant as your CDA won’t be collecting and storing much data.

Don’t forget to add your unit test cases for each!

---

1 You can read more about apscheduler on its website (https://apscheduler.readthedocs.io/en/latest/userguide.html).

About the Author

Andy King is a seasoned computer scientist, educator, and technology executive with over 20 years of experience, largely focused on designing and building network management tools, telematics systems, sensor networks, and–more recently–the Internet of Things ecosystem. As a Department Head, he's led IoT research and integration projects, currently advises clients across North America on a wide range of IoT initiatives and teaches the Connected Devices course in the Cyber Physical Systems program at Northeastern University in Boston, MA.