

Design and Build Great Web APIs

Robust, Reliable, and Resilient



Mike Amundsen
edited by Katharine Dvorak

Design and Build Great Web APIs

Robust, Reliable, and Resilient

by Mike Amundsen

Version: P1.0 (October 2020)

Copyright © 2020 The Pragmatic Programmers, LLC. This book is licensed to the individual who purchased it. We don't copy-protect it because that would limit your ability to use it for your own purposes. Please don't break this trust—you can use this across all of your devices but please do not share this copy with other members of your team, with friends, or via file sharing services. Thanks.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

About the Pragmatic Bookshelf

The Pragmatic Bookshelf is an agile publishing company. We're here because we want to improve the lives of developers. We do this by creating timely, practical titles, written by programmers for programmers.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at <http://pragprog.com>.

Our ebooks do not contain any Digital Restrictions Management, and have always been DRM-free. We pioneered the beta book concept, where you can purchase and read a book while it's still being written, and provide feedback to the author to help make a better book for everyone. Free resources for all purchasers include source code downloads (if applicable), errata and discussion forums, all available on the book's home page at pragprog.com. We're here to make your life easier.

New Book Announcements

Want to keep up on our latest titles and announcements, and occasional special offers? Just create an account on pragprog.com (an email address and a password is all it takes) and select the checkbox to receive newsletters. You can also follow us on twitter as @pragprog.

About Ebook Formats

If you buy directly from pragprog.com, you get ebooks in all available formats for one price. You can sync your ebooks amongst all your devices (including iPhone/iPad, Android, laptops, etc.) via Dropbox. You get free updates for the life of the edition. And, of course, you can always come back and re-download your books when needed. Ebooks bought from the Amazon Kindle store are subject to Amazon's policies. Limitations in Amazon's file format may cause ebooks to display differently on different devices. For more information, please see our FAQ at pragprog.com/frequently-asked-questions/ebooks. To learn more about this book and access the free resources, go to <https://pragprog.com/book/maapis>, the book's homepage.

Thanks for your continued support,

Andy Hunt
The Pragmatic Programmers

The team that produced this book includes: Andy Hunt (Publisher),
Janet Furlow (VP of Operations), Dave Rankin (Executive Editor),
Molly McBeath (Copy Editor), Potomac Indexing, LLC (Indexing), Gilson Graphics (Layout)

For customer support, please contact support@pragprog.com.

For international rights, please contact rights@pragprog.com.

Table of Contents

Acknowledgments

Preface

- Your API Journey
 - Who Should Read This Book
 - How This Book Is Organized
 - What's Covered (And What's Not)
 - About the BigCo, Inc., Sample Project
 - Online Resources
-

Part I. Getting Started

1. Getting Started with API First

- Adopting the API-First Principle
- Exploring APIs with curl
- What's Next?
- Chapter Exercise

2. Understanding HTTP, REST, and APIs

- Understanding Web API Protocols, Practices, and Styles
- Managing Files with Git
- What's Next?

[Chapter Exercise](#)

Part II. The Design Phase

3. Modeling APIs

[Understanding Norman's Action Lifecycle](#)

[Modeling Our Onboarding API Lifecycle](#)

[Managing Your Project with npm](#)

[What's Next?](#)

[Chapter Exercise](#)

4. Designing APIs

[The Power of Design](#)

[The API Design Method](#)

[Identifying Your API Descriptors](#)

[Creating Your Sequence Diagram](#)

[What's Next?](#)

[Chapter Exercise](#)

5. Describing APIs

[Learning the Role of Description Formats](#)

[Describing Your API with ALPS](#)

[Updating Your API Project](#)

[What's Next?](#)

[Chapter Exercise](#)

Part III. The Build Phase

6. Sketching APIs

[Learning from Frank Gehry's Sketches](#)

[API Sketching Example](#)

[The Advantages of Sketching](#)

[Sketching APIs with Apiary Blueprint](#)

[API Sketching Tips and Tricks](#)

[What's Next?](#)

[Chapter Exercise](#)

7. Prototyping APIs

[What Is an API Prototype?](#)

[API Prototyping with OpenAPI](#)

[Translating Your API Design into HTTP](#)

[Creating Your OpenAPI Document with SwaggerHub](#)

[Saving and Exporting Your API](#)

[Mocking Your API](#)

[Generating Your API Documentation](#)

[What's Next?](#)

[Chapter Exercise](#)

8. Building APIs

[Defining the API Build Process](#)

[Relying on a Repeatable Process](#)

[Coding APIs with NodeJS and DARRT](#)

[Putting It All Together](#)

[What's Next?](#)

[Chapter Exercise](#)

Part IV. The Release Phase

9. Testing APIs

[The Goals of API Testing](#)
[Testing with SRTs](#)
[Using Postman for API Testing](#)
[Running Tests Locally with Newman](#)
[What's Next?](#)
[Chapter Exercise](#)

10. Securing APIs

[Understanding Security Basics](#)
[Implementing API Security with Auth0](#)
[Supporting Machine-to-Machine Security](#)
[What's Next](#)
[Chapter Exercise](#)

11. Deploying APIs

[The Basics of Deployment Pipelines](#)
[The Role of DevOps](#)
[Deploying with Heroku](#)
[What's Next?](#)
[Chapter Exercise](#)

12. Modifying APIs

[Going Beyond Versioning](#)
[The Three Rules for Safely Modifying APIs](#)

[The Recommended Pattern for Testing for API Changes](#)

[The Process for Safely Deploying APIs](#)

[Shutting Down an Existing API](#)

[What's Next?](#)

13. Some Parting Thoughts

[Getting Started](#)

[The Design Phase](#)

[The Build Phase](#)

[The Release Phase](#)

[What's Next?](#)

Part V. Appendixes

A1. Installation Hints

[curl](#)

[Git](#)

[GitHub and SSH](#)

[NodeJS and npm](#)

[Postman](#)

[Newman](#)

[Heroku Client](#)

[Installing the Local Utilities](#)

A2. Exercise Solutions

[Where's the Code?](#)

[Solution for Chapter 1: Getting Started with API First](#)

[Solution for Chapter 2: Understanding HTTP, REST, and APIs](#)

[Solution for Chapter 3: Modeling APIs](#)
[Solution for Chapter 4: Designing APIs](#)
[Solution for Chapter 5: Describing APIs](#)
[Solution for Chapter 6: Sketching APIs](#)
[Solution for Chapter 7: Prototyping APIs](#)
[Solution for Chapter 8: Building APIs](#)
[Solution for Chapter 9: Testing APIs](#)
[Solution for Chapter 10: Securing APIs](#)
[Solution for Chapter 11: Deploying APIs](#)

A3. API Project Assets Checklist

[Using the API Project Assets Checklist](#)
[Notes on the Assets Checklist](#)
[Your API Project Assets Checklist](#)

Early praise for *Design and Build Great Web APIs*

Written in a lively and entertaining style, *Design and Build Great Web APIs* is a truly comprehensive journey through the development of high-quality APIs: it's accessible for those who are new to the field while also providing food for thought for the more experienced. The worked examples, alongside chapter-by-chapter exercises, provide excellent hands-on learning, which is unusual for books in the API field.

→ Fiona McLoughlin

API Design and Governance, HSBC Global Services

Mike Amundsen's latest book is a hands-on text that makes building web APIs accessible to new developers while reminding more experienced ones of the simple concepts that drive good API design. I'm recommending this book to every developer or technical product manager who joins my team.

→ Shelby Switzer

Digital Service Expert

This is a practical guide for anyone developing APIs. My favorite part about this book is that the design principles and approaches are universally applicable, no matter what API style or protocol you choose.

→ Matt McLarty

Global Leader of API Strategy, MuleSoft, a Salesforce Company

Design and Build Great Web APIs is a comprehensive resource that will teach you the methods to create and represent your APIs and transfer them into production with modern techniques and clear definitions.

→ Michael Nygard

Author of *Release It! Design and Deploy Production-Ready Software, Second Edition*

If you are new to API development, Mike will propel you forward in understanding and practical knowledge that those of us in the API space spent years acquiring through trial and error. If you are an API veteran, you'll learn new things and gain broader understanding.

→ Mark W. Foster

Principal Software Engineer

This book has everything you need to develop and maintain web APIs. It's valuable for anyone getting started, but it also has a lot of information that's helpful for veteran developers.

→ Nick McGinness

Software Engineer

A clear and compassionate guide to both technical and business aspects of API design. You're in good hands.

→ Leonard Richardson

Co-author of *RESTful Web APIs*

Design and Build Great Web APIs will help you do exactly that. Mike Amundsen shares his knowledge of designing APIs in an approachable, logical, and manageable way that anyone involved in API design and development can benefit from.

→ Ashish Dixit

Senior Software Engineer, Fastly, Inc.

Acknowledgments

First, I owe a debt of thanks to the many people who were involved in my API workshops and training sessions over the last several years. Many of the ideas and some of the tooling you'll find in these pages is the direct result of the generous and valuable feedback provided by attendees from all walks of life.

I also want to thank all the conference organizers, reviewers, and event staff for the opportunity to share my thoughts and ideas with their skilled and discerning audiences. Every encounter has taught me important lessons and helped shape the content of this book.

And my gratitude extends to all the corporations and foundations that invited me into their organizations and allowed me to ask questions, probe into their API programs, and learn from their accumulated knowledge from teams around the world.

Of course, this book could not have come about without the help and support of the Pragmatic Bookshelf and its dedicated and professional staff. I thank Brian MacDonald, who helped me find a home for a book project I'd been pitching for quite a while. And thanks to the whole team at the Bookshelf for their patience and assistance as I worked to bring together all I've learned into a single place.

Special thanks goes to my fantastic editor, Katharine Dvorak. Katie helped me learn the "Pragmatic way" and guided me through all the details of

pulling a book together. But more than that, Katie was an active contributor to the tone and style of the book. I truly couldn't have done it without her.

Another person I'd like to call out is Dana Amundsen, who provided the wonderful illustrations for the book. Her ability to turn my vague ideas and sloppy sketches into clear, informative, and engaging images is an important addition to the project.

I want to also acknowledge the work of a team of skilled and relentless reviewers for all the time they spent reading and rereading drafts of my text, adding corrections, suggestions, and observations along the way. Thank you to Nick Capito, Ashish Dixit, Mark Foster, Rod Hilton, Ivan S. Kirkpatrick, Niranjan Marathe, Nick McGinness, Sean Miller, Ronnie Mitra, and Tanuj Shroff. If this book is helpful, it's most likely because of the work of this team. And the parts that fall short are likely because I wasn't able to make all the changes they suggested.

I also want to extend thanks to all the tool and Software-as-a-Service vendors that are mentioned in the book. In almost every case, I reached out to representatives to ask questions and solicit advice on how best to use their products or services. And everyone I contacted was helpful and encouraging. I especially want to thank Steve Hanov of WebSequenceDiagrams.com, who gave me permission to build a tool around his site's API and share it with you in this book.

Finally, I dedicate this book to all those API designers and developers out there who are the backbone and lifeblood of this ever-growing and constantly changing world of APIs for the web. I hope this book proves helpful, and I'm looking forward to seeing all the new great web APIs that are ahead for all of us.

Mike Amundsen
Kentucky, USA, August 2020

Copyright © 2020, The Pragmatic Bookshelf.

Preface

Welcome to the world of API development! This book is designed to help you go from start to finish on the path to designing, building, and deploying great application programming interfaces (APIs) for the web. Along the way you'll learn a handful of important practices and principles that will help you design and build APIs that are robust, reliable, and resilient. You'll also acquire skills in a wide range of developer tools, including tools for design, documentation, building, testing, security, and deployment. By the time you work through this book, you should have enough experience to tackle just about any challenge that comes your way.

Your API Journey

The content in this book follows a learning path I've developed over the years to help developers from varying backgrounds get a foothold and gain experience in the API space.

Who Should Read This Book

This book is aimed at software developers and designers looking to quickly get up to speed on how to design and build web APIs. Since the content is based on training material I've used over the years, this book could be handy should you put together your own API training course for your team or company.

It can also be used as a general reference guide on basic topics like API design methods, the API build process, testing, and other steps in API development. For this reason, the book is also valuable to senior software architects and API product managers looking to provide common guidance to all their teams.

How This Book Is Organized

The book is organized into five parts: Getting Started, The Design Phase, The Build Phase, The Release Phase, and Appendixes. Each part marks a key set of skills in the journey from thinking about APIs to actually getting them up and running in production. The following is a quick summary of each of the five parts of the book.

Part I: Getting Started

The first part helps set the foundation for the world of APIs on the web. We first explore what it means to adopt an “API-First” approach to designing and building APIs. We then get some background on the existing services we’ll be working with and our plan for implementing an API requested by a fictitious customer, BigCo, Inc.

In the second chapter, we spend time exploring the basics of the protocols and practices commonly used to create APIs on the web, including the HTTP protocol and the REST style for creating services that run on HTTP.

Part II: The Design Phase

In Part II, we focus on the process of modeling, designing, and describing our API. Along the way we look at tools to help us model and visualize the API and ones that help us create a machine-readable description that can be used to produce our final API.

Part III: The Build Phase

The build phase of API development is all about converting our well-designed and well-described API into working code. In this part we take a three-step approach by sketching, prototyping, and finally building the API itself. We use the Apiary Blueprint format for sketches, the Open API Specification (aka Swagger) to create the exact definition, and then NodeJS

and a custom framework called DARRT to turn the Swagger spec into working code.

Part IV: The Release Phase

In the fourth part of the book, we look at how to test, secure, and deploy our completed API into the cloud using the local Postman platform for testing and the Auth0 cloud service for authentication and access control, and then we release the finished API onto the Heroku public cloud.

Once we know that's all working as expected, we look at how to introduce minor changes to our production API without breaking any existing API consumers.

Part V: Appendixes

At the end of the book is a small collection of appendixes that you can use as a reference while you're reading through the book. The first appendix covers what programs you may want to install as you follow along in the chapters. The next appendix offers solutions to the chapter exercises. The final appendix lists all the API project assets you should be creating and managing as you go through all the stages of API design, build, and release.

In Reality

While the book takes a very linear approach to all this (driving from modeling to building to releasing the API), in reality API development involves lots of iteration and bouncing back and forth between skills along the way. In most production work, you end up working in a series of "loops" that allow you to model, then design, then go back and fix the model a bit, and then design some more, and so on until you finally get to the point where your API is ready to be released to the cloud.

While this looping is the right way to do the work, it isn't a good way to write a book about it. For that reason, we'll stick to our straight-line

approach from start to finish, with lots of reminders along the way about iteration and looping as a key element in designing and building great APIs.

What's Covered (And What's Not)

The goal of this book is to give you comprehensive “front-to-back” coverage of what’s needed to get a new API up and running in production. To keep things focused, we’ll be covering the skills needed to create “RESTful” APIs. These are HTTP-based APIs that rely on the common request-response style used by the vast majority of APIs on the web today. There are other important API styles (such as gRPC and graphQL) that we won’t have time to cover in a single book. The good news is that most other API styles rely on the HTTP-based web API approach we’ll cover here. Getting a solid understanding of this RESTful style will come in handy if you want to explore other styles in the future.

The topic of APIs for the web is quite large, and we won’t be able to cover it all. This book focuses on a series of steps centered around the external interfaces (the APIs) used to access back-end services. However, I won’t be getting into how to build and deploy back-end services (sometimes called microservices) themselves. If you’re interested in that topic, I recommend you check out Chris Richardson’s *Microservices Patterns*^[1] or Sam Newman’s *Building Microservices*.^[2]

I also won’t be covering the details of writing the client applications that *consume* web APIs. If you’re looking for a source on that, I suggest you check out another of my books, *RESTful Web Clients*.^[3]

Finally, I won’t be covering topics related to the broader topic of API management. Things like API gateways, portals, monitoring, and overall governance are covered well in books like *Continuous API Management*,^[4] which I co-wrote with Mehdi Medjaoui, Erik Wilde, and Ronnie Mitra.

About the BigCo, Inc., Sample Project

In this book, we'll be helping a fictional company (BigCo, Inc.) modernize its process of bringing on new customers by designing, building, and deploying a brand-new API called the "Onboarding API." The company's onboarding process is still handled mostly by humans filling in forms and then adding data to several stand-alone applications running in the company's network. Our job will be to learn how BigCo currently handles this customer onboarding process, convert that into an API design, and then —using existing services already in place—implement a single onboarding API that handles all the steps currently handled by BigCo staff.

Along the way, we'll look at the basics of API design and how to produce and document our design in a consistent way that will be understood by both BigCo onboarding staff and developers. We'll also look at how to use a simple three-stage process (sketch, prototype, and build) for converting an API design into running code. And once the code is available, we'll work through the release process of testing, securing, and finally deploying the working onboarding API. Finally, we'll look at what it takes to make updates to an already-released API without breaking any of the apps currently using it.

About BigCo, Inc.

As a bit of background, here's the "origin story" for the company behind all the examples in this book.

The Bayesian International Group of Companies (BigCo) is an organization with a long history of both prosperity and generosity. Founded in 1827 in Scotland, BigCo's motto is "The principle end of the company is the happiness of its customers and employees." Originally focused on demography and finance and based on the work of Richard Price, by the 1900s BigCo was a manufacturer of important turn-of-the-century technologies, including mobile x-ray machines and hydrophones, by working with inventors Frederick Jones and Reginald Fessenden, respectively. BigCo was also one of the few non-U.S. contractors attached

to the Manhattan project, providing parts for the first nuclear reactors designed by Dr. Leona Woods.

During the postwar boom, BigCo opened offices in the United States and played an instrumental role in helping to establish fields of study in the modern computer era, including predictive algorithms and robotics controls. Currently, BigCo products focus on “inverse probability” problems and how they can be used to help improve decisions and policy-making in both the private and public sectors.

While BigCo is an imaginary company, all the people mentioned in its description (Price, Jones, Fessenden, and Woods) are real historical figures. If BigCo *did* exist, it’s possible that these innovators would have been part of its history, too.

Online Resources

To follow along with the project throughout the book, download the source code from the book’s web page at pragprog.com.^[5] And if you find any errata, there’s also a place to let us know about it.^[6]

But before we dive into designing and programming the API, let’s spend a little “getting started” time on the concepts of API First and refreshing our knowledge of the HTTP protocol, web concepts, and the API style known as REST (REpresentational State Transfer).

Let’s get started!

Footnotes

[1] <https://microservices.io/book>

[2] <https://learning.oreilly.com/library/view/building-microservices/9781491950340>

[3] <https://learning.oreilly.com/library/view/restful-web-clients/9781491921890>

[4] <https://learning.oreilly.com/library/view/continuous-api-management/9781492043546>

[5] <https://pragprog.com/book/maapis>

[6] <https://devtalk.com/books/design-and-build-great-web-apis>

Part 1

Getting Started

Chapter 1

Getting Started with API First

This first chapter provides background information that'll come in handy throughout the rest of the book. We'll start by talking about an important principle—API First—and then move on to a quick review of a handful of existing services at our fictitious company, BigCo, Inc. We'll use curl, the first in a series of command-line tools, as part of our API toolkit for exploring BigCo's APIs.

That's a lot to cover! So let's start by discussing the power and advantage of the principle known as API First.

Adopting the API-First Principle

The concept of API First has a long and varied history. Although the phrase has been in use for almost ten years (as of this writing), it's become particularly popular over the last several years. For example, the website ProgrammableWeb ran a series called “Understanding API First Design” in 2016, [\[7\]](#) and I still hear people mention this series when they talk about API First. A website created in 2013 was even dedicated to API First, [\[8\]](#) although it hasn’t been maintained over the years. And a simple web search on the term “API First” brings up lots of related articles from all corners of the IT world targeted to a wide audience, from developers to CEOs.

I first heard about API First in 2009 in a blog post called “API First Design,” by Kas Thomas. [\[9\]](#) He described the idea this way: “API-first design means identifying and/or defining key actors and personas, determining what those actors and personas expect to be able to do with APIs.” I think this definition is important because it points out that creating APIs is about understanding who’s using the API and what they want to *do* with it. Frankly, it’s pretty hard to go wrong with this point of view in mind.

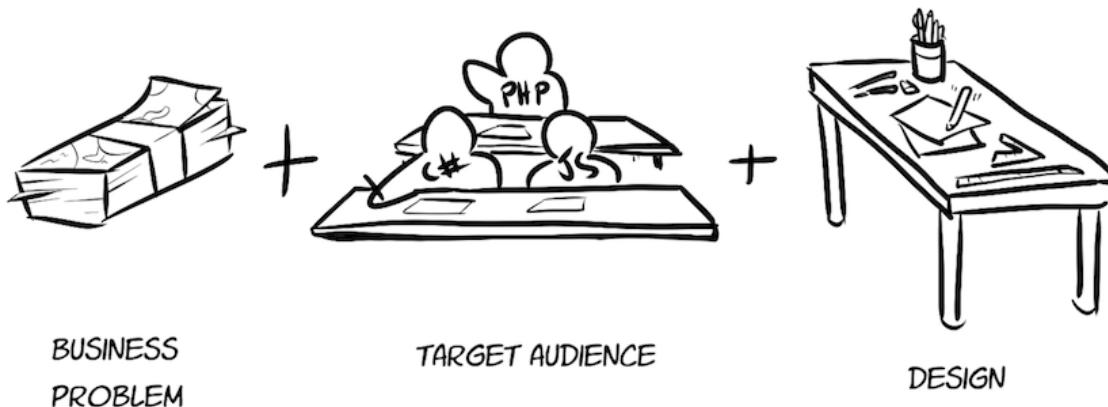
Another key element wrapped up in the API First meme is the notion of designing the API *before* you get excited about all the implementation details, such as programming languages, protocols, URLs, and message formats. Well-known API advocate Kin Lane puts it directly in his blog post “What Is an API First Strategy?”[\[10\]](#) when he states, “Before you build your website, web, mobile or single page application you develop an API first.”

So, two key elements of the API-First principle are (1) focus on who’s using the API and what they want to accomplish and (2) design the API *first*, before you start to think about technology details. A third important element in all this pertains to the business side of things. APIs exist to solve business problems. Let’s explore these three elements a bit more to get a

solid footing on the API-First principle, starting with how APIs are used to solve business problems.

Using APIs to Solve Business Problems

In the real world, APIs exist to solve a business problem, like improving service quality, increasing sales, reducing costs, and so forth. When you’re working on an API—whether you’re tasked with designing it from scratch or simply working to implement a design someone else has provided—it helps to keep the business goals in mind.



Typically, APIs are needed to solve problems in three basic categories:

- Reducing the time/cost of getting something done
- Increasing the ease or likelihood of getting something done
- Solving a problem no one else has solved

To solve the first problem (how to reduce time/cost), APIs are often used behind the scenes—between teams in the same company, for example—in ways that are meant to automate some process or bridge a gap between one team (or software component) and another. In these cases, the design needs to be specific enough to solve the problem and flexible enough to be modified in the future without causing more pain for the teams involved. In other words, you’re providing a short-term fix with long-term support in mind. That mindset changes the way you design and implement the solution and is the basis of the API we’ll build in this book.

To solve the second problem (how to increase the ease), APIs are usually used to improve throughput or increase sales, user sign-ups, and so on. These APIs are aimed at making developers more productive as well as making it easier to solve interaction problems for humans. That often means APIs that support user experiences (UX) and are geared toward supporting mobile or web user interfaces (UIs). Supporting a live UI is a tough job. UI designers often have very specific demands on how the interfaces “look and feel” and need APIs that make that quick and easy. When you are supporting APIs for UX designers, the most important thing is getting them the data they need at the right moment. Often these APIs are quite “chatty” and have lots of variations and nuances. We won’t spend too much time on UX-style APIs in this book, but I’ll cover some considerations later in the book when we deploy and then modify an existing API.

Finally, you might be asked to build APIs that provide a solution to a problem no one has yet solved. These can be fun APIs to work on, but they have their own challenges. Most of the time, all the functional elements needed to solve the problem already exist within the company (for example, existing services, third-party products, remote web APIs, and so on). However, no one has figured out the best way to *mix* all these disparate components into a single solution. Depending on your company’s IT ecosystem, this might be the most common work you need to do. And it might be the most frustrating. A lot of your time might be spent translating output from one service into something another service can understand. And if one of the existing services changes their output, your solution might fail. Designing and implementing this kind of *composed* API solution takes lots of attention to details and planning for possible failure. I’ll spend some time in the second part of the book talking about how to deal with this kind of API challenge.

Designing APIs for People

As mentioned at the start of this chapter, a key element of API First is understanding the people using the API and the reason they need it. It’s

important to have a clear sense of the expected skills and interests of your API *consumer* (the person using the API) as well as the actual work they’re trying to do (such as automate a process, support a UX, solve unique problems, and so on). Getting a good sense of these aspects of the API will help you spend time focusing on the right things when it comes to translating their needs into your design.

For example, are most of your API consumers working within your company? If they are, they likely already understand your company’s business model, internal acronyms, and other “inside” information. Do most of them work in IT? If yes, you can assume they have some programming skills and understand things like HTTP, JavaScript, and PHP. Or maybe they work in marketing, human resources, or sales. They might not have extensive developer training or have much time to spend testing and debugging APIs. In this case, your API design needs to be straightforward and easy to use. You’ll design and document the API differently depending on who you expect to use it.

If the target audience for your API is someone outside the company (such as a partner company or some third-party mobile developer), your approach will need to be different. You can’t assume the developers understand (or even *care about*) your business model. In fact, you probably don’t want to share too much “inside” information with them—they might be a competitor! In this case, you’ll design and document an API that solves a very narrow set of problems and makes it hard for API consumers to mistakenly or maliciously use the API in a way that does harm to your company, data, or users.

Knowing the target audience of your API and the kinds of problems those people want to solve is critical when it comes to creating APIs that meet the needs of your audience without jeopardizing the safety and security of your own company.

Understanding APIs and Design

Finally, the way you design the API itself depends on several factors. The API-First principle tells us to design APIs without regard to the technology you might use to build it. That means we focus on the interactions in the API experience as well as the data behind it. In fact, in well-designed APIs, the interaction is paramount. It's important to first determine what tasks need to be done. Only then do you spend time figuring out just what data you need to pass back and forth for each interaction in order to accomplish the required tasks. I'll spend time talking about a design strategy in depth in Part II.

A really good process for supporting API-First design is to gather and then document the list of actions (for example, `FindCustomerRecord`, `UpdateCustomerHistory`, `RemoveCustomerHold`). Once you have a complete list, go back and start to list all the data properties (`customerId`, `givenName`, `familyName`, `email`, `shoeSize`, and so on) you need to share for each interaction. However, collecting and detailing the interactions is just the start. The real design work begins *after* you gather all this information.

In some cases, your API consumers will have all the data required ahead of time (`customerId`, `givenName`, `familyName`, `email`, and so on) and just need to plug values into your API and execute a single call. In other cases, only part of the required data is available (such as `email`). When your API consumers don't have all the data, they need helper APIs to fill in the blanks. For example, if your API consumer needs to retrieve a customer record using a `customerId` but only has the customer's `email` address, you'll need to design an API that accepts an email address and returns one or more customer records to choose from. Getting an understanding of the circumstances your API consumers are working with will help you design an API that allows them to solve their problems relatively quickly, easily, and—above all—safely.

With API First in mind, let's now take a look at the existing services we'll need to work with as we work to design and build an API for our client—

BigCo, Inc.

Exploring APIs with curl

In this book we'll be helping a fictional company (BigCo, Inc.) design, build, and deploy a brand-new API called the Onboarding API. This API will be used to streamline the onboarding of new customers (called [companies](#) at BigCo). True to the API-First design ethos, we'll only be building the API and not spending time writing the *services* behind the API. The good news is that the services we need to power our Onboarding API already exist at BigCo; we just need to take some time to get familiar with them and then we'll be ready to start the work of designing, building, and releasing our API into the wild.

Like any major company, BigCo has lots of services up and running at the company. We'll only be focusing on three of them for this book:

- Activity
- Account
- Company

These three services are all fully functional, stand-alone services that do the jobs you would expect. The Account and Company services handle the details of managing partner companies and their associated account records. The Activity service handles the details of scheduling and tracking any activity (such as emails, phone calls, office visits, and so on) related to the company and account records.

Looking ahead, what we'll be tasked to do is design and build a single workflow process that handles the customer onboarding experience. To do that, we'll need to blend a bit of all three of the back-end services available to us.

For that reason, it's important that we get to know each of these back-end services well enough to recognize the available operations each of them

supports. We'll also need to eventually blend these three services in order to create the onboarding experience our manager at BigCo, Inc., is requesting. For now, let's take a closer look at the three services and learn how they work.

About curl



The tool we'll use to explore these APIs is curl, the command-line tool for "transferring data with URLs" (see <https://curl.haxx.se>). If you don't already have this tool installed, refer to Appendix 1, [Installation Hints](#), for details on how to download and install it. I'll just cover the basics of curl here. You can learn more about the power of curl in a free online book called *Everything curl*, which is available at <https://curl.haxx.se/book.html>.

The Activity Service

The Activity service is a very simple component that supports scheduling contacts between BigCo and partner companies and their accounts. BigCo commonly sends welcome-type emails, schedules follow-up phone calls, and has other points of contact. For this book, we'll focus on scheduling emails for new accounts and companies.

The following is an example of a single `activity` record, which has just a few fields: `id`, `activityType`, `companyId`, `accountId`, `dateScheduled`, `notes`, `status`, `dateCreated`, and `dateUpdated`:

```
{  
  id: "2dra7wqcyup",  
  activityType: "email",  
  companyId: "1or2ab05imf",  
  accountId: "",  
  dateScheduled: "",  
  notes: "",  
  status: "pending",
```

```
    dateCreated: "2019-02-25T08:22:15.648Z",
    dateUpdated: "2019-02-25T08:22:15.648Z",
}
```

Not all these fields are always present. For example, the `accountId` field will not appear for activities dealing with the `company` record. But at least one of them will appear in each record. And some of the fields (`id`, `dateCreated`, and `dateUpdated`) are supplied by the service itself, not set by any client applications. Also, the `activityType` field has a fixed set of valid action values. These values are listed in the human-readable documentation available in this book's code download files at pragprog.com.^[11] Finally, the `notes` field will be present only when the client has some comment or follow-up information about the activity.

This is just a single example record from the Activity service. You can see an actual live instance of that service running by entering the following at the command line:

```
curl https://activity-atk.herokuapp.com/list
```

Your response should look like this:

```
{
  activity: [
    {
      id: "2dra7wqcyup",
      activityType: "email",
      companyId: "1or2ab05imf",
      accountId: "",
      dataScheduled: "",
      notes: "Send welcome email to new partner",
      status: "pending",
      dateCreated: "2019-02-25T08:22:15.648Z",
      dateUpdated: "2019-03-25T018:01:33.000Z",
    },
    {
      id: "m0mvoh0loj",
      activityType: "email",
      companyId: "1or2ab05imf",
    }
  ]
}
```

```
        accountId: "21r1aeuj87e",
        dataScheduled: "2019-09-01",
        notes: "send reminder for a follow up visit",
        status: "pending",
        dateCreated: "2019-03-25T018:01:33.000Z",
        dateUpdated: "2019-03-25T018:01:33.000Z",
    }
],
}
```

You can see from this listing that someone created an activity record to send a welcome email for a new partner **company**, modified it, and then created a related activity record to send a reminder email for that company's new **account** record. As you might imagine, a system with many **company** and **account** records will also have lots of **activity** records.

What you can't tell from this listing is that you have several possible actions to take with the Activity service. These are covered in the human-readable documentation, but here's a quick list:

- Create an activity record.
- Get a list of activity records.
- Get a single activity record by **id**.
- Filter activity records by **accountId**, **companyId**, and/or **activityType**.

All of the details on how to do these things are also in the documentation. That includes the HTTP methods (**GET** or **POST**), the URLs (<https://activity-atk.herokuapp.com/2dra7wqcyup>), and the shape of the bodies returned (like in the previous example).

This practice of keeping all the operational details out of the message and only including them in the documentation is a common pattern for JSON-based APIs. It keeps the API responses cleaner and a bit smaller than if the URLs, methods, and so forth were included in the response message. In fact, many of these JSON-based APIs rely on a pattern known as **CRUD** (create-read-update-delete), which assumes the possible actions a developer

can invoke within the API. I talk more about CRUD in Part II of the book, where we get into the implementation details.

When we get around to incorporating the Activity service into our own Onboarding API, we'll need to read the documentation and then code in the details for adding records and, possibly, for filtering them. For now, it's enough to know what's possible and keep that in mind for the future.

Now, let's look at another service we'll be using in our onboarding project: the Account service.

The Account Service

The Account service handles the creation of specific account records for each partner company at BigCo. For example, a company might have an account with the marketing division for purchasing marketing services, and that same company may have an account with the home-goods division for purchasing wholesale home goods for resale. Each company has several possible accounts, and while companies *can* exist without having any assigned accounts, it's common for each company to have at least one active account record.

Account records look like this:

```
{  
  id: "1c0x0mzp83t",  
  division: "DryGoods",  
  companyId: "1or2ab05imf",  
  spendingLimit: "100000",  
  discountPercentage: "2.5",  
  status: "pending",  
  dateCreated: "2019-02-25T08:12:00.579Z",  
  dateUpdated: "2019-02-25T08:12:00.579Z",  
}
```

By now, most of this probably looks familiar. One of the interesting fields in this account record is the **status** field, which indicates the current state of

the account record. The `status` field (like the `division` field) has a fixed set of possible values; these values, along with other details about the Account service, are covered in the documentation available in the book's code download files. In this example, the account is set to `pending`, which means it's awaiting approval.

A call to a live instance of the Account service shows some interesting additional elements worth reviewing. For example, using the curl command-line tool, you can make the following request:

```
curl https://account-atk.herokuapp.com/list
```

...and the live service should return the following when you ask for the same account record shown earlier:

```
{
  "links": [
    {"rel": "list", "href": "/account/list"},
    {"rel": "add", "href": "/account/list"},
    {"rel": "home", "href": "/account/"}
  ],
  "account": [
    {
      "id": "1c0x0mzp83t",
      "division": "DryGoods",
      "companyId": "1or2ab05imf",
      "spendingLimit": "",
      "discountPercentage": "",
      "status": "active",
      "dateCreated": "2019-02-25T08:12:00.579Z",
      "dateUpdated": "2019-02-25T08:12:00.579Z",
      "links": [
        {"rel": "read", "href": "/account/1c0x0mzp83t"},
        {"rel": "update", "href": "/account/1c0x0mzp83t"},
        {"rel": "delete", "href": "/account/1c0x0mzp83t"},
        {"rel": "update-limit", "href": "/account/1c0x0mzp83t"}
        {"rel": "update-status", "href": "/account/1c0x0mzp83t"}
      ]
    },
    ...
  ]
},
```

```

{
  "id": "6x7hpmtxbss",
  "division": "Pharmacy",
  "companyId": "1or2ab05imf",
  "spendingLimit": "",
  "discountPercentage": "",
  "status": "active",
  "dateCreated": "2019-02-25T08:12:36.747Z",
  "dateUpdated": "2019-02-25T08:12:36.747Z",
  "links": [
    {"rel": "read", "href": "/account/6x7hpmtxbss"},  

    {"rel": "update", "href": "/account/6x7hpmtxbss"},  

    {"rel": "delete", "href": "/account/6x7hpmtxbss"},  

    {"rel": "update-limit", "href": "/account/6x7hpmtxbss"}  

    {"rel": "update-status", "href": "/account/6x7hpmtxbss"}
  ]
}
]
}

```

Along with the expected `account` record, we also have an array of link elements (called `links`). These link elements have two values: `rel` and `href`. The `rel` property indicates the name of the action that's supported at the URL, and the `href` property indicates the actual URL to use when executing the named action. This is more information in the response than we saw with the Activity service. It makes it easier for developers to know what's possible without having to read the written documentation. However, some important things are missing from this response. For example, we can't be sure which HTTP method to use when executing the actions. We also don't know what data we're supposed to pass when performing actions like `add` or `update`. For that information, we still need to rely on the human-readable documentation.

This style of API response is used by Amazon Web Services (AWS) and other companies. I'll talk more about how to deal with this JSON-link style API in Part II.

Now let's look at one more API we'll need when we implement our Onboarding API: the Company service.

The Company Service

The Company service at BigCo handles all the basics of managing partner company records for the organization. It supports creating, reading, updating, and deleting company records, along with supporting search for finding companies in the system. Let's take a look at a live instance of the Company service to see what it looks like. To do that we'll use the curl command-line tool. Type the following at the operating system prompt and press return:

```
curl https://company-atk.herokuapp.com/
```

You should get a response that looks like this:

```
{
  "home" :
  {
    "metadata" :
    [
      {"name" : "title", "value" : "BigCo Company Records"},  

      {"name" : "author", "value" : "Mike Amundsen"},  

      {"name" : "release", "value" : "1.0.0"}
    ],
    "links" :
    [
      {"id" : "self", "href" : "http://company-atk.herokuapp.com/"},
      {"id" : "home", "href" : "http://company-atk.herokuapp.com/"},
      {"id" : "list", "href" : "http://company-atk.herokuapp.com/list/"}
    ],
    "items" : []
  }
}
```

This response shows a single root object (`home`) with three arrays: `metadata`, `actions`, and `item`. All the APIs we'll be working with will have this basic structure. Structured messages are the key to stable, scalable APIs. You may

not see this particular structure very often (it's one I adopted several years ago), but all good APIs have a predictable structure to make it easy for developers to understand in order to write client code. I'll talk more about message structure in Part II.

The other important thing to notice from this initial response is that the *root* or *home* resource doesn't contain any data records (see the empty `items` array). It does, however, have links—in this case a link that points to this *home* resource and a link that points to the *list* resource. If you want to view the company data, you need to follow the `list` link. You might think the resource we just called should just return the company data and not ask devs (or machines) to navigate to another link to get the data. But offering a root or home resource like this (one that has just links) is a good practice. It leaves room for future expansion of the service (for example, adding a specialized `/company-search/` service) and for supporting multiple incompatible versions of the Company service (for example, `/company/v2/` and `/company/v3`). And, finally, if developers don't want to deal with this root resource, they can just hard-code the URL they're interested in directly into their client app. Implementing a root resource gives the service developer options without causing much trouble for the service consumer.

An even more interesting response from the Company service follows. This time, since the response to this command is rather long, you can use curl to save the service response to disk and then open that file up in your favorite editor. That call looks like this:

```
curl https://company-atk.herokuapp.com/258ie4t68jv > company-read.json
```

The Company service should write a disk file (`company-read.json`) that looks like this when you open it up in your editor:

```
{
  "company": {
    "metadata": [
      {
        "id": "54321",
        "name": "Acme Corp."
      }
    ]
  }
}
```

```
        "value": "BigCo Company Records",
        "name": "title"
    },
    {
        "value": "Mike Amundsen",
        "name": "author"
    },
    {
        "value": "1.0.0",
        "name": "release"
    }
],
"links": [
    {
        "properties": [],
        "method": "GET",
        "title": "Self",
        "tags": "collection company self home list item",
        "rel": "self collection company",
        "href": "http://company-atk.herokuapp.com/72g8euppcm",
        "name": "self",
        "id": "self"
    },
    {
        "properties": [],
        "method": "GET",
        "title": "Home",
        "tags": "collection company home list item",
        "rel": "collection company",
        "href": "http://company-atk.herokuapp.com/",
        "name": "home",
        "id": "home"
    },
    {
        "properties": [],
        "method": "GET",
        "title": "List",
        "tags": "collection company home list item",
        "rel": "collection company",
        "href": "http://company-atk.herokuapp.com/list/",
        "name": "list",
        "id": "list"
    }
],
```

```
{
  "properties": [
    {
      "value": "",
      "name": "status"
    },
    {
      "value": "",
      "name": "companyName"
    },
    {
      "value": "",
      "name": "stateProvince"
    },
    {
      "value": "",
      "name": "country"
    }
  ],
  "method": "GET",
  "title": "Search",
  "tags": "collection company filter list item",
  "rel": "collection company filter",
  "href": "http://company-atk.herokuapp.com/filter/",
  "name": "filter",
  "id": "filter"
}
],
"items": [
{
  "id": "72g8euppcm",
  "companyName": "mike-co",
  "streetAddress": "",
  "city": "",
  "stateProvince": "",
  "postalCode": "",
  "country": "",
  "dateUpdated": "2020-03-13T04:51:12.506Z",
  "dateCreated": "2020-03-13T04:51:12.506Z",
  "status": "pending",
  "email": "mike-co@gmail.com",
  "telephone": "",
  "links": [

```

```
{
  "properties": [],
  "method": "GET",
  "title": "Read",
  "rel": "item company read",
  "href": "http://company-atk.herokuapp.com/72g8euppcm",
  "name": "read",
  "id": "read_72g8euppcm"
},
{
  "properties": [
    {
      "value": "72g8euppcm",
      "name": "id"
    },
    {
      "value": "mike-co",
      "name": "companyName"
    },
    {
      "value": "mike-co@gmail.com",
      "name": "email"
    },
    {
      "value": "pending",
      "name": "status"
    },
    {
      "value": "",
      "name": "streetAddress"
    },
    {
      "value": "",
      "name": "city"
    },
    {
      "value": "",
      "name": "stateProvince"
    },
    {
      "value": "",
      "name": "postalCode"
    }
  ]
}
```

```
{
  "value": "",
  "name": "country"
},
{
  "value": "",
  "name": "telephone"
},
{
  "value": "mike-co@gmail.com",
  "name": "email"
},
],
"method": "PUT",
"title": "Edit",
"tags": "company list item",
"rel": "item edit-form company",
"href": "http://company-atk.herokuapp.com/72g8euppcm",
"name": "update",
"id": "update_72g8euppcm"
},
{
  "properties": [
    {
      "value": "pending",
      "name": "status"
    }
  ],
"method": "PATCH",
"title": "Status",
"tags": "company item list status",
"rel": "item company status",
"href": "http://company-atk.herokuapp.com/status/72g8euppcm",
"name": "status",
"id": "status_72g8euppcm"
},
{
  "properties": [],
"method": "DELETE",
"title": "Remove",
"rel": "item company remove",
"href": "http://company-atk.herokuapp.com/72g8euppcm",
"name": "remove",
"method": "DELETE"
}
```

```
        "id": "remove_72g8euppcm"
    }
]
}
]
}
}
```

Note that this response has quite a bit of information in it. And the one on disk has even more information than the one above—not just the single company record but a whole host of information about what actions are possible. And unlike the Account service we looked at earlier, this response includes not just the URL to use, but also the HTTP method and the data to pass for each action. This kind of information in API responses looks a lot like the way forms look in HTML responses. This JSON-forms style of API responses occurs in many APIs. Specific formats have even been designed to support this. Two popular examples are the Siren format^[12] and Collection+JSON.^[13] We'll dig into this in Part II when we use the Company service to implement our Onboarding API.

That wraps up our review of the existing services we'll be using throughout this book.

What's Next?

In this chapter, we covered the notion of *API First* as a mindset for creating and building APIs. The most important point in the API-First approach is that we should focus on what people want to *do* with our API. It's hard to go wrong when you keep in mind both the people using the API and the jobs they need to get done with your API.

You also saw how to use the curl command-line utility to make HTTP calls in order to inspect the APIs of other services. In this chapter, we studied three existing services for our fictitious company, BigCo, Inc.: Activity, Account, and Company. We noticed they each have a different API style (JSON-only, JSON-link, and JSON-form styles), and you learned how to use the curl tool to make requests to those APIs and store the requests on disk for later reference.

Now that you've gotten a start on using curl and reviewed the thinking behind API First, it's time to take a moment to explore the basics of the protocols and practices commonly used to create APIs on the web. That means looking into the HTTP protocol (that's the one we used with curl in this chapter) and the REST style for creating services that run on HTTP. Along the way you'll also learn another tool—Git—that we'll use throughout the book to manage all the files in our API projects.

Chapter Exercise

For this exercise, you'll extend the skills you learned in this chapter by curling all three of the services you learned about in this chapter (Company, Account, and Activity) and storing the results in our `services` folder. You'll then update your local repository and, finally, sync the remote GitHub repository with your local copy.

First, here are the URLs you can use to make API calls with curl to save in your `services` folder:

- <https://company-atk.herokuapp.com> (save the output as `company-home.json`)
- <https://company-atk.herokuapp.com/list> (save the output as `company-list.json`)
- <https://company-atk.herokuapp.com/2258ie4t68jv> (save the output as `company-record.json`)
- <https://account-atk.herokuapp.com/list> (save the output as `account-list.json`)
- <https://account-atk.herokuapp.com/1c0x0mzp83t> (save the output as `account-record.json`)
- <https://activity-atk.herokuapp.com/list> (save the output as `activity-list.json`)
- <https://activity-atk.herokuapp.com/2dra7wqcyup> (save the output as `activity-record.json`)

Use these URLs with the curl utility in the format `curl <URL> > <filename>` to save all seven responses to disk in the `services` subfolder of your project. We'll use this `services` folder as the start of our API project and will start managing that project with Git in the next chapter.

See Appendix 2 ([*Solution for Chapter 1: Getting Started with API First*](#)) for the solution.

Footnotes

- [7] <https://www.programmableweb.com/api-university/understanding-api-first-design>
- [8] <http://api-first.com>
- [9] <http://asserttrue.blogspot.com/2009/04/api-first-design.html>
- [10] <https://apievangelist.com/2014/08/11/what-is-an-api-first-strategy-adding-some-dimensions-to-this-new-question>
- [11] https://pragprog.com/titles/maapis/source_code
- [12] <https://github.com/kevinswiber/siren>
- [13] <https://github.com/collection-json/spec>

Copyright © 2020, The Pragmatic Bookshelf.

Chapter 2

Understanding HTTP, REST, and APIs

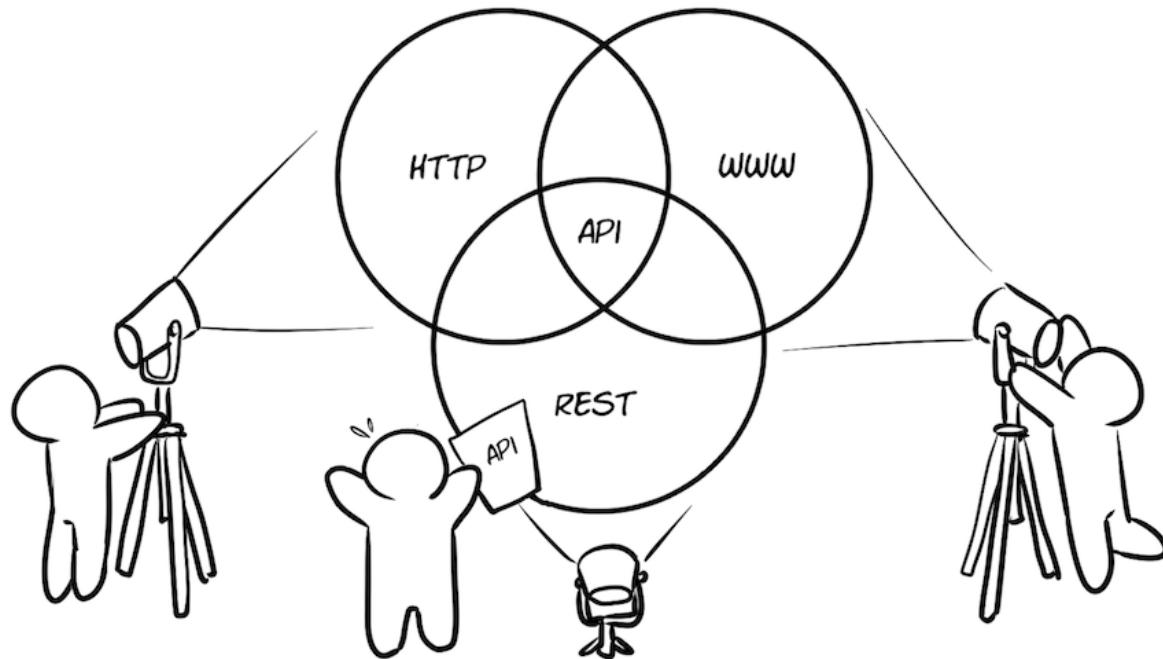
In the previous chapter, we explored the API-First principle and learned to use the curl utility to start making simple API calls to some of the services we'll be using throughout this book to put together our own API. In this chapter, I'm going to step back a bit and talk about some of the technical details behind those `curl` calls and how the common practices and patterns used to build applications on the web can help us when we design and build our own APIs.

We'll also spend time using another important command-line tool—Git—and a related website: <http://github.com>. We'll use these together to manage all the files and folders in our API project and to track and publish changes to the source code, design documents, and test results we'll be generating as we go through the API design and build process. I commonly design and build my APIs using lots of small tasks that build up over time into the final result. And I keep track of all those small tasks (and the changes they entail) using Git. A big advantage to doing this is that, if something goes wrong with one of my changes, it's almost always small enough that I can just use Git to reverse my last set of changes and get the system back to a stable, working version. Small changes are great for maintaining a safe, healthy API, and Git is the tool that makes this possible.

Understanding Web API Protocols, Practices, and Styles

Before jumping into the details of Git, let's review the open standards and common practices that have helped make the Internet, and the web in particular, a great place to build and publish APIs. For that, I'll cover three general topics:

- The HTTP protocol
- The common practices behind the success of the web
- The REST style for designing and building HTTP-based applications



HTTP is an open standard that's helped stabilize and power the web for about thirty years. HTTP isn't the only open protocol used to build APIs on the Internet, but it is the most popular one to date, and we'll use it for all the APIs in this book.

The web itself isn't actually a standard. It's more of a set of common patterns and practices initially established by Tim Berners-Lee when he first

started to implement early versions of HTTP and HTML in 1989. Our APIs will follow those principles too.

Finally, Roy T. Fielding's REST style for building applications over HTTP is even less than a standard or set of common practices. REST is really a *style*, or way of doing things, that Fielding documented to help software architects and programmers build robust, reliable, and scalable applications and APIs. We'll be relying on most of Fielding's guidance as we design and implement the APIs in this book.

The Basics of HTTP

Most APIs on the web rely on a standard protocol to make sure machines can talk to each other over the Internet. That protocol is the hypertext transfer protocol, or HTTP. The first public version of HTTP was released in 1991,^[14] and several more versions have been documented and released in the almost thirty years since. The HTTP protocol standard is currently managed by the Internet Engineering Task Force (IETF) and, as this book is being written, is at version 2.0,^[15] with a new version in process.

I won't spend a lot of time explaining HTTP here (you can check out the footnotes if you want to dig into the details). However, we do need to cover some of the basics of HTTP. Understanding how HTTP works and the elements of every HTTP message that's sent over the Internet can be very helpful when it comes to designing and building APIs that rely on this important protocol.

The elements I'll cover here are these:

- The HTTP message: HTTP defines a single text-message format for all requests and responses.
- HTTP methods: To ensure both sender and receiver understand each other, HTTP defines a set of method names that we'll need to deal with when we implement our APIs.

- Other HTTP features: A handful of other aspects of HTTP are worth mentioning in order to help everyone use HTTP effectively and efficiently.

HTTP Messages

The most important thing to know about the HTTP protocol is that it was designed entirely on the notion of passing *messages*, not objects or database query results. This message-orientation is one of the reasons HTTP (and HTML) became so popular and so pervasive so quickly. As long as you could write a client or service that can deal with HTTP messages, you could create an HTTP server and start serving up content, no matter what programming language, object model, or database you used.

So, what does an HTTP message look like? It turns out they're very simple. Every HTTP request or response message has just three parts:

- The start-line
- The header collection
- The message body

Here's what a real HTTP request and response look like:

```
POST /create-user HTTP/2
Host: www.example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 46

name=mike&email=mike@example.org&sms=123456789
```

The first line is the *start-line*, which contains three elements:

- HTTP method (**POST**)
- URL (**/create-user**)
- Protocol version (**HTTP/2**)

The next two lines are part of the *header collection*. In this example, it contains two values:

- The type of message body being sent ([Content-Type: application/x-www-form-urlencoded](#))
- The size of the message body ([Content-Length: 46](#))

The end of the header collection is indicated by a blank line, and then you see the *message body*. In this case, the body is a list of names and values that the server will use to create a new [user](#) record.

HTTP servers accept requests (like the one above) and return responses. Here's an example response a server might send after receiving the above request:

```
HTTP/2 201 Created
Date: Mon, 27 Jul 2020 12:28:53 GMT
Content-Length: 0
```

In this response, the start-line has three slightly different parts:

- The HTTP version ([HTTP/2](#))
- The status code ([201](#))
- The status message ([Created](#))

The next two lines contain two headers: the date the response was sent (in Greenwich Mean Time, or GMT) and the length of the message body. You'll see that the [Content-Length](#) is set to zero since there's no message body with this response. In HTTP, the message body is an optional element. Many times the request or response won't have a message-body element.

That's all there is to understanding HTTP messages. There are lots of exceptions and additional details, but we don't need to go into those here. If you want to learn more about HTTP, you can read the specs or pick up a book.

HTTP Methods

Another important element to HTTP is its use of *methods* at the start of a request to allow clients (the ones sending the request) to tell servers (the

ones receiving the request) what kind of action they would like the server to take on their behalf. As I write this book, I can see that close to 40 different HTTP methods are registered with the Internet Assigned Numbers Authority (IANA)^[16]—the people responsible for keeping track of these kinds of things.

The CRUD Methods



Many people use the four basic HTTP methods (POST, GET, PUT, DELETE) in a very relational database-oriented way. In this approach, POST is for creating a new record, GET is for reading an existing record, PUT is for updating an existing record, and DELETE is for removing an existing record. These four methods are sometimes called the CRUD methods (for *create*, *read*, *update*, and *delete*). This database approach isn't the way the authors of the HTTP specification thought about the HTTP methods, and I won't be using the term *CRUD* in this book.

While these keywords are called HTTP “methods,” they aren’t really methods as programmers would think of them. They aren’t functions or routines that servers call. They’re just keywords in the specification that help clients and servers understand what’s expected when making requests. Because of the way the HTTP standard is set up, new keywords can be added at any time. For example, the **PATCH** keyword was standardized in 2010,^[17] and other keywords were used in the past. We’ll be using the four most common HTTP methods (GET, PUT, POST, DELETE) in our work for this book, but it’s a good idea to know that others exist.

Safe and Idempotent

HTTP method keywords represent an agreement between servers and clients. That agreement includes which of the keywords represent *safe* requests and which represent *idempotent* requests. A safe request is one that doesn’t make

any meaningful changes to existing data. For example, HTTP GET is defined as “safe” since making a GET request is really just a read operation. On the other hand, HTTP DELETE is defined as “unsafe” since making a DELETE request should result in removing or deleting data.

Along with the property of *safety*, HTTP methods have the property of *idempotence*. An idempotent method can be repeated over and over again and still have the same results. For example, if you use the DELETE keyword in a request to remove record number 21 from the server, you can send that request once or one hundred times and the results will always be the same: only record number 21 will be removed from the system.

Despite its common use, the POST method is *not* idempotent. That means if you use the POST keyword to add a new record to the system and you execute that request multiple times, you will get multiple records added to the system. Sometimes that’s the behavior you want, but often you only intend for a single record to be created. And that’s where using the HTTP POST method can be challenging.

For example, what happens if you use HTTP POST to send a message to a banking service telling that service to transfer \$200 from account A to account B and you never get a response back? No “OK, done.” No “Sorry, there was an error.” Nothing. Now what should you do? Should you send the same request again?

In this example, we don’t know if (1) our request never reached the server (meaning the account transfer never happened) or (2) our request reached the server and was executed just fine, and just the *response* (telling us it all went well) never got back to us. If (1) is true, we can send the request again. If (2) is true, we shouldn’t send it again. It’s hard to know.

However, if we used HTTP PUT (an idempotent method) to send the request, we’d know that we could safely repeat the request without worrying about executing the bank transfer multiple times. Hopefully, this bank

example makes sense. We'll talk more about using PUT and POST in Part II when we start building our working API.

The Power of the Web

When creating our web APIs, it's important to remember that we're building them on top of the existing web. People used to call it the *World Wide Web* or *WWW*, but that seems a bit redundant now since we all assume the web is everywhere. As mentioned earlier, what's interesting about the web is that it's not really a standard in the way that HTTP is a standard. There's no definite committee-managed document that defines the web. However, the way people build on the web is pretty much the same everywhere. In this way, the web can be called a common practice or set of patterns. And being familiar with these common practices can help you get up to speed quickly on building great APIs.

The web was the brainchild of Tim Berners-Lee in 1989.^[18] He wanted to build a “linked information system” based on a “non-linear text system known as hypertext.” Basically, he wanted to come up with a relatively easy way for people to publish their data online and connect their publications to others online. And he wanted to do this in a way that could easily change over time without a lot of detailed coordination. When you think of the original reasons for Berners-Lee to create his World Wide Web, that sounds quite a bit like the reasons people want to use APIs.

To make this idea a reality, Berners-Lee created the initial rules for the HTTP protocol and for the hypertext markup language (HTML) message format. Not long after that, others contributed the Cascading Style Sheets (CSS) and JavaScript standards that helped launch the web browser as we know it today. This mix of simple, stand-alone elements (protocol, format, styles, and scripting) grew at an exponential rate not just because they were each well-designed specifications, but also because the rules on how they were used together were so easy to adopt. You didn't need to be a computer scientist to start creating pages on the web. The power of the web comes

from (1) clear standards everyone can agree on and (2) an easy on-ramp for those just getting started.

And that's the way we'll approach creating our APIs. Much of this book will focus on clear standards everyone on your team and in your company can agree on. Like the web standards, some of the specifications we'll cover here will be a bit quirky and sometimes frustratingly limited. But they work because they're well established and clearly defined.

We'll use those standards to build APIs that are relatively easy to use too. By making the APIs pretty easy to understand and try out, we can create solutions that other programmers (in your organization or even around the world) can also try out and use without having to talk to you first.

Finally, we'll be creating APIs that rely on Berners-Lee's idea of using hypertext to link information together. That's an element many APIs lack, and dropping the hypertext (sometimes known as *hypermedia*) feature makes it hard to maintain APIs over time when small changes arise. And Berners-Lee wanted to make it possible to support small changes over time without the need for detailed coordination between the publishers and the consumers of that data.

The web is a set of practices that use clear agreed-upon standards, rely on hypermedia to create a linked information system, and is built in a way that makes it easy to get started and supports small changes over time. That has been the mission and vision of the web for thirty years.

The Style of REST

While HTTP is a standard and the web is a set of common practices, another important element of web APIs deserves a bit of attention: the REST (or RESTful) style for APIs. The acronym REST comes from the title of a single chapter of a PhD dissertation published in 2000 by Roy Fielding. The title of the chapter was “Representational State Transfer (REST)” and the dissertation title was “Architectural Styles and the Design of Network-Based

Software Architectures.”^[19] Like most dissertations, this one’s a bit hard to read and covers quite a bit of history and analysis that usually isn’t too interesting for those programming working software systems. However, unlike most dissertations, Fielding’s work to define the REST architectural style has had a huge impact on the way we think and talk about APIs for the web.

Fielding’s work centered around how to define a software architecture *style*. He wasn’t describing a standard (like HTTP) or even a set of common practices (like the web), just a style or manner of doing something. In this case that “something” is software architecture that runs on networks like the Internet. The reason so many people reference Fielding’s work when talking about APIs is that his way of describing his REST style was so clear. It goes right along with what we’ve already been talking about here—clear standards that everyone agrees on can be very powerful when creating software that runs on lots of machines, like the way the web works.

Using Berners-Lee’s World Wide Web as a starting point, the REST style defines a set of constraints that are selected based on a set of what Fielding calls “architectural properties of key interest.” To look at it another way, Fielding listed out what he thought were the valuable properties of the web and then selected a set of rules (the constraints) that software developers would need to follow in order to realize the best aspects of the web.

Fielding’s Dissertation



If you want to learn more about Fielding’s work, I highly recommend reading his dissertation. It’s not as challenging to get through as you might think, and I find some parts of it rather funny too.

Fielding identified the following list of important properties he thought every web-based system should do:^[20]

- Performance: Offer high performance through design
 - *Network performance*: Make the most of throughput, limit overhead, and maximize usable bandwidth
 - *User-perceived performance*: Reduce latency and wait times by rendering content as early as possible
 - *Network efficiency*: Don't overuse the network; when possible respond to requests from local storage
- Scalability: Support large numbers of machines that are easily configured and deployed
- Simplicity: Rely on the principle of the separation of concerns and the generality of interfaces
- Modifiability: Make it easy to modify a running system to support changes over time
 - *Evolvability*: Change an existing component without negatively impacting other components
 - *Extensibility*: Be able to safely add new functionality to the system over time
 - *Customizability*: Be able to modify a single instance of a component without having to modify all other instances
 - *Configurability*: Be able to change the functionality of a deployed component through configuration alone
 - *Reusability*: Be able to use the same component in more than one place in the system
- Visibility: Make it easy to monitor and manage running components

- Portability: Make it easy to provide the same functionality in different environments, operating systems, and so on
- Reliability: Make it unlikely for the failure of a single component to impact the entire system

That's a pretty long list of properties, and I didn't even explain many details in this brief summary. But you get a pretty good idea of what Fielding was thinking about. One of the challenges of this list is that it's very difficult to measure compliance. How would we know if our API exhibits the proper amount of "simplicity" or "modifiability"? How do you even measure "simplicity" in a system?

To solve this problem, Fielding came up with a set of rules or constraints that he said developers and software architects should follow when creating their systems. And he selected these rules based on their ability to increase the likelihood that the resulting system would have the above list of properties. Finally, he made his set of constraints pretty easy to measure.

Here's Fielding's list of REST constraints:^[21]

- Client-server: Implement a clear separation between the server and the client app with request/response interactions.
- Stateless systems: Make sure each request from the client contains all that's needed to complete that request. Don't rely on any request context information stored on the server.
- Cache: Make it possible for clients to know whether the server response can be cached locally and replayed in the future.
- Uniform interface: Apply the generality of interface patterns between all components.

- *Identification of resources*: Make each requestable resource have its own unique identity (for example, an HTTP URL).
- *Resource representations*: Allow clients and servers to control the representation of each request and response through message formats.
- *Self-descriptive messages*: Every request and response message should contain all information needed to understand the message.
- *Hypermedia to change application state*: Changing the state of the data stored on servers should be accomplished using links and forms that completely describe the details of the interaction.
- Layered system: Each component should only be aware of the next component in the chain and not know about other layers of components in the system.
- Code on demand: Client applications should be extensible through the use of downloadable code-on-demand instead of having to build new client applications for each new feature.

Like the list of properties, Fielding's list of constraints is rather extensive and, at times, a bit difficult to grasp. Although it may not be clear from this brief listing, his dissertation explains each constraint in terms of the properties that constraint is meant to support or encourage. When combined, the properties and constraints provide a clear set of goals and principles developers can reference when creating APIs for the web.

Pulling It All Together

Understanding the HTTP protocol, the motivations for Berners-Lee's web practices, and Fielding's goals for defining a style of designing and building applications for the web can help us make our own decisions about how we'll go about designing and implementing our APIs. We can improve the

quality of our APIs by using the HTTP's message-based approach, adopting the web's use of a small set of standards, and making it easy to get started, as well by applying as the REST style of constraining our options in order to build more consistent solutions. And when we run into challenges, we can look back on these three pillars of good web practice to see if we can find possible answers that fit into the standards, practices, and stylistic approach that have made the web so successful over the last few decades.

As you can tell from this short trip through the history of the web, a lot of work has gone into thinking ahead and engaging in managing possibilities and assumptions for developers. Managing possibilities is also important for individual projects, which is what I'll cover in the next section of this chapter. This time, we'll focus on managing all the files in our API projects, and we'll do that with a very handy command-line tool called Git and its related website, <http://github.com>.

Managing Files with Git

A critical element in doing any API work is managing all the project files. That means being able to track all the changes you make to your files, sharing them in a way that allows others on your team to also make their own changes without overwriting yours, and—in case of serious problems—the ability to reverse one or more of those changes in order to undo whatever mistakes were made in the recent past. This kind of work falls under the category of version control systems (VCS) or source code management (SCM).

Over the years, developers have seen several systems for managing code files. Concurrent Versions System (CVS) was released in 1990, and Perforce came out in 1995. Another product, Apache Subversion (or SVN), was first released in 2001 and is still quite common today. For example, all the files for this book were managed in an SVN repository. Two other systems were released in 2005: Mercurial and Git. There have been several other systems for managing code files over the years, but so far, Git has been tracked as the most commonly used code management system, according to the report “Version Control Systems Popularity in 2016.”^[22]

About Git



The tool we'll use to manage our project files is Git, the “free and open source distributed version control system designed to handle everything from small to very large projects” (<https://git-scm.com>). If you don't already have this tool installed, see Appendix 1, [*Installation Hints*](#), for details on how to download and install it. We'll just cover the basics of Git here. You can learn more about the power of Git in the book *Pro Git*, by Scott Chacon and Ben Straub, which is available for free at <https://git-scm.com/book/en/v2>.

I like to use Git because of its rich command-line interface. There are also some very good graphical user interface (GUI) versions too. You can also find many code editors that support `git` commands within the UI of the editor. There are lots of options for using Git for your projects. For this book, we will be using the command-line interface since it is the quickest way to learn and requires very few resources to start. If you are already familiar with Git from using a GUI version or from using it within your current code editor, this command-line tutorial can give you a chance to brush up on your Git skills too.

For the work we'll be doing in this book, we need to cover some basic `git` commands that allow us to create a new Git repository (often called a *repo*), check the `status` of the repo, and `add` and `commit` any changes to it. We'll also learn how to check for differences (`diff`) between the current copy of your files and the ones already in the repo. And as a bonus, we'll learn how to copy an existing repo to a remote location (in our case, to the GitHub website), where other people can interact with your project.

Creating Your Git Repository

The Git system allows you to mark any file folder on disk as a project repository. You can do that by initializing that folder using the `init` command. The following is an example interactive session where we'll create a new folder on disk, use the `cd` command to change directories and move into your new folder, and then execute Git's `init` command to mark it as our project repository:

```
mca@mamund-ws:~$ mkdir onboarding
mca@mamund-ws:~$ cd onboarding/
mca@mamund-ws:~/onboarding$ git init
Initialized empty Git repository in /home/mca/onboarding/.git/
mca@mamund-ws:~/onboarding$
```

Your command-line prompt might look a bit different, but the interaction should be the same. Note that all the Git commands are preceded by the `git`

keyword. We now have a working Git repository ready for use.

We can test this by executing the `status` command to check the state of the repo:

```
mca@mamund-ws:~/onboarding$ git status
On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)
mca@mamund-ws:~/onboarding$
```

The reply from Git contains a couple of important things. First, you see the phrase `On branch master`. This means you're currently looking at the root (master) instance of your files. Git allows you to create another copy of your files (a branch) in order to make small changes and test them without affecting the root instance (master branch). We'll use the practice of always working on the master branch, so we'll skip over the details of branching for this book.

The other part of the response (`nothing to commit`) tells us there have been no changes since the repo was last updated (in our case, since it was created). We're about to change that.

Adding and Committing Files to Your Repository

Any time you change the contents of your project folder, such as (1) adding a new file, (2) editing an existing file, or (3) deleting a file, the Git system tracks that change. Even renaming or moving a file or adding/renaming/removing folders gets tracked by Git. In this way, any changes can be monitored and, if needed, reversed.

For now, let's add a file to the root of the project: the README.md file. This file is used to tell anyone who has a copy of our repo what this project is about. For now, open up your favorite text editor and enter the following markdown text:

ch02-http/git-readme.md

```
## Onboarding API Project
```

```
This project contains files related the BigCo Onboarding API.
```

Once you've completed the text, save it to the project folder we created in the previous section. Next, return to the command line and check the status of your repo:

```
mca@mamund-ws:~/onboarding$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

  README.md

nothing added to commit but untracked files present (use "git add" to track)
mca@mamund-ws:~/onboarding$
```

You should see that the reply from Git's `status` command now shows an *untracked* file named `README.md`. You can also see that Git is hinting that we might want to use the `add` command to add the file to our tracking history. In fact, we want to do more than just add the file. We also want to `commit` it as a permanent change to our repository. We can do both like this:

```
mca@mamund-ws:~/onboarding$ git add --all
mca@mamund-ws:~/onboarding$ git commit -m"add README to the repo"
[master (root-commit) 6f509d4] add README to the repo
 1 file changed, 4 insertions(+)
 create mode 100644 README.md
mca@mamund-ws:~/onboarding$
```

And that's it. We've made changes to our project and committed those changes to the repository for tracking.

Managing Your Changes

Sometimes you may forget if you've made any change to your repo since you last executed the `commit` command. You can use the `status` command to check for any changes. Let's change the `README.md` file and then use the `status` command to validate our change.

Edit your `README.md` file to look like this (note I added the words "all the" before the word "files"):

```
## Onboarding API Project

This project contains all the files related to the BigCo Onboarding API.
```

We then check the repo status like this:

```
mca@mamund-ws:~/onboarding$ git status
On branch master

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
mca@mamund-ws:~/onboarding$
```

You can see that Git tells us a file has changed. But it doesn't tell us exactly *what* was changed within the file. We can use another command to get that information: the `diff` command. The `diff` command compares the file we have on our disk to the one already committed to the repository. For our example, when you use the `diff` command you should see this:

```
mca@mamund-ws:~/onboarding$ git diff
diff --git a/README.md b/README.md
index 6647b6d..e136808 100644
--- a/README.md
+++ b/README.md
@@ -1,4 +1,4 @@
## Onboarding API Project
```

```
-This project contains files related to the BigCo Onboarding API.  
+This project contains all the files related to the BigCo Onboarding API.
```

```
mca@mamund-ws:~/onboarding$
```

Along with some internal tracking details, you'll see two lines in the response: one made of minus signs that shows what was committed to the repo, and one made of plus signs that shows the changes you made and have on your disk. The plus and minus information tells us that, when we do our next **add** and **commit** commands, Git will remove one line (the minus line) and add another line (the plus line).

However, we might not want to add and commit that change. Instead we might want to reverse (or revert) the change. We can do that using the **checkout** command. Git's **checkout** command lets us take the version of the file already in the repo and use it to overwrite the copy we have currently on our disk. This replaces our edited copy with the one already committed to the repo. That action looks like this:

```
mca@mamund-ws:~/onboarding$ git checkout -- README.md  
mca@mamund-ws:~/onboarding$ git status  
On branch master  
nothing to commit, working directory clean  
mca@mamund-ws:~/onboarding$
```

In this example, I executed the **checkout** command and then did a **status** command to confirm that the revert was successful.

Adding a New Folder

Let's try something a bit more involved. We'll create a new folder and combine the use of curl and Git to record API responses on disk for later reference. For example, let's create a folder named **services** within the repo and then make a call to the **Company** service root, saving it to disk as **company-root.json** and then committing these additions to our project repo.

The first step is to create the folder and use curl to make the API call and save it to disk:

```
mca@mamund-ws:~/onboarding$ mkdir services
mca@mamund-ws:~/onboarding$ curl http://company-atk.herokuapp.com/company/
>services/company-root.json
% Total    % Received % Xferd  Average Speed   Time   Time   Time  Current
          Dload  Upload Total   Spent    Left  Speed
100    465     0    465     0      0  145k      0 --:--:-- --:--:-- --:--:-- 227k
mca@mamund-ws:~/onboarding$
```

The next step is to check out the repo's [status](#) to confirm the changes and then [add](#) and [commit](#) those changes to the repo:

```
mca@mamund-ws:~/onboarding$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    services/

nothing added to commit but untracked files present (use "git add" to track)
mca@mamund-ws:~/onboarding$ git add --all
mca@mamund-ws:~/onboarding$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   services/company-root.json

mca@mamund-ws:~/onboarding$ git commit -m"add services folder"
[master 491f4c1] add services folder and company-root response
 1 file changed, 26 insertions(+)
 create mode 100644 services
```

You probably noticed that the first [status](#) check only showed that a new *folder* was added ([services](#)) without listing the files within that folder. However, once we executed the [add1](#) command, calling [status](#) shows us all the affected files within our new folder. I admit I find some of the responses to [git](#) commands to be a bit confusing and/or missing important information.

To get past this, I use the `status` command quite often—even multiple times in a Git session—in order to confirm that my changes were handled as expected.

That completes the basic Git commands you'll need to manage your API project for this book. As I mentioned earlier in the chapter, quite a few `git` commands are worth learning, and I highly recommend the book *Pro Git* if you're interested in digging deeper into this very valuable tool.^[23]

I need to cover one more bit of information before closing out this chapter, and that's creating a remote copy of your repo for others to access. To do that, we'll use the public website, GitHub.

Creating Your Public GitHub Project

Sometimes you want to share your Git repository with other people. You may want to allow others to use your code project on their own, you may be looking for collaborators on your project, or you may just want to be sure your project has a public copy somewhere else in case something happens to the one on your local machine. There are a handful of ways to do this, and your team may have an internal copy of Git or some other software repository you can use. For this book, we'll be using the GitHub website as our remote repo for our API project.

About GitHub



The website we'll use to handle our remote repo is <http://github.com>, “the world’s largest development platform.” GitHub is free, but you’ll need a GitHub account and you’ll need to do some setup work to make it possible to safely manage your remote repo from your desktop or laptop machine. For details, see Appendix 1, [Installation Hints](#). You can also find some decent assistance at GitHub’s Help website. I recommend starting here: <https://help.github.com/categories/setup>.

You need to take three steps to complete the process of tying your local Git repository to a remote one on GitHub:

1. Create an empty repo on GitHub.
2. Use the `remote` command to link your local repo to the one on GitHub.
3. Use the `push` command to sync the contents of your repo to GitHub.

Once you complete these steps, you only need to do the last one (the `push` command) to keep both repositories in sync. Here's how it all works.

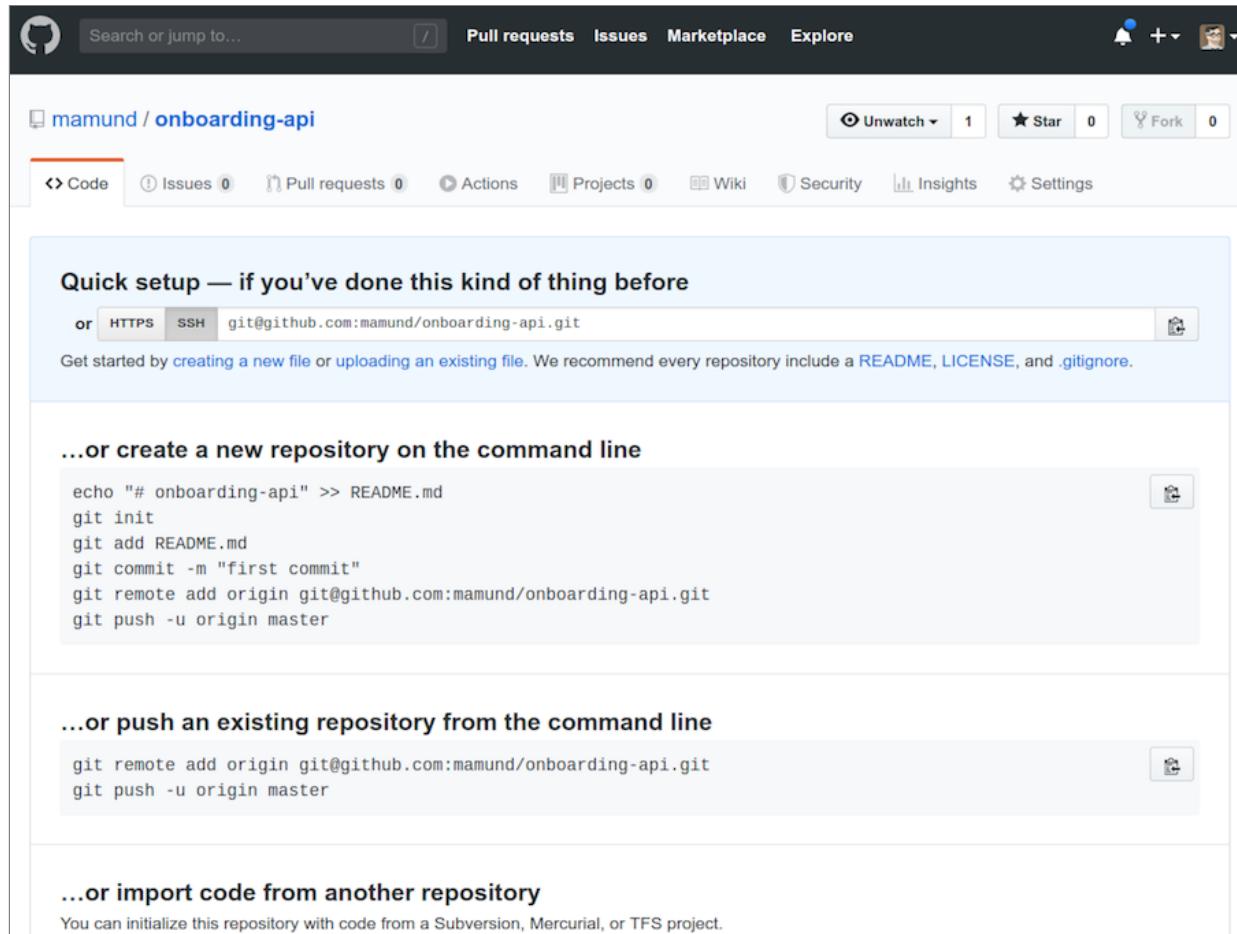
Create an Empty Repository on GitHub

The first thing we need to do is create a new repository on GitHub. This will act as the remote location for our local repository on disk. To do this, log into the GitHub website and click the New Repo button or just navigate to <https://github.com/new>. Type in a name for your repo (I used `onboarding-api`) and a short description (“remote repo for the BigCo onboarding API project”). You can leave all the other options set to their default values. At this point your screen should look similar to the one shown here:

The screenshot shows the GitHub interface for creating a new repository. At the top, there's a navigation bar with links for 'Pull requests', 'Issues', 'Marketplace', and 'Explore'. Below the navigation is a search bar and a user profile icon. The main content area has a title 'Create a new repository' and a sub-instruction: 'A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.' There are two tabs: 'Owner' (selected) and 'Repository name *'. The 'Owner' tab shows a dropdown menu with 'mamund' and a checked radio button. The 'Repository name' field contains 'onboarding-api' with a green checkmark. A note below says, 'Great repository names are short and memorable. Need inspiration? How about [improved-happiness?](#)' Under 'Description (optional)', there's a text input field containing 'remote repo for BigCo onboardingAPI project'. Below that, there are two radio button options: 'Public' (selected) and 'Private'. The 'Public' option is described as 'Anyone can see this repository. You choose who can commit.' The 'Private' option is described as 'You choose who can see and commit to this repository.' A note below says, 'Skip this step if you're importing an existing repository.' There's also a checkbox for 'Initialize this repository with a README', which is unchecked. At the bottom, there are buttons for 'Add .gitignore: None ▾', 'Add a license: None ▾', and a help icon. A large green 'Create repository' button is at the very bottom.

Once the fields are filled in properly, click the “Create repository” button to commit your changes and create the actual GitHub repo. When you do that, GitHub responds with a screen that contains lots of hints on how to fill in the contents of your repository, as shown in [the next screenshot](#).

One of the options is labeled “push an existing repository from the command line.” That’s what we’ll do. Your screen should include the two commands you’ll need to execute in order to complete the job as shown in the [screenshot](#). The first is the `remote` command to link the repos, and the second is the `push` command to sync the contents of the repos.



The screenshot shows a GitHub repository setup page for a repository named 'mamund / onboarding-api'. The top navigation bar includes links for 'Pull requests', 'Issues', 'Marketplace', and 'Explore'. On the right, there are icons for notifications, a plus sign, and user profile. The repository details show 1 unwatched star and 0 forks. Below the header, there are tabs for 'Code', 'Issues (0)', 'Pull requests (0)', 'Actions', 'Projects (0)', 'Wiki', 'Security', 'Insights', and 'Settings'. A section titled 'Quick setup — if you've done this kind of thing before' provides instructions for cloning the repository via HTTPS or SSH. It also suggests creating a new repository on the command line with the following commands:

```
echo "# onboarding-api" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin git@github.com:mamund/onboarding-api.git
git push -u origin master
```

Another section shows commands for pushing an existing repository from the command line:

```
git remote add origin git@github.com:mamund/onboarding-api.git
git push -u origin master
```

A final section discusses importing code from another repository, noting that it can be initialized from Subversion, Mercurial, or TFS projects.

Use the remote and push Commands

The `remote` command adds a link from your local repository to another repository. In this case, we're going to link our file repo to the one we created at GitHub. To do that, type this at the command line:

```
git remote add origin git@github.com:mamund/onboarding-api.git`
```

In this example, the name of the repo on GitHub is `onboarding-api`. Your repo name might be slightly different and you should type the name you chose where I entered `onboarding-api`. What we just did is create a remote link for our local repo named `origin` that points to our repo at GitHub.

Right now that repo on GitHub is still empty. We can change that by executing the `push` command to push the contents of our local repo up to the repo on GitHub. That command looks like this:

```
mca@mamund-ws:~/onboarding$ git push -u origin master
Counting objects: 7, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (7/7), 790 bytes | 0 bytes/s, done.
Total 7 (delta 0), reused 0 (delta 0)
To git@github.com:mamund/onboarding-api.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
mca@mamund-ws:~/onboarding$
```

This command tells Git to “[push](#) to the remote location named [origin](#) everything in the [master](#) copy here on my machine.” As you can see from the previous session, the [push](#) command then writes data on the remote location and sets up tracking to make it easy to keep both repos in sync.

You can confirm everything worked as expected by visiting the project page at GitHub and refreshing it. It should now show a file listing similar to what you see on your own local machine and shown in the following screenshot:

The screenshot shows the GitHub repository page for 'mamund / onboarding'. The top navigation bar includes 'Pull requests', 'Issues', 'Marketplace', and 'Explore'. The repository name 'mamund / onboarding' is displayed, along with 'Unwatch', 'Star', and 'Fork' buttons. Below the repository name, there are sections for 'Code', 'Issues 0', 'Pull requests 0', 'Actions', 'Projects 0', 'Wiki', 'Security', 'Insights', and 'Settings'. A summary box shows '2 commits', '1 branch', '0 releases', and '1 contributor'. The 'Branch: master' dropdown is set to 'master'. Buttons for 'New pull request', 'Create new file', 'Upload files', 'Find File', and 'Clone or download' are visible. The main content area lists two commits: 'mamund add services folder' and 'services add services folder'. It also shows 'README.md' with the commit message 'added README'. At the bottom, a section titled 'Onboarding API Project' contains the text 'This project contains files related the BigCo Onboarding API.'

Now any time you make changes to the local repo (that is, whenever you [add](#) and [commit](#)), you can also do a [push](#) to make sure your public GitHub repo has the same contents as the repo on your local machine.

What's Next?

In this chapter, we covered the basics of HTTP, reviewed common practices for creating services for the web, and reviewed the REST style for implementing services over HTTP. Essentially, you learned that HTTP is a standardized protocol that all web servers and clients use in order to ensure they can connect and share data with each other. This protocol has existed for over thirty years and has continued to work even through several changes. That's a goal we'll try to live up to when we create our APIs too.

We also covered the idea of the World Wide Web, sometimes called WWW or just *the web*. Although not an actual protocol like HTTP, the practices of the web—the use of HTML for data and actions, JavaScript for extending functionality, and CSS for improving the user experience—have been responsible for the success of the web today. While our APIs won't need to deal with CSS styling of user experience and we won't be using JavaScript to extend them, we'll follow the common practice of the web by adopting the pattern of sharing messages instead of objects. And we'll make sure our messages include more than just simple data but also include rich metadata (in the form of HTTP headers) and action elements such as links and forms.

In addition, we briefly reviewed Roy Fielding's definition of REST (REpresentational State Transfer) since it often comes up when creating APIs for the web. Even though REST is just a *style* and not a protocol or practice, the style has been an important influence in the way programmers for both service providers and consumers design and implement their APIs. And we'll use many of Fielding's REST principles in our API too.

Finally, you learned how to use the Git command-line tool to set up your own project repository to track all the files and folders we will add, edit, and delete throughout the lifetime of our API project. You also added a remote repository at the GitHub website and used Git to link and sync our local repository to the remote one on the GitHub website.

Now that we've gotten our start on using Git utilities and reviewed the basics of HTTP, it's time to start creating our own Onboarding API that will combine all three of these services into a living, working API that others can access. In the next chapter, we'll look at how to collect the information needed to create our new Onboarding API. You'll also learn how to use the npm command-line tool to define our API project.

Chapter Exercise

In the previous chapter, we used curl to make a series of API calls (via HTTP) and saved them to disk. Now it's time to organize those responses into our first Git repository. We'll use this repository (or *repo*) to hold all the project-specific materials we'll be generating in the upcoming chapters.

In the last exercise (see [*Chapter Exercise*](#)), we used curl to make requests to BigCo's existing services and saved the responses to a folder on our machine called **services**. We also added a folder with the same name to our Git repo earlier in this chapter (See [*Adding a New Folder*](#)). Now it's time to put these two exercises together and make sure all our saved curl responses are added to our Git project's **services** folder.

Once you add all the responses from the Company, Account, and Activity services to your **services** folder, use **git add** and **git commit** to commit those changes to your local repository. Then use the **git push** command to push your local changes to the shared GitHub repository in the cloud.

See Appendix 2 ([*Solution for Chapter 2: Understanding HTTP, REST, and APIs*](#)) for the solution.

Footnotes

[14] <https://www.w3.org/Protocols/HTTP/AsImplemented.html>

[15] <https://tools.ietf.org/html/rfc7540>

[16] <https://www.iana.org/assignments/http-methods/http-methods.xhtml>

[17] <https://tools.ietf.org/html/rfc5789>

[18] <https://www.w3.org/History/1989/proposal.html>

[19] <https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>

[20] https://www.ics.uci.edu/~fielding/pubs/dissertation/net_app_arch.htm#sec_2_3

[21] https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_1

[22] <https://rhodecode.com/insights/version-control-systems-2016>

[23] <https://git-scm.com/book/en/v2>

Copyright © 2020, The Pragmatic Bookshelf.

Part 2

The Design Phase

Chapter 3

Modeling APIs

Now that we have the groundwork set from the previous two chapters, we’re ready to start the process of building our Onboarding API project. And as you can tell from the title of this part of the book (The Design Phase), we’ll spend time working on the various design aspects of API development that happen before we start writing code.

In this chapter, we’ll take a look at some general theory on how *interfaces* work (UI and API) and then talk about how we can harness these interface patterns as we model our own API project. Along the way we’ll learn about API stories and about translating these prose stories into API workflow documents. We’ll also learn how to use a very powerful tool—the Node Package Manager (npm)—to manage our API project. As we’ll discuss later in this chapter, the npm tool is both versatile and handy. We’ll use it throughout the rest of the book to help us manage our project, integrate other modules into our code, and even test and deploy the finished API.

It’s also worth noting that even though this chapter is titled “Modeling APIs,” we’ll still be talking about the design process here. As we set out to model our API, it’s important to have a good grasp of the way design influences what we create. First we’ll review the work of Donald Norman and how his insights can help us create the kind of API that’s not just functional but relatively intuitive and pleasing to work with. Once we have

this kind of understanding in our kit of techniques, the actual act of designing our Onboarding API will go smoothly.

The Importance of Design

Before we jump into the work of stories, interfaces, and npm, let's take a couple of minutes to talk about the importance of the design phase for any API project. Design work sets the stage for everything. It helps us get a full understanding of the problem users are trying to solve, the workflow and data shared by various parties along the way, and the expectations of developers and users when they access our API. Design helps us lay the groundwork for the actual *building* of the API.

However, in our excitement to get started on an API project, design work can easily be skipped or given only a cursory effort. Many of us are attracted to the field of computing because we like *programming*, and we can easily be lured into writing code too early—before we have all the details we need for creating a great API. At the same time, if we're not careful, we can get bogged down in too much *up-front* effort and end up overdesigning. We need to maintain a balance.

The good news is that we can use an important principle of all good design—a repeating cycle—to balance design needs against programming needs. We can design a bit, code that, see how it looks, and then go back and design some more. This cycle, or *loop*, has been understood by designers and engineers for quite some time, with quite a bit of writing about it in the design and design-thinking arena.

Understanding Norman's Action Lifecycle

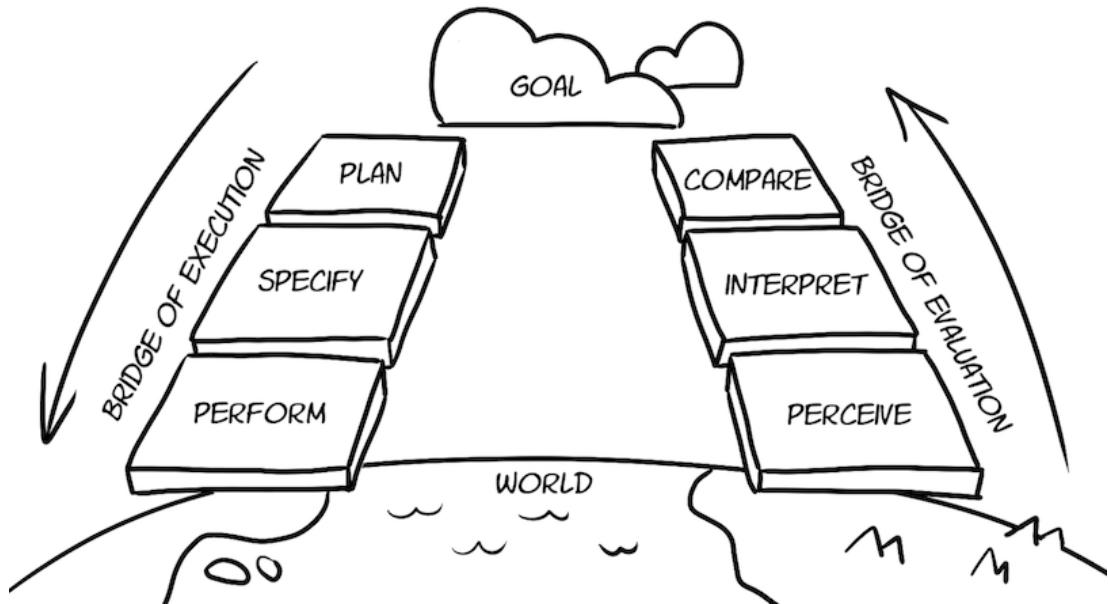
In his 1988 book, *The Design of Everyday Things*,^[24] author Donald Norman lays out several principles of designing and building “things” (doorways, telephone, computers, and so on) for human interaction. A former Apple Fellow and co-founder of the human-computer interaction design firm Nielsen-Norman Group, Norman was the first to use the phrase “user experience” in a job title and is considered by many to be one of the founders of the field of human-computer interaction (HCI). He released an updated edition of his book in 2013, which continues to influence the field of computing.

One important concept in Norman’s work that can be directly applied to our work of modeling APIs is Norman’s *Action Lifecycle* ([see the diagram](#)). It’s a model for the way humans interact with any object, whether on a computer screen or in the real world.

Norman’s Action Lifecycle is composed of seven stages divided into two sections. Section 1 is the “Bridge of Execution.” This is where humans plan and then commit some action. The execution side of the model has three parts:

- Plan
- Specify
- Perform

For example, I might decide I need to turn the light on in my room in order to continue reading a book as the sun sets—that’s my *plan*. I then need to figure out where the light switch is (I *specify* exactly what to do). Finally, I need to actually go over to the wall and flip on the light switch—I *perform* the task. In that last step (perform), I affect the world around me—I change something.



The next set of steps, the ones Norman calls the “Bridge of Evaluation,” are what I use to determine what happened in my world and decide whether or not I actually accomplished my intended goal:

- Perceive
- Interpret
- Compare

I *perceive* whether the light actually turned on. I *interpret* the results of my actions—meaning I associate my change of the light switch to the light that is now on in the room. And I *compare* my expectations (I wanted the room to be brighter) with the actual real world results (in this case the light is on and the room is bright enough for me to read).

That last step (compare) leads me to decide whether I’ve accomplished my goal. The goal, by the way, is Norman’s seventh step. If I’ve met my goal, I’m all done and can sit down and get back to reading my book. However, if the switch I used actually controls the overhead fan and not the light, I did not accomplish my goal yet. That means I need to go back and start the cycle again (plan, specify, perform, perceive, interpret, and compare) and continue until I find the proper switch and make the room bright enough to continue reading.

Everyday Things

While this model sounds rather tedious, it's really quite handy. You can apply it to all sorts of things in your everyday life. The light switch example is rather simple, but this action lifecycle is present in many activities.

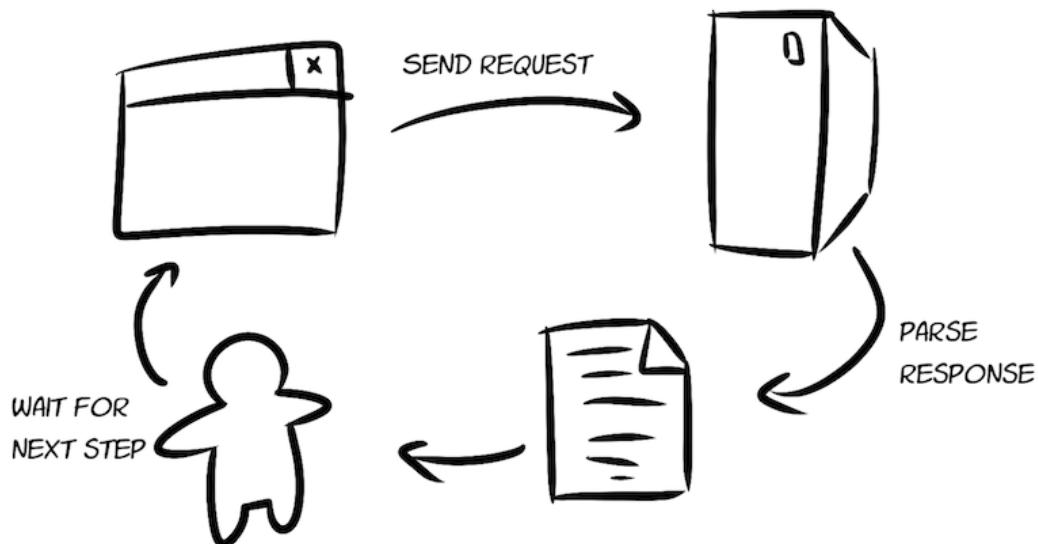
Consider, for example, the task of searching for this book on the web. You need a plan, you need to determine some specific search criteria, and you need to type that criteria into an Internet search engine. Depending on what you typed as your criteria, you might not get the answer you expect when you perceive, interpret, and compare the search results on the computer to the response you're thinking of in your head. That means you need to start the loop again, and so on, until you get your expected results (or just give up and go outside for a walk).

All sorts of things you do every day use this action lifecycle, such as more complex activities like driving to the store to buy groceries or meeting friends for dinner. In everyday life you often need to interact with several things in order to accomplish an important task. Commuting to work in the morning is one example: You need to first get out of the house (that's one lifecycle), then you need to drive to the commuter parking lot (that's another lifecycle), and then you need to catch the next train into the city (another lifecycle), and then you need to make your way from the downtown train stop to your office (one more lifecycle). Each of these cycles has its own challenges; you may need to stop for gas on the way to the parking lot, you may need to take a detour along the way, you may need to catch a different train into the city, and so forth. We often use this loop quite subconsciously, but we *do* use it.

We can use the same process as a model for our API—and as a model for the act of creating it too.

API Action Lifecycle: The RPW Loop

Norman's Action Lifecycle is a good approach for us when we model our own APIs. But, as already mentioned, it can be a bit tedious to work with. Knowing exactly what happens in the *perceive* step or the *interpret* step might not be very helpful when all we're trying to do is model an API to solve someone's problem. For this reason, we can simplify Norman's seven-step model into three general actions that work for all the types of APIs you need to model: (1) request, (2) parse, and (3) wait, as illustrated in the figure [shown](#).



This request–parse–wait (RPW) loop makes lots of sense if you think about how the web works—especially how a web browser works. Users type something into the browser to start a *request* (for example, I type <https://training.amundsen.com> into the address bar and press Enter). Next, some server somewhere sends a response that the browser needs to *parse* (for example, the HTML message returned by a server gets parsed and displayed as a web page with graphics, tables, CSS, and so forth). Finally, after the browser displays the parsed response, it *waits* for me to click something, fill in a form, or just type something else in the address bar; and when I do that, the loop starts again. If you think of your potential API client applications as a kind of web browser, using the RPW loop as a design pattern can be very helpful.

The RPW loop is a good way to think about your API *server* too. Web servers behave very much like web clients in that they spend most of their time waiting for a request to come in. And when it does, a server needs to parse the incoming request (What URL was requested? Is there a body of data to process? What kind of response format is requested? and so forth). After processing that request, the server needs to create a response to send to the client. Once that's done, the server just waits until another request appears.

What About Real-Time APIs?



Even if your server and your client are not using typical request/response HTTP messages and are, instead, using something like reactive services, WebSockets, or some other real-time interface, the same RPW loop exists. It just has very small wait times and even smaller response and request messages.

So, modeling your API with the RPW loop in mind is a good idea. That means taking all you know about what your users are trying to accomplish (the goal), all the tasks they need to complete along the way (each of those is its own cycle), and how your users determine whether they're making progress (comparing) and what it is that lets them know they've actually completed their work (met their goal).

This can be a bit tricky since people don't always *think* in the way Norman does or even the way most developers think. Your users will often just tell you how they do their daily work, what forms they fill out, and where they send the forms when they're done. And they may tell you only *part* of the story. To use our fictional company as an example, some people at BigCo just work on handling company records and don't know anything about account records (that's another department). But when you're working to model an API that combines the work of both the company team and the

account team, you'll need to come up with your own set of action lifecycles to accomplish a goal that no one else yet knows about.

That's what we'll be doing here when we model our own Onboarding API lifecycle.

Modeling Our Onboarding API Lifecycle

The process of modeling an API involves collecting all the information you need in order to properly describe the tasks you want to accomplish, the data needed to get that work done, and the process or workflow needed to support all the tasks. For the most part, this is common-sense data collection and description, but it's important to put together a kind of method or repeatable set of steps that you can use each time you want to model a new API.

It's likely that you'll be asked to help a team turn a manual process into a automated one using an API. Sometimes APIs or applications already exist. They do one or more parts of the expected work but don't work well together. Sometimes just a few parts are missing. Quite often, a company has a fully functioning API, but the workflow or data requirements have changed, and it's your job to go in and help update an existing API to meet the company's new needs.

No matter the reasons why, the real work of modeling the API starts with asking lots of questions and collecting lots of information. Then you need to turn that information into some type of document that everyone can understand, review, and then approve before you get too far into the actual process of building the API.

An easy way to do this is to use a simple pattern of discovery that involves two key steps you can repeat again and again until you've gathered enough information to work on the API directly:

1. *Map out the API workflow and identify the internal cycles.* This API workflow document helps you get the big picture of what you're working on. The internal cycles document helps you focus on the details of each small step in the overall workflow.

2. *Write an API story document.* The API story is just that—a narrative document that explains just what the API is meant to do, along with the data properties and actions that “animate” that story.

Notice that step 1 is focused on gathering and organizing all the details of the API. Step 2 is where it all comes together in a simple narrative document. I’m presenting these as two discrete steps, but most of the time you’ll end up working on both of them at the same time, moving back and forth between the two as you learn more about the internal details and clarify the big picture. However, keeping these two tasks separate (at least in your own head) can help make sure you don’t get too bogged down in explaining the big picture at the start, as well as help make sure you have time to focus on important details.

By making sure to gather all the details as you find them *and* document a high-level story, you’ll end up with a solid representation—a *model*—of the API you’re tasked to design and build.

Let’s take a closer look at each of these steps.

Mapping the Overall API Flow

Mapping out the overall flow of the API is a great way to get the big picture out where everyone can see it and agree on the goals. Remember, APIs exist to solve real business problems. When you outline the API in this big-picture kind of way, it should be clear just what problem this API solves and how it will be used.

In our case, we’ve been tasked with creating the company’s Onboarding API—the API that makes it possible to safely and easily add new companies and accounts to the system. This work is currently done by three different people:

1. The company representative collects basic information from the potential company and enters it into the system.

2. The account manager is notified and sets up basic information about the sales account for that company.
3. The support supervisor then schedules one or more appointments to contact the company to follow up on requests, help them find what they need, and so forth.

I usually learn this information by asking questions of people in the company, and sometimes by reading some form of documentation or checklist document that tells me, the developer, what it is I'm supposed to be working on. Often I need to do both: read the documentation *and* talk to the individuals involved—the people really doing the work. It's important to ask questions since sometimes the people doing the work have more insight into what's going on than the people writing up the programming request documentation. This isn't a bug or a failing in the company. Most organizations have this pattern, where the people closer to the work have more information than those further from the action.

Asking Questions

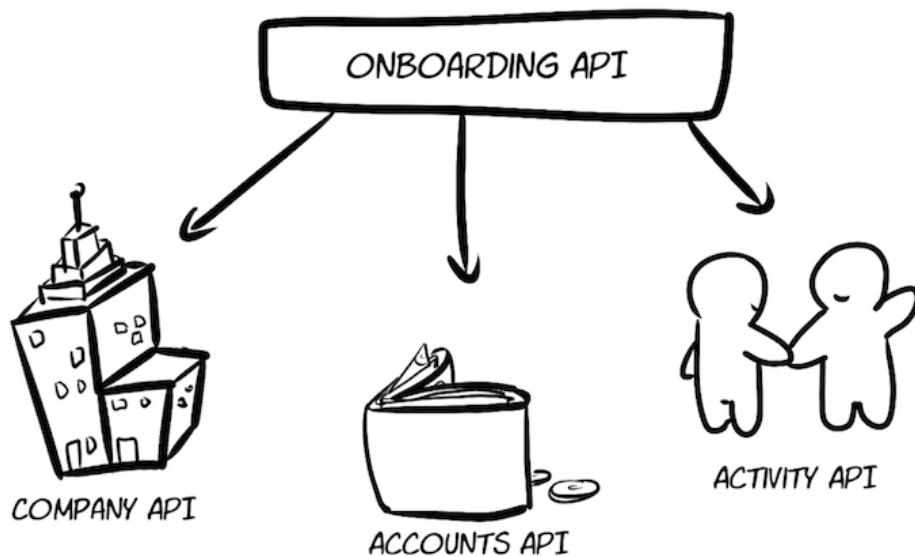
In the 2013 revised and expanded edition of *The Design of Everyday Things*, Donald Norman says, “Good designers never start out by trying to solve the problem given to them: they start by trying to understand what the real issues are.” This results in designers making the problem *larger* and can sometimes frustrate managers and other stakeholders. They want solutions, not more problems.

However, it's very important to ask lots of questions. In my experience, people don't always know how to explain what they need and instead just ask for what they think they want. These are not always the same thing. A technique I use to help people explain their expectations in more detail is to use what's known as Toyota's “Five Whys.” Taichi Ohno, in his 1988 book, *Toyota Production System*, explains: “By asking ‘why’ five times and answering each time, the real cause of a problem can be discovered.”

With this information as a start, I can create my first attempt at describing the overall flow of the Onboarding API:

`create_company_record -> create_account_record -> create_activity_record`

I usually create a simple diagram, like the following diagram, and share it with all the parties I've been talking with. This document becomes part of the project assets—the collection of material used to model the API. And some version of this diagram may end up as part of the API documentation at some future point.



It may seem trivial at this stage to share such a simple drawing, but creating one can be very helpful. Sometimes I learn from one of the people I'm talking to that a major step has been left out. For example, after sharing my onboarding diagram with the support supervisor, I might be told, "You know, Jan usually gets a copy of this and uses that to enter data into the credit system." Now I know there's an important part of the story I'd missed earlier. It always pays to share your information early and often!

In our example, these three major points in the process (`create_company_record`, `create_account_record`, `create_activity_record`) are all steps we'll need to complete our Onboarding API solution. Now, with the

big picture in hand, we can start to focus on the details of each step along the way. That means identifying the internal cycles hidden within each of these three steps.

Identifying the Internal Cycles

Each of the three steps defined in the previous diagram has one or more internal steps or cycles that need to be discovered, defined, and documented. These internal cycles are often hidden from typical users and/or are just related to the way computers work and don't appear as part of a set of instructions given to a human when they want to perform the same task. By discovering and exposing all the hidden details, you can gather the information about both the data and the workflow that will be needed to actually build the API.

For example, the first part of our Onboarding API workflow is called `create_company_record`. What does that really mean? Often you need to collect data about the company (name, address, contact email, phone number, and so on). You also often need to create some data points not supplied by the company, such as a unique internal ID, the creation date of the data record, the status of the company in our system (`pending`, `approved`, and so forth), or some other bits of information. Some of the information will be required and some will be optional. Some might be *conditional* too. For example, if the company record was added during a designated promotion, you might be required to add the promotion code to the company record. In other cases, you can leave that field blank. The `promotion-code` field is conditional upon the date when the company record was added to the system.

It's the job of the API designer to discover all these details—the data points and the rules associated with them—and document them in a way that everyone involved understands. Usually you can use an existing document as a starting point. In our case, the company representatives have a

company form that's filled out each time they add a new company, similar to the form shown here:



Company Form

Captures all important data for adding a new company to BigCo, Inc.

Company Information

Company Name	Smithy Worx, Ltd
Full Address	123 Main St. Byteville, MD 12345-6789
Telephone	123-456-7890
SMS Number	(If different than telephone)
Email Address	smithy-worx@example.org (optional)

For Internal Use

Company ID	SMI-2019-1001
Date Created	October 1, 2019
Company Status	Approved ▾

This document is a good start, but it's not always enough to capture all the detail and rules for filling out the form. Often the people working with the form have memorized what's required, what's optional, and what's conditional, and no single document or set of instructions may capture all the information. It's the API creators' job to handle that. I usually end up going through any forms or instructions with people who work on the process several times in order to make sure I capture all the information needed.

In cases where a working API already exists, we may be able to use API documentation as a guide instead of using a printed form. But it's important to remember that API documentation may not be complete or up-to-date. You should still spend time talking to the people involved in the process. That might mean both developers working on the existing API or solution as well as people using the solution every day.

As you can see from the company form, just a few fields need to be collected when adding a new company to the BigCo system:

- **Company Name**: The name of the company
- **Full Address**: The complete address, including street, state/province, postal code, and country
- **Telephone Number**: The phone number for the company contact
- **Email Address**: The email address of the company contact

You can also tell from the form that **Email Address** is an optional field. Also, the **smsNumber** field is conditional. You only need to supply it if the **SMS Number** value is different than the **Telephone Number** value.

Also, some fields on the form appear in the *For Internal Use Only* section of the form. These are supplied by the company representative when the form is first filled in. The fields are as follows:

- **Company ID**: A unique value for this company in the system
- **Date Created**: The date the company form was first filled out
- **Company Status**: The initial status of the company in the system

When you look at the completed form you can see that the **Company ID** field is a combination of the first three letters from the company's name and the date the form was first completed. This is probably generated automatically by the API. You'll also notice that the **Company Status** field is conditional. It defaults to **APPROVED** unless some other value is selected from the drop-down list.

By reviewing this form and asking some questions, we're able to come up with a single document that briefly describes all the steps, the data names,

and the required, optional, and conditional fields. This is what I call an *internal cycle*, or IC, document.

Now that we've determined all the details of the `create_company_record` process, we're ready to write it up in a more detailed fashion. Normally, we would also need to do this same discovery work for the `create_account_record` and `create_activity_record` processes too. However, we'll use documents already completed for the Account and Activity internal cycles. You can find a full set of the IC documents mentioned in this chapter, as well as all the BigCo company forms and initial workflow diagrams, in the code download files on the book's web page.^[25]

Stories, Workflow, and Cycles



The process of modeling your API involves a repeating loop of writing stories, mapping workflow, and documenting internal cycles. At this point, work on each element will influence what appears in the other parts of your model. You may find you need to go back and forth between documents to keep things aligned. That's normal. At this early stage, keep in mind that nothing is set in stone. The more you discover and document, the better off you'll be when it comes time to actually design and build your API.

Now, armed with the details for all three of our major Onboarding API steps, we're ready to put it all together in a clear, concise form—your API story.

Writing the API Story

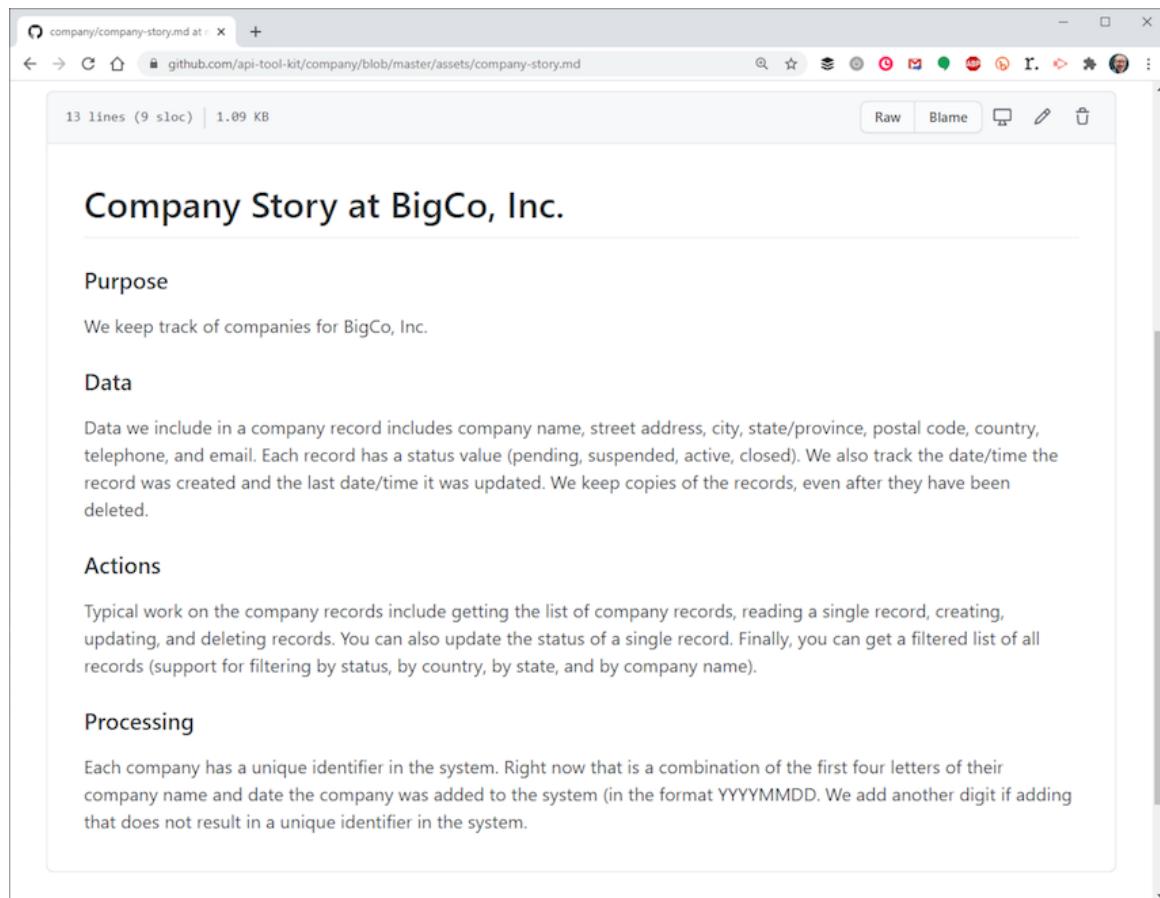
Many programmers and designers have been exposed to the notion of “stories” to help build services and interfaces. A common example of this practice is found in the “user story” concept that comes from the early days of the agile software community. A good explanation of user stories and how they can be used comes from Mike Cohn of Mountain Goat Software.

He defines user stories as “short, simple descriptions of a feature told from the perspective of the person who desires the new capability, usually a user or customer of the system.”^[26] Cohn’s stories typically follow a common template: “As a TYPE-OF-USER, I want SOME-GOAL so that SOME-REASON.” This template approach makes it easy to generate stories that are both short and have perspective. We’ll be borrowing this same approach for our API stories. I won’t go into user stories in this book, but I encourage you to check out Cohn’s book *User Stories Applied* for more on this important topic.^[27]

The API stories I write have up to five sections:

- Purpose: This is a short statement that defines the reason this API exists and what it does.
- Data: This section lists all the data properties (inputs and outputs) involved in fulfilling the API’s purpose.
- Actions: This section lists all the actions the API is expected to accomplish.
- Rules: This section lists any rules the API is expected to enforce. For example, “The `companyId` must be globally unique.”
- Processing: This section lists any processes the API is responsible for handling, such as, “Each time `company` record is updated, a log record will be written.”

The [screenshot](#) is an example API story for BigCo’s Company API.



The screenshot shows a GitHub browser window displaying a file named 'company-story.md'. The page title is 'Company Story at BigCo, Inc.'. The content is organized into several sections:

- Purpose**: We keep track of companies for BigCo, Inc.
- Data**: Data we include in a company record includes company name, street address, city, state/province, postal code, country, telephone, and email. Each record has a status value (pending, suspended, active, closed). We also track the date/time the record was created and the last date/time it was updated. We keep copies of the records, even after they have been deleted.
- Actions**: Typical work on the company records include getting the list of company records, reading a single record, creating, updating, and deleting records. You can also update the status of a single record. Finally, you can get a filtered list of all records (support for filtering by status, by country, by state, and by company name).
- Processing**: Each company has a unique identifier in the system. Right now that is a combination of the first four letters of their company name and date the company was added to the system (in the format YYYYMMDD. We add another digit if adding that does not result in a unique identifier in the system.)

In all API stories, the first three sections (purpose, data, and actions) are required elements. You don't have a complete story without them. The other two sections (rules and processing) may not be needed, depending on the API itself.

Putting It All Together

In the previous sections, we walked through the process of converting a printed form we were supplied by our stakeholder (the company representative) into the essential interaction information we'll need to support our Onboarding API. Although it wasn't covered in the text, the work of collecting the **account** interactions from the account manager and the **activity** information from the support supervisor has also been completed.

At this stage, it's a good idea to create a single document that represents all the major steps in the process and all the data expected to be passed between the steps. This document doesn't need to be very fancy. Often I just use a plain text file with headers indicating a start of the workflow, some arrows to show movement between steps, and data names to show arguments being passed around. Here's an example of the initial workflow we've been working on for the Onboarding API:

```
Home
-> Company
-> CreateCompany(companyName, email, phone, status)
-> CreatedCompany
-> Home

Home
-> Account
-> CreateAccount(companyId, division, defaultDiscount, creditLimit)
-> CreatedAccount
-> Home

Home
-> SalesActivity
-> CreateActivity(companyId, accountId, activityType, dateScheduled, note)
-> CreatedActivity
-> Home
```

Note that in the example, each grouping is a collection of small steps. The three elements of this document are the `company`, `account`, and `activity` interactions. Each of these interactions have smaller steps—the internal cycles mentioned earlier in this chapter. By expressing the overall workflow in these small internal cycles, it's easy to understand what's going on at each point and (as we'll see later) easier to convert this diagram into formal API definition documents and—eventually—into working code.

Another thing you probably noticed is that I slipped something into the workflow here. Actually, I had it in the initial diagram too (refer back to the [diagram](#)). I added a `Home` workflow item. In fact, the `start` item appears

multiple times. This `Home` step is what I call a workflow *anchor* step. It provides an easily identified starting (and resting) point for the overall workflow. I could have written the overall workflow to look like this:

```
Home
->Company
->CreateCompany(companyName, emailAddress, phoneNumber, smsNumber)
->CreatedCompany
->Account
->CreateAccount(companyId, division, defaultDiscount, creditLimit)
->CreatedAccount
->SalesActivity
->CreateActivity(companyId, accountId, activitytype, dateScheduled, note)
->CreatedActivity
->Home
```

In this example, I defined a straight-line workflow from one activity to the next. This looks logical but has some problems. This version of the workflow has the following flaw:

```
CreatedCompany -> Account
```

As this snippet shows, the `CreatedCompany` step “knows about” the `Account` step. That means they have a kind of dependency. In fact, when you look at my second workflow example, all the steps rely on each other. For this to work, every step must work or the whole thing fails. It’s not a good idea to model long workflow chains like this. Instead, it makes sense to break them up into smaller steps. That’s what I did in the first example (see the [code](#)). And I used the `Home` anchor element to make that work. By having each set of internal cycles begin with (and return to) the `Home` element, I can isolate each of the important tasks (`createCompany`, `createAccount`, and `createActivity`). This isolation not only makes the workflow easier to understand, it makes the workflow more reliable and easier to manage once it goes into production.

Now we can store the modeling documents we’ve been working on in our `onboarding-api` folder using Git. First, at the command-line, navigate to the

project folder, create a sub-folder named `assets`, and then navigate to that new folder.

```
mca@mamund-ws:~/onboarding-api$ mkdir assets  
mca@mamund-ws:~/onboarding-api$ cd assets  
mca@mamund-ws:~/onboarding-api/assets$
```

Next, copy the Company Form pdf, the initial diagram, and the Company IC document available in the code download files into your current `assets` folder.^[28]

```
mca@mamund-ws:~/onboarding-api$ cp [path-to-code-folder] .  
mca@mamund-ws:~/onboarding-api$ ls  
company-digram.png company-form.pdf company-ic.txt  
mca@mamund-ws:~/onboarding-api/assets$
```

Now that we have a more complete set of API creation documents (the forms, initial diagram, and the workflow document), it's time to place these files in the `onboarding-api` folder we created in the previous chapter and start the process of managing our project on disk. To do that, we'll use a very handy utility—the Node Package Manager (npm).

Managing Your Project with npm

The Node Package Manager (npm) was created to help NodeJS developers manage the lifecycle of coding, building, testing, and installing a fully featured NodeJs application.^[29] It's also designed to help developers manage the details of dependencies (other node packages used as part of a single application).

First released in 2010, npm was originally developed by Isaac Schlueter, who was inspired by similar language-specific module management tools like PEAR for PHP and CPAN for Perl. Since in this book we'll be writing our code using NodeJS, npm (usually installed whenever you use common install scripts for NodeJS) is a logical choice for managing the modules and other related details of our API project.

However, I use npm as more than a package manager (as it was originally intended). I like to use npm as a *project* management tool. All projects—the work of creating APIs or applications in general—need more than just a list of modules used in that application. You need to handle building the code into something that can be released, and you need to handle testing and remote deployment and/or local installation of the resulting application. I like using npm for this job.

The company you work for may have its own project management tools it wants you to use. Most of the popular editing platforms (integrated editing environments or IDEs) such as Eclipse, IntelliJ, Visual Studio, and others have their own project management tooling too. That includes managing source code files, testing, building, and deployment. As you work through this book, you may choose to use these platform tools instead of npm, and that's fine. The important thing is that you use something that helps you keep everything related to your project organized and provides support for coding, building, testing, and releasing it in a safe and consistent way.

While the npm utility has lots of powerful features, I'll introduce just the basics right now. And I'll build on that information throughout the rest of the book, since we'll use npm to handle the important aspects of managing our project from start to finish.

For now we'll use the npm command-line utility to do the following tasks:

- Create an npm package.
- Use npm to install other NodeJS modules on your machine.
- Edit your npm package file directly.
- Use the `npm start` command to start your local NodeJS application.

Let's start with what it takes to create a new npm package in your `onboarding-api` folder.

Creating Your npm Package

The first step in using npm to manage your API project is to use npm to *initialize* your folder as an npm package folder. You do this, as you might guess, using the `npm init` command. The `init` command will prompt you with a few questions to help you set up your package folder properly. Most of these questions are optional, and it makes sense to answer enough of them to make sure you're well set for using npm to manage your API project.

To start, go to the command line on your workstation and navigate to the `onboarding-api` folder we created in Chapter 2, [Understanding HTTP, REST, and APIs](#). Once you're in your `onboarding-api` folder, you're ready to start setting up your API project management.

Installing npm



If you've installed NodeJS, you should also have a copy of npm installed too. You'll find details on the NodeJS/npm install process in Appendix 1, [Installation Hints](#).

Package Name

To start the process, type `npm init` and press the Enter key. After some introductory text, you'll see the prompt asking you for the `package name` and suggesting `onboarding-api` (the folder name). I like to keep the package name the same as the folder name, so just press Enter here.

```
mca@mamund-ws:~/onboarding-api$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.
```

See `'npm help json'` for definitive documentation on these fields and exactly what they do.

Use `'npm install [pkg]'` afterward to install a package and save it as a dependency in the `package.json` file.

```
Press ^C at any time to quit.
package name: (onboarding-api)
```

Version

Next, you'll be asked to supply a `version` for the package; `1.0.0` is a suggestion. I like to start all my projects as `1.0.0`. Some people like to use a version number other than `1.0.0` when they first start out—your company might have rules about this—and that's okay too. For now, press Enter to accept the suggested value.

```
version: (1.0.0)
```

Description

When prompted for the package `description`, enter something short and informative. For this project, enter `BigCo Onboarding API Project` and press Enter again.

```
description: BigCo Onboarding API Project
```

Entry Point

The next prompt suggests `index.js` as the `entry point` for your project. This is the file that, by default, will be executed when you use the `npm start` command. We can just press Enter here.

```
entry point: (index.js)
```

Test Command

The `test command` is a string that can be used to kick off a suite of tests to validate your project code. We will cover testing later in the book; for now you can just press Enter here. That will result in a default string being added to the package file that will be displayed when you type `npm test`. We'll ignore that string for now.

```
test command:
```

Git Repository

Because we're running npm within a folder already initialized as a Git repo, you'll now see a prompt asking you to supply the pointer to your `git repository` and you'll see a suggested string that matches the actual Git repo address we created in the last chapter. You can press Enter here to accept the default string.

```
git repository: (https://github.com/mamund/onboarding-api.git)
```

Keywords

The next prompt you'll see is `keywords`. This is an opportunity to enter some words or phrases that can be used as search terms when you (or other people) are looking for a package. For now, just enter the following keywords (each one separated by a comma): `api, onboarding, bigco`.

```
keywords: api, onboarding, bigco
```

Author

Next you'll see a prompt asking you for the `author` information. For npm, the `author` is always assumed to be one person (not a group), so I typically enter

my name and optionally an email and/or web URL. For now, let's enter a fictional author and email. We can modify this later. Type in **Jan Devsmith <jdevsmith@example.org>**. Note the email value is wrapped in the **<** and **>** characters.

```
author: Jan Devsmith <jdevsmith@example.org>
```

License

This field will hold the software license information for the package. The npm utility provides a default value of **ISC**. This is a license from the Open Source Initiative.^[30] Your company/team may have a value they want here, so check with them. For now, let's just press Enter to accept the default.

```
license: (ISC)
```

Is This OK?

Once you press Enter on the **license** value, you'll be presented with a list of all your settings formatted as a JSON string and one final prompt asking you, **Is this OK?** When you press Enter and accept the default value of **yes**, this information will be written to disk and stored in the **package.json** file.

About to write to /onboarding-api/package.json:

```
{
  "name": "onboarding-api",
  "version": "1.0.0",
  "description": "BigCo Onboarding API Project",
  "main": "index.js",
  "scripts": {
    "test": "echo |\"Error: no test specified|\" && exit 1"
  },
  "repository": {
    "type": "git",
    "url": "git+https://github.com/mamund/onboarding-api.git"
  },
  "keywords": [
    "api",
    "onboarding",
```

```
    "bigco"
  ],
  "author": "Jan Devsmith <jdevsmith@example.org>",
  "license": "ISC",
  "bugs": {
    "url": "https://github.com/mamund/onboarding-api/issues"
  },
  "homepage": "https://github.com/mamund/onboarding-api#readme"
}
```

Is **this** OK? (yes)

After pressing Enter this last time, you can now check the file list in your **onboarding-api** folder: you should see a new file named **package.json**. This is the file we just created, the one we'll be using as our main project control file throughout the rest of the book.

Note that, along with all the strings you supplied, two additional settings in the **package.json** file appear at the bottom: the **bugs** setting and the **homepage** setting. Since we're working with a project in a Git folder that's connected to GitHub, the npm utility was able to use GitHub conventions to fill in these added settings.

The addition of the **bugs** and **homepage** settings in your **package.json** file means that you can now use two additional npm commands. When you type **npm docs**, the utility will open up your local web browser to the page listed in the **homepage** setting of your **package.json** file. When you type **npm bugs**, your browser will open up to the web page listed in the **bugs.url** setting in your **package.json** file. Since we're using GitHub to host our API project, you'll see GitHub pages in your browser when you use these npm commands.

Finally, if your **package.json** file doesn't look quite like this, don't worry; we'll get a chance to edit this file a little later in this chapter.

Installing an npm Package

The npm utility is quite powerful. One of its most important powers is the ability to find and install NodeJS packages created by other developers. This is the main reason npm was created, of course, so let's try that out.

To see how npm works when installing packages, we'll use it to install a simple (and somewhat silly) package that produces cool faces drawn using ASCII characters. To do this, open up a command-line window, navigate to your **onboarding-api** folder, and enter the following on the command line:

```
mca@mamund-ws:~/onboarding-api$ npm install --save cool-ascii-faces
```

After a few seconds, you should see a response that looks something like this:

```
npm notice created a lockfile as package-lock.json. You should
commit this file.
+ cool-ascii-faces@1.3.4
added 12 packages from 9 contributors and audited 12 packages in 6.421s
found 0 vulnerabilities
```

“Wait!” you might say. “This response shows that 12 packages were added, not just one!” And you would be correct. One of the features of npm package management is that when you install a package using npm, any packages that module depends on will also be installed.

You can also see that npm checks the installed modules for known vulnerabilities and, if needed, will help you identify and fix them too.

In this example, the package name is **cool-ascii-faces**. The **--save** argument tells npm to save the module within the current project and update the **package.json** file to note that this project now has a *dependency* on the **cool-ascii-faces** module. You can confirm this by loading the **package.json** file into your favorite text editor and looking for the **dependencies** node in the JSON document. Mine looks like this:

```
"dependencies": {
  "cool-ascii-faces": "^1.3.4"
```

```
}
```

Along with the name of the package, you'll find a version number indicator. We don't have time to go into the details of how versions work for dependencies here, but you can check out the npm documentation for details if you're interested.

You can see all the packages added to your project by doing a directory listing of the `node_modules/` folder in your project. Mine looks like this:

```
mca@mamund-ws:~/onboarding-api$ ls node_modules/
cool-ascii-faces  inherits  process-nextick-args  safe-buffer
core-util-is       isarray   readable-stream      stream-spigot
string_decoder    util-deprecate                  xtend
```



```
mca@mamund-ws:~/onboarding-api$
```

We can test our newly installed module by writing a tiny NodeJS script that uses it. To do this, open your text editor and enter the following:

ch03-modeling/cool.js

```
var cool = require('cool-ascii-faces');

console.log(cool());
```

Then save the file in the project folder using the name `cool.js`.

At the command line type the following: `node cool.js`. When you do, the script should respond by emitting a cool face drawn using ASCII characters. Mine looks like this:

```
mca@mamund-ws:~/onboarding-api$ node cool.js
(◦‿◦)
mca@mamund-ws:~/onboarding-api$
```

Typing `node cool.js` at the command line is fine, but it can get a bit tedious. There's a way to make this easier; we'll get to that in a minute. But first we need to learn to edit the `package.json` file directly.

Editing Your npm Package

Since the `package.json` file is just a text file holding a JSON document, you can use any text editor to open and modify that file. You just need to be careful not to edit in a way that results in a malformed JSON document or ends up creating an invalid `package.json` file.

For our example, we'll edit the `keywords` associated with the project. Open the `package.json` file in your favorite text editor and add three new keywords to the list that already appears in the document. These three keywords are `cool`, `ascii`, and `faces`. After adding the new keywords, save the file. When you're done, the `keywords` setting of your document should match the following:

```
...
  "keywords": [
    "api",
    "onboarding",
    "bigco",
    "cool",
    "ascii",
    "faces"
  ],
...
...
```

After updating the keywords, leave the `package.json` file open in your editor so that we can do one more thing before we complete our initial tour of the npm package management systems.

Starting Your npm Package

I mentioned earlier that there's a way to make launching NodeJS apps easier when using the npm utility: the `npm start` command. This command will automatically start your app using the default document, no matter what name it is. This is especially helpful for other people using your project who might not be able to tell which JS file is the one to use when starting your

app. To take advantage of this feature of npm, we need to edit the `scripts` settings in the `package.json` document.

When you open the `package.json` file in your editor, the `scripts` settings should look like this:

```
...
"scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
},
...
...
```

We need to add a `start` element with the value of `cool.js`. When you've finished doing that, the `scripts` element should look like this:

```
...
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "start" : "node cool.js"
},
...
...
```

The `start` element now contains the same text we were typing at the command line before (`node cool.js`). That's all there is to it. Once you've completed the changes, save the `package.json` file, open up a command-line window in the `onboarding-api` folder, and type `npm start`. This should fire off the sample script we created earlier and display another cool ASCII face. Your output should look similar to the following:

```
mca@mamund-ws:~/onboarding-api$ npm start  
  
> onboarding-api@1.0.0 start /onboarding-api  
> node cool.js  
  
{ }  
mca@mamund-ws:~/onboarding-api$
```

Now that we've seen how to use the npm utility to create and edit our own `package.json` file, it's time to commit our project changes to Git and then update our remote GitHub repository to close out this chapter's work. But before we do that, we need to update our local Git settings.

Updating Git Settings for npm

Now that we're using npm as our project manager, we need to modify our Git settings to make sure it ignores some npm-related files that we don't want to keep in our GitHub repository. What we need to do is to tell Git to *ignore* some of the files on our disk. To do this, we need to create a file in the root of our project folder named `.gitignore`, and we need to edit it to look like the following:

```
ch03-modeling/.gitignore
```

```
node_modules/
*.log
*~
.env
```

Does `.gitignore` Already Exist?



If you already have a `.gitignore` file in the root folder of your Onboarding API project, make sure the three entries shown are also in your existing `.gitignore` file.

After editing your `.gitignore` file, save it to disk. Now you're ready to use Git to update your local files and ship those changes to your remote GitHub repository, which should look like the following:

```
mca@mamund-ws:~/onboarding-api$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
```

```
Untracked files:
  (use "git add [file]..." to include in what will be committed)
```

```
.gitignore
cool.js
package-lock.json
package.json

nothing added to commit but untracked files present (use "git add" to track)
mca@mamund-ws:~/onboarding-api$ git add --all
mca@mamund-ws:~/onboarding-api$ git commit -m"add package mgmt files"
[master 164ecf2] add package mgmt files
 4 files changed, 129 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 cool.js
 create mode 100644 package-lock.json
 create mode 100644 package.json
mca@mamund-ws:~/onboarding-api$ git push origin master
Counting objects: 7, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (6/6), 1.85 KiB | 0 bytes/s, done.
Total 6 (delta 0), reused 0 (delta 0)
To git@github.com:mamund/onboarding-api.git
 491f4c1..164ecf2  master -> master
mca@mamund-ws:~/onboarding-api$
```

What's Next?

In this chapter, you learned the importance of Donald Norman’s Action Lifecycle as a way to think about how we model our own APIs, and we introduced our own API action lifecycle called the RPW Loop.

We then applied our understanding of this common interaction cycle to model our own Onboarding API lifecycle. And we did this in two main steps. First, we mapped out the API workflow in a simple diagram to help us see the big picture. Second, we talked to people at BigCo, Inc., gathered up their handwritten forms, and created our internal cycle (or IC) documents we’ll use later when we need to actually *design* and then *build* our Onboarding API.

You also learned how to use the npm command-line utility as a project management tool. You learned how to create our [package.json](#) control document, how to install other packages, and how to modify configuration settings.

Finally, we modified our Git settings to ignore some files and folders managed by npm and then used Git to update our local files and send those changes to our remote GitHub repository for later use.

Now that we’ve collected important information on how the BigCo onboarding process works, we’re ready to use that data to create an important asset in any API design—a sequence diagram we can use to validate our understanding with all the people we talked to at BigCo. We’ll also be able to use this diagram as a guide as we work toward building a working API later in the book.

In the next chapter, we’ll graduate from these rough notes to a more interesting rendering of our API. We’ll take what we’ve learned from our exploration and turn that into a visual diagram that others can use as a guide

to understanding the API we plan to build. This diagram will then form the basis for a more detailed machine-readable API description format that acts as a bridge between the design phase and the build phase of our API.

Chapter Exercise

In this exercise, you'll get a chance to work on modeling your own modified version of the Onboarding API. This time we'll include the work of BigCo's credit manager in the overall workflow. We'll also be able to look at the form the credit manager uses for her work and convert that into our own internal cycles document to be used later when we want to update the Onboarding API.

To start, take a look at the credit manager's form and the credit-check story documents located in the code download files for this chapter.^[31] You can use these as a guide to create your own workflow ([credit-check-workflow.txt](#)) document that identifies all the actions needed to support the [credit-check-story.md](#).

You can use the partially filled-in [credit-check-workflow.txt](#) document to record your observations and build up the model for this related service.

Add all these elements (story, form, and workflow documents) to your [onboarding-api](#) project folder in a subfolder named [assets](#), and then commit these changes to your local Git file and the remote GitHub repository.

See Appendix 2 ([Solution for Chapter 3: Modeling APIs](#)) for the solution.

Footnotes

[24] https://en.wikipedia.org/wiki/The_Design_of_Everyday_Things

[25] https://pragprog.com/titles/maapis/source_code

[26] <https://www.mountaingoatsoftware.com/agile/user-stories>

[27] <https://www.mountaingoatsoftware.com/agile/user-stories>

[28] https://pragprog.com/titles/maapis/source_code

[29] <https://www.npmjs.com>

[30] <https://opensource.org/licenses/ISC>

[31] https://pragprog.com/titles/maapis/source_code

Copyright © 2020, The Pragmatic Bookshelf.

Chapter 4

Designing APIs

One of the challenges of designing APIs is that so many results are possible. Design, like art, is pretty much in the eye of the beholder. What some find beautiful or correct, others find lacking. If you get a group of five API developers together to review the same user interview information we gathered in the previous chapter, you’re likely to end up with five different API designs. And that’s normal. We don’t all need to agree on the exact details of every API you and your company release. However, we do need to agree on some things and on a general process to use to design the APIs.

In this chapter, we’ll explore the notion of design thinking using patterns like “jobs to be done,” and we’ll look at a simple API design method you can apply to make sure your API designs are consistent. Finally, you’ll learn how to render your API designs with sequence diagrams to help visualize the API before you start writing the code.

The Power of Design

Art, architecture, and design all rely on the unique traits of the individual in order to turn out the final result—the building, the canvas, the musical score, or the API. Even though the final results rarely look the same from person to person, artists and designers often share a similar method or style when they go through the process of creating their work. And it's this process or method that can be described and standardized. When we all use the same general method, even when we come to different final results, those results are very likely to be compatible—they work together easily. This is the power of a good design process. It allows people to continue to think for themselves, to contribute in unique ways, and still come up with results that are compatible across the organization.

Creating compatible solutions is an important element of a healthy API program. In most companies, the API program will span many years, have lots of different people involved, and over time use several different technologies and products. Throughout all those changes, one of the things you can keep as a constant is your design process or method. By teaching your developers the same design method, you gain consistency without overconstraining their creativity.

A good design method also doesn't assume any single technology stack or tool set. That means you'll be able to grow an API program over time that's built on the power of consistent design methods and not reliant on any one API format or protocol or other technical elements. In fact, good design process makes it easier to change tooling over time without introducing incompatible implementations.

So design helps you bring consistency and compatibility to your API program. But how does that happen? Two key elements of any successful design program are these:

- Design thinking: The way people think about the process of designing an API
- Jobs to be done: A focus on the tasks your API users are trying to accomplish

Before jumping right into our API design method, let's take a moment to review design thinking and jobs to be done in a bit more depth.

Design Thinking

The phrase “design thinking” is used quite a bit in product design to identify a point of view and a collection of practices that help people think differently about the way they design and build products. Tim Brown, one-time CEO of the design firm IDEO, describes design thinking as “a design discipline that uses the designer’s sensibility and methods to match people’s needs with what is technologically feasible and what a viable business strategy can convert into customer value and market opportunity.”^[32] The key words in that quote are “match people’s needs” and “viable business strategy.”

While we may not think about it much, APIs are really meant to do both those things. They need to match people’s needs and they need to support a viable business strategy. The people we need to pay attention to when designing APIs are often not just end users but also developers. Knowing your audience well is key to creating APIs that do a good job of solving the problem. For example, designing APIs for internal use by your own team is different than designing APIs for developers working for your customers or partners. Each is likely to have a different point of view, different set of skills, and different priorities.

Creating something that supports a viable business strategy or goal is important too. It’s not a good idea to set out to design and build a bunch of APIs without knowing there is an audience for them and a business problem

to be solved. Just creating APIs to say you *have* APIs is not a viable strategy. At the same time, building APIs that no one asked for, that no one really needs, is also a bad idea. I've seen API teams work hard to get funding from leadership, then work hard to build and release generic APIs that no one requested, only to find that company leaders complain that the new APIs aren't what they need and thus go unused. This isn't a viable business strategy.

What does it take to build APIs that "match people's needs" and support a "viable business strategy"? That is where the concept of "jobs to be done" can help.

Jobs to Be Done

An important reason to design and build an API is to solve a problem—to "match people's needs." In Chapter 3, [Modeling APIs](#), we worked to discover the problem (onboarding new customers). In our case, the customer onboarding work is a job that needs to be done. That's what Harvard professor and author of the books *Innovator's Dilemma* and *Innovator's Solution* Clayton Christensen calls "jobs to be done."^[33] He says, "People don't simply buy products or services, they 'hire' them to make progress in specific circumstances." Translated into API terms, developers want to make progress (solve problems) and they'll use (or hire) whatever APIs will help them do that.

This means that designing APIs to solve existing problems is essential. Every organization, large and small, has problems that need to be solved. Focusing on them and figuring out how APIs can help is a good strategy for a number of reasons.

First, when you uncover problems to be solved, you're tapping into an opportunity to make things better. When you can create an API that makes work easier, safer, faster, and so on, that's a win. Second, when you complete an API that solves a problem, it's easy to know if you reached

your goal. The truth is that sometimes our APIs don't work as we expected and may not actually do a good job of solving the stated problem. When that happens, we need to learn from our attempt and then move on to design and build an API that *does* solve the problem. When you start with the problem, it's easier to measure your success.

And measuring success is how you make sure your API supports a viable business strategy.

Business Viable

As mentioned earlier, it doesn't make sense to invest a lot of time and money into an API no one really wants or needs. That's why "matching people's needs" is an important element of good API design. Another important element to good API work is making sure the design and implementation work makes good business sense. This can be a challenge since it's common for API designers and developers to be a bit "out of the loop" when it comes to business details. In fact, I know many programmers prefer it that way. But in reality, all of us working on IT projects for our companies have a responsibility to make sure the work makes good sense both technically and business-wise.

While developers aren't often the ones who decide business strategy, we influence what time and money goes into executing on that strategy. In my experience, there's more than enough opportunity to improve the way companies use APIs in their business. In fact, the challenge is not in finding an API opportunity, the challenge is in prioritizing the many existing opportunities. And that's a place where developer teams and API designers can help.

Starting an API project often represents a risk. It requires a commitment of time, money, and personnel. Working on one project can mean *not* working on other API projects. And the more time and money your API project consumes, the less is available for other projects. For this reason, one of the

best things you can do when designing APIs is to reduce the risk incurred. You can do this by building an API that matches people's needs *and* has enough flexibility built in to allow for small changes in the future without big costs. That's a lot to think about. But as time goes on, I've found most API developers get pretty good at balancing matching needs versus anticipating changes. We'll see examples of this trade-off throughout the API design and build we'll be doing in this book.

Basically, those who are tasked with designing and building the APIs—the glue that makes services work well together—have the job of uncovering important needs and figuring out how to implement the APIs in a way that makes sense both now and for the foreseeable future. We started on that path in Chapter 3, [*Modeling APIs*](#), by uncovering the job that needs to be done (the OnBoarding API), and now we need to turn what we learned into a clear, concise API design that developers can use to successfully build the final product.

To make that possible, we'll first use a repeatable method—an API design method—to convert our information into a design. Then we'll take the output of that method and produce a helpful diagram (a sequence diagram) that documents our design and provides developers with added information they can use to actually build the API we need.

First, let's look at the API design method.

The API Design Method

As I mentioned earlier in this chapter, an important way to bring consistency and compatibility to your APIs is to use a common design method or process. Doing this means all API designers go through a similar process, answer the same basic questions, provide recognizable solutions, and end up creating common results. These similarities make it easier for developers and other people involved in the work of building and releasing the API. And when it comes time for others to use your API, they'll recognize similar design details and common implementation elements from other APIs they've already used within the company.

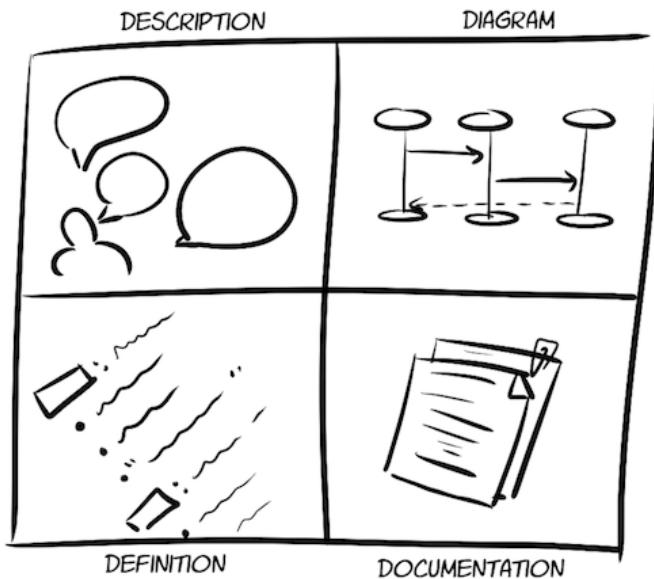
The process I'll describe here is one I've used for many years with good success. It starts where [Modeling Our Onboarding API Lifecycle](#), left off. It assumes you've gathered lots of information from key stakeholders, that you have copies of forms and worksheets, and that you've built up simple flow diagrams and other assets, just as we did in the previous chapter. With all that in hand, it's now time to turn your interviews and information into an API design.

This API design method consists of four steps:

- Descriptor: Review all the names of the data and action elements and, where needed, convert them into the names already understood in your API program.
- Diagram: Convert your notes on workflow and internal cycles into something visual people can understand.
- Definition: Create a machine-readable API definition document that developers can use as a starting point for their implementation.

- Documentation: Write basic reference documentation for both service developers and service consumers (those *using* the API in the future).

These four elements will be covered in multiple places in this book. We'll deal with the first two (descriptors and diagrams) in this chapter and the second two (definitions and documentation) in Chapter 7, [Prototyping APIs](#).



When you're done working through this API design method, you'll have an API that looks and feels consistent with other APIs in your company, one that has a clear diagram and set of documentation that others can use to reason about your API, and a machine-readable document that developers can use to start to build a working version of your API.

Let's get started on our design method as we work through our Onboarding API example.

Onboarding Cycles Document



Throughout the following section, we'll use the Onboarding API Cycles document we created with the [code](#).

Identifying Your API Descriptors

The first step in our API design method deals with identifying the names we'll use for all our data elements and actions. I call these bits of information *descriptors* since they're meant to describe the contents of the data elements (for example, `companyName`) and describe the action a developer would be taking (for example, `approveUser`). While the actual name we use for this isn't too important, what is important is that we'll be using the same name (descriptor) for both *data* and *action* identifiers. Often we spend a great deal of time covering the names of data elements but don't always spend as much time on the names of the actions.

Another important aspect of focusing on the descriptors in our API design is to make sure the actual names we use are well understood. It turns out many companies have their own vocabulary—their own glossary of terms—they use every day. Often these terms take the form of acronyms or abbreviated terms. For example, a customer record might be referred to as a “CustRec” in meetings within the company and everyone would understand what that means. This is a common aspect of all groups of people; even teams within a company may have their own shorthand or unique terms for objects and actions.

Using Schema.org as a Validating Source

The work of picking descriptors for your API is the process of validating that the data elements and actions are provided names that are normal for the target audience using the API. For this reason, I refer to this work as “normalizing the identifiers.” The trick is to find something everyone agrees on to act as the reference list of terms. Often companies already have a published glossary of common terms. Many companies working in the fields of banking, insurance, healthcare, and other large professional marketplaces have a set of terms managed by a standards body in the related field. For example, the banking sector has a set of information standards known as the

Banking Industry Architecture Network (BIAN). And the Fast Healthcare Interoperability Resources (FHIR) standards pertain to the healthcare industry in the United States. Many other science and engineering fields have similar dictionaries or glossaries that can be used as the validating source when normalizing API identifiers.

To keep things simple for our example, I'll use a general validating source called schema.org. It's described as "a collaborative, community activity with a mission to create, maintain, and promote schemas for structured data on the Internet, on web pages, in email messages, and beyond."^[34]

Schema.org is a joint effort by Google, Microsoft, Yahoo, and Yandex to create a set of shared vocabularies that can be used by a wide range of web developers. I'll use schema.org for this API because it's a handy and widely available set of terms. In your own organization, you may already have a set of terms you commonly use and you may also have some vocabularies managed by outside organizations that you can use. The important thing to keep in mind is that you should normalize your API identifiers against one or more of these well-known vocabularies.

Normalizing the Onboarding API Identifiers

Now we're ready to review our current API design document and normalize it against our validating source, schema.org. First, let's look at our Onboarding API workflow document again:

ch04-designing/onboarding-api-workflow.txt

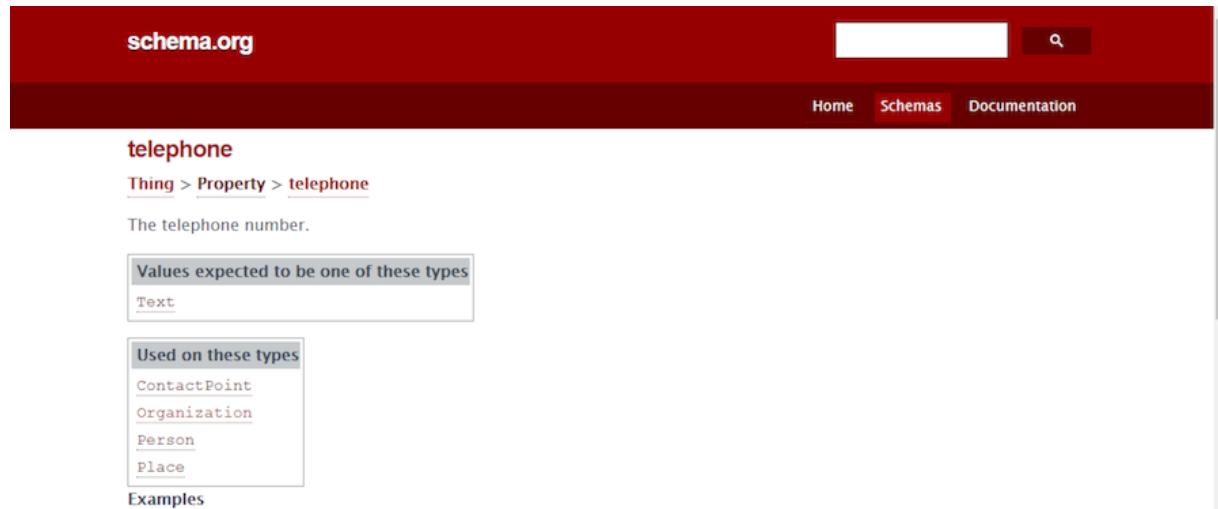
```
## Onboarding API Workflow

Home -> WIP
WIP -> CancelWIP -> Home
WIP -> CompanyData(companyName, address, email, phone) -> WIP
WIP -> AccountData(companyId, division, discount, creditLimit) -> WIP
WIP -> ActivityData(companyId, accountId, activityType, dataScheduled) ->
WIP
WIP -> FinalizeWIP -> Home
```

This workflow document contains names of actions ([CreateCompany](#), [CreateAccount](#), and [CreateActivity](#)). It also has a set of data names associated with those actions ([companyName](#), [address](#), [email](#), [phone](#), [status](#), [companyId](#), [defaultDiscount](#), [creditLimit](#), [accountId](#), [division](#), and [dateScheduled](#)). There are also some additional names in the document ([Home](#), [Company](#), [CreatedCompany](#), [Account](#), [CreatedAccount](#), [Activity](#), and [CreatedActivity](#)).

Technically these names don't identify data points and they aren't actions; they're names of resting places or *states* of the API. You start from the [Home](#) state, perform the [CreateCompany](#) action, and end up at the [CreatedCompany](#) state. We'll be dealing with these *states* quite a bit throughout the book. (Another common name for the state of an API is a *resource*. You'll see me use these two names interchangeably throughout the book.)

Our work now is to validate all the identifiers we're using in our API match the names in the schema.org vocabulary. An easy way to do that is to visit the schema.org website and type our names in the Search box on the home page. For example, when I type the identifier [phone](#) into the schema.org search box, the first entry I get in return is for [telephone](#), as shown in the following screenshot:



The screenshot shows the schema.org website interface. At the top, there's a red header bar with the "schema.org" logo on the left and a search bar on the right. Below the header, the main content area has a dark red background. The title "telephone" is displayed in white. Below the title, the breadcrumb navigation "Thing > Property > telephone" is shown in white. A brief description "The telephone number." follows. Two boxes are present: one titled "Values expected to be one of these types" containing "Text", and another titled "Used on these types" containing "ContactPoint", "Organization", "Person", and "Place". At the bottom left, there's a link "Examples".

To improve the understanding of our Onboarding API, I'll change the design to use [telephone](#) instead of [phone](#). When I go through the rest of the data names and actions from our workflow document and search for matching

elements in the schema.org vocabulary, I get the following updated document:

ch04-designing/updated-onboarding-api-workflow.txt

Home

- > Customer
- > CreateCompany(companyName, address, email, telephone, status)
- > CreatedCustomer
- > Home

Home

- > Account
- > CreateAccount(companyId, division, discount, creditLimit)
- > CreatedAccount
- > Home

Home

- > Activity
- > CreateActivity(companyId, accountId, activityType, dateScheduled)
- > CreatedActivity
- > Home

Validated Names

- * companyName := https://schema.org/legalName
- * address := https://schema.org/PostalAddress
- * email := https://schema.org/email
- * telephone := https://schema.org/telephone
- * dateScheduled := https://schema.org>Date
- * discount := https://schema.org/discount
- * companyId := http://glossary.bigco.com/companyId
- * creditLimit := http://glossary.bigco.com/creditLimit
- * accountId := http://glossary.bigco.com/accountId
- * division := https://schema.org/department
- * activityType := https://schema.org/action
- * Home := http://glossary.bigco.com/Home
- * CreateCompany := http://glossary.bigco.com/CreateCompany
- * CreatedCompany := http://glossary.bigco.com/CreatedCompany
- * Account := http://glossary.bigco.com/Account
- * CreateAccount := http://glossary.bigco.com/CreateAccount
- * CreatedAccount := http://glossary.bigco.com/CreatedAccount
- * SalesActivity := http://glossary.bigco.com/SalesActivty

```
* CreatedActivity := http://glossary.bigco.com/CreatedActivity
```

Note that in my updated flow document, I now have URLs added for every identifier. These URLs point to a definition in the schema.org website. In this way, I've added a reference from the identifier used in the API to a definition found in the vocabulary site schema.org.

You'll also see not every identifier could easily be matched to the schema.org vocabulary. This is a very real possibility for all APIs. Every company has unique terms that don't appear in an external shared dictionary. In this case, you have two options. The first option is to use another existing term as the *reference* source (see [companyName](#) and [division](#) in the example). The second option is to create your own internal glossary, similar to the way schema.org works, and add your unique terms there. A general caution is to be careful not to create too many data names in your own dictionary. If you do, pretty soon you'll have added so many new terms that it's hard to keep up with the changes and the dictionary loses its value. However, it's worth mentioning the action names ([CreateActivity](#)) and state names ([CreatedAcccount](#)) are quite likely to be unique for your company. That's fine. In a healthy vocabulary, you'll have many unique action names but lots of shared data names.

Okay, we've normalized our identifiers. The next step is to turn your workflow text document into something visual. For that we need to concentrate on diagramming our Onboarding API.

Creating Your Sequence Diagram

The second step in the API design method is to create a visual diagram that shows the general sequence of events. This is an important part of any good API design. While it doesn't need to have too many details, such as URLs, HTTP methods, response formats, and so on, it *does* need to give both stakeholders and programmers a clear idea of what actions will be available and what data must be passed back and forth when using the API. Essentially, what's needed is a visual representation of the workflow and internal cycles we've been working on over the last couple of chapters.

There are many ways to create the diagrams. For example, I created a really simple [hand drawing](#) that shows how several services interact. And if you're pretty good with a stylus and drawing tools, you can create some nice freehand drawings. Your current API team may have a visual drawing tool and set of practices you can use too.

The tool I like to use is an online web application called WebSequenceDiagrams (WSD). [\[35\]](#) I like it because it's easy to use, has some nice features, and—even better—has its own API. This last element means I can access the WebSequenceDiagrams functionality directly from my own command-line apps and utilities. I've used it to build handy tools and even integrate the diagramming features into other internal design tools for some of my clients.

Other Diagramming Tools



A handful of diagramming tools are available that are similar to the WebSequenceDiagrams online app. If this app doesn't quite work for you or your team, try other drawing apps to find one that fits your needs. Other apps I've seen used are PlantUML, Lucidchart, and SequenceDiagram.org. Your team may already have an approved tool available for you too.

For this book, we'll walk through the following steps for diagramming our OnBoarding API:

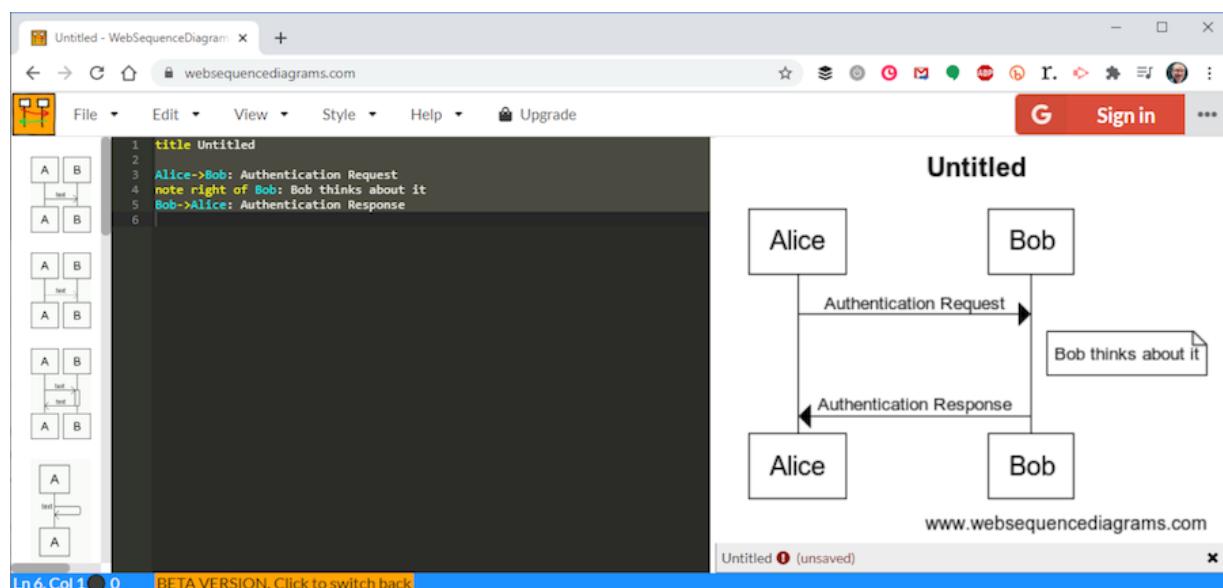
1. Create an account at [WebSequenceDiagrams.com](#).
2. Map out the resource elements of our [OnBoardingAPI](#) workflow.
3. Describe the transitions of our API.
4. Include the action parameters for our API transitions.

I'll also show you a simple utility I've built that lets you do all your diagramming work directly from your local machine's command line. This is an optional utility since you'll be able to do all your work directly from the online application.

Let's get started.

Creating an Account

The first step in our diagramming project is to open our web browser and navigate to <http://websequencediagrams.com>. This is the app we'll use to create our diagrams for the Onboarding API project. When you first load the app, it should look similar to the following screenshot:



The basic parts of the user interface include the diagram templates on the left, a large white rendering space on the right, and a middle section where

you type in text that gets rendered on the right. You might even see a preloaded example diagram when you first visit the site.

We'll get to the drawing part soon, but first we'll create our own free WSD account so that we can save and recall diagrams for later use. To create an account, follow these steps from the home page of the WSD app:

1. Click either the “G Sign In” button (to use Google as your account) or click the three dots next to “Sign in” to create an account using your email and password.
2. If you’re using a Google account, follow the prompts.
3. If you’re using an email account, enter your email address and your password and press the “Create account” button.
4. The dialog(s) should disappear and you’ll be returned to the home screen again.

That’s all there is to it. You now have an account at WebSequenceDiagrams and are ready to start creating diagrams.

Upgrading Your Account

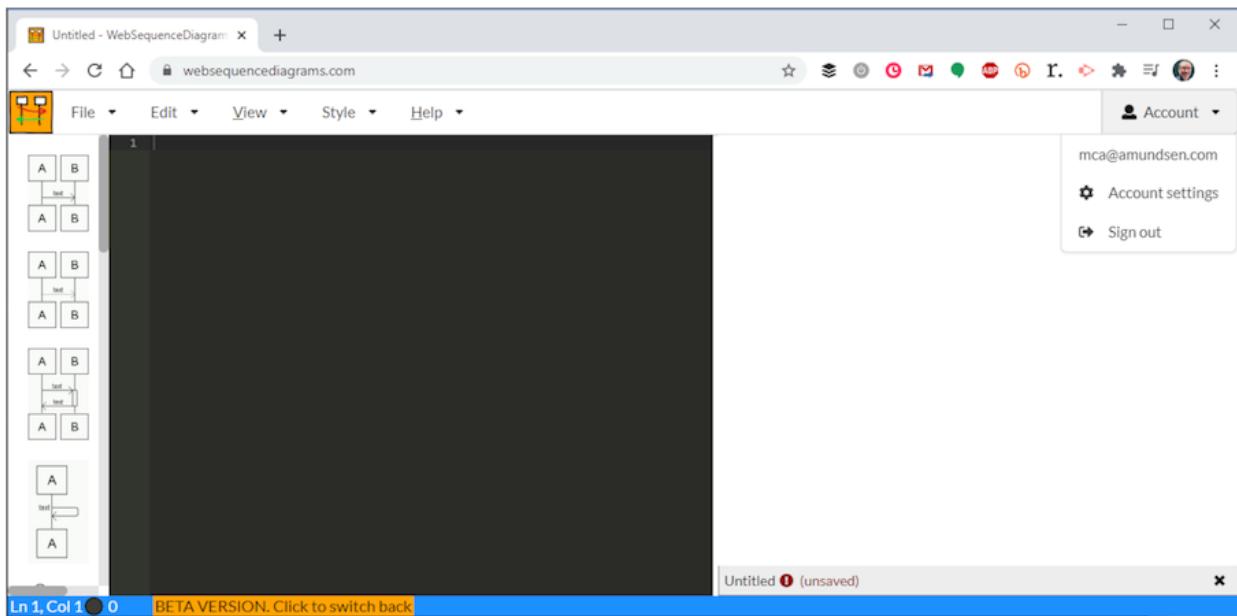


You’ll start getting email reminders to encourage you to upgrade from your free account to a paid account at WSD. You don’t need to upgrade to get all the features you’ll need for this book. I have a paid account because I really like the app and want to support it, but you can decide for yourself if that’s something you want to do.

Creating a New Diagram

Now that you’re logged in, you’re ready to create your first diagram. To do that, use these steps:

1. Click the Files menu item in the upper-left corner of the home screen.
2. Next, locate and click the New File option in the drop-down menu (see the [screenshot](#)).
3. You may see the Unsaved Changes dialog; for now, just ignore that. That should return you to the home screen with a clean page with some sample diagram text.
4. Delete the sample text and we'll be ready to create our own diagram.



Now we are ready to start translating our workflow document into a WSD diagram.

Mapping Resources

To convert our workflow document into a sequence diagram, we'll need to first document the list of resource states in the OnBoarding API project. In the document that we created in the last chapter, the resource states are as follows:

- Home
- Company

- CreatedCompany
- Account
- CreatedAccount
- Activity
- CreatedActivity

You'll also notice that the workflow document shows several internal cycles (we talked about internal cycles in Chapter 3, [Modeling APIs](#)) that each start and end with the `Home` resource state (for example, `Home` to `Company` to `CreatedCompany` to `Home`). These internal cycles are the sequences that we'll place in our diagram.

All WSD documents use the same format. In fact WSD documents have their own “language,” and we’ll learn to use enough of that language to solve our diagramming problem. If you’re interested in learning more about text-based modeling tools and languages, you can do that as an “extra credit” project outside of this book.

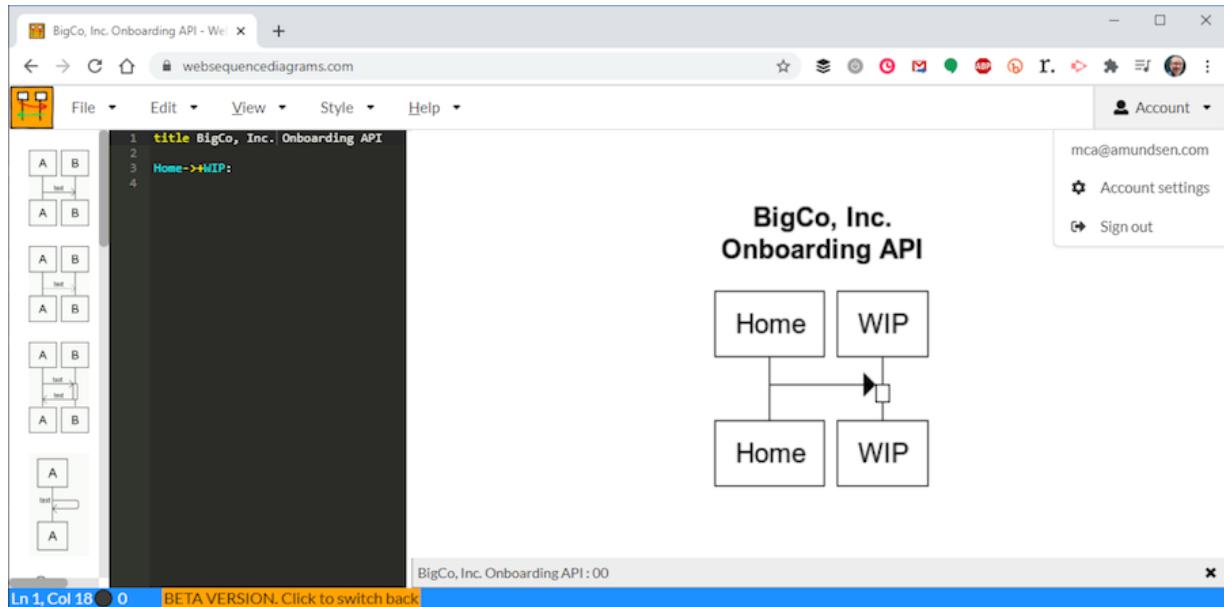
Now, with the online app initialized to create a new diagram, type the following into the center text field of the WSD app:

```
ch04-designing/diagram-onboarding-00.wsd
```

```
title BigCo, Inc. Onboarding API
```

```
Home->+WIP:
```

This should be rendered in the app to look like this:



This diagram shows a workflow that allows users to move from the **Home** resource state to the **WIP** resource state. Another element in the workflow document is the ability to cancel any work in progress (**cancelWIP**). You can include that in the diagram by adding two more lines that look like this:

ch04-designing/diagram-onboarding-01.wsd

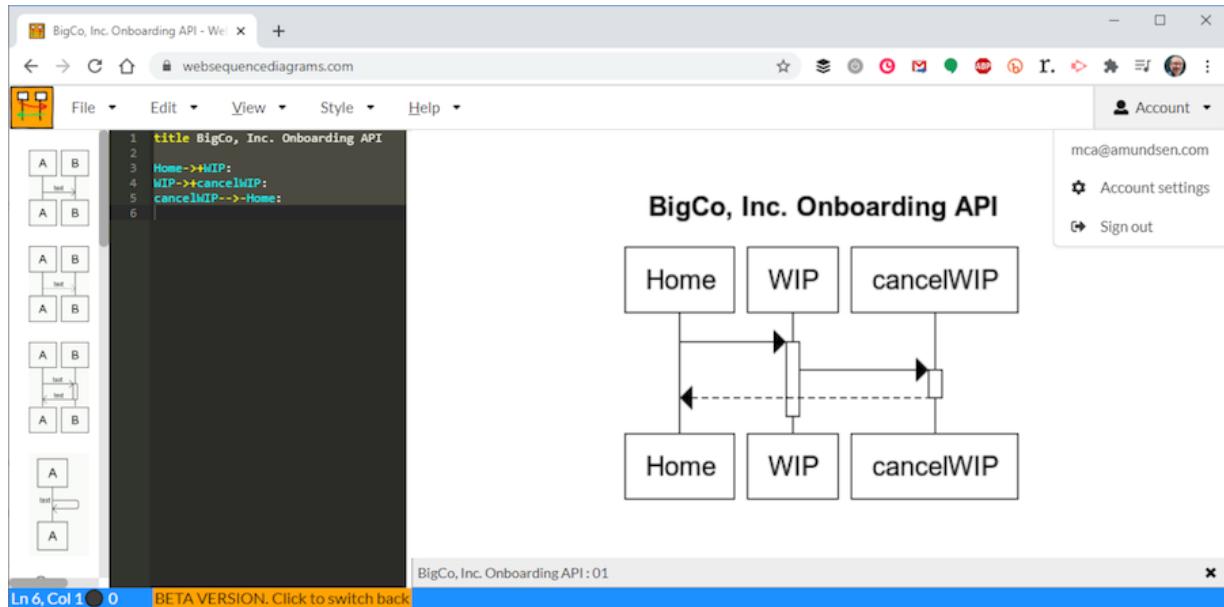
```

title BigCo, Inc. Onboarding API

Home->+WIP:
WIP->+cancelWIP:
cancelWIP-->-Home:

```

In our diagram that looks like the [screenshot](#).



Notice that the workflow that moves *forward* to a new state has an arrow with a solid line. Workflow that moves *back* to a previous state has an arrow with a dotted line. This is a convention we'll use as we build out the rest of the diagram.

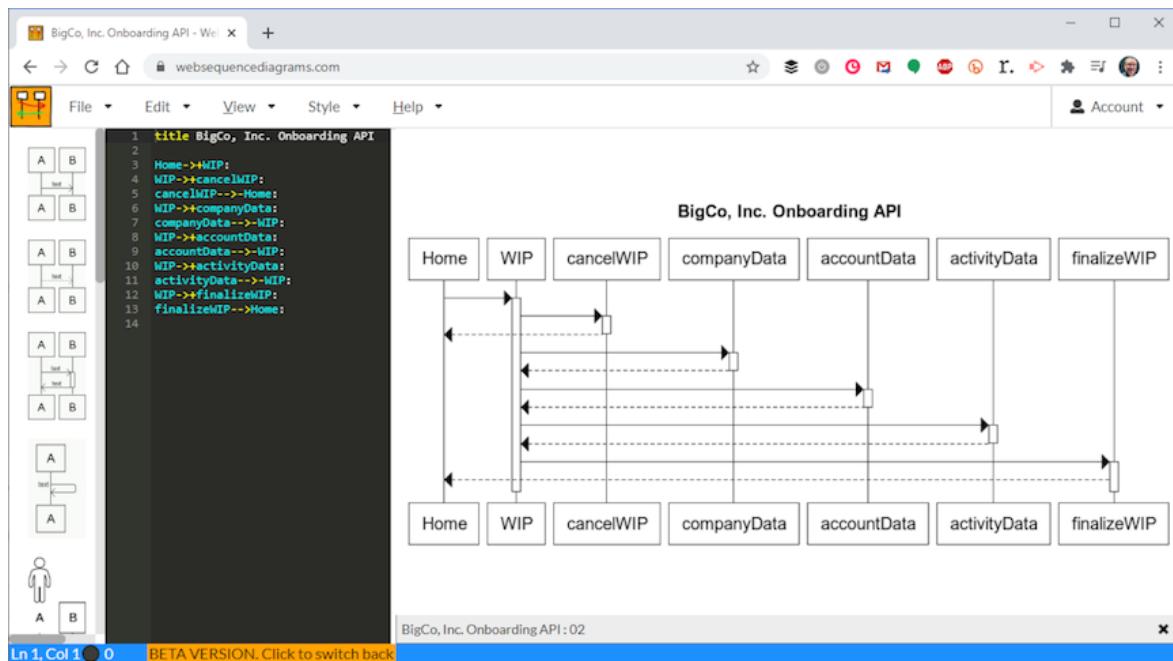
When we convert the rest of the resource workflow from our document into our sequence diagram, the text portion looks like this:

ch04-designing/diagram-onboarding-02.wsd

```
title BigCo, Inc. Onboarding API

Home->+WIP:
WIP->+cancelWIP:
cancelWIP-->-Home:
WIP->+companyData:
companyData-->-WIP:
WIP->+accountData:
accountData-->-WIP:
WIP->+activityData:
activityData-->-WIP:
WIP->+finalizeWIP:
finalizeWIP-->-Home:
```

And the diagram looks like the [screenshot](#).



Now that we've completed the resource portion of the diagram, we should save our work (click the button labeled "Not Saved" in the upper-left corner of the screen). Then we can move on to adding the actions from our workflow document to the diagram.

Adding Transitions

In order to move between resource states, we need some actions. These actions indicate the ability to execute a *transition* from one resource state to another. On typical web pages, these actions are links and buttons. Here are the actions we'll need to add to our diagram:

- startOnboarding
- completeOnboarding
- abandonOnboarding
- collectCompanyData
- collectAccountData
- collectActivityData
- SaveToWIP
- goHome

In the application's editor window, type the action name right after the full colon on each line. That makes it clear which action is used to execute each transition. The following is the WSD text with the action names next to each transition:

ch04-designing/diagram-onboarding-03.wsd

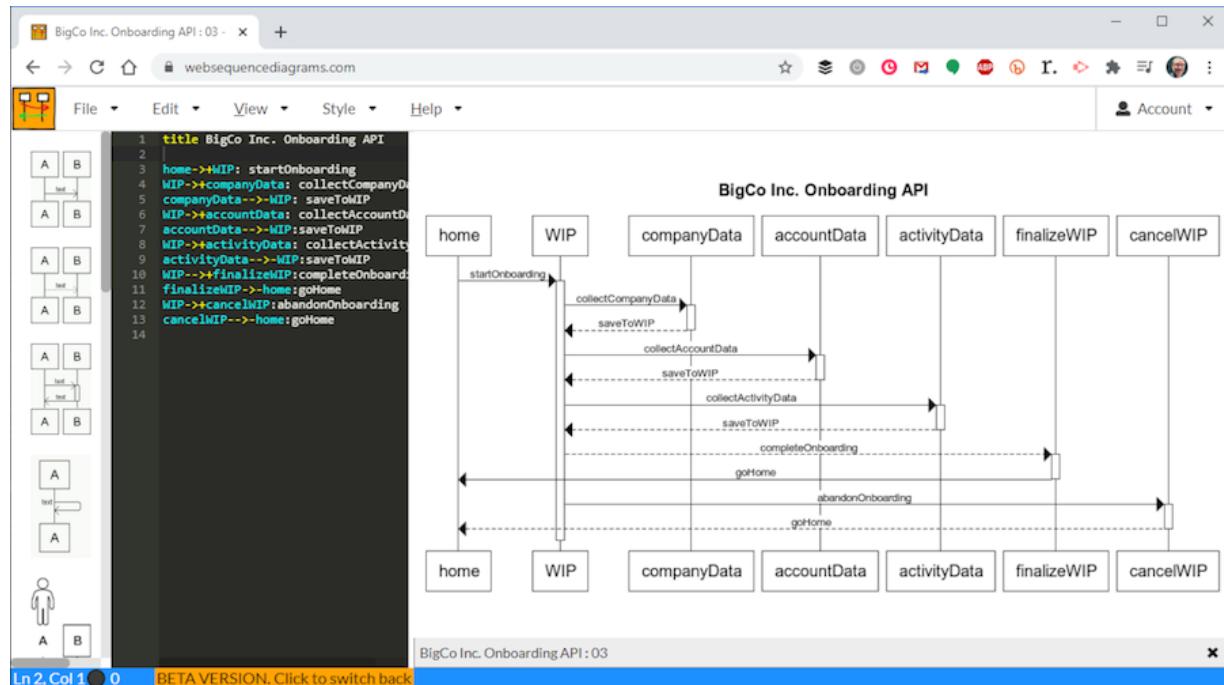
```

title BigCo, Inc. Onboarding API

home->+WIP: startOnboarding
WIP->+companyData: collectCompanyData
companyData-->-WIP: saveToWIP
WIP->+accountData: collectAccountData
accountData-->-WIP: saveToWIP
WIP->+activityData: collectActivityData
activityData-->-WIP: saveToWIP
WIP-->+finalizeWIP: completeOnboarding
finalizeWIP->-home: goHome
WIP->+cancelWIP: abandonOnboarding
cancelWIP-->-home: goHome

```

And here's the rendered diagram:



Now our diagram is taking shape. We can clearly see both the *states* (or resources) and the possible *transitions* (or actions) that this API will support. There's one more step we can take to clarify the API design even a bit more —adding the *parameters* passed back and forth with each HTTP request. That's what we'll take up in the next (and final) section.

Including Parameters

Along with the states and transitions, it's important to include the data elements passed back and forth between the service provider and the service consumer. I'll refer to these data elements as *parameters*. Adding this information helps developers implement a working instance of the API by identifying key data that should appear in the links and forms that make up the HTTP requests and responses. This information is necessary for completing the API, and the sooner we resolve these details, the better it is for everyone. These parameters are the names we resolved against our schema.org dictionary earlier in the chapter (see [Using Schema.org as a Validating Source](#)). What we need to do now is transfer those resolved names from our workflow document into our sequence diagram. Once that's complete, we'll have a detailed diagram that we can share with key project stakeholders (users, developers, and so on) in order to verify our design before we commit it to code.

We'll add the parameters to each transition by including them within parentheses next to the transition name. For example, in the workflow document, the `saveToWIP` transition has six parameters identified:

- `identifier`
- `companyName`
- `address`
- `email`
- `telephone`
- `status`

We can add these to the diagram by doing the following in the WebSequenceDiagrams online editor:

```
companyData-->-WIP: saveToWIP(identifier, companyName, address, email, telephone, status)
```

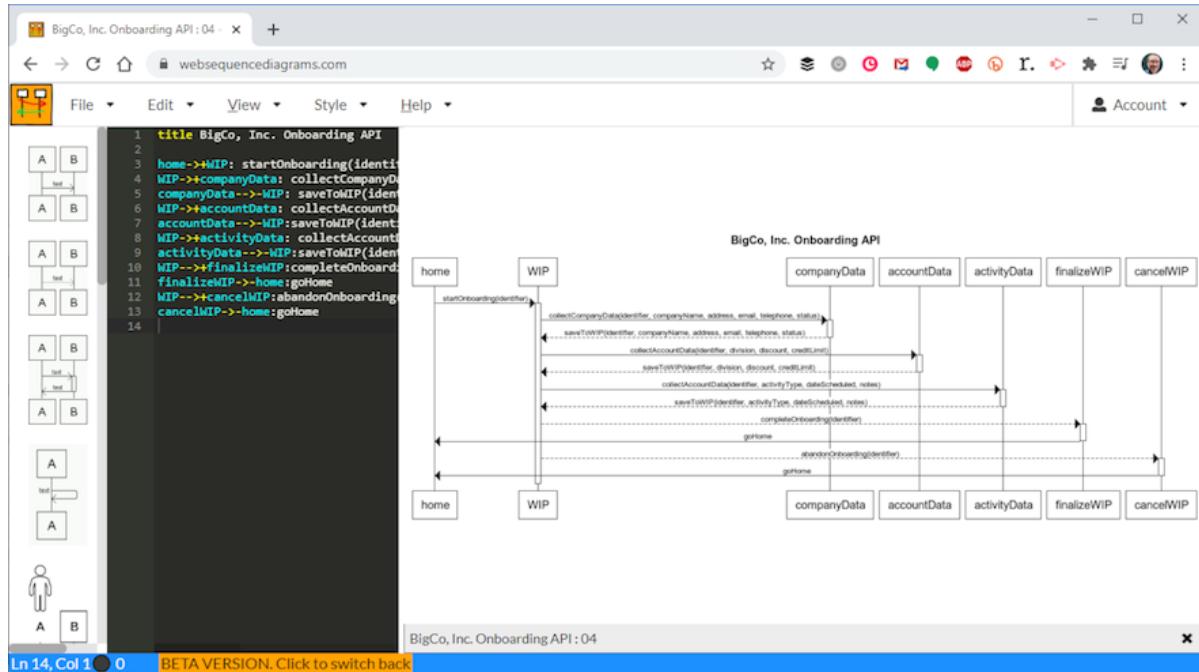
We can then transfer the other parameter identifiers from the workflow document to our diagram. The completed result should look like this:

ch04-designing/diagram-onboarding-04.wsd

```
title BigCo, Inc. Onboarding API

home->+WIP: startOnboarding(identifier)
WIP->+companyData: collectCompanyData(identifier, companyName, address,
email, telephone, status)
compamyData-->-WIP: saveToWIP(identifier, companyName, address,
email, telephone, status)
WIP->+accountData: collectAccountData(identifier, division, discount,
creditLimit)
accountData-->-WIP:saveToWIP(identifier, division, discount, creditLimit)
WIP->+activityData: collectAccountData(identifier, activityType,
dateScheduled, notes)
activityData-->-WIP:saveToWIP(identifier, activityType, dateScheduled, notes)
WIP-->+finalizeWIP:completeOnboarding(identifier)
finalizeWIP->-home:goHome
WIP-->+cancelWIP:abandonOnboarding(identifier)
cancelWIP->-home:goHome
```

The completed diagram now looks like the [screenshot](#).



You probably noticed adding these important parameters tends to clutter up the rendered diagram. Although the diagram is a bit harder to read, I've found it very helpful to other team members to include this information anyway.

That completes our work of transferring our API workflow information into a handy diagram. In the next section, I'll show you a handy command-line utility that makes converting your text-based markup into PNG images quickly and easily.

Using wsd-gen to Generate Your Diagram

When I work on an API project with a customer, we often go through multiple iterations before we arrive at the final agreed-upon API model. That means editing our workflow document, rendering it as a diagram, reviewing the diagram, making adjustments to the workflow, re-rendering, and reviewing again until we've captured everything we need (edit -> render -> review; edit -> render -> review, and so on). To help make this process quick and easy, I use a small utility, [wsdgen](#), to convert the text-based markup for diagrams into a full PNG image everyone can review.

I've included the `wsdgen` command-line utility in the sample code available in the book's code download files.^[36] It's written as a stand-alone NodeJS project that you can easily copy onto your machine and install using the npm utility we covered in Chapter 3, *[Modeling APIs](#)*. After copying the `wsdgen` folder onto your machine, open up a command window within that folder and type the following to install it:

```
npm install -g
```

The `-g` parameter makes sure the utility is available globally on your machine —that means you can execute it from any folder on your workstation/laptop.

Now, move to the folder that holds our latest WSD file and type the following:

```
wsdgen onboarding-api.wsd
```

This will read the WSD file and pass it to the WebSequenceDiagrams online utility, where it will be converted into a PNG diagram, download the resulting rendered diagram, and save it (as a PNG file) in the same folder as the WSD file.

This does essentially the same work as the online editor without having to open up your browser in order to render the text file into a PNG. I use this utility as part of my regular process when working on an API design project and even include it in my build scripts to make sure my sequence diagrams are always up-to-date.

Now you, too, can use it to render your WSD files whenever you wish.

What's Next?

In this chapter, we learned how to convert our initial workflow document into a rendered sequence diagram. That process included using the schema.org online vocabulary as a guide when resolving the data field names in our API to names commonly understood on the web. We also learned about the difference between resource state and transitions and the importance of identifying parameters when we create our API designs.

We covered the basics of the WebSequenceDiagrams online editor and we learned how to use it to generate sequence diagrams and save these diagrams as PNG images. The diagrams will be helpful during the design process and will also aid developers when they convert our design into a working API.

Finally, we used a handy command-line utility called `wsdgen` to generate the PNG diagram from our text-based WSD files, all without having to open up our browser and execute the online editor. This can be used as a quick way to generate updated diagrams and can be included in our build scripts to make sure our diagrams are always up-to-date.

This marks the end of our discussion on designing APIs. Next, we need to use this diagram to complete a machine-readable version of our design called an *API description document*. This description document is the last step on our design journey before we start the work of actually coding our working Onboarding API.

Chapter Exercise

In this chapter, we created the WSD documents for our API workflow. That meant resolving the names and creating a WSD file that can be rendered into a completed PNG diagram.

For this exercise, we'll do the same work using a sample credit-check service. This is a service that our customer (BigCo, Inc.) wants to eventually build and may, at some future point, include in our Onboarding API project. It's a simple service that supports a `checkCredit` method while passing the `companyName` and return a `rating` value between 1 and 10. The story document (`credit-check-story.md`) and other supporting documents are in the code folder for this chapter. The following is the `credit-check-workflow.txt` document as a reference (from the `code/ch04-designing/exercise/before/onboarding/assets` folder):

```
## Credit-Check Workflow

Home -> CreditCheckHistory
CreditCheckHistory -> CreditCheckForm(companyName) -> CreditCheckItem
CreditCheckItem -> CreditCheckHistory
CreditCheckHistory -> Home
```

Using the `credit-check-workflow.txt` as a guide (along with the `credit-check-story.md` and `credit-check-form.pdf` documents from the same folder), do the following:

- Create a detailed workflow document that identifies all the properties referenced in the `credit-check-story.md` document.
- Resolve all the data element names in the updated `credit-check-workflow.txt` document against the schema.org online vocabulary. This will result in an updated workflow document that includes a section on

Validated Names similar to the one shown earlier in this chapter (see [Normalizing the Onboarding API Identifiers](#)).

- Use the `credit-check-workflow.txt` document as a guide in creating your own WSD document (`credit-check-diagram.wsd`). This should look similar to the one shown earlier in this chapter (See [Creating a New Diagram](#)).
- Finally, use the `wsdgen` command-line utility to generate a sequence PNG diagram from the `credit-check-diagram.wsd` file. This should result in a diagram that looks similar to the ones shown earlier in this chapter.

See Appendix 2 ([Solution for Chapter 4: Designing APIs](#)) for the solution.

Footnotes

[32] <https://hbr.org/2008/06/design-thinking>

[33] <https://www.christenseninstitute.org/jobs-to-be-done>

[34] <https://schema.org>

[35] <https://www.WebSequenceDiagrams.com>

[36] https://pragprog.com/titles/maapis/source_code

Chapter 5

Describing APIs

This chapter focuses on turning the design information we gathered in the previous chapter into a comprehensive, machine-readable format that can be used as a guide for creating a working instance of the API itself. The same information can be used to generate basic (low-level) human-readable API documentation.

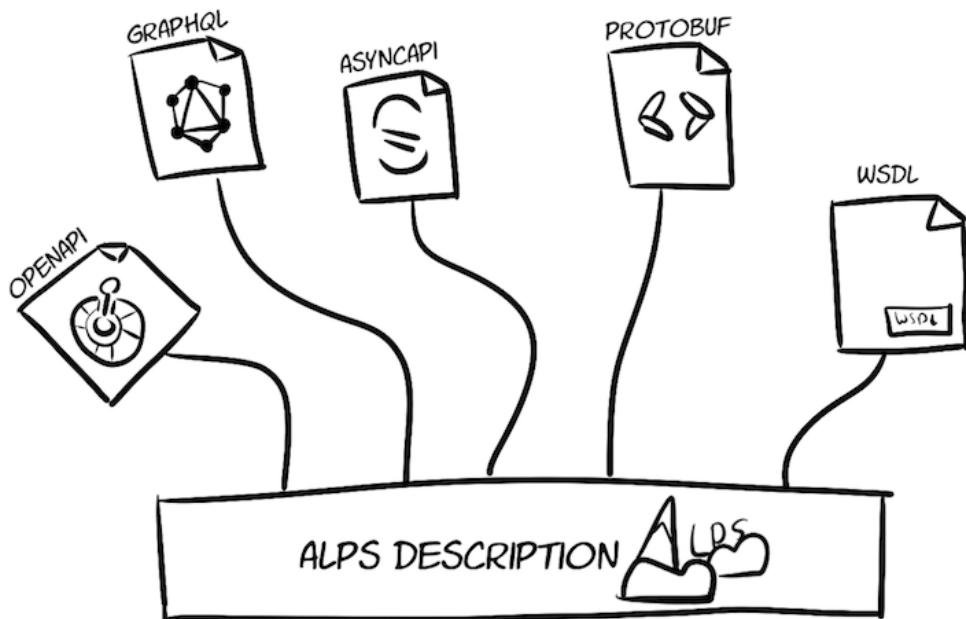
First, we'll cover the importance of API descriptions as a way to standardize the way we talk about the API independent of the way we *implement* the API (for example, protocols, formats, URLs, and so on). We'll also briefly cover two of the most commonly used description formats: DCAP (Dublin Core Application Profiles) and ALPS (Application-Level Profile Semantics). Finally, we'll walk through the process of creating our own general API description document using the ALPS format.

By the time you're done with this chapter, you'll have a complete API description document ready to use when you start building the API itself.

Learning the Role of Description Formats

A good API description provides a solid foundation for implementing the API itself. It converts all those requirements and feature requests we gathered in Chapter 3, [Modeling APIs](#), into a standardized and understandable document that we can safely share with others in the organization and on our team.

It's important to have a shared description of the API because API designers and API builders may not always have a chance to talk to each other directly. Consider situations when the designer works on another team, in another office building, or in a location across the world. At times, API builders may need clarification about some aspect of the design, but it may not be practical to reach the right person right when you need them. That's when a good API description document comes in handy.



Another handy aspect of description formats is that they don't depend on any single technology, document format, or API style. They simply describe the data to be passed back and forth (`givenName`, `familyName`, `userId`) and the actions that should be available (`approveUser`, `updateCompany`, `cancelOrder`).

API descriptions provide enough information to tell everyone *what* must be done, but they leave the *how* of doing it up to the API developers.

Descriptions allow us to share enough information with all parties without prematurely deciding how that information will be used. Let's explore the differences between description and definition a bit more and then look at a couple of examples of common description formats in use today.

Description vs. Definition

Just like the description of a real life object, describing an API gives a general understanding of it but doesn't include enough information by itself to actually create it. You need "definitive" information—a definition—in order to build the API. We'll cover definition languages in Part III when we actually start building the API.

The dictionary definition of *description* is "a statement or account giving the characteristics of someone or something: a descriptive statement or account." The role of an API description is to do just that: give the characteristics of the API we're working on together. What are those characteristics? They're things like data properties (`userName`, `password`, and so on) and actions (`addUser`, `grantUserAccess`, and so on). For example, the following markup is a simple description of an action that creates a new user:

ch05-describing/adduser.txt

```
action: addUser
properties:
  userName: ?
  password: ?
returns: userRecord
```

Note that the action (`addUser`) also includes data properties to pass in (`userName` and `password`). The description also points out that the action will return a `userRecord`. These additional elements (properties and return values)

represent the information needed to make sure the `addUser` action is properly described. But it doesn't include enough information to properly *implement* the API. For example, this description includes no URLs, no indication of which protocol to use (Is this done using HTTP or WebSockets?), and no details on the format used to pass data (JSON? XML? HTML?).

The dictionary defines the word *definition* as “a statement expressing the essential nature of something.” In the API world, expressing the essential nature of the API means including all the details used in implementing it. For example, the API definition of our `addUser` description might look like this:

ch05-describing/adduser-definition.txt

```
action: addUser
  protocol: http
  url: http://apis.usermanagement.org/users/
  method: POST
  format: application/json
  properties:
    userName:
      required: true
      type: string
      validation: alphanumeric
      min-size: 4
      max-size: 32
    password:
      required: true
      type: string
      validation: alphanumeric
      min-size: 8
      max-size: 64
  returns:
    format: application/json
    url: http://apis.usermanagement.org/users/{userId}
    properties: userId, userName, password, dateCreated
```

Note that definitions include many more details than descriptions do. For example, the previous definition includes such information as the protocol,

the format, the method, and a host of details on the input properties (`userName` and `password`), along with additional information on the properties to be returned. We'll cover definition formats in depth in Part II. For now, it's important to recognize the difference between descriptions and definitions and how we can leverage each of them individually.

API descriptions are not the same as definitions. Descriptions give us a general view of the API—its outline or profile—without getting bogged down in the details. That can be handy especially early in the modeling and design process, when you want to focus on the broad outlines first. This idea of the outline- or profile-level interface design has been around for quite a while. One of the early description formats (which we'll look at next) was created, in fact, to offer this “set of metadata elements, policies, and guidelines.”^[37] These guidelines are free of implementation details that can be decided later.

Let's look at some real-life examples of these description or profile formats and how they can be applied to our Onboarding API.

Early Description Formats

The idea of using a document to describe interfaces in large-scale information systems has a relatively recent history—less than twenty years. Without getting too deep into the details, the earliest examples of these descriptions were called *metadata profiles* and started around the year 2000 with library professionals. For example, the Bath Profile was created for library applications and resource discovery.^[38] Another format created around the same time was the U.K. eGovernment Metadata Standard (e-GMS).^[39] The e-GMS was created around 2002 and is based on what's known as the Dublin Core (DC) standards. We'll take a look at that in the next section.

As you might imagine, both of these standards were created to help very large organizations (international libraries and the U.K. government). These

organizations saw the need for describing things in a way that translated well for a wide range of people and for a varying range of implementations. And while most API developers aren't working on quite the same planetary or governmental level, the ability to describe things in a uniform way can be very useful, especially when you consider that the people interacting with your API may never have a chance to meet you or ask you questions. The more useful a general profile you can create, the easier it'll be for developers to interact with your design.

Dublin Core Application Profile (DCAP)

One of the most mature of the description formats is the Dublin Core Application Profile, or DCAP specification.^[40] The DCAP spec is pretty complex, and we won't cover all of it here. However, a small example is helpful. The following is a DCAP profile that describes a **person** record:

DCAP Person Description

ch05-describing/dcap-person.txt

```
Description template: Person id=person
    minimum = 0; maximum = unlimited
    Statement template: givenName
        Property: http://xmlns.com/foaf/0.1/givenname
        minimum = 0; maximum = 1
        Type of Value = "literal"
    Statement template: familyName
        Property: http://xmlns.com/foaf/0.1/family_name
        minimum = 0; maximum = 1
        Type of Value = "literal"
    Statement template: email
        Property: http://xmlns.com/foaf/0.1/mbox
        minimum = 0; maximum = unlimited
        Type of Value = "non-literal"
        value URI = mandatory
```

This **person** DCAP document contains three **Statement** templates for **givenName**, **familyName**, and **email**. Each of those statements contains information on the minimum and maximum occurrences of each of those

values and on the **type** of data for each property (**literal** and **non-literal**). Again, it isn't important that all this makes sense, but it's interesting to understand where the notion of profiles comes from and how it's used in various cases.

For our project, we'll use a more basic and (hopefully) more useful format: the Application-Level Profile Semantics (ALPS) description format.

Application-Level Profile Semantics (ALPS)

The ALPS description format^[41] was designed to help API designers and developers easily and accurately share information needed to create APIs for the web.^[42] For that reason, the ALPS format emphasizes the ability to *do* something—to describe an action along with the data properties to pass with these actions. Like the other description formats, ALPS doesn't get into implementation details such as protocols, URLs, or response formats.

Here's a sample ALPS document that describes the same **person** that appears in the DCAP example we looked at earlier:

ALPS Person Description

ch05-describing/alps-person.yaml

```
descriptor:
  - id: person
    type: semantic
    descriptor:
      - id: givenName
        type: semantic
      - id: familyName
        type: semantic
      - id: email
        type: semantic
      - id: read
        type: safe
        rt: person
```

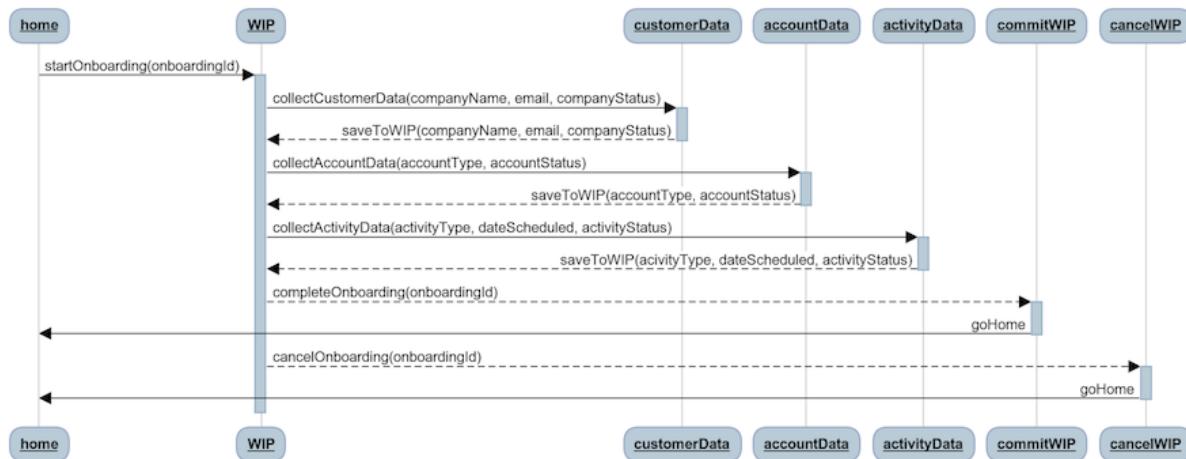
As you can see, not only are the data properties listed (**givenName**, **familyName**, and **email**), we also have an element that describes the **read** action that returns

a **person** record. In a typical ALPS description, there is at least one action descriptor for each of the actions in the API design. Just how each action is implemented (for example, with HTTP GET) is left out of the ALPS description and left for those writing the actual API definition documents.

That gives us a sense of what description formats are and what they look like. Now let's get back to the Onboarding API we've been working on and create a valid ALPS description document for our API.

Describing Your API with ALPS

Recall from Chapter 4, [Designing APIs](#), that we designed the onboarding experience and documented it using, among other things, a sequence diagram, as shown in the [figure](#).



We can use this diagram, along with the data properties we outlined, as a guide to create an ALPS description document for our API (see [Table 1](#)).

The work of authoring an ALPS document is the process of converting the diagram (along with the property information) into a machine-readable format. This format can then be used to generate basic human-readable documentation, as well as some “starter” code for our implementation.

Table 1. Onboarding Properties

Term	Notes	Source
identifier	Onboarding record identifier	http://schema.org/identifier
companyName	Company’s legal name	http://schema.org/legalName
address	Company’s address	http://schema.org/PostalAddress

Term	Notes	Source
email	Company's email address	http://schema.org/email
telephone	Company's phone number	http://schema.org/telephone
maxValue	Account spending limit	http://schema.org/maxValue
status	Account status	http://schema.org/status
discount	Account discount percentage	http://schema.org/discount

Starting from the Top

The first step in the process of creating an ALPS description document is to create the initial YAML document along with a title and simple description. For our Onboarding API example, the initial document looks like this:

ch05-describing/onboarding-alps-01.yaml

```
alps:
  version: '1.0'
  title: "Onboarding API"
  doc:
    type: "markdown"
    value: "This is the ALPS document for BigCo's **Onboarding API**"
```

As you can see from the example, the first element is the `alps` node—this defines the object type for this document. The second element is the `title`, and the third element is the `doc` node. This last node includes a simple text description along with an indicator defining the `type` of content. In our case, we're just writing simple `markdown`. Other supported formats are `html` and `text`.

Adding Data Properties

Next, we need to add the list of the values of the onboarding properties (refer to Table 1) to the ALPS document. Here's what the document looks like now:

```
ch05-describing/onboarding-alps-02.yaml
```

```
alps:  
  version: '1.0'  
  title: "Onboarding API"  
  doc:  
    type: "markdown"  
    value: "This is the ALPS document for BigCo's **Onboarding API**"  
  
  descriptor:  
    - id: "identifier"  
      type: "semantic"  
      text: "Unique identifier for this record"  
      ref: "http://schema.org/identifier"  
  
    - id: "companyName"  
      type: "semantic"  
      text: "Company's legal name"  
      ref: "http://schema.org/legalName"  
  
    - id: "email"  
      type: "semantic"  
      text: "Company's primary email account"  
      ref: "http://schema.org/email"  
  
    - id: "telephone"  
      type: "semantic"  
      text: "Company's phone number"  
      ref: "http://schema.org/telephone"  
  
    - id: "status"  
      type: "semantic"  
      text: "Account status (active inactive suspended)"  
      ref: "http://schema.org/status"  
  
    - id: "maxValue"
```

```
type: "semantic"
text: "Account's maximum spending limit"
ref: "http://schema.org/maxValue"

- id: "discount"
  type: "semantic"
  text: "Account's default sales discount (as a percentage)"
  ref: "http://schema.org/discount"
```

That's all the vocabulary our Onboarding API needs. You can see several important pieces of information here, including these:

id

The identifier for each property

type

The ALPS **descriptor** type—all data properties have a **type** value of "semantic".

text

A short, human-readable description of the property and how it's used—this may even include possible valid values (see the **status** property in the earlier example).

ref

A URL that points to the source of the property **id** value. We covered the use of shared vocabulary names (here they're all from the schema.org site) in Chapter 4, [Designing APIs](#).

For all the remaining examples of our `OnboardingAPI.yaml` document, I'll leave out the **text** and **ref** values in order to make the examples easier to read. You'll find the full version of this ALPS document in the code download files on the book's web page. [\[43\]](#)

With our API's vocabulary fully described in the ALPS document, we have one more related task: adding vocabulary **group** elements.

Adding Group Elements

Now that we have a list of properties for our Onboarding API, it can be helpful to create some **group** elements to make it easy to understand how the properties relate to each other. This can also help anyone documenting or implementing our API to better understand what the experience of using the API should look like.

ALPS **group** elements are actually just **descriptors** with their **type** property set to "**group**". For our Onboarding API we only need one group—the **WIP** group. Here's what it looks like when we add a group to the **OnboardingAPI.yaml** ALPS document:

ch05-describing/onboarding-alps-03.yaml

```
alps:
  version: '1.0'
  title: "Onboarding API"
  doc:

    type: "markdown"
    value: "This is the ALPS document for BigCo's **Onboarding API**"

descriptor:
  # vocabulary properties
  - id: "identifier"
    type: "semantic"

  - id: "companyName"
    type: "semantic"

  - id: "email"
    type: "semantic"

  - id: "telephone"
    type: "semantic"

  - id: "status"
    type: "semantic"
```

```

- id: "maxValue"
  type: "semantic"

- id: "discount"
  type: "semantic"

# reference grouping
- id: "wip"
  type: "group"
  descriptor:
    - href: "#identifier"
    - href: "#companyName"
    - href: "#email"
    - href: "#telephone"
    - href: "#status"
    - href: "#maxValue"
    - href: "#discount"

```

First, you'll notice I shortened the contents of the `#properties` listing to make this example easier to read. Also, as you can see from this example, adding an ALPS `group` is straightforward. You just need to create a new `descriptor` of the type `"group"` and then nest a series of ALPS properties within that `descriptor`. This is, essentially, creating a `shorthand` name (`"wip"`) for a collection of properties.

Pointing to Other ALPS Descriptors



Note the use of the hash tag (#) and the `href` properties for nested `descriptors`. This notation is a way to indicate that the `descriptor` is *pointing* to another `descriptor` in this document. This shorthand is used quite often in ALPS documents.

Now that we have both properties and groups defined in our ALPS document, it's time to add information about ALPS actions.

Adding Action Elements

In ALPS, the action elements are also listed as **descriptors**. But these descriptors have different **type** values that indicate whether the action is **safe**, **unsafe**, or **idempotent**. We covered the concepts of **safety** and **idempotence** in Chapter 2, [Understanding HTTP, REST, and APIs](#).

Remember, when looking at the [onboarding experience diagram](#), the keywords on the arrows are what identify actions. With that in mind, we can transcribe the information in the diagram into the ALPS document. Once you add actions to the **OnboardingAPI.yaml** document, your ALPS file should look something like this:

ch05-describing/onboarding-alps-04.yaml

```
alps:
  version: '1.0'
  title: "Onboarding API"
  doc:
    type: "markdown"
    value: "This is the ALPS document for BigCo's **Onboarding API**"

  descriptor:
    # vocabulary properties
    - id: "identifier"
      type: "semantic"

    - id: "companyName"
      type: "semantic"

    - id: "email"
      type: "semantic"

    - id: "telephone"
      type: "semantic"

    - id: "status"
      type: "semantic"

    - id: "maxValue"
      type: "semantic"
```

```

- id: "discount"
  type: "semantic"

# reference grouping
- id: "wip"
  type: "group"
  descriptor:
    - href: "#identifier"
    - href: "#companyName"
    - href: "#email"
    - href: "#telephone"
    - href: "#status"
    - href: "#maxValue"
    - href: "#discount"

# actions
- id: "startOnboarding"
  type: "unsafe"
- id: "collectCompanyData"
  type: "safe"
- id: "SaveToWIP"
  type: "idempotent"
- id: "collectAccountData"
  type: "safe"
- id: "completeOnboarding"
  type: "idempotent"
- id: "abandonOnboarding"
  type: "idempotent"
- id: "goHome"
  type: "safe"

```

Now the updated document contains the actions found in the [sequence diagram](#):

startOnboarding

This action is marked **unsafe** since it creates a new work-in-progress (WIP) record, ready to populate with collected company and account data.

collectCompanyData

This action is the one users can activate in order to get to the screen that collects the **company**-related information. It's marked **safe** since it's just a link to a form. It does not modify any stored data.

saveToWIP

This action is the one that saves any collected data back to the **WIP** record. For this reason it's marked **idempotent** (which ALPS understands as both “unsafe” because it writes data, and “idempotent” since it can be repeated without causing errors in the data that gets written).

collectAccountData

Like the **collectCompanyData** action, this is used to forward the user to the screen that collects **account**-related data. It's also marked **safe**.

completeOnboarding

This is an **idempotent** action that takes all the collected data in the **WIP** record and sends it to the service back end. This is why we're creating this API, right (grin)?

abandonOnboarding

As we saw in Chapter 4, *Designing APIs*, it is important to allow users to simply cancel the onboarding process when needed. This action removes all the collected data for this **WIP**, so it's marked **idempotent** too.

goHome

Finally, this action takes users back to the starting point in case they want to start the process again (for example, for a new company). Since this is just a navigation, it's marked **safe**.

In this example, I left out the **text** and **rel** elements to keep the document easy to read. You'll find the full **OnboardingAPI.yaml** ALPS document in the code download available on the book's web page.^[44] That version will have

all the human-readable descriptions along with pointers to the original source definitions for each `id` in the document.

We now have an ALPS document ([OnboardingAPI.yaml](#)) that contains a title, the list of properties, and the list of actions for our Onboarding API. We're not quite done, though. We can improve our ALPS document by adding the arguments passed with each action.

Adding Action Parameters

While not a requirement for ALPS documents, it's a good idea to indicate all the arguments passed when executing the actions. In a typical HTTP-based API, these are the query-string arguments and/or the data elements passed in the body of a response or request.

Adding this information improves the details of the ALPS description. That means any generated documentation will be better; and when we get to the work of building the API in Part III, we'll have more information on how to actually implement the data-passing algorithms in code.

Based on the notes from our design work in the previous chapter, here's the same [OnboardingAPI.yaml](#) document with arguments and return values added:

ch05-describing/onboarding-alps-05.yaml

```
alps:
  version: '1.0'
  title: "Onboarding API"
  doc:
    type: "markdown"
    value: "This is the ALPS document for BigCo's **Onboarding API**"

  descriptor:
    # vocabulary properties
    - id: "identifier"
      type: "semantic"

    - id: "companyName"
```

```

type: "semantic"

- id: "email"
  type: "semantic"

- id: "telephone"
  type: "semantic"

- id: "status"
  type: "semantic"

- id: "maxValue"
  type: "semantic"

- id: "discount"
  type: "semantic"

# reference grouping
- id: "wip"
  type: "group"
  descriptor:
    - href: "#identifier"
    - href: "#companyName"
    - href: "#email"
    - href: "#telephone"
    - href: "#status"
    - href: "#maxValue"
    - href: "#discount"

# actions
- id: "startOnboarding"
  type: "unsafe"
  rt: "wip"

- id: "collectCompanyData"
  type: "safe"
  rt: "wip"
  descriptor:
    - href: "#identifier"
  # more properties go here...

- id: "SaveToWIP"
  type: "idempotent"

```

```

  rt: "wip"
  descriptor:
    - href: "#identifier"
    - href: "#companyName"
    - href: "#email"
    - href: "#telephone"
    - href: "#status"
    - href: "#maxValue"
    - href: "#discount"

  - id: "collectAccountData"
    type: "safe"
    rt: "wip"
    descriptor:
      - href: "#identifier"
      # more properties go here...

  - id: "completeOnboarding"
    type: "idempotent"
    rt: "wip"
    descriptor:
      - href: "#identifier"

  - id: "abandonOnboarding"
    type: "idempotent"
    rt: "wip"
    descriptor:
      - href: "#identifier"

  - id: "goHome"
    type: "safe"

```

Now our Onboarding API document includes lots of additional information that pulls everything together. For almost all of the actions, you can see a nested set of **descriptors** indicating which arguments are passed for each request. You can also see a new property—the **rt** (or **return** property), which indicates what kind of data is passed in response to each request. In our API, the **wip** property group gets returned each time. In more complex APIs, we would likely have several property groups defined, each being returned at different times based on the API experience we wish to design.

That's a lot of information to add, but it's very important information. Anyone working to document or implement our Onboarding API will now have a good idea of what kind of data will be required for each action.

The next (and last) task for this chapter is to update our project folder with these new assets so that they'll be available in the future.

Updating Your API Project

Now that we've created our ALPS description document for the Onboarding API and used the ALPS file to generate basic human-readable API documentation, it's time to add these two important assets to our ongoing project. That means first adding them to the project repository and then updating the [package.json](#) file.

The first step is to make sure both the [OnboardingAPI.yaml](#) ALPS file and the [OnboardingAPI.md](#) API documentation files are copied to your local project folder. I keep these kinds of documents in a project subfolder named [assets/](#). This is the folder I use to hold important files that aren't required for the build or the runtime of the API. Your team may have another common place for these kinds of files, or you may prefer to place them somewhere else. It's up to you.

Once you copy them to your project, you need to add them to your Git repo. Committing your Onboarding API files to the repo looks like this:

```
git add assets/OnboardingAPI.*  
git commit -m"add onboarding api assets"
```

Once they've been added to your repo, you can update your [package.json](#) file with Onboarding Assets to include them too.

ch05-describing/package-file.js

```
{  
  "name": "Onboaring API",  
  "version": "1.0.0",  
  "description": "Onboarding API for BigCo, Inc.",  
  "main": "app.js",  
  "scripts": {  
    "test": "echo |\"Error: no test specified|\" && exit 1"  
  },  
  "keywords": [  
    "onboarding",  
    "api",  
    "bigco",  
    "inc."  
  ]  
}
```

```
"onboarding",
"company",
"account",
"activity",
"api"
],
"author": "Mike Amundsen",
"license": "MIT",
"assets" : {
  "diagram" : "OnboardingAPI.png",
  "alps" : "OnboardingAPI.json",
  "apiDocs" : "Onboarding.md"
}
}
```

What's Next?

This chapter reviewed a number of things related to documenting your API design in both machine-readable and human-readable formats. Along the way we covered the importance of implementation-agnostic description formats for converting your diagrams into other formats, and the difference between general description formats and implementation-specific definition formats. We also reviewed both the Dublin Core Application Profile (DCAP) and the Application-Level Profile Semantics (ALPS) formats.

In addition, we worked through the process of describing our Onboarding API using ALPS. That included adding the title and general documentation text ([*Starting from the Top*](#)), adding all the vocabulary from our design session ([*Adding Data Properties*](#)), organizing the response data into shorthand names ([*Adding Group Elements*](#)), describing all possible state transitions in the API ([*Adding Action Elements*](#)), and providing added detail to all the state transitions ([*Adding Action Parameters*](#)). We also added the resulting ALPS description document to our API project folder and updated the [`package.json`](#) file.

That wraps up the design portion of the book. You started in Chapter 3, [*Modeling APIs*](#), learning about Donald Norman’s Action Lifecycle and how to use it to design APIs. You also learned to use Git and npm to start managing our API project.

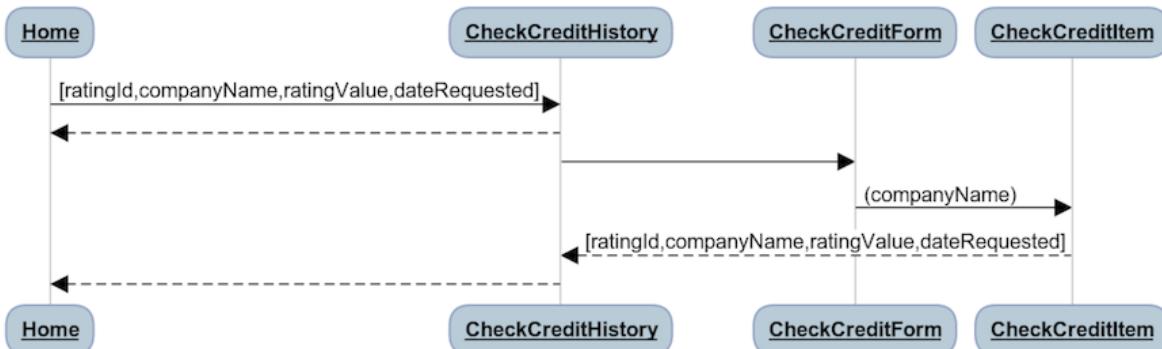
In Chapter 4, [*Designing APIs*](#), you learned how to apply a simple API design method as a way to gather important information about the planned user experience. That included collecting not just *data* names but also information about *actions* taken with that data. You then learned how to normalize our data names using ontologies like Schema.org and how to use the WebSequenceDiagrams service to turn our design results into a visual that’s easy to read and share with others.

Now that you've translated that visual design information into both human-readable HTML and machine-readable ALPS data, you're ready to start actually *building* the API from the ground up.

That's where we'll start in Part III of this book.

Chapter Exercise

In this chapter, you learned how to use ALPS to describe the data and the action elements of our Onboarding API. The code folder for this chapter contains several documents for the **Credit-Check** service we've been working on over the past few chapters. That includes the workflow document ([credit-check-workflow.txt](#)), the web sequence source file ([credit-check-diagram.wsd](#)), and the WSD diagram ([credit-check-diagram.png](#)) [shown](#).



Using these assets as a guide, create the accompanying ALPS description document that includes all the properties and actions for the Credit-Check API. You can render the ALPS document in either YAML, JSON, or XML —whichever you prefer.

Once you complete the ALPS document, be sure to add it to the **assets** folder of your Onboarding API project and update your project repo using Git.

Hint: You'll find some partially completed files in the [code/ch05-describing/exercise/before/onboarding/assets](#) folder to help you get started.

See Appendix 2 ([Solution for Chapter 5: Describing APIs](#)) for the solution.

Footnotes

[37] <http://www.dublincore.org/documents/2001/04/12/usageguide/glossary>

- [38] http://www.ukoln.ac.uk/interop-focus/activities/z3950/int_profile/bath/draft/stable1.html
- [39] <https://en.wikipedia.org/wiki/E-GMS>
- [40] <http://dublincore.org/documents/profile-guidelines>
- [41] <http://alps.io/spec>
- [42] <https://www.infoq.com/articles/programming-semantic-profiles>
- [43] https://pragprog.com/titles/maapis/source_code
- [44] https://pragprog.com/titles/maapis/source_code

Copyright © 2020, The Pragmatic Bookshelf.

Part 3

The Build Phase

Chapter 6

Sketching APIs

In Part II, we covered the work of turning API requirements into a solid design plan using a repeatable API design process. At the end of our design journey, we had a set of documents (sometimes called *artifacts*) that included our human-readable workflow document, a sequence diagram, and our machine-readable ALPS document.

Now it's time to turn those artifacts into a working API by converting our design into an implementation. That's what we'll do over the next three chapters. Along the way, you'll learn how to create quick API sketches, how to make a prototype of your API, and how to convert your sketches and prototypes into running NodeJS code. At each step of the journey, we'll refine our implementation plans through our sketches and prototypes.

Sketching and prototyping is a way to test ideas. We can create low-fidelity samples that we can show others and make adjustments and improvements all without having to write a lot of code. I like using sketches and prototypes because I can learn from early mistakes without doing a lot of tedious coding and debugging.

You'll learn about prototypes—which are more detailed than sketches—in Chapter 7, [Prototyping APIs](#). For now, we'll focus on converting our designs into sketches.

Learning from Frank Gehry's Sketches

When I start work on translating my API design into working code, I always like to start with small, simple examples that I can put together in just a few minutes. This helps me work out just how I'll convert the design into an API without getting too wrapped up in editing code, loading libraries, and all the other details that are needed for *real* running code.

I got the idea of using time-saving sketches from a talk by Ronnie Mitra called “The Art of Effective API Design.”^[45] In his talk, Mitra introduced me to the way celebrated architect Frank Gehry uses sketches to test out his design ideas for physical buildings. For Gehry, sketches are a way to quickly work out possibilities and test ideas long before he tries to actually build the building. As Gehry puts it: “As soon as I understand the scale of the building and the relationship to the site and the relationship to the client, as it becomes more and more clear to me, I start doing sketches.”^[46]

Born in Toronto in 1929, Gehry grew up playing with “everyday materials” to build little cities to entertain himself. After he and his family moved to California when he was 18, he earned his bachelor of architecture degree from the University of Southern California (USC). By the age of 28, he received his first commission to design a friend’s private home. It was at USC where Gehry found himself hanging around artists more often than architects, and that’s when he developed the habit of sketching his ideas on paper.

Sketches of Frank Gehry



Gehry’s sketches have become a source of artistic interest all by themselves. You can learn more about Gehry and his amazing drawings from the 2006 documentary, *Sketches of Frank Gehry* (<http://www.pbs.org/wnet/americanmasters/frank-gehry-sketches-of-frank-gehry/602>).

I won't be able to include a Gehry sketch in this book, but you can find them online. Instead, I'll show you an original sketch to give you an idea of what I'm talking about. For example, suppose I need to add an entryway to my new country home that transitions visitors from the small garden area in front of the house onto an inviting porch area that leads to a large front door. There are lots of possible ways to do this. I could use small trees along an open walkway or a long narrow hall leading to the door, or maybe an ornate marble entry to impress my visitors. The [image](#) show some example sketches of these ideas.



Sketching is a quick and easy way to test your ideas without having to actually build them in the real world. With these drawings, I can show them to my family and to builders on-site to get feedback and refine the ideas into their final form.

You can use the same approach with your APIs too. Let's look at an example sketching session to see what I mean.

API Sketching Example

Like the pencil sketches Gehry uses to explore possible building designs, you can create “API sketches” to explore your API designs. For example, in the Onboarding API project we’ve been working on over the past several chapters, we have the `startOnboarding` action defined in our ALPS document in [Adding Action Elements](#). Let’s walk through a sample “design sprint” for just this one element of our API.

For starters, our ALPS document includes the following element that describes the `startOnboarding` action:

...

```
{"id" : "startOnboarding", "type" : "unsafe", "rt" : "wip"}
```

...

Now we need to turn that into an actual *message* that would be sent from the API provider to an API client application. One possible API response might look like this:

ch06-sketching/start-onboarding-01.json

```
{  
  "startOnboarding" : "http://api.example.org/start"  
}
```

From this example, you can see I simply translated the `startOnboarding` action in the ALPS document into a name/value pair in a JSON message. That certainly works. But as I look at it, quite a bit of information is missing (or assumed). Maybe we should add a bit more content in the message to make it easier to understand?

Here’s my next sketch for the `startOnboarding` action:

ch06-sketching/start-onboarding-02.json

```
{  
  "link" : {"id" : "startOnboarding", "href" : "http://api.example.org/start"}  
}
```

That, I think, is a bit easier to read and understand. I created a `link` element in the message that contains the `id` element from the ALPS document and created an `href` element to identify the actual URL.

Again, as I look at this sketch, it dawns on me that my response messages are likely to have several `link` elements in them. Maybe this sketch is better:

ch06-sketching/start-onboarding-03.json

```
{  
  "links" : [  
    {"id" : "startOnboarding", "href" : "http://api.example.org/start"}  
  ]  
}
```

Now I have an array of links, each with their own `id` and `href` values.

Something else I think is missing from this message is some indication of its *type*—what kind of message is this response? For that, I'd like to add another *named* element at the root of the response that looks like this:

ch06-sketching/start-onboarding-04.json

```
{  
  "onboarding" : {  
    "links" : [  
      {"id" : "startOnboarding", "href" : "http://api.example.org/start"}  
    ]  
  }  
}
```

Now it's clear that this is an `onboarding` message. That can come in handy for API consumer apps that need to know what type of response message they

have in order to know how to parse the incoming data and/or render it on a screen.

How about we jump ahead a bit? The following is a design I came up with after a handful of iterations. This one actually looks close to the final design I plan to use for my Onboarding API:

ch06-sketching/start-onboarding-05.json

```
{
  "onboardingAPI" : {

    "metadata" : [
      {"id" : "title", "value" : "BigCo, Inc. Onboarding"},  

      {"id" : "author", "value" : "Mike Amundsen"}  

    ],  
  

    "links": [  

      {  

        "id" : "startOnboarding",  

        "href" : "http://api.example.org/start",  

        "method" : "POST",  

        "properties" : []},  

      {  

        "id" : "home",  

        "href" : "http://api.example.org/",  

        "method" : "GET"
      }
    ],  
  

    "items" : [  

      ...  

    ]
  }
}
```

Here, I extended the idea of a `links` array with two more arrays: `metadata` will hold general information about this message and `items` will contain any data records that get returned in the response. I also added more information in

each `link` element and an additional `metadata` element. More could be done to improve this sketch, but it's fine for now. I don't need to account for everything about the API in this one simple sketch.

In a real-life situation, I'd be showing some of these examples to stakeholders and/or target customers—maybe even my teammates—to get their feedback and suggestions on how to improve the interface. Each sketch is a chance to try out new ideas until I get to a point where the results seem “good enough.” And getting to this “good enough” point early is really important for the overall quality of the API.

Now I have a pretty solid idea of what my API responses will look like. And I got here by creating a handful of small sketches in quick succession. Even though I started with just one element of my ALPS description document, I ended up with a general message design that should work in most situations.

I still need to do some more sketching for other action and property elements, but this is a good start. Short, cheap explorations allow me to be creative without much downside. That's just one of the advantages of using this API sketching approach.

The Advantages of Sketching

Sketching APIs offers a handful of advantages. As I mention in the previous section, sketching is a great way to try things out and quickly get feedback from others on your ideas. When it takes so little time to try something out, you can afford to try several ideas before you settle on your preferred implementation approach. Sketching APIs can even help you be more creative when it comes to translating the design documents into working API examples.

Also, because the sketches are just small samples of responses (or even requests, if you are designing those too), you don't need to do lots of work ahead of time. Sketches are simple, stand-alone snippets that convey your ideas without the need to write actual code and deploy it on some server somewhere. Even if you don't plan to go through lots of iterations of your sketching, you can take advantage of the "low-tech" nature of sketches and ignore the corner-cases, exceptions, and other minor details you'd be forced to deal with if you tried to "sketch" your API in working code.

Sketching APIs can make it easy for stakeholders or customers to "see" what the API will look like and to provide quick feedback and suggestions on how to improve the API. Since it takes me no time at all to whip up a sketch, I don't mind if a customer changes his mind and tells me to "go back and do it over" a few times. In fact, I sometimes do the sketching right on the spot with other interested parties in the room (or on the same video call). In most cases, it's quite enjoyable to work directly with the people who plan to use the API you're working on.

Finally, the good news is that your sketches are disposable! I almost always end up creating several sketches before I create one that hits that "good enough" level I talked about earlier in this chapter. And I don't mind that I have lots of extra sketches. They helped me get to the right spot. Who

knows, I might go back and reuse some of these rejected sketches in the future.

Since you don't need any special tooling, you can start using sketches right away. But you can take your sketches a bit further with an API design tool that can use your sketches as a guide for creating a sample API interface. I cover that in the next part of this chapter.

Sketching APIs with Apiary Blueprint

A handy tool for generating API sketches is called the Apiary Blueprint Editor (<https://apiary.io>). It's easy to use and supports creating a live online version of your sketches so that you can share them with others and even use tools like curl or coded API client apps to interact with your sketches. This ability to interact with your sketches makes it really easy to quickly test and experiment with your API implementation ideas before committing to writing lots of code.

Let's walk through a sketching session using the Apiary editor to see how it can help you sketch your API ideas. We'll look at the following:

- Loading the editor and logging in
- Creating a new Apiary project
- Adding your first API sketch
- Testing your first sketch with Apiary's Mock Server
- Saving your sketch to your local disk as another asset in your project

By the time we're done, you should have a pretty good idea of how you can use the Apiary Blueprint Editor to create and save your own API sketches in the future.

A Brief History of Apiary

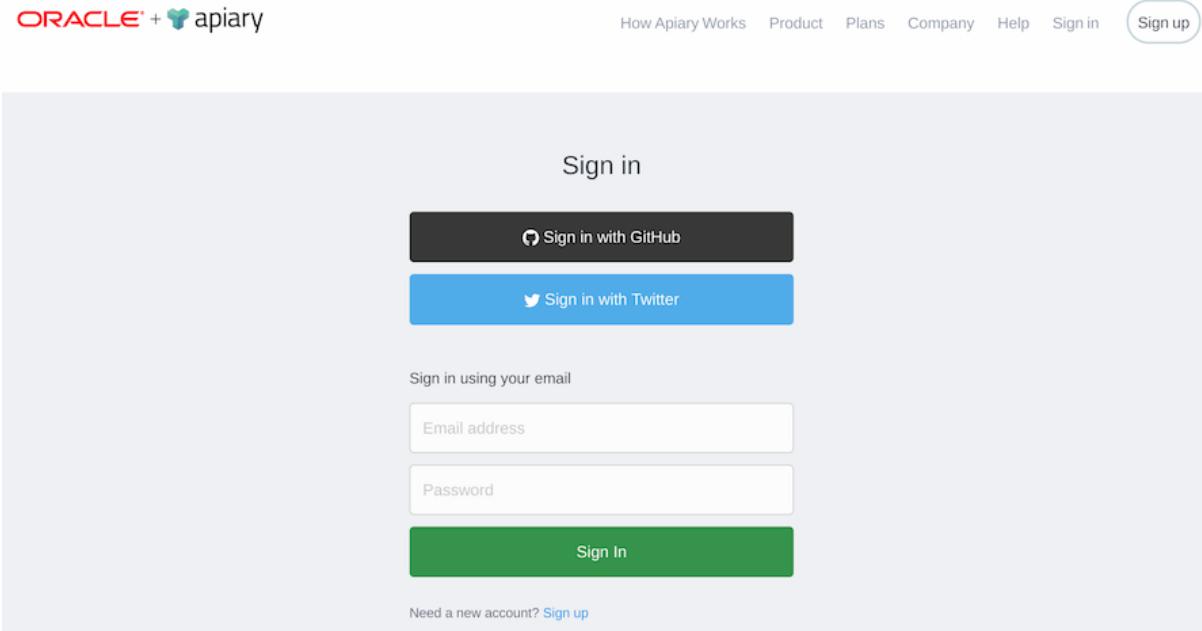


The company known as Apiary was founded in 2013 by Czech entrepreneur Jakub Nesetril. Originally focused on quickly documenting and testing HTTP APIs, Apiary allows users to write markdown language in order to specify HTTP requests and responses. Over time, Apiary has created a series of API-related tools for designing, documenting, mocking, and testing APIs. Apiary was sold to Oracle in 2017, which continues to offer free online versions of all of Apiary's API tools.

Loading the Apiary Editor and Logging In

The first step in creating API sketches with the Apiary editor is loading the web app and logging in. While use of the app is free, you'll need to create an account with the Apiary site in order to continue to build and save your sketches.

First, to load the app, open your browser and navigate to <https://app.apiary.io>. When you do that, you should see a “Sign in” screen that looks like the [screenshot](#).



Before you can use the Apiary editor, you need to sign in. Since I have a GitHub account, I use that identity to sign in. It's safe and easy and I don't need to create a separate login identity for Apiary's site. However, if you'd prefer to create a stand-alone account, you'll find the “Sign up” link near the bottom of the “Sign in” page, where you can create new account.

After you log in, you'll see the full editor screen, as shown in the following screenshot:

The screenshot shows the Apiary Blueprint editor interface. On the left, there is a code editor window containing JSON-like API definitions. The code includes sections for 'INTRODUCTION', 'REFERENCE', and 'Questions Collection'. The 'Questions Collection' section contains actions for 'List All Questions' and 'Create a New Question'. On the right, there is a preview pane titled 'To-Do API' which displays the API's documentation. The preview pane includes sections for 'INTRODUCTION', 'REFERENCE', and 'Questions Collection', mirroring the structure of the code editor.

```

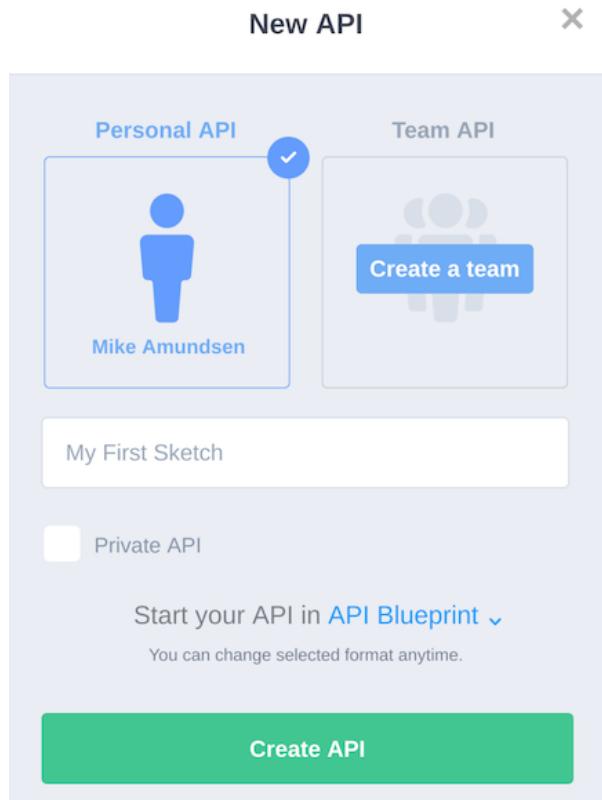
1 FORMAT: 1A
2 HOST: http://polls.apiblueprint.org/
3
4 *# To-Do API
5
6 Polls is a simple API allowing consumers to view polls and vote in them.
7
8 ## Questions Collection [/questions]
9
10 *## List All Questions [GET]
11
12 + Response 200 (application/json)
13
14
15 [
16   {
17     "question": "favorite programming language?",
18     "published_at": "2015-08-05T00:40:51.620Z",
19     "choices": [
20       {
21         "choice": "Swift",
22         "votes": 2048
23       },
24       {
25         "choice": "Python",
26         "votes": 1024
27       },
28       {
29         "choice": "Objective-C",
30         "votes": 512
31       },
32       {
33         "choice": "Ruby",
34         "votes": 256
35       }
36     ]
37   }
38 ]
39
40 *## Create a New Question [POST]
41
42 You may create your own question using this action. It takes a JSON object containing a question and a collection of answers in the form of choices.
43
44 + Request (application/json)
45

```

The first time you sign in, you'll be shown a sample Apiary Blueprint document. If you've logged in before, you'll see the document you were working on last time. Whatever you see there, our next step is to create a new Apiary document to start our API sketching.

Creating a New Apiary Project

We need to create a *new* Apiary project to hold the sketches for our API. To do this (after logging in), go to the upper-left corner of your editing screen and select the project drop-down menu. There you'll see a list of your projects and a bar at the top of the list with the text, “Create New API Project.” Click the bar to pop up the “New API” dialog and enter a project name. (See the following screenshot.) I entered “My First Sketch.”



When the project loads the first time, it'll be populated with a sample API sketch. We don't need much of this sample sketch. The only lines we're interested in are the first three lines of text. Those three lines set up our sketch format, our API root URL, and our sketch title. If you're following along in your own editor, make sure the first three lines in your project look like this:

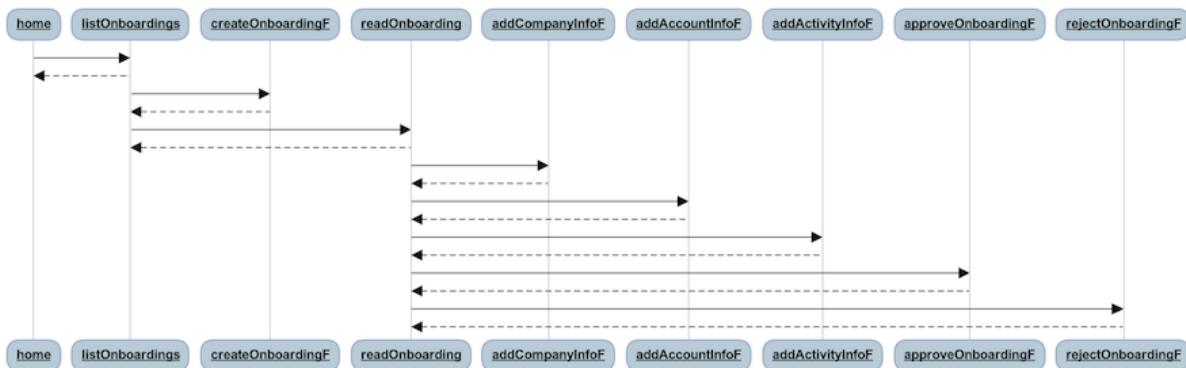
```
FORMAT: 1A  
HOST: http://onboarding.example.org
```

```
# My First Sketch
```

Then delete all the other lines below your title line (`# My First Sketch`) and press the Save button that appears in the upper-right corner of your editor screen. Now we're ready to do some sketching!

Adding Your First API Sketch

It's a good idea to keep your initial API sketches super simple. A reasonable goal is to capture just the basics of the information collected in the diagram from your design phase work. For this work, we'll reference the diagram first shown[here](#) and shown [again](#).



Using this diagram as a guide, let's start our first sketch by adding a line of description for the sketch and then filling in the Home resource from the diagram. To do this, add the following text to your API project in the Apiary editor:

```
FORMAT: 1A
HOST: http://onboarding.example.org

# My First Sketch

This is a preliminary sketch of the Onboarding API

## Home [/]

### Get Home Resource [GET]

+ Response 200 (text/html)

<h1>Home</h1>
```

In this example, I added the description line of text and then started an HTTP resource definition (`## Home [/]`). That line gives the resource a name (`Home`) and a URL (`/`). In Apiary sketches, all resource definitions start with two hash characters (`##`).

The next line defines an HTTP method for that resource ([\[GET\]](#)) and, in this case, an HTTP response for that method (starting with [+ Response...](#)). Since each resource may support multiple HTTP methods, you use lines that start with three hashes ([###](#)) to define each one.

I defined the sample response for this sketch as returning an HTTP status of [200](#) and a message body (in [text/html](#) format) of [`<h1>Home</h1>`](#). I usually do my early sketches in HTML rather than XML or JSON. I find them easier to read, and (as you'll see in the next step) I can also use my browser to test out this API. Also, don't forget to press the Save button after each update in the Apiary editor.

Now that we have our first HTTP resource defined, let's test that out with our browser.

Spaces and Tabs



The number of spaces and/or tabs on a line is *very* important to the Apiary editor. For example, the response body content for HTTP messages needs to be indented 12 spaces (or three tabs). This space-counting can be pretty annoying, and the editor has a built-in routine to check each line for errors to help you sort this out.

Testing Your First Sketch with Apiary's Mock Server

After you save your API sketch, use the panel on the right side of the screen to scroll down to Get Home Resource and click it. This will bring up the example screen, with hints on how you can run a test request of this resource. You should see three parts to this example panel. The first is the title and a complete URL. Each URL will be unique for your project, but it should start with [https://private-](#) and end with [.apiary-mock.com](#). The parts in the middle will be different for each project. The second section is the Request section. You'll see two drop-down lists here. Make sure they're set

to Mock Server and Raw, respectively. The final section is the Response section, which should show the expected response for this API call. This should match the definition we created earlier.

Apiary's Mock Server supports a handful of ways to test your API sketch. The quickest way is to click the Try It button that appears to the right of the Request portion of the example. This calls up the Console screen, where you can press the Call Resource button to execute the request and see the response, as shown in the following screenshot:

The screenshot shows a user interface for testing an API sketch. At the top, there is a close button (X) and a 'Switch to Example' button. Below that, there are two buttons: 'Show Code Example' and 'Mock Server' (which is selected), followed by a 'Call Resource' button. The main area displays a request and its response.

Request:
GET <http://private-707c52-myfirstsketch.apiary-mock.com/>

Response: 200

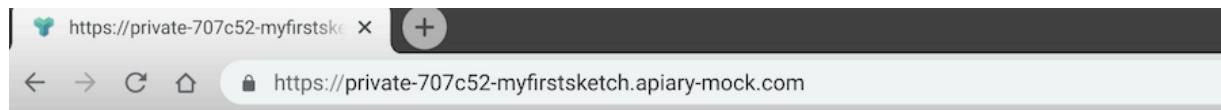
Response Headers Real Diff Specification

1 Content-Type: text/html
+ 2 Access-Control-Allow-Origin: *
+ 3 Access-Control-Allow-Methods: OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE, CONNECT
+ 4 Access-Control-Max-Age: 10
+ 5 x-apiary-transaction-id: 5d12acb507e9380a003163d2
+ 6 Content-Length: 17

Response Body Real Diff Specification

```
<h1>Home</h1>
```

Another way to test your sketch is to copy the complete URL you see in the example screen and paste it into your browser. This works for all the HTTP GET methods on your resources. For example, here's the response from my browser when I test out the URL in my project for the Home resource:



Home

Finally, if you want to get really fancy, you can use one of the drop-down menus in the Apiary example screen (the one that defaults to Raw) to get a suggested code snippet to test the resource. For example, if you change the second request from “Raw” to “cURL” in the drop-down menu, you’ll see a snippet that looks something like this:

```
curl --include \
'https://private-707c52-myfirstsketch.apiary-mock.com/'
```

You can paste that snippet into your command-line window to see the results. The response should look like this:

```
HTTP/2 200
server: Cowboy
x-apiary-ratelimit-limit: 120
x-apiary-ratelimit-remaining: 119
content-type: text/html
access-control-allow-origin: *
access-control-allow-methods: OPTIONS,GET,HEAD,POST,PUT,DELETE,TRACE,CONNECT
access-control-max-age: 10
x-apiary-transaction-id: 5d12aef421de550a00ad844a
content-length: 17
date: Tue, 25 Jun 2019 23:32:04 GMT
```

```
<h1>Home</h1>
```

Before we get too far ahead, we need to figure out how to save our sketches to our local disk for future reference. That turns out to be a bit tricky.

Saving Your API Sketch to Disk

The Apiary editor was designed to store API projects on GitHub. When you load the Apiary editor, you'll see a "Link this Project to GitHub" button in the upper-left corner of the screen. While you can definitely do that, it's not always the best solution. You may want to keep lots of sketches on hand for a project and don't always want to share all of them in a GitHub repository. In those cases, it would be best if you could save your sketches to your local hard disk directly. However, the editor doesn't support that option. So in order to save (and restore) your sketches in the editor, you can resort to a bit of a hack.

To save your sketches to disk, first open them up in the Apiary editor. Then, with your cursor on the left side of the editor (the part where you type the sketch), execute a "select all and copy" command to copy the text onto your shared clipboard. Next, open up your text editor to a new, blank document and paste that text into your local editor.

Finally, use your local text editor to save the contents of the file locally (I usually save mine in my own project folder). When you do this, use the same name on your hard disk as Apiary uses for the online version of the sketch. You can find this by looking at the URL of your project in the Apiary editor. For example, the URL for the new project we've been working on is <https://app.apiary.io/myfirstsketch/editor>, which means the project name is `myfirstsketch`. Use that name as the disk file name and add `.apib` as the extension to get `myfirstsketch.apib`. Naming the file this way makes it easy to figure out where (online) this sketch is stored.

Another good idea is to edit your sketches locally on disk and then use a similar process to copy/paste the contents of your local file into the Apiary editor. This way you can edit the document offline and then copy/paste the text into the online editor when needed.

API Sketching Tips and Tricks

In this quick run-through of API sketching, we focused on how to use the Apiary editor. Before we end the chapter, let's go over a couple of key points to help you with your own sketching practice. The tips and tricks I've picked up along the way may help you when you start to use API sketches to design and build your APIs.

Sketching APIs in HTML

Earlier, I used HTML for one of my early sketches. I do this a lot. It makes it easy for me to visualize the way resources connect to each other, and it also makes it possible for me to use the Apiary Mock Server to share a user-friendly rendering of my sketch with other stakeholders on my team, with my clients, and so forth.

For example, here's the Onboarding Record sketch in HTML. This is the resource that supports all the actions, like `addCompanyInfo`, `addAccountInfo`, `addActivityInfo`, and `approveOnboarding` and `rejectOnboarding`:

```
Onboarding Record [/1q2w3e4r5t]

### Show Onboarding Record [GET]

+ Response 200 (text/html)

<h1>ITEM:1q2w3e4r5t</h1>

<p><a href="/list">LIST</a></p>

<p>data fields display here...</p>

<p><a href="/1q2w3e4r5t/company">Show Company Info Form</a></p>
<p><a href="/1q2w3e4r5t/account">Show Account Info Form</a></p>
<p><a href="/1q2w3e4r5t/activity">Show Activity Info Form</a></p>
<p><a href="/1q2w3e4r5t/approve">Show Approve Form</a></p>
<p><a href="/1q2w3e4r5t/reject">Show Reject Form</a></p>
```

Of course, this sketch renders nicely in a browser—all the links actually work!

HTML sketching is also handy when you need to capture the arguments that will be passed for HTTP requests too. You can just mock that up using HTML forms. The following is an example of the HTML mock-up of the Create Onboarding action. In this sketch, HTTP POST is used to create a new, blank Onboarding resource, ready to start filling in on subsequent steps.

```
### Create Onboarding Record [POST]

+ Request (application/x-www-form-urlencoded)

    onboardingId=1q2w3e4r5t

+ Response 201 (text/html)

+ Header

    Location: /1q2w3e4r5t

+ Body

    <p><a href="/1q2w3e4r5t">created ITEM:1q2w3e4r5t</a></p>

## Onboarding Create Form [/create]

### Show Onboarding Create Form [GET]
+ Response 200 (text/html)

<h1>CREATE-FORM</h1>

<form name="create" action="/list" method="post">
    onboardingId:<input name="onboardingId" value="1q2w3e4r5t" />
    <input type="submit" />
</form>
```

This example also shows how you can use the Apiary editor to mock up POST request bodies and response headers. While this example uses

HTML, it works the same for JSON, XML, and other response formats.

You can get a better look at a completed HTML sketch of our Onboarding API by checking out the full sketch in the source code for this chapter.^[47]

Different Formats, Different API

I also completed an API sketch using the `application/forms+json` format briefly discussed in Part II of this book. You can see the full FORMS+JSON sketch in the book's source code for this chapter. When you compare that FORMS+JSON sketch to the HTML sketch, you'll notice a pretty big difference in the implementation details.

The HTML sketch is quite loyal to the initial diagram and has a total of nine resources defined: one for each state (Home, List, and Item) and one for each action (Create, CompanyInfo, AccountInfo, ActivityInfo, Approve, and Reject). However, the FORMS+JSON version of the same diagram has only five resources defined (Home, List, Item, Approve, and Reject). And, with a little work, we could easily come up with another sketch that has as little as three resources (Home, List, Item).

This highlights an important point and a reason it's a good idea to do these sketches early in the build process. Using different formats can lead to recognizing different patterns in your API and different ways to express the sequence diagrams. It's valuable to test things out, show them to others, and get feedback on what implementations seem too complex, too verbose, or just too confusing for your target developers. And since sketching is so quick and easy, there's no excuse for *not* creating lots of them in the beginning.

Remember, Sketches Are Disposable

Sketches are meant to be thrown away when you're done with them. This was mentioned early in the chapter and, based on my past experience, it's worth repeating. It's important to keep in mind that your API sketches are

just that—sketches. They’re meant to be created quickly and tossed aside just as quickly. Don’t spend too much time on them and don’t try to “fix” your sketches too much. It’s fine to adjust them, tweak them, and test them. But feel free to throw most of them away when you’re done.

Actually, you don’t *really* need to throw them away. Just place them in a folder in the project and don’t look at them anymore. The point here is to use sketches as a way to explore ideas, and it’s okay if some of those ideas aren’t very interesting. If you don’t like what you’ve sketched so far, just make more!

What's Next?

This chapter started our journey of turning our API design into an actual working API. You learned about the power of using sketches to explore possibilities and ideas without having to commit to writing lots of code. The idea for API sketching comes from architect Frank Gehry's habit of using sketches to explore his own building designs before committing to expensive detailed drawings and, ultimately, actual construction. Sketches are simple, cheap, and easy ways to test out your ideas and share them with your team and/or customers.

You also learned how to use the online Apiary API editor as a sketching tool. You don't need to use an online tool for sketches (you could just draw them out on a napkin). However, I like the Apiary tool because, along with the ability to easily create sketches, you can use Apiary's Mock Server feature to get them up and running at a live URL end point. That makes it possible to share your sketches with others; it even supports writing API calling code snippets to test your API sketch.

Finally, I shared some tips for creating helpful sketches, including (1) sketching your APIs in HTML to make them easy to test in a browser; (2) exploring different formats and API patterns that result in varying design, including reducing the number of resources in your API; and (3) remembering that your sketches are just simple explorations. If you don't like the ones you have, just toss them away and create more.

Chapter Exercise

In this exercise, you'll get a chance to create some sketches for the CreditCheck service we've been working on throughout the book. Starting from the diagram you created in the exercise in Chapter 5 ([Chapter Exercise](#)), use the Apiary editor to create a number of sketches.

First, create a simple HTML sketch that just contains the URL resources for your API, with notes on what will appear on each resource. Don't worry about filling in the resource pages with data; just define each resource to support HTTP GET. This sketch should give you an idea of the number of URLs you'll need when you finally code your API.

Second, create a new sketch that uses the JSON format. In this sketch, just define a resource that lists credit-check records ([/list](#)) and a resource that displays a single `ratingItem` record ([/list/123](#)). Focus on including all the properties that need to appear for a `ratingItem` record and what that would look like in a response. Don't worry about links to any other resource—just sketch out those two. This sketch should give you a pretty good idea of what a detailed JSON response would look like for your API.

For both sketches, be sure to use the Apiary Mock Server feature to test these out using your browser.

Finally, save both sketches to your local disk drive and place them in the [/assets/](#) folder of your project and check them in to source control using Git.

See Appendix 2 ([Solution for Chapter 6: Sketching APIs](#)) for the solution.

Footnotes

[45] <https://www.slideshare.net/rnewton/the-art-of-api-design-ronnie-mitra-director-of-api-design-api-academy-at-nordic-apis>

[46] <https://hyperallergic.com/265734/the-freeform-scribbles-that-give-rise-to-frank-gehrys-buildings>

[47] https://pragprog.com/titles/maapis/source_code

Copyright © 2020, The Pragmatic Bookshelf.

Chapter 7

Prototyping APIs

In the previous chapter, we discussed the idea of creating simple sketches of your API design. We saw from renowned architect Frank Gehry’s habit that creating quick sketches lets us test out our ideas and see how they might look before committing a great deal of resources and effort to actually building them out completely. Like Gehry, we can create simple examples of our planned APIs to get an idea of how they will “look” once they’re implemented. Sketches are simple explorations that can lead us to discover the design details that are most likely to solve our API problem.

One step past sketching is what I call *prototyping*. Prototypes are more than simple drawings or snippets of JSON or HTML. Prototypes have more detail, more depth, and more information in them. Unlike sketches—which are just lines on paper—prototypes are physical models of the design. They are multidimensional.

We can learn a lot from multidimensional API prototypes. They help us see how parts of the design relate to each other and how our proposed API design fits with other APIs. That’s what our prototyping does for us—it adds dimension and detail to our API designs.

What Is an API Prototype?

A prototype is a multidimensional model. These models can be the source material or the guide for creating the final full-sized object you’re creating. The very word *prototype* derives from Greek and translates as “original,” “first,” or “impression.” All of these words speak to the notion of creating a prototype *first* to help you work out details at a small scale. And that’s important for APIs too.

Prototypes are valuable because they allow you to turn your sketches into something more substantial without taking on too much time and added expense. They also let you actually practice your design and see it in detail from multiple angles. In essence, prototypes take the “surprise” out of building your final product.

Making a Toile

An excellent example of how prototyping can be valuable is found in the garment-making practice of creating a *toile* (from the French for “canvas” or “cloth” and pronounced “twall”; also known in the United States as a “muslin”) to test a sewing pattern before making the final garment. [\[48\]](#)

Imagine you, an accomplished dressmaker, have been handed a complex design that you’ll use to create an elaborate gown or suit for a special event, such as a movie premiere or a start-up’s initial public offering (IPO) celebration. These outfits are usually created out of rare and expensive fabric, and it’s probably *not* a good idea to make your first version of the garment using the final, expensive materials.



By using a cheap material, you can test out the pattern, measurements, layers, seam allowances, and so on to make sure the clothing will properly fit your customer and, if things don't quite work out in your first attempt, you can keep trying out ideas until eventually you get it right. Only then do you move on to create the final version of the gown using the high-end fabric.

Advantages of Prototyping

In the physical world, creating a one-off custom outfit or even mass-producing off-the-rack clothing can be time-consuming and involve a great deal of investment in materials, setup of production equipment, and validation of the final product for safety and usability. This can also be true in the virtual world of APIs. We need to allocate a budget for the work, put a team together (or take time from an existing team), and set up build-test-deploy pipelines (see Chapter 11, [Deploying APIs](#)) as we bring the API to life. And as in the physical world, it can be costly and frustrating to discover errors in the design if you're already in the process of creating the final product. API prototyping is a way to discover—and fix—possible problems early and at a low cost.

Solving problems early can save lots of time and money later in the virtual world of software. One study from 2009 found that fixing a bug in code costs six times as much as fixing it in the design phase.^[49] Imagine what it

would cost to start sewing your elaborate red-carpet gown only to find out your pattern is a couple of centimeters too short!

Limits of Prototyping

As valuable as prototyping can be, it has its limits. Because we’re creating our API prototype using “inexpensive materials,” we can only test the basic design of the API. We won’t be able to gain much information about how resilient or reliable the API will be until we build a more robust version of it. It’s hard to prototype *scale* for an API. We’ll need to use testing for that (see Chapter 9, [Testing APIs](#)).

Prototyping APIs can help you find problems in your design that you might not discover in simple sketches. (“Oh, that seems like too many parameters to try to send in a single payload.”) However, until you actually build a working version of the API, you won’t get to see all the problems. This is especially true when it comes to security issues with APIs. It isn’t always easy to create a prototype that accounts for all the possible security risks you may see in production. That’s why paying extra attention to security is so important (see Chapter 10, [Securing APIs](#)).

Finally, prototypes will help you work out interoperability details with other APIs. (“We need to make sure this API design can talk to the existing customer API.”) But lots of integration and interoperability details are only available once you have a working version of the API.

You may notice a pattern in the discussion here: API prototypes are most valuable when they’re a *working prototype*.

Working Prototypes

There’s more than one kind of prototype. A sketch (like the one we built in the last chapter) is sometimes referred to as a “paper prototype.” A “visual prototype” focuses on the appearance of the final product. That’s not too important for our API designs. You can also create a “user experience” or

“proof of concept” prototype. These come pretty close to looking and acting like the final product and are often the precursor to investing in creating another kind of prototype: the “working” or “functional” prototype. That’s the one that comes closest to the real thing without actually being a production-ready version.

Working prototypes can be expensive to build in the physical world too. Imagine a working prototype of an automobile, for example. The good news is that, in the virtual world of APIs, working prototypes are usually not very costly to create, especially when you have the proper editing and support tools. It’s even possible to create more than one prototype or to build several iterations of the same virtual prototype for your API without much added cost. And as it turns out, API practitioners have some very powerful tooling for API prototypes in the form of API definition formats like the OpenAPI Specification (OAS).^[50]

API Prototyping with OpenAPI

Earlier, in the story about creating a toile, you learned that garment makers rely on detailed design documents, called *patterns*,^[51] for directions on how to create the final piece of clothing. It turns out there is a whole world of pattern-making. Home patterns (called *sloper* patterns) are not the same as block patterns (used for industrial garment machines), and no matter the type of pattern, the process of adjusting or custom-fitting the pattern is called *grading*. All this pattern work relies on a standardized set of pattern symbols—a language of patterns—that the “cutter” or “sewer” needs to understand ahead of time.

This kind of “pattern language” exists when prototyping APIs too. The main GitHub repository for OAS describes it as “a standard, programming language–agnostic interface description for REST APIs.”^[52] The general goal of OAS is to accurately and comprehensively define an API so that API consumers don’t need access to any other information—just the OAS document—in order to understand and successfully use the API.

Tam’s API Specification

The roots of OAS reach back to then-developer Tony Tam’s 2009 project to create a definition language of RESTful APIs.^[53] He was working at an online dictionary company called Wordnik at the time. The first published version of what was then called SWAGR (pronounced “Swagger”) was released in 2011.^[54] Tam released version 2 of the Swagger Specification in 2014,^[55] and the most recent release, version 3, was published in 2018.^[56]

OAS isn’t the only API specification format, but it is one of the most widely used formats for RESTful APIs today.

RESTful?

RESTful?



The notion of “RESTful APIs” comes from the history of the HTTP protocol, common practices for programming the web, and Roy Fielding’s dissertation that defined the term “REST.” For a quick refresher on all that, you can check out Chapter 2, [Understanding HTTP, REST, and APIs](#).

Defining the API

One of the reasons OAS is so good for creating API prototypes is that it’s a rich format for *defining* the implementation details of APIs. It allows you to outline the URLs your API will publish, along with the inputs API users can send and the outputs those API users can expect to get in return. For example, you can publish all the API details we discussed in [The Basics of HTTP](#). Like the language of sewing patterns, OAS is an engineering language that allows you to accurately convert your prototypes into production.

For this reason, I refer to OAS as a definition format. Apiary Blueprint (which we learned about in Chapter 6, [Sketching APIs](#)) is also an API definition language. In contrast, we used the ALPS document format as our *description* language in Chapter 5, [Describing APIs](#). As we saw there, ALPS is a good way to describe the API design in general terms, but it isn’t appropriate for actually defining the implementation details of your production API.

Finally, you don’t need to define every single detail of your production API using your prototype, but OAS is one of the best formats for expressing those details. And that means your prototypes have the potential to be very explicit and, eventually, very accurate. That can eliminate any surprises when moving from the final prototype into production implementation.

OAS Versions and Formats

OAS Versions and Formats



The OpenAPI Specification has been around long enough that more than one version is in use today. I use version 3.0.0 in this book. Many organizations and OAS tooling also support version 2.0. And both versions support both YAML and JSON serializations of OAS documents. Be sure to check what version your team is using and that it's compatible with the OAS tooling you're using.

The following sections outline the steps to create a complete OAS document for API prototypes. We'll walk through the process of creating your OAS document and, along the way, learn the basics of how the OpenAPI Specification works.

Translating Your API Design into HTTP

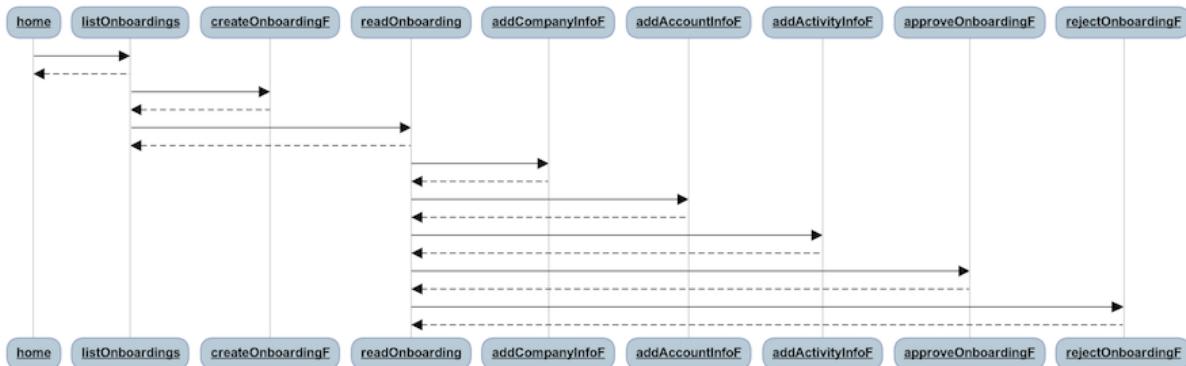
The critical element in all this work is to take our original API design into a valid HTTP design. We need to convert the list of properties and actions from our ALPS description in Chapter 5, [*Describing APIs*](#), into a well-defined HTTP API specification. This is where the magic happens—the art of translation. I use the word *art* because there is no science or engineering that can properly convert your design into a protocol-specific definition. We still need to rely on creative humans to do this work. That’s why sketching and prototyping can be so handy. It lets us work out decisions quickly and easily without a lot of expense.

It’s also important to point out that we’ve already decided that we’ll use an HTTP API as our target implementation. Not an event-driven React API. Not a query-driven GraphQL API. Not even a function-driven gRPC API. This decision to create an HTTP-based API is our “starting point.” We could imagine that someone at BigCo, where the Onboarding API will be built, has decided that for us. Or, in other cases, the API style might be determined by a customer, by some other team in the company, or just by ourselves. Deciding on the target is the *start* of turning your ALPS description into an implementation definition.

In Chapter 4, [*Designing APIs*](#), we designed our API and generated our web sequence diagram (WSD). We then translated that diagram into an ALPS description. Now we can use both of them as a reference in laying out our HTTP-style API.

A key element in HTTP-style APIs is a focus on individual *resources* (identified by a URL) and the ability to read and write *state* (the data properties associated with each resource). When we lay out our HTTP implementation, we need to declare our resources and determine which properties are displayed (and/or edited) for each of them.

The following is the output of our WSD diagram:



In that diagram, we see a set of actions (from left to right) that result in a completed (or rejected) onboarding record. If we were creating an event-driven (React) API or a remote procedure (gRPC) API, it would be an easy task to simply declare each of these elements in the diagram as an event or a procedure. However, that's not how HTTP-style APIs look. Instead, we need to squint a bit to see if we can find a set of *resources* that we can manipulate by changing their *state*.

For example, I notice words like `company`, `account`, and `activity` here. Those would likely be resources. I also see `home` and `onboardings`—these, too, are likely resources. Here's our list so far:

- `home`
- `onboardings`
- `company`
- `account`
- `activity`

Not bad. The next step is to associate read and write operations to those resources in a way that reflects the action elements in our design. For example, we could associate read and write actions to the `company`, `account`, and `activity` elements. We could also associate read and write actions to the `onboarding` resource (read a list of them, read one of them, create one of them). Finally, we could associate the approval and rejection actions with a

resource. It seems like a good idea to associate those actions with the `onboardings` resource, but I'm going to make a design decision and create a new resource named `status` to handle this. Now we can modify the state of a `status` resource (from `pending` to `approved` or `rejected`).

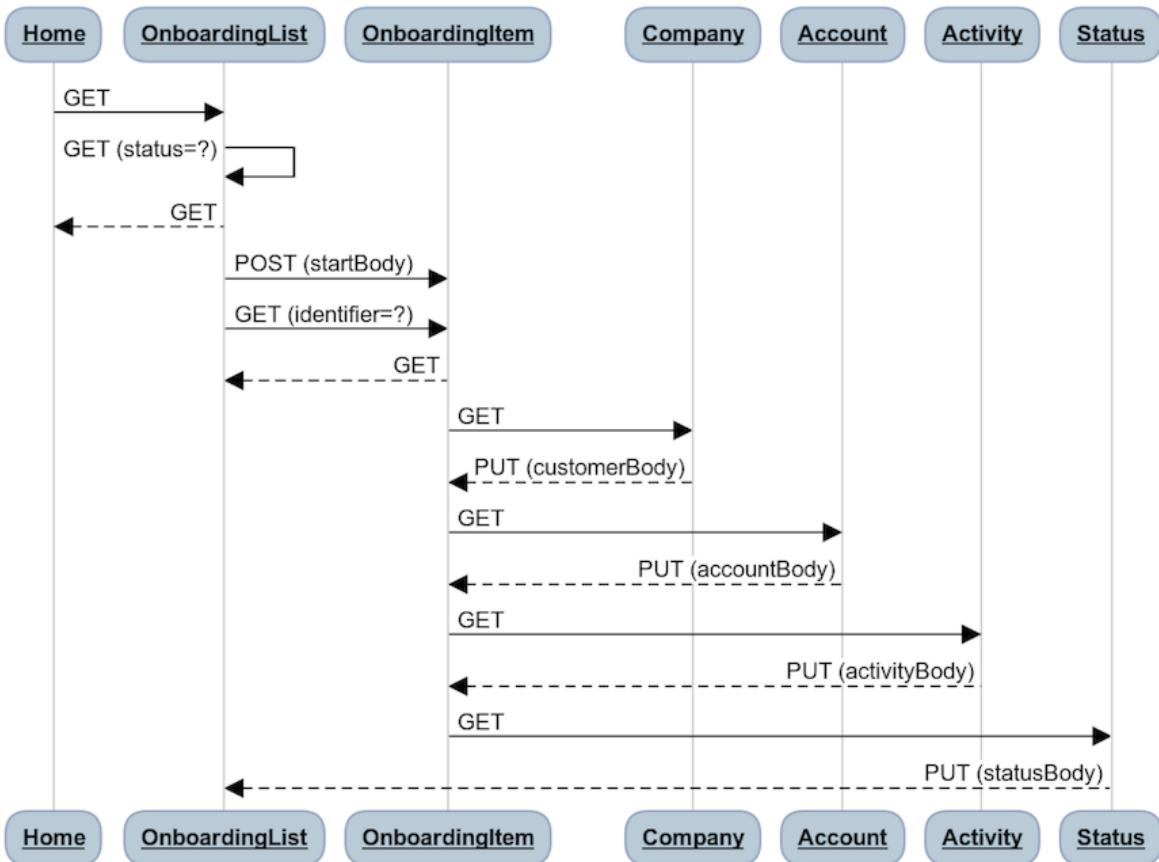
Here's our updated resource list with associated read and write actions:

- `home`—READ
- `onboardings`—READ list, READ item, WRITE new item
- `company`—READ item, WRITE item
- `account`—READ item, WRITE item
- `activity`—READ item, WRITE item
- `status`—READ item, WRITE item (approve, rejected)

That starts to look pretty close to an HTTP-style API, with defined resources and clear actions. Let's take one more step and convert our READ/WRITE terms into HTTP methods. This brings us even closer to a real HTTP implementation:

- `home`—GET
- `onboardings`—GET list, GET item, POST new item
- `company`—GET item, PUT item
- `account`—GET item, PUT item
- `activity`—GET item, PUT item
- `status`—GET item, PUT item (approve, rejected)

Now it looks *really* close to an HTTP API! We have resources and we have HTTP methods you can use against those resources. There's one more step: associating data properties with each resource. We'll handle that when we create our prototype definition file using the OpenAPI Specification in the next part of this chapter. For now, I'll show you a new WSD document that lists the resources and HTTP methods along with general hints on the kind of state we'll be modifying in each HTTP request. (You can find a copy in the code folder for this chapter.^[57]) Check it out the [diagram](#).



Don't worry if this diagram is a bit confusing. We'll work through the details in the following sections as we build up our API definition document.

Creating Your OpenAPI Document with SwaggerHub

The “engineering” document we’ll use to prototype our API is the OpenAPI Specification. The advantage of OAS documents are that they’re easy to create and edit. There are also several supporting tools that can read OAS documents and generate things like server-side sample code, example API client applications, and human-readable documentation. These are all great shortcuts or prototypes that we can use to validate our design before committing expensive resources to building the production version of our API.

OAS documents have a well-defined structure. An OAS file includes three main sections at the top level:

- The **info** section holds such information as the title, a brief description, and some additional identifying data.
- The **components** section is where we’ll place our request and response definitions and other reusable chunks of API specification data.
- The **paths** section is where we’ll define the exact URLs, response codes, inputs, and outputs for each endpoint in our API.

With that quick overview, let’s start writing up our OAS document for BigCo, Inc.’s Onboarding API.

Logging Into SwaggerHub

We’ll use the online editor at SwaggerHub to create our OAS file. You can create your OAS documents in any editor, but I like to use the one at SwaggerHub since it offers lots of syntax help and is a good place to host a

“mock” version of the API too. We’ll cover API mocks later in this chapter in [Mocking Your API](#).

To start, visit the landing page for SwaggerHub at <https://swagger.io/tools/swaggerhub>. If you already have an account, you can just click the Sign In button located near the top-right corner of the page. If you’re new to SwaggerHub, click the Try Free button to create a new account. Creating an account is easy; you can even use your existing Github account as your identity at SwaggerHub. This makes sharing your SwaggerHub content with your GitHub repositories really easy too.

Swagger or OpenAPI?



Originally, the specification was known as “Swagger Specification.” Nowadays the proper name is OpenAPI Specification, or OAS. However, lots of the tooling associated with OAS still uses the name “Swagger.” I’ll continue to follow this practice here.

Once you’ve logged in, you’ll be forwarded to your “My Hub” page. This page shows all the OAS documents associated with your personal account and with any organizations where you are a member, as shown in the following screenshot.

The screenshot shows the SwaggerHub web application interface. At the top, there's a header with the SmartBear logo, a search bar, and user account information. Below the header, the main area is titled 'MY hub'. On the left, there's a sidebar with a 'Create New' button and a 'Search' input field. The main content area displays three API documents listed in a table:

Owner	Title	Description	Actions
mamund	onboarding-master	This is the OAS document for BigCo Onboarding API	API OAS3
amundsen	bigco-onboarding	API for bringing on new customers at BigCo, Inc.	API OAS3
mamund	onboarding-api_forms.json	Onboarding API for BigCo, Inc. This is a fORMS+JSON API sketch	API

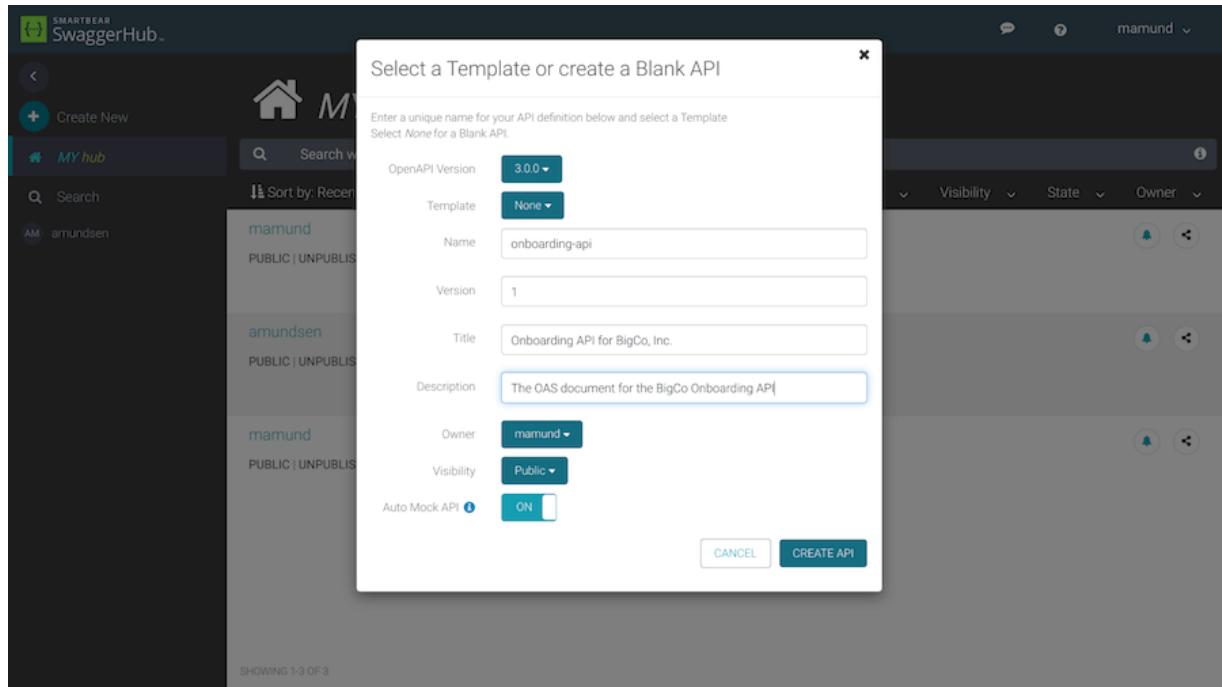
At the bottom of the main content area, it says 'SHOWING 1-3 OF 3'.

To add a new OAS document to your hub, click the Create New icon in the left sidebar to call up the Template dialog box. Here you can add some basic information about your API to start the editing process, as shown in [the screenshot](#).

For this API, enter the following data:

- OpenAPIVersion: **3.0.0**
- Template: **None**
- Name: **onboarding-api**
- Version: **1.0.0**
- Title: **Onboarding API for BigCo, Inc.**
- Description: **The OAS document for the BigCo Onboarding API**
- Owner: (your accountname)
- Viability: **Public**
- Auto Mock API: **ON**

Most of these settings are the default. Be sure to select “None” for the template and fill in the **Name**, **Version**, **Title**, and **Description** fields too.



Once you've updated the information in the dialog box, press the Create API button. The SwaggerHub editor will create a new OAS document in your hub using your settings and then open that document for editing. Your SwaggerHub editor page should look similar to the one shown in the [screenshot](#).

```

openapi: 3.0.0
info:
  version: '1.0.0'
  title: 'Onboarding API for BigCo, Inc.'
  description: 'The OAS document for the BigCo Onboarding API'
paths: {}
# Added by API Auto Mocking Plugin
servers:
  - description: SwaggerHub API Auto Mocking
    url: https://virtserver.swaggerhub.com/amundsen/onboarding-api/1.0.0
  
```

No content to display
Want [Help Building your API](#)

Last Saved: 2:17:05 pm - Dec 18, 2019 VALID

Now you're ready to start building up your OpenAPI engineering document for the Onboarding API.

Info Section

The `info` section of the OAS file holds general information about the OpenAPI Specification, such as the `version` number of the specification (not the version of the implemented API, just the specification document), the `title`, and the `description`. It turns out that you supplied these values when the SwaggerHub editor asked you to enter data to start the editing process. I also add a comment block that contains some additional metadata about the specification, as shown in the following code:

ch07-prototyping/info-section.yaml

```
openapi: 3.0.0

#####
# BigCo Onboarding API
# 2020-16-01
# Design And Build Great APIs
# Mike Amundsen (@mamund)
#####

### general information block ###
info:
  version: '1.0.0'
  title: 'BigCo Onboarding API'
  description: 'This is the OAS document for BigCo **Onboarding API**'
```

A Mock Server, Too

A Mock Server, Too



You'll see another section appears in the document loaded into SwaggerHub—the `servers` section. The information there was supplied by SwaggerHub to support a mock service for this API definition. The word *mock* means “not authentic” or “artificial.” Mock API servers *act* like the real thing but are just fakes. I’ll cover the mocking aspect of SwaggerHub a little later in this chapter.

Components Section

Another important section in an OAS document is the `components` section. Technically the `components` section is an optional section, but I use it all the time and consider it key to a well-formed OpenAPI Specification document. The `components` section holds content that’s referred to in other parts of the definition—mainly the `paths` section that we’ll deal with next. Several elements of an HTTP CRUD-style API are often repeated. Placing the definition of those items in the `components` section *once* and then referring to them many times in the `paths` section is a good way to properly build your API definition.

To start, add the following at the bottom of your OAS document:

```
ch07-prototyping/components-template.yaml
```

```
components:  
  
  parameters: {}  
  requestBodies: {}  
  responses: {}  
  schemas: {}
```

This declares the `components` section and several important subsections. The subsections are where we’ll place the shared definitions that we’ll use in the `paths` section.

Most of the interesting work of defining your API will happen in the **components** section. For now, let's add a few things here to illustrate how the section can help you build solid API definitions.

Parameters

The **parameters** subsection of the **components** section is where you can define API-related items like HTTP header parameters, URL elements, query strings, and even browser cookies. For example, the following example defines a query string that can be added at the end of an existing URL to filter the records returned by their **status** value:

ch07-prototyping/status-parameter.yaml

```
status:  
  name: status  
  description: status flag  
  in: query  
  schema:  
    type: string  
    enum:  
      - pending  
      - completed  
      - abandoned  
    example: "pending"
```

For example, if the URL was this:

<http://api.example.org/records>

The above **status** query parameter could be added to make the URL look like this:

<http://api.example.org/records?status=pending>.

Here's another example of a handy **parameter** definition:

ch07-prototyping/identifier-parameter.yaml

```
identifier:  
  name: identifier  
  description: record identifier  
  in: path  
  required: true  
  schema:  
    type: string
```

The `identifier` parameter definition (above) is a *path* parameter. Path parameters appear within the URL itself (not after the `?` near the end). Using the same base URL example from the previous definition, if the `identifier` was `123`, the URL would look like this:

<http://api.example.org/records/123>

Finally, we'll define HTTP header parameters to support something called "conditional requests." These are instructions to the HTTP server to only complete our requests if the identified condition exists. For example, the following is the header definition that will ensure a new record is created *only* if there isn't an existing record at the identifier URL:

ch07-prototyping/header-parameter.yaml

```
ifNoneMatch:  
  name: if-None-Match  
  description: Conditional Create Header  
  in: header  
  schema:  
    type: string  
    example: "*"
```

This last example might be a bit confusing—you'll see it in action when we get to the work of actually coding the API server in the next chapter. For now, just keep in mind that we can define headers as shared components to be used in other places in the OAS document.

That's a quick tour of the `parameters` subsection of the `components` section. You can check the completed section in the code folder for this chapter.[\[58\]](#)

RequestBodies

The `requestBodies` subsection of the `components` section is where you define any input blocks for HTTP write operations. Any time you use HTTP POST, PUT, or PATCH, you can pass your input values using an HTTP request body. This section of the OAS document is where you can define one or more of those HTTP request bodies.

Defining a request body in an OAS document is pretty simple. Here's one we'll use to write `company` data to our Onboarding API:

```
ch07-prototyping/company-body.yaml
```

```
company:
  content:
    application/x-www-form-urlencoded:
      schema:
        $ref: '#/components/schemas/company'
    application/json:
      schema:
        $ref: '#/components/schemas/company'
```

This definition says that a request body named `company` that contains `content` may appear in two forms. The customer data might appear formatted as `application/x-www-form-urlencoded` data (that's the way HTML forms send data). Or the data might appear as simple `application/json` data (that's the most common way to send data for JSON-based APIs). You'll note that the actual definition of *what* a company body looks like is referenced by the line `$ref: '#/components/schemas/company'`. This refers to another part of the `components` section (the `schemas` subsection) that I'll cover shortly.

Typically, an OAS API specification will have several definitions in the `requestBodies` section. You can look at the full set of them in the code folder associated with this book.

Responses

Just as we defined inputs blocks for HTTP requests, we need to define output blocks for HTTP responses. In OAS documents, those appear in the **responses** subsection. In our API, we only need a few defined HTTP responses. One that we'll need to have is a general error response. There will be times when an API request is badly formed or the service is unable to process the incoming request due to an error on the server side. In both those cases, we still need to send a response back to the API client. And that response *should* include enough information to explain the problem and, if possible, guide the API client app (or the human assigned to create the client app) into fixing the problem and resending the request. Here's the OAS response definition we'll use as our error response:

ch07-prototyping/error-response.yaml

```
error:
  description: Unexpected error
  content:
    application/problem+json:
      schema:
        $ref: '#/components/schemas/error'
    application/json:
      schema:
        $ref: '#/components/schemas/error'
```

Notice that OAS responses look a lot like OAS **requestBodies**. In the previous example, you see a **description** and then the **content** element that tells us the response could appear in two different formats: the **application/json** format and the **application/problem+json** format. This second format was designed specifically to carry helpful API error information. You can read more about the Problem+JSON standard in RFC7807.^[59]

In our API, we'll only need three response definitions:

- **#/components/responses/created**—Used whenever a client creates a new onboarding record

- `#/components/responses/reply`—Used as the general response whenever a client successfully reads or writes an onboarding record
- `#/components/responses/error`—Used whenever an unexpected error occurs

You can check out the complete set of `responses` in the code folder for this chapter.

Schemas

The last subsection of the `components` section I'll cover here is the `schemas` section. I saved the best for last. The `schemas` subsection is the part of OAS components that's used the most. It holds all the input and output blocks you'll need in your API. And that can be quite a few. For example, our Onboarding API has 10 models to define. Some APIs have many more.

We won't dwell on all of them here; but as an example of what `schema` definitions look like, the following is the schema for our `error` model:

`ch07-prototyping/error-schema.yaml`

```
error:
  type: object
  properties:
    type:
      type: string
      example: "Invalid Record"
    title:
      type: string
      example: "One or more missing properties"
    detail:
      type: string
      example: "Review the submitted record for missing required
properties."
```

Notice that the `error` schema is declared as an `object` with a handful of `properties` and that each property has a `type` and an `example` value. Several

values are possible for `type`, and you can add additional properties, such as `minimum`, `maximum`, `minLength`, `maxLength`, and many others. Becoming good at defining OAS schemas is a skill unto itself. Since we're only using OAS documents for prototyping, we can stick to very simple schema definitions. However, if you plan on using the OpenAPI Specification as your exact API engineering document, you'll want to learn more about OAS schemas. Check out the OpenAPI Specification documentation for details.^[60]

If you want to see the full set of schema models I defined for the Onboarding API, you can check them out in the code folder associated with this chapter.

Okay, we've defined a lot of components: parameters, request bodies, responses, and schemas. Now it's time to bring all that together and define actual HTTP API endpoints. That happens in the `paths` section.

Paths Section

The most visible feature of OAS documents is in the `paths` section. That's where each HTTP endpoint is defined, along with the identification of all the supported methods, input and output formats, and parameters. Everything we've done up to this point leads to creating entries in the `paths` section of your OpenAPI document.

In OpenAPI documents, each path definition starts with an HTTP URL. That's followed by one or more HTTP method, and each method can have details on parameters, request bodies, and responses. The following is a template for a path definition:

ch07-prototyping/path-template.yaml

```
/sample-url:  
  get:  
    operationId: readSample  
    summary: Use this to read a sample file
```

```

parameters: []

responses:
  '200':
    description: OK

put:
  operationId: updateSample
  summary: Use this to update a sample file

parameters: []

requestBody:
  content: {}

responses:
  '200':
    description: OK

```

The example shows the endpoint URL (in this case, `/sample-url`) and two methods (`get` and `put`). You can have a definition of one or more HTTP methods (`get`, `put`, `patch`, `post`, and `delete`) for each endpoint. You'll also see an `operationId` and a `summary` property for each method. These aren't required but are highly recommended. After that you see placeholders for `parameters`, `requestBody`, and `responses`. The `requestBody` portion of a path definition is only needed for write operations (`post`, `put`, and `patch`).

Using the previous template as a guide, here's a path definition for the `/start` endpoint in the Onboarding API:

ch07-prototyping/start-path.yaml

```

/start:
post:
  operationId: startOnboarding
  summary: Use this to start process of onboarding a company

parameters:
  - $ref: "#/components/parameters/ifNoneMatch"

```

```
requestBody:  
  $ref: "#/components/requestBodies/start"  
  
responses:  
  '201':  
    $ref: "#/components/responses/created"  
  
  default:  
    $ref: "#/components/responses/error"
```

You can see that each of the key portions of the definition ([parameters](#), [requestBody](#), and [responses](#)) refers to one of the shared definitions in the [components](#) section. Notice that the [parameters](#) section is an array in YAML (the - character) and that the [responses](#) section supports more than one possible response reference, depending on the HTTP status code. I've only defined the most common status codes (200 and 201) for this prototype and then relied on the [default](#) response to return the [error](#) block we defined earlier. If you were documenting a production API, you'll probably add more possible responses here.

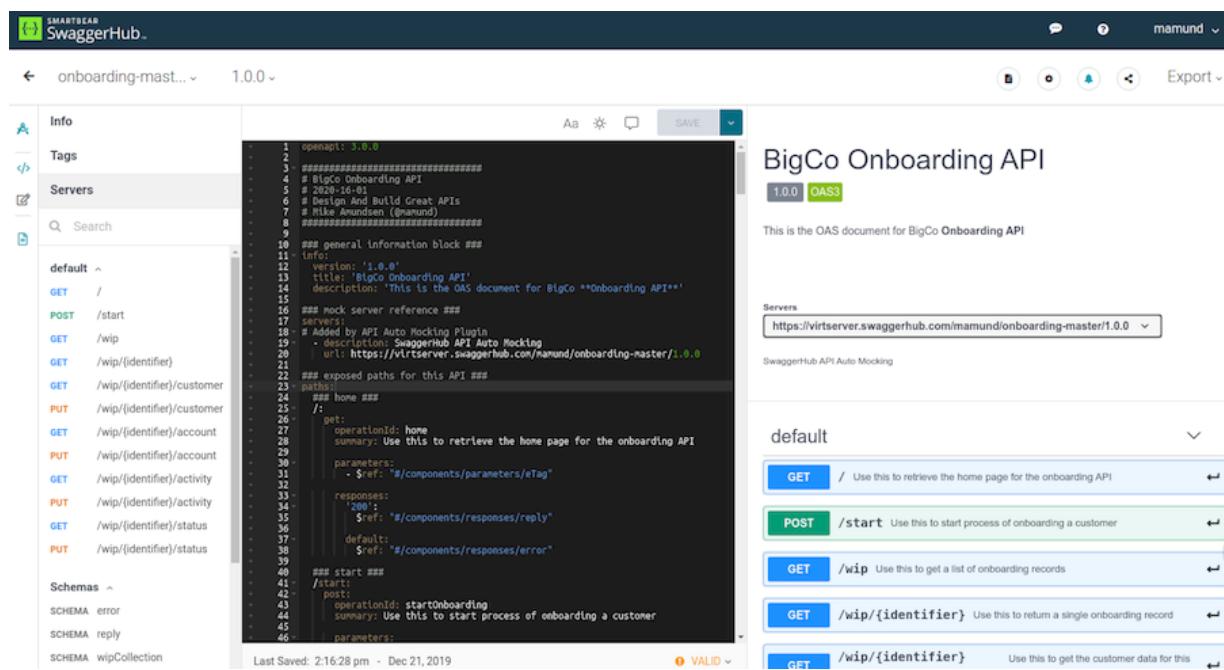
I created eight entries in the [paths](#) section of the OAS document to implement my prototype for the Onboarding API. Depending on the size and complexity of your API, you could have many more. Keep in mind that this is just a prototype, and I worked to keep the OAS file rather basic in order to focus on the format itself and not get too tied up in my design for this book.

You can check out all the endpoints in the definition document located in the code section associated with this chapter.

Saving and Exporting Your API

Now that you have a good idea of what an OAS document looks like, we can cover saving and exporting a completed OAS document. For this part of the book, go ahead and load my completed Onboarding API OAS document from the code folder for this chapter and paste it into your open editor.^[61] You can use the initial `onboarding-api` definition you started at the beginning of this lesson, or use the SwaggerHub editor to create a brand new OAS definition. Whichever you use, just copy-paste the OAS file from disk into the web page and let the SwaggerHub editor parse the document for any bugs or problems.

Once you paste the full document into the editor, it should look like this:



The screenshot shows the SwaggerHub editor interface. The left sidebar has a tree view with nodes like 'Info', 'Tags', 'Servers', and 'default'. The main area displays the OAS document content. The right side shows the API title 'BigCo Onboarding API' (version 1.0.0, OAS3), a summary, and a 'Servers' section with a URL. Below that is a 'default' section with several API operations listed.

```
openapi: 3.0.0
#####
# BigCo Onboarding API
# 2020-10-07
# Design And Build Great APIs
# Mike Amundsen (@mamund)
#####
## general information block ##
info:
  version: '1.0.0'
  title: 'BigCo Onboarding API'
  description: 'This is the OAS document for BigCo **Onboarding API**'
##
## mock server reference ##
servers:
  - Added by API Auto Mocking Plugin
    - description: Swaggerhub API Auto Mocking
      url: https://virtserver.swaggerhub.com/mamund/onboarding-master/1.0.0
##
## exposed paths for this API ##
paths:
  ## home ##
  /:
    get:
      operationId: home
      summary: Use this to retrieve the home page for the onboarding API
      parameters:
        - $ref: '#/components/parameters/eTag'
      responses:
        '200':
          $ref: '#/components/responses/reply'
        default:
          $ref: '#/components/responses/error'
  ## start ##
  /start:
    post:
      operationId: startonboarding
      summary: Use this to start process of onboarding a customer
      parameters:
```

You can save your OAS edits using the Save button that appears in the top-right corner of the edit window. This button will only be enabled if you've made changes since you last saved the document. You can use the drop-down arrow next to the Save button to save the loaded document as a new OAS file or to revert to the previous saved version of this file.

You can also export the OAS definition document onto your local machine. To do that, click the Export drop-down arrow to see a list of options. The last item on the list is Download API. Click the arrow to the left of that phrase to see a sub-menu of download options. Selecting YAML Unresolved will download the file exactly as you see it in the editor. This preserves all of the references, comments, and other details of your document. I like to do this and store the document in my project repository.

You can also export a *resolved* version of your API definition in a ZIP file. This version will replace all the parameter `$ref` elements with the actual definition from the `components` section and rearrange the order of the file slightly. This ZIP file also includes both a YAML and a JSON version of the API definition, which can be handy since some tools only work with one or the other format. I like to download this ZIP file as a backup for what's stored in my SwaggerHub account in order to have quick access to both the YAML and JSON versions of the file.

Now that you have a complete OpenAPI definition loaded into SwaggerHub, you're ready to try out a working version of our Onboarding API using the mocking service.

Mocking Your API

Mocking your API is your chance to see a working version of the API up and running. SwaggerHub has a built-in API mocking service. It reads your OAS document and uses the definitions (and the example values) in that document to set up a mock API server that you can use to test against. As you may recall from the early part of this chapter in [Working Prototypes](#), the real value of an API prototype is that it allows you to test out the interface, and mocks are an excellent way to do that.

To get the SwaggerHub API mock up and running, you need to make sure you have a `server` element in your OAS document that points to the SwaggerHub `virtserver`. We turned it on when we created our OAS file, but you can also add it later by clicking the name of your loaded definition file in the upper-left corner of your editor screen and then clicking the Integrations tab in the drop-down menu. When you do that, you'll see API Auto Mocking as one of the options. To establish the mocking service with all the defaults, just press the dark arrow on the API Auto Mocking line. That will initiate an instance of the mocking service for you based on the loaded API definition file. It will also add/update a `servers` entry in your OAS file that'll look something like this:

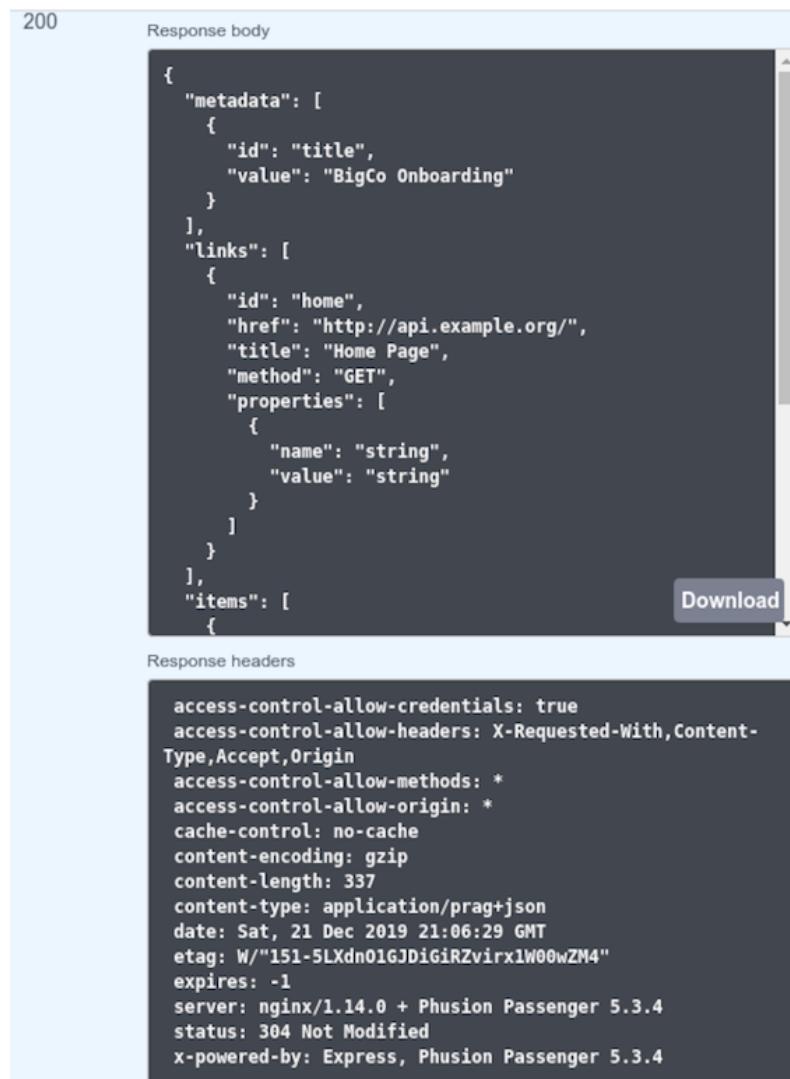
ch07-prototyping/mock-server.yaml

```
### mock server reference ###
servers:
# Added by API Auto Mocking Plugin
- description: SwaggerHub API Auto Mocking
  url: https://virtserver.swaggerhub.com/mamund/onboarding-master/1.0.0
```

With this up and running, you can now start testing the API endpoints. You can use the editor UI, do it from an external tool like curl, or even use an existing API client application. The UI version is easy to use. Just click one of the endpoints in the right-side documentation section and press the “Try it

out” button. When you do, a blue Execute button will appear. When you click that, the editor will run the request and display the response in a window below the button.

The following is a sample of the output for the Home endpoint (/):



The screenshot shows the Swagger UI interface for a Home endpoint. At the top left, it says "200 Response body". Below that is a JSON response object. On the right side of the JSON area, there is a "Download" button. At the bottom left, it says "Response headers". Below that is a list of HTTP headers. The JSON response body is as follows:

```
{  
  "metadata": [  
    {  
      "id": "title",  
      "value": "BigCo Onboarding"  
    }  
  ],  
  "links": [  
    {  
      "id": "home",  
      "href": "http://api.example.org/",  
      "title": "Home Page",  
      "method": "GET",  
      "properties": [  
        {  
          "name": "string",  
          "value": "string"  
        }  
      ]  
    }  
  ],  
  "items": [  
  ]  
}
```

The Response headers section lists the following:

```
access-control-allow-credentials: true  
access-control-allow-headers: X-Requested-With,Content-Type,Accept,Origin  
access-control-allow-methods: *  
access-control-allow-origin: *  
cache-control: no-cache  
content-encoding: gzip  
content-length: 337  
content-type: application/prag+json  
date: Sat, 21 Dec 2019 21:06:29 GMT  
etag: W/"151-5LXd01GJDgRZvrx1W00wZM4"  
expires: -1  
server: nginx/1.14.0 + Phusion Passenger 5.3.4  
status: 304 Not Modified  
x-powered-by: Express, Phusion Passenger 5.3.4
```

If you scroll up a bit, you’ll also notice that the editor UI displays a box showing a curl command-line example that calls the same endpoint. That should look something like this:

```
curl -X GET \  
  "https://virtserver.swaggerhub.com/mamund/onboarding-master/1.0.0/" \  
  -H "accept: application/prag+json" \  
  
```

```
-H "ETag: zaxscdvfbg"
```

When you execute that at the command line, you'll get a response similar to the one shown in the UI screenshot shown earlier.

You can also test out write operations using the SwaggerHub mock service. It won't really write any data to a server, but it will "mock" all the steps, including making sure you supply the proper arguments. A curl-ed version of the [updateCompanyData](#) action looks like this:

```
curl -X PUT \  
  "https://virtserver.swaggerhub.com/.../1.0.0/wip/q1w2e3r4/company" \  
  -H "accept: application/prag+json" \  
  -H "If-Match: zaxscdvfbg" \  
  -H "Content-Type: application/x-www-form-urlencoded" \  
  -d "identifier=q1w2e3r4&companyName=SmithCo"
```

Note that, for the purposes of display in this book, I shortened the URL line (...) to make sure it fit on the page. You'll also notice that the responses for all these tests are the same. The mock server just responds with the [#/components/reply](#) response defined within the OAS file.

Now that you know you can test the API using both the SwaggerHub editor interface and tools like curl, you have a working prototype you can use to test your ideas, show to others, and—when needed—modify the prototype to fix bugs and improve the interface in order to get it all set for the final work of the actual code-based implementation.

Before I wrap up this chapter, I want to cover one more thing you can do with OAS files: generate basic documentation for your API.

Generating Your API Documentation

No matter how well you design your API, both server developers and API client developers will need some level of documentation in order to understand and use your API. And the OpenAPI Specification has some really nice support for generating simple documentation.

The API documentation is very much along the lines of “technical” docs: it only includes the set of endpoints and the associated parameters, request bodies, response bodies, and schemas you included in your document. There is no extended prose covering *why* someone might want to use your API—nothing like a how-to guide or common problems list or anything like that. But basic technical docs are a good start.

SwaggerHub’s Generated Documentation

You can use SwaggerHub to generate stand-alone HTML documentation with just a few clicks of the mouse. The “Export” menu in the SwaggerHub editor gives you three options. I’ll describe them here briefly and let you explore them yourself:

- *dynamic-html*—This is a NodeJS website you can host that renders the documentation in a fairly simple layout. It might make a nice “starter” site for your API.
- *html*—This is a static site that lists all the operations and has a very simple layout. I’m not a fan of this one, but I like that it’s a single, stand-alone page. No need to host a NodeJS site.
- *html2*—This is also a static site, but it has the added feature of showing API calling examples in a number of languages (curl, Java, Objective-C, JavaScript, C#, PHP, Perl, and Python). I like this one a bit better, but the UI is still pretty simple.

External API Documentation Tooling

Many other tools and services will render your OAS documents. One that I've been using quite a bit recently is known as ReDocly, [\[62\]](#) which currently offers an open-source GitHub repository where you can download the code and read through examples. The company behind ReDocly is also planning to launch an online hosted version of the service in the future. [\[63\]](#)

The thing I like best about ReDocly is that you can use the HTML template page that relies on its hosted JavaScript library to render a very nice HTTP API documentation site very easily. All you need is the ReDocly HTML template and a URL that holds the OAS document. For this book, I placed a copy of the prototype OAS definition file in a public GitHub repository and then updated a copy of the ReDocly HTML page to point to that file, and then I had a fully working set of API documentation. (See the following screenshot.)

The screenshot shows two side-by-side views of the ReDocly API documentation. The left view is a detailed API endpoint description for a PUT request to '/wip/{identifier}/status'. It includes sections for PATH PARAMETERS (with 'identifier' as required string), REQUEST BODY SCHEMA (application/json with 'identifier' and 'status' fields), and RESPONSES (200 OK with application/prag+json schema for metadata, links, and items). The right view shows the corresponding Request and Response samples in JSON format. The Request sample shows a payload with 'identifier' and 'status' fields. The Response sample shows a 200 OK response with a single item in the 'items' array containing 'id', 'value', and 'links'.

```
PUT /wip/{identifier}/status

Request samples
Payload
Content-type: application/json
{
  "identifier": "q1w2e3r4",
  "status": "pending"
}

Response samples
200 default
Content-type: application/prag+json
{
  "items": [
    {
      "id": "title",
      "value": "BigCo Onboarding",
      "links": [
        ...
      ]
    }
  ]
}
```

ReDocly is just one of a number of options for generating documentation from your OAS file. Your API team may already have something in use today. No matter what the tool is, as long as you can use it to create basic technical docs, you'll be one step farther along the path to releasing your completed API.

What's Next?

In this chapter, we covered the concept of generating prototypes for your API as a final step before committing resources and time to writing the actual code to implement your design. For our API work, we used OpenAPI Specification (OAS), formerly known as Swagger, to build up our working prototype in detail from our initial design. You learned how to use the OAS document elements `info`, `components`, and `paths` to build up the definitions of our endpoints, and how to use `parameters`, `requestBodies`, and `responses` to fill in the details of each API URL and method in our prototype.

Finally, you learned to use the SwaggerHub editor to do our live editing, generate our mock server for prototype testing, and generate technical API documentation. You also learned how to use an external documentation generator—ReDocly—to quickly and easily create interactive API documentation from our OAS file.

In the last two chapters we've advanced our original ALPS-based design using Apiary's Blueprint format to create sketches and the OpenAPI Specification to create prototypes. Having done all that, we're now ready to take our prototype to the next stage: actually writing code and implementing a fully functional API that we can test, secure, and eventually deploy.

Writing code is what I'll cover in the next chapter.

Chapter Exercise

In this exercise, you get to convert your credit-check sketch into an API prototype using the OpenAPI Specification format. When you’re done with this exercise, you should have a working prototype of your Credit-Check API that you can use as a reference when we move on to writing code in the next chapter.

To start, log in to the WSD editor at <https://www.websequencediagrams.com> to create a new WSD document that reflects your translation of your credit-check-alps and credit-check-diagram.wsd into an HTTP-style API. This is the creative part of the exercise. There’s no right or wrong answer. Once you’re done, export your PNG and text versions of the WSD diagram and add them to the `assets` folder of your Onboarding repository. Commit your changes using `git`.

Next, log in to your account at SwaggerHub (<https://swaggerhub.com>) and start a new version 3 OAS document named `creditcheck-api` using the same defaults we used in this chapter. Next, using the components template and the path template we reviewed in this chapter, fill out your CreditCheck prototype with all the parameters, request bodies, responses, and schemas you’ll need to complete your paths section of the OAS document. When you’re done with this step, you should have a fully defined CreditCheck API.

Next, use the SwaggerHub mocking service to run some simple tests of your interface. If needed, perform any minor bug fixes or interface changes.

Once you are done making any changes to your OAS document, export your API definition as both a “YAML Unresolved” and a “YAML Resolved” output. Extract the JSON and YAML documents from the ZIP

file and, along with the “unresolved” YAML file, check them into the `assets` folder of your `repo` and commit your changes using `git`.

Finally, set up a folder for your API technical documentation and use the ReDocly HTML template (<https://github.com/Redocly/redoc>) to create your own interactive API documentation for your CreditCheck API. Once you’ve completed your API technical documentation, add a new folder to your repo (`/documentation`), place the HTML file template into that folder, and commit your changes using `git`.

See Appendix 2 ([*Solution for Chapter 7: Prototyping APIs*](#)) for the solution.

Footnotes

- [48] <https://en.wikipedia.org/wiki/Toile>
- [49] [https://www.researchgate.net/publication/255965523 Integrating Software Assurance into the Software Development Life Cycle SDLC](https://www.researchgate.net/publication/255965523_Integrating_Software_Assurance_into_the_Software_Development_Life_Cycle_SDLC)
- [50] <https://openapis.org>
- [51] [https://en.wikipedia.org/wiki/Pattern_\(sewing\)](https://en.wikipedia.org/wiki/Pattern_(sewing))
- [52] <https://github.com/OAI/OpenAPI-Specification>
- [53] <https://twitter.com/fehguy>
- [54] <https://developer.wordnik.com/docs>
- [55] <https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md>
- [56] <http://spec.openapis.org/oas/v3.0.2>
- [57] https://pragprog.com/titles/maapis/source_code
- [58] https://pragprog.com/titles/maapis/source_code
- [59] <https://tools.ietf.org/html/rfc7807>
- [60] <https://swagger.io/specification>
- [61] https://pragprog.com/titles/maapis/source_code

[62] <https://github.com/Redocly/redoc>

[63] <https://redoc.ly>

Copyright © 2020, The Pragmatic Bookshelf.

Chapter 8

Building APIs

In the last two chapters, we covered the work of converting our API design in the form of our ALPS document into sketches and prototypes. All of this preliminary work was done to explore possibilities and lay the groundwork before we actually committed ourselves to the expensive act (both in time and money) of writing code. The good news is that we're done with the preliminaries. It's now time to crack open our code editor and write some NodeJS that results in a fully functional working API!



The role of Gehry's sketches, which we looked at in [Learning from Frank Gehry's Sketches](#), is to test ideas out quickly. The dressmaker's practice of

making a toile or muslin (see [Making a Toile](#)) before committing to creating the final garment is a way to ensure you've explored as many aspects of a complicated design as you can before you actually turn it into a reality. Now, having taken the time to do a bit of exploration, we should have enough feedback from our potential users and enough evidence from our prototyping work to know which design we want to use and how we want to approach it. This feedback and evidence will guide us as we finally work through the process of writing our code and getting the API up and running.

We All Get Here Eventually



I suspect some readers have been patiently reading along in hopes of reaching this point in the “API story.” To all of them, I say your reward is at hand! I also suspect at least a few of you have skipped over some of the earlier model, design, describe, sketch, and prototype exercises in order to hasten your arrival at this critical point. That’s okay. However, I still think you should (eventually) go back to make sure you get the most out of the experience in the design and build phases too.

Defining the API Build Process

For the purposes of this book, the “build process” is the work of translating our design work into source code that actually does the work we set out to do. Sometimes the word *build* is reserved for that portion of coding that starts with checking in written code and involves things like compiling, testing, and deploying the code on a server somewhere. I’ll cover all that in the next and final part of the book (The Release Phase). For now, we’ll focus on writing code to match our design.

Our build process will be the work of turning the prototype we created in the previous chapter into our ultimate monument: the Onboarding API for BigCo, Inc.

Relying on a Repeatable Process

If you only build one API in your life, it's not too important just *how* you do it as long as you get it done. However, for those who need to build lots of APIs—those whose job it is to convert designs into functional code—it's valuable to have a consistent, repeatable process. Just as we covered processes for modeling, designing, describing, sketching, and prototyping APIs, we need to come up with a reliable process for the act of building the actual API.

The main benefit of a repeatable process is that it's something you can show and teach others. A good process is also easily monitored. For example, in the design process, we can easily ask teams things like, "Have you completed your model yet?" or "Can you show me your ALPS description for this API?" These processes become a kind of shared understanding everyone can rely on when talking about their work. This is especially handy when you have teams in various locations around the world. When even speaking the same language can be a challenge, the ability to share API design and build processes can make working together easier and more efficient.

With all this in mind, the build process I want to share with you is one that I've been using for quite a while. It's a process that covers the main portions of code every API needs deal with. For this book, I've even created a small framework in NodeJS based on the same process. That should make it easier to show you the main concepts and enable us to get our code up and running quickly.

The following is a short walk-through of the build process I use called DARRT and how we'll use it to build working APIs from your design documents with NodeJS.

What Is DARRT?

What Is DARRT?



DARRT is a process for building code from design documents. I created the DARRT framework as a teaching tool and as a quick way to start lightweight microservice and API projects. The version you're using here is a downsized edition of the one I use in my own projects. This version is not a production-ready implementation. I took lots of shortcuts and left out complicated error-handling code to make sure the routines are easy to read and modify for these lessons. Even though this framework is not recommend for production use, you can take the lessons you learn here and apply them to whatever framework or programming language you and your team are using.

Coding APIs with NodeJS and DARRT

By relying on a consistent process for building APIs, we can get things done quickly and easily and focus on the work of accurately translating the design into code. And by using the same process more than once, we can get more skilled at the work and even learn how to modify the process to better fit your own (and your team's) needs in the future.

DARRT stands for *data, actions, resources, representations, and transitions*. The first three terms probably look familiar. We discussed the role of data properties and actions early in the design process, and I pointed out the importance of resources as we worked through our sketches and prototyping. I haven't discussed the last two terms (representations and transitions) yet, but they're also very important for implementing a successful API. I'll talk about those in detail as we work through the process.

Over the next several pages we'll do the following:

- Set up our NodeJS DARRT project
- Define the API's data properties using the `data.js` file
- Use the `action.js` file to hold the actions our API needs to handle
- Create the `resource.js` file to expose our HTTP resources and map them to actions
- Update the `representation.js` file to control how our resource responses are displayed
- Build up the `transition.js` file to define how API clients can complete desired actions

If some of this doesn't quite make sense yet, that's okay. Hopefully, as we work through each step in the process, it'll become clear and you'll be able to see the advantage of using the DARRT process for your APIs.

When we're done, we'll have a fully functional API that we can test, secure, and deploy in the next section of the book.

Starting a DARRT Project

In this first step, we'll initialize the DARRT library in our API project. The DARRT library holds several internal modules that handle the “behind-the-scenes” work along with a set of five key modules we'll use to define and implement the desired API.

First, we need to copy the [/darrt/](#) folder from the source code available for this chapter and place it in the root folder of the Onboarding API project we've been working on throughout the book.^[64] The DARRT folder (and its contents) looks like this:

ch08-building/darrt-folder-tree.txt

```
.  
├── darrt  
│   ├── actions.js  
│   ├── data.js  
│   └── lib  
│       ├── component.js  
│       ├── ejs-helpers.js  
│       ├── storage.js  
│       └── utils.js  
│   ├── representation.js  
│   └── representors  
│       ├── app-json.js  
│       ├── forms-json.js  
│       ├── links-json.js  
│       ├── prag-json.js  
│       └── text-csv.js  
└── resources.js  
    └── transitions.js
```

```
└── index.js  
    3 directories, 17 files
```

We'll edit only the top-level files in that collection. Everything in the **lib** and **representors** folders are just support routines.

After copying that folder into the root of your project, you also need to initialize the **index.js** file from the **starting-folder** and add that to the root of your project. This will be the file that pulls things together and launches the NodeJS project at runtime. That file looks like this:

ch08-building/starting-folder/index.js

```
*****  
// DARRT Framework  
// root of the service API  
// 2020-02-01 : mamund  
*****  
  
var express = require('express');  
var app = express();  
var cors = require('cors');  
var resources = require('./darrt/resources');  
var port = process.env.PORT || 8181;  
  
// support calls from JS in browser  
app.use(cors());  
app.options('*',cors());  
  
// point to exposed resources for this API  
app.use('/',resources);  
  
// start listening for requests  
app.listen(port, () => console.log(`listening on port ${port}!`));
```

Finally, we need to update the project with supporting packages using the **npm** tool. (See Chapter 3, [Modeling APIs](#), for more on npm.) Call up a command window, move to the root folder of your project, and run the following commands to update your project packages:

```
npm install -s body-parser  
npm install -s cors  
npm install -s ejs  
npm install -s express
```

These commands will update your [package.js](#) file with proper information and ensure the project will run as expected.

Now that we have the DARRT framework set up, it's time to modify the existing files to reflect our HTTP API design.

Data

The first letter in the DARRT framework acronym stands for *data*. The [data.js](#) file holds all the information about the properties (or “state”) the service needs to deal with. In many API implementations, this is handled by a database. But APIs don’t really understand data storage; they just understand passing properties back and forth. For our implementation, we’ll just focus on the state we need to pass back and forth for each request. We’ll rely on the DARRT framework to actually deal with the storage and retrieval of that state.

The [data.js](#) file includes four small sections:

- Properties
- Requireds
- Enums
- Defaults

Let’s look at properties first.

Properties

The properties collection is a list of state values this API needs to keep track of throughout the lifetime of the service. It’s the data we want to pass back and forth.

For our Onboarding service, we need to track all the state values needed to support the creation and completion of the work-in-progress (WIP) records we defined in the prototyping stage in Chapter 7, [Prototyping APIs](#).

The following is the list of properties our Onboarding API will be tracking:

```
// this service's message properties
exports.props = [
  'id', 'status', 'dateCreated', 'dateUpdated',
  'companyId', 'companyName', 'streetAddress', 'city', 'stateProvince',
  'postalCode', 'country', 'telephone', 'email',
  'accountId', 'division', 'spendingLimit', 'discountPercentage',
  'activityId', 'activityType', 'dateScheduled', 'notes'
];
```

I've grouped the state values into related collections. The first group contains values *this* API will be tracking (unique ID, status, and edit dates). The next three property groups relate to the **company**, **account**, and **activity** APIs.

Requireds

Along with the names of the state values to track, some of them need to be treated as *required* elements when writing an onboarding record. For this API, the only required values are the **id** and **status** properties:

```
// required properties
exports.reqd = ['id', 'status'];
```

Note that each of the related APIs (**company**, **account**, and **activity**) have their own list of required fields. We don't need to worry about them here. That will be handled when we finally resolve the WIP records in the **approve** step.

Enums

A few properties in our API have a limited set of valid values. For example, the `status` field should only be set to “pending,” “active,” “suspended,” and “closed.” For the Onboarding API, we have three enumerated properties: `status`, `division`, and `activityType`. Here’s the snippet of code to add to the `data.js` file that defines this API’s set of enumerations:

```
// enumerated properties
exports.enums = [
  {status:
    ['pending', 'active', 'suspended', 'closed']
  },
  {division:
    ['DryGoods', 'Hardware', 'Software', 'Grocery', 'Pharmacy', 'Military']
  },
  {activityType:
    ['email', 'inperson', 'phone', 'letter']
  }
];
```

Defaults

The last bit of setup we need for our `data.js` file is the list of default values. It’s a good idea to implement a set of default values for some of your properties. Doing this makes it easier to create a valid record in the system and offers developers using your API some guidance on typical values for your API properties.

The following is the list of properties and their defaults for the Onboarding API:

```
exports.defs = [
  {name: "spendingLimit", value: "10000"},
  {name: "discountPercentage", value: "10"},
  {name: "activityType", value: "email"},
  {name: "status", value: "pending"}
];
```

That’s the contents of the `data.js` file for our Onboarding API. With that work done (including saving the file to disk and updating the project repository

with [git](#)), it's time to move to the next element in our DARRT framework: the [actions.js](#) module.

Actions

The next letter in the DARRT framework acronym stands for *actions*. The [action.js](#) file in the DARRT framework contains all the actions we defined in our API design. These are the internal functions of our service, not the external endpoints of the API. We documented these actions for our [onboarding](#) in the initial WSD diagram we created in the previous chapter (see [Translating Your API Design into HTTP](#), and [the figure](#) as a reminder of what it looked like). Now we need to take the actions in that diagram and create them in NodeJS code within our project.

Each action is represented by a single function. For example, the following is the [listOnboardings](#) action from our diagram, rendered in NodeJS:

ch08-building/action-listWIP.js

```
module.exports.listWIP = function(req,res) {
  return new Promise(function(resolve,reject) {
    resolve(
      component({name: 'onboarding',action: 'list'})
    );
  });
}
```

As you can see, the function ([listWIP](#)) executes a call to the [component](#) module to get a [list](#) of [onboarding](#) records. All [action.js](#) functions return a NodeJS [Promise](#) object.

A more interesting example action is the one that allows API clients to write [company](#) data to the WIP record:

ch08-building/action-writeCompany.js

```
module.exports.writeCompany = function(req,res) {
```

```

var id,body;
return new Promise(function(resolve,reject){
    id = req.params.id||null;
    body = req.body||null;
    if(id==null && body==null) {
        resolve(component(
            {name: 'onboarding',
            action: 'update',
            id:id,
            item:body,
            props:data.props,
            reqd:data.reqd,
            enums:data.enums}));
    }
    else {
        reject({error: "missing id and/or body"});
    }
});
}

```

You can see from this code that the function `writeCompany` collects the `id` value from the URL (`id = req.params.id`) and the associated parameters in the message body (`body = req.body`) and, if both are valid, uses the DARRT `component` service to request an update to the WIP record using the declared property settings we created in the `data.js` file (`data.props`, `data.reqd`, and `data.enums`).

One more action worth viewing is the one for the `readStatus` operation:

ch08-building/action-readStatus.js

```

module.exports.readStatus = function(req,res) {
    return new Promise(function(resolve,reject){
        if(req.params.id && req.params.id!=null) {
            var id = req.params.id;
            var fields="id, status, dateCreated, dateUpdated"
            resolve(
                component(
                    {name: 'onboarding',action: 'item',id:id, fields:fields}
                )
            )
        }
    })
}

```

```
    );
}
else {
    reject({error: "missing id"});
}
});
}
```

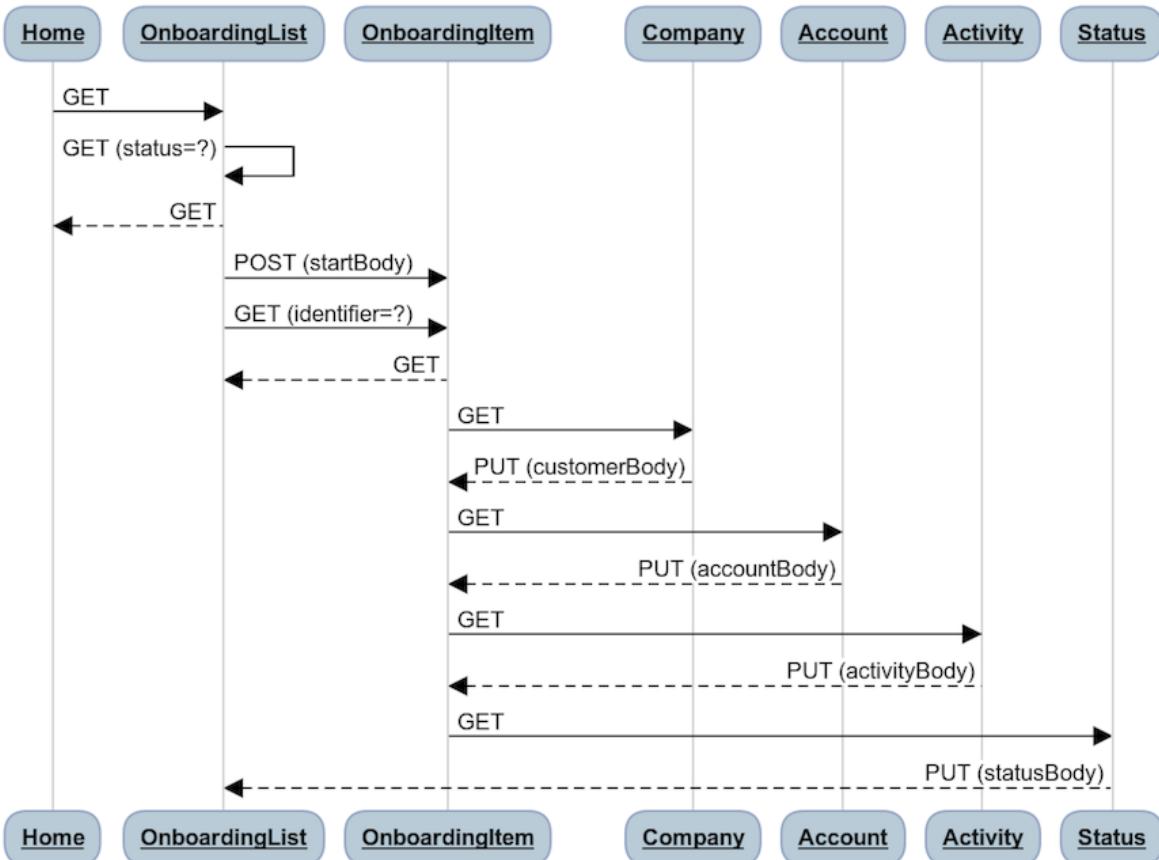
Notice that in this function, a new argument called **Fields** is used to tell the **component** library which properties to return in the response. This makes it possible to customize the contents of HTTP responses for your API clients.

Check out the project file source code to see the additional functions in the **actions.js** file.^[65]

With the **data.js** and **actions.js** files, we have created the *internal* elements of our API service. Now we can focus on the *external* elements: the exposed resources, response formats, and the request transitions.

Resources

Now it's time to implement all the externally addressable API endpoints for this service. We do that in the **resources.js** file of the DARRT framework. The reference diagram for our external endpoints is the one we created in Chapter 7, [Prototyping APIs](#). To refresh your memory, it looks like this:



The key elements on this diagram (the circled items) are the external endpoints of our HTTP API. This diagram also tells us which HTTP methods are supported for each resource. That's what we need to implement in our `resources.js` file—not just the endpoint but also the methods to be supported.

The following is a quick rendering of the `resource.js` file with all the endpoints and methods “stubbed in.” There’s no code in any of these yet—we’ll add it shortly. For now, scan the list of operations and note how they match up to the diagram:

ch08-building/resource-list.js

```

// *****
// public resources for the onboarding service
// *****

```

```
router.get('/',function(req,res){ });
router.post('/wip/', function(req,res){ });
router.get('/wip/',function(req,res){ });
router.get('/wip/filter/', function(req,res){ });
router.get('/wip/:id', function(req,res){ });
router.get('/wip/:id/company', function(req,res){ });
router.put('/wip/:id/company', function(req,res){ });
router.get('/wip/:id/account', function(req,res){ });
router.put('/wip/:id/account', function(req,res){ });
router.get('/wip/:id/activity', function(req,res){ });
router.put('/wip/:id/activity', function(req,res){ });
router.get('/wip/:id/status', function(req,res){ });
router.put('/wip/:id/status', function(req,res){ });
```

The `resources.js` module replies on a popular NodeJS HTTP library called Express.js.^[66] We initialized that library earlier in our `index.js` file with these lines:

```
var express = require('express');
var app = express();
var resources = require('./darrt/resources');
...
app.use('/',resources);
```

The Express library allows us to define functions that respond to a combination of the HTTP method (`router.get`) and the HTTP URL (`wip/:id/status`) that client apps use when making API requests. The list you see in the code represents all the resources and all the methods we defined in our diagram in the previous chapter.

Each of these functions gets called when the associated URL and HTTP method are used by a client app. The role of each function is to (1) accept the request parameters (contained in the `req` input argument), (2) process those parameters, and (3) formulate an HTTP response and return it to the caller (using the `res` input argument).

One of the functions in the `resources.js` file is shown here:

ch08-building/filter-resource.js

```
// filter the list of onboarding records
router.get('/wip/filter/', function(req,res){
  var args = {};
  args.request = req;
  args.response = res;
  args.action = actions.filterWIP;
  args.type = "onboarding";
  args.config = {
    metadata:metadata,
    templates:templates,
    forms:forms,
    filter:"list"
  }
  respond(args);
});
```

This function calls just one internal DARRT method (`respond(args)`). That method takes several values that are worth reviewing here. The first two (`args.request` and `args.response`) represent the HTTP request and response objects mentioned earlier. The next argument (`args.action = actions.filterWIP`) is the internal action function to be executed for this endpoint. That's the function used to formulate a response to this request. The following string argument (`args.type = "onboarding"`) is used to indicate the type of storage objects that are being dealt with.

The last parameter is a JavaScript object (`args.config = {...}`) that refers to the contents that will be used to create the intended HTTP response. This object includes the `metadata` object, which holds some shared data properties; the `templates` object, which holds definitions for the responses; the `forms` collection, which holds a set of links and forms to be included in any responses; and finally a `filter` object, which helps select the proper forms and templates.

If these last few items are not quite clear, don't worry. We'll get a chance to dig deeper into the contents of this last structured object.

Each of the functions in the `resources.js` file look pretty much the same. You can check out the complete `resources.js` file in the source code associated with this chapter for a complete example.

The three DARRT elements described in the previous sections—data, actions, and resources—represent the complete “loop” of requesting a resource and executing an associated action. The next two DARRT elements (representations and transitions) pertain to the details of how responses will be formulated—how to populate the response and which format to use when returning that response. We’ll deal with those two elements next.

Representation

The second *R* in DARRT stands for *representations*. This refers to the format of the HTTP API responses. HTTP APIs can use a wide variety of formats to represent HTTP responses. The most common response for APIs today is plain JSON (`application/json`), but a number of other formats exist, including those shown in Chapter 1, [Getting Started with API First](#).

In the DARRT framework, you can vary the response representations by simply loading one of the preset formats and referencing them when implementing the API. For this book, I’ve collected a small set of formats. Each of them were implemented using the EJS templating engine for NodeJS.^[67] You can review these templates in the `darrt/representors/` folder of the book’s source code.^[68] If you wish, you can design additional representors and add them to any existing DARRT API.

The following is the EJS template for plain JSON responses:

ch08-building/app-json-template.js

```
// plain JSON representor template
exports.template =
{
  format: "application/json",
  view:
```

```

` 
{
  "<%=type%>":
  [
    <%var x=0;%>
    <%rtn.forEach(function(item){%
      <%if(x!==0){%,<%}%>
      {
        <%var y=0;%>
        <%for(var p in item){%
          <%if(y!==0){%,<%}%>
          "<%=p%>": "<%=helpers.stateValue(item[p],item,request,item[p])%>"
          <%y=1;%>
        <%}%>
      }
      <%x=1;%>
    <%});%>
  ]
}
` 
}

```

The version of DARRT used for the project in this book includes EJS templates to support the following response formats:

- [text/csv](#)
- [application/json](#)
- [application/links+json](#)
- [application/forms+json](#)
- [application/prag+json](#)

You can inspect the EJS templates in the code folder for this chapter.

For the project in this book, you won't need to modify the [representation.js](#) DARRT file. For completeness, I've included one in the project:

ch08-building/load-representors.js

```

// load representors
var appJson = require('./representors/app-json');
var formsJson = require('./representors/forms-json');

```

```

var linksJson = require('./representors/links-json');
var pragJson = require('./representors/prag-json');
var textCsv = require('./representors/text-csv');

// return supported response bodies
exports.getTemplates = function() {
  var list = [];

  list.push(appJson.template);
  list.push(formsJson.template);
  list.push(linksJson.template);
  list.push(pragJson.template);
  list.push(textCsv.template);

  return list;
}

// return supported response identifiers
exports.getResponseTypes = function() {
  var rtn = [];
  var viewList = this.getTemplates();

  viewList.forEach(function(item) {
    rtn.push(item.format);
  });

  return rtn;
}

```

You'll see where the EJS templates are loaded and then collected into an array for templates and an array for format names. Both of these arrays are used by the DARRT framework at runtime.

The [representation.js](#) file controls which formats are available for API responses. The last file in our implementation work—[transitions.js](#)—defines which operations are available for each resource representation.

Transitions

The last item in our DARRT framework setup is the [transitions.js](#) file. This file holds a set of declarations for forms and links related to the API. An

example of a link transition in HTML is the anchor tag:

```
<a href="/list/" rel="collection list company">Company List</a>
```

An example of a form transition in HTML is the form tag:

```
<form action="/filter/" method="get">
  <input name="search" value="" />
  <input type="submit" />
</form>
```

The HTML format has lots of transition elements—elements that describe a way to “transition” from one view to the next. Several API formats support link and form transitions too. We saw some of them early in the book in Chapter 1, [Getting Started with API First](#). And, in Chapter 2, [Understanding HTTP, REST, and APIs](#), we discussed the role HTML plays in the common practice of the web. Since one of our goals in creating APIs for this book is to learn from and emulate features of the web, the DARRT framework was designed to allow developers to declare and use transitions in API responses.

Just as in HTML, adding transitions to API responses informs the API caller of the next possible actions in the API workflow. And, like HTML, well-designed APIs don’t have just one possible, fixed sequence of actions. Instead, users (humans for HTML, machines for APIs) can make their own choices on what steps to take next.

The following is what a transition looks like in the `transitions.js` file of the DARRT framework (it’s similar to the HTML anchor tag we saw earlier):

ch08-building/list-transition.js

```
{
  id: "list",
  name: "list",
  href: "{fullhost}/wip/",
  rel: "collection onboarding",
  tags: "collection onboarding home list item",
  title: "List",
```

```
    method: "GET",
    properties: []
}
```

Note: The `{fullhost}` macro in the code snippet gets filled in at runtime with the full host name (<http://onboarding.example.org/wip>). DARRT supports a handful of these types of macros.

A more interesting transition is the one that describes the details of adding `account` information to a work-in-progress (WIP) record for the Onboarding API:

ch08-building/add-account-transition.js

```
{
  id: "addAccount_{id}",
  name: "addAccount",
  href: "{fullhost}/wip/{id}/account",
  rel: "item edit-form onboarding",
  tags: "onboarding list item",
  title: "Add Account",
  method: "PUT",
  properties: [
    {name: "accountId", value: "{accountId}"},
    {name: "division", value: "{division}"},
    {name: "spendingLimit", value: "{spendingLimit}"},
    {name: "discountPercentage", value: "{discountPercentage}"}
  ]
}
```

The `transition.js` file for the Onboarding API project contains about a dozen transitions. You can view them in detail by checking out the source code associated with this chapter.

Don't Skip the Transitions

Don't Skip the Transitions



Many API developers shortcut their API work by skipping the task of defining transitions for their responses. This happens, in part, because most API libraries make including transitions difficult. If you're using plain JSON or CSV files as your response representation, it's easy to produce an API without any forms or links. But that API is harder to use and more complicated to maintain.

By adopting transition-friendly formats like HAL, Siren, Collection+JSON, and others shown throughout this book, you'll be producing APIs that are easier to use, harder to be misused, and generally more stable over time. Even if your team doesn't currently use tooling that makes supporting transitions easy, you can help things along by taking lessons learned from the DARRT framework and incorporating them into the tooling you and your team use today.

Putting It All Together

Now that we've reviewed all the individual parts of the DARRT framework (data, actions, resources, representations, and transitions) and seen how it's put together, we can run some simple queries against our instance of the [onboardingAPI](#) to see how it behaves. To do that, we need to start up our local instance of the [onboardingAPI](#) and then use curl to execute some simple tests.

First, to start the local instance of the [onboardingAPI](#), open a command window, move to the [/building/all-together/](#) folder in the source code download associated with this chapter, and run the following command:

```
npm run dev
```

This command will fire up the [onboardingAPI](#) service and wait for requests. Note that this command assumes you already have a NodeJS utility called nodemon installed on your machine. If you don't have this handy tool installed, you can execute the following command: [npm install -g nodemon](#).^[69]

The results of this one command should look something like this:

ch08-building/test-nodemon.txt

```
> onboarding@1.0.0 dev /building/all-together/onboarding
> nodemon index

[nodemon] 2.0.2
[nodemon] to restart at any time, enter `rs`
[nodemon] watching dir(s): ***!
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index index.js`
listening on port 8080!
```

To run some queries against our [onboardingAPI](#), you need to open another command window. This is where we'll use curl to send some requests to the API to inspect its responses.

To start, let's check out the root of the API service and see what's returned. Type `curl localhost:8080` at the command line and view the response:

ch08-building/test-onboarding-01.txt

```
mca@mamund-ws:~/onboarding$ curl localhost:8080
{
  "home" :
  [
    {
      "id" : "list",
      "name" : "onboarding",
      "rel" : "collection onboardings",
      "href" : "http://localhost:8080/wip/"
    }
  ]
}
mca@mamund-ws:~/onboarding$
```

This is the root resource, which is pointing us to the [list](#) resource. Note the format is returned in plain JSON, the default for this API. We can alter the response format by passing the HTTP [accept](#) header like this:

ch08-building/test-onboarding-02.txt

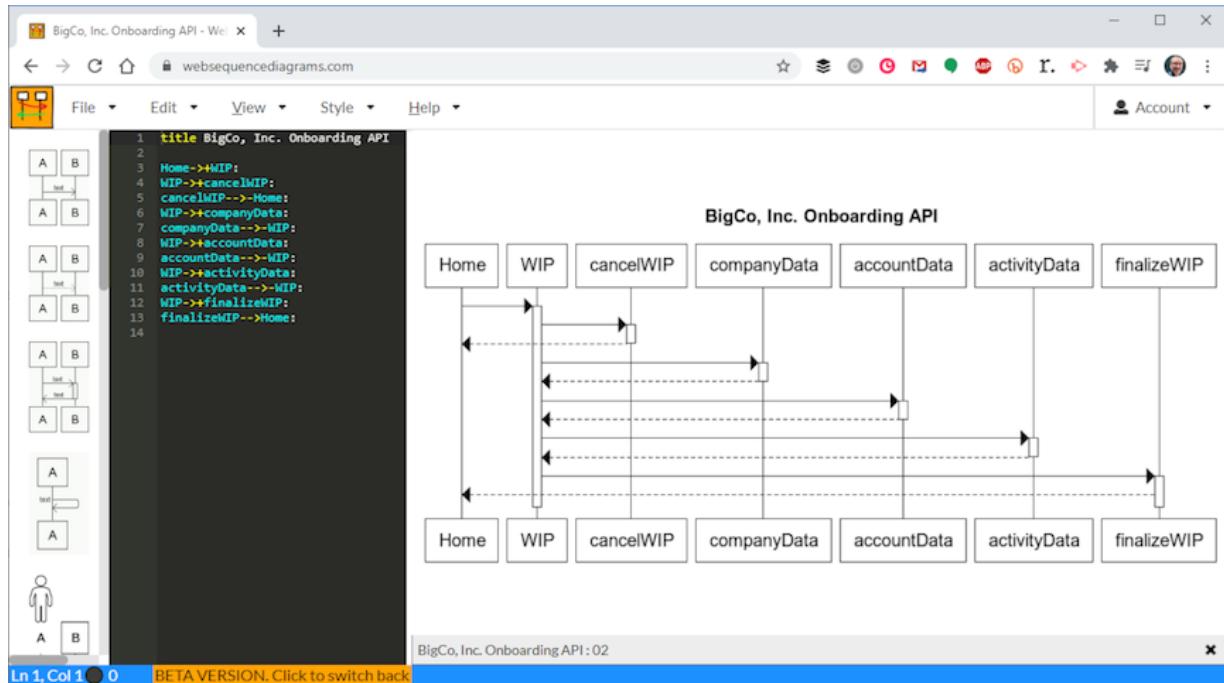
```
mca@mamund-ws:~/onboarding$ curl localhost:8080 \
-H "accept:application/forms+json"
{
  "home" :
  {
    "metadata" : [
      {"name" : "title","value" : "BigCo Onboarding Records"},
      {"name" : "author","value" : "Mike Amundsen"},
      {"name" : "release","value" : "1.0.0"}
    ],
    "links" : [
      {"id" : "self","name" : "self","href" : "http://localhost:8080/",
       "rel" : "self collection onboarding",
       "tags" : "collection onboarding self home list item",
       "title" : "Self","method" : "GET","properties" : []
    ],
    "links" : [
      {"id" : "self","name" : "self","href" : "http://localhost:8080/",
       "rel" : "self collection onboarding",
       "tags" : "collection onboarding self home list item",
       "title" : "Self","method" : "GET","properties" : []
    ]
  }
}
```

```
{"id" : "home","name" : "home","href" : "http://localhost:8080/",  
  "rel" : "collection onboarding",  
  "tags" : "collection onboarding home list item",  
  "title" : "Home","method" : "GET","properties" : []  
},  
{"id" : "list","name" : "list","href" : "http://localhost:8080/wip/",  
  "rel" : "collection onboarding",  
  "tags" : "collection onboarding home list item",  
  "title" : "List","method" : "GET","properties" : []  
}  
],  
"items" : []  
}  
}  
mca@mamund-ws:~/onboarding$
```

The [application/forms+json](#) representation contains additional information, including not just [href](#) values, but methods, descriptions, and when appropriate, request parameters (using the [properties\[\]](#) collection). I like programming against these types of responses since I get lots of important information without having to read through extensive documentation. The good news is, when using a framework that supports varying representations and transitions (like DARRT does), developers can decide for themselves which response format they prefer to work with.

Now that we've seen the basics of how the [onboardingAPI](#) behaves, we can put together a set of simple tests to exercise the API and explore some options.

The original workflow we put together in the design phase of our API work (see [Mapping Resources](#)) looked something like this:



And that ended up as the HTTP implementation workflow we saw earlier in this chapter in [Resources](#).

That last implementation diagram is what we used to code our `onboardingAPI` here in this chapter. So it follows that we could do a simple test of our interface by running a set of HTTP requests using the curl tool we looked at in [Exploring APIs with curl](#).

I usually put together a short command-line script that tests what I call the “happy path” of the API. If all goes well and the API is used exactly as intended, I should be happy with the results. Here’s a peek at part of the script I put together to test my `onboardingAPI` “happy path”:

ch08-building/happy-script-01.txt

```

# ****
# WIP list read
echo Read $svr/wip/
curl $svr/wip/ \
    -H "$acc" | jq "."
echo

```

```
# ****
# WIP add
echo Add $svr/wip/
curl $svr/wip/ -X POST -d "id=$wid&status=pending" -H $acc | jq "."
echo
```

This snippet shows the command to pull the list of WIP records and the command that creates a new WIP onboarding record. Notice that I'm using two utilities in this script: the curl utility we talked about earlier in the book and the `jq` utility. The `jq` utility is just a nice way to format JSON output and isn't required. If you don't have `jq` installed, you can modify the script included with this book.

Are We Testing Now?



This “happy path” script is a super simple way to validate the API. But it doesn’t really qualify as testing the API. We’ll go over a much more extensive approach to API testing in the next chapter.

Another thing you might notice in the script snippet is my use of variables (`$acc` and `$wid`). I have a full set of script variables for the server connection, as well as all the test data to be written to the API. Here’s what that looks like in the script:

ch08-building/happy-script-02.txt

```
# ****
# network vars
svr="http://localhost:8080"
acc="accept:application/json"

# ****
# WIP data
wid="q1w2e3r4"
status="pending"

# ****
```

```

# company data
cid="w2e3r4t5"
name="Ems Corp"
email="emscorp@example.org"
street="123 Main"
city="Emtown"
state="MN"
pcode="12345"
country="USA"
tel="123-456-7890"

# ****
# account data
aid="e3r4t5y6"
div="Military"
limit="10000"
pct="10"

# ****
# activity data
tid="r4t5y6u7"
type="email"
sched="2020-04-01"
notes=""

# ****
# status
approve="active"
reject="closed"

```

I won't spend more time on this script here. You can check it out in the source code folder for this book ([code/ch08-building/happy-path/happy-path.sh](#)) as well as some sample output ([code/ch08-building/happy-path/2020-02-03-output.txt](#)) from a run of the script.

What's Next?

In this chapter, we focused on translating the final API prototype into an actual working implementation of the Onboarding API. To do that, we used a simple API library called DARRT that was built with NodeJS, the Express web framework, and the EJS template library.

The DARRT library leads developers to translate designs into five parts: data, actions, resources, representations, and transitions. The first three parts are common in all web API approaches; the last two parts are added to improve the quality of the implementation and especially to support the use of links and forms. And links and forms make up the distinguishing elements of APIs built using the principles of web development we looked at in Chapter 1, [Getting Started with API First](#).

This chapter wraps up the build phase of the book (sketch, prototype, and build). Starting with the next chapter, we move into the release phase. There we'll cover testing, securing, and deploying your working API. We'll also take a look at *modifying* your API after it's been released into production.

Chapter Exercise

For this chapter's exercise, you finally get to code your API. That means you'll be using the design assets from previous chapters along with NodeJS and the DARRT library to implement a working version of the [credit-check](#) API.

Hopefully, the task list for implementing your API will look familiar.

First, copy the DARRT library into your NodeJS project. Next, use information from your ALPS document and WSD diagram to update the DARRT library files. And finally, fire up the completed [credit-check](#) service and run your validator script to ensure the service at least supports your intended “happy path” workflow. It might seem like quite a bit to accomplish, but if all goes well you should be able to complete this exercise in less than thirty minutes.

Install the DARRT Library

For this exercise, the [/before/](#) folder has the DARRT library already installed. I did this so you could have some files already configured instead of having to start from scratch. However, you'll need to use npm to update dependencies in the project and then run the API service in “monitor” mode. We covered that earlier in the chapter.

Using DARRT in the Future



For future projects, you'll find a fresh copy of the DARRT library in the [code/ch08-building/starting-folder](#) section of the source code package associated with the book. You can copy that (along with the [index.js](#) file in that same folder) to the root of your NodeJS API projects whenever you want to get a quick start on your API implementation.

Implement Your API

You'll need to use design documents from your project's `assets` folder as guides as you implement your API. I usually rely on my resource diagram (`credit-check-http.png`) and my ALPS profile (`credit-check-alps.xml` or `credit-check-alps.json`) to help me fill in the `data.js`, `actions.js`, `resources.js`, and `transitions.js` files.

For this exercise, you won't need to update the `representations.js` since it's already been set up for all the output formats needed for this project. You'll also notice that the `data.js` file has been set up with the common basic data properties (`id`, `status`, and `dateCreated`, `dateUpdated`). You'll only need to add the fields defined by your `credit-check` design. The `actions.js`, `resources.js`, and `transitions.js` files will need to be updated to match your design specs.

Validate Your API's Workflow

Finally, after updating your DARRT files, your API should be ready for validation. For this exercise, you can run the `validate.sh` script in the `tests` folder of the project on disk.

See Appendix 2 ([Solution for Chapter 8: Building APIs](#)) for the solution.

Footnotes

[64] https://pragprog.com/titles/maapis/source_code

[65] https://pragprog.com/titles/maapis/source_code

[66] <https://expressjs.com>

[67] <https://ejs.co>

[68] https://pragprog.com/titles/maapis/source_code

[69] <https://nodemon.io>

Copyright © 2020, The Pragmatic Bookshelf.

Part 4

The Release Phase

Chapter 9

Testing APIs

So far, we've focused on the sketch-prototype-build pattern to move from ideas to full implementation of your API. In this part of the book, we'll focus on other important elements of designing and building great APIs, including the role of testing, securing, and deploying APIs. In reality, you'll be dealing with these issues throughout the design and build process. But bringing up all these things at the beginning of the process makes for a confusing book! Instead, I opted to save these details for a bit later so that we could spend time working through them individually.

In this chapter, we'll explore the world of API testing. We'll start by taking a look at the goal of API testing and some basic principles behind that goal, including behavior-driven and happy path/sad path testing concepts.



We'll also look at a simple form of testing (validation, really) that I call *simple request tests*, or SRTs. These are just calls to a URL to confirm that the URL is working and that the input and output information seems to be what was expected. This isn't really API testing in depth, but SRTs come in handy for quick validation of URLs as you're writing your code.

After exploring SRTs, we'll move to full-on API testing using the Postman platform. Postman combines a locally installed client with a Software-as-a-Service (SaaS) back end to create a powerful testing platform. You'll learn how to apply our testing principles using the Postman platform to write test collections, save them to the Postman servers, and run them using the Postman client. You'll also learn how to export Postman test collections and run them locally via a command-line tool called `newman` in order to support scripting build pipelines.

Along the way we'll look at a testing pattern I often use called *protocol, structure, and values*. This is a pattern you can use no matter what style of API you're testing or what framework or tools you're using to author and execute the tests. It's also an approach that works well over time as you scale up your tests, fix bugs, and add new features to your existing API.

That's a lot to cover in one chapter, so let's get started with some basic goals and principles of API testing.

The Goals of API Testing

First and foremost, the goal of API testing is to do just that—test the interface. That means you need to test each URL (or “endpoint”) in the API, including any possible input parameters. You also need to confirm the responses to those endpoint requests. For starters, you need to make sure each promised endpoint exists and accepts the inputs and produces the outputs expected. But that’s just a start.

Test the API’s Behavior

In addition to testing the interface, you also need to make sure the API *behaves* the way you’d expect from reading the Application-Level Profile Semantics (ALPS) description and the OpenAPI Specification (OAS) documents. For example, an API might have a rule that each request for a credit rating will return a value between 1 and 10. If the API returns a value of 0 or 13, that’s unexpected behavior.

Another API might have a rule that you can’t create two records with the same `companyName` value. That means any attempt to break that rule results in an API behavior of refusing to write the duplicate record and returning an HTTP `400 Bad Request` response.

Finally, you might have an API that supports an HTTP request to approve a purchase order, as shown in the `curl` example that follows:

```
curl http://api.example.org/purchases/ -X PUT -d status=approved
```

However, the API should not allow anyone to approve a purchase order when that order record is missing a valid value for the `salesPerson` property. We expect any attempt to approve a purchase order before all the required information is supplied to result in an HTTP `400 Bad Request`.

Testing for expected (or unexpected) behavior is critical to proper API testing. And there's an entire practice and set of tools around this approach, known as behavior-driven development, or BDD.

The practice of behavior-driven development was created by Dan North in 2006.^[70] North's frustration with the way test-driven development (TDD) was being designed and used led him to come up with a different point of view regarding tests in general. He wanted to make the whole process of thinking about and implementing testing accessible to an audience that goes beyond programmers.

TDD can be described as a code-centric practice of improving the quality of the *code* you're writing, while BDD is the practice of improving the behavior of the *system* you're building. As North tells it, "Behavior-driven development is an 'outside-in' methodology."^[71] And even though North developed his BDD system to help write better code, the BDD model works very well when creating tests for APIs. One of the big reasons for that is North's "outside-in" focus on testing. When we're working with APIs, the only target we have for testing is the "outside"—the interface itself.

When focusing on testing the interface, looking for expected behavior is an excellent way to build a quality testing strategy. That leads to another important principle I rely on when developing my tests: the difference between "happy path" and "sad path" tests.

Test Both Happy and Sad Paths

A common approach to testing is to confirm that the interface is doing what it's designed to do. For example, I should be able to get a list of customers, filter that list using selected properties, add new customers, edit them, and maybe even remove them from storage. Logically, I should be able to turn these expectations into simple tests and execute those tests to "prove" that my expectations are met.

For example, here's a set of requests that will test some of these expectations:

```
http://localhost:8181/list/  
http://localhost:8181/filter?status=active  
http://localhost:8181/ -X POST -d \  
  id=q1w2e3r4&status=pending&email=test@example.org  
http://localhost:8181/status/q1w2e3r4 -X PATCH -d status=active  
http://localhost:8181/q1w2e3r4 -X DELETE
```

Each of these requests represents a test of one of my expectations. Essentially, I expect all of these requests to succeed. In HTTP API lingo, I expect the server to return a **200 OK** response along with the proper text body. These are examples of “happy path” tests. When they work, everyone is happy.

However, I have another important set of expectations about the customer API. I expect, for example, that the API won't let me add a new customer record unless I supply all the required fields. I expect the API to prevent me from adding duplicate customer records. I expect it to validate the data qualities of each field to make sure I don't save a string to a numeric field, or attempt to write an invalid string to an email field, and so forth. These are “sad path” tests. Using HTTP API–speak, they should return **400 Bad Request** (or some other 4xx HTTP status code). It's important to validate that your API *prevents* undesirable things from happening too.

For this reason, I always try to come up both “happy path” and “sad path” tests for my APIs. And that means I need to collect the tests in a set and then execute them in a consistent way.

Testing with SRTs

An easy way to start testing your API is to focus just on the interface itself with what I call *simple request tests*, or SRTs. SRTs are just what the name sounds like: a simple HTTP request you can run to validate that the minimal elements of the API are working as you would expect. And that's how I use SRTs—to validate the API.

Here are a few examples of SRTs used for testing an API that supports managing people's contact information:

ch09-testing/srt-requests.txt

```
# person api test requests
# 2020-02 mamund

http://localhost:8181/
http://localhost:8181/list/
http://localhost:8181/filter?status=active
http://localhost:8181/ -X POST -d \
    id=q1w2e3r4&status=pending&email=test@example.org

http://localhost:8181/q1w2e3r4 -X PUT -d \
    givenName=Mike&familyName=Mork&telephone=123-456-7890
http://localhost:8181/status/q1w2e3r4 -X PATCH -d status=active
http://localhost:8181/q1w2e3r4 -X DELETE

# EOF
```

I also used some SRTs very early in the book in [Exploring APIs with curl](#).

Because HTTP APIs rely on URLs, HTTP methods, and a few HTTP headers, tools like curl give you all the power you need to create a collection of SRTs that you can run against your API. You can do this while you're implementing the API as a kind of quick validation of your progress as you write your API code. You can also use SRTs as a simple validator once

you've released your APIs into production to make sure the deployed API works as expected.

To make working with SRTs easy, I created a small command-line script (written in Bash) that does a couple of things:

1. It makes sure a local instance of the API is up and running.
2. It “plays” the collection of requests against the API and captures the results.

Here's the scripting portion of the [srt.sh](#) Bash script, which is also included in the source code files for this book:

ch09-testing/srt-scripting.txt

```
#####
# start target service
echo
echo start API service...
npm run dev &

#####
# allow service to spin up
echo
echo sleeping...
sleep 5

#####
# run requests
echo
echo start request run...
while IFS= read -r line
do
  if [ ! -z "$line" ] && [ ${line:0:1} != "#" ]
  then
    echo
    echo "$line"

    if [ -z "$outfile" ]
    then
      curl $line
```

```
else
    echo "$line" >> $outfile
    curl --silent --show-error --fail $line >> $outfile
fi
fi
done < $infile
```

Running the complete script makes requests to the API and echoes the results to the screen. You can also store the results of the request run by piping the script output to a file ([./srt.sh > srt-output.txt](#)) so that you can inspect them more carefully and/or use them as aids for debugging your implementation code.

It's a good idea to store this script within the [/tests/](#) folder of your API project. Each time you make a change to the API project code, run that script to make sure everything still works as expected.

SRTs are a good first step in validating your interface implementation, but this doesn't qualify as real testing since we're not yet focused on that all-important expected behavior. To do that, you need more powerful tools and a more in-depth approach. For that, I like to use a tool called Postman, which we'll look at next.

Your Own SRTs



You'll find a complete copy of the [srt.sh](#) script in the source code associated with this chapter at <https://pragprog.com/book/maapis>. You can use that file as a guide for creating your own simple request testing tool.

Using Postman for API Testing

Postman is a platform for working with web APIs.^[72] Originally focused just on testing, the Postman platform now offers support for generating documentation, spinning up API mock servers, and even designing APIs. For this book, I'm focusing on the basic testing features, but I encourage you to explore Postman because it may be able to solve some other challenges we've already discussed in this book around designing, prototyping, and documenting APIs.

You can start using Postman for free. However, you'll need a valid Postman user account in order to create, store, and manage your test collections. Refer to Appendix 1 ([Postman](#)) for details on how to create an account and install the Postman client on your local workstation.

Once you have created your Postman account and installed the Postman client application locally, you can fire it up and get started creating and running your API tests. When you first open Postman, you see the main screen, as shown in the following screenshot. You can start creating collections and tests right away. But before you do, you should be sure to sign in (see the icon bar in the upper-right corner) when you start the Postman client for the first time.

A screenshot of the Postman application. On the left, there's a sidebar with a 'Collections' tab selected, showing three collections: 'BigCo Company API' (15 requests), 'happy_path' (5 requests), and 'sad_path' (5 requests). The main area shows a 'Company Home' collection with a single GET request to 'Company Home'. The 'Tests' tab is active, displaying 8 test results, all of which have passed. The results include: Status is 200, Header content-type contains application/forms+json, Meta property title contains BigCo Company Records, Meta property release contains 1.0.0, and Meta property author contains Amundsen.

This screenshot shows one of my recent Postman sessions. In my client, you can see the list of tests (called *collections* in Postman) on the left side of the screen, and the results of the last test in the middle of the screen (note the “Pass” boxes). The first time you load the client app you won’t see any collections, but we’ll build some in this chapter.

Sign In to Save Your Data



When you sign in using your Postman account, all your test scripts and settings are saved to the server. If you don’t sign in, your data is only saved locally and will be permanently deleted when you sign out.

In the center of the screen, just below the **GET** line, a dot appears next to the Tests tab. This is where the actual testing code appears in the Postman interface. For this particular test of the BigCo API’s home page, the testing code looks like this:

```
ch09-testing/company-home-tests.js
```

```
/*****
```

```

* FORMS+JSON TESTS
***** */

// shared vars for this script
var body = pm.response.json();
var utils = eval(globals.loadUtils);
utils.setObject({object: 'home'});

// 200 OK
utils.checkStatus(200);

// HEADERS
utils.checkHeader({name: 'content-type', value: 'application/forms+json'});

// METADATA
utils.checkMetaProperty({name: 'title', value: 'BigCo Company Records'});
utils.checkMetaProperty({name: 'release', value: '1.0.0'});
utils.checkMetaProperty({name: 'author', value: 'Amundsen'});

// LINKS
utils.checkPageLink({name: 'home', has:[ 'id', 'href', 'rel']});
utils.checkPageLink({name: 'self', has:[ 'id', 'href', 'rel']});
utils.checkPageLink({name: 'list', has:[ 'id', 'href', 'rel']});

// EOF

```

I'll cover what all this code means later in this chapter. (If you can't wait, check out [Programming Your Postman Tests](#).) For now, it's important to know that the real power of Postman is the ability to not just make requests (like we did with SRTs) but also to carefully inspect the response to make sure the API is behaving as expected. And we do that by writing our own testing code. That code is written and stored *inside* our Postman collections. Then, when we're ready, we can use the Postman client application to retrieve our test collection and execute that set of tests against our running API.

Lucky for us, Postman has a built-in set of programming tools just for this purpose.

Using ChaiJS Assertions

The Postman platform has a handful of powerful programming tools built right into the client application. This includes a set of what are called *assertion libraries* to make it easy to write tests. The library in Postman that I use most often is called ChaiJS.^[73] I like the ChaiJS library because it's designed to look and "read" to be similar to the way people speak. This kind of approach is called a *fluent interface* and dates back to 2005.^[74]

Here's an example ChaiJS test written in the Postman client:

```
// 200 OK
pm.test('Status is 200', function() {
    pm.expect(pm.response.code).to.equal(200);
});

// application/json
pm.test('Content-Type header is application/json', function() {
    var hdr = pm.response.headers.get('content-type');
    pm.expect(hdr).to.include('application/json');
});
```

This code snippet contains two tests. The first tests the HTTP status code returned by the API server; the second tests the value of the `content-type` header in the HTTP response. Each test has the same basic structure:

- They declare a test (`pm.test(...)`).
- They describe the test (`...'Status is 200'...`).
- They write a function that implements the test (`...function() {...})`).

The contents of the inner function are what actually implement the test. For example, here's the test for checking the HTTP status code in a response:

```
pm.expect(pm.response.code).to.equal(200);
```

And here's the test for checking the `content-type` header:

```
var hdr = pm.response.headers.get('content-type');
pm.expect(hdr).to.include('application/json');
```

In looking at these code snippets, you can see the fluent style of programming at work. The code in each test reads rather close to speaking English (“I expect the response code to equal 200” and “I expect the header to include the string `application/json`”). This way of speaking/thinking/coding also follows rather closely with Dan North’s BDD philosophy of creating tests that can be understood by more than expert programmers.

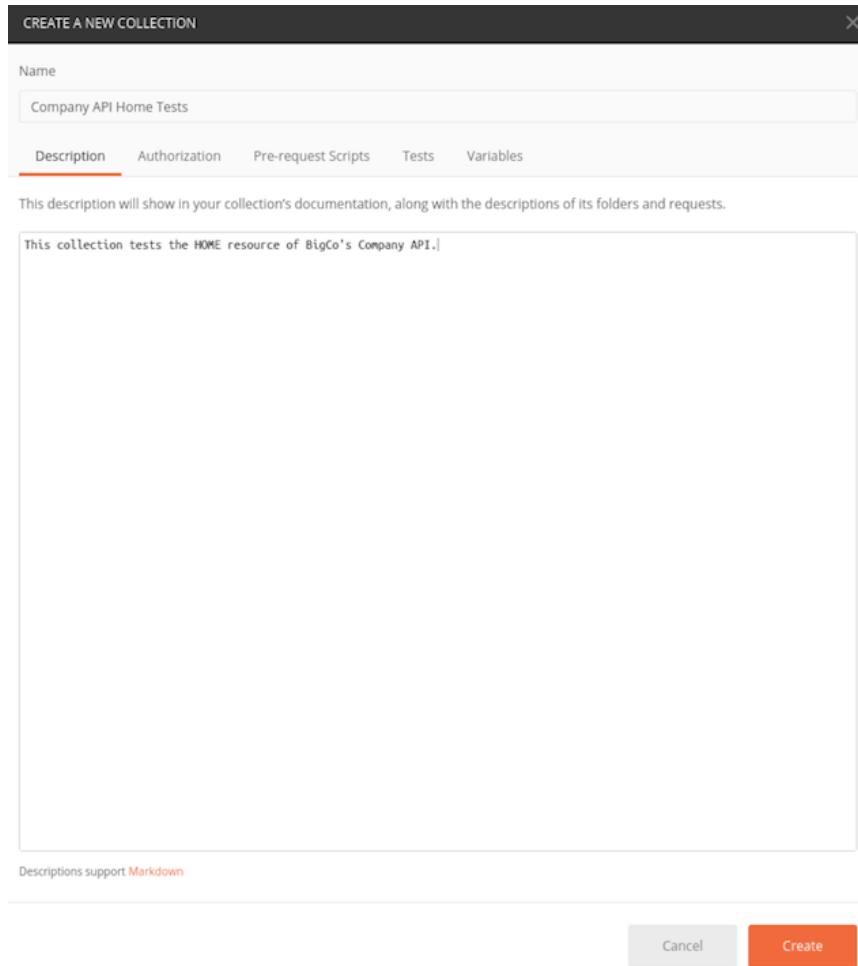
Now that you’ve gotten a taste of what BDD-style testing with ChaiJS looks like, let’s work through a detailed example on an existing API.

Programming Your Postman Tests

In this section, we’ll create a new test collection in Postman, write a series of “happy path” tests, and then execute them to inspect our results. To do this, you need to fire up your locally installed Postman client and log in.

Create a Collection

The first step in testing with Postman is to create a new test collection. Click the “+ New Collection” link that appears near the top of the left sidebar of the Postman interface (as shown earlier in [the screenshot](#)). This calls up the Create a New Collection dialog box, as shown in the following screenshot, where you can enter a name for the collection and a description. For now, enter “Company API Home Tests” for the name and enter a short description, such as “This collection tests the HOME resource of BigCo Inc.’s API.” You can skip the other tabs for now and just press the Create button in the lower-right corner of the dialog box.



Create a New Request

Once the new collection has been created, you'll see the collection appear in the left sidebar, along with a highlighted link to "Add request." Click that link; and in the dialog box that appears, enter "Company Home" as the Request Name and then press the "Save to Company API Home Tests" button, as shown in the next screenshot.

You should now see your request appear in the "Company API Home Tests" collection in the sidebar on the left. You'll see the name of the test and the HTTP method "GET" next to it. By default, Postman makes all new tests HTTP GET requests.

SAVE REQUEST

Requests in Postman are saved in collections (a group of requests).

[Learn more about creating collections](#)

Request name

Company Home

Request description (Optional)

Make things easier for your teammates with a complete request description.

Descriptions support [Markdown](#)

Select a collection or folder to save to:

A screenshot of a 'Save Request' dialog. At the top is a search bar with a magnifying glass icon and the placeholder 'Search for a collection or folder'. Below the search bar is a list of collections. The first item in the list is 'Company API Home Tests' with a left arrow icon to its left. To the right of the list is a red '+ Create Folder' button. The rest of the list area is empty and grayed out.

Cancel

Save to Company API Home Tests

Enter the Request Details

The next step is to set up the actual HTTP call to be associated with this test. To do that, click the test name to bring up the main page in the Postman client. In the upper part of the main screen, you'll see the word *GET* (as in "HTTP GET") and a space for entering the URL for the HTTP request. For this lesson, type <http://company-atk.herokuapp.com> into the request line and press the Send button.

If everything goes well, you should get an **OK 200** response back with an HTTP body that looks like this:

ch09-testing/company-response.json

```
{  
  "home": [  
    {  
      "id": "list",  
      "name": "company",  
      "rel": "collection company",  
      "href": "http://company-atk.herokuapp.com/list/"  
    }  
  ]  
}
```

For now, press the Save button (next to the Send button) before we go to the next step of actually writing tests for this request.

Write Your Tests

We're finally ready to write real tests. First, click the Tests link that appears along the upper part of the main Postman screen. That should bring up a large blank editing window. That's where we'll be writing our tests. As a tip, it's a good idea to execute the HTTP request (press the Send button) to make sure it's valid before you start writing tests. It can also help to make sure the response body is displayed at the bottom of the page as a reference.

The first test we'll write will check the HTTP-level elements of the response. It's always a good idea to validate at least the HTTP status code and the **Content-Type** header on each response. To do that, just copy the tests we saw in [Using ChaiJS Assertions](#), and paste them into the editor window, like this:

ch09-testing/company-tests-http.js

```
*****  
* COMPANY HOME  
*****
```

```

// 200 OK
pm.test('Status is 200', function() {
    pm.expect(pm.response.code).to.equal(200);
});

// application/json
pm.test('Content-Type header is application/json', function() {
    var hdr = pm.response.headers.get('content-type');
    pm.expect(hdr).to.include('application/json');
});

```

After copying the tests into your Postman editor and pressing the Send button, you should see the response body and the indicator (2/2) next to the Test Results tab.

Click the Test Results tab to display your test details. They should look something like what's shown in the following screenshot:

The screenshot shows the Postman interface with the 'Test Results' tab selected. There are two test results listed:

- PASS** Status is 200
- PASS** Content-Type header is application/json

Next we need to test the body of the response to make sure it has the expected structure. In our case, we expect a call to the root of the Company API to return a `home` object that contains a link object with `id`, `name`, `rel`, and `href` properties. So let's write that test and add it to the Postman editor.

Here's how I wrote my test:

ch09-testing/company-tests-body.js

```

// collect the response body
var body = pm.response.json();

```

```
// expect body be valid home object
pm.test('Response body contains a valid home object', function() {
    pm.expect(body.home).to.be.an('array');
    pm.expect(body.home[0]).to.have.property('id');
    pm.expect(body.home[0]).to.have.property('name');
    pm.expect(body.home[0]).to.have.property('rel');
    pm.expect(body.home[0]).to.have.property('href');
});
```

When you save the test and then press Send, you should see “Test Results (3/3),” indicating that all three tests passed.

We’ve checked the HTTP values of the response and validated that the body includes a `home` object with the expected properties. Before we’re done here, let’s do one more test. Let’s test the *values* of the `home` object to make sure the expected properties actually contain the expected values.

For example, we expect the `home` object to contain an `id` property set to `list`, a `name` property set to `company`, and a `rel` property that contains both `company` and `collection`. Here’s a set of tests for that:

ch09-testing/company-tests-properties.js

```
// check property values
pm.test('home.id is set to list', function() {
    pm.expect(body.home[0].id).to.include('list');
});
pm.test('home.name is set to company', function() {
    pm.expect(body.home[0].name).to.include('company');
});
pm.test('home.rel contains company and collection', function() {
    pm.expect(body.home[0].rel).to.include('company');
    pm.expect(body.home[0].rel).to.include('collection');
});
```

After saving and running the test, you should get an indicator that shows “Test Results (6/6)” and a display like this:

Body Cookies Headers (9) Test Results (6/6)

All Passed Skipped Failed

PASS Status is 200

PASS Content-Type header is application/json

PASS Response body contains a valid home object

PASS home.id is set to list

PASS home.name is set to company

PASS home.rel contains company and collection

We could keep writing more tests here, but I think you get the pattern. For this series, we tested the HTTP (Did we get a “200 OK” status code and the expected content type?), the structure of the response body (Does it have the right properties?), and the response values (Are the properties set to expected values?). One set of tests we didn’t do yet are the “sad path” tests. For example, what errors do we expect to encounter? I’ll show some of those near the end of this chapter.

What About Schema Testing?

If you’ve done API testing before, you might be wondering why I haven’t mentioned writing tests using JSON Schema. Many times I see developers writing tests that apply a JSON Schema document to the response body from an API to make sure the body matches the expected schema.

I discourage relying on JSON Schema for writing API consumer tests because this can result in rather brittle tests. Each time the schema changes, the test breaks. Especially early in the build process, you’ll be making lots of changes. That means lots of broken tests, which can really slow down the build process in general.

Instead, I rely on the pattern I showed you in this chapter. Test the protocol (HTTP), test the body structure, then test the body values. This works even when there are changes in the shape and detail of the response.

Over time, you'll need to make changes to the API, and that's when taking the protocol-structure-values pattern for testing pays off. Even when the response *schema* changes, you can be assured your original tests are still valid. Maintaining valid tests means you can keep using them in the future and be confident that your new changes aren't breaking existing API clients.

Writing a Sad-Path Test in Postman

I spoke earlier in this chapter about the difference between “happy path” and “sad path” testing and how important it is to use both when testing your API (see [Test Both Happy and Sad Paths](#)). Using what you just learned, let’s write up a sad-path test for BigCo’s API. For example, let’s create a test that confirms we can’t add a new company record if we’re missing required properties.

First, we can add a new request to the Company API Home Tests collection. From the main Postman interface, click the three dots near the collection name in the left sidebar and select Add Request from the drop-down menu. In the dialog box that appears, type “Company Create (missing properties)” as the request name and press the Save button in the lower-right corner of the dialog box.

You’ll now see the new request added to the collection (in the left sidebar). Click that request to bring it into the main editor and enter the following in the Request URL field: <http://company-atk.herokuapp.com>. Also, select **POST** from the pull-down menu to the left of the URL. Save this request to your collection and press the Send button to confirm the request actually works. You should see an error message in the Body tab of the main display telling you the request completed as expected.

Now it's time to write the tests. Click the Tests tab under Request URL and add the following text in the editor (you can find a copy of this test set in the code folder associated with this chapter):

ch09-testing/company-create-tests.js

```
*****
COMPANY CREATE (missing properties)
*****  
  
// protocol
pm.test('Status is 400', function() {
    pm.expect(pm.response.code).to.equal(400);
});  
  
pm.test('Content-Type includes application/problem+json', function() {
    var hdr = pm.response.headers.get('content-type');
    pm.expect(hdr).includes('application/problem+json');
});  
  
// structure
var body = pm.response.json();
var error = body.error  
  
pm.test('root has a valid error object', function() {
    pm.expect(error).to.be.an('array');
    error.forEach(function(obj) {
        pm.expect(obj).has.property('type');
        pm.expect(obj).has.property('title');
        pm.expect(obj).has.property('detail');
        pm.expect(obj).has.property('status');
        pm.expect(obj).has.property('instance');
    });
});  
  
// values
var obj = error[0];
pm.test('error.type includes error', function() {
    pm.expect(obj.type).includes('error');
});
pm.test('error.status includes 400', function() {
    pm.expect(obj.status).includes('400');
```

```

});
pm.test('error.title includes Missing companyName, email, and status',
  function() {
    pm.expect(obj.title).includes('Missing companyName');
    pm.expect(obj.title).includes('Missing email');
    pm.expect(obj.title).includes('Missing status');
  });
pm.test('error.detail includes Missing companyName, email, and status',
  function() {
    pm.expect(obj.detail).includes('Missing companyName');
    pm.expect(obj.detail).includes('Missing email');
    pm.expect(obj.detail).includes('Missing status');
  });

```

Notice in our protocol testing we expect a status code of [400 Bad Request](#) instead of [200 OK](#). This sad-path test “passes” when the request gets rejected by the service behind the API. Also, note that we expect the [content-type](#) value to be set to [application/problem+json](#). This is the standard format for API errors, as described in RFC7807. [\[25\]](#) The tests for structure follow the specification for the RFC, and the tests for the values follow the rules we set for our Company API.

Once you write the tests and save them in the Postman client, you can press the Send button to confirm that all seven of the tests pass as expected.

Creating Your Own Postman Test Libraries

Sometimes writing tests can seem a bit tedious. It definitely takes time and commitment to create valuable tests that won’t easily break as you modify your API over time. Also, one of the advantages of BDD-style assertion kits like the ChaiJS library is that it has a fluent interface that results in lots of text to type when you’re writing the tests. For this, and other reasons, I have created some Postman test libraries that I can easily add to any Postman collection to make writing comprehensive tests a bit easier and faster.

For example, I have a Postman test library that includes a number of handy functions for writing protocol-structure-value test collections like the one we

saw earlier in this chapter. The following is a summary of the library to give you an idea of how it looks:

ch09-testing/postman-utils-stubs.js

```
pm.globals.set('loadUtils', function loadUtils() {
    let utils = {};
    let obj = '';

    utils.checkStatus = function(value) {...};
    utils.checkHeader = function(args) {...};
    utils.setObject = function(args) {...};
    utils.checkObject = function(args) {...};
    utils.checkObjectProperty = function(args) {...};
    utils.checkMetaProperty = function(args) {...};
    utils.checkPageLink = function(args) {...};
    utils.checkPageLinkProperty = function(args) {...};
    utils.checkItem = function(args) {...};
    utils.checkItemLink = function(args) {...};
    utils.checkItemLinkProperty = function(args) {...};
    utils.checkItemProperty = function(args) {...};
    utils.checkError = function() {...};
    utils.checkErrorProperty = function(args) {...};

    return utils;
} + ';\nloadUtils();');
```

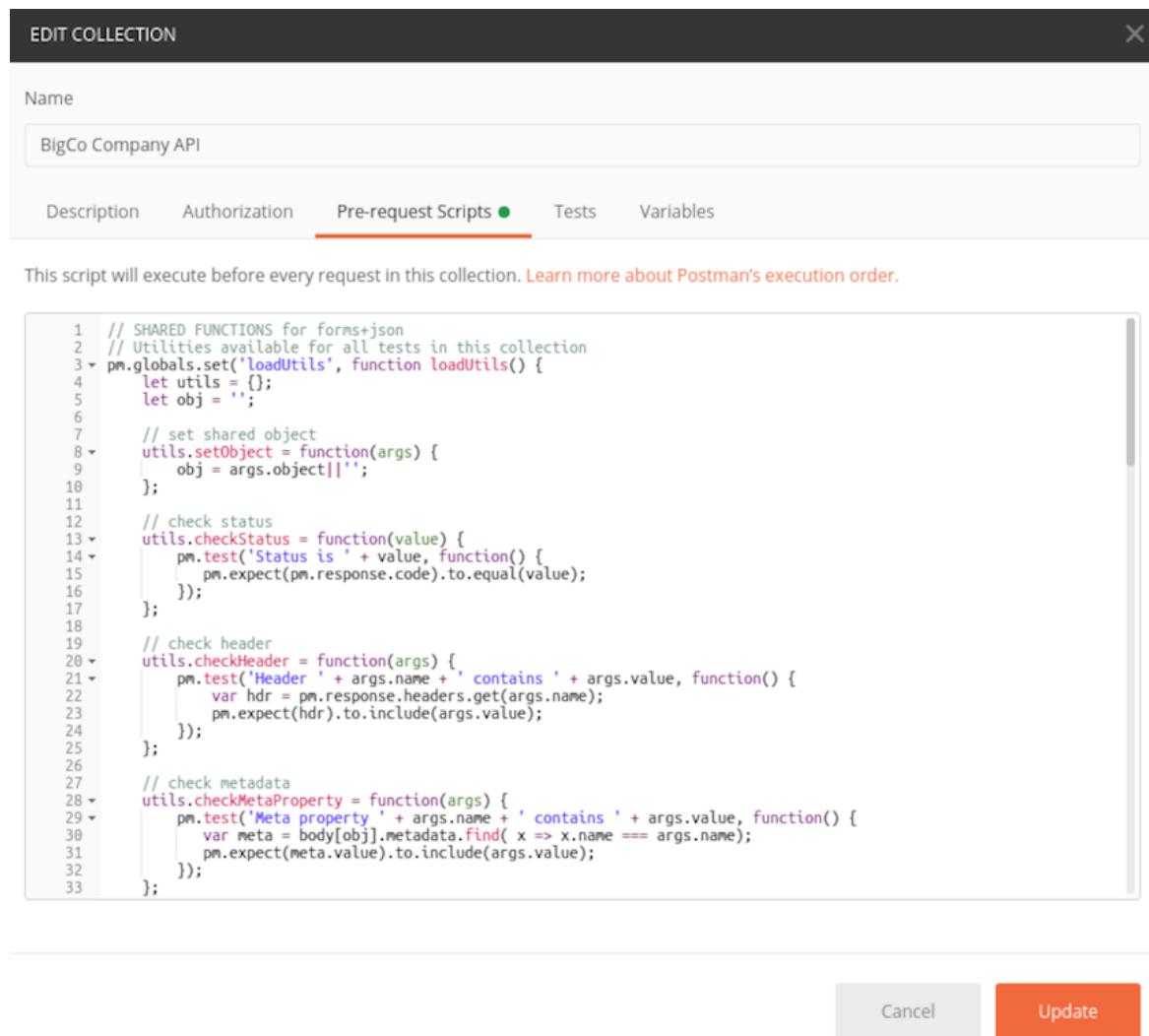
I won't go into the details of the library here, but you can find the complete set of functions in the code folder associated with this chapter of the book.

You might have noticed that the start and end of the code snippet looks a bit peculiar. The collection starts and ends with some code unique to Postman:

```
pm.globals.set('loadUtils', function loadUtils() {....} + ';\nloadUtils();');
```

What's going on here is that we're loading the utility library into Postman's global memory space. Using this technique, you can access the utility library from any test in the collection. To be able to do this, you need to paste the complete utility library into the collection's "Pre-request Scripts" window.

Here's how you make this work. First, roll your mouse over the three dots next to the collection name in the left sidebar of the Postman client. Then click the three dots to pull up the context menu and select Edit from that menu. This will pull up the Edit Collection dialog box. Then select the "Pre-request Scripts" tab and paste the utility code (from the chapter code folder), as shown in [the screenshot](#). After pressing the Update button, you'll be back at the main Postman client interface, ready to start writing individual tests that access the library.



The screenshot shows the 'Edit Collection' dialog box. At the top, it says 'EDIT COLLECTION' and has a close button ('X'). Below that is a 'Name' field containing 'BigCo Company API'. Underneath the name is a navigation bar with tabs: 'Description', 'Authorization', 'Pre-request Scripts' (which is highlighted in red), 'Tests', and 'Variables'. A note below the tabs states: 'This script will execute before every request in this collection. [Learn more about Postman's execution order.](#)' The main area contains a large block of JavaScript code:

```
1 // SHARED FUNCTIONS for forms+json
2 // Utilities available for all tests in this collection
3 pm.globals.set('loadUtils', function loadUtils() {
4     let utils = {};
5     let obj = '';
6
7     // set shared object
8     utils.setObject = function(args) {
9         obj = args.object|| '';
10    };
11
12    // check status
13    utils.checkStatus = function(value) {
14        pm.test('Status is ' + value, function() {
15            pm.expect(pm.response.code).to.equal(value);
16        });
17    };
18
19    // check header
20    utils.checkHeader = function(args) {
21        pm.test('Header ' + args.name + ' contains ' + args.value, function() {
22            var hdr = pm.response.headers.get(args.name);
23            pm.expect(hdr).to.include(args.value);
24        });
25    };
26
27    // check metadata
28    utils.checkMetaProperty = function(args) {
29        pm.test('Meta property ' + args.name + ' contains ' + args.value, function() {
30            var meta = body[obj].metadata.find( x => x.name === args.name);
31            pm.expect(meta.value).to.include(args.value);
32        });
33    };
}
```

At the bottom right of the dialog are two buttons: 'Cancel' and 'Update' (in an orange box).

For example, with this library loaded, you can rewrite the Company Home test to look like this:

```
ch09-testing/company-home-tests-utils.js
```

```
*****  
* COMPANY HOME UTILS  
*****/  
  
var body = pm.response.json();  
var utils = eval(globals.loadUtils);  
  
utils.checkStatus(200);  
utils.checkHeader({name: 'content-type', value: 'application/json'});  
utils.checkObject({name: 'home', has:[ 'id', 'name', 'rel', 'href']});  
utils.checkObjectProperty({name: 'home', property: 'id', value: 'list'});  
utils.checkObjectProperty({name: 'home', property: 'name', value: 'company'});  
utils.checkObjectProperty({name: 'home', property: 'rel', value: 'company'});  
utils.checkObjectProperty({name: 'home', property: 'rel', value: 'collection'});
```

Notice the added line in the script that references the utility library stored in Postman's global memory:

```
var utils = eval(globals.loadUtils);
```

Once you have your tests written and working, it's time to automate the testing process to make it quick and easy to execute from the command line. For that, we'll use a new tool called [newman](#).

Make Sure You Trust the Code



The trick of loading Postman's global memory with JavaScript utility code relies on JavaScript's `eval` keyword. That keyword essentially treats plain text as source code without checking for any problems. This is a risky move, and Postman will give you a little warning icon next to the line with the `eval` keyword to remind you. You should only do this when you trust and understand the utility code. That includes the library I'm sharing with you.

Running Tests Locally with Newman

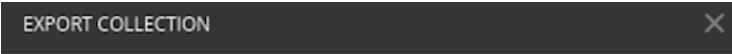
Now that we have some API tests built with Postman, it's time to move from the Postman UI to a command-line interface. While the UI is nice, you'll find the CLI will let you run the tests from simple scripts, launch them very quickly, and make it easier to incorporate test runs into your regular development habits.

To run Postman tests locally on your own machine, you can use a CLI utility called `newman`. (See [Newman](#), for details on how to check for and install this utility.) `newman` is a tool created by the Postman team. It supports running test collections directly from the command line and outputting the results for viewing or storage within your project. Once you have `newman` installed, you'll need to pull one of your test collections from Postman and save it locally.

Exporting a Postman Collection

The first step to running tests locally is to export an existing test collection from the Postman app down to your local workstation. To do that, open the Postman client and locate a collection you want to export in the left sidebar. For this example, let's pick the Company Home API Tests collection we were using earlier in this chapter.

To start the export, click the three dots next to your collection name to bring up the drop-down menu. Select Export from the menu to call up the export collection dialog box, as shown in the following screenshot:



Company API Home Tests will be exported as a JSON file. Export as:

- Collection v1 (deprecated)
- Collection v2
- Collection v2.1 (recommended)

[Learn more about collection formats](#)

Cancel

Export

Be sure the “Collection v2.1” radio button is selected and then press the Export button. This calls up the File Save dialog box, where you can navigate to your project folder and save the file. For this example, save the file to your project folder [/tests/postman/](#) with the name **postman-collection-export.json**. Once you do that, you’ll be ready to test your **newman** utility.

Pick Me!



To make it a bit easier to follow along in this section, you’ll find a Postman collection export stored in the source code associated with this chapter at <https://pragprog.com/book/maapis>. Just copy the **postman-collection-export.json** file to your local project and you will be all set.

Reading Newman’s Test Results

Once you have the collection stored on your local machine, you can run your tests directly on your laptop or workstation. To do this, open a command-line window and navigate to the folder that holds the test collection export file (**postman-collection-export.json**). Then you just need to invoke the **newman** utility using the **run** command, like this:

```
mca@mamund-ws:~/testing$ newman run postman-collection-export.json
```

When you run the command, `newman` will run the tests and pipe the results to the screen. It should look something like this:

ch09-testing/newman-output.txt

```
newman

Company API Home Tests

→ Company Home Starter
GET http://company-atk.herokuapp.com [200 OK, 563B, 148ms]

→ Company Home
GET http://company-atk.herokuapp.com [200 OK, 563B, 22ms]
✓ Status is 200
✓ Content-Type header is application/json
✓ Response body contains a valid home object
✓ home.id is set to list
✓ home.name is set to company
✓ home.rel contains company and collection

→ Company Home Utils
GET http://company-atk.herokuapp.com [200 OK, 563B, 21ms]
✓ Status is 200
✓ Header content-type contains application/json
✓ Valid home object
✓ home has property id set to list
✓ home has property name set to company
✓ home has property rel set to company
✓ home has property rel set to collection
```

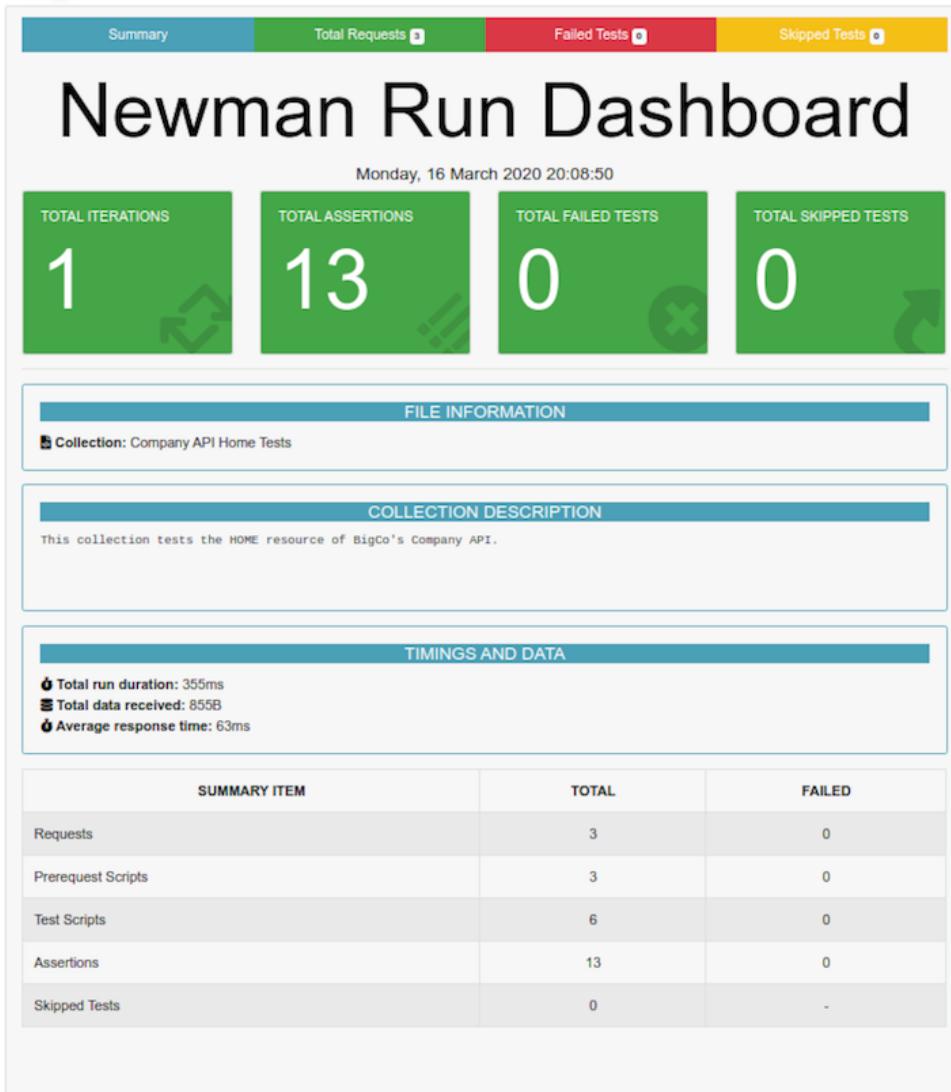
	executed	failed
iterations	1	0
requests	3	0
test-scripts	6	0
prerequest-scripts	3	0

assertions	13	0
total run duration: 360ms		
total data received: 855B (approx)		
average response time: 63ms		

For long test runs, you can redirect the output to a file like this:

```
mca@mamund-ws:~/testing$ newman run postman-collection-export.json \
> newman-output.txt
```

But there's an even better way to produce test output using the [reporter](#) options in [newman](#). By using the `-r` command-line option, you can instruct [newman](#) to generate an HTML report that provides additional details in a more interactive experience (check out the [screenshot](#)).



To produce this report, add `-r htmlextra` to the end of the command:

```
mca@mamund-ws:~/testing$ newman run postman-collection-export.json \
-r htmlextra
```

This creates a new HTML document (with a generated unique name) in a subfolder called `newman`. You can store these test reports in your API project and check them into your Git repository.

And that wraps up testing and reporting with Postman and Newman.

What's Next?

We've covered the basics of applying Dan North's "outside-in" approach to API testing and using that model for writing simple request tests (SRTs) and more extensive Postman collections. Now we're ready to move on to another important element of creating great APIs: security.

In the next chapter, we'll look at the difference between API authentication and API authorization and learn how to use OpenAuth (or OAuth) to enable fine-grained security for your API projects. And we'll use the Auth0 ("Auth-zero") SaaS platform to make it all work.

Chapter Exercise

For this exercise, we'll create a Postman test collection for the `credit-check` API you built in the previous chapter. After creating a new collection and adding both happy- and sad-path tests, we'll run them to validate the API.

For extra credit, you can export your API and run it locally using the `newman` utility to produce an HTML report.

Write the Credit-Check Postman Test Collection

First, fire up your Postman client app and create a new collection called `Credit-Check`. Then create the following two test entries:

Happy-Path Test

- Test the `home` resource
 - Request URL: `http://localhost:8181/`
 - Method: `GET`
 - Body: None
 - Protocol Tests
 - http status equals `200`
 - `content-type` includes `application/json`
 - Structure Tests
 - `home` object at the root is an array
 - `home` has `id`, `name`, `href`, and `rel` properties
 - Content Tests
 - `home.id` includes `list`
 - `home.name` includes `credit-check`
 - `home.rel` includes `credit-check` and `collection`

Sad-Path Test

- Test the `from` resource
 - Request URL: `http://localhost:8181/form/`

- Method: **POST**
- Body: None
- Protocol Tests
 - http status equals **400**
 - **content-type** includes **application/problem+json**
- Structure Tests
 - **error** object at the root is an array
 - **error** has **type**, **title**, **detail**, **status**, and **instance** properties
- Value Tests
 - **type** contains **error**
 - **title** contains **Missing companyName**
 - **detail** contains **Missing companyName**
 - **status** contains **400**

Use the Postman client to add these two requests and write the test script using the ChaiJS assertion library as we did earlier in this chapter. Save the tests to Postman.

Run the Test with the Postman Client

After writing and saving the tests, use the Postman client to run the tests and review the results.

Extra Credit: Run the Tests with the Newman Client

After you are sure all tests pass as expected, export your test collection and save it to disk. Then use the **newman** CLI utility to run the tests locally. Be sure to use the **htmlextra** reporting option to produce an HTML report of your test results.

See Appendix 2 ([*Solution for Chapter 9: Testing APIs*](#)) for the solution.

Footnotes

[²⁰] <https://dannorth.net/introducing-bdd>

[²¹] <https://dannorth.net/whats-in-a-story>

[72] <https://www.postman.com>

[73] <https://www.chaijs.com>

[74] https://en.wikipedia.org/wiki/Fluent_interface

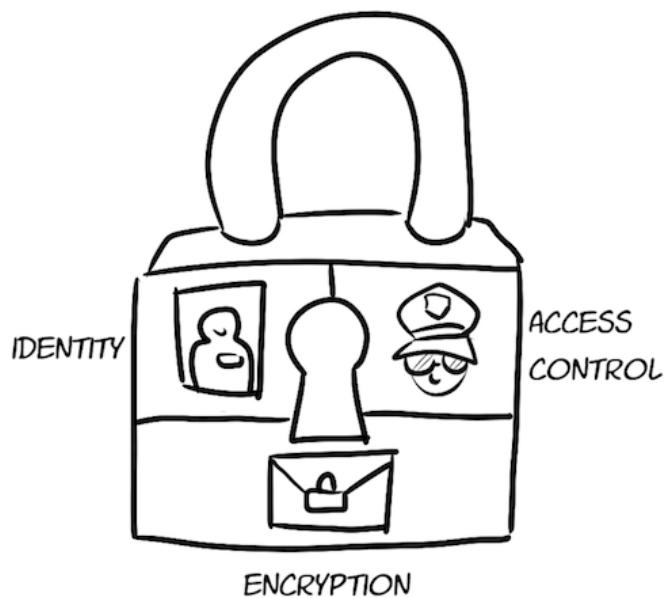
[75] <https://tools.ietf.org/html/rfc7807>

Copyright © 2020, The Pragmatic Bookshelf.

Chapter 10

Securing APIs

In this chapter, we learn how to secure your API. API security consists of three key elements: identity, access control, and encryption. We'll explore each one in turn and then focus on a solution for implementing each of them in a direct way.



After we review the security basics, we'll dive into implementing them via standards called OpenAuth (OAuth)^[76] and JSON Web Token (JWT).^[77] OAuth is a protocol for requesting and sharing access control information for a particular user or machine. JWT is a standard way of representing access control in the form of a *token*. When used together, you can easily add security features to your API in an independent and standardized way.

To make the whole process easier, we'll use an online provider called Auth0 (“auth-zero”) to do the work of generating and validating our access control tokens.^[28] We'll also make some modifications to our API project in order to communicate with the Auth0 website when we need to enforce our security rules. Finally, we'll take advantage of a couple of local bash scripts to simplify managing and testing our API security.

We'll cover a lot of details in this chapter, so let's get started with some security basics.

Understanding Security Basics

The key to understanding API security is to focus on two related elements: identity and access control. These work at the API level. In other words, when you are implementing the API, you need to decide if and when you'll apply identity and access control checks.

It's also important to understand the role of encryption as an additional layer of security. For HTTP-based APIs, this works at the protocol level. The most common way to recognize the use of encryption on the web is through the use of the [https](#) identifier (called a URI scheme) instead of the [http](#) identifier in your URLs.

These two items—identity/access control and encryption—can work independently of one another too. In this first part of the chapter, we'll focus on identity and access control and then discuss encryption. In the second half, you'll learn how to implement identity and access control for your APIs using the Auth0 online service and the OAuth^[79] and OpenID^[80] protocols.

Before we get wrapped up in the implementation details of securing your APIs, let's take a minute to review the roles identity, access control, and encryption play on the web.

Identifying API Users with Authentication

A good place to start when tackling the topic of API security is the concept of *identity* (sometimes called *authentication*). Determining the identity of the person or machine making the API call is a key step in the process of securing your API.

The most common way to determine identity on the web is by using the OpenID and OAuth protocols together. OAuth is a set of standards focused

on authorization. OpenID is a set of standards focused on authentication. OpenID Connect is the identity layer built on top of the OAuth access control specification. [81]

OAuth or OpenID?



Often when people talk about “OAuth,” they actually mean both authentication and authorization. In this chapter, I use the word OAuth in that way—to cover both identity and access control.

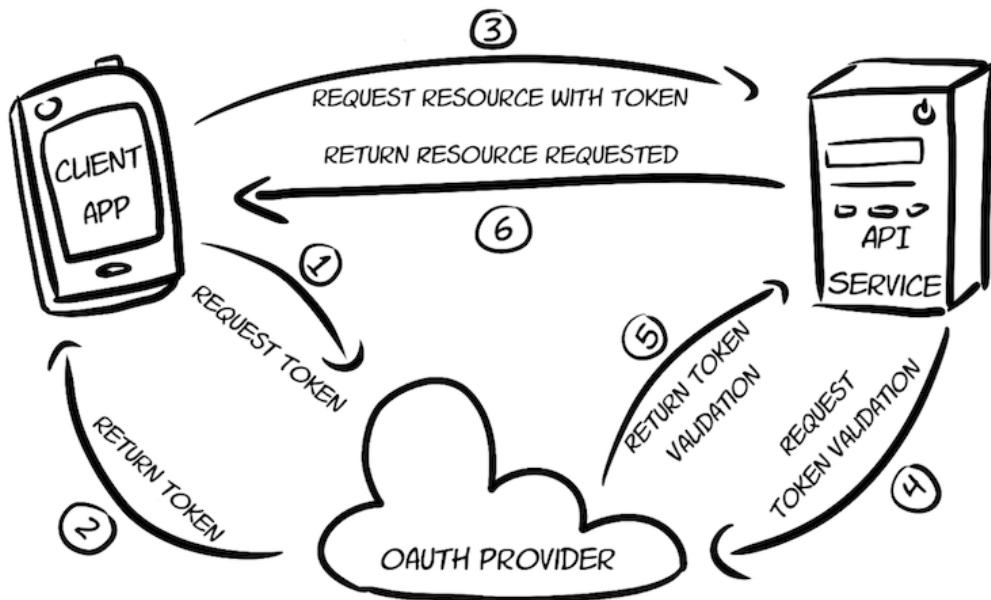
OAuth is a common API security solution because it offers quite a number of authentication workflows (called *grant types*) to fit various levels of security and interaction needs. An important feature of OAuth is that it supports what’s called *three-legged authentication*. This means the actual authentication event happens at the provider service, which provides an authentication token that’s then used by the client (application or API) to present to the actual service.

The process works like this:

1. The client application makes a request to an OAuth provider service to get a valid token. This is where the username and password are supplied by the client application. For machine-to-machine APIs, we’ll supply the client ID and the client secret (more on this later).
2. The OAuth provider validates the login identity and returns a special encoded token. This token is what the client application will use when making requests to the API service.
3. The client application sends a request to the API service (for example, `GET http://api.example.org/company/132435`). The client application includes the token it got from the OAuth provider in the `Authorization` HTTP header.

4. The API service accepts the request and the token from the client application and sends the token to the OAuth provider to ask if that token is valid.
5. The OAuth provider validates the token and returns it to the API service.
6. The API service, having determined this is a valid token for this request, returns the requested resource response that the client application asked for in Step 3.

See the [diagram](#) for an illustration of these steps.



Although this process is a bit more complicated than sending the API service a simple username and password, three-legged models have a number of important advantages to consider. First, using an external OAuth provider to handle the login steps means the user passwords don't need to be stored by the API service. Second, it allows the API service and the client application to support more than one OAuth provider. For example, the API service might support using Microsoft, Google, Twitter, Facebook, and other OAuth providers. They can even add new providers over time without disrupting the existing client applications.

All the examples in this book use Auth0 as our “third leg” in the triangle. API user applications will first log in to Auth0 to get a security token and then use that token when talking to our API. (I go into the details of this in [Implementing API Security with Auth0](#).)

Identifying the API user making the request is just part of the job of completing a secure API transaction. It’s also important to understand the access control limits each request has for any services it attempts to contact.

Controlling API Access with Authorization

The act of authorizing a request is, essentially, associating access rights to the request. For example, if an API call to the `company` services attempting to execute the `removeCompany` action was initiated by me, and the third-party authentication service (in our case, Auth0) was able to identify that it was me (`MikeAmundsen`) making that request, then the work of authorizing that API request would be to verify that `MikeAmundsen` has rights to access the `company` service *and* that `MikeAmundsen` has rights to execute the `removeCompany` action.

Access control can be applied and validated a couple of different ways. For example, identities can be associated with roles (such as `admin`, `manager`, `user`, `guest`). The `company` might have a built-in rule that any request with an identity associated with the `admin` role is allowed to execute the `removeCompany` action. Access control can also be applied directly to identities. Using the earlier example, that would mean that the `company` service would know about a list of identities (such as `MikeAmundsen`) and their associated execution rights.

In both of these examples, the real work of sorting out which identity has which access right is the work of matching validated identities with pre-determined actions. How and where that’s done varies by organization. All the services shown in this book rely on an external service (Auth0) to do

that work. In your company, you may have your own third-party OAuth provider or you may have an OAuth service running inside your network. It's also possible that your company does not use the OAuth pattern but just requests usernames and passwords directly. However your organization does it, you'll still need to handle both authentication (determining identity) and authorization (validating access control).

Now that we've looked at the roles of identifying users through authentication and determining their access rights through authorization, we need to cover one more API security basic: message-level encryption.

Protecting API Traffic with Encryption

Encrypting API requests and responses offers a level of security for messages as they pass from one place (the client app) to another (the service). It's good practice to implement message-level encryption for your API. This is especially true when interaction with your service crosses a network boundary. Your company's network, for example, is a boundary. If any API calls come from outside that network, you should encrypt the traffic in order to protect it from eavesdropping, tampering, and message forgery.

The most common message-level encryption implementation is to use Transport Layer Security (TLS).^[82] The role of TLS is to prove what's called a "secure channel" between two parties. TLS requires a bit of setup, or *handshaking*, to establish the identities of each party and a mutual encryption scheme. When that's done, the two parties use the same encryption details for each message they pass back and forth. You may be familiar with the Secure Sockets Layer (SSL) protocol.^[83] TLS is the successor to SSL. They work pretty much the same way and are used for the same purpose.

In order to access TLS encryption over HTTP, you need to use the Hypertext Transfer Protocol Secure, or HTTPS protocol. That means using

the `https` URI scheme when making your calls to another machine on the web. The HTTPS protocol encrypts the HTTP headers, body, and cookies before it sends them to the other party. The party receiving the message will be able to decrypt these same message parts (headers, body, and cookies) once the message arrives at its destination.

The good news is that the TLS encryption and decryption functionality is built into all browsers and almost all HTTP coding libraries. That's true for the NodeJS and Express libraries we're using for the examples in this book. You simply need to prefix your API calls with `https://...` instead of `http://...` and you'll get all the benefits of TLS encryption.

The URL Is Always in the Clear



It's important to point out that the contents of the HTTP request line are *not* encrypted. If you're passing sensitive data over the URL, as in `/user-service/filter?SSN=123-45-6789&lastName=Smith`), this data will be sent unencrypted, or "in the clear," even when using HTTPS protocol.

A past roadblock of using HTTPS is that for you to implement it, you need to acquire a TLS certificate and install it on the server that's hosting your API. This used to be a complicated and sometimes expensive affair.

However, starting in 2014, the Internet Security Research Group (ISRG^[84]) has been hosting the site Let's Encrypt,^[85] where you can get a free TLS certificate along with help installing it on your servers. Access to free, trusted TLS certificates means there are few good reasons *not* to implement encryption for your APIs.

Encrypting the BigCo, Inc., APIs

Encrypting the BigCo, Inc., APIs



All the major API hosting platforms (Microsoft Azure, Google Cloud, Amazon Web Services, and others) can support TLS certificates over [https](https://). For this book, we're hosting all our BigCo API services on the Heroku platform and will get to use message encryption via TLS certificates through [https](https://) without any additional setup.

Now that we've covered the basics of identity, access control, and encryption, it's time to actually implement these concepts for our APIs.

Implementing API Security with Auth0

We're going to use the Auth0 online platform for our API security. We'll need to log in (or sign up) at the website, define our API in the Auth0 system, and collect key authentication parameters that we'll need in order to access the secured API (for example, our access token). We'll also learn how to validate access tokens with the [JWT.io](#) website.^[86]

Once we have that taken care of, we can modify our API service to support secure connections, and then we can test that using the access token supplied by Auth0. But first, let's log in to our security provider and define our secure API.

Logging In to Auth0

The first step in adding security to our API will be to log in to Auth0.^[87] I like using Auth0 because you can start small and build up a more complex security profile as you need it. Your company may be using a different external service, or you may want to implement one of your own. The important thing here is to understand the key concepts so that you can translate them to your own environment.

If you already have an Auth0 account, just go to the home page and click the Login button. If you need to create an account, visit the Sign Up page to get started.^[88]

Once you sign up, you'll see your main Auth0 dashboard, as shown in the first [screenshot](#). We'll use this screen quite a bit throughout this chapter.



Dashboard

Applications

APIs

SSO Integrations

Connections

Universal Login

Users & Roles

Rules

Hooks

Multifactor Auth

Emails

Logs

Anomaly Detection

Extensions

Get Support

Login Activity

Day boundaries in UTC

May June July August September October November December January February March April

4 0 0

Users All time Logins Last 7 days New Signups Last 7 days

Latest Logins No recent activity New Signups No recent activity

A screenshot of the Auth0 dashboard. On the left is a sidebar with various navigation options. The main area shows a heatmap of login activity from May to April. Below the heatmap are three summary statistics: 4 users (All time), 0 logins (Last 7 days), and 0 new signups (Last 7 days). At the bottom, there are two sections: "Latest Logins" and "New Signups", both showing "No recent activity". A prominent orange button in the top right corner says "+ CREATE APPLICATION".

Defining Your API

On the main Auth0 dashboard, several options appear in the menu on the left side of the screen. Click the APIs option to bring up the list of defined APIs. (You might not see any if this is your first login at Auth0.)

To define your API, click the Create Application button that appears in the top-right corner of the screen. That brings up the New API dialog box, shown in [the second screenshot](#).

New API

X

Name *

A friendly name for the API.

Identifier *

A logical identifier for this API. We recommend using a URL but note that this doesn't have to be a publicly available URL, Auth0 will not call your API at all. **This field cannot be modified.**

Signing Algorithm *

Algorithm to sign the tokens with. When selecting RS256 the token will be signed with Auth0's private key.

CREATE

CANCEL

First, enter the name of the API you wish to define. For this example, we'll use **bigco-company**. (This is just a text field that you can change later.) The next field you need to fill in is the Identifier field. You can enter any unique value here (no spaces). It's common practice to use a URI in this field. For this example, we'll supply **http://mamund.com/bigco-company**. Leave the Signing Algorithm text field set to the default value and press the Create button.

After clicking Create, you'll see the landing page for the **bigco-company** API. Your API is now defined in Auth0. You just need to collect some key parameters generated by Auth0, and then you'll be ready to test your new credentials and generate an API access token.

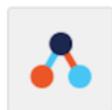
Collecting API Identity Parameters

When you defined your API at Auth0, it created a handful of important parameters you'll need to keep track of when you update your API service to check for access tokens and to use a client app to send the access token when calling the API service. Let's go through the values you need to keep track of and then we'll talk about how to use them.

First, on the Auth0 website, go to your APIs page (through the link located in the menu on the left) and select the API we just created ([bigco-company](#)). You should now see the landing page for the bigco-company API. Select the Settings tab and note the Name ([bigco-company](#)) and Identifier (<http://mamund.com/bigco-company>) values. Copy these to a text file named `auth0.env` and save that file to a new subfolder in your project named `security/`. We'll use this folder to hold a number of important security-related files and scripts.

Next, select the Applications option from the menu on the left and, when the list of applications appears, select *bigco-company (Test Application)* in the list, which was automatically created when you defined your API earlier. Clicking the name of the application brings up the application landing page with the Settings tab selected. It will look similar to the screen [shown](#).

[← Back to Applications](#)



bigco-company (Test Application)

MACHINE TO MACHINE

Client ID

JEAvSA9ecQB5TUg7Fp0e4Kr8pDQd9ArP

Quick Start Settings APIs

Basic Information

Name *

bigco-company (Test Application)



Domain

mamund.auth0.com



Client ID

JEAvSA9ecQB5TUg7Fp0e4Kr8pDQd9ArP



Client Secret

.....



The Client Secret is not base64 encoded.

From this screen, copy and paste the domain, client ID, and client secret values into your **auth0.env** text file.

Our Little Secret



Be very careful with these API parameter values, especially the client secret. Anyone who has access to this information can create a valid access token and interact with your API.

Using the five parameter values you've collected (name, identifier, domain, client ID, and client secret), complete your **auth0.env** file using the template that follows as a guide:

ch10-securing/auth0.txt

```
#####
# auth0.env environment vars for bigco-company API
#####

name=<NAME>

id=<CLIENT ID>
secret=<CLIENT SECRET>

url=<DOMAIN>/oauth/token

jwksuri=<DOMAIN>/.well-known/jwks.json
audience=<IDENTIFIER>
issuer=<DOMAIN>

### EOF
```

Now we can use this file in some scripts later to generate valid access tokens and test the security of your API. The `id`, `secret`, and `audience` parameters will be used by client apps to generate new access tokens. The `url`, `wjksuri`, `audience`, and `issuer` parameters will be used by your API service to communicate with Auth0 and validate the access token passed to the service by the client apps.

Now let's generate our first access token for the `bigco-company` API.

Generating Your Access Token

Your client applications will need a valid access token in order to communicate with your secured API service. For human-driven apps (like mobile browser apps), this access token is generated when the human enters his or her username and password (also called their *credentials*). But our API won't be talking directly to a human, it'll be talking to another application—another machine. For this reason, this kind of API security profile is referred to as “machine-to-machine” interaction.

To get a valid access token for a machine application, we need to generate that token ahead of time. This is the replacement for a login dialog. An example of how to do this with curl is shown here:

ch10-securing/generate-token.txt

```
curl --request POST \
--url <DOMAIN>/oauth/token \
--header 'content-type: application/json' \
--data '{"client_id":<CLIENT ID>,
         "client_secret":<CLIENT SECRET>,
         "audience":<AUDIENCE>,
         "grant_type": "client_credentials"
}'
```

Notice you need to insert values from the [auth0.env](#) file into this single [curl](#) command. Also notice the use of the `"grant_type": "client_credentials"` parameter in the body posted to Auth0. There are several possible ways (*grant types*) to complete an access token request. In this book we'll use the [client_credentials](#) method.

When you execute this command, you're contacting the Auth0 server and asking it to generate an access token for future API calls. The HTTP response you'll get from Auth0 will look like this:

ch10-securing/access-token.json

```
{
  "access_token": "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCI...",
  "token_type": "Bearer"
}
```

Your `access_token` value will be a very long encoded string. I included a few of the characters in the book. We'll need to copy this really long string into each and every API request in order to be able to access a secured API service. (More on that in the next section of this chapter.) For now, be sure to copy and paste this value into a text file because we have one more step to go before we're done with our access token—we need to validate it.

Validating Your Access Token

Access tokens from the Auth0 website are based on the JSON Web Token (JWT) specifications (RFC7519).^[89] JWTs are *encoded* tokens that contain three parts: a header, a body, and a signature. You can actually decode the contents of a JSON Web Token too. This can come in handy when you want to validate or debug a token that doesn't seem to be working as expected.

You'll find a JWT decoder at the [JWT.io](#) website,^[90] which is operated by Auth0. To see what's encoded into a JWT, copy and paste the access token you generated in the previous section of the book into the decoder at [jwt.io](#). When you do, you should see something similar to what's shown in this screenshot:

The screenshot shows the jwt.io interface with two main sections: 'Encoded' and 'Decoded'.

Encoded: A text input field containing a long, base64-encoded JWT string. The string is as follows:

```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCIsImtpZCI6Ik9UTTFNVGMxTVRVeU5qQTNSREUyUmpRM1FrTkNRakpFT1VJd1JURkNRakE0T1RoR05rRXdRZyJ9.eyJpc3MiOiJodHRwczovL21hbXVuZC5hdXR0MC5jb20vIiwic3ViIjoiSkVBd1NB0WVjUU1VFVnN0ZwMGU0S3I4cERRZDlBclBAY2xpZW50cyIsImF1ZCI6Imh0dHA6Ly9tYW11bmQuY29tL2JpZ2NvLWNvbXBhbnkiLCJpYXQiOjE10Dc0Mjk1MTMsImV4cCI6MTU4NzUxNTkxMywiYXpwIjoiSkVBd1NB0WVjUU1VFVnN0ZwMGU0S3I4cERRZDlBclA1CJzY29wZSI6ImN0ZWNrOnJlcXVlc3QgY2h1Y2s6Y3J1YXRlIiwiZ3R5IjoiY2xpZW50LNnyZWR1bnRpYWxzIn0.eyJEN-vrAVESanriyR7MxgpHoElzsvSp4-CXU060coT0mJzT_E_87xuk85v3jLPHugaZkDVCoqMp4v3pkPLzyAIKZ9sWVTFs6EfZ0GrhlyIVNnRYgM6XhhNkm2uNTY7T0pVW4QUfYUigBsmPyqtDmwQcvznJKsudKY1pEV_-c1vDZucs8YXshoXuG3bbrXfLLF96LErS0p-IRql3rRZnxACMGMpbjcmYOMI6L14uFPUU4gtdGBPwauFKvPDYW6tceIUYxVS-Aqv09laBsKkGLyDE4_AK4EK9Srs6WaQVqpcf8qznp4fTvNGBhb0tS7u-0o6Zsn40uBwoW-TMUc-9FkQ
```

Decoded: The decoded components of the JWT.

- HEADER:** ALGORITHM & TOKEN TYPE
- ```
{ "alg": "RS256", "typ": "JWT", "kid": "OTM1MTc1MTUyNjA3RDE2RjQ2QkNCQjJE0UIwRTFCQjA4NThGNkEwQg" }
```
- PAYOUT:** DATA
- ```
{ "iss": "https://mamund.auth0.com/", "sub": "JEAvSA9ecQ85TUg7Fp0e4Kr8pDQd9ArP@clients", "aud": "http://mamund.com/bigco-company", "iat": 1587429513, "exp": 1587515913, "azp": "JEAvSA9ecQ85TUg7Fp0e4Kr8pDQd9ArP", "scope": "check:request check:create", "gtv": "client-credentials" }
```
- VERIFY SIGNATURE**
- ```
RSASHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), -----BEGIN PUBLIC KEY----- MIIBIjANBgkqhkiG9w0BAQEFAAOQCgYEAbiAivztJzWRr70GaqtK -----END PUBLIC KEY-----)
```

Now you know how to generate and, if needed, validate JWT access tokens. Next we'll update one of our BigCo APIs to support machine-to-machine authentication using our JWT tokens.

## Reading Assignment



There isn't enough room in this chapter to explain all the details of a JWT, but you'll find lots of information at the <https://jwt.io> website and the related RFC7519 specification document.

# Supporting Machine-to-Machine Security

Adding support for machine-to-machine (M2M) security to your API takes just a few steps. You need to add a few modules to your project that contain the functionality to communicate with OAuth providers and evaluate JWTs. You also need to modify your own API service interface to look for and process JWTs when they appear. Finally, you need to import the OAuth authentication parameters you collected from the third-party provider (see [Collecting API Identity Parameters](#)).

## Company-Secure



For the example in this chapter, I've forked the existing `company` sample API service to one named `company-secure`. I'll show you excerpts from that example as we walk through the steps to secure a Node/Express API project. You'll find a copy of the completed API project in the code folder associated with this chapter on the book's web page at [https://pragprog.com/titles/maapis/source\\_code](https://pragprog.com/titles/maapis/source_code).

## Adding Security Modules to Your NodeJS project

First, let's add the following security modules to our NodeJS project. These will provide support for JSON Web Tokens (JWTs) and OpenAuth (OAuth):

- `jsonwebtoken`—NodeJS implementation of the JWT standard
- `jwks-rsa`—RSA signing keys for JWTs
- `express-jwt`—ExpressJS middleware for validating JWTs
- `express-jwt-authz`—ExpressJS middleware for supporting OAuth

The following is the online `npm` command for installing all four modules and updating your `package.json` file:

```
mca@mamund-ws:~/company-secure$ npm install -s jwks-rsa jsonwebtoken \
```

```
express-jwt express-jwt-authz
+ jsonwebtoken@8.5.1
+ express-jwt@5.3.3
+ express-jwt-authz@2.3.1
+ jwks-rsa@1.8.0
added 42 packages from 106 contributors, removed 2 packages and \
audited 255 packages in 24.495s
found 0 vulnerabilities

mca@mamund-ws:~/company-secure$
```

That sets the foundation for implementing OAuth for your API service. The next step is to add supporting code to your project.

## Editing api-auth.js

You'll need some boilerplate code in your project to process incoming JWTs and handle calls to your OAuth provider service. The good news is that the DARRT library we've been using for our API services has just this kind of code built into the library. The [/darrt/lib/api-auth.js](#) contains some basic code for handing JWTs along with some configuration information you need to provide.

You'll need to supply the [api-auth.js](#) module with the following OAuth parameters you got from Auth0 earlier in this chapter (see [Collecting API Identity Parameters](#)):

- **jwksuri**—The location of your Auth0 signing certificate
- **audience**—The identifier of your access control collection at Auth0
- **issuer**—The domain Auth0 assigned to you when you signed up

You should have this information stored in your [security/auth0.env](#) file. You just need to transfer those three values to the [api-auth.js](#) project file:

ch10-securing/api-auth.js

```

// DARRT Framework
// auth0 support
```

```

// 2020-02-01 : mamund
*****/

// modules
var jwt = require('express-jwt');
var jwks = require('jwks-rsa');
var jwtAuthz = require('express-jwt-authz');
var utils = require('./utils.js');

// auth variables
var auth = {};
auth.cache = true;
auth.rateLimit = true;
auth.requestsPerMinute = 5;
auth.jwksUri = 'https://mamund.auth0.com/.well-known/jwks.json';
auth.audience = 'http://mamund.com/bigco-company';
auth.issuer = 'https://mamund.auth0.com/';
auth.algorithms = ['RS256'];

// auth support
var jwtCheck = jwt({
 secret : jwks.expressJwtSecret({
 cache: auth.cache,
 rateLimit: auth.rateLimit,
 jwksRequestsPerMinute: auth.requestsPerMinute,
 jwksUri: auth.jwksUri
 }),
 audience: auth.audience,
 issuer: auth.issuer,
 algorithms: auth.algorithms
});

// export
module.exports = {
 auth,
 jwtCheck,
 jwtAuthz
};

```

Now that you've updated your [api-auth.js](#) file, you can reference it in your running code to "turn on" security for your API.

## Updating index.js

The next step is to reference the [api-auth.js](#) library in your running code. You need to add this security middleware in NodeJS/Express projects in your [index.js](#) file. Open the [index.js](#) file in the root of the [company-secure](#) project and you should see the following code:

```
//*****
// start of auth support
var secure = require('./darrt/lib/api-auth.js');
app.use(secure.jwtCheck);
// end of auth support

```

This code does two things. First, it pulls the code from the [api-auth.js](#) module into your [index.js](#) file for reference. Second, it registers the [secure.jwtCheck](#) function as part of the Express middleware chain. That means every HTTP request that arrives for this API service will pass through the [jwtCheck](#) module, where that request will be checked for a valid JWT in the headers section of the request. If the token exists and is valid, the request will be processed as usual. If not, the API service will issue a 401 status response reminding the caller that the request wasn't properly authorized.

Here's an example of what that response looks like:

```
mca@mamund-ws:~/company-secure$ curl localhost:8484/
{
 "type": "Error",
 "title": "UnauthorizedError",
 "detail": "No authorization token was found",
 "status": 401
}
mca@mamund-ws:~/company-secure$
```

Now that we have basic API security in place on the service side, let's switch to the client side and see how we can properly request and use a JSON web token to access resources from a secured API service.

## Testing with auth-token.sh and curl-auth.sh

There are two steps to completing a secure M2M request using the OAuth standard. First, you need to acquire a valid JSON Web Token from the provider service. Second, you need to send that token to the API service with each HTTP request. Since JWT strings can be quite long (880+ characters) and fiddly to deal with, I've included a couple of shell scripts in the chapter's code folder that you can use to make these two steps a bit easier to manage.<sup>[91]</sup>

You can use the `auth0-token.sh` script in the project's `/security/` folder to retrieve a new, valid JWT from Auth0. To do that, you need to have filled in your `auth0.env` file with your API's authentication parameters (which we did earlier in this chapter in [Collecting API Identity Parameters](#)). All you need to do is execute the script (no arguments) and it will contact Auth0, pass the proper values, and write a new file to your `/security/` folder called `auth0-token.env`, which contains a JSON response that includes a valid JWT. Here's an example:

```
{"access_token": "eyJhbGciOiJS...","expires_in":86400,"token_type": "Bearer"}
```

That file contains your valid JWT (`access_token`) along with some other metadata (`expires_in` and `token_type`).

Now you need to copy the `access_token` value (just the parts between the quotes) into another file in your `/security/` folder named `curl-auth.env`. That file looks like this:

ch10-securing/curl-auth.txt

```
#####
auth-test variables
#####

url="http://localhost:8484/"
method=GET
```

```
accept="application/json"
contentType="application/json"

token=<JWT-GOES-HERE>"
```

You need to paste the JWT you got from running [auth0-token.sh](#) and insert it into the [curl-auth.env](#) file where you see [`<JWT-GOES-HERE>`](#). When you do that, you'll be able to run the [curl-auth.sh](#) script (which reads the values in [curl-auth.env](#)) and that will make an HTTP request that includes your valid JWT access token. The following shows the results of running the [curl-auth.sh](#) script with a valid JWT:

```
mca@mamund-ws:~/security$./curl-auth.sh

OAuth Request Utility
=====
Sun Apr 26 14:54:57 EDT 2020

...: requesting GET http://localhost:8484/
{
 "home" :
 [
 {
 "id" : "list"
 , "name" : "company"
 , "rel" : "collection company"
 , "href" : "http://localhost:8484/list/"
 }
]
}
mca@mamund-ws:~/security$
```

You can modify the values of the [curl-auth.env](#) script to contain any [URL](#), [method](#), [content-type](#), [accept](#) header, or other values. In this way, you have a handy utility for testing secure API requests.

This wraps up the process of securing your API using the OAuth protocol and the Auth0 provider service.

## What's Next

In this chapter, we covered the basics of web API security, including identity, access control, and encryption. We also learned about the OpenAuth protocol and about JSON Web Tokens. Finally, we covered the steps needed to use the Auth0 third-party OAuth provider to generate and validate JWT access tokens for your web API and how to modify your API code to support secured requests over HTTP.

In the next chapter, we'll pull all this together and deploy our working API to the cloud using the Heroku hosting service.

# Chapter Exercise

For this exercise, you'll define an M2M identity in Auth0 for your [credit-check](#) service and then update your code to support access control using OAuth and JWTs. Along the way you'll use the security bash scripts to request a valid JWT and then use it to make secured requests of your updated [credit-check](#) service.

## Define Your API in Auth0 and Collect Access Control Parameters

First, sign into the Auth0 website and define or create a new API called [bigco-credit-check](#). Then collect the five important access control parameters (Name, ClientID, ClientSecret, Domain, and Identifier) and update your copy of the [auth0.env](#) file in your [/security/](#) folder. (See [Collecting API Identity Parameters](#), for details on how to complete this step.)

Next, use the [auth0-token.sh](#) script to request a valid JWT access token for use in HTTP calls to your API. Copy the token value in the response into the [curl-auth.env](#) file.

Finally, use the <http://jwt.io> website to validate the access token you were issued by the [auth0-token.sh](#) script.

## Update the credit-check-secure NodeJS Project

First, update the project's package collection by adding the proper OAuth packages using npm. See [Adding Security Modules to Your NodeJS project](#), for the list of packages to install.

Next, open the [index.js](#) file in the [credit-check-secure](#) project folder and update that file to reference the [api-auth.js](#) code file from the [/DARRT/lib](#) folder. Add

the security middleware into the NodeJS Express pipeline by adding the following lines to your `index.js` file:

```
//*****
// start of auth support
var secure = require('./darrt/lib/api-auth.js');
app.use(secure.jwtCheck);
// end of auth support

```

Next, open the `/DARRT/lib/api-auth.js` file and update the `auth` object values to match the access control parameters you pulled from the Auth0 website in the previous step.

Be sure to save all your changes to the project and check them into the Git repo.

## Test Your API Security

Now you can try accessing your API to validate your security changes. First, try using a simple `curl http://localhost:8181/` call (without a security token) to confirm that your API call gets an HTTP 401 status code response. Then use the `curl-auth.sh` utility (with the access token from the previous step and other appropriate config settings) to make the same call. This time you should get the root response as expected, without any errors.

You now have a fully secured API service using OAuth and JWT.

See Appendix 2 ([Solution for Chapter 10: Securing APIs](#)) for the suggested solution.

---

### Footnotes

[<sup>76</sup>] <https://tools.ietf.org/html/rfc6749>

[<sup>77</sup>] <https://tools.ietf.org/html/rfc7519>

[<sup>78</sup>] <http://auth0.com>

- [79] <https://oauth.net>
- [80] <https://openid.net>
- [81] <https://openid.net/connect>
- [82] <https://tools.ietf.org/html/rfc8446>
- [83] <https://tools.ietf.org/html/rfc6101>
- [84] <https://www.abetterinternet.org/about>
- [85] <https://letsencrypt.org>
- [86] <http://jwt.io>
- [87] <http://auth0.com>
- [88] <https://auth0.com/signup>
- [89] <https://tools.ietf.org/html/rfc7519>
- [90] <https://jwt.io>
- [91] [https://pragprog.com/titles/maapis/source\\_code](https://pragprog.com/titles/maapis/source_code)

# Chapter 11

## Deploying APIs

---

Now that we've completed our API testing in Chapter 9, [Testing APIs](#), and added OAuth support for security in Chapter 10, [Securing APIs](#), it's time to package, build, and deploy our API project onto a public server where others can use it. Along the way we'll take a look at something called *pipelines* and how they can be used to automate deployment. We'll also delve a little bit into the DevOps world and learn about three levels of deployment automation. Finally, we'll use the Heroku cloud platform to set up our own deployment automation using Heroku's simple command-line tools and a bit of magic from Git to do all the work.

But before we jump into learning how to use Heroku, let's spend a bit of time on the background challenges of deploying API projects to the cloud and how deployment pipelines can make this easier, safer, and more reliable than just doing it all manually.

# The Basics of Deployment Pipelines

One of the most challenging elements to designing and building APIs is the act of deploying the completed work into production. In many organizations, this “release into production” step is so fraught with danger that they have a special set of rituals established to deal with it.

Here are some of the rituals I’ve heard about when talking to companies about their release or deployment processes:

- “We can only release on the weekend, when no one’s in the building.”
- “Before release, we have a meeting with all the department heads at which everyone has to give their ‘go/no-go’ to proceed.”
- “We stop all coding and bug fixes for two weeks while we package up and validate the build for the upcoming release.”
- “Production releases take so much time and energy from the team that we only do it a few times a year.”

As you can see from this short list (I have many more in my files), releasing can be tough. There are lots of reasons for this, most of them historical, but it all boils down to some basic issues:

## *Too much time passes between releases.*

Writing code and building software is a pretty detailed and exacting practice. Small changes can have big impacts on the system you’re working on. And as time passes, every software system adds more data, which affects performance and can result in new, unexpected behaviors. When the code for new features and bug fixes “sits around” for while, the tests used to validate them become less reliable, and that means the release package itself is less likely to succeed.

*Most release packages are too big.*

Like time, size affects release quality. The more changes in a single release, the less likely that release is to succeed. And if the release into production fails for some reason, the more changes in the release, the harder it can be to discover which of those changes (or which combination of those changes) are the cause of the failure.

*There are too many variables to control for each release.*

The number of changes in the release is just one variable in the process. Making sure all the tests were run on the same hardware as production, using the same version of the operating system, and using the same edition of the software framework, libraries, and utilities can be difficult to achieve. Teams often spend hours or days just re-creating a parallel production environment to run tests again before committing the changes to production, and still success is not guaranteed.

# The Role of DevOps

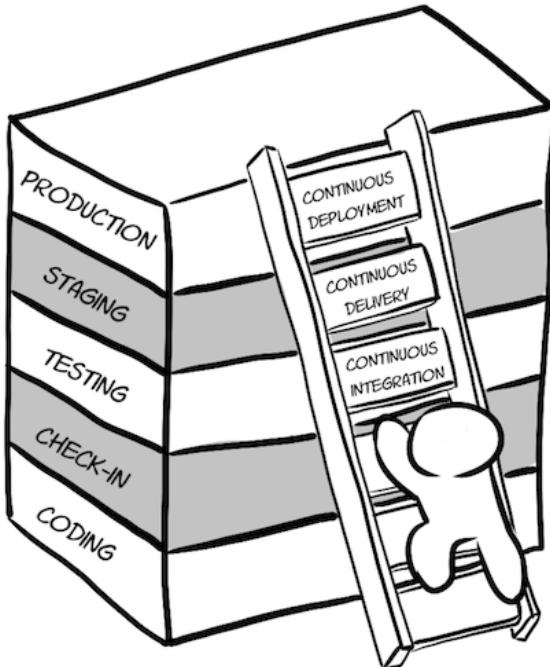
How can companies improve their chances of success when releasing code into production? They tackle the three issues mentioned earlier with one single approach that combines the skills of both software development and IT operations. The name for this approach is DevOps (for development and operations). The history and practice of DevOps goes back to at least 2008, when Patrick Debois presented a talk called “Agile Infrastructure and Operations” in Toronto. [\[92\]](#)

The aim of DevOps is to reduce the time it takes to build and release changes into the system. DevOps does this by encouraging teams to create smaller release packages, release them more often, and automate as much of the process as possible to reduce the chances of variability.

Three practices cover the role of DevOps in deploying your app:

- Continuous integration
- Continuous delivery
- Continuous deployment

They form a kind of ladder of DevOps maturity or advancement. Each rung of that ladder has a set of goals as well as a set of costs or challenges.



## For More DevOps



To learn more about DevOps, I recommend checking out *Continuous Delivery* by Jez Humble and David Farley (<https://www.oreilly.com/library/view/continuous-delivery-reliable/9780321670250>) and *Release It! Design and Deploy Production-Ready Software* by Michael Nygard (<https://pragprog.com/book/mnnee2/release-it-second-edition>).

## Continuous Integration

The first step on the ladder to more successful and reliable deployments is *continuous integration*. At this stage, everyone on the team adopts the practice of checking the code into the repository often. This means merging any changes into the master branch (or main trunk) of the repo. This is the practice I covered in Chapter 1, [Getting Started with API First](#). Those check-ins should kick off automated tests too. We saw how to do that using Postman and Newman in Chapter 9, [Testing APIs](#).

By checking in your changes often and running automated tests for every check-in, you end up validating the project quite a few times and getting immediate success/fail feedback on each small set of changes. This means you catch problems early, when they're easier to fix. Using scripted/automated tests means your tests are more consistent and the results are more reliable.

At this first step, companies still deploy to a final staging server and then on to production using manual processes and checklists. Relying on manual deployment like this has drawbacks, but at least the work of automating building the code package and testing it is greatly diminished, and that can improve the odds of success for the release.

Continuous integration handles the coding, check-in, and testing steps.

## **Continuous Delivery**

The second step on the ladder is *continuous delivery*. At this point, the process of releasing into final staging is automated through scripting. That means deployment is reduced to making some selections (or configurations) and pressing a button. Now, along with scripted testing from continuous integration you also have scripted deployment to the staging level.

By scripting the deployment process, you greatly reduce the variability from one release to the next. The deployment script makes sure the proper baseline hardware and operating systems are used for the build along with all the correct libraries, frameworks, and other software. The script also validates any configuration data, confirms integration and connection to other services, and does a quick set of tests to make sure everything was deployed as expected. Even better, good deployment scripting tools will be able to detect when the deployment did *not* go as planned and will be able to back out all changes and return the system to the last known working version.

That's quite a bit of work to automate, but it's not at all impossible. Lots of products are available today to help DevOps teams get this done. Later in this chapter, we'll look at how to use the deployment toolchain offered by Heroku to accomplish our automated deployment.

At the continuous delivery stage, the deployment is scripted but not *automatic*. Deploying into final production is initiated by someone somewhere pressing the Start button or launching a script.

Continuous delivery handles the coding, check-in, testing, and staging steps.

## Continuous Deployment

The third rung on our DevOps ladder is called *continuous deployment*. At this stage, we're not just scripting testing and staging deployment. We're also making deployment into production *automatic*. That means making the entire process of testing and deploying your app completely driven by scripts and other tooling without the need for a human to "press a button." Typically this is done by setting up your source code check-in process to handle the entire test-and-deploy process.

The test-and-deploy process usually looks like this:

- A developer checks code into the repository.
- That check-in kicks off a series of local tests.
- If the tests pass, the code is built into a release package.
- If the build succeeds, the build is deployed to a staging server.
- If the staging server deployment succeeds, another set of integration tests are run.
- If the integration tests succeed, the build is deployed on a production server.

- If the production deployment succeeds, the job is done.

Of course, at each step there is a possibility of the process failing (failed local tests, failed build, failed staging deployment, and so on). And, in some of the failures, the update needs to be “backed out” or reversed. This is especially true for the production deployment. When that fails, the automated system needs to restore the “old” build into production in order to maintain stability.

That’s quite a bit of work, and a number of tools and services are available to handle it all. Your company may have the tools and processes in place to support continuous deployment. You’ll need to check with your team to learn more about where they are on this ladder of DevOps maturity.

For this chapter, we’ll be aiming for the continuous delivery–level of DevOps maturity. That means we need to script the testing and then make deployment a “one-click” on-demand experience. Since we already saw how to handle automated testing using Postman and Newman in Chapter 9, [Testing APIs](#), we just need to find a way to automate on-demand deployment. And in this chapter, we’ll learn to do that using the capabilities of the Heroku cloud platform.

# Deploying with Heroku

Earlier in this chapter, I described the process of continuous delivery (see [Continuous Delivery](#)). To automate on-demand deployment we need to manage lots of seemingly tedious details, such as making sure the app is deployed using the right hardware platform, the correct version of the host operating system, the proper web framework and library dependencies, and so forth. It can be a very daunting task to get that all correct, especially for developers who don't already know this level of detail on their own systems, let alone the systems running in production.

Luckily, thousands of developers and systems operators before us have boiled the process of continuous delivery down to a stable set of tasks to perform for each deployment. Even better for us, we can take advantage of this accumulated knowledge by using tools and platforms purpose-built for solving this problem. The one we'll be using in this book is the Heroku cloud platform.

## The Heroku Platform

Heroku is a cloud-based Platform-as-a-Service (PaaS). It's designed to host Internet-based applications and has lots of tools and services to make that kind of work safe, easy, and reliable.<sup>[93]</sup> Started in 2007 by a handful of enterprising Ruby language developers, Heroku has transformed into a company that supports most of the major web programming languages and frameworks. In 2010, Heroku was purchased by Salesforce.com and continues to operate as a stand-alone company.

One of the key elements of Heroku's technology is its use of what it calls **dynos**. These **dynos** are small operating-system units based on Linux. They're meant to mimic a full-blown instance of Linux, but they do it in a very cost-effective and resource-efficient way. Each app you deploy on

Heroku runs on one or more **dynos**. Your app is *contained* within these **dynos**, which are easy to build, start, stop, and even duplicate in real time.

Heroku's **dynos** are sometimes just called *containers*. Other popular container-based platforms include Docker, Apache Mesos, and CoreOS. They all work on the same idea: that deployment can be safer and easier if you base your releases on lightweight Linux containers. You can check with your team to see which container technology your company is using (or plans to use) to learn more about how you can take advantage of it for your deployments.

For our purposes here, we'll rely on Heroku's **dynos** system to help automate our deployments. Once we work through this process, you'll have your API project up and running on a public Internet server, ready to accept requests.

## Git-Based Deployment on Heroku

There are a couple of different ways to deploy to the Heroku platform. Since we've been using NodeJS throughout this book, I'll show you a very easy and reliable way to use your Git repository as your deployment package along with Heroku's command-line tool. This will take advantage of a bunch of built-in Heroku deployment tools and do it all from the command-line so that it'll be easy to include in any other scripts you might use to customize your own DevOps deployment process.

One of the reasons I like to use Heroku as my PaaS provider is that it offers a free tier of service for those just starting out. This is a great way to learn about automated deployment, how to integrate your local test and build tools into a fully functional DevOps chain, and how to get familiar with advanced techniques and tools needed to reach the continuous deployment rung of the DevOps maturity ladder.

To make this all work, we need to do a few things:

- Confirm your API project is ready for deployment to Heroku.
- Log into your Heroku account using the Heroku CLI.
- Create a new Heroku “app” that will host your API project.
- Validate your Git repo support for Heroku deployment.
- Run a single `git` command to kick off the Heroku deployment.

It may look like a long list, but once you go through the steps the first time, you just need to repeat the last step each time you want to update your published API project.

Let’s get started.

### Download and Install the Heroku CLI



You’ll need to download and install the free Heroku command-line interface (CLI) in order to continue with this chapter. You can get details on how to do this in Appendix 1, [Installation Hints](#).

## Confirm Your API Project for Deployment

The first step in the process of preparing your API project for Heroku deployment is to make sure your project is being tracked with the Git source control system. We set this up for our Onboarding API project back in Chapter 1, [Getting Started with API First](#). You can confirm this by opening a command-line window, moving to the root of your Onboarding API project, and typing the `git status` command. Your results should look similar to this:

```
mca@mamund-ws:~/onboarding$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

nothing to commit, working directory clean
mca@mamund-ws:~/onboarding$
```

Next, you need to make sure your project's `package.json` file includes a `scripts.start` element that kicks off your server app. We added this element in Chapter 2, [\*Understanding HTTP, REST, and APIs\*](#). If you open your `package.json` file of your `onboardingAPI` project, you should see this:

```
{
 "name": "onboardingAPI",
 "description": "onboarding project for BigCo, Inc.",
 ...
 "scripts": {
 "start": "node index"
 },
 ...
}
```

That `package.json` file will have lots of other content, but it's important that the line `"start": "node index"` is included in the `scripts` element of the file. This is the line the Heroku deployment process will be looking for in order to successfully launch your app once it's installed on the Heroku platform.

You should also make sure your `package.json` file has references to all the other NodeJS modules your API project uses. These are in the `dependencies` section of the file and will look something like this:

```
{
 "name": "onboardingAPI",
 "description": "onboarding project for BigCo, Inc.",
 ...
 "dependencies": {
 "body-parser": "^1.18.3",
 "ejs": "^2.6.1",
 "express": "^4.16.4",
 "express-jwt": "^5.3.1",
 "jsonwebtoken": "^8.5.0",
 "jwks-rsa": "^1.4.0"
 }
}
```

These tell the Heroku deployment service just which module to install (including what versions to use for each of them). You may recall from Chapter 3, [Modeling APIs](#), that we set up our API project to *not* include the dependencies in our build package. That made it possible to keep the package small when checking it into Git and, it now turns out, makes it easier to deploy to platforms like Heroku.

With these two elements of your package file confirmed (start script and dependencies), we're ready to use the Heroku CLI to create our deployment target.

## Log Into Heroku

Before you can start doing work on your apps at Heroku, you need to log into the Heroku system using the Heroku CLI tool. To do that, simply open a command window and enter the command `heroku login`. That will start a process that prompts you for your email and password (your email may be provided as a default). The exchange should look like this:

```
mca@mamund-ws:~/onboarding$ heroku login
heroku: Enter your login credentials
Email [mamund@example.com]:
Password: *****
Logged in as mamund@example.com
mca@mamund-ws:~/onboarding$
```

Once the CLI says you're [Logged in as ...](#) you're ready to start using the command-line tool to remotely manage your server space on the Heroku servers.

## Create a New Heroku App

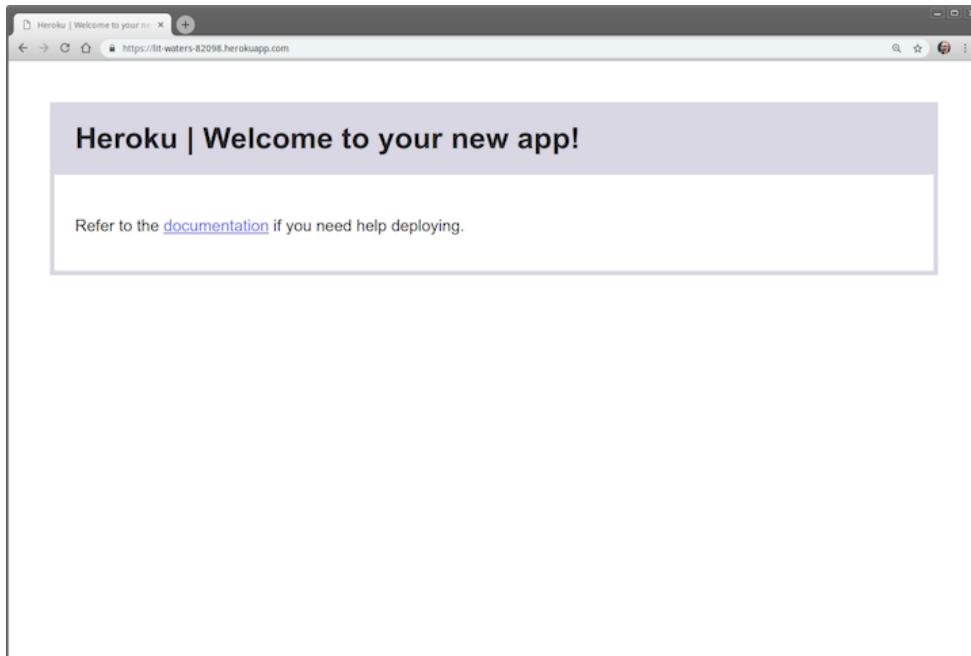
To be able to deploy to the Heroku platform, you first need to create [target](#) or app space. This will be the place your deployment [goes](#) and, if successful, will be the address of the running instance of your API project. Creating a

Heroku app is easy; you just type the command **heroku create**, and the CLI does the rest.

It should look similar to this:

```
mca@mamund-ws:~/onboarding$ heroku create
Creating app... done, lit-waters-82098
https://lit-waters-82098.herokuapp.com/ | https://git.heroku.com/
lit-waters-82098.git
mca@mamund-ws:~/onboarding$
```

What you see here is that Heroku created an app space on its server and named it **lit-waters-82098**. That's the unique name of your Heroku app space. Notice also that Heroku displays two URLs. The first one is the public URL for your running API project. If you visit that link right now, you'll see a default Welcome page displayed, as shown in the following screenshot:



Once we deploy our NodeJs package to Heroku, you'll start seeing the real API project output at this URL.

The second URL is the address of a Git repo on the Heroku platform. This is the URL we'll use when we **push** our build package to Heroku for

processing and deployment. This allows us to use a simple `git` command to start the deployment; Heroku does the rest.

### What's in a Name?

Heroku generates unique names for all its app spaces unless you give it a particular name. I skipped that step since every app name in Heroku needs to be unique, and it would be frustrating if I instructed readers to all use the same name. You can check out the Heroku CLI documentation (<https://devcenter.heroku.com>) to learn more about how to customize your app creation experience.

## Validate Your Git Repo

You can validate our Git repo to confirm it was properly set up by the Heroku CLI to support using `git` as our deployment tool. To do this, you just need to run the command `git remote -v`. The results should look something like this:

```
mca@mamund-ws:~/onboarding$ git remote -v
heroku https://git.heroku.com/lit-waters-82098.git (fetch)
heroku https://git.heroku.com/lit-waters-82098.git (push)
origin git@github.com:mamund/onboarding.git (fetch)
origin git@github.com:mamund/onboarding.git (push)
mca@mamund-ws:~/onboarding$
```

What you see in this output is that, along with the `origin` remote repo address we added when we connected our local repository to GitHub (back in Chapter 2, [Understanding HTTP, REST, and APIs](#)), there is a new repo address. That's the one on the Heroku platform. Note that it's named `heroku`. That's the default name the Heroku command-line app used. You can customize this name too. Check out the Heroku CLI documentation for details.

We now have everything in place to run the first DevOps deployment of our API project.

## Deploy Your API Project to Heroku

Now that we've done all the setup work—confirmed our [package.json](#) file, created our Heroku app, and validated our Git repo—all we need to do to kick off the deployment process is to type one command: [git push heroku master](#). This command uses the [git](#) command-line tool to sync the contents of our local repository with the repository on the Heroku platform. When that happens, Heroku will then start the deployment process that will eventually lead to our Onboarding API project running live on Heroku's servers.

The deployment process takes a little while (especially the first time) and generates quite a bit of output. The following is a listing from one of my deployments. I'll break it up into smaller parts so you can see what's happening at each stage of the process.

First, Git does the work of packaging up the NodeJS project and sending it to the repository on Heroku:

```
mca@mamund-ws:~/onboarding$ git push heroku master
Counting objects: 65, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (35/35), done.
Writing objects: 100% (65/65), 25.93 KiB | 0 bytes/s, done.
Total 65 (delta 26), reused 65 (delta 26)
```

Then Heroku starts the process of creating a container (called a [dyno](#) at Heroku) and loading it with all the operating-system-level elements to support a NodeJS web server:

```
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Node.js app detected
remote:
remote: -----> Creating runtime environment
remote:
remote: NPM_CONFIG_LOGLEVEL=error
```

```
remote: NODE_ENV=production
remote: NODE_MODULES_CACHE=true
remote: NODE_VERBOSE=false
remote:
```

Next, the deployment process figures out which version of NodeJS and npm to install onto the container:

```
remote: -----> Installing binaries
remote: engines.node (package.json): unspecified
remote: engines.npm (package.json): unspecified (use default)
remote:
remote: Resolving node version 10.x...
remote: Downloading and installing node 10.15.3...
remote: Using default npm version: 6.4.1
remote:
```

## Which Version of NodeJS?



Note here that Heroku chose the latest stable edition of NodeJS/npm, since our package doesn't specify a version. We could have done that, if we wished. You can read more about how to do that in Heroku's deployment documentation at <https://devcenter.heroku.com/categories/reference#deployment>.

The next step in the deployment process is to check for any NodeJS modules that need to be installed, add them if necessary, and test them for any security issues:

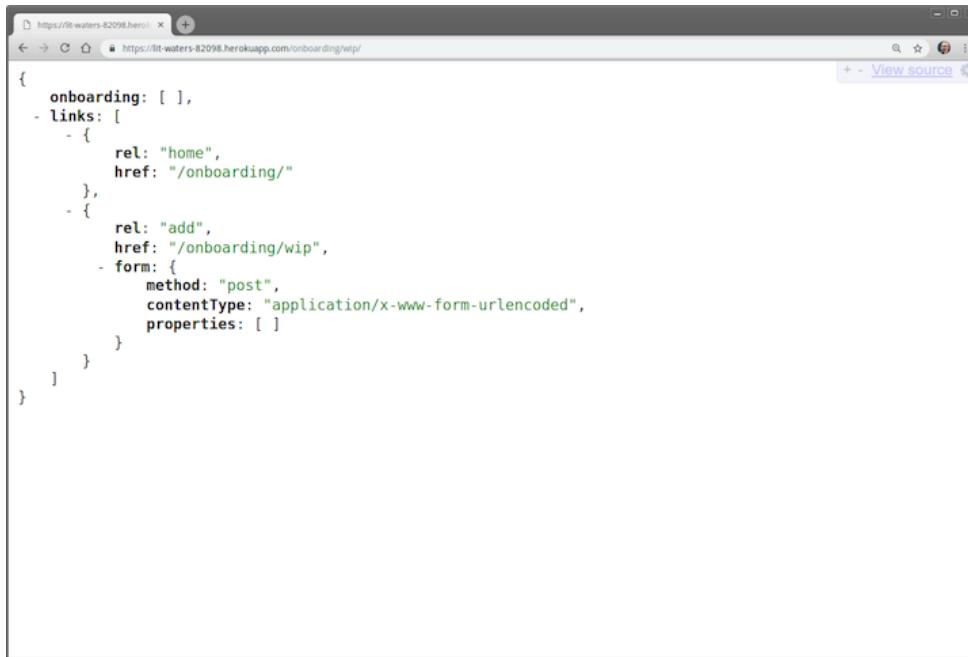
```
remote: -----> Building dependencies
remote: Installing node modules (package.json + package-lock)
remote: added 132 packages from 161 contributors and audited
remote: 293 packages in 3.789s
remote: found 0 vulnerabilities
remote:
remote: -----> Caching build
remote: - node_modules
```

```
remote:
remote: -----> Pruning devDependencies
remote: audited 293 packages in 1.417s
remote: found 0 vulnerabilities
```

Finally, if all the build details succeed, Heroku places the container into production and launches the app:

```
remote:
remote: -----> Build succeeded!
remote: -----> Discovering process types
remote: Procfile declares types -> (none)
remote: Default types for buildpack -> web
remote:
remote: -----> Compressing...
remote: Done: 20M
remote: -----> Launching...
remote: Released v3
remote: https://lit-waters-82098.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/lit-waters-82098.git
 * (new branch) master -> master
mca@mamund-ws:~/onboarding$
```

If everything worked as expected, you should now be able to visit a URL within the Onboarding API project and see a live response. For example, if you visit the first step in the work-in-progress phase of the API, you should see a screen similar to the one shown here:

A screenshot of a web browser window. The address bar shows the URL "https://lit-waters-82098.herokuapp.com/onboarding/wip/". The main content area displays a JSON object with the following structure:

```
{ onboarding: [], links: [- { rel: "home", href: "/onboarding/" }, - { rel: "add", href: "/onboarding/wip", form: { method: "post", contentType: "application/x-www-form-urlencoded", properties: [] } }]}
```

The JSON is displayed in a monospaced font, with colors used for different data types: green for strings and numbers, blue for objects and arrays, and black for the rest of the code.

Of course, your URL will be different since the exact name of your Heroku app space will be unique.

Each time you make any changes to the project and check those into your Git repo (master), you'll also be able to deploy those changes into production on-demand by typing the command **git push heroku master**. You've now reached the continuous delivery stage of DevOps maturity.

## What's Next?

In this chapter, we learned about the importance of using automation and deployment pipelines to meet the challenge of packaging, building, and releasing your API project safely and easily to a public web server. We also took a dive into the world of DevOps to understand how to use these three DevOps practices:

- *Continuous integration* to make sure our code is stable and tested often.
- *Continuous delivery* to make the process of building and releasing our projects safe and consistent.
- *Continuous deployment* to automate the entire process of testing, building, and releasing into production.

Along the way we learned how to use the Heroku cloud platform to implement level two of our DevOps maturity ladder by supporting continuous delivery using the Git command-line tool.

Now that we've completed a full cycle of design, build, and deploy, we have one more critical situation in the life of an API to deal with: how to handle changes in the API once it's been released to production. We'll handle that in the final two chapters.

# Chapter Exercise

In this exercise, you'll get a chance to expand the functionality of `newman`, the automated testing app we used in Chapter 9, *Testing APIs*. This time, along with using the `newman` library to test our package, you'll get a chance to add an additional step that will automatically deploy your package to the Heroku cloud platform.

In this exercise you'll need to update the `norman` utility to handle continuous deployment. There's a copy of the source code for this utility in the `utilities` folder of the downloadable source code associated with this book.<sup>[94]</sup>

To update the utility, you'll need to go to the `code/deploying/exercise/before/norman` folder and open the `index.js` file to locate the commented section with the title `add deployment command here`. Modify the `shell.exec(...)` command on the subsequent line to include the correct `git` command needed to fire off Heroku's continuous delivery process. (You should need just four keywords on a single line to make this work.)

Once you have the code updated, run the same command as before (`norman assets/api-onboarding,postman_collection.json`). If all goes well, you should see your project pushed into production again on the Heroku platform.

See Appendix 2 ([\*Solution for Chapter 11: Deploying APIs\*](#)) for the solution.

---

## Footnotes

[92] <http://www.jedi.be/blog/2008/10/09/agile-2008-toronto-agile-infrastructure-and-operations-presentation>

[93] <https://www.heroku.com>

[94] [https://pragprog.com/titles/maapis/source\\_code](https://pragprog.com/titles/maapis/source_code)

Copyright © 2020, The Pragmatic Bookshelf.

# Chapter 12

## Modifying APIs

---

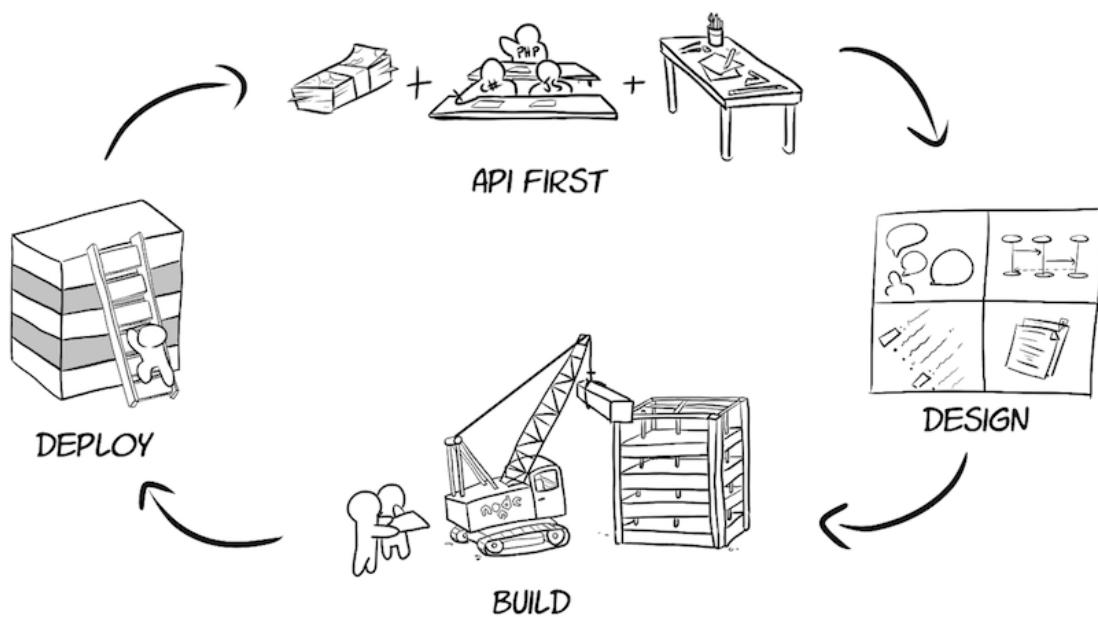
Things seems to be at a happy end. We've designed, built, and deployed our API into production and, thanks to a solid and well-executed design experience and a solid build process, our API is not languishing on some list of available services; it has active signups and is frequently used. That's great! However, now that the API is in heavy rotation in several parts of our fictional company, BigCo, Inc., a new problem has emerged.

We're getting requests for added features.

One of the challenges of a successful API is that people start thinking of new and creative ways to use it. And that almost always results in requests for modifications. Modifying the API is actually an important part of the overall lifecycle. In fact, the longest portion of an API's life is spent in production. And the longer an API is in use, the more likely it is you'll be asked to change it in some way.

In this chapter, we'll explore the key steps to successfully implementing changes to your APIs after they're released into production. Not surprisingly, these are the same steps we employed when creating the original APIs. First, model and design the requested changes and produce a machine-readable ALPS document. Second, employ the sketch, prototype, and build process to explore, confirm, and ultimately build your updated API. Last, use testing and security checks to validate your implementation before finally deploying the modified interface into production.

However, along the way we'll cover some principles for dealing with changes to the existing production API safely, including the principle of "First, do no harm." We'll also explore the three rules of designing and implementing backward-compatible API updates. We'll see how you can use a simple "backward-compatibility" test to determine the safest way to deploy your API changes. Finally, we'll cover the right way to shut down an API and take it out of production permanently.



# Going Beyond Versioning

Change is an inevitable part of APIs. It's pretty unlikely that you'll design, build, and deploy an API that's "just right" and never needs to be updated. In fact, it's a sign of a good API that people want to use it more often and come up with ways to make it even better. Popular APIs generate requests for changes. For that reason, we need to make changing APIs a regular part of our API practice.

The most common technique for handling API changes is to simply release a new, updated API in place of the old one. This is usually called "versioning your API." It's the most common, but it's not the safest. If you're not careful, you can release a new API that's incompatible with existing uses. When this happens you are "breaking" current API clients—and that's a bad thing.

In the next section, we'll look at a few principles of safe and effective API changes:

- First, do no harm.
- Fork your API.
- Know when to say no.

These principles lay the groundwork for the techniques I recommend for properly changing APIs and for safely deploying these changes into production.

## First, Do No Harm

The phrase "First, do no harm" comes from the Hippocratic oath.<sup>[95]</sup> Dating back to the 4th century BCE, the Hippocratic oath has been taken by many physicians as a pledge to give proper care to patients. This phrase is just one small part of the oath, but it's the best-known part. I like to use the same sentiment when contemplating changes to a production API.

Once your API is in production, it takes on a life of its own. When other applications are using your API, *those* applications become dependent on your API. That means you, as the author of the API, have a responsibility to all your API users. And one of the very first responsibilities is to “do no harm.” If you make a change to a running API that breaks one of your API consumers, you are not following the Hippocratic oath of APIs.

So, whatever you do, you need to be sure your changes won’t break an existing API consumer. One of the best ways to do this is to continue to write and perform extensive API tests, like the ones we looked at in Chapter 9, [Testing APIs](#).

## Forking Your API

Sometimes, however, you’ll run into a situation where you must make a breaking change to a production API. For example, you might release an API that allows customers to update their personal information on your servers, and a year later the company decides that it will no longer allow customers to edit that data directly. When this happens, what you need to do is release a *new* API with the proper functionality and instruct all users of the existing API to update their applications to point to the new interface. While that happens, you need some time when both APIs are available in production.

This process is called “forking your API.” You create a new API based on the existing one and run both in production until everyone has updated their client applications. Along the way, you can do a few things to make it easy to migrate from the old release to the new one. For example, you can monitor the usage of the old release and identify key API users that may need assistance in upgrading to your newest release. You may even be able to provide “migration kits”—utilities and guides to help API consumers move from their current release to the more recent one. Finally, once all traffic to the old API has stopped, you can “turn off” the old API

completely. We'll cover more about the process of shutting down API releases later in the book (see [\*Shutting Down an Existing API\*](#)).

## Knowing When to Say No

Finally, it's important to know when to say no to requests for API changes. APIs take time and resources to build. And once released, they often take some time to "settle down." You need to work out small bugs, people need to learn how to use the API, and operations may need to learn how to properly scale and manage the API in production. Once these elements are worked out, the API can run without much attention, providing benefits to everyone.

But as soon as you change the API, the whole cycle of design, build, and deploy starts up again, as does the predictable instability period you may experience when you release an update. As long as you keep changing the API, you're likely to see some level of instability. Multiply these occasional periods of disruption by the 100s or 1,000s of APIs in your organization's ecosystem, and you may be looking at a continual level of instability that can be quite frustrating.

For that reason, it's important to know when you can just say no to API change requests. A good way to measure the potential value of an API change is to refer back to your initial API story and the early modeling work you completed for the initial API (see Chapter 3, [\*Modeling APIs\*](#)). Then ask yourself (and your team), "Will this new change help meet the initial purpose of the API design?" or "Do the proposed changes fit into the initial models used to create this API?" Sometimes the answer is no.

Instead of modifying an existing API, it may make more sense to create a new API that better matches the described need. If the new API eventually supersedes an existing API, you can close down the old one at some future date too.

Now that we have some basic principles for deciding whether to make a change to a production API, let's go over some rules for how to implement these changes and a decision-making process for how to package and deploy those changes into production.

# The Three Rules for Safely Modifying APIs

As you've learned throughout the book, APIs are a collection of interface rules and promises. We design the interface to have certain features, like URLs, HTTP methods, arguments, return formats, and so forth, and then publish those interface rules as ALPS profiles, OpenAPI documents, and readable documentation. It's this material that other developers will rely on when they want to use our API. That collection of information is our promise to API developers.

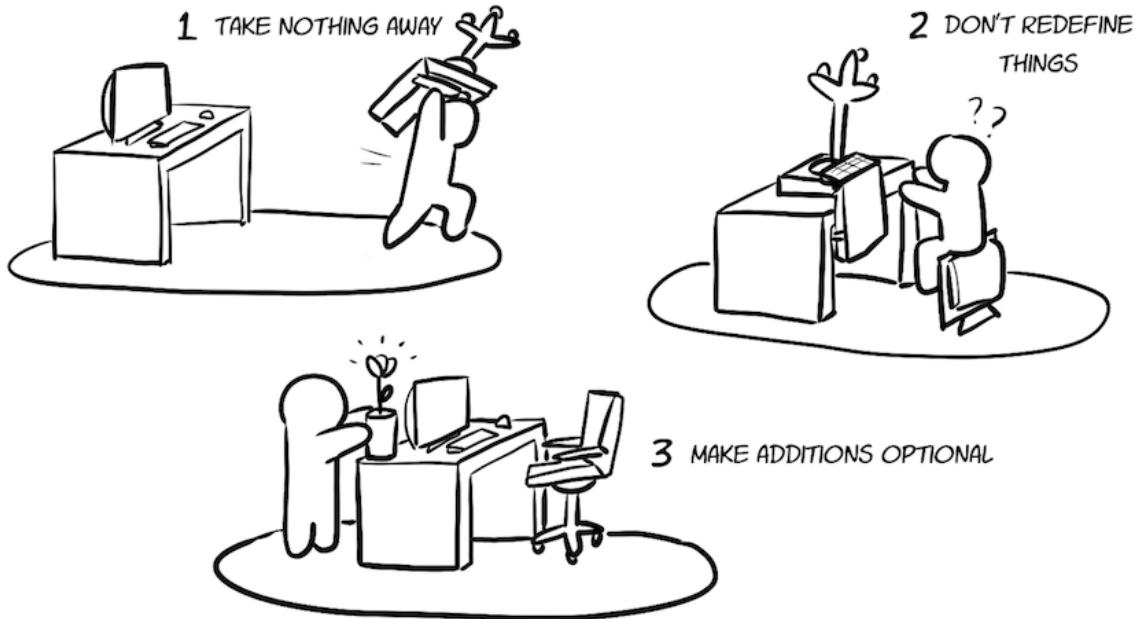
Once the API is released, we can't break our promise to developers.

Two main areas need attention when updating an existing production API. The first is how you actually implement the change in the API service itself. This means how you handle the *interface* as well as the *code*. Once that work is done, you need to deal with how to *deploy* the change. In this section, we'll focus on the interface and code side. We'll deal with the deployment details in the next section (see [The Process for Safely Deploying APIs](#)).

When modifying the API itself, we need to keep the "First, do no harm" principle in mind. We should not make any changes that cause existing API consumers any harm. That means no "breakage." This principle applies specifically to the interface itself. But we also need to make sure we don't change the code in a way that causes API consumers to see errors or "breaks" when they use our APIs.

From the interface point of view, you can rely on these three handy rules when contemplating changes to the API:

1. Take nothing away.
2. Don't redefine things.
3. Make additions optional.



When you follow these three rules, you greatly reduce the chances that your modifications will break API clients. Let's take a look at each one in detail.

## Take Nothing Away

One way we can make sure not to break our promise to API developers is to make sure any future modifications to our API don't result in us taking anything away. Taking things away has the potential to make some other API developer's application break or not work as expected. To use an analogy, taking something away from your API interface is like taking the chair away from your work desk. It makes the whole experience of working at your desk unpleasant at best and maybe even impossible. The same is true for taking things away from your API.

For example, if our existing API has an endpoint URL of `/customers/filter`, we can't remove that URL in the next release. Or if one of our query parameters for `customers/filter` is `lastName`, we can't rename it to `familyName` in the next release. And we can't remove or rename a response property or action, either. Each subsequent release must contain all the things in the previous release if we want to do our best to prevent "breakage" for existing API consumer applications.

You might think you can safely change a few of the interface details of your API without causing API consumer problems, but that's often not the case. This was pointed out by Google's Hyrum Wright when he coined what would later be known as "Hyrum's Law".<sup>[96]</sup>

With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody.

Even when you document that some aspects of your API might change over time, you should assume that someone, somewhere has written an API consumer app that *depends* on that element of your API, and that taking it away will break that API consumer application.

Then what's your alternative? That's easy—you can *add* as many things as you wish in future releases. For example, if the new release has a much improved filtering resource (say, one that supports using HTTP POST to send complex queries), we just *add* that as a new endpoint to our interface (maybe as `customers/complexQueries`). We can *add* a new parameter (`familyName`) to the existing `customers/filter` too.

Adding new endpoints, query parameters, and response properties is the best way to keep from breaking existing API consumer applications. It's not foolproof, but it's the best way forward.

## Don't Redefine Things

Another important rule to follow when updating your interfaces is to avoid redefining existing endpoints, actions, input parameters, or response properties.

For example, if you have a query that looks like this...

```
http://api.example.org/products?size=5
```

...where the numeral **5** represents the product size in inches, you can't change the API or service to interpret that **5** to represent centimeters. This goes for all sorts of measurements and amounts, including currency and time. For example, switching time references from local time to Greenwich Mean Time is a breaking change.

Just as in the case of "Take nothing away," the solution for not redefining things is to *add* new properties and parameters to represent the new values you want to add to your interface. Usually, the nonbreaking way to do this is to support an additional parameter that adds context to the first one.

For example, if our API uses <http://api.example.org/products?size=5> and the documentation explains that **size** is expressed in inches, you can add a new parameter (**units**) that allows client apps to modify the default unit from inches to something else, like centimeters. Now your API can support the following:

```
http://api.example.org/products?size=13&units=centimeters
```

Note that it's important to keep the interface behavior backward-compatible in the new release. That means that sending [http://api.example.org/products?](http://api.example.org/products?size=5) **size=5** continues to default to inches. Of course, you can also make sure your interface supports this explicitly, as in: [http://api.example.org/products?](http://api.example.org/products?size=5&units=inches) **size=5&units=inches**.

In cases where you want to redefine the meaning of existing properties or parameters, the correct approach is to continue to support the existing meaning and *add* a new context element that can optionally be used to modify the meaning of an existing element in the interface.

## Make Additions Optional

The third rule of modifying your API is to make sure all new elements (endpoints, parameters, actions, and properties) are *optional*. That means making sure that existing API client applications can continue to use the

same actions and sequences, send the same properties, and expect the same responses, even when you add new things to the release.

For example, if your current API defines a write action like this:

```
{
 "id" : "create",
 "name" : "create",
 "href" : "http://localhost:8181/",
 "rel" : "create-form api",
 "tags" : "collection api list",
 "title" : "Create User",
 "method" : "POST",
 "properties" : [
 {"name" : "id","value" : "1vdb1pyn5sf", required:"true"},
 {"name" : "givenName","value" : "", required:"false"},
 {"name" : "familyName","value" : "", required:"false"},
 {"name" : "telephone","value" : "", required:"false"},
 {"name" : "email","value" : "", required:"true"},
 {"name" : "status","value" : "pending", required:"true"}
]
}
```

You can't add a new field to the list of write properties (for example, **postalCode**) and mark that new field **required**. You can, however, add the new field as optional:

```
{
 "id" : "create",
 "name" : "create",
 "href" : "http://localhost:8181/",
 "rel" : "create-form api",
 "tags" : "collection api list",
 "title" : "Create User",
 "method" : "POST",
 "properties" : [
 {"name" : "id","value" : "1vdb1pyn5sf", "required":"true"},
 {"name" : "givenName","value" : "", "required":"false"},
 {"name" : "familyName","value" : "", "required":"false"},
 {"name" : "telephone","value" : "", "required":"false"},
 {"name" : "email","value" : "", "required":"true"},
 {"name" : "status","value" : "pending", "required":"true"}
]
}
```

```
 {"name" : "status", "value" : "pending", "required": "true"}
 {"name" : "postalCode", "value" : "00000-0000", "required": "false"}
]
}
```

Notice in the above example that the new field is not only marked **required:false**, it also has a supplied default value (**00000-0000**). This is a good practice when adding new input fields to your API. You should also make sure the server-side code that accepts the inputs for this action checks for the optional field and, if missing, supplies the default value anyway.

You may recall that this ability to supply missing default values is a built-in feature of the DARRT framework we explored earlier in the book (see [Defaults](#)). Be sure you have a similar feature in whatever coding framework you're using to build your APIs.

# The Recommended Pattern for Testing for API Changes

Applying the three rules for modifying APIs starts at the design phase of your API update. It's important you take the time to review all the requested changes before moving forward to make your API modifications. Then it's important to implement those design changes faithfully. You also need to validate that your changes won't break existing API consumers.

Validating API changes involves a few steps:

1. Asking yourself, “Who is using your API?”
2. Validating your API changes with existing tests
3. Confirming your API changes with new, stand-alone tests

## Who Is Using Your API?

This can be a tricky business since you don't always have access to all the API consumer apps or the teams that build them. Within a single organization, you might be able to inspect all the API consumer source code and check for problems.

But if your API is used by external partners, you may only be able to tell them to do their own inspections to confirm your changes are safe. Also, in cases where your API is available on the open web, you may not even know who's using the API and won't be able to contact them directly.

*Don't rely too much on knowing all your API consumers and their code.*

Even in small organizations, it's possible that some person or team is using your API and you don't know about it. Instead, you need a more reliable approach to validating your changes. And that means API testing.

## Validating Your API Changes with Existing Tests

What you can do is test the updated API yourself. And the best way to do that is to subject your modified API to the same tests you used for the previous release. In other words, you need to make sure your modified API passes all the existing tests you already created for this API. If you have a well-written set of API tests—ones that include both happy- and sad-path tests—you can have a high degree of confidence that any modified API that passes all the tests won’t break existing API client applications.

The idea of using existing tests for the updated API relies on the same thinking as the second rule about modifying the interface (see [Don't Redefine Things](#)). Essentially, you don’t want to “redefine” the test suite. By running your existing tests against your modified API, you’re more likely to discover unexpected changes to the interface or the response behaviors than if you just wrote all new tests for the new, changed API.

If possible, it’s also a good idea to invite existing API client applications to point their client to the new API (running in a beta or test mode) and ask them to confirm that their applications still work as expected. This doesn’t guarantee that your changes are all nonbreaking, but it gives you additional feedback as you work through your modification lifecycle toward releasing the new API in production.

*Use your existing tests to validate that your design and implementation changes don’t introduce breaking changes for any existing API consumers.*

## **Confirming Your API Changes with New, Stand-Alone Tests**

Now that you’re using existing tests to validate that your changed API isn’t going to break current API consumers, your next step is to write additional tests for the features added to your modified API. It’s important to write separate tests for the new edition of the API in order to isolate the changes and confirm they all work as expected.

That means you need to write both happy-path and sad-path tests for the changes. You need to write tests for the interface (URLs, method, parameters, and responses) as well as for the behavior (default values, returned properties, new workflows, and so on). Stand-alone tests are your best bet for confirming that the new features were implemented as expected.

It's also important your new stand-alone tests can be executed *without* running the existing tests. For example, if your new design calls for adding optional fields to write actions, you need to write complete happy- and sad-path tests for that write action that takes into account the new fields. Don't just write a test to see if the new field appears in the response. Make sure the behavior associated with the optional field (including server-side defaults as described in [\*Make Additions Optional\*](#)) is also covered as expected.

By the way, each time you release an update to your API, you need to write new stand-alone tests for that update. And you need to run all the previous tests for all the previous releases of your API. As you can guess, as the API changes collect over time, you'll also collect a larger suite of tests. This is a feature, not a bug! Resist the temptation to consolidate all these layers of tests into a single rewrite. Especially with APIs that have undergone extensive changes over time, you may find it difficult to convert the accumulated test suite into one set of comprehensive assertions. And even if you successfully do that, you won't necessarily increase the quality of your tests. You'll just reduce their quantity.

*Write stand-alone tests to validate your API changes without altering any of the existing tests.*

Okay, now that we've covered the rules for both the design/build and the testing of API changes, it's time to talk about the rules for releasing these modified APIs into production.

# The Process for Safely Deploying APIs

When you get to the point where you plan to release your API updates, you have one more opportunity to check for backward-compatibility issues and confirm that you've done all you can to eliminate "breakage" for your API consumer clients. That means you need to figure out the best way to *deploy* your updates into production. To cover all these details, we'll need to focus on a number of things:

- Supporting reversibility
- Using side-by-side deployments
- Automatically routing traffic vs. waiting for clients
- Overwriting old API releases

That's a lot of things to account for, so let's dig in.

## Supporting Reversability

Ironically, the first thing that you need to do when implementing your API update deployment system is make sure you can quickly and easily take your API out of production. In other words, you need to be able to back out any API update and revert back to the way things were before you attempted the new deployment.

This *reversibility* is your safety net in case something goes wrong during deployment.

In Chapter 11, [Deploying APIs](#), you learned the value of scripted deployments. You can consistently run (and rerun) your deployment script knowing that, each time you do, the process is the same. The same is true for reversing your deployment. As soon as you detect a problem with your install, you should be able to instantly undo your latest update.

**Don't Forget Your Data!**

## Don't Forget Your Data!



Deploying and reversing APIs can be complicated if your change involves adding new data properties to your storage model. If you deploy a new release that starts collecting new data properties and then need to reverse that release, you might end up losing the added data you were collecting with the new release. To avoid this, you'll need to include data capture and recovery as part of your reversibility plans.

The easiest way to do that is to simply rerun the script from your *previous* deployment. This would potentially overwrite your mistakes and set you back to “normal” operating conditions. But there’s another pattern you can use that can be easier to implement and can result in a safer, more stable deployment process: side-by-side deployment.

## Using Side-By-Side Deployment

The best approach for releasing updates to existing production APIs is to use a “side-by-side” release pattern. That means both the new and the existing editions of your API are running production at the same time. Side-by-side releases allow you to gradually roll out your changes with the least impact on existing services and API consumers.

First, you build, test, and—assuming everything passes—deploy your updated API into production. You don’t need anyone to actually be using the service yet—just get it up and running. If you have any problems with the deployment, you can remove the updated service from production without any disruption.

If the install goes fine, you can run your test suite against the updated API to confirm all the behaviors operate as expected. Once you know the install was successful and all tests pass, you’re ready to open up your updated API for production traffic.

## Automatically Routing Traffic to the New Release

As long as your update has followed the three rules we discussed earlier (see [The Three Rules for Safely Modifying APIs](#)), you can arrange to automatically route production traffic from API consumers to your new release. A common way to do this is to use an API gateway rule to select a random segment of API requests and route those to your new release. Many API gateways include this as an option, or it's relatively easy to add such a rule to the routing tables.

### Don't Get Left in the Dark



A whole set of tools and practices has been built around the ability to release a new component silently and then automatically route traffic to it over time. Two common names used in this space are “dark release” and “feature flags.” There are lots of sources of information and even cloud-based platforms that you can use to implement these patterns within your own company.

## Waiting for Clients to Update to the New Release

In side-by-side release cases, where there's no master API gateway controlling all the API traffic, the API provider will need to wait on API consumers to update their code (or configuration) from the “old” release to the “new” release. This is a common scenario in the API space today.

Keep a couple of things in mind in this scenario. First, you need to be sure that running both releases—say release 1 (R1) and release 2 (R2)—at the same time does not cause problems. That means testing R1 and R2 simultaneously within the same API ecosystem. Second, you need to validate that reverting from R2 to R1 doesn't create problems (see [Supporting Reversibility](#)). This is especially important because you may have cases where an API consumer updates their own code to use the API

provider's R2 but then discovers a bug in their own code and decides to revert to the API consumer code that uses R1 again. API providers will need to support this kind of behavior by API consumers.

Finally, in this "waiting" scenario, API providers should prepare to support "old" releases for quite some time—possibly forever. Your API R2 might add some really cool new features that are simply not needed by some of your R1 consumers. They'll just choose to say no to the time and effort to update from R1 to R2. As your API consumer base grows larger, this will become more likely too.

It's also important to point out that API providers will need to be ready for some API consumers to want to "leap" from R1 to R5 or something like that. They might even want to reverse from R5 back to R1. These kinds of updates become really tricky to test and validate; and in the end, some API consumers may simply make mistakes and rely on API providers to help them sort it all out. Be prepared with backup and reversing techniques that will assist API consumers in these situations.

## The Problem with Overwriting Releases

You can see from these examples that while they're attractive for API providers, side-by-side releases can cause their own challenges. Each added release creates a larger "footprint" that API consumers need to navigate. Eventually you can have so many active releases that your system becomes essentially nondeterminate ("I have no idea what will happen if you try to move from R1 to R13!").

One sure way to prevent the ever-growing footprint problem is to simply *overwrite* your releases. When it's time for R2, simply place R2 into production at the exact same routing address as R1 and remove R1 from the system completely. API consumers will have no say in the matter. They'll be forced to start using R2 without any extensive testing and will have no option to revert to R1 if they run into problems. This "we're doing this

“whether you like it or not” approach is probably the most common deployment pattern in the API space today.

You can soften the blow to your API consumers by announcing your intention to kill off R1 at some fixed future date (“six months from now...”) and providing a “beta” endpoint where you can mount a test edition of the R2 update for consumers to use as their own test target while they work to update their own code to work with your R2. But eventually you’ll need to make a single “flip of the switch” to move from R1 to R2 globally. Of course, you’ll still need to be ready to reverse the release if things go badly.

## When Overwriting a Release Is Acceptable

Sometimes overwriting your current release is acceptable. In fact, this “quiet update” happens on the web all the time. The most common example is with HTML-based websites. It’s rare to see a website that tells users, “We are releasing an update soon. Please change your URL to point to our new site.” Instead, websites simply make changes to URLs, page layouts, forms, and links and the HTML web browsers handle it all just fine. True, sometimes humans get frustrated when they can’t find their favorite content anymore, but that’s a different problem.

The quiet update pattern works for websites because these sites rely on HTML—a highly structured message format that supports moving page content around, modifying links and forms, and so forth, without breaking the client application (the browser). HTML and the HTML client make it possible to update details within the server responses that won’t break client apps. We discussed an API style based on this approach early in the book (see [\*The Style of REST\*](#), for details).

You can duplicate this support for “quiet updates” in your own APIs by adopting the use of highly structured formats for your API responses. For example, the default message formats for the `account` and `company` services both rely on highly structured formats, but the `activity` format doesn’t. As the

structure of the message goes up, so does the ability to support “quiet updates” without breakage. This is a good reason to consider structured formats as your standard API responses.

But a hypermedia format as a response is only one of the things needed in order to safely overwrite existing releases. The other key element is to follow the three rules (see [\*The Three Rules for Safely Modifying APIs\*](#)) we discussed earlier in this chapter. By following the principles of take nothing away, don’t redefine things, and make additions optional, you can greatly reduce the chances that your API modifications will break existing client applications.

# Shutting Down an Existing API

As you create more and more APIs and deploy updates for the ones you have in production, eventually you'll run into the need to shut down one or more of them. The process of "turning off" your API means addressing several important issues in order to ensure data integrity and operational safety, including these:

- Communicating the API shutdown
- Placing code in the public domain
- Publishing the interface design as open source
- Making consumer data recoverable
- Marking your API 410 GONE

Let's briefly look at each one of these issues.

## Communicating the API Shutdown

Before you shut things down, you need to make sure you've notified all the API consumers you can. For internal APIs, this is often done through company-wide email lists, wiki pages, or other shared communications. For externally facing APIs, you'll need to send emails and/or update other public spaces (for example, support forums).

In these communications, you should indicate (1) why the API is closing down, (2) when the API will close, and (3) what alternatives are available for API consumers who may still want to access their data and/or perform similar work. It's this last point that deserves a bit of attention.

When you publish an API, you're essentially inviting others to depend on you for a service. When you shut off an API, you're basically telling your API consumers you'll no longer provide the thing they've grown to depend on. In many cases this isn't a big deal. The service may no longer be needed, or lots of other providers may be doing just as good as or better

than you were, or (as in the regulated examples) it may no longer be possible to do that work at all. However, there are also times when the API you are about to close is still valuable to some people. These folks will be left out in the cold when you shut down your API. In those cases, you should do your best to provide some options that allow them to continue their work.

## **Placing Code in the Public Domain**

In the content publishing business, one option is to place the content in the public domain—to just release a final copy of the work free and available to all. The API equivalent of this is to mount a free server somewhere using the latest instance of your code. You could help start an open-source software project to continue to maintain the code too.

## **Publishing the Interface Design as Open Source**

In cases where giving away all the code isn't an option, you can publish open source copies of the design (API story and diagrams), description (ALPS), and definition (OpenAPI) documents for the API. It can help to publish the test suites (Postman files and SRTs) that validate the interface and API behaviors. With this information, interested parties can create their own services that match the published API specifications.

## **Making Consumer Data Recoverable**

Whenever possible, you should allow API consumers to recover a copy of the data they were storing with the API. For many APIs, there isn't customer-specific data to recover and, in that case, if possible, you should publish a fixed set of data for anyone else who wants to host or build their own edition of the API.

## **Marking Your API as 410 GONE**

The final step when you shut down your API is to mark it “GONE.” This is a formal step in the HTTP protocol that can help others who never got the

communication regarding your plans to shut down the API. Each published endpoint for the closed API should return an HTTP status code of **410 GONE** along with a text message telling API callers that the API is shut down and, whenever possible, where they can find alternative services.

I do this using a simple HTTP 410 status response with the [application/problem+json](#) message body.<sup>[97]</sup> That body typically contains the information API callers need to sort out the situation. Here's a typical example:

```
HTTP/1.1 410 Gone
Content-Type: application/problem+json
Link: <http://example.org/hatsizing/help.html>; rel=help

{
 "type": "https://example.com/probs/gone",
 "title": "This resource no longer exists.",
 "detail": "The Hat Sizing service was shut down on
 1999-04-01 due to lack of funding. Check out
 http://example.org/hatsizing/help.html for alternatives",
 "instance": "/sizing/hombergs/"
}
```

Notice that the response includes information in the **detail** element to help the caller find alternatives for the service that's no longer supported. Also, I provide a [help](#) link header in case some client applications ignore the response body.

As you can see, deploying API updates and dealing with the eventual shutdown of old APIs requires paying attention to a handful of things.

# What's Next?

In this chapter, we explored the challenges of modifying APIs once they're placed into production. We reviewed the principles that govern changes to production APIs: first, do no harm; forking; and knowing when to say no.

We also covered the three rules of designing and implementing API updates: take nothing away, don't redefine things, and make additions optional. We also discussed the importance of continuing to use existing tests to confirm that your updates don't break any current API consumers, as well as discussed new, stand-alone tests for the added features of your updated API.

We spent time on the set of tasks needed to properly deploy API updates into production, including supporting reversibility, using side-by-side deployment, and the challenges of employing the overwriting release tactic. Finally, we reviewed the issues you need to address when shutting down a production API, such as communicating the shutdown, releasing the code and/or interface for others to use, and marking your API with HTTP 410 GONE and a response message to ensure others who encounter the original service in the future will know what happened.

And with that review, we've come full circle in our API journey. At this point, you should be armed with the skills and tools required to design and build the APIs you and your company need in order to meet your business's and customer's expectations. In the next chapter, I have a few parting thoughts on how you can take the next steps in process designing and building great web APIs.

---

## Footnotes

[95] [https://en.wikipedia.org/wiki/Hippocratic\\_Oath](https://en.wikipedia.org/wiki/Hippocratic_Oath)

[96] <https://www.hyrumslaw.com>

[97] <https://tools.ietf.org/html/rfc7807>

Copyright © 2020, The Pragmatic Bookshelf.

# Chapter 13

## Some Parting Thoughts

---

We've come to the end of the book. But really, we've come to the beginning. As we learned from Donald Norman, we're all engaged in our own Action Lifecycle, constantly looping between our goals and the world, crossing the bridges of execution and evaluation. Just like the web itself, we're living out our lives in a repeating request-parse-wait loop and, at each cycle, we get to learn something new.

I hope this book has been a positive learning experience for you; it certainly has been for me. The work of organizing my accumulated experiences and practices into a single volume turned out to be a bigger project than expected and, at times, seemed to take on a life of its own. The resulting content between these pages covers quite a bit of ground. So much that it's worth it to briefly touch on the highlights before we close the cover (virtually?) for the last time.

# Getting Started

We started the book by visiting some important foundational concepts that can help you make sense of the way web APIs look and feel, along with how you can use current technology to create them. In Chapter 1, [Getting Started with API First](#), we saw that APIs are meant to solve real business problems. Creating APIs just to say, “We have APIs,” isn’t enough. It’s important, as Kas Thomas says, to determine what your users “expect to be able to do with APIs.”

We also looked at the basic technology behind web APIs in Chapter 2, [Understanding HTTP, REST, and APIs](#). The HTTP protocol defines how clients and servers communicate on the wire. The common practice of the web, as outlined by Tim Berners-Lee and others, sets the tone for how we should design APIs that live on the web. And Roy Fielding’s REST style establishes a repeatable set of constraints we can rely on when translating our API’s design into a fully working implementation.

The work of using protocols, practices, and a consistent style to solve real business problems for a target audience makes up the baseline by which you can judge all the APIs you plan to design and build going forward.

## The Design Phase

In the design phase, we explored the processes of capturing the essence of the problem you need to solve and translating that into a consistent design ready for developers to use. In Chapter 3, [Modeling APIs](#), you saw the power of Donald Norman’s Action Lifecycle and the importance of writing your API story and documenting the API workflow that captures the essence of the business problem. We covered the actual design process of identifying and detailing all the actions and properties of your API interface in Chapter 4, [Designing APIs](#), and learned how to document these API properties in a machine-readable, technology-agnostic way with the ALPS description format in Chapter 5, [Describing APIs](#).

Above all you learned that properly detailing and describing your API design sets the stage for a solid implementation that will meet the needs of both your business and your developer community.

## The Build Phase

In the build phase, you learned to apply Frank Gehry's sketching technique to APIs in Chapter 6, [\*Sketching APIs\*](#). In Chapter 7, [\*Prototyping APIs\*](#), we looked at the prototyping ideas offered by the garment-making practice of creating a toile to test a sewing pattern to explore the details of our API descriptions. These practices enable us to learn quickly and improve the chances of us building the “right” API without spending a great deal of resources ahead of time.

Finally, in Chapter 8, [\*Building APIs\*](#), we got to bring all our design and implementation work to a peak through the use of NodeJS and the DARRT model to complete the coding of our fully functional API. You also learned that while the internal aspects of our API implementation (data properties and actions) are important, knowing how to map those internal elements to the external world through resources, representations, and transitions is the key to building a highly usable and loosely coupled API.

In the build phase, creating API sketches and prototypes helps us refine our understanding of the implementation details before we have to commit them to code.

## The Release Phase

It wasn't until the release phase that we got around to talking about testing and security. In real life, we'd constantly be iterating through design, build, test, and secure loops throughout the life of the API project. But, as was mentioned in the text, the linear character of the book form meant it made sense to save some activities for later in the cycle.

In this part of the book, we discussed the power of SRTs (simple request tests) and Postman collections in Chapter 9, [\*Testing APIs\*](#). We also explored the value of creating your own custom testing library using the ChaiJS framework built into the Postman testing platform. And we created our own command-line scripted interface for executing Postman test collections locally.

In Chapter 10, [\*Securing APIs\*](#), we covered the basics of message encryption, user identity, and access control. You also learned to use the Auth0 online platform for defining and supporting the security model for our APIs. Finally, you learned how to easily add machine-to-machine access control with JSON Web Tokens with a small addition to the DARRT framework in NodeJS.

With our API designed, built, tested, and secured, we then got to deploy it into production on the Heroku platform in Chapter 11, [\*Deploying APIs\*](#). Along the way we explored the basics of deployment automation and learned about the differences between continuous integration, continuous delivery, and continuous deployment.

And last, in Chapter 12, [\*Modifying APIs\*](#), we covered best practices for updating an API after it's been released into production. We reviewed the Hippocratic oath for APIs ("First, do no harm") and learned the three rules of designing and building nonbreaking changes: (1) take nothing away, (2) don't redefine things, and (3) make additions optional. We explored

decision-making around the use of side-by-side deployments and overwriting releases and, to wrap it all up, we covered the proper way to permanently shut down an existing production API.

Releasing APIs is all about validating the API with testing, securing it properly, automating production releases, and safely and consistently updating APIs once they've been deployed into production.

And that brings us full circle. Once you've released your API into production, the whole thing starts up again either through creating updates for your existing API or through designing and building entirely new APIs that can solve new business problems and meet the needs of a new set of developers. Everything is a circle!

## What's Next?

So what's next? Well, that's up to you. Throughout the book, I've tried to give you pointers on how to identify and tackle the range of tasks needed to design and build great web APIs. At each stop along the journey, I've offered samples of wisdom I've learned from others and through my own experience. I've also provided examples of tools and techniques that I've found handy as I faced each new opportunity.

One of the challenges of writing a book like this is that technology and practices change over time. It's likely that readers will one day find that some parts of this book have become outdated or invalid. Despite that seeming inevitability, it's been my experience that all through the history of computing, the *problems* we face don't change much, but the *technology* we have at our disposal does. I hope the core ideas in this book will still provide help and encouragement even when the technical details are no longer applicable.

Above all, I hope this book will inspire a whole new collection of designers, programmers, and software architects to set their sights high as they continue to create valuable APIs, whether on their own, in a team, at their company, or on the open web itself. We need lots of creative people tackling difficult problems in ways that improve the lives of everyone touched by technology.

As you continue your travels in the API space, I wish you all the best. I hope this book will continue to be valuable to you as you set out on your own personal journey to design and build great web APIs.



# **Part 5**

# **Appendices**

# Appendix 1

## Installation Hints

---

The examples and exercises in this book rely on a handful of handy tools and utilities. This appendix is a short guide to locating and safely installing these tools on your local machine. Most of the install routines are pretty straightforward. However, a couple are a bit involved since the tools are supported on many different operating systems and involve multiple versions. I've tried to provide additional advice along the way to help steer you through any potential problems.

I advise you stick to installing these tools from the web locations mentioned in this appendix. Many of these utilities are free, open-source software products, and some version of them may be hosted on private web pages. But there's a slight chance that these unofficial download pages will serve up versions of the software with added trackers or even malicious code injected into the download. For that reason, be careful when you install software from non-standard locations on the web.

All the instructions assume you have access rights to installing software on your local machine. In some cases, you may need to get additional permission to do this. You may need to check with your local hardware administrator for details.

Finally, you don't need to install all the utilities mentioned here before you start reading the book. In fact, if you plan to just skim the book and/or not do the exercises at the end of the chapters, you may not need to install the

software at all. One approach I encourage in my classroom version of this content is to install each app as needed along the way. That spreads out the install effort a bit.

# curl

The curl utility is basically a command-line browser. As we discuss in Chapter 1, [Getting Started with API First](#), we'll use curl to make API calls from the command line throughout the book.

## Checking for curl

You may have curl already installed on your machine. You can check on this by typing the following in a command-line window:

```
$> curl
```

If it's installed, you should get the following response:

```
curl: try 'curl --help' or 'curl --manual' for more information
```

## Installing curl

If you need to install curl, the best way to do this is to visit the home page for the curl utility and use the download wizard at that website to help you find the proper package to install for your operating system.

1. Start by going to this web page: <https://curl.haxx.se/dlwiz>. That should start the download wizard. To answer the first question, select the curl executable option.
2. At the next screen, select your operating system from the drop-down list and press the Select button. For example, I selected the Linux option.
3. At the next screen, you need to select your Flavour and press the Select button. For example, I selected the Ubuntu option.

4. Depending on your selections to this point, you may see a page listing one or more download files or you may need to select your OS version and press the Select button. For example, for my Linux -> Ubuntu selection, I needed to select an OS version and I selected “cosmic.”
  
5. Depending on your selections you may see a page asking for you to select a CPU. Since I selected Linux to start, I don’t see this CPU screen. Instead, I see the screen [shown](#) that lists the download files.

curl / [Download](#) / [Download Wizard](#)

## curl Download Wizard

Welcome to the download wizard. This helps you figure out what package to download. Proceed and answer the questions below! [Show all Downloads](#)

Number of packages: 224  
Number of OSes covered: 33

[Package Type](#) ▶ [OS](#) ▶ [Flavour](#) ▶ [OS Version](#) ▶ [CPU](#)

### Select Type of Package

We provide packages of different types. Select one (or select ‘show all’ to view all types)

**curl executable** - You will get a pre-built ‘curl’ binary from this link (or in some cases, by using the information that is provided at the page this link takes you). You may or may not get ‘libcurl’ installed as a shared library/DLL.

**libcurl development** - This is for libcurl development - but does not always contain libcurl itself. Most likely header files and documentation. If you intend to compile or build something that uses libcurl, this is most likely the package you want.

Once you get to the download listing page, you may see just one file to download along with some notes about how the install process will work. You may also see more than one download option. These options might point to various places to download the same file. You may also have options to download ZIP or executable versions of the package. Finally, multiple release versions of curl may be available for your machine. You should usually pick the most recent version. It doesn’t matter too much since we’re not doing anything advanced with curl in this book.

Once you download the file, just follow along with the instructions in the notes to get your version of curl up and running. Like before, you can test your install using the example at the start of this section (see [curl](#)).

# Git

We use Git as the source control manager for all the projects in the book. It's first mentioned in Chapter 2, [Understanding HTTP, REST, and APIs](#), but almost every chapter after that has some Git commands.

## Checking for Git

You may already have Git installed on your machine. You can check this by typing the following at the command line:

```
git --version
```

If Git is installed you should see something like the following:

```
git version 2.11.0
```

The exact version isn't too important, but I advise that you use version 2 or higher when working with the examples in this book.

## Installing Git

If you need to install Git, your best bet is to visit the official download page (<https://git-scm.com/downloads>), as shown in the following screenshot. Once you get there you should see a suggested download on the right. You can also click on the operating system links to download the latest releases. Depending on the operating system you select, you'll see a list of commands for downloading the release, or in some cases the download will automatically start. In either case, you just need to follow along with any instructions you see along the way to complete the install.



--local-branching-on-the-cheap

Search entire site...

[About](#)

[Documentation](#)

[Downloads](#)

GUI Clients  
Logos

[Community](#)

The entire [Pro Git book](#) written by Scott Chacon and Ben Straub is available to [read online for free](#). Dead tree versions are available on [Amazon.com](#).

## Downloads



[Mac OS X](#)



[Windows](#)



[Linux/Unix](#)

Older releases are available and the [Git source repository](#) is on GitHub.

### GUI Clients

Git comes with built-in GUI tools ([git-gui](#), [gitk](#)), but there are several third-party tools for users looking for a platform-specific experience.

[View GUI Clients →](#)



### Logos

Various Git logos in PNG (bitmap) and EPS (vector) formats are available for use in online and print projects.

[View Logos →](#)

You can confirm the install using the same test mentioned earlier.

# GitHub and SSH

Several chapters include commands for accessing and modifying Git repositories on the GitHub public website. You don't need to install any GitHub client, but you'll need to make sure you have support for command-line level SSH and have generated a private/public key pair that will be used when you read and write data to GitHub from the command line on your local machine.

## GitHub Account



You'll need to have a GitHub account and password to use this website.

## Adding Your SSH Certificate to GitHub

In order to interact with the Git engine on the GitHub website, you'll need an SSH certificate on your machine. This is how the GitHub server will be able to recognize you and will be used to control access rights to your repositories. In my experience, the process of generating an SSH key and getting onto GitHub is a bit fiddly. The exact process is different for each operating system (Windows, Mac, and Linux). I'll walk you through the process that I've been able to consistently use (with only slight variations for operating systems). However, you may *still* need to refer to specific operating system instructions on the web. If you're lucky, this has all been done already and you can skip it. If not, here goes!

*Step 1:* First, at the command line (for Windows, open [git bash](#)), check to see if you already have an SSH cert installed on your machine. To do that, type the following:

```
ls -al ~/.ssh
```

If you see the following files: `id_rsa` and `id_rsa.pub`, you have a usable cert all set and ready to go. You can skip to Step 4.

*Step 2:* If you need to generate a new private/public SSH key pair, type the following in your command window:

```
ssh-keygen -t rsa -b 4096 -C "your@email.com"
```

For `"your@email.com"`, enter the email address you want to use for this certificate. To keep things simple, I use the same email address I used to create my GitHub account. If you want to use a different one, that's fine too.

You'll be prompted to enter a password for this generated certificate (you'll be asked twice), and then you'll see several lines of text on the screen that indicate the certificate was generated successfully.

*Step 3:* The next step is to store your certificate safely in memory on your machine to make it less tedious to check content in and out of GitHub. To do this, fire up the `ssh-agent` utility and then add your certificate to that utility's storage. For example, this is what I typed and the responses I got along the way:

```
$> eval "$(ssh-agent -s)"
Agent pid XXXX
$> ssh-add ~/.ssh/id_rsa
Enter passphrase for /home/mca/.ssh/id_rsa:
Identity added: /home/mca/.ssh/id_rsa (/home/mca/.ssh/id_rsa)
```

At this point, you've created an SSH certificate and stored the private portion of it (`id_rsa`) safely in your machine's memory. Now you need to copy the public portion of the key (`id_rsa.pub`) onto your machine's clipboard and then paste it into GitHub directly. That will complete the process of linking your machine to GitHub's servers.

*Step 4:* To copy the public portion of your SSH key onto your machine’s clipboard, type the following in your command-line window:

On Linux:

```
xclip -sel clip < ~/.ssh/id_rsa.pub
```

On Mac:

```
pbcopy < ~/.ssh/id_rsa.pub
```

On Windows:

```
clip < ~/.ssh/id_rsa.pub
```

Now you should have the public key copied onto your in-memory clipboard and are ready to paste it into GitHub. That’ll be a couple of steps too.

*Step 5:* In this final step, you’ll log into the GitHub web site, navigate to the SSH key page, add your new key, and save it to GitHub’s servers.

- Open your browser, navigate to <http://github.com>, and log in with your username and password.
- In the upper-right corner of the page, click your profile icon.
- In the drop-down menu, find and select the Settings option.
- When the page loads, locate the “SSH and GPG Keys” option on the left and click it. A new screen should load (the one showing all your registered SSH keys).
- Locate the New SSH Key or Add SSH Key button and click it.
- You’ll be prompted to enter two things: a title and your SSH key. Type some memorable name for this key in the title box and then paste your SSH key into the big input box.

- Finally, click the Add SSH Key button to store all this onto GitHub's servers. You might be asked to enter your GitHub password too (not your certificate password!).

Now you should be all set for passing data between your workstation and the GitHub website. If you want to be able to check content in and out of GitHub from another machine, you'll need to repeat this SSH setup process for each machine. I have about five or six machine keys registered in GitHub right now.

# NodeJS and npm

We'll be using NodeJS and the related npm (Node Package Manager) utility throughout the book. For example, many of the sample command-line utilities are written in NodeJS and are installed and updated using npm. We first refer to these two important tools in Chapter 3, [Modeling APIs](#).

## Checking for NodeJS

You may already have NodeJS and npm installed. You can check on this by typing the following on the command line:

```
node -v
```

If NodeJS is installed, you should get a response that is the version number of your release of NodeJS. For example, the response on my machine is this:

```
v10.14.10
```

The exact version isn't important, but you should be using version 10 or above since that's the version I used to write the code in this book.

## Checking for npm

When NodeJS gets installed, the latest version of npm should also be installed automatically. You can check this with the following command:

```
npm -v
```

The response should be the current installed version on your machine. For example, my machine responds with this:

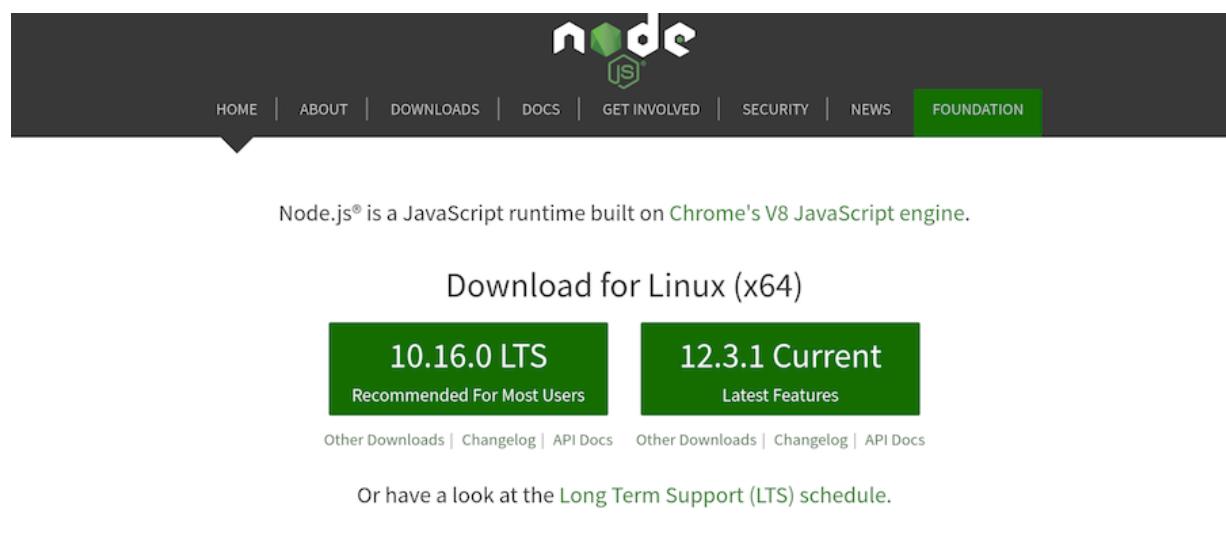
```
6.4.1
```

The exact version isn't important, but you should use at least version 6 or above. You may also be prompted to update your npm instance, and that's

usually a good idea.

## Installing Nodejs and npm

Installing NodeJS on your machine is usually pretty simple. The best place to start is to go to the NodeJS home page (<http://nodejs.org>). Right on the home screen you should see one or more buttons prompting you to download the install package for NodeJS, as shown in the following screenshot. The first button will have the letters *LTS*, which stand for long-term support. You may also see a Current button. I advise installing the LTS option since it will be the most stable and reliable release. The examples written for this book were created using LTS version 10.14.10. Any stable version above release 10 should be fine.



The screenshot shows the official Node.js website. At the top, there's a dark header with the Node.js logo and a navigation bar with links: HOME, ABOUT, DOWNLOADS, DOCS, GET INVOLVED, SECURITY, NEWS, and FOUNDATION. The FOUNDATION link is highlighted with a green background. Below the header, a dark banner states: "Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine." In the center, there's a heading "Download for Linux (x64)". Two large green buttons are prominently displayed: "10.16.0 LTS" (labeled "Recommended For Most Users") and "12.3.1 Current" (labeled "Latest Features"). Below these buttons, smaller links provide "Other Downloads | Changelog | API Docs" for each option. Further down, a link says "Or have a look at the Long Term Support (LTS) schedule." At the bottom of the screenshot, there's a footer with links for "Report Node.js issue | Report website issue | Get Help" and a note: "© Node.js Foundation. All Rights Reserved. Portions of this site originally © Joyent."

If you're looking for more choices, you can visit the download page (<https://nodejs.org/en/download>) to find lots of options. Since NodeJS is available on many platforms and in multiple releases, you'll need to take a minute to sort through these options if you want to do anything other than install the default option offered to you on the home page. My advice is to

use the default option unless it doesn't work for you, and then dive into the downloads page.

Once you complete the install, you should have both NodeJS and npm available. To confirm this, use the instructions provided earlier for checking for both NodeJS and npm.

# Postman

Postman is the platform we'll use to run our API tests. Unlike all the other apps you can install for this book, Postman is a GUI app. That means to launch the app, you'll need to find it in your machine's application menu. This will be a bit different for each operating system.

The best way to install it is to go to the Postman download page (<https://www.getpostman.com/downloads>), as shown in the following screenshot. There you should see a button on the screen that simply says Download. This should be the proper release package for your operating system. If not, you'll find other links on the page to download and install the package you would like to install.



## Postman Account



I advise creating an account and password for your Postman tests. This makes it easier to get things done on the site and allows you to easily store tests and other things on Postman's servers.

# Newman

The Newman command-line app can run the tests created by the Postman app. This makes running Postman tests really simple, and it also makes it easy to include these tests in command-line-driven build scripts.

## Checking for Newman

You can check if you already have Newman installed by typing the following at the command line:

```
newman -v
```

If it's installed, you'll get a response showing the version number of your release of Newman. On my machine I get this:

```
4.5.0
```

## Installing Newman

You can install the Newman test runner using the npm utility you installed earlier (see [NodeJS and npm](#)). To do that, just type the following at the command line:

```
npm -g install newman
```

That's all you need to do. You can check to see if it was properly installed using the instructions in [Newman](#).

## Installing HTMLExtra

A handy add-on to the `newman` toolchain produces good-looking HTML reports that show the results of your `newman` test run. It's called HTMLExtra. You can install this using npm with the following command:

```
npm i newman-reporter-htmlextra
```

This will come in handy when you want to post your latest test results for others to view.

# Heroku Client

Heroku is the deployment and runtime platform used for all the API services we use in the book. In order to define applications and move code from your local machine to the Heroku platform, you need to install the Heroku client command-line app.

## Heroku Account



You'll need to create an account and password on the Heroku platform to use the client app to deploy your API code to Heroku.

## Checking for Heroku Client

To see if you have the Heroku client already installed on your local machine, just type the following into the command line:

```
heroku -v
```

If the app is available, you should see a response that includes version information for the installed release on your machine. For example, on my machine, the response looks like this:

```
heroku/7.24.4 linux-x64 node-v11-14.0
```

## Installing the Heroku Client

The best way to install the Heroku client app is to visit the Heroku CLI page (<https://devcenter.heroku.com/articles/heroku-cli>), where you'll see several buttons and prompts to download and install the right version for your operating system. You'll also see a handful of other ways to install the Heroku client, but I advise you only use these alternatives if you can't get one of the common OS options to work for you.

Once you have the Heroku client installed, you can test your results using the instructions provided in [Heroku Client](#).

# Installing the Local Utilities

I wrote a few local utilities to help you handle some assignments in the book. These are simple command-line tools that you can modify to better fit your needs.

All the apps are in the [code/utilities/](#) folder in the downloadable source code for this book. You'll find three folders there, one for each utility app: [norman](#), [wsd-gen](#), and [wsd-util](#).

## norman

The [norman](#) utility helps automate deployment tasks. We'll use it in Chapter 11, [Deploying APIs](#). You can install the [norman](#) utility by moving into that folder ([/code/utilities/norman/](#)) and typing the following on the command line:

```
npm -g install
```

You can test the install by typing the following on the command line and checking the response:

```
$ norman --help
Usage: norman [options] <coll> [en]

Options:
 -h, --help output usage information
```

## wsd-gen

The [wsd-gen](#) utility will help you generate sequence diagrams from your WSD text files. We'll see this utility in action in Chapter 4, [Designing APIs](#). Move into the [code/utilities/wsd-gen/](#) folder and type the following:

```
npm -g install
```

You can test the install by typing the following on the command line and checking the response:

```
$ wsddgen --help
Usage: wsddgen [options] <file>

Options:
-h, --help output usage information
```

## wsd-util

The [wsd-util](#) app can generate a valid ALPS (Application-Level Profile Semantics) document from the same WSD text file used to generate your sequence diagrams. Move into the [code/utilities/wsd-util/](#) folder and type the following:

```
npm -g install
```

You can test the install by typing the following on the command line and checking the response:

```
mca@mamund-ws:.../wsd-util$ wsd2alps --help
Usage: wsd2alps [options] <file>

Options:
-h, --help output usage information
```

Those are all the local utilities you'll need to do the assignments in the book.

## Appendix 2

### Exercise Solutions

---

This appendix contains solutions to the exercises that appear at the end of each chapter. Of course, many of the exercises have more than one “correct” solution. The ones offered here are meant as a general guide to help you think about how to tackle the various challenges posed throughout the book.

## Where's the Code?

In some cases, the printed solution here contains only part of the expected input and/or output. You can find the complete content in the source code download package associated with this book on the Pragmatic Bookshelf website at [https://pragprog.com/titles/maapis/source\\_code](https://pragprog.com/titles/maapis/source_code).

The code download contains a folder for each chapter in the book along with any associated code examples and a folder for the chapter exercise. The **exercise** folder includes two subfolders: a **before** folder and a **completed** folder. You can use the **before** folder when you're about to begin an exercise, and you can refer to the **completed** folder for an example of a successfully completed exercise. You may also find some notes or other comments in the folders that help you understand the steps to completing the solution.

# Solution for Chapter 1: Getting Started with API First

In the [Chapter Exercise](#), the task is to use the curl command-line tool to pull a series of responses from three related services (`company`, `account`, and `activity`) and save them to disk for later reference. Here is a step-by-step solution.

First, create a folder called `services` on your local drive and then change directories to move into that newly created `services` folder:

```
mca@mamund-ws:~/services$ mkdir services
mca@mamund-ws:~/services$ cd services
mca@mamund-ws:~/services$
```

Next, using the list supplied in the [Chapter Exercise](#), as a guide, start to pull the API responses from the services and save them to disk. Here's one way to do this:

```
mca@mamund-ws:~/services$ curl /
https://company-atk.herokuapp.com/company > company-home.json

% Total % Received % Xferd Average Speed Time Time Time Current
 Dload Upload Total Spent Left Speed
100 77 100 77 0 0 11 0 0:00:07 0:00:06 0:00:01 18

mca@mamund-ws:~/services$
```

The resulting data saved in the file `company-home.json` will look like this:

```
{"home" : {"name": "company", "rel" : "collection", "href": "/company/list/"}}
```

You can also use curl in a few other ways to get the same results. Here are some other examples:

```
mca@mamund-ws:~/services$ curl \
https://company-atk.herokuapp.com/company/list \
\\
```

```
--output company-list.json

mca@mamund-ws:~/services$ curl -X GET \
 https://company-atk.herokuapp.com/company/21r1aeuj87e \
> company-record.json
```

This exercise has a total of seven URLs. When you're done calling all of them, you should have a `services` folder that looks like this:

```
mca@mamund-ws:~/services$ ls
account-list.json
account-record.json
activity-list.json
activity-record.json
company-home.json
company-list.json
company-record.json
```

You can check the files in the `completed` folder to compare your API output to the output on the disk.

# Solution for Chapter 2: Understanding HTTP, REST, and APIs

The task in this exercise is to move our `services` folder (and all the API calls stored there) into the project repository we created in chapter 2 (see [Chapter Exercise](#)).

You can use the `before` folder for this chapter as a clean start for this exercise. It picks up where our last exercise left off.

Depending on whether you followed along in chapter 2 with the process of creating your `onboarding` project folder and subsequent changes, you may have some of this exercise already completed. To keep things clear, we'll start from the beginning by creating a new `onboarding` folder, making it a `git` repository, and then copying the `services` folder into the repository and committing the changes.

Here's a step-by-step solution.

## Create Your Local Git Repo

First, create your `onboarding` project folder and change directories into that new folder:

```
mca@mamund-ws:~/ $ mkdir onboarding
mca@mamund-ws:~/ $ cd onboarding
mca@mamund-ws:~/onboarding$
```

Next, make this new empty folder a `git` repository:

```
mca@mamund-ws:~/onboarding$ git init
Initialized empty Git repository in /home/mca/onboarding/.git/
mca@mamund-ws:~/onboarding$
```

Next, let's add a `README.md` file to the repo. It's always a good idea to use `README` files to help remember what's in the repo and why it was created.

For this example, we can create a simple text file that contains the following text and save it to disk as a file named “README.md”:

```
Onboarding API Project
This project contains files related to the BigCo Onboarding API.
```

Now do a `git status` to check the status of the repo:

```
mca@mamund-ws:~/onboarding$ git status
On branch master

Initial commit

Untracked files:
(use "git add <file>..." to include in what will be committed)

 README.md

nothing added to commit but untracked files present (use "git add" to track)
mca@mamund-ws:~/onboarding$
```

You’re ready to make your first commit to the repo:

```
mca@mamund-ws:~/onboarding$ git add README.md
mca@mamund-ws:~/onboarding$ git commit -m "added README"
[master (root-commit) ce69677] added README
 1 file changed, 3 insertions(+)
 create mode 100644 README.md
mca@mamund-ws:~/onboarding$
```

## Move the Services Folder into the Repo

Now that you have an up-to-date repository, you’re ready to move the `services` folder (and all its contents) into that repo and check the repo’s status:

```
mca@mamund-ws:~/onboarding$ mv ../*.services .
mca@mamund-ws:~/onboarding$ ls
README.md services
mca@mamund-ws:~/onboarding$
mca@mamund-ws:~/onboarding$ git status
On branch master
Untracked files:
```

```
(use "git add <file>..." to include in what will be committed)

services/

nothing added to commit but untracked files present (use "git add" to track)
mca@mamund-ws:~/onboarding$
```

Finally, you can commit the changes (the new folder and its contents) into the Git repository:

```
mca@mamund-ws:~/onboarding$ git add services/
mca@mamund-ws:~/onboarding$ git status
On branch master
Changes to be committed:
 (use "git reset HEAD <file>..." to unstage)

 new file: services/account-list.json
 new file: services/account-record.json
 new file: services/activity-list.json
 new file: services/activity-record.json
 new file: services/company-home.json
 new file: services/company-list.json
 new file: services/company-record.json

mca@mamund-ws:~/onboarding$ git commit -m "add services folder"
[master df445e6] add services folder
 7 files changed, 536 insertions(+)
 create mode 100644 services/account-list.json
 create mode 100644 services/account-record.json
 create mode 100644 services/activity-list.json
 create mode 100644 services/activity-record.json
 create mode 100644 services/company-home.json
 create mode 100644 services/company-list.json
 create mode 100644 services/company-record.json
mca@mamund-ws:~/onboarding$
```

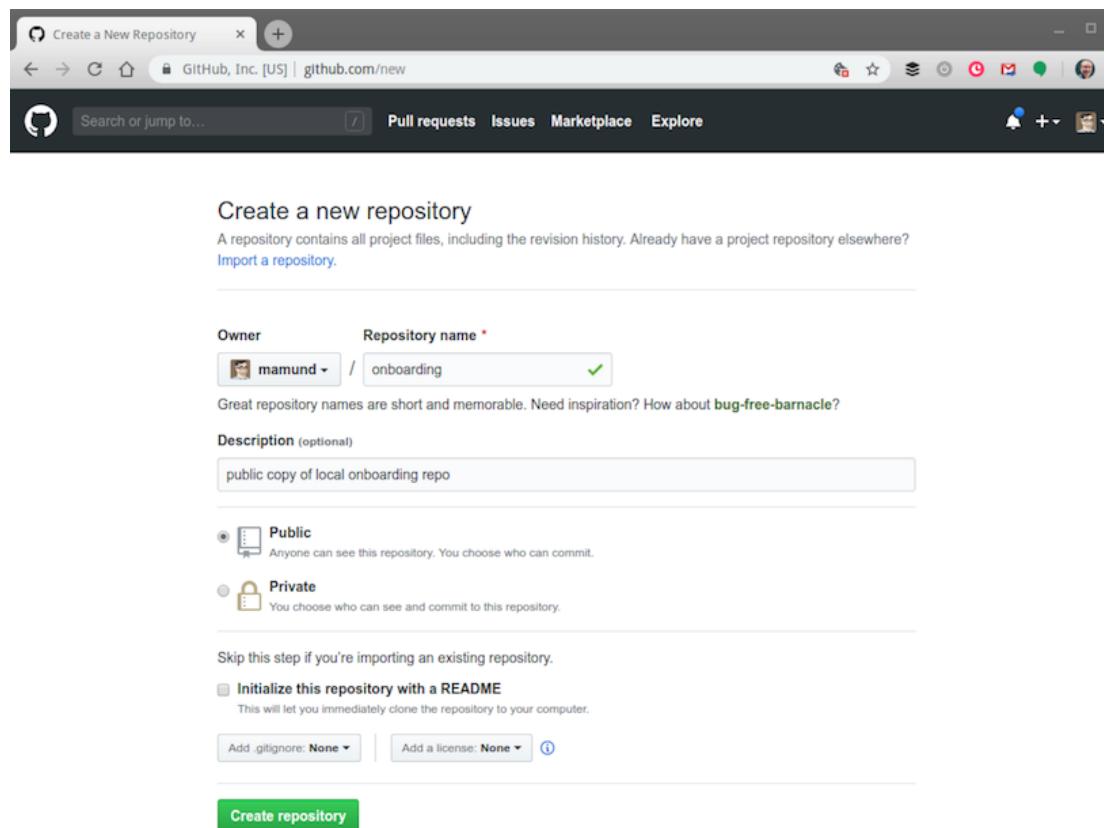
## Push Your Local Repo to GitHub

Our last task is to push this local repo up to <https://github.com> to make it publicly available for others. To do that we need to log into your GitHub account, create a repository there, copy the repository address onto our

clipboard, and then add that public address to our local repo. Then we can use the `git push` command to copy our local repo contents to GitHub.

First, log into your GitHub account and click New to start a new repository. Or you can just navigate to <https://github.com/new>.

Next, fill in the repository name (`onboarding`) and add a short description (`public copy of local onboarding repo`). Leave all the other entries as is and click “Create repository,” as shown in the following screenshot:



Once that’s created, you’ll see a response screen that contains a lot of text on how to start populating your public repository. For this solution, copy the two lines that appear under the `...or push an existing repository from the command line` option, as shown in the [screenshot](#). (You should see a small clipboard icon near that text to make it easy to copy the commands.)

The screenshot shows a GitHub repository page for 'mamund/onboarding'. At the top, there's a search bar and navigation links for Pull requests, Issues, Marketplace, and Explore. Below the header, there are tabs for Code, Issues, Pull requests, Actions, Projects, Wiki, Security, Insights, and Settings. A prominent section titled 'Quick setup — if you've done this kind of thing before' provides instructions for cloning the repository via HTTPS or SSH, along with a link to a README file. It also suggests creating a new repository on the command line with the following commands:

```
echo "# onboard" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin git@github.com:mamund/onboarding.git
git push -u origin master
```

Another section shows commands for pushing an existing repository from the command line:

```
git remote add origin git@github.com:mamund/onboarding.git
git push -u origin master
```

A third section, '...or import code from another repository', includes a link to import code from Subversion, Mercurial, or TFS.

Paste the commands from your clipboard into your command window and press the Enter key to commit the **git push** from your local repository to the public one. Your command line should look something like this:

```
mca@mamund-ws:~/onboarding$ git remote add origin \
 git@github.com:mamund/onboarding.git
mca@mamund-ws:~/onboarding$ git push -u origin master
Counting objects: 13, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (12/12), done.
Writing objects: 100% (13/13), 2.06 KiB | 0 bytes/s, done.
Total 13 (delta 4), reused 0 (delta 0)
remote: Resolving deltas: 100% (4/4), done.
To git@github.com:mamund/onboarding.git
 * [new branch] master -> master
```

```
Branch master set up to track remote branch master from origin.
mca@mamund-ws:~/onboarding$
```

You can check your work by refreshing your project page (from the previous step). You should then see something that looks like this:

The screenshot shows a GitHub project page for the repository 'mamund / onboarding'. The page has a dark theme. At the top, there's a header with a search bar, navigation links for 'Pull requests', 'Issues', 'Marketplace', and 'Explore', and a user profile icon. Below the header, the repository name 'mamund / onboarding' is displayed, along with statistics: 2 commits, 1 branch, 0 releases, and 1 contributor. A 'Clone or download' button is highlighted in green. The main content area shows a commit history with two entries:

| Author | Commit Message      | Time                                 |
|--------|---------------------|--------------------------------------|
| mamund | add services folder | Latest commit df445e6 19 minutes ago |
|        | services            | add services folder                  |
|        | README.md           | added README                         |

Below the commit history, there's a section titled 'Onboarding API Project' with the subtext: 'This project contains files related the BigCo Onboarding API.'

# Solution for Chapter 3: Modeling APIs

In the [\*Chapter Exercise\*](#), we introduced the related CreditCheck service. This will be used to check the creditworthiness of our new on-boarded customers.

The specific task is to create the `credit-check-workflow.txt` document. By inspecting the `credit-check-story.md` and `credit-check-story.pdf` documents in the code download for this chapter, you should be able to come up with a simple workflow that captures the CreditCheck service.

## Updating the Credit-Check-Workflow Document

To make this a bit easier, you have a partially completed `credit-check-workflow.txt` document to start with. It has two steps in the flow already named (`Home` and `CreditCheckForm`) and two steps identified but not named (`Action1???` and `Action2???`). By reading through the story you should be able to come up with the two remaining steps that aren't in the workflow document.

The two unnamed workflow actions are `CreditCheckHistory` and `CreditCheckItem`. When you update your document, it should look like this:

```
Credit Check Workflow

Home -> CreditCheckHistory
CreditCheckHistory -> CreditCheckForm(companyName) -> CreditCheckItem
CreditCheckItem -> CreditCheckHistory
CreditCheckHistory -> Home
```

## Saving the Documents to the `assets` Folder

After saving the updated document (`credit-check-workflow.txt`), copy that document and the others for this solution (`credit-check-story.md` and `credit-`

[check-form.pdf](#)) into a subfolder of the `onboarding` project (`onboarding/assets`).

Your folder should now look like this:

```
mca@mamund-ws:~/onboarding$ ls assets/
credit-check-cycles.md credit-check-story.pdf credit-manager-form.html
credit-check-story.md credit-check-workflow.txt credit-manager-form.pdf
mca@mamund-ws:~/onboarding$
```

## Updating the Project with Git

The final step is to commit these changes to the project using Git:

```
mca@mamund-ws:~/onboarding$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Untracked files:
 (use "git add <file>..." to include in what will be committed)

 assets/

nothing added to commit but untracked files present (use "git add" to track)
mca@mamund-ws:~/onboarding$ git add --all
mca@mamund-ws:~/onboarding$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes to be committed:
 (use "git reset HEAD <file>..." to unstage)

 new file: assets/credit-check-cycles.md
 new file: assets/credit-check-story.md
 new file: assets/credit-check-story.pdf
 new file: assets/credit-check-workflow.txt
 new file: assets/credit-manager-form.html
 new file: assets/credit-manager-form.pdf

mca@mamund-ws:~/onboarding$ git commit -m"update assets for credit-check"
[master 106f529] update assets for credit-check
 6 files changed, 77 insertions(+)
 create mode 100644 assets/credit-check-cycles.md
 create mode 100644 assets/credit-check-story.md
```

```
create mode 100644 assets/credit-check-story.pdf
create mode 100644 assets/credit-check-workflow.txt
create mode 100644 assets/credit-manager-form.html
create mode 100644 assets/credit-manager-form.pdf
mca@mamund-ws:~/onboarding$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
 (use "git push" to publish your local commits)

nothing to commit, working directory clean
mca@mamund-ws:~/onboarding$
```

# Solution for Chapter 4: Designing APIs

In the [\*Chapter Exercise\*](#), you were instructed to complete a detailed version of the `credit-check-workflow.txt` document, resolve the names in that workflow against the <https://schema.org> site, and then use that updated workflow document to create your `credit-check-diagram.wsd` document. Finally, you were instructed to use that WSD document to generate a PNG diagram of the CreditCheck service workflow.

## Detailed `credit-check-workflow.txt`

This first step involves making sure you include possible property values and navigation actions that will be exposed by the CreditCheck service. This is also a good time to resolve any property names against your company dictionary to make sure you only expose names that everyone else using your API already understands. I combined both steps into one and produced the following updated `credit-check-workflow.txt` document:

```
Credit Check Workflow

Home -> CreditCheckHistory
 [ratingId, companyName, dateRequested, ratingValue]
CreditCheckHistory -> CreditCheckForm(companyName) -> CreditCheckItem
 [ratingId, companyName, dateRequested, ratingValue]
CreditCheckItem -> CreditCheckHistory
 [ratingId, companyName, dateRequested, ratingValue]
CreditCheckHistory -> Home

Validated Names:
* ratingId := https://schema.org/identifier
* companyName := https://schema.org/legalName
* dateRequested := https://schema.org/Date
* ratingValue := https://schema.org/ratingValue
```

## Create a WSD Document

The next step is to create a web sequence diagram (WSD) version of the workflow. I created one called [credit-check-diagram.wsd](#) that looks like this:

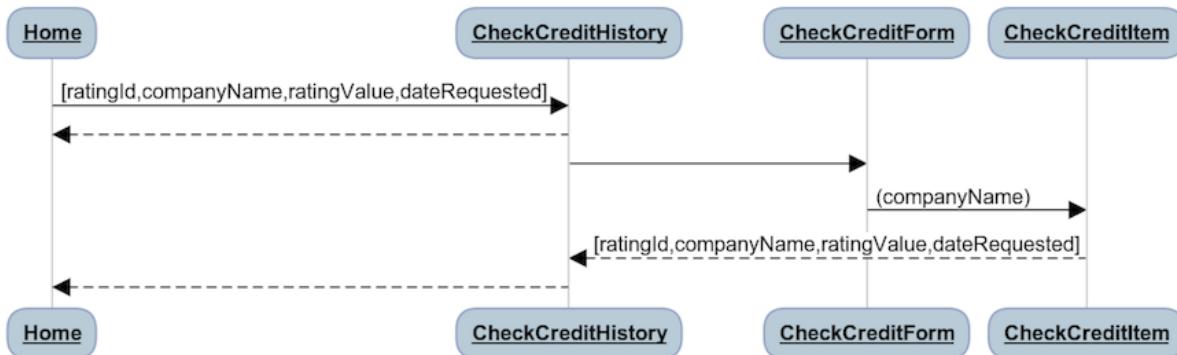
```
Home -> CheckCreditHistory:
 [ratingId, companyName, ratingValue, dateRequested]
CheckCreditHistory --> Home:
CheckCreditHistory -> CheckCreditForm:
CheckCreditForm -> CheckCreditItem: (companyName)
CheckCreditItem --> CheckCreditHistory:
 [ratingId, companyName, ratingValue, dateRequested]
CheckCreditHistory --> Home:
```

## Generate a PNG File

Finally, I used the [wsdgen](#) utility to generate a PNG version of the WSD file using the following command-line action:

```
wsd-gen credit-check-diagram.png
```

This generates the file [credit-check-diagram.png](#), which should look something like this:



Be sure to save your project updates using [git](#) too.

# Solution for Chapter 5: Describing APIs

The exercise outlined in the [Chapter Exercise](#), is to use the assets you created in the exercises in Chapter 3, [Modeling APIs](#), and Chapter 4, [Designing APIs](#)) to create a complete API description document in the Application-Level Profile Semantics (ALPS) format.

This involves collecting all the *properties* and *actions* in your design document and coding them into the ALPS machine-readable format. To help you along, the [before](#) folder for this exercise contains partially completed editions of both the ALPS XML format ([credit-check-alps.xml](#)) and the ALPS JSON format ([credit-check-alps.json](#)).

## Using the ALPS JSON Starter File

When you open the ALPS JSON starter file for this exercise ([credit-check-alps.json](#)), you'll see all of the top-level elements you'll need for this solution: [alps](#), [title](#), and [doc](#). There's nothing to add there. You'll also see the top-level [descriptor](#) collection with initial elements for the [ratingId](#) property, the [ratingItem](#) container (like an object in programming code), and a couple of actions ([home](#) and [checkCreditHistory](#)). These should give you a pretty good hint about how to complete your version of the document.

## Completing the Credit-Check ALPS Properties

The first section of the document describes all the possible properties the CreditCheck service will use. We can find that list in the [credit-check-workflow.txt](#) document in the [Validated Names](#) section. You just need to transfer those properties into the ALPS format.

Here's what the "properties" section of the completed ALPS document looks like:

```

{
 "id" : "ratingId", "type" : "semantic", "tags" : "property",
 "ref" : "https://schema.org/identifier",
 "text" : "The unique identifier for the rating record."
},
{
 "id" : "companyName", "type" : "semantic", "tags" : "property",
 "ref" : "https://schema.org/legalName",
 "text" : "Name of the company; used to look up the rating value"
},
{
 "id" : "dateRequested", "type" : "semantic", "tags" : "property",
 "ref" : "https://schema.org/Date",
 "text" : "Date/Time (UTC format) the rating record was created."
},
{
 "id" : "ratingValue", "type" : "semantic", "tags" : "property",
 "ref" : "https://schema.org/ratingValue",
 "text" : "Actual credit rating (value between 1 and 10 [0=unrated])"
}

```

## Adding the `ratingItem` Container to the ALPS Document

ALPS documents let you combine a set of properties into a collection that ALPS calls `containers`. We only need one container for the CreditCheck service, and it looks like this:

```

{
 "id" : "ratingItem", "type" : "semantic", "tags" : "container",
 "descriptors" : [
 {"href" : "#ratingId"},
 {"href" : "#companyName"},
 {"href" : "#dateRequested"},
 {"href" : "#ratingValue"}
]
}

```

You can add this container right after the list of properties in the ALPS document.

## Adding the Actions to the ALPS Document

The final step is to describe in detail all the possible *actions* the CreditCheck service will need to support. You can find that in the first part of the [credit-check-workflow.txt](#) document. The starter file already has two of these described ([home](#) and [creditCheckHistory](#)). You just need to add the other two from the workflow document ([creditCheckForm](#) and [creditCheckItem](#)). That looks something like this:

```
{
 "id" : "home", "type" : "safe", "tags" : "actions",
 "text" : "Root resource; Points to other actions in the service."
},
{
 "id" : "creditCheckForm", "type" : "safe",
 "tags" : "actions", "rtn" : "ratingItem",
 "text" : "Returns the input form for making a credit-check request."
},
{
 "id" : "creditCheckHistory", "type" : "safe",
 "tags" : "actions", "rtn" : "ratingItem",
 "text" : "Returns a list of past credit rating records.",
 "descriptors" : [
 {"href" : "#companyName"}
]
},
{
 "id" : "creditCheckItem", "type" : "safe",
 "tags" : "actions", "rtn" : "ratingItem",
 "text" : "Returns a single credit check record.",
 "descriptors" : [
 {"href" : "#ratingId"}
]
}
```

And that's all you need for this solution. You can compare your finished ALPS document to the one in the [completed](#) folder for this exercise. The folder also includes an ALPS XML format of the document, in case you're working in XML for this solution.

# Solution for Chapter 6: Sketching APIs

The solution for the [Chapter Exercise](#), is your first chance to actually sketch out what your API resources and responses will look like. The instructions are to start with the WSD diagram from the previous exercise ([credit-check-diagram.png](#)) and then create sketches of what the API would look like.

The first sketch is an HTML sketch that shows all possible resources and the navigations between them. The second sketch is a JSON sketch that shows the details of what a list and an item response look like. Of course, you should be using the Apiary editor to create these sketches.<sup>[98]</sup>

## Your Apiary Starting Sketch

To begin, open your browser and enter the address for the Apiary editor. Next, create a new API project by opening the drop-down menu at the top of the page and typing in a new project name (for example, credit-check-sketch) and pressing the Create API button. This opens a new project preloaded with a sample API sketch. You can delete everything in the left panel after your sketch name ([# credit-check-sketch](#)). Now you're ready to start creating your own sketches for the CreditCheck service.

## The Credit-Check Resource Sketch

The first sketch is an HTML version that shows each of the resources your API will expose along with links *between* those resources to show the workflow from one step to the next. Recall that our ALPS description document has the following steps:

- Home
- CreditCheckHistory
- CreditCheckForm
- CreditCheckItem

The first step is to create these four elements in our Apiary blueprint document. If you look in the **before** folder for this solution in the code download for the book, you'll find a file (**credit-check-resources.apib**) you can use to start with. Just open that file in your editor and copy-paste the text directly into your Apiary editor in your browser. You should see three resources already sketched out for you: Home, CreditCheckForm, and CreditCheckItem. It should look something like this:

```
FORMAT: 1A
HOST: http://polls.apiblueprint.org/

credit-check-sketch

The public API profile for BigCo's CreditCheck service.

Home [/]

Root resource; points to other resources/actions in the service

Root [GET]

+ Response 200 (text/html)

<html>
 <head>
 <title>Home</title>
 </head>
 <body>
 <h1>Home</h1>

 CreditCheck History

 </body>
</html>

CreditCheck History [/list]

Returns a list of past credit rating records.
```

```
List [GET]
```

```
+ Response 200 (text/html)
```

```
<html>
</html>
```

```
CreditCheck Form [/form]
```

```
Returns the input form for making a credit-check request.
```

```
Form [GET]
```

```
+ Response 200 (text/html)
```

```
<html>
 <head>
 <title>Credit Check Form</title>
 </head>
 <body>
 <h1>Credit Check Form</h1>

 Home

 Credit Check History

 <form action="/list/123">
 <label>Company Name</label>
 <input name="companyName" />
 <input type="submit" />
 </form>
 </body>
</html>
```

```
CreditCheck Item [/list/123]
```

```
Returns a single credit check record
```

```

Item [GET]

+ Response 200 (text/html)

<html>
 <head>
 <title>Credit Check Item</title>
 </head>
 <body>
 <h1>Credit Check Item</h1>

 Home

 Credit Check History

 Credit Check Form

 <p>RatingItem goes here...</p>
 </body>
</html>

```

You can use this starter as a guide for completing the rest of the HTML sketch. You just need to add an HTML document for the `CreditCheckHistory` resource. Nothing fancy. Just make something that can act as a placeholder to show the eventual workflow of the API. You can find a finished version of this `apiB` document in the `completed` folder for this solution.

Because this sketch is in HTML format, you can use your browser to “test” the workflow by loading the starter URL in your browser and following the links back and forth in the API.

## The CreditCheck Response Sketch

You’ll notice that our HTML version of the sketch didn’t actually show what the response data looks like. It just showed the resources and the links between them. So in this second sketch, we’ll actually show the details of

the response data, and this time we'll do it using the JSON format—the most common API response format to date.

First, start a new sketch ([credit-check-responses](#)), and when the default sample content appears, remove it to make room for your two sketches ([creditCheckHistory](#) and [creditCheckItem](#)). Remember, you only need to show the detailed response bodies in this sketch. No workflow or links. You can paste the starter document in the [before](#) folder called [credit-check-responses.apib](#) into your Apiary editor page. It should look like this:

```
FORMAT: 1A
HOST: http://polls.apiblueprint.org/

Credit Check Responses

Public API profile for BigCo's CreditCheck service.

Credit Check History [/list]

Returns a list of past credit rating records.

List Past Credit Checks [GET]

+ Response 200 (application/json)

{
 "creditCheck" : [
 {
 "id" : "123",
 "
 },
 {
 "id" : "456",
 },
 {
 "id" : "789",
 }
]
}
```

```
Credit Check Item [/list/123]

Returns a list of past credit rating records.

Single Credit Checks [GET]

+ Response 200 (application/json)

{
 "creditCheck" : [
 {
 "id" : "123",
 }
]
}
```

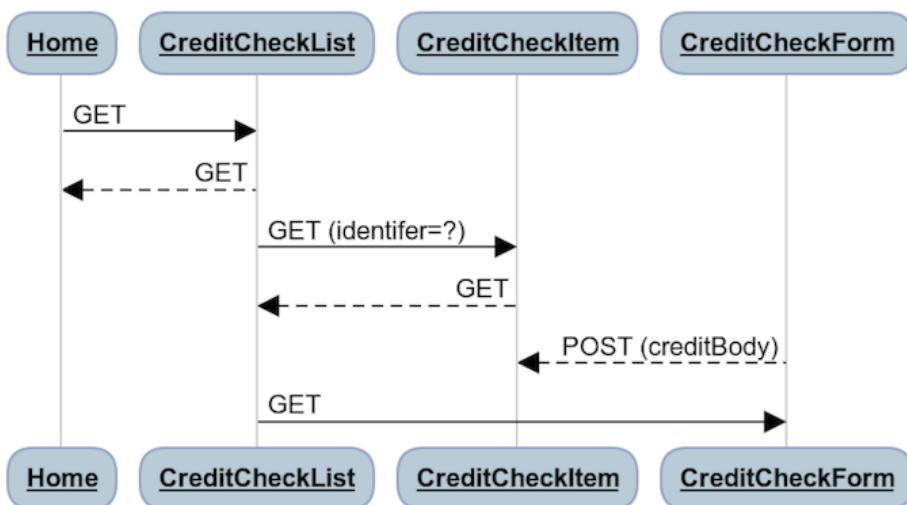
These two responses just need to be filled in with all of the property elements found in the ALPS description you created in the last exercise (see [Solution for Chapter 5: Describing APIs](#)). You can check your work against the `credit-check-responses.apib` document in the `completed` folder for this exercise.

# Solution for Chapter 7: Prototyping APIs

In Chapter 7, [Prototyping APIs](#), we focused on translating our API design into a more-detailed working prototype using the OpenAPI Specification (OAS). The exercise at the end of that chapter ([Chapter Exercise](#)) contained instructions to first create a new WSD document that reflects a translation of the CreditCheck WSD diagram and the CreditCheck ALPS description document into an HTTP-style API. Then you were to take that WSD and produce a completed API definition document using SwaggerHub and the OAS format. Finally, you were to combine the ReDocly HTML template and your completed CreditCheck API definition (in JSON format) to produce a single-page interactive API document for your API.

## HTTP-Style WSD File

The first step is to come up with the HTTP-style diagram of your initial API design. As you learned in Chapter 7, this act of converting the design documents (WSD and ALPS) into an implementation document (OAS) is an art, not a science. There are lots of possible solutions. The [diagram](#) is the one I created. You can use it as a guide and compare it with the one you came up with:



I saved both versions of this file (the PNG image and the .wsd text file) to the `assets` folder in my `onboarding` repository and committed them to the project using `git`.

## Credit-Check OAS Definition

The next step was to produce a completed API definition document using SwaggerHub and the OpenAPI Specification (OAS) format. Recall that OAS documents have three important root elements: `info`, `components`, and `paths`. I won't include all the documents in these pages (you can find them in the `exercise/completed` folder of the code associated with this chapter), but I'll share an abbreviated version of the `paths` section of my OAS file. This should give you a good idea of how I used the HTTP-style WSD to guide me as I put together the API definition:

```
paths section
paths:

 ### home ###
 /:
 get:
 operationId: home
 summary: Home resource for the API
 ...
 ### credit check list ###
 /list:
 get:
 operationId: creditCheckList
 summary: Use this to get a list of credit checks
 ...
 ### credit check item ###
 /list/{ratingId}:
 get:
 operationId: creditCheckItem
 summary: Use this to get a single credit check record
 ...
 ### credit check form ###
 /form:
 get:
```

```
operationId: creditCheckForm
summary: Use this to start a new credit check record
post:
 operationId: creditCheckWrite
 summary: Use this to complete a new credit check record
```

Notice that only one WRITE operation is exposed in this interface—the **POST** method on the [/form](#) resource. This implementation is perfectly valid, but it’s not very common. Often developers will **POST** to the [/list](#) resource. That’s fine too. I used the [/form](#) resource since most of the use of this API will be to generate a single credit-check rating, not to manipulate lists of ratings. This design will allow programmers to focus on just one URL ([/form](#)) for both READING the empty document to fill in (an option) *and* actually making the WRITE request to get a rating back. The other URLs are just for background use.

After completing my OAS document, I used the SwaggerHub Export function to download the “Unresolved YAML” file and the ZIP package that contained the JSON and YAML versions of my “resolved” OAS document. I added these to the **assets** folder in my repository and committed them to the project using [git](#).

## Creating API Documentation

The final step in this exercise was to use the ReDocly template we discussed in the chapter to create your own interactive API documentation for the CreditCheck API. For this, I just copied the HTML template from the ReDocly online repository (<https://github.com/Redocly/redoc#deployment>) and saved it to a new folder in my repository named **documentation**. I then updated one line of the template by replacing the sample OAS URL to one pointing to the URL I stored in my GitHub repository earlier.

I changed this line:

```
<redoc spec-url='http://petstore.swagger.io/v2/swagger.json'></redoc>
```

To this:

```
<redoc
 spec-url="https://raw.githubusercontent.com/api-tool-kit/open-api-spec/...
 master/credit-check-api.json">
</redoc>
```

*Note that the “...” in the URL above is just for text-wrapping to make sure it fits on the book page.*

Once I saved the updated template to my [documentation](#) folder, I also committed those changes to my Git repo.

With those three steps, you've created a WSD that reflects your HTTP-style API decisions, an OAS file that documents all the details of your working prototype, and finally a simple interactive API documentation page to include in your project.

# Solution for Chapter 8: Building APIs

In Chapter 8, [\*Building APIs\*](#), the task was to use the design assets to implement a working API service using NodeJS and the DARRT library. The three key tasks are these:

- Install the DARRT library into your NodeJS project.
- Update your DARRT files (data, action, resources, representations, and transitions).
- Once the API is completed, run a “happy path” validator script.

The instructions in the chapter also included a hint on how to initialize your NodeJS project using npm to update the project dependencies and start up the API service using [`nodemon`](#).

## Install the DARRT Library

The [`/before/`](#) version of this exercise has the DARRT library already installed. I did that to copy a few hints into the script to make it easier to complete the exercise.

You’ll need to use npm to update any dependencies for this project and then start the project in “monitor mode”:

```
npm install
npm run dev
```

That should start up your API project and output something that looks like this:

```
> credit-check@1.0.0 dev .../completed
> nodemon index

[nodemon] 2.0.2
```

```
[nodemon] to restart at any time, enter `rs`
[nodemon] watching dir(s): ***!
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index index.js`
listening on port 8181!
```

You can test your API service by typing the following at the command line in another window:

```
mca@mamund-ws:~/ $ curl localhost:8181/ -H "accept:application/json" | jq "."
```

This should produce output that looks like this:

```
% Total % Received % Xferd Average Speed Time Time Time
Current Dload Upload Total Spent Left
Speed
100 274 100 274 0 0 45712 0 --::-- --::-- --::-- 54800
{
 "home": [
 {
 "href": "http://localhost:8181/list/",
 "rel": "collection api",
 "name": "api-starter",
 "id": "list"
 }
]
}
mca@mamund-ws:~/exercise/completed$ ^C
```

Now that you have the initial project up and running, you can start to update the DARRT files to implement your [credit-check](#) API.

## Update Your DARRT Files

You need to update (or at least confirm the contents of) the following DARRT files:

- [data.js](#)
- [actions.js](#)

- [resources.js](#)
- [representations.js](#)
- [transitions.js](#)

Each of these files needs to reflect the details of your [credit-check API design assets](#) from previous chapters.

### [data.js](#)

To start, open the [data.js](#) file and enter the data property details from your [bigco-credit-check-api-1.0.0-swagger.yaml](#) document. When you're done, it should look like this:

```
// this service's message properties
exports.props = [
 'id',
 'status',
 'companyName',
 'ratingValue',
 'dateCreated',
 'dateUpdated'
];

// required properties
exports.reqd = ['id', 'companyName', 'status'];

// enumerated properties
exports.enums = [
 {status:['pending', 'active', 'suspended', 'closed']}
];

// default values
exports.defs = [
 {name:'status', value:'active'}
];
```

Note that in my implementation, I continue to use the DARRT [id](#) value instead of the design's [ratingId](#). I also dropped the [dateRequested](#) in the design and will just use the DARRT value [dateCreated](#). DARRT will always generate those values, so we might as well use them.

## actions.js

The `actions.js` file holds all the functions that are associated with the API service. In our case, we only need to modify the file a little bit. For our `credit-check` API, we'll need the following functions:

```
module.exports.home = function(req,res) {...}
module.exports.create = function(req,res) {...}
module.exports.list = function(req,res) {...}
module.exports.item = function(req,res) {...}
module.exports.form = function(req,res) {...}
```

Since this service also needs to generate a credit rating value for each request, we'll need an internal function that (for now) generates a random number between 1 and 9 as a `ratingValue`. I've added that to the top of my `actions.js` file. It looks like this:

```
// compute rating value for each request
// use this internal function as a stub for now
function computeRating(body) {

 body.ratingValue = (Math.floor(Math.random() * 9) + 1).toString() ;

 return body;
}
```

Then I added a call to this function in the `create` function. That whole function now looks like this:

```
// credit check create
module.exports.create = function(req,res) {
 return new Promise(function(resolve,reject) {
 if(req.body) {
 var body = req.body;
```

```

// go get this companies' credit rating
body = computeRating(body)

resolve(
 component(
 {
 name:object,
 action:'add',
 item:body,
 props:data.props,
 reqd:data.reqd,
 enums:data.enums
 }
)
);
}
else {
 reject({error:"invalid body"});
}
));
);

```

You can look at the [actions.js](#) file in the [completed](#) folder of the source code for this chapter for details on the rest of the changes to the default file.

### [resources.js](#)

The [resources.js](#) file defines all the HTTP resources for your [credit-check](#) API. By consulting the [credit-check-http.png](#) diagram in your project's [/assets/](#) folder, you can see the four documented HTTP resources:

- Home (GET /)
- CreditCheckList (GET /list/ )
- CreditCheckItem (GET /list/:id)
- CreditCheckForm (GET, POST /form/)

Those are the entries we'll need to see in the [resources.js](#) file:

```

// home
router.get('/',function(req,res){...});

```

```

// credit check list
router.get('/list/',function(req,res){...});

// credit check item
router.get('/list/:id', function(req,res){...});

// credit check form READ
router.get('/form/', function(req,res){...});

// credit check form CREATE
router.post('/form/', function(req,res){...});

```

All the functions in this file look pretty much the same. Here's the complete code for the [home](#) resource:

```

// home
router.get('/',function(req,res){
 utils.handler(req,res,actions.home,"home",
 {
 metadata:metadata,
 templates:templates,
 forms:forms,
 filter:"home"
 }
)
});

```

You can check out the [resources.js](#) file in the [/completed/](#) folder of the source code for details on the rest of the functions in that file.

### [representations.js](#)

For this exercise, you don't need to make any changes to the default [representations.js](#) document. It's already set up to support several useful output formats.

### [transitions.js](#)

The transitions file holds the definition of all the request forms for the API. Remember most APIs today skip the step of explicitly defining transitions (and skip returning them in responses). However, I always include them so

that developers who prefer fully described API responses will be able to ask for them using registered hypermedia types.

For this exercise, we only need six transitions:

- **self** : what it takes to refresh the current page
- **home** : the root response for this API (includes the **list** transition)
- **list** : to request a list of records for this API
- **read** : to request a single record
- **form** : to view the form needed to create a new record
- **create** : to actually create a new record

The following shows two transition definitions: one for the **list** action and one for the **create** action:

```
// the list of all credit check records
{
 id:"list",
 name:"list",
 href:{fullhost}/list/",
 rel:"collection credit",
 tags:"collection credit home list item",
 title:"List",
 method:"GET",
 properties:[]
},
// create a new credit-check record
{
 id: "create",
 name: "create",
 href: "{fullhost}/form/",
 rel: "create-form credit",
 tags:"collection credit list item",
 title: "Create",
 method: "POST",
 properties: [
 {name:"id",value:"{makeid}"},
 {name:"status",value:"active"},
 {name:"companyName",value:""}
]
}
```

```
]
}
```

You can review the rest of the forms in the [transitions.js](#) file in the `/completed/` folder of the source code.

## Run the Validation Script

The final step in the coding process is to run your validation script. This script contains the “happy path” steps for validating your API implementation.

For this exercise, look in the [tests](#) subfolder of the project for a happy-path script. The start of the script looks like this:

```

network vars
svr="http://localhost:8181"
acc="accept:application/forms+json"

credit check data
cid="r4t5y6u7"
name="Ems Corp"
status="active"

heading
clear
echo "Credit-Check Happy Path Script"
date
echo

home read
echo Home $svr/
curl $svr/ -H "$acc" | jq ".."
echo

list read
```

```
echo List $svr/list/
curl $svr/list/ -H "$acc" | jq "."
echo
```

You'll need to change the `cid` variable value in order to get the script to fully run. And once you run it, you'll need to keep changing the `cid` value because each record ID value for this API needs to be unique. You'll also see the script line that *creates* a new credit-check record here:

```

make a credit-check request
echo Write $svr/form/
curl $svr/form/ -X POST \
-d "id=$cid&companyName=$name&status=$status" -H $acc | jq "."
echo
```

Assuming your script runs successfully, you've properly built your [credit-check](#) API and are ready to move on to the next steps: testing, securing, and deploying your API.

# Solution for Chapter 9: Testing APIs

In the exercise for this chapter (Chapter 9, [Testing APIs](#)), your task is to create a Postman test collection for your `credit-check` API project and write two tests: a “happy path” test for the Home resource (<http://localhost:8181/>) and a “sad path” test for the From resource (<http://localhost:8181/form/>). You were supplied with extensive instructions for each test.

For extra credit, you were given the task to export the completed tests and use the `newman` command-line utility to run the tests locally.

Note: Before you start this exercise, make sure your local version of the `credit-check` API that you built in the last chapter is up and running. You can use the copy of that service in the [/testing/exercises/before/credit-check](#) folder of the source code associated with this book.

## Create the Credit-Check Collection in Postman

First, we need to create a new test collection in Postman. Let’s name it “Credit-Check.” In the Postman client, click the “+ New Collection” link in the left sidebar, enter “Credit-Check” in the Name field, and then press the Create button in the lower-right corner of the dialog box.

## Add the Home Test

Click the three dots near the collection name and pick Add Request from the drop-down menu. Enter Home as the Request Name and press the Save button. Next, click the Home test in the left sidebar to bring it up to the main editor view.

Now we can add the Request URL. Enter “localhost:8181/” and press the Send button. You should get a response body back that looks like this:

{

```

"home": [
 {
 "id": "list",
 "name": "credit-check",
 "rel": "collection credit-check",
 "href": "http://localhost:8181/list/"
 }
]
}

```

## Write the Happy-Path Home Test

Now we're ready to write our test script. Click the Tests tab underneath the Request URL field to bring up the blank editor and enter the following script:

(Remember, you can find all the source code for this solution in the code folder ([code/testing/exercise/](#)) associated with this chapter.)

```

/*****
CREDIT CHECK HOME
****/

// protocol
pm.test('Status is 200', function() {
 pm.expect(pm.response.code).to.equal(200);
});

pm.test('Content-Type includes application/json', function() {
 var hdr = pm.response.headers.get('content-type');
 pm.expect(hdr).includes('application/json');
});

// structure
var body = pm.response.json();
var obj = body.home.find(x => x.id === 'list');

pm.test('root has a valid home object', function() {
 pm.expect(body.home).to.be.an('array');
 pm.expect(obj).has.property('id');
 pm.expect(obj).has.property('name');
});

```

```

pm.expect(obj).has.property('href');
pm.expect(obj).has.property('rel');
});

// values
pm.test('home.id includes list', function() {
 pm.expect(obj.id).includes('list');
});
pm.test('home.name includes credit-check', function() {
 pm.expect(obj.name).includes('credit-check');
});
pm.test('home.rel includes credit-check', function() {
 pm.expect(obj.rel).includes('credit-check');
});
pm.test('home.rel includes collection', function() {
 pm.expect(obj.rel).includes('collection');
});

```

Save the test and press the Send button again to confirm all the tests pass.

## Create the Form Test

To add our next test, click the three dots near the collection name and pick Add Request from the drop-down menu. Enter “Form (missing properties)” as the Request Name and press the Save button. Next, click the FORM test in the left sidebar to bring it up to the main editor view.

Now we can add the Request URL. Enter “localhost:8181/” and change the GET option from the drop-down menu next to Request URL to POST. Then press the Send button. You should get a response body back that looks like this:

```
{
 "error": [
 {
 "type": "error",
 "title": "Missing companyName",
 "detail": "Missing companyName",
 "status": "400",
 "instance": "http://localhost:8181/form/"
```

```
 }
]
}
```

## Write the Sad-Path Form Test

Now it's time to write our sad-path test to confirm that the API rejects any attempts to add a company record when required fields are missing. Using the test rules from the exercise as a guide, your results should look similar to the script that follows. (A copy of this script is available in the [/testing/exercise/before/](#) folder in the source code for this book.)

```

CREDIT CHECK FORM (missing company)

```

```
// protocol
pm.test('Status is 400', function() {
 pm.expect(pm.response.code).to.equal(400);
});

pm.test('Content-Type includes application/problem+json', function() {
 var hdr = pm.response.headers.get('content-type');
 pm.expect(hdr).includes('application/problem+json');
});

// structure
var body = pm.response.json();
var error = body.error

pm.test('root has a valid error object', function() {
 pm.expect(error).to.be.an('array');
 error.forEach(function(obj) {
 pm.expect(obj).has.property('type');
 pm.expect(obj).has.property('title');
 pm.expect(obj).has.property('detail');
 pm.expect(obj).has.property('status');
 pm.expect(obj).has.property('instance');
 });
});
```

```
// values
var obj = error[0];
pm.test('error.type includes error', function() {
 pm.expect(obj.type).includes('error');
});
pm.test('error.title includes Missing companyName', function() {
 pm.expect(obj.title).includes('Missing companyName');
});
pm.test('error.detail includes Missing companyName', function() {
 pm.expect(obj.detail).includes('Missing companyName');
});
pm.test('error.status includes 400', function() {
 pm.expect(obj.status).includes('400');
});
```

After adding the tests to the Postman client, press the Save button and then Send to confirm all the tests pass.

## Export the Test Collection

To export your test collection from Postman onto your local drive, click the three dots near the collection name in the Postman UI sidebar and select Export from the drop-down menu. Make sure the “Collection v2.1” radio button is selected and press the Export button that appears in the lower-right corner of the dialog box. Then save the collection to your local drive. The default name for your collection will look something like [Credit-Check.postman\\_collection.json](#).

## Run the Exported Tests with Newman

Finally, you can run the exported test collection with the [newman](#) utility by opening up a command window and navigating to the folder that holds your exported collection document. Then type the following:

```
mca@mamund-ws:~$ newman run Credit-Check.postman_collection.json -r htmlextra
```

This will execute both your Home and Form tests and write out an HTML report that will appear in the [code/testing/newman/](#) folder.

That wraps up the testing portion of our project. Next, you'll be ready to add security features and then deploy the API to a public server.

# Solution for Chapter 10: Securing APIs

The exercise for this chapter (Chapter 10, *Securing APIs*) focuses on using Auth0 to define your API security, collect the access control parameters, modify your API source code, and then test the results.

## Define Your API in Auth0

The exercise instructions included the name of the new API security definition (**bigco-credit-check**). To create this definition, you need to log in to the <http://auth0.com> website and navigate to the dashboard page. There you can select the APIs option in the left navigation pane and, when the list of APIs appears, click the Create API button that appears in the upper-right corner of the screen. This brings up the New API dialog, where you can enter “bigco-credit-check” into the Name field. You also need to enter your API identifier (for example, <http://api.mamund.com/bigco-credit-check>). Once both values are supplied, click the Create button at the bottom of the dialog. That completes the definition and takes you to the new landing page for that API.

## Collect Your API's Access Control Parameters

The next step is to collect the five important access control parameters (Name, Identifier, ClientID, ClientSecret, and Domain) for your API. The Name and Identifier values are on the Settings tab of the API’s landing page. The other three values (ClientID, ClientSecret, and Domain) are on the API’s application page. You can find this page by clicking the Applications option in the left navigation pane of the dashboard and then selecting the **bigco-credit-check** application from that list. When you select it, you’ll be taken to the landing page where the ClientID, ClientSecret, and Domain are displayed.

You need to collect all five values and write them into the proper spots in the `auth0.env` file in your project's `/security/` folder. That file looks something like this:

```
#####
auth0-util environment vars
#####

name="credit-check"

id=<CLIENT-ID>
secret=<CLIENT-SECRET>

url="https://<DOMAIN>/oauth/token"

jwksuri="https://<DOMAIN>/.well-known/jwks.json"
audience=<IDENTIFIER>
issuer=<DOMAIN>

EOF
```

## ENV or TXT?



Note that in the code folder for this project, all the files that end in `.env` are actually saved to the project as `.txt` files. That's because the ENV files are automatically ignored when checking the project into source control. This is a safety feature to make it hard for you to accidentally check your API's secret keys into source control, where others could see them. Be sure to copy the `auth0.txt` file to `auth0.env` on your local machine when you're completing this project.

With the values copied to your `auth0.env` file, you can run the `./auth0-token.sh` script to retrieve a fresh access token from Auth0. It will be written to the `auth0-token.env` file. You can open that file and copy the `access_token` value

into your `curl-auth.env` file. You can also copy-paste that token into the <http://jwt.io> website to validate the contents of that token.

## Update the credit-check-secure NodeJS Project

Now you are ready to update your NodeJS API project. First, be sure to add all the security-related code modules to your project using npm. To do that you can move to the project folder on disk and type the following into the command line:

```
mca@mamund-ws:~/company-secure$ npm install -s jwks-rsa jsonwebtoken \
express-jwt express-jwt-authz
```

Next, open up the `index.js` file in the `credit-check-secure` project folder and update that file to reference the `api-auth.js` code file from the `/DARRT/lib` folder and add the security middleware into the NodeJS Express pipeline by adding the following lines to your `index.js` file:

```

// start of auth support
var secure = require('./darrt/lib/api-auth.js');
app.use(secure.jwtCheck);
// end of auth support

```

Finally, open the `/DARRT/lib/api-auth.js` file and update the `auth` object values to match the access control parameters you pulled from the Auth0 website in the previous step.

Your code should look like this:

```
// auth variables
var auth = {};
auth.cache = true;
auth.rateLimit = true;
auth.requestsPerMinute = 5;
auth.jwksUri = '<DOMAIN>/well-known/jwks.json';
auth.audience = '<IDENTIFIER>';
```

```
auth.issuer = '<DOMAIN>';
auth.algorithms = ['RS256'];
```

Be sure to copy your API definition's access control parameters where indicated in the code snippet above.

Finally, save all your changes to the project and check them into the Git repository.

## Test Your API Security

Now you can try accessing your API to validate your security changes. First, try using a simple [curl http://localhost:8181/](http://localhost:8181) call (without a security token) to confirm that your API call gets an HTTP 401 status code response. Your response should look like this:

```
mca@mamund-ws:~/company-secure$ curl localhost:8181/
{
 "type": "Error",
 "title": "UnauthorizedError",
 "detail": "No authorization token was found",
 "status": 401
}
mca@mamund-ws:~/company-secure$
```

Next, use the [curl-auth.sh](#) utility (with the access token from the previous step and other appropriate configuration settings) to make the same call. Your [curl-auth.env](#) file should look something like this (with your new JWT pasted in where indicated):

```
#####
auth-test variables
#####

url="http://localhost:8181/"
method=GET
accept="application/json"

token=<VALID-JWT-ACCESS-TOKEN>"
```

Now when you execute the command [./curl-auth.sh](#), you should get the root response as expected without any errors. It should look like the example shown here:

```
mca@mamund-ws:~/security$./curl-auth.sh

OAuth Request Utility
=====
Sun Apr 26 14:54:57 EDT 2020

...: requesting GET http://localhost:8181/
{
 "home" :
 [
 {
 "id" : "list"
 , "name" : "credit-check"
 , "rel" : "collection credit-check"
 , "href" : "http://localhost:8181/list/"
 }
]
}
mca@mamund-ws:~/security$
```

Assuming you've made it this far, you now have designed, built, tested, and secured your NodeJS API! The next big thing is to release it into production. We'll do that in the next solution.

# Solution for Chapter 11: Deploying APIs

Most of the work of deploying our API project into production is handled by the Heroku CLI Client we reviewed in the chapter. However, we need to *manually* deploy the application (in other words, type in the proper command in the command window) in order to complete the deployment process. That means our current level of automation is at *continuous delivery*. (See [Continuous Delivery](#), for details.)

This exercise challenge is for you to elevate your deployment automation to *continuous deployment*—to automate that last step of deploying into production. To do that, you were given a hint to modify the `norman` test runner tool we used in Chapter 9, [Testing APIs](#), to not only automate the testing but also automate the final production deployment. You even got the hint that you only needed to modify a single line of the test runner script to do that.

Another copy of the source code for the `norman` utility can be found in the `code/deploying/exercise/before/norman` folder in the code files for this book. If you open the `index.js` file in that folder, you'll find the following:

```
newman.run(options, function (err, summary) {
 if (err) { throw err; }

 if (summary.run.failures.length!==0) {
 console.log("One or more tests failed.")
 }
 else {
 console.log('collection run complete!');
 // add deployment command here
 shell.exec("insert proper git command here");
 }
});
```

In the current script, when all tests pass successfully, a message appears on the command line (“collection run complete!”) and the script stops.

However, as you can see in the source code, you can modify a commented-out line to update the test runner to also deploy the code to production when all tests pass.

That code looks like this:

```
newman.run(options, function (err, summary) {
 if (err) { throw err; }
 if (summary.run.failures.length!==0) {
 console.log("One or more tests failed.")
 }
 else {
 console.log('collection run complete!');
 // add deployment command here
 shell.exec("git push heroku master");
 }
});
```

Note that the only change here is the addition of the command `git push heroku master`. That's the same command you type manually to kick off the deployment process on Heroku's servers. Now we're just getting the `norman` utility to do that last step for us!

Once you've updated the source code and saved it, you need to run the `npm install` command again to update your copy of the `norman` command line utility.

Now you can move to the `code/deploying/before/onboarding` folder and do the following:

First, use npm to update the local project source code: (Note: Make sure you have adequate access rights to update the local file tree.)

```
npm update
```

Next, you need to make sure the version of your `onboarding-api` project is set up to deploy to Heroku. If you're using the project in the `code1` folder,

you'll need to create a new Heroku app. This was covered in detail in [Create a New Heroku App](#).

After you have successfully created a new Heroku app and associated that with your copy of your local git repo, you're ready to run the test and, if all goes well, automatically deploy it to the cloud.

Next, you can start the local version of your onboarding service:

```
npm start
```

Once that service is up and running, you can execute the `norman` test runner to validate your build against your test suite:

```
`norman assets/api-onboarding.postman_collection.json`
```

Assuming both your `norman` utility code and your API project code are clean and all tests pass, you'll see the updated test runner automatically kick off the deployment to Heroku's servers. If all goes well, you should be able to view your newly deployed app live in the cloud by following the URL supplied by Heroku.

And that's it. Congratulations! You have moved your API project from the continuous delivery stage to the continuous deployment stage.

---

## Footnotes

[98] <https://app.apiary.io>

## Appendix 3

### API Project Assets Checklist

---

One of the important takeaways from this book is that every API project should have a consistent set of elements—what I call *project assets*. This appendix provides a condensed list of those important API project assets that you can use to help ensure that you’ve covered the important aspects of your API project (“Do you have a coherent API story document?”) and that you’ve covered all the key tasks for a successful API (“Did you write, and run, your API tests?”). I also include some commentary and pointers to various places in the book where details can be found about each asset.

## Using the API Project Assets Checklist

If you are a stand-alone “full stack” developer working on the project, I suggest you work through this assets checklist in the order presented. As you gain experience with the list, you might change up the order in a way that works better for you. For example, you may write your tests earlier and create smaller prototypes that you can test before completing all your design elements.

If you’re working on a team that divides responsibilities between roles—for example, designers, developers, and operations staff—then you can arrange the list in a way that allows all roles to operate in parallel. I’ve worked on several teams where the person in charge of deployment writes an initial working script the first day of the project and continues to update and modify it as needed.

No matter your method of working, you should have an API assets list that’s shared with all members working on the project and that’s referred to frequently throughout the life of the API.

## Notes on the Assets Checklist

As a rule, each asset should be something you can create in less than a day. If your API is extensive (dozens of actions), you can break the API into smaller subsections and produce assets for each subset. For example, the API Definition document and API Test collection may take several days to complete, but it should not take more than a workweek. If that seems unlikely, break the definition and tests into smaller collections and complete those as needed.

You'll notice that each asset in the list that follows is marked with MUST, SHOULD, and MAY references as well as brief notes about formats and recommendations. These reflect my common experience, and YMMV (your mileage may vary). Feel free to modify the recommendations to fit your working experience, needs, and company culture.

Finally, you'll find pointers to other sections in the book that discuss the particular asset and/or recommendation in greater detail. Over time, you may find additional external references to add to your customized checklist too. When authoring your API project assets checklist, it's a good idea to include reference articles, tool documentation, blogs, and other forms of background information. This helps people see the thinking behind the recommendations and encourages others to seek out and amend the references over time.

# Your API Project Assets Checklist

The following is the general assets list I use to start an API project.

## 1. API story document: *MUST*

- See [Writing the API Story](#)
- Recommend using an approved template/pattern for stories
- As a — I want to — so that —
- Purpose, Data, Actions, Rules, Processes
  - Recommend collecting supporting documents (forms, write-ups, wireframes, and so on)
  - If available, include output from event-storming or other sessions

## 2. API diagram: *MUST*

- See [Creating Your Sequence Diagram](#)
- Recommend WSD format
- Sequence diagrams suggested, but any consistent method accepted

## 3. API vocabulary list: *SHOULD*

- See [Identifying Your API Descriptors](#)
- Include a complete list of all properties and actions used in the life of the API
- Recommend tying these directly to existing dictionaries of approved terms

## 4. API profile document: *SHOULD*

- See [Describing Your API with ALPS](#)
- Use ALPS, DCAP, or other machine-readable implementation-agnostic format

## 1. API sketches: *MAY*

- See [\*Sketching APIs with Apiary Blueprint\*](#)
- Recommend Apiary Blueprint (markdown) format

## 1. API definition: *MUST*

- See [\*Translating Your API Design into HTTP\*](#)
- Use OpenAPI, AsyncAPI, RAML, SOAP, and so on, as required by implementation
- Recommend writing the definition files yourself
- May be autogenerated from the API profile
- Not recommended to autogenerate from existing code

## 2. API source code: *MUST*

- See [\*Coding APIs with NodeJS and DARRT\*](#)
- Recommend using a framework or library that makes it easy to convert your assets (profile/definition) into code
- Not recommended to use a code generator, as it rarely does an effective job at converting definitions into quality code

## 3. API documentation: *MUST*

- See [\*Generating Your API Documentation\*](#)
- Recommend (at minimum) generating code-centric documentation automatically (for example, Swagger-style docs)
  - This is the documentation for developers who'll use your API to solve their problems
- Additional (optional) documentation can include:
  - “Getting Started” (for quick start)

- “How Can I … ?” (to help developers solve common problems)
- “Sample Solutions” (for teaching developers how to apply the API to a solution)
- “Concepts” (for sharing background and conceptual basis for the design and implementation of the API)
- Recommend writing the *minimum* amount of documentation needed to meet community needs:
  - For local internal teams you MAY get by with only the code-centric docs
  - For remote internal teams you SHOULD at least add “How Can I…?”
  - For external teams (partners) you SHOULD add “Sample Solutions”
  - For inexperienced, remote, external audiences, you MAY add the “Concepts”
- Remember: The more detailed your API docs, the more work you’ll need to do to maintain them over time

#### 4. API mocks: *MUST*

- See [Mocking Your API](#)
- Recommend generating the mock directly from the API definition files
  - These are used for testing purposes (both automated and human experience testing)

#### 5. API tests: *MUST*

- See [Using Postman for API Testing](#)

- Recommend both happy path (returns HTTP 200) and sad path (returns HTTP 400)
- Recommend using Postman/Newman to write/script testing
- Can also use SRTs (simple request tests) using curl, WGET, and so on

## 6. API security: *MUST*

- See [Implementing API Security with Auth0](#)
- Include a document that outlines the security requirements for the API
- Recommend writing this as part of your API story and validating that through testing

## 1. API deployment: *MUST*

- See [Git-Based Deployment on Heroku](#)
- Recommend using some form of automated deployment
- At minimum, run your test suite both before and after deployment. Ideally, do the following:
  - Run your scripted test suite against the local instance (deployment candidate)
  - If all local tests pass, run a script to deploy the candidate to the target (into production, staging, and so on)
  - If deployment is successful, run your scripted test suite against the deployed instance
  - If all deployed tests pass, make any changes needed to update routers and gateways to point to the new instance
- Be ready to back out changes at any step along the way

Copyright © 2020, The Pragmatic Bookshelf.

Thank you!

How did you enjoy this book? Please let us know. Take a moment and email us at support@pragprog.com with your feedback. Tell us your story and you could win free ebooks. Please use the subject line “Book Feedback.”

Ready for your next great Pragmatic Bookshelf book? Come on over to <https://pragprog.com> and use the coupon code BUYANOTHER2020 to save 30% on your next ebook.

Void where prohibited, restricted, or otherwise unwelcome. Do not use ebooks near water. If rash persists, see a doctor. Doesn’t apply to *The Pragmatic Programmer* ebook because it’s older than the Pragmatic Bookshelf itself. Side effects may include increased knowledge and skill, increased marketability, and deep satisfaction. Increase dosage regularly.

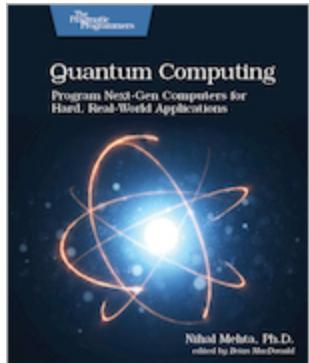
And thank you for your continued support,

Andy Hunt, Publisher

## You May Be Interested In...

*Select a cover for more information*

### Quantum Computing

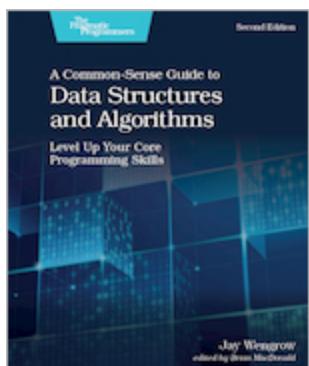


You've heard that quantum computing is going to change the world. Now you can check it out for yourself. Learn how quantum computing works, and write programs that run on the IBM Q quantum computer, one of the world's first functioning quantum computers. Develop your intuition to apply quantum concepts for challenging computational tasks. Write programs to trigger quantum effects and speed up finding the right solution for your problem. Get your hands on the future of computing today.

Nihal Mehta, Ph.D.

(580 pages) ISBN: 9781680507201 \$45.95

### A Common-Sense Guide to Data Structures and Algorithms, Second Edition



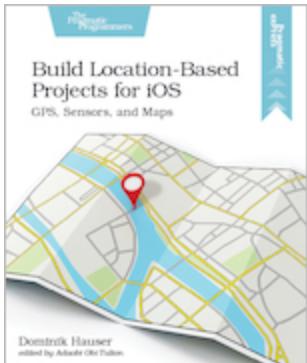
If you thought that data structures and algorithms were all just theory, you're missing out on what they can do for your code. Learn to use Big O Notation to make your code run faster by orders of magnitude. Choose from data structures such as hash tables, trees, and graphs to increase your code's efficiency exponentially. With simple

language and clear diagrams, this book makes this complex topic accessible, no matter your background. This new edition features practice exercises in every chapter, and new chapters on topics such as dynamic programming and heaps and tries. Get the hands-on info you need to master data structures and algorithms for your day-to-day work.

Jay Wengrow

(506 pages) ISBN: 9781680507225 \$45.95

## Build Location-Based Projects for iOS



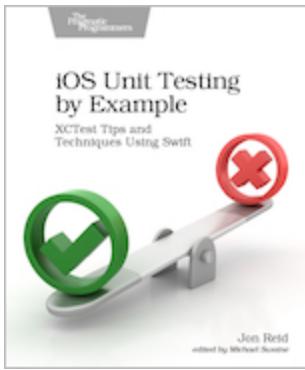
Coding is awesome. So is being outside. With location-based iOS apps, you can combine the two for an enhanced outdoor experience. Use Swift to create your own apps that use GPS data, read sensor data from your iPhone, draw on maps, automate with geofences, and store augmented reality world maps. You'll have a great time without even noticing that you're learning. And even better, each of the projects is designed to be extended and eventually submitted to the App Store. Explore, share, and have fun.

Dominik Hauser

(154 pages) ISBN: 9781680507812 \$26.95

## iOS Unit Testing by Example

Fearlessly change the design of your iOS code with solid unit tests. Use Xcode's built-in test framework XCTest and Swift to get rapid feedback on all your code — including legacy code. Learn the tricks and

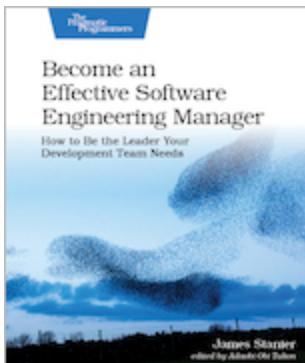


techniques of testing all iOS code, especially view controllers (UITableViewController), which are critical to iOS apps. Learn to isolate and replace dependencies in legacy code written without tests. Practice safe refactoring that makes these tests possible, and watch all your changes get verified quickly and automatically. Make even the boldest code changes with complete confidence.

Jon Reid

(300 pages) ISBN: 9781680506815 \$47.95

## Become an Effective Software Engineering Manager



Software startups make global headlines every day. As technology companies succeed and grow, so do their engineering departments. In your career, you'll may suddenly get the opportunity to lead teams: to become a manager. But this is often uncharted territory. How do you decide whether this career move is right for you? And if you do, what do you need to learn to succeed? Where do you start? How do you know that you're doing it right? What does "it" even mean? And isn't management a dirty word? This book will share the secrets you need to know to manage engineers successfully.

James Stanier

(396 pages) ISBN: 9781680507249 \$45.95

## Build Websites with Hugo

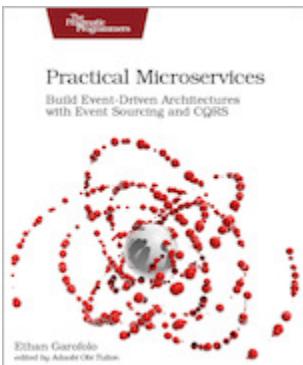


Rediscover how fun web development can be with Hugo, the static site generator and web framework that lets you build content sites quickly, using the skills you already have. Design layouts with HTML and share common components across pages. Create Markdown templates that let you create new content quickly. Consume and generate JSON, enhance layouts with logic, and generate a site that works on any platform with no runtime dependencies or database. Hugo gives you everything you need to build your next content site and have fun doing it.

Brian P. Hogan

(154 pages) ISBN: 9781680507263 \$26.95

## Practical Microservices

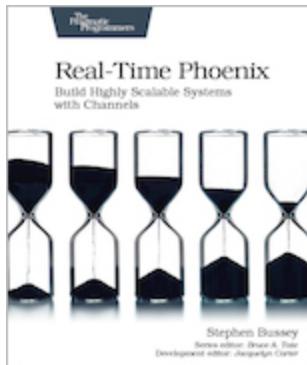


MVC and CRUD make software easier to write, but harder to change. Microservice-based architectures can help even the smallest of projects remain agile in the long term, but most tutorials meander in theory or completely miss the point of what it means to be microservice based. Roll up your sleeves with real projects and learn the most important concepts of evented architectures. You'll have your own deployable, testable project and a direction for where to go next.

Ethan Garofolo

(290 pages) ISBN: 9781680506457 \$45.95

## Real-Time Phoenix

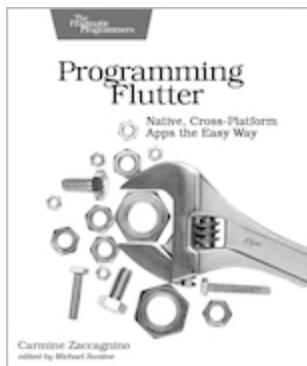


Give users the real-time experience they expect, by using Elixir and Phoenix Channels to build applications that instantly react to changes and reflect the application's true state. Learn how Elixir and Phoenix make it easy and enjoyable to create real-time applications that scale to a large number of users. Apply system design and development best practices to create applications that are easy to maintain. Gain confidence by learning how to break your applications before your users do. Deploy applications with minimized resource use and maximized performance.

Stephen Bussey

(326 pages) ISBN: 9781680507195 \$45.95

## Programming Flutter



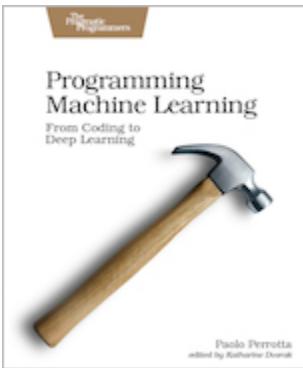
Develop your next app with Flutter and deliver native look, feel, and performance on both iOS and Android from a single code base. Bring along your favorite libraries and existing code from Java, Kotlin, Objective-C, and Swift, so you don't have to start over from scratch. Write your next app in one language, and build it for both Android and iOS. Deliver the native look, feel, and performance you and your users expect from an app written with each platform's own tools and languages. Deliver apps fast, doing half the work you were

doing before and exploiting powerful new features to speed up development. Write once, run anywhere.

Carmine Zaccagnino

(368 pages) ISBN: 9781680506952 \$47.95

## Programming Machine Learning



You've decided to tackle machine learning — because you're job hunting, embarking on a new project, or just think self-driving cars are cool. But where to start? It's easy to be intimidated, even as a software developer. The good news is that it doesn't have to be that hard. Master machine learning by writing code one line at a time, from simple learning programs all the way to a true deep learning system. Tackle the hard topics by breaking them down so they're easier to understand, and build your confidence by getting your hands dirty.

Paolo Perrotta

(340 pages) ISBN: 9781680506600 \$47.95

---