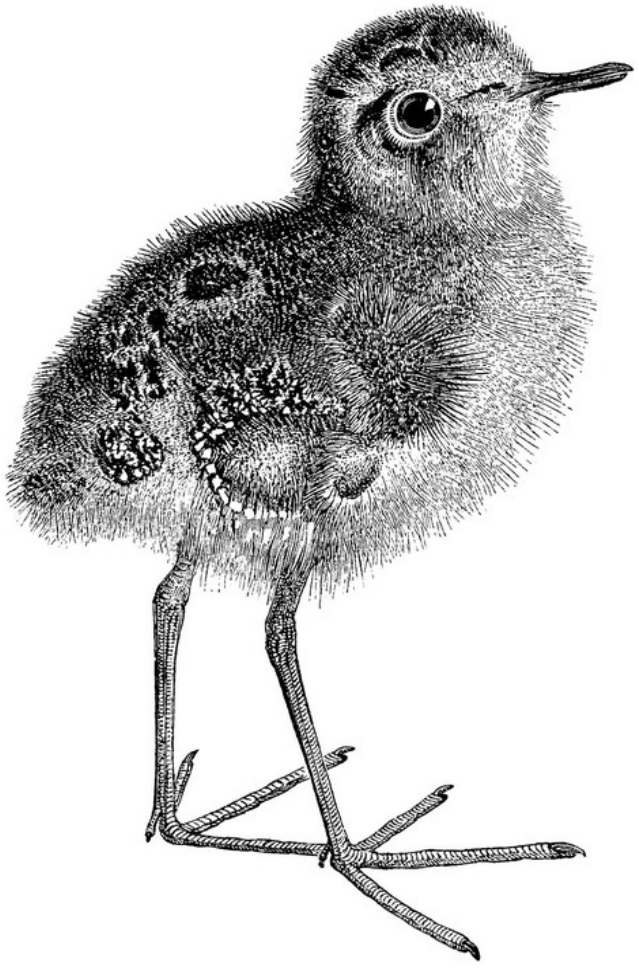
O'REILLY®

JavaScript
The Hidden Parts

Building More Performant, Flexible, and Maintainable Applications



Early Release
RAW & UNEDITED

Milecia McGregor

JavaScript: The Hidden Parts

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Building More Performant, Flexible, and Maintainable Applications

Milecia McGregor

Beijing • Boston • Farnham • Sebastopol • Tokyo

JavaScript: The Hidden Parts

by Milecia McGregor

Printed in the United States of America.

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

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

- Acquisitions Editor: Amanda Quinn
- Development Editor: Sarah Grey
- Production Editor: Christopher Faucher
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery

- June 2023: First Edition

Revision History for the Early Release

- 2022-04-26: First Release
- 2022-08-29: Second Release
- 2022-11-15: Third Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098122256 for release details.

sibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-12219-5

Chapter 1. Standard Project File Structures

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at milecia.mcgregor@gmail.com.

The longer you work as a JavaScript developer, the more project structures you'll get exposed to and you'll start to see some common trends. Not every organization follows the exact same conventions, but there are a few key things you should be able to find within five minutes of cloning a repo from GitHub.

Regardless of if you're working with legacy applications, a brand new start-up project, or something in between, you can make changes to the folder/file structure and conventions. This is one of the first hidden things.

Changing the file structure of a project can make it easier for developers to find functionality faster and it can lead to better long-term maintainability because everything follows a more standard structure. Sometimes projects get messy because there are so many developers in and out of a project that they do things their way as long as tickets get completed. Leaving a messy file structure with non-uniform conventions leads to technical debt, like making it harder to track down bugs, figuring out where to implement new code, and even understanding which packages are being used.

Many times we inherit these projects that have grown and morphed into something that's hard to manage. As one of the developers working on the project, you have the power to make changes to existing code organization. You should feel empowered to bring up any questions or concerns to others on your team and get everyone on the same page on what to do going forward and how to handle technical debt.

General Organization

It doesn't matter if you're working on the front-end or the back-end, a project should be organized in a way that makes it clear where everything is to anyone new jumping into the project. Your code organization acts as a form of documentation. This is the first thing developers see when they start on a project.

This is one of the most subtle ways that projects can get out of hand. When you go into any project you haven't worked on, do a quick audit of the folder names and file names. This will help you figure out where things are faster than running the code.

The project README

If you're new to a group and there isn't a README for the project, this is a great suggestion to make. This is another hidden thing that tends to pop up later in projects. Seniors try to make the project accessible to all developers that need to work on it and that starts with including a good README.

This README should explain the structure of the project, what parts of the project the folders contain, any naming conventions, how to start the project, and other things that would help someone new to a project. Once this is created, it's important to update it at least once

every 3-6 months to make sure it doesn't fall behind the current state of the project.

You don't have to wait to have a working project to start writing this document. One thing that's important for senior developers to know is how to document things. You want to make sure that when you and half of the team take off at the same time, the new person isn't completely lost on how things work and they know where they can go to find info on common procedures.

General README template

Good README documents have a few common elements. You can change the titles of the sections as you need, but the info they contain should remain roughly the same.

How to start the project

This should contain all of the info for any dependencies that need to be installed, any databases that need to be set up, and any services that a developer will need access to. When you're including things like `.env` files, make sure you **do not** add any secrets or credentials to the repo's Git history. You don't want attackers to find these things in your commit history.

File/folder conventions

You should let developers know what each folder contains. It doesn't have to be super long, just as long as it gets the point across. For example, you might have a folder in your project named `components` and this is where you store the components for your front-end that get reused in different views.

Every developer has their own style so defining what type of casing, spacing, and coding formats you use will help keep codebase standarized. This could even be as quick as linking to a linting config file or just stating that you use camelcase everywhere with single quotes. Just have something that keeps all of the code in a unified format.

How to handle common issues

There are always some weird things that pop up consistently in starting new projects. Maybe you have to set up services in a certain order or you need someone to give you a level of permission to access resources. These are quick FAQs that should be addressed as they come up from onboarding new developers.

You can add on to these sections as you see fit for your project, but these should definitely be there. You should also include anything specific to your organization, like contact info for people over various systems or key stakeholders, where to get sensitive credentials to get the app running, and how to get through common setup issues.

It’s important to make sure that you keep this document up to date as things in the project change. Anytime you add new services or change a process flow, it’s a good time to update the README. Let’s go over a few different folder and file conventions.

Folder structures

There are an amazing number of ways you’ll see codebases organized. I’ve seen some folder structures that basically taught me how the project worked and I’ve seen some that have numbered folders. The range that you’ll see throughout your career will leave you with some experiences that will definitely stretch your brain.

That’s why it’s important for you to feel empowered to go in and standardize a project’s structure if you see that it’s lacking or to create the standard for a new project. I want to show you a few of the most common folder structures that I’ve seen in the wild. We’ll get into the differences between front-end and back-end a little later, but here are the main parts.

Folder naming conventions

Some of the seemingly obvious things are the easiest to miss. You want to make sure things like the names are consistent among the folders. So from the beginning, you should decide or find out the little

things like whether or not the folder names should start with a capital letter. Or if you want to have a certain prefix or suffix for the folder names.

There is no right or wrong approach to this. That's why you'll hear senior developers commonly say, "It depends." Everything comes with a trade-off. The main thing is to make sure everyone is on the same page. If you aren't quite sure where to start here are a few naming conventions you can follow.

Name your folders for the type of functionality they contain.

The name of your folders should give a clue to the code that a developer can expect to find inside. For example, if you have a folder that holds UI components for your front-end, a good name may be `components`. Nothing earth-shattering here, but it's surprising how sometimes people can complicate things.

Keep your folder names simple.

That brings up this point. The name of a folder doesn't have to be something unique to every codebase you've ever seen. If you always have a `services` folder on your back-end that holds the endpoints you're responsible for, it's ok to use that in other projects if it makes

sense. When you start getting too specific with names, it can lead to having more folders than you need and it makes it can make it messy to find functionality.

Choose a casing and stick to it no matter what.

This is a petty point that comes up in a number of development teams. In reality, it doesn't really matter how you do your casing as long as it's consistent. Just make sure that everyone follows the choice in order to keep the codebase looking uniform. I remember one project I worked on where almost every folder had different casing and similar variable names and that made it take forever to debug the simplest issues because I kept getting confused on which variables followed which types of casing.

These are a few points to keep in mind when you're doing a technical debt audit or when you're looking for small improvements to make in a project.

Common organization

One thing that intimidates some mid-level developers is getting that first blank project repo and being responsible for filling it in. An approach you can take to make this less daunting is to setup a place to have template files you can copy and paste in your different folders.

That way developers don't have to start from scratch every time they need a new file. Here are some common organization ideas you might want to consider.

Keep the front-end and back-end in different folders.

Thankfully, there aren't a lot of new monolithic projects getting created. There are still mono-repos and other frameworks that may require you to have the front-end and back-end in the same repo. No matter what you do, keep those responsibilities in separate folders. At the most basic level, your structure should look something like this.

```
|__client
|__server
```

Create folders based on functionality.

This is the most common type of folder structure you'll see. If you're on the back-end, there will probably be a folder for your database models called `models` or there will be a folder for any cron jobs that run called `crons`. The front-end equivalent would be having a folder for your UI components called `components` or a folder named `helpers` for any functions that get reused throughout the app. Let's expand on the structure above.

```
|__client
|____components
|____helpers
|__server
|____models
|____crons
```

Create folders based on business use.

In more industry-specific apps, you might see code grouped by business use. Maybe payments are a huge deal so you have a `payments` folder on both the front-end and back-end. Or you might have a `reports` folder to hold the back-end code that generates user reports everyday. We'll add even more to the structure we started with to show this.

```
|__client
|____components
|____payments
|____helpers
|__server
|____models
|______payments
|____crons
|____reports
```

When you run into situations where you aren't quite sure where some piece of functionality goes, then you could put the files in a `helpers` or `utils` folder until the location becomes more apparent.

Hopefully you can see how this makes it easier to figure out how an app works and where you can find things. Now let's go over a few things for file conventions because these can bring out a lot of opinions.

File conventions

Next to variable naming conventions, one of the most pedantic things developers get hung up on in the beginning. Here's the secret, it doesn't really matter what naming convention you use unless your framework follows specific rules in order to work, like Next.js. There's no right or wrong way and any effciency differences someone tries to mention are incredibly neglible.

The key to file conventions is consistency. While you have everything grouped under different folders, having descriptive file names will make it easier for you to figure out the specific functionality contained in that code. No file names should leave you wondering what you'll find when you open it.

For example, if you have a folder called `components`, most developers would expect to find reusable component code. So a file name might be `Modal.tsx`. Just a quick glance at the name and you can figure out that there's probably something about modals in that file. It's a very short, straight to the point name and that's how you want all of your file names to be. Use the least amount of words you need to label a file.

Also, be aware of the format of the names. Never use spaces or special characters in file names, unless it's an underscore. Not only are these poor practice, they also make it harder to work with the files in a command line interface or in an automated script. It's also a good practice to avoid numbers in file names if you can, but sometimes they clear up things.

Let's add some files to the folder structure we made earlier to demonstrate what you might expect to see.

```
|__client
|____components
|______Modal.tsx
|______ReportTable.tsx
|____payments
|______AccountInfo.tsx
|____helpers
|__server
```

```
|____models
|_____payments
|____crons
|____reports
|_____income.ts
|_____taxes.ts
|_____profit.ts
|_____monthlyRevenue.ts
|_____contactList.ts
```

A quick glance through this structure and you can see what some of these files might do. On the front-end, we've gone with Pascal casing for the file names while on the back-end we're using camel casing. This is just to show you that you can follow different conventions in throughout the tech stack.

NOTE

There are several different types of casing: Pascal, camel, and snake. Here are a few examples. Pascal casing uses a capital letter for each word in a variable name. Camel casing uses a lower case letter for the first work in a variable name and all other words start with a capital letter. Snake casing means keeping all of the words in a variable name lower case and separated with an underscore.

- Pascal casing: `SignInModal`, `PrimaryDropdown`
- Camel casing: `filterReportByDay`, `getData`
- Snake casing: `image_classifier`, `file_counter_by_type`

In some cases, you might have a back-end in another language like Python or Ruby follow completely different conventions. You can see that we kept the file names short and to the point. There are clearly three reports that do something on the back-end. We have a file named `ReportTable` on the front-end that we can assume holds a report table that shows some of the data from the reports and account info that lets the user interact with their personal information.

This is why file naming conventions are a small, but very important part of any project. They help tell the story of what's happening in the code.

Front-end considerations

You'll see some distinct differences in the structure of the front-end and back-end. I'm going to assume that you're using JavaScript across the full-stack to keep things simple. You'll likely follow the same naming conventions on both sides, like camel casing or any other name rules that have been put in place. The main differences lie in the way you lay out the project.

Sometimes the framework you use is more opinionated about the folder structure and naming conventions. For example, Angular is more opinionated than React when it comes to components and how

you pass props to them. When you're project has less strict guidelines on structure, then you can have fun making up your own. Here is a pretty common front-end setup for less opinionated frameworks, like React.

NOTE

I'll use examples in React and TypeScript throughout this book when discussing the front-end.

```
|__.ciCdConfig
|____config.yaml
|__src
|____assets
|______svgs
|______pngs
|______jpgs
|____auth
|______LoginModal.tsx
|______MfaOptions.tsx
|____components
|______Modal.tsx
|______Table.tsx
|____hooks
|______useAuth.tsx
|______useUpdate.tsx
```

```
  |____layouts
  |______AuthedLayout.tsx
  |______RestrictedLayout.tsx
  |____pages
  |______Login.tsx
  |______settings
  |________UserProfileSettings.tsx
  |____tests
  |______Modal.test.tsx
  |______UserProfileSettings.test.tsx
  |____utils
  |______theme.ts
  |____App.tsx
  |____index.tsx
  |____Routes.ts
  |____serviceWorker.ts
  |__.env
  |__.eslintrc.json
  |__.gitignore
  |__.prettierrc
  |__package.json
  |__package-lock.json
  |__README.md
  |__tsconfig.json
```

These are some of the common files and folders I've seen in working on numerous projects. It's a pattern of grouping files by their high-level functionality that stands out and it tends to work really well. Surpris-

ingly, it usually ends up in this structure through trial and error with organizing things as the team builds or adds on to a project. There are some best practices snuck in here, like having a `.env` file or auto-formatting the code with `.prettierrc` that we'll discuss in chapter *X*.

I've also added a few example files to show what could be in a folder. This way you can get a feel for what it would be like to implement something like this. Feel free to take this structure and use it in your own projects! It's a great starting point even if you change some names and move things around. One of the hardest parts about JavaScript development can be knowing where to start, so having a general template to begin with does help.

Back-end considerations

On the back-end, you'll likely be more concerned with authentication and authorization for the data users have access to in the app and how you get and transfer that data to various places.

REST APIs

When it comes to REST, most of your development will happen inside of a framework like Express, Next, Nest, or any of the other popular frameworks. These usually come with some built in rules for how

the folders and files should be created and maintained and any other expectations the framework has, so there's not as much to organize.

Some will give you more flexibility in your project architecture while others are more opinionated and keep you within specific set of requirements. For the ones that give you more flexibility, like Express, it might help to follow a simple structure like this:

```
|__src
|____common
|____db
|______models
|______migrations
|____services
|__jobs
|__tests
```

Microservices

With microservices, you have a lot of independently running functions that get deployed as their own app or service. Out of all of the back-end architectures, this will likely give you the most freedom in your project structure because you don't have to follow uniform conventions across microservices.

You could have one microservice built with Express, another built with Nest, and another built in a different language like Python or Go. Because these projects are small, you can keep the structure simple, similar to this.

```
|__api
|____users
|______getUserProfile.ts
|______updateUserProfile.ts
|______userProfiles.ts
```

Since microservices are focused on modular pieces of functionality, you could decide to have a single function in each file, like in the example. This is where you have to start considering long-term maintainability and how you think the app might change and grow over time with business needs and the product roadmap.

GraphQL

Compared to the other types of back-ends we've discussed, GraphQL is relatively new and the ways you'll see it used will be widely varied. It's commonly used when there are complex data structures that need to be displayed in the front-end. Although you'll likely see it used to replace REST APIs as well.

GraphQL is made up of types, schemas, queries, and mutations so your project structure can be pretty flat. You might consider something like this.

```
|__src
|____resolvers
|______users.ts
|______accounts.ts
|______products.ts
|____types
|______users.sdl.ts
|______accounts.sdl.ts
|______products.sdl.ts
```

All of your queries and mutations to work with data from the database will be stored in the files under `resolvers` and the type definitions for the resolvers will be defined in the `types` folder. You'll likely work with Apollo, which is an industry standard tool, to build this type of back-end and it has some opinions on how the project should be structured that will hugely influence where you put files.

Types of projects

Organizing the core folder structure in a way that's more specific to the type of project you're working on can also be very helpful. An app

built around FinTech will have different priorities and goals than an app built around recommending products to a user.

There's different functionality you need to focus on depending on the exact type of app you're working on. Some will have to comply with regulations like HIPAA or PCI and that can drive a lot of development decisions. How you manage data can also effect the way you handle your code choices.

Let's take a look at a few examples of different types of projects and the structures you could use. We won't dive into the functionality inside of the files here. The names you see are examples of what you might find in these types of projects.

FinTech

Many apps in this space involve some type of banking, credit system, or peer-to-peer payments. While all apps should be very aware of potential security vulnerabilities, apps in this space should have a heavy emphasis on it because of the sensitive information they work with. You definitely don't want users' banking information to be compromised.

Here's an example of how you might organize one of these applications.

Front-end

One good strategy is to focus on component-driven development on the front-end. This applies to any app you build, regardless of the functionality. It helps keep data-heavy components in a reusable and testable state that you can build more complex views on top of.

In all of the following front-end examples, the `components` are the building blocks for `views`, which are specific sections of `pages`.

```
|__src
|____tests
|______LoginModal_test.ts
|______AdminAuth_test.ts
|______UserAuth_test.ts
|____components
|______LoginModal.ts
|______AdminAuth.ts
|______UserAuth.ts
|______FilterDropdown.ts
|______SearchInput.ts
|______Form.ts
|____pages
|______Accounts.ts
|______Settings.ts
|______Tools.ts
|____views
```

```
|______AccountActivity.ts
|______AccountInfo.ts
|______AccountStatements.ts
|______AccountSettings.ts
|______UserSettings.ts
|______AdminSettings.ts
|______BudgetTool.ts
|______PaymentTool.ts
```

Back-end

You'll see much more variation on the back-end according to the framework you're working with, but here's an example of something you might see with REST APIs.

```
|__api
|____db
|______userModel.ts
|______accountModel.ts
|______settingsModel.ts
|____repositories
|______users.ts
|______accounts.ts
|______settings.ts
|____utils
|______logger.ts
|______helpers.ts
```

```
  |____services
  |______auth.ts
  |______email.ts
  |______caching.ts
  |____tests
  |____routes.ts
```

User Dashboards

You see user dashboards everywhere. They aren't specific to any industry, but they usually provider users access to some type of information. This could be analytics about their website, business, or community members. Usually you'll see more roles-based access in these kinds of apps.

Since these could be for any industry, the project structures you'll find in the wild will be extremely varied, but this is a good starting point and you can change or move things around to meet the project needs better.

Front-end

The user interface (UI) of these apps is typically the major focus so things like design, mobile responsiveness, and performance can be a bigger focus than with other types of apps. These types of apps will

usually include some kind of user-facing functionality as well as admin-facing functionality.

```
|__src
|____tests
|______LoginForm_test.ts
|______Home_test.ts
|______ReportTable_test.ts
|____components
|______LoginForm.ts
|______DateDropdown.ts
|______ReportTable.ts
|______SaveBar.ts
|____layouts
|______Home.ts
|______Settings.ts
|______Uploads.ts
|____pages
|______Home.ts
|______Settings.ts
|______Uploads.ts
|____views
|______home
|________TrafficSummary.ts
|________ActivitySummary.ts
|________UserSummary.ts
|______settings
|________ReportSettings.ts
```

```
|________ProfileSettings.ts
|________AdminSettings.ts
|______uploads
|________DocumentUpload.ts
|________ProductUpload.ts
```

Back-end

The focus of the back-end of these types of apps will likely be around user authorization and making secure JavaScript Web Tokens (JWTs). You'll also likely handle large amounts of data from third-party services so it's important to know how to get the data in a format that is easily consumable by the front-end.

```
|__api
|____db
|______userRolesModel.ts
|______trafficModel.ts
|______activityModel.ts
|______productModel.ts
|______documentModel.ts
|______userModel.ts
|______reportSettingsModel.ts
|______profileSettingsModel.ts
|______adminSettingsModel.ts
|____repositories
|______users.ts
```

```
  |_____userRoles.ts
  |_____activities.ts
  |_____settings.ts
  |____utils
  |_____globals.ts
  |_____logger.ts
  |_____helpers.ts
  |____services
  |_____auth.ts
  |_____email.ts
  |_____caching.ts
  |_____parser.ts
  |____tests
  |____routes.ts
```

E-Commerce

The longer you work in tech, the more likely it is for you to encounter an e-commerce app. It could be a full-fledged store or it could be a piece of the website. Either way, handling payments, displaying images, and having a well-defined user flow are crucial to this type of app. You'll see integrations with things like Shopify, Stripe, or some other payment processor.

Front-end

There will definitely be a large focus on performance here because you'll likely be trying to load a lot of images on the page for users to scroll through. Tied with that concern is the user experience (UX) because the only way users make purchases is if they are able to get to and through the check out process as quickly and painlessly as possible. Learning how to implement lazy loading here will help a lot and we'll talk about other ways to boost performance in Chapter 7.

```
|__src
|____tests
|______BaseForm_test.ts
|______Checkout_test.ts
|______BaseTable_test.ts
|____components
|______BaseForm.ts
|______BaseFilterDropdown.ts
|______BaseTable.ts
|______StatusBar.ts
|____layouts
|______Products.ts
|______Checkout.ts
|____pages
|______Products.ts
|______Checkout.ts
|____views
|______products
|________Clothes.ts
```

```
|________Accessories.ts
|________Shoes.ts
|______checkout
|________Cart.ts
|________ShippingInfo.ts
|________PaymentInfo.ts
```

Back-end

Since you'll rely on third-party services to handle a lot for you, one big consideration is error handling. What does your back-end do when it can't connect to a service? Another interesting thing that might come up is whether the company wants to handle customer payment information. This could bring your app under PCI regulations which dictates how you should store customer information and other security concerns.

```
|__api
|____db
|______productsModel.ts
|______ownerModel.ts
|______checkoutModel.ts
|____repositories
|______products.ts
|______checkout.ts
|______owner.ts
```

```
  |____utils
  |______globals.ts
  |______logger.ts
  |______helpers.ts
  |____services
  |______auth.ts
  |______email.ts
  |______imageFetching.ts
  |______payments.ts
  |____tests
  |____routes.ts
```

Healthcare

The last type of project we'll cover involves apps in the heathcare industry. These are used by a number of different people like doctors, patients, and administrative staff. The most important thing here is keeping patient information secure so apps in this area will have to comply with HIPAA regulations.

Front-end

This is another type of application that will rely on user roles and authorization access heavily. There will be a lot of different types of people interfacing with the app in a number of different ways so focusing on that division of functionality is key.

```
|__src
|____tests
|______PatientPortal_test.ts
|______DateDropdown_test.ts
|______BaseTable_test.ts
|____components
|______ScheduleForm.ts
|______DateDropdown.ts
|______BaseTable.ts
|______SaveBar.ts
|____layouts
|______AdministrativePortal.ts
|______DoctorPortal.ts
|______PatientPortal.ts
|____pages
|______AdministrativePortal.ts
|______DoctorPortal.ts
|______PatientPortal.ts
|____views
|______appointments
|________AdminAppointment.ts
|________PatientAppointment.ts
|________DoctorAppointment.ts
```

Back-end

The most important thing on the back-end and data side is making sure the app meets HIPAA regulations, like making sure patients always have access to their medical records and not disclosing more than the bare minimum information required for third party services. Most of the regulations are around patient data, but the back-end can assist with that by ensuring access controls are in place.

```
|__api
|____db
|______administrativeModel.ts
|______doctorModel.ts
|______patientModel.ts
|______scheduleModel.ts
|____repositories
|______administration.ts
|______doctors.ts
|______patients.ts
|____utils
|______globals.ts
|______logger.ts
|______helpers.ts
|____services
|______auth.ts
|______email.ts
|______dataParser.ts
|____tests
|____routes.ts
```

These are a few of the concerns you might encounter as you develop software across different industries and hopefully these project structure templates give you a good starting point. They don't have all of the folders and files defined and you may decide to go in a completely different direction, but the hardest part is usually figuring out where to start.

Feel free to pick these outlines apart and make them your own! In the next chapter, we'll get into some of the technical details that go into developing the full-stack for a project in any industry.

Chapter 2. Full-Stack Setup

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at milecia.mcgregor@gmail.com.

We're going to start slowly getting into more of the details, the hidden parts, that exist in a project that involve more than just great coding skills. There are a lot of layers to account for. You might notice a pattern in these "hidden things" I keep talking about: None of them are directly related to JavaScript development. All of these "hidden things" in the background affect what happens to the app in production. The reason it helps to learn about all of these things is because

it gives you a more holistic view of how the engineering department works together and how different teams affect parts of the project.

Even though you probably won't be responsible for everything in a project, knowing how all of the pieces in a full-stack application work together can give you some much needed context to write more maintainable code. This leads to a better user experience and improves your experience from the developer side.

In this chapter, I'll cover:

- How the app design, front-end, and back-end are connected to the data layer
- How things connect in continuous integration and deployment (CI/CD) pipelines to log errors and note when third-party services are down, and how site reliability engineering (SRE) aids in all of this
- How to handle worst-case scenarios

Application Design

The way the application flows and looks to the user is something that developers try to consider when building components, although it's always great to have a design to work with. Design documents cur-

rently come in a lot of different forms. You might get a detailed Figma doc to work with, a PDF, or a picture of a drawing on a board.

Figure 2-1. A design in Figma

Figure 2-2. A design in a PDF

Figure 2-3. A drawing on a board

Regardless of how you receive them, it's important to have design requirements, either from a product manager and designer or directly from a client, in place before you start so you know what functionality is expected and agreed upon.

A common pitfall for developers is jumping straight to code without fully understanding the problem you're trying to solve. Earlier in my career, I'd get assigned tasks and immediately start coding. Then I'd hit a point where I didn't know which direction to go next.

Had I stopped and asked some contextual questions, sketched out a few things, and really spent time upfront figuring out the little details, it would have saved me a lot of refactoring time and frustration.

Take the time to really look at the design you're given and see if you can also get behavorial specifications. The behavioral specs will define how the parts of the user interface or API will work when a user interacts with the app.

It's one thing to know how the app should look on mobile and desktop, but it's another thing to know what should happen when a user clicks a button or interacts with some element on the page. The design phase is the best time to start deciding how the app should work, down to the smallest detail.

If there isn't a strong product team or designer at your company, you might need to fill this gap by asking lots of questions. This scenario happens a lot in early-stage start ups where you as the JavaScript engineer may be tasked with a lot of these things. Some companies love when you ask questions; others don't. No matter what, continue asking questions until you have definitive answers for how the app should look and behave.

Some questions include:

- How will the user experience differ between mobile and desktop?
- Are there any edge case scenarios that come up?
- What is the user flow for a given page?
- Are there any screens that only certain types of users will see?
- Where and how should data be collected?

This initial research will make everything smoother because there will be less ambiguity as you start development. The better defined your design and behavioral specs are, the faster you can write good code with minimal refactors.

You never work complelety alone, no matter what part of the app you're responsible for. So early, consistent communication is essential to limit scope creep (when the product requirements or technical

implementations for the project slowly expand), keep development on track with any deadlines, and to ensure you deliver what is expected.

NOTE

Push back if the design or behavioral specs leave a lot open for you to guess. Junior and mid-level developers try to figure out how something should work on their own because of imposter syndrome: they feel like they should already know the answer and don't want to admit that they don't.

Never feel bad about asking questions. Some product managers, team leads, engineering managers, and clients might get frustrated with you, but don't let that stop you from getting clarification.

Building the front-end

Once you have a design, or at least some agreed-upon functionality expectations, you can start working on the front-end. If I'm the only developer on a project, I usually like to start on the back-end to address any data concerns before the project has moved too far along. However, it's a totally fine approach for a developer to start building the front-end without knowing the exact details about what APIs you need to call or what shape data will be returned in.

The thing to be most aware of is the framework that you're working in. While it's still possible to build an entire front-end with just HTML, CSS, and JavaScript files, frameworks have taken over the front-end world. We'll discuss how you can decide which framework to use in the next chapter, but for now, it's important to understand the basics of what the front-end code should do.

(See the previous chapter for a discussion of folder structure.)

A good approach is to start by building the smallest components possible — or choosing a library that has the small components made for you. I talk about how to choose appropriate packages for your project in chapter 4. For now, I'm going to assume that you are responsible for making everything from scratch.

Starting on a completely new front-end

Since you have a design or some kind of behavioral docs available, start by breaking each behavior or visual component down to the absolute smallest logic you can get.

Let's say you have the page design shown in Figure 2.2-1.

[[fig 2.2-1]] .Example design of a dashboard page image::images/image-to-come.png[]

Just by looking at this, you know that you'll need a list of options for the dropdown, you'll need to know how selections will change elements on the rest of the page, and that the dropdown needs to be responsive and accessible across different devices.

NOTE

Try to push for accessibility and responsiveness early. Accessibility is crucial for anyone with visual impairments, deaf/hard of hearing folks, and neurodivergent people. Everyone says "we'll come back to it later", but that almost never happens unless a huge issue comes up. But accessibility is actually a legal requirement in many countries and responsiveness keeps users coming back to the app. With any luck, explaining this to your product managers and team will give you some extra time to implement things like, WAI-ARIA for accessible navigation and repsonsiveness across many devices.

All of these things will need to be addressed with styles, HTML elements, and some JavaScript, even if you're using a framework. That can get overwhelming to think about when you're starting with a blank slate. So start with the HTML elements and some fake data. It's not going to be pretty at first — and that's the point.

You don't have to work on styles and functionality at the same time. It's usually better to have the component working before you start worrying about how it looks. In the case of our dropdown filter, that means writing some code similar to the following:

```
type BranchOption {
    name: string
    displayName: string
}

const branchOptions: BranchOption[] = [
    {
        name: "",
        displayName: "--Please choose a branch--"
    },
    {
        name: "main",
        displayName: "Main"
    },
    {

        name: "staging",
        displayName: "Staging"
    },
    {
        name: "beta",
        displayName: "Beta"
    }
]

return (
    <co id="select-element" linkends="selectEleme
    <select name="branch" id="branch-select">
```

```
            {
                branchOptions.map(branchOption =>
                    <option value={branchOption.name}
                )
            }
        </select>
    )
```

This has no styles and it doesn't trigger anything when you choose a value. That's how things should start off. Notice that we have the type definition for the data we expect from the back-end and we have some fake data to mock the response from the back-end. Then we use a simple `<select>` element for the dropdown. Inside of that element, we map all of the options to an `<option>` element.

All of these details will make it possible to use any data that gets returned from the back-end. Maybe you'll need a dropdown on a different page of your app with completely different values. It won't matter because you already have that component ready to use with any data that matches this format. Even if it doesn't match this format, you know exactly where to expand functionality.

From here, you can start making styles for this component so that it matches the designs visually. Then you can add accessibility and responsiveness to the component. By getting the functionality complete first, you give yourself time to find code bugs before visual bugs. That

will give you and any stakeholders a chance to find out if anything else needs to be clarified before you start adding more layers on top of this.

Then you repeat this process as you build out the larger views on a page. For example, the dropdown we made earlier <1> will change the table displayed on the page. So you'd start building the table component for pure functionality and displaying the data and then connect its display state to the current dropdown value.

Next, start building the other components on the page in this same way. Start with the simplest component and then work your way up in complexity. This allows you to catch weird edge cases before users do, and seeing elements connected to each other could bring up potential issues with the visual design.

Working on an existing front-end

It would be nice if you could start every project from scratch and you implement all the best practices from the beginning, but it's far more likely that you'll inherit an existing codebase. After you learn how the project is organized and how it functions, one thing you should start looking for is technical debt. Technical debt includes updating packages, cleaning up code that's out of date with best practices, and reorganizing code to match new product requirements.

This will help you bring out the areas where the project isn't following best practices, like responsiveness across different devices and form validation. Finding and fixing technical debt is another one of those hidden things that doesn't get taught. Waiting for users to find issues is not the best approach to learning where your application is lacking the technical backing it needs, but it does happen quite often.

Even if you don't have the time to go through and fix things as you churn out new features, that's ok. Keep a running document of issues and make sure to communicate them to the rest of your team and stakeholders. Many times it takes a fresh pair of eyes to find the things on an existing project that everyone has learned to develop around.

Usually if you see some type of technical debt that will help speed up new feature implementation, you can get the time you need to clean things up. Like, if you see convoluted logic around showing modals and you're working on a new modal, cleaning up the existing code will make it easier to extend the modal functionality, write tests, and speed up future development in this area. Just make sure that you have a strategy in mind, because there's a good chance that fixing it will become your responsibility.

Taking this level of responsibilty isn't as bad as some developers make it seem. You get to update the codebase to work how you see

fit, especially if no one else has strong opinions. Even if they do, this is a good chance to learn more about why the project works the way it does and to get some feedback on your thought process. It's always interesting to see how other developers would approach a technical debt issue because they can give you some insight on things you may not have considered.

Building the back-end

There is a core set of concerns for the back-end of a project including: security, authorization, performance, interacting with a database, working with a number of third-party services, and handling job queues. Depending on the type of project, you might also need to consider messaging queues, caching, or handling large amounts of data.

So the back-end has a different set of challenges than the front-end and that's why we shouldn't compare them, even though that happens a lot. When you're starting a new back-end project, you have to consider not only what framework to use, but what language to build with.

Choosing a programming language for a new project

The back-end could be written in a number of programming languages, but the front-end will always use some JavaScript. This leads to some nuances that you have to watch out for, because different languages handle functionality in slightly different ways. For example, if you know you'll be working with a machine learning model, using Python would likely be the practical choice because it has lots of machine learning libraries.

A well-written design document will outline the functionality you'll need to implement. If the biggest concern is speed, you might look at lanugages like Go. If you need something with plenty of libraries and packages you can import, Python, JavaScript, or Ruby might be good choices. If you're working on a game-specific application, C# could be a good choice.

Everything goes back to that design document — even back-end development. For every visual component, there is a back-end piece that provides the data and conditions it requires.

Handling back-end considerations

Every component of an app does something with data. If users need the ability to update their profile information, you have to decide what kind of authorization they need and how to check it. If there's a dropdown that dynamically changes the page layout, it's options are prob-

ably being called from the back-end. So for security purposes, you want to make sure no one can alter the dropdown options that are accepted by the database.

Authentication is how users are able to login to your app. This includes things like, two-factor authentication, single sign-on, and plain old username and password. *Authorization* is what gives users access to different parts of the app. For example, after a user logs in, should they be able to update the settings for other users, like an admin might have permissions to do?

Design documentation on the back-end

That's why the design document is important to the back-end as well. All of the questions will arise from working with the front-end to see what it needs to function correctly. Some companies split these needs out into a separate technical document because there are other things that are needed to provide a good user experience.

Writing documentation is yet another hidden thing that comes up in any project. It might feel like you're moving slow because you aren't writing code, but it's like the "measure twice, cut once" saying. If you know what you need to implement before you start writing code, you'll know what best practices to follow and potential issues that might

come up, like running background jobs to process user requests to other services.

I did some back-end work that required the app to parse XML documents so that the company could quickly update product inventory for the small businesses that used the platform. It took a while to finish the parser because the project manager was creating requirements as we ran into issues.

Since this was a recurring issue, I took the time to write up a quick document outlining everything the project manager and I had discussed along with some notes from the documentation for what the XML fields should be. Taking the time to do this saved us all a lot of time and confusion.

What to do about architecture

Then you'll need to make architectural decisions to keep your apps running fast. Sure, you *can* put all of the API responses for your data in a single file and call everything at the same time, but that will definitely cause some latency issues. That's why *modularity* is a good thing. Modularity means that the code is split into self-contained chunks of functionality. It helps you keep things in order.

With modularity, you'll be able to quickly identify which responses need different levels of authorization and which responses need to deliver large payloads so you can determine if caching the data is a good option. Writing a rough outline of the code you plan to implement will quickly show you where the technical gaps are. After you've written the documentation for how things should function, you can start to fill in the details.

This will give you a chance to research the tools and services available, like Redis for caching, OAuth standards for different authentication/authorization needs, and Sentry for logging. Another hidden thing nobody talks about is that it's ok to use tools and services to handle complex functionality like caching, handling message queues, or even payment systems.

Unless the application you're working on is supposed to provide one of these services, it's expected for you to use something instead of trying to build it yourself. This is why it's important to have a code outline to start with. Diving into the details of anything too soon can end with you locked into a specific vendor or a strict architecture that you have to follow, even when they become increasingly convoluted to maintain.

Once you've had a chance to get feedback on the functional document you've written and you've had time to really go through the

knots in your project, then you can pick one area to start diving deep into. It might seem like writing all of this documentation is a lot of work that should be done by someone else, but when the app gets deployed to production and you start getting bugs and change requests (which will happen no matter how hard you plan), all of this documentation will serve as your sanity check when everyone forgets what you discussed initially (which they will).

Starting a new back-end project can be tedious process because of all the different responsibilities this layer of the app has. Like with the front-end, it's actually simpler to set up a completely new project than it is to work on an existing project because there isn't as much technical debt to sift through and figure out.

Picking up an existing back-end

The upside to working on an existing project is that you have something to reference and you don't have to research appropriate tools. You can jump in and start adding new features or change the way data is being returned. There are a few pitfalls to watch out for though. You might find a lot of ways to refactor the code or even the entire architecture, but remember that this code is already in production and making huge changes will impact the way users interface with the project.

Keep that running list of technical debt and add any architectural changes you see that would improve the maintainability of the project, for instance:

- Creating a new version for the API you produce for other apps to consume.
- Adding new packages that would clean up some of the custom spaghetti code you see all over the place.
- Adding more test coverage so fewer bugs get out to production.

Security

One of the first things you should consider on an existing back-end is security. Do the JavaScript Web Tokens (JWT) have the bare minimum amount of information needed? Is there any data that should be restricted to an individual user or a group of users with the right permissions? Do you see anything that might be a vulnerability identified by the OWASP Top Ten, the standard awareness document for developers and web application security? How easy is it for outsiders to find internal resources through your API? Is it possible for a malicious user to run a distributed denial of service (DDOS) attack your app by requesting too much data?

This is just the beginning of the list of questions you can ask about security and it's important that you do because if the app you're work-

ing on isn't secure, then it *will* lead to issues later when vulnerabilities get revealed by someone outside of the organization.

After you've done a quick security audit, start looking at how the data is used. If you know there will be a lot of requests to a specific endpoint, how can you optimize the endpoint so that it doesn't become a bottleneck for users? Are there some calculations that could be offloaded to the database and served to the back-end via a view instead of a table? Can you split out any functionality that might be better on a separate endpoint? Improving performance around data can be a huge win for a company because everyone likes to get what they need as fast as they can.

The next thing to take a look at are all of the services being used. Is anything redundant? Or there alternatives that are more cost-effective or that have been tooling for what you need? Maybe there *is* a service that needs to rewritten internally. Perhaps someone forked a repo and has been making internal updates so it can't be updated with the core changes easily. Are there any services that just don't meet industry standards?

Getting used to picking apart existing code without being overly critical of the developers who wrote it before you is a skill that no one teaches. Some of the decisions that were made before you started probably had some valid reasons. It's just your turn to make sure that

the code is going to live up to the current reasons *you* have to deliver on.

The database layer

As a JavaScript developer, you will need to know how to work with the database layer. You will have to add new columns to tables to accomodate features and understand how migrations will affect the current data schema (the columns and relationships in tables) and any other applications that consume data from the tables. One of the primary things you'll do as a back-end developer is help define the data schema based on feature requirements and then serve that data through an API. As a front-end developer, you'll need to render the data based on the same feature requirements.

Understanding how the database schema is defined, how performance is handled at the database level, how to write basic SQL queries to check for values, and how to manage scaling are extremely valuable skills. You can get to the root cause of issues faster if you know where to check for issues outside of the codebase. No matter if your focus on a project is specifically the front-end or back-end, both of these areas depend on the data they work with and understanding how to work with raw data helps tremendously.

I'm going to discuss relational databases that use SQL throughout this book because these are the most widely used databases, but it's worth quickly mentioning the other types of databases.

NoSQL databases

The main thing to know about NoSQL databases is that they are not relational databases. That means the data is stored in any other format than relational tables.

Many teams default to relational databases because that's what their existing projects are built on, but there are a number of reasons your team might go with a NoSQL database:

- Large amounts of data that need to be stored
- Having a combination of structured and semi-structured data
- Working with microservice back-ends
- Needing an architecture that can be scaled out

The most commonly used types of NoSQL databases are: document, graph, wide-column, and key-value. We'll quickly sum up how each type of database works.

Document

Stores data in a format similar to JSON. So each entry in the database will be similar to a JSON object.

Graph

Stores data in node-edge format. The nodes hold information about things and the edges define the relationships between those things.

Wide-column

Stores data in tables, rows, and columns. It sounds similar to relational datbases, but the key difference is that the column names and format can vary row to row.

Key-value

Stores data in key-value pairs. This is a simpler type because it only has keys and pairs, so there aren't any tables or connections at all. You query by a key and the value returned can be anything from a string to a complex object.

A relational database can end up having multiple tables compared to a document database that will just have one record. There are a lot of trade-offs when picking on database type over another, so if you *are* ever in the position to choose, make sure you do more research.

SQL database

Now let's discuss relational databases since these are the ones you'll be working with the majority of the time. When you're given a feature

to develop, one of the first things you need to look at are the data requirements. Check for things like:

- New dropdowns or inputs
- Changes to user permission levels
- New connections between pages
- New third-party integrations

It's simple and fast to throw up a few relational tables and set some primary and foreign keys, but one big consideration is how the app will grow over time. Understanding how tables should relate to each other and why is huge in creating a flexible data schema. This requires learning more about the business domain and expectations for the industry. A database's schema isn't set in stone as soon as it's created. It will grow and morph over time just like the code.

Let's consider a database for an online store. It'll have tables that contain customer information, product information, order information, and so on, and they will all have their own columns and data restrictions. The database administrator (DBA) will have to pay special attention to the data types for all of these columns, who will have access to certain tables, how the tables are related to each other, and a lot of other details. Some of the common requirements for building out a schema are:

- How much data needs to be stored
- Any restrictions on data types and how data is connected
- What the functionality of the app is
- The type of authorization required to access certain data
- How the data will be used
- The frequency of data changes
- The database platform that can be used

All of these statements will need a clear definition or else the project can run into problems in the long-term. Once you have the schema defined, it's still fine to make updates later. Sometimes tables need new column names or data types need to be changed.

NOTE

You typically don't need to worry about the infrastructure part of databases, that's what the DBA is for and they might just be a member of the DevOps team. As long as you can connect to the databases in different environments, how they're provisioned in the cloud is handled by another team.

How performance is handled at the database level

Many developers focus strictly on code improvements when we talk about making an app more performant. However, there are a lot of changes you can make at the database level to speed up responses.

Sometimes the DBA might handle calculations in the database and store the values in views because you as a developer noticed that pulling values from a lot of different tables was slowing down the response time. What makes this a hidden part of development is that you might not be going into the cloud provider and updating resources yourself, but you'll be working closely with the people that do.

If the need arises, you can do a quick audit to make sure a database is as high performing as possible. Here are a few things to check:

- Ensure that the database server has adequate resources. Find a good balanace between CPU, memory, disk space, and the allocated budget.
- Try indexing the data. This will involve creating new tables that serve as indexes for the relationships between tables which can make queries more simple and faster than combing through all of the data before it's necessary.
- Look at the connection pool for the database. The configurations might need to be updated to handle more active connections.
- See if there are any queries that can be written as SQL statements instead of loops.
- Consider doing some data defragmentation. It helps clean up some of the free space left from deleting and inserting new data.

Thankfully, JavaScript developers don't have to worry about this! Having this amount of knowledge will allow you to look at performance issues from a different angle when it looks like the back-end has become a bottleneck and make suggestions for other places to check.

How to manage scaling

When an app becomes widely used and it's handling thousands of transactions every hour, the DBA will need to figure out the best approach for scaling the database. One common way to do this is by sharding the data.

Sharding means splitting the data into smaller subsets and then distributing them to a lot of different database servers. These servers will be physically separated, so connection strings will have to be updated and you'll need to take extra care to access the correct ones.

Another strategy to handle scaling is to *partition* a database. That is similar to sharding except all of the data stays on the same server. It just allows the data to be broken into smaller pieces that are accessible on a more granular level.

This isn't something you'll likely have to worry about. It's just good to know what's going on.

How to write basic SQL queries

There is one database thing you might end up doing if you work on the back-end and that's write SQL queries. If you're in charge of any database migrations, which is likely, then knowing how to query data, insert data, and drop data will be important. Sure there are object-relational management (ORM) tools that will generate the queries you need, but it's always helpful to know how SQL works.

There are some subtle differences in SQL queries depending on which database you use. We'll look at PostgreSQL since it's a widely used, open-source database.

SELECT query

You'll need to be able to get the data you need to send to the front-end and that will happen with `SELECT` statements. Here's an example:

```
SELECT name, quantity, price
FROM Products
WHERE id=3
```

Referencing the example tables earlier, this query is how you would get one product based on its `id`. These statements can get far more

complex when you start joining different tables together to get specific subsets of data, but this is how you'll start off.

INSERT query

You'll definitely be creating some new records to add to the database. The `INSERT` statement will take care of that for you. Here's an example:

```
INSERT INTO Products (name, category, quantity, p
VALUES ('pecans', 'snack', 2, 19.99)
```

The `id` for this new record will be generated automatically. All you have to do is provide a value for each column and make sure it is the correct data type.

UPDATE query

Updating records happens all the time. Whether it's a user resetting their password or a vendor updating their inventory, you'll see this query a lot. Here's an example:

```
UPDATE Products
SET quantity=52, price=1.29
WHERE id=1
```

The big thing to note here is the `WHERE` clause. You have to specify which record you want to update or else the action won't be successful.

DELETE query

There are times you'll need to delete records. It may be to increase the amount of space available to your database or it might be rooted in security concerns. Regardless of the reason, here's an example of the statement:

```
DELETE FROM Carts
WHERE id=4
```

It is extremely important that you have a `WHERE` clause in `DELETE` statements because if you don't, you risk deleting an entire table and not just the records you wanted.

Now you know how to perform the basic CRUD queries that go into every app. Remember, you don't have to be very proficent with the database level. It's just good to have some background knowledge in your tool belt.

Logging

When things go wrong with your app, it will be helpful to have logging in place. As you work with different companies and projects, you'll see that some take more time to implement this than others and the ones that do usually find root causes for issues much faster than those that don't. Logging errors and warnings in an easily accessible way is important for the overall health of the application once it's live to users.

Sometimes you'll roll out a new feature and no one will notice any problems for months --until several problems hit at once. One project I worked on didn't initially have logging. Every so often we would get bombarded with support tickets because users were experiencing unexpected behavior.

It turned out that the issue was that one of the third party services kept changing parameter names without notifying anyone. By the time we got support tickets, they would have pushed a fix. The only reason we figured this out is because we started logging errors on requests being sent to this service from our back-end.

I'll talk about service monitoring in a couple of sections, but logging really provided a sanity check for us. It's surprising how many warnings are can pile up in an application without you noticing. It could be little things like type errors or big things like incorrect/missing environment variables.

Aside from troubleshooting issues, there are a number of reasons to add logging to your applications, including creating audit trails and handling events and requests. This task might well fall to you, the JavaScript developer.

Audit trails

Keep a record of any changes made to your data. This will be invaluable when you're trying to figure when a change was made and who made it. It's also extremely important when you're working with data that is covered by any compliance requirements, like HIPAA or PCI-DSS, but it also aids in regular security precautions.

WHAT IS HIPAA

HIPAA stands for Health Insurance Portability and Accountability Act. This is a federal law that required the creation of national standards to protect sensitive patient health information from being disclosed without their consent.

WHAT IS PCI-DSS

PCI-DSS stands for Payment Card Industry Data Security Standard. This is an information security standard for companies that handle credit cards from the major card schemes, like Visa, MasterCard, and Discover.

These audit logs will typically include information about the data that was changed, the user who changed it, the system they were on, and the date and time the changes were made. If you're ever in a situation where data is compromised, these logs will give you the ability to undo changes and to find out when they were made and by who.

Events

Event logging is one of the most overlooked types of logging. *Events* are any type of activity users make: button clicks, pages viewed, or other actions taken. There's no real limit or definition for what events you log. Logging can give you some incredible business insight on how users interact with the apps you work on, and you can answer most business questions with some targeted logging.

You can see when customers abandon carts, the time of day they are likely to make appointments, and where exactly they're located based on GPS coordinates. The data you can get from logging user activity is something you can't get directly from the user most of the time.

So if you notice the company looking for more ways to appeal to users, make sure you mention event logging.

Requests

Any time a request is sent to an API, it should be logged. This helps you know when to upgrade resources to handle increases in traffic and it helps to keep your APIs more secure. You can set alerts for suspicious activity and you can see which endpoints get the most use.

The information typically tracked with request logs includes the date and time a request was made, the user id or IP address that made the request, the actual request data, and any header or body information sent with the request.

When you're using request logging for security purposes, a few things you'll want to take note of are:

- Any unauthorized access to restricted functionality or data in the app
- Any invalid API keys that are sent in requests to the back-end
- Every failed login attempt to be able to find any patterns
- All of the invalid input parameters sent in requests

These can reveal weak areas in your back-end apps that need to be looked into. The other side to request logging is that it can help you figure out when your application is used the most and by who. When you have this information, you can suggest when to increase or scale down resources.

How to log information

We've covered a few of the things you should consider logging, but how do you actually create logs? Depending on your time constraints, you can use a tool like Papertrail, Sentry, or DataDog to handle logging for you or write your own logging tool. It's much faster to use an existing tool than to create your own and there are plenty of packages available that let you do that. As a developer, it'll be up to you to implement logging regardless of whether you use an existing tool or create your own.

Some are attached to services you pay for and others let you store the logs in your own database. If you want the highest level of control possible though, you'll need to build your own logging tool. Sometimes the existing tools have a number of dependencies that could bloat your project. On the other hand, building your own tool and debugging it can take a bit of time.

You'll find that many companies and projects go with an existing tool. Although some companies that are in heavily regulated industries, like finance and healthcare, may opt to build their own tool.

Logging isn't just for the back-end either. Many tools will create logs directly from the front-end, especially in the case of event logging. If you don't see any type of logging on a project you're working on,

make sure to bring it up. It'll save everyone major headaches later on and it will give the company some valueable insights that they can't get any other way.

Continuous integration and continuous deployment

Once you have all of your code written and the database set up, it's time to get this project deployed to different environments. In the beforetimes, we used to deploy an artifact directly from a developer's computer through FTP. This led to a lot of unreliable deploys and issues across developer machines made it hard to keep consistent processes.

Then continuous integration and continuous deployment (CI/CD) came along and changed things forever. Now, it would be extremely weird to push updated files to production or any other environment directly from a developer's machine. CI/CD gave us an automated way to get code changes out to users, quality assurance (QA) testers, and other stakeholders consistently and quickly.

This happens behind the scenes after you've written the code and you have a feature ready to be released. Depending on the size of the company you work at and the culture they have around DevOps

and Agile practices, you could be involved in setting up a deploy pipeline so there are a few things you need to know.

What is a deploy pipeline?

A deploy pipeline is a set of repeatable, automated steps that happen each time code is pushed to a branch. This gives you consistent deploy processes where no steps get skipped because someone forgot to do something or because they were in a rush.

Build → Test → Deploy to feature branch → Test → Deploy to staging → Test → Deploy to production

These are the basic steps in a pipeline. You'll likely update environment variables to match the environment that the artifact is being deployed to and there will be different types of tests that get run depending on the environment as well. You might even have different pipelines depending on the environment you need to deploy changes to.

Different environments

Now that you know what a deploy pipeline is, let's go over the different types of environments you may end up deploying to. The common ones are QA, staging, and production. Of course there could be

any number of other environments and they could have different names, but these are the three you'll usually run into.

QA environment

Where manual testers check for any issues with a new feature or bug fix that made it through the initial round of developer testing. They'll also be looking for any unexpected regressions from code changes.

Staging environment

Where integration testing happens. If you've been working on a feature epic or smaller items that need to be merged together to get an accurate picture of how things are interacting, those individual items will be QA tested and then passed to this level. You'll be able to see how third-party services will work with the app before it gets to users.

This is the second most important environment next to production because it's the last chance to catch any issues before they go live. This environment should mirror production as closely as possible. That means it should have similar, if not the same, data, it should use all of the same services that production will, and it should trigger similar warnings and errors that production does. Having a good staging environment will help

the whole team find and debug any issues that come up without effecting users.

Production environment

The environment that users will interact with. You don't want to have any testing happening on production if it can be helped. By the time you deploy to production, the changes should have been through several rounds of automated testing and manual testing. That way you and the team can deploy without worrying about regressions and bugs sneaking through.

Setting up a pipeline

There are a number of tools you can use to set up your deploy pipeline, including CircleCI, Jenkins, and GitHub Actions. There are tools specific to different cloud providers, like Azure, GCP, and AWS, as well.

The first thing you'll want to do is create different branches in your version control system to represent the different environments. They can be named anything you like. Then you'll write a configuration file that will trigger the various deploy processes when you merge new code to the branches.

This is where the CI/CD part comes in. Everything automatically happens as soon as you merge changes to any branch. Depending on how you set up the configs for your deploys, you could trigger any number of actions to run in parallel or only happen when other actions finish. Just so you have a rough idea of what the config file for a CI/CD pipeline looks like, here's an example using CircleCI:

```
version: 2.1
jobs:
  unit-tests:
    docker:
      - image: cimg/node:14.20.0
    steps:
      - checkout
      - run:
          name: "install dependencies"
          command: yarn
      - run:
          name: "run project unit tests"
          command: yarn redwood test
  sast:
    docker:
      - image: cimg/node:14:20.0
    steps:
      - checkout
      - run:
          name: "install dependencies"
          command: yarn
```

```
          command: yarn
      - run:
          name: "execute retire.js"
          command: cd web; retire --path web
  build-app:
    docker:
      - image: cimg/node:14.20.0
    steps:
      - checkout
      - run:
          name: "install dependencies"
          command: yarn
      - run:
          name: "build deploy artifact"
          command: yarn redwood build
  deploy-qa:
    docker:
      - image: cimg/node:17.1.0
    steps:
      - checkout
      - run:
          name: "deploy to QA env"
          command: echo "Deployed to QA environme
  dast:
    docker:
      - image: cimg/go:1.19.0
    steps:
      - checkout
      - run: go version
```

```
        - run:
            name: "install nuclei-cli"
            command: go install -v github.com/proje
        - run:

            name: "Nuclei scan on QA"
            command: nuclei -u https://flippedcodi

  workflows:
    deploy-to-qa:
      jobs:
        - unit-tests
        - sast
        - build-app
        - deploy-qa
        - dast
```

This runs several security tests whenever this pipeline gets executed. Pretty much anything you can run in a terminal can be run in your CI/CD pipeline. It might take a while to test and get it executing all the steps like you need, but once it's in place it'll save you hours of time trying to figure out what went wrong in a deploy.

Service monitoring

While it feels great to have all of your code deployed to production, the technical journey doesn't stop there. Now that the application is live, you have to keep an eye on it to make sure that everything is working for your users. This is where service monitoring comes in.

Monitoring is a very broad topic and overlaps somewhat with logging. You'll run into companies that don't have service monitoring at all, more often than you might imagine. I once had a client argue that service monitoring was an unnecessary use of resources, even though they regularly got some interesting support tickets from their customers.

This is something you as a senior JavaScript developer will be expected to think about and implement. When there are fires on production when users aren't getting the data or functionality they expect, it will be up to you to figure out why. Understanding how monitoring tools work will give you a deeper understanding of where issues come from with respect to other teams. So the next section is a quick overview of how service monitoring works and where you can find it.

External dependencies

It's highly unlikely that a company will develop *everything* it needs for its product to function. Things like single sign-on (SSO) authentica-

tion, payment systems, and APIs will be integrated from external sources.

The main things you'll want to monitor in the third-party systems you use are availability and latency. These will tell you if the service is having any errors or timeouts happen. If you're using any cloud providers to host the application, check out your provider's health dashboard to see if anything weird is going on. This could be watching graphs in AWS or looking at lists in your worker queues.

Application monitoring

Once you have monitoring set up for the services you use, make sure you're monitoring your own services for things like downtime and internal errors. These have a direct impact on the end-user experience, which ultimately determines whether people will continue to use the service you provide. Unlike external dependencies, you have total control over your application and the way they behave.

So in terms of having control, application monitoring is a lot like logging. The difference is that you may want to set up alerts for when your own services are down or for specific errors that start to come through. You might also check for things centered around business key performance indicators, like average order values or number of monthly transactions. You can set thresholds for these values to de-

termine if you need to take action for specific users or a certain times of the day.

Network monitoring

At the network infrastructure level, the primary concerns are making sure you don't exceed the max number of open connections and any bandwidth limits. Knowing when we have issues with these two things show us where the bottlenecks for the application and how we need to scale the app's resources. You'll be able to see when you need to increase resources or change settings with your load balancer as well as if distributed denial of service (DDOS) attacks are happening.

As a JavaScript developer, your responsibility is to know where to look for answers to production issues at this level. You'll be able to determine if it is something that can be fixed through your code, through setting some environment variables, or passing it to the team responsible for maintaining the infrastructure.

Site reliability engineering

That's why we're going to cover a bit about site reliability engineering (SRE). SRE is a part of the operations team at many companies. Site

reliability engineers are typically developers who understand how to automate functions at the operations level across all of the engineering teams that deploy code to a production environment. They write scripts and use tools to scale resources. They're looking at things like latency and emergency response processes as well as how code is configured, deployed to different environments, and monitored.

SRE might sound a lot like DevOps. In practice, they work really well together. DevOps really focuses on keeping the deployment pipelines clear and up to date based on your concerns as a developer, while SRE is more focused on building automation to help address the underlying operations concerns.

Since SREs are separated from any particular development team, they focus on high level issues that can affect the whole engineering department. They also learn how all of the projects work and how they are connected to different services, so they get a unique perspective on how everything fits together. Here are a few specific things SREs do:

- They containerize applications. Having applications wrapped in a container can increase their reliability as they get deployed to different environments.
- They write documentation for how all of the projects interact with their individual services. This will be a different set of docs than

the development team will maintain because they focus on a different set of knowledge.
- They help decide what features can be launched based on service-level agreements (SLAs), agreements with users that specify the required level of reliability.
- They help find root causes for support issues.
- They create tooling to help make projects more consistent. If there are major upgrades to databases or cloud provider services that may cause downtime, they help manage those changes.

There is a lot more to SRE, but if you really like writing code and learning more about operations level topics, like managing resources in the cloud, you should definitely explore it more. It might end up being a fun career path for you!

Choosing cost-effective services

Another thing you could be tasked with doing is selecting different services your project will use. This might not be the same as picking out packages for your code. There could be some real costs involved when you consider things like, who to use to host images or videos, what message service you want to work with, or any queue systems you want to use.

With all of the different services available, it'll help to have a checklist of questions to make a good comparison between your options.

Service comparison checklist

- What is the pricing structure?
- What are the benefits at each price tier?
- How often are updates released?
- Does it supoort different cloud platforms or programming languages?
- How easy is it to integrate with an existing product or port from another service?
- How long do you have to make a decision?
- What type of support is offered?
- What do they offer if they experience an outage?
- Do they have other services you might use?

These are some questions that will give you a good report to show other people on the team so that you have the big answers. There will always be some trade-offs, but there's going to be a tool somewhere that can fit your needs.

Handling support issues

Every company that serves users normally has some support process they can go through to report bugs or get help with their accounts. There might be a dedicated support team that answers questions and writes engineering cards or the engineers might handle support directly. Either way, there are some issues that arise that might be a fire and require immediate engineering attention. These are the ones that will need your input.

Depending on how your company is set up, there are a number of ways to handle support requests. This could mean an on-call rotation, someone that monitors support emails, or the responsibility floats across different teams. If you find yourself in a situation where there isn't an adequate support process, there are a few things you can suggest to keep everyone moderately happy.

The support process

If there is a separate support team, take some time to talk with them and learn about their needs and the tools they use. This will give you some invaluable insight to give the engineering department. By learning the common issues they have reported and how they handle them, you'll be able to address these issues in your engineering work. It might even lead to new features that will really increase user satisfaction.

With a support team, try and create a process where they write cards or tickets that can be seen by engineering and make sure that they include all of the reproduction steps. That way, when the cards get passed to engineering, we will know how to prioritize it and how to see the exact errors users are seeing. Just always remember to keep the support team updated. They are the ones interfacing with users, so don't leave them hanging when they need to report back to someone with an answer.

Now if you don't have a separate support team, that usually means engineering is going to handle user issues. The decision to have a support team is made at the company-level, so you won't have much input here. If you *do* end up being on the team that handles user issues, an on-call rotation really matters. You don't want a particular engineer getting tagged in every issue that pops up all the time because this leads to burnout. There are a few ways to process issues from users.

Have some type of issue tracking system in place so that no one is sifting through emails. This makes it clearer to see when issues come in and how they are moving through the process from user reporting to engineering pushing a fix. It also helps to time-box people on how long they should spend on the initial research on the issues. After about 30 minutes, the issue should be prioritized on the task list for

the engineering team so that the researcher doesn't end up spending a whole day on it, unless it's a fire.

Once you have this process down, it helps keep people calm because they know when they need to drop everything and put out a fire in production or if they can move the issue to a later date. This is a great place to work with the product team to make sure they're aware of any issues that come in. It could have an effect on the feature roadmap they have!

Disaster recovery

There won't be many times that the whole database gets dropped, but in these extreme cases, it's important to know that you have the tools to handle them. This is something else that you as a JavaScript developer won't likely have to handle, but you will get tagged into the discussion so that processes around these situations can be defined.

While these situations tend to be rare, they are the most catastrophic because sometimes you can't go back to the state the app was in before the event happened or it leads to prolonged downtime for users. That's why it's important to have a disaster recovery plan in place.

Some of the reasons that are out of your control that might lead to downtime include: random natural disasters, malicious attacks, or the

needed dependencies being installed on the server hardware. Although you can't prevent those things from happening, the SRE team can build some resiliency in your applications.

Creating a disaster recovery plan

As a developer, there are some questions you need to ask your team and others in the company so you can have input on the process for disaster recovery because you will be asked to jump in when things are on fire.

- Where are server passwords stored and who has access to them?
- Who should be contacted in case of an emergency and where is their contact info?
- How can you periodically test the plan?
- Are there different teams that should work together on specific downtime issues?

Once you have discussed the answers to these questions, look at are the tools your cloud provider has available. AWS, Azure, and GCP all have disaster recovery tools, like database recovery options, multiple backup and restore options, and even the ability to ramp up resources in other regions in the event something happens to servers in a particular area.

Now you know about everything that goes on behind the scenes before and after your code is approved. If you ever find yourself working in a startup, this will come in handy often. You'll be able to understand and help build and debug every piece of an application. You don't have to be an expert in everything to know how to make it work together and that's the good part.

Chapter 3. Managing Packages

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at milecia.mcgregor@gmail.com.

In most companies, you'll be working within a framework and you'll have to build out some complex functionality. Then you'll be faced with the choice to build out that functionality yourself or choose a Node package manager (NPM) package that handles it for you. Many times it will be easier and faster to integrate a package into your project and it's a common practice to do so.

I've seen companies try to build their own authentication, form handling, and UI libraries and it usually becomes a huge undertaking.

You have to worry about maintaining a separate set of code that is typically different from the main functionality of the application, but also needs to be maintained and updated.

These types of internal packages tend to fall out of date pretty quickly and can take months to years to build, depending on the level of detail necessary to meet needs in areas like accessibility, security, or responsiveness.

One company that I worked with insisted on making their own custom UI library instead of going with an existing one and forgot that they needed to account for things like accessibility, responsiveness across a lot of different devices, cross-browser functionality, and theming like dark mode and other stylistic preferences.

Another client was determined to create their own authentication and authorization packages. They succeeded to a large extent, but it took about 8 or 9 months of development time to make that happen. So how do you know if you should use a package or make a custom implementation?

Deciding if you should use a package

Packages are like shortcuts to complex, common functionalities like rendering tables and modals or connecting to databases and working

with media. The quickest way to decide on whether to use a package or not is to estimate how long it would take to build the functionality versus using something that's already built to handle what you need.

These are some of the more technical considerations you should use to evaluate whether you should use an external package or create one in-house:

- How well-documented is a proposed package?
- How easy is it to customize a package?
- Would it be more beneficial to have total control over the package by creating an internal version?
- What are the alternatives?
- How often are new features released to a package?
- Can you keep a more consistent update cycle than the package's maintainers?
- What would happen if a package was no longer maintained?
- Do you see any features or needs for the project on the horizon that might change how you currently use a package?
- Do you know anyone who would recommend a package?

For example, if you're considering creating your own package to handle datetimes, you should really evaluate code that has already been written, tested, and used by a number of engineers. This type of functionality gets very complex and has a lot of edge cases.

Usually, if the functionality would require its own repo to keep the core project separate from all of the code it would take, it's a great candidate for an external package. It also depends on which framework you're using as they have different built-in functionality and support for different packages.

Another time you might use packages is when you need to implement the same functionality across multiple projects. All packages have a learning curve, so keeping everything in sync as much as possible will help onboarding new engineers and make it possible for them to hop around all of the projects with the same tools.

NOTE

Remember, even if you *can* implement your own UI library or payment system, you don't have to. Packages save you time so that you can focus on custom features for your application. It's not cheating to use a package to handle something you don't have time to write the code for.

Choosing the right packages

Once you've decided to use a package, it's time to start doing some research into the best package for your project. This usually brings out opinions from all of the developers on a team.

I've seen discussions about package choices take hours. It might seem tedious at first, but by the time you're three months into using a package, you'll be glad you did the upfront discussion because you will have a good understanding of how the package will integrate with your existing code.

When the package discussions are happening, take into account some of these things:

- What documentation is available and how often is it updated?
- Where you can go for code snippets and example implementations?
- How strong is community support for a package?
- How many dependencies does the package have?
- When was the last update released?
- How easy is it to find answers to questions that come up?
- Does the package solve a niche problem?
- What other organizations are using it in their projects?
- Are there any paid tiers that offer extra functionality you'll need later?
- How large is the package and how will it effect the bundle size and performance of the project?
- How well does it integrate with your existing code?

As you can see, there are a lot of details that go into choosing a package. The fun part of working on many projects is that these types of questions usually don't come up until something weird happens during implementation.

NOTE

One principle I try to bring from my time as a mechanical/aerospace engineer is "measure twice, cut once". When you're building machines, you don't want to waste material because it can be expensive or really hard to replace. The same thing can apply to software. You don't want to spend a lot of time and thought implementing something that will need to be undone shortly after you finish.

Knowing which package is the "best"

In general, there is no such thing as the "best" UI library, payment system, or feature flag package. They all implement functionality in slightly different ways. For example, a UI libraries will have different component props or class names that you need to use in order to get the user interface you want. Or feature flag packages might have different ways of handling documentation.

The "best" package for your project will usually:

- Be easy for everyone on the team to learn
- Integrate relatively seamlessly into your project

- Have clear documentation
- Contain functionality that supports long-term development for upcoming features

This typically helps you narrow it down to several packages that could be good to use, in which case, it comes down to preference. For example, if you're working with React, you might be torn between using Material UI, Chakra UI, or TailwindCSS to build out your front-end views. All of these are great options so it would be up to the team to decide what everyone is comfortable with and what will work best with the project-specific requirements.

Even though there is a lot of consideration that goes into choosing a package, the selection process can be relatively quick, often a week or less. You'll be able to catch any glaring issues pretty early on and make changes accordingly. It starts to get tricky when you need to expand the package with functionality it doesn't come with.

Forking packages

When you need custom functionality but you don't have the time to build an entire package from scratch, you can fork the repo for the package and make the few changes you need.

There comes a point on some projects when you need a package to do more than what's on the box. When this happens, you have a few options: write some hacky code around the problem, fork the package repo and make changes directly to the package, or make your own internal package. All of these options have pros and cons, but since you've already decided not to make your own package, we're going to focus on forking a package repo.

NOTE

Since many packages are a part of the open source JavaScript ecosystem, you should be able to access the source code and make edits. If you're using a third-party service, your access will be limited.

There are some benefits to forking a package. Instead of creating your own from scratch, you can just modify the thing someone else made! It can be a nice balance if you know you need to support things that are only relevant to your app, but you don't have the time to spin up a completely new thing. It's like expanding an existing project to meet your specific needs.

Once you've decided that modifying the source code for the repo is the path you want to take, you need to be prepared for what comes next.

Now that you have the source code forked from original repo, you are going to find out how well-written the package you work with is. In open-source projects, code quality can vary drastically. Some packages don't even properly support TypeScript yet.

For starters, you'll have to figure out how to run the package locally. You might test it out with the `yarn link` functionality. This is one approach to connect your project to the forked version of your package. That way you can see the changes you make to the forked repo reflected in your project locally. Then you'll dig through that code base to find the exact location for your update. When you have that update in place, spend some time testing your changes locally.

This is where I've seen projects experience significant scope creep: there can be an overlap in adding a new feature to your core project and making an update to a package repo. Since feature updates usually reveal where packages are missing the functionality we need, the two often end up tangled.

One strategy to keep this from happening is defining separate tasks to define the changes that should be made in both projects. You might start by implementing a new core project feature, realize your package needs some updates, pause on the core feature to make the package update, test out the package update, and then get back to core feature development.

When you have your own fork of a package and you've made your own code updates, there are a few things you need to be aware of. One big thing is that your package will no longer be updated like it was before. Unless you submit a pull request to the open-source repo and the maintainers implement your change, your code will only exist in your copy of the package. So when any patches get released, you'll have to be very careful with how you handle them.

When you make changes to a forked repo, make sure you keep documentation. This is super important to note because there will come a time when you aren't the primary engineer on that part of the project anymore and someone else will need to know about this code that you added and why it's there. Not only do you need in-code comments to explain the changes, you also need documentation somewhere else for the project that goes into more detail about why and where any package updates are and how to use them.

Doing package updates

Eventually you'll have a number of packages you depend on. They will interact with each other in different ways so knowing the props and methods you have available is key.

The maintainers of these packages are constantly releasing new features based on user feedback and where they see the package going, so keeping your packages up to date will give you a lot of new functionality that you may need. The maintainers will also patch security vulnerabilities, so that makes updates even more important.

Since you aren't the one making pull requests, you typically don't think about how old your package is. Next thing you know, your project is using packages that haven't been updated in years-and now it's vulnerable to any security risks that have been patched and you're missing out on potentially useful new features.

Taking the time to keep packages up to date will help keep your projects from drowning in technical debt. This section will discuss different package updates strategies.

Update on a schedule

Arguably the best approach, planning package updates at set times throughout the year will keep you from falling behind. This could be monthly or quarterly, but it'll make sure you don't have to go through huge, unexpected code refactors because a package introduces breaking changes. Breaking changes make your code throw unexpected errors and this can cause the application to crash for users.

Updating on a set schedule will help others on the team and in other departments stay aware of these necessary updates during sprints. You'll also start to notice when packages aren't necessary anymore and can be replaced with something more efficient.

There might be a major update to popular packages once or twice a year with several minor releases, but it's not very often that packages will introduce breaking changes throughout a single year.

Update packages incrementally

If you haven't been able to update packages on a schedule and you inherit a project with a bunch of out of date packages, you have a few options. The first is updating everything incrementally. This process is more involved because you need to know which packages need to be updated and how far behind they are.

Start by making a complete list of the packages your project currently uses along with their current versions and the most recent version available from Node Package Manager. Note which ones have been deprecated or not updated and decide if the project still needs them. Once you've determined which packages to keep, it's time to start listing dependencies. If any of your packages are dependent on each other, it's important to do those updates together.

With this documentation ready, you can start updating. These incremental updates give you more control over testing and how the updates are merged into production.

Incremental updates can make an app more resilent and help you find areas that need to be re-written entirely. However, this might involve a considerable amount of re-work each time a package is updated.

Update packages all at once

When you decide that you want to update all of your packages at the same time, be ready for a lot of weird behavior from your app. The very first thing you should do is create a new branch for your project to merge these changes to so you don't block your release pipeline.

Then you need to coordinate your development effort. This could mean dividing the packages up among the developers or having someone update all of the packages and then make a group party out of fixing bugs, updating tests, and updating props and methods. The main goal here is to prevent duplicate work and keep everyone on the same page.

Of all the approaches, this one is the messiest and most involved, but it does force a lot of quick progress. There won't be as much re-work

involved since everything is happening at the same time, but it is harder to find where all of the breakages are.

How to handle testing

If possible, make some regression tests to ensure breaking changes aren't getting deployed to existing functionality. No matter which approach you use, there's always a chance a weird regression will sneak in.

I once had an update to a modal component break the way datetimes were handled in a completely different part of the app. It was something that we never would have thought to test, but if we'd had regression tests in place, we would have caught and fixed that bug without a user ever knowing.

NOTE

This is a great argument to get more QA engineers for your team or to get time to write these tests yourself.

Refactoring code to integrate packages

There is a bit of refactoring that has to happen in existing code bases to get the full benefit out of the packages. The good thing about adding a new package is that you can slowly replace components as you go. You don't have to worry about breaking changes in unexpected places because this will be the first time your code even uses this package.

One of the package additions I had the most fun with was changing some homespun forms to use one of the form packages and it made things so much easier to maintain. Using something that has already been tested for weird edge cases, supports multiple browsers, and has functionality that can support different scenarios is a great feeling.

The refactor process

The process starts by you going through the code and finding the exact components that need to be replaced and then you'll make a list of the files that use this component. The next step is to go through each of those files and update one component at a time. It might seem slower than updating all of the files at once, but it'll help you work through and document any oddities, so other developers know what to expect and why you implemented something a certain way.

This also lets you keep up any continuous deploy cycles you have going so you aren't blocking any feature releases.

NOTE

One of the trickest things to manage is a branch that's backed up with code that can't be deployed. This happens a lot with code refactors, which is why it's a good practice to do the smallest updates and deploys possible. Trying to untangle Git commits is something you want to avoid as much as you can.

After you have all of the files for one component updated, repeat this process for the next component. As you are updating files, you should also be deploying these changes regularly. That way you can find out if there are any differences between your local environment and others and make corrections quickly in isolation.

When other systems are affected

Usually, you'll make the new package work with the current data format, like integrating a new user interface (UI) library. Although there are some cases where it would make more sense to update the data to fit the new package, like with data visualation tools. When this happens, you need to have clear discussions with the back-end team and any other teams these changes may affect.

It’s not very often that this will happen, but it’s one of those weird cases that comes up from time to time. Most of the time it happens when we start using a new front-end package to visualize complex data. The back-end may be able to return data in a more performant format instead of doing transformations on the client-side. Any back-end packages that get added don’t typically impact the database schema. That’s the layer JavaScript devs don’t touch unless absolutely necessary.

Using a new package can also mean using a new third-party service, like AWS or some other paid product. That’s why the most important part is communicating with everyone. One of the things senior engineers do is make sure that any architectural decisions are clearly commuincated to everyone possibly affected by it. New services can involve purchasing a subscription or switching multiple systems over time.

Migrating to a different package

Over time, packages rise and fall in popularity, the maintainers might decide to stop releasing changes, or your project might slowly morph into something that needs different functionality. Migrating from one package to another is just a different type of refactoring task.

You'll have to check the parameters that get passed to different methods and components to see if your proposed new package is compatible enough with the old one to make a quick switch. If that's the case, you can just switch the package using import statements. On the other hand, if you see that a package migration will cause breaking changes, check the scope of the breakage.

One of the best ways to ensure that breaking changes don't make it to production is to have regression testing in place. This could be an automated suite of tests written in something like Selenium or Cypress. Having these automated tests available will be a huge help with package migrations.

There are a few little "gotchas" when it comes to migrations. Here are some things to keep your eyes on.

- Look out for anything that might impact any Webpack configs
- Watch out for subtle style changes across different screen sizes
- Update any unit tests that are associated with the new and old packages
- See if there are any complementary packages for the one you're migrating to
- Consider any re-architecturing opportunities

Switching packages might be a great idea if it gives you the functionality you need in a better way for your project.

Chapter 4. Deploying Code to Staging

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at milecia.mcgregor@gmail.com.

Staging, or an environment like it, can be your sandbox to try out all kinds of things that you want to see in production without taking the risk of breaking anything for users. Deploying your code to staging is one of the most exciting parts of development! For one thing, it means you can work on something else. Sure, issues might arise when QA is doing their testing, but they won't take nearly as much time as the initial development. Staging is usually just a step away from going live to users.

That's when you finally get to see how your changes affect users and call a task truly complete. This is usually where the work we do as JavaScript developers turns into more of the hidden things. We work together with DevOps or some other team that's responsible for getting the code to the correct environments.

Understanding some basics of how the cloud platform works and how all of the external services tie together will help you work with DevOps more efficiently. There's always a blurry area between what developers are responsible for and what DevOps is responsible for because there's a bit of overlap. With infrastructure as code (IaC) becoming the standard way to setup DevOps pipelines, there will be some small tasks that you'll be able to handle without the DevOps team.

If you work for an early-stage startup, you might end up doing some DevOps work. At more established companies, there will be a team dedicated to this kind of work. You'll talk to them when there are new requirements for an app or if there's a bug that can't be traced back to the code or data. When bugs come up that you're positive aren't related to any code changes, take a close look. There may be issues in the enviroment itself that are causing errors and weird app behavior.

In this chapter, we're going to take a look at some of the cloud environments you may run into after the app is built. Some of the things

included in this chapter will be different views of things we've discussed before so that you get a well-rounded idea of how your code works in different places.

If you don't know anything about cloud providers right now, that's ok. It's not something JavaScript developers usually focus on and this is all knowledge that just comes over time.

An *environment* is a server where your app is run. This could be on production for end users, on staging for internal users, or another location for any other use. Cloud providers like AWS, Microsoft Azure, Google Cloud Provider (GCP), Heroku, and Netlify make it easy and fast to spin up new servers, with a lot of cool integrations to automate even more things. You'll probably end up working with one of the first three providers since they are the biggest in the market, but the others come up a decent amount.

Environment variables

One of the first things you need to check before deploying to staging is all of the credentials you need. At this point, you already have the services you're going to use set up. So you'll need the authentication/authorization credentials and any specific endpoints or other values for your app to use them. These values are called *envi-*

roment variables because they can change depending on which enviroment your app is being run in.

When you're working across multiple enviroments, it's likely you'll have different service accounts so that you aren't running up a bill on your production accounts while you're testing. That means you'll have sandbox or test accounts that might not have the full functionality that production has, but are close enough for testing.

Having different service accounts means that you'll switch between different sets of credentials and strings across the environments you have. For local testing, maybe you have them hard-coded in certain places. If so, extract the hard-coded values to a `.env` file and set the app up to reference the values from the environment variables.

The variables in your environment can be strings, numbers, booleans, or even objects and arrays. A `.env` file could look something like this:

```
LOG_LEVEL=INFO
LOG_COLORED=true
HTTP_PORT=3000
HTTP_HOST=0.0.0.0
DB_USER=username

DB_PASS=password
DB_NAME=qouple
```

```
DB_HOST=staging
DB_CHECK=true
DB_RUN_MIGRATIONS=true
PUBLIC_API_URL=http://localhost:3000
UPLOAD_SERVICE_URL=http://localhost:7070
UPLOAD_SERVICE_API_KEY=secret_key
CHECKOUT_COM_API_URL=https://api.sandbox.stripe.c
STRIPE_SECRET_KEY=sk_text_secret_key
STRIPE_PUBLIC_KEY=sk_text_public_key
```

Anything that shouldn't be exposed on the client-side should be stored in environment variables. Depending on how your CI/CD pipeline is configured, the enviroment variables could be set directly in the CI/CD service. They could also be set dynamically on deploy based on some environment conditions.

Environment variables aren't hard to manage. As a developer, it's important to keep the values up to date and make sure everyone is using the right values in the right places. This is where communication with DevOps becomes important. You are usually able to go into your CI/CD pipeline and update those values, but DevOps needs to be looped in so they aren't hit with unexpected changes.

Cloud providers

Some companies with very sensitive data won't host their apps in the cloud. They keep a physical server rack somewhere and dedicate someone on the Operations team or the general IT department to allocate server resources, monitor for uptime, and handle hardware tasks. This used to be the standard way that all websites were put online. Having an on-premise server is the most secure way to host an app online, but not all apps need that level of security.

You'll see on-premise servers for government vendors, research labs, and maybe a few other industries. For the most part, the apps and websites we work on every day are in the cloud somewhere. We don't have any access to the hardware, but the cloud providers are responsible for maintaining the uptime for their servers and reporting any outages.

So even though I'm going to discuss cloud options, don't forget that an on-premise server is an option as well. I'll focus on the parts of the major cloud platforms that senior developers interact with.

Amazon Web Services

Depending on the services you use, AWS might be more expensive than the other providers. Of all the cloud platforms, AWS is the most mature and it's recognized as the gold standard for security and cloud reliability. It also has more compute capacity than other providers. It

can be overwhelming getting started because there is so much to look through.

Amazon Web Services (AWS) is the most widely used cloud platform around. It has over 200 services you can use with your projects, like data storage, machine learning, monitoring, and way more. You can sign up for a free account and poke around to see all of the things they offer.

WARNING

If you do make an account, make sure that you check the configs for how you use services so that you don't get an unexpected bill!

Most of the time, the front-end of the app will be uploaded to an AWS S3 bucket. These buckets can store anything from applications to large datasets. The app usually gets uploaded as part of an automated DevOps pipeline, but you can easily drag and drop files directly into buckets.

One of the ways a JavaScript developer might work with an S3 bucket is when an issue comes up on staging that can't be tracked back to the code or data. You might need to check the code in the bucket to make sure the right version got released.

It's surprising how easily the wrong code can get deployed automatically — and it's a really hard issue to pin down. In one project I handled, the images in the app wouldn't show up in staging because of the way AWS was reading a string. It took months to figure that out. We only found it when someone on the DevOps team paired up with one of the JavaScript developers to walk through all of the code around images.

Another commonly used service to serve back-end applications is EC2. You'll hear about EC2 instances quite a bit because that's how we handle load-balancing and monitoring for the back-end. This is usually where the DevOps team takes over and they handle scaling up and down resources here.

Permissions can be highly configurable by user and that helps provide an extra layer of security. As a developer, you likely won't have all of the permissions for the functionality on your cloud platform and that's fine. If you work on an app in a more regulated industry like health or finance, it's a necessity that no one has a higher level of permissions than they need.

Microsoft Azure

The second most popular cloud platform is Microsoft Azure. It's pretty close to AWS in the number of services it offers. What it does give

you is better support for enterprise applications and easy integration with your current Microsoft services. Out of the big three cloud platforms (AWS, Azure, GCP), Azure is relatively cheaper.

You aren't locked into Microsoft tools either — although that's a common misconception. You can deploy any type of code on this platform and use any external services you want without having to do anything with Microsoft. It also does well with a hybrid cloud strategy.

A *hybrid cloud strategy* is when you decide that some parts of your applications and data should be hosted on a public cloud infrastructure, like Azure, and others should be hosted on a private cloud infrastructure, like an on-premises data center. This is one of the ways Azure caters to enterprise customers better than the others.

When you can easily split data and functionality, it can help you comply with any laws or regulations that cover that particular industry. For example, an application being built in the healthcare industry will have to comply with HIPAA. So they might decide to host their application in Azure while they have the data stored on-premises in a closet somewhere in the building.

Another unique feature is that Azure lets its cloud services run on both AWS and GCP. This can be a completely game-changing feature if you already have things hosted on AWS or GCP and you don't

want to do a full migration. You can even distribute functionality distributed across several cloud platforms.

While you aren't locked into using Microsoft products and tools with Azure, it *does* offer seamless integration with Microsoft tools and Windows OS. Azure Active Directory identity services, for instance, allow single sign-on across a number of environments.

You do have to watch out for some security vulnerabilities with Azure. For example, when you create an instance of a virtual machine, all of the ports are open by default. So you'll need to make sure the services you use have the right security configurations.

Azure's documentation, while still really strong, is lacking in a few areas. It might be hard to track down recommendations directly from the Azure documentation, so getting started may take a bit more time. There are also difficulties with technical support responding to questions at times. With all of its pros and cons, keep in mind that Azure is right behind AWS on almost all fronts. There are services being constantly added and expanded to fit the market.

Google Cloud Platform

Coming in third (and newest) on the list of cloud platforms is Google Cloud Platform (GCP). Quite a few startups use GCP because it of-

fers some really strong incentives for new companies to come to them, like various discounts and a strong free tier.

GCP has invested heavily in its machine learning services which are better than Azure's and right up there with AWS. You can build and deploy models using their open-source machine learning library, Tensorflow. This is partly thanks to GCP's support for containerized workloads. GCP also has services that work with Android, the Google Workspace, and Chrome OS if you have any integrations you need to do.

GCP can compete with AWS when it comes to privacy and traffic security configurations and the way payments are handled. Overall, it's one of the most secure cloud platforms, with features like identity-aware proxy and data and communication channel encryption.

GCP tries to offer the lowest price, so you can experiment with multiple services. They also use a different type of storage system, similar to Google Drive, so it's incredibly fast. If the app you're building works with large data, GCP might be the best option.

The documentation and organization you'll find for GCP is helpful and clear which is something it has over Azure. You'll find that GCP has more support for open-source technologies, especially since they contributed heavily to the development of Kubernetes. GCP also

started working on their hybrid cloud services in 2019, but they're still a bit behind the other two in this area.

When it comes to the number of regions GCP covers, it's still developing and trying to add more. They do struggle a little working with enterprise companies, but they do tend see use as part of many multi-cloud approaches. They also have a global fiber network so that your experience is as fast as it can be.

Overall, GCP is a great choice if you're working with big data and machine learning. It's got a great security suite and the ways you're able to configure your account stand out amongst the others. It is still lagging behind Azure and GCP in terms of service offerings and number of regions they cover, but it's the platform experiencing the fastest growth so they are working hard to catch up.

Data on staging

Once you deploy the application to your staging environment, it's usually ready for testing. One of the tricky things about testing on staging, which we'll cover in the next chapter, is getting realistic data to test with. Production has all of the real user data; staging is relatively empty in comparison. For tests in staging to be useful and realistic, you have to have data that is very similiar to production.

When you consider that some industries like healthcare and finance have regulations around how user data is handled, this becomes an even more interesting problem. How do you get production-like data in staging? There are two options: create your own dataset or sync production data to staging. Let's look at each in turn.

Create realistic, fake data

You can make a totally custom dataset that meets all of your use cases from scratch or using an auto-generation tool, like Mockaroo or dbForge for your SQL database. This can be a hefty investment if you have to mock a lot of data, but it also gives you a consistent snapshot into changes in the application. You can add new data as you add new features so you don't have to wait for updates from a third-party service.

It takes some time to build up a good dataset for staging with this approach, but it's possible. It tends to work best with projects that have had mock datasets since the very beginning. Usually you'll add new data to staging with each feature release. This can be helpful with finding bugs earlier in the software development lifecycle (SDLC). It requires you as a developer to really think about the data you're working with.

You're also making a new record for edge cases you find. This will help with regression testing because the staging environment will account for all of those weird conditions. One downside is that developers will have to manually update data each time a new feature goes out, which could eventually lead to unstable data if not properly maintained.

Sure, updating data can be lumped in with activities like development and testing. It becomes an issue when you need to mock a lot of specific data, there are tight deadlines, or the shape of the data you're working with has frequent changes. This leads to bad data in staging with will cause increase the time for features to be deployed because you're testing for issues in staging *and* production.

Sync production data to staging

Syncing your production data to staging is usually the better approach because it makes the staging environment a replica of production, so testing is more realistic and meaningful. But there are some important considerations, including:

- Do you need to obfuscate data to stay in compliance with HIPAA or PCI?
- Should you sync *all* of the tables or a subset of them?

- Are there certain times of the day you have to wait for data to be available?
- Does staging need to sync with production daily or would weekly or monthly syncing work?

Most of these concerns will be handled by data and operations teams, but you as the developer you know how the data is being used and can context to help drive the answers to these questions.

A good best practice is to obfuscate any sensitive information on staging. This might be done through a stored procedure or some ETL process, but you don't need the real production values to develop against. Because security is something that everyone from product management to the DevOps team should be thinking about, it makes sense to keep real information off of staging. That way data isn't compromised anywhere.

Setting up the sync process might take some time. Staging usually doesn't work with the same amount of cloud resources as production does, so you'll need to do some research to make sure the data doesn't overwhelm the environment. Even making sure that your staging database is read-write can take some time. The initial set up for an automatic sync with production is larger than manually adding data, but when it's in place and the bugs have been worked out, nothing is more accurate.

Logging

I'm going to discuss logging quite a bit in this book because it is crucial for every app and constantly gets overlooked. The focus here is on logging in staging and the things you should consider logging. There are a few different types of logging you'll want to do on staging like, server activity logs, database change logs, application event logs, and others. Let's discuss all of these.

Server activity logs

To deal with downtime or slow responsiveness, you need to know when these issues are happening. It adds to the data you have available and can highlight when you need increase resources and when you can scale them down. This data can also help you spot a security breach.

Logs don't just record user behavior. They also record any server activity you tell them to. This can give insights on when users are interacting with your application and how many users different endpoints get. On staging, you might log things like response times to get an idea of how long processes will take on production.

You can test out different metrics you want to log and see what they look like, how often they get generated, and if they are as useful as

you thought. Since logs are underrated until an issue happens, staging is a great time to test out any logging you want to do in production.

Database change logs

It's very important to log database changes. Some of the keys bits of information to capture are the changes made, the time and date, and who made the changes. This could be anything from password resets to settings changes.

Database logging is sometimes legally required. It helps you identify any data breaches quickly and pinpoint where they came from. Hopefully you won't have to deal with one of those, but it's much better to have this in place *before* something happens.

On staging, you can play with how you define the schema for logging tables and how you want to capture data. Then you can run experiments to make sure that your logging provides enough coverage. Logging is also useful for debugging issues when you aren't sure if something is happening just on staging or production too.

Application event logs

Application event logging is the one of the fun parts of logging. You can get some incredible business insights into how people are using your app. Knowing where users drop off in a process or which buttons they spend the most time clicking on can drive new feature development.

There many tools available for this kind of logging. Use staging to test them out and see which one works best in your application. You can also test out different report structures to determine which logs are most influential to business decisions. Sometimes, developers are expected to bring up features or changes based on the patterns they see. This is where you can start doing some analysis to bring out some interesting data-driven conversations.

When you take the time to log app events, you learn more about your app than you can from surveys, interviews, or support emails. It can also reveal potential security vulnerabilities in workflows that weren't considered before.

There may be some pushback on including this much logging on staging, so make the push to include it. It's one of the things senior developers do because they have seen what it's like when problems come up and you have to add logging in the middle of trying to troubleshoot. It gives everyone some exposure to the things that they need to look for when unusual activity happens on production.

CI/CD Automation

Even though maintaining of the pipeline isn't usually your responsibility, any issues that are uncovered in the deploy process are your responsibility to research. Sometimes it will be a code change you made, something with a third-party service, a data issue, or even something wrong with the pipeline itself, like missing credentials.

You'll be the one that gets tagged in for the first review, so understanding how your deploys work will help you save time and frustration from looking in the wrong place. Doing this kind of research and pipeline testing on staging helps build confidence across a number of teams for any releases that get shipped to production.

Many projects will already have continuous integration and continuous delivery (CI/CD) in place and maintained by the DevOps team. The thing you need to know as a developer are the conditions to deploy your code to different environments.

[[fig 5.5-1]] .Object map showing how different things trigger deployments image::images/image-to-come.png["object map showing how different things trigger deployments"]

You need to know what actions are triggered in the CI/CD pipeline so that you know how to handle your own workflow. For instance, some

CI/CD pipelines will run tests any time a change is pushed to any branch; others will only execute tests in specific environments, like staging or QA. Learning what makes things happen in your pipeline will help you decide how to manage branching strategies.

You can usually find the CI/CD config file within the project repo you're working on. Senior developers check out this file to understand how the app gets deployed to all of the environments. This can be great way to spur conversations between developers and the Dev-Ops team. If you have a QA team, it might be helpful to deploy code changes to feature environments where they can test functionality in isolation against production-like data.

Usually changes won't get deployed to staging until they have been tested in isolation or had unit tests run on them. A basic pipeline to staging can look like this.

```
version: 2.1
jobs:
  unit-tests:
    docker:
      - image: cimg/node:14.20.0
    steps:
      - checkout
      - run:
          name: "install dependencies"
          command: yarn
```

```
          command: yarn
      - run:
          name: "run project unit tests"
          command: yarn redwood test
  sast:
    docker:
      - image: cimg/node:14:20.0
    steps:
      - checkout
      - run:

          name: "install dependencies"
          command: yarn
      - run:
          name: "execute retire.js"
          command: cd web; retire --path web
  build-app:
    docker:
      - image: cimg/node:14.20.0
    steps:
      - checkout
      - run:
          name: "install dependencies"
          command: yarn
      - run:
          name: "build deploy artifact"
          command: yarn redwood build
  deploy-qa:
    docker:
      - image: cimg/node:17.1.0
```

```
    steps:
      - checkout
      - run:
          name: "deploy to staging env"
          command: echo "Deployed to staging envi
  dast:
    docker:
      - image: cimg/go:1.19.0
    steps:

      - checkout
      - run: go version
      - run:
          name: "install nuclei-cli"
          command: go install -v github.com/proje
      - run:
          name: "Nuclei scan on QA"
          command: nuclei -u https://flippedcodi

workflows:
  deploy-to-qa:
    jobs:
      - unit-tests
      - sast
      - build-app
      - deploy-qa
      - dast
```

The pipeline will go through some initial testing that can include unit tests to check for any regressions and a static application security testing (SAST) tool to catch any out of date packages or other security risks that can be detected through code. Then a build will be created and uploaded to a server. If your team is security-focused, you can include dynamic application security testing (DAST) or interactive application security testing (IAST) on staging. This will help reveal potential security risks long before you release to production.

Testing on staging

This is another topic that will come up a few times throughout this book. Testing is an incredibly useful tool, but it does take some upfront work to implement correctly. The early phases of testing will fall completely on you as the developer. Making sure that acceptance criteria is met, writing unit tests and sometimes automated tests, and being able to implement some simple security testing are all things senior developers think about for every project.

Acceptance criteria testing

Acceptance criteria testing is the first place you'll start right after implementing your code. It involves just clicking around, making sure that your new feature or bug fix actually works. Some developers will

toss their code over to QA or even say it's ready for staging and production without checking that it does what it's supposed to do without breaking everything else. If you've ever wondered why senior developers tend to release less buggy code, it's because they do good testing before anybody knows they're done with the implementation.

Run through a few common scenarios before you put up a pull request. Log in as a different type of user, see what the app looks like on a mobile device, make sure that permissions are set around endpoints. These are quick checks that can catch simple bugs so you can fix them without kicking off a whole process. If there's any functionality that you aren't sure of how to test, get clarification around the acceptance criteria. You might have found something no one considered before and just saved everyone weeks of work.

Unit testing

You can write tests alongside your code. Most JavaScript frameworks have a testing library you use. A project's level of code coverage a project depends on how well the unit tests cover functionality. If you're lucky enough to start a greenfield project, you can implement tests down to the component level so you can test things like buttons and dropdowns. It's more common to test on user functionality though.

Can a user log in with the right credentials? Will they see an appropriate error message if something goes wrong? Does the correct data load when they click a button? Does the data in the response match what is expected? These are tests that can be run on several pages in any application. A best practice is to write tests as you write functionality so that you immediately have coverage and a good understanding of the scenarios that may happen with a feature or page.

Integration testing

We'll dig into integration testing a bit more in the chapter on testing, but you can always include some integration testing on staging. This is where you can automate tests to make sure you're connected to any third-party services and that they are working well with your app. You can also automate some user actions. You'll see tools like Cypress and Selenium used for this.

```
// Cypress example
it('clicks and fires a change event with the new
  const onChangeSpy = cy.spy().as('onChangeSpy')

  cy.mount(<Stepper onChange={onChangeSpy} />)

  cy.get(incrementSelector).click()
})
```

Static application security testing

Static application security testing (SAST) is something that even many senior developers don't take advantage of. Some companies have dedicated security engineers, but SAST is something you can quickly add to your testing process, even in the CI/CD pipeline.

SAST usually involves a library you can install in your repo and run against your code to generate a report listing any out of date components or any packages that have known security risks. If you find out about insecure elements in your application early, you can switch over to different packages, update the existing one to a version with a patch, or create a security patch of your own.

Front-end testing

I've worked with several companies that made developers feel like writing tests was a waste of time and they usually had more issues in production than the companies that value testing. Senior developers can lead the way for writing tests. Adding a test or two every time you write some code is a great way to lead this initiative.

That's one of the great things about tests. You don't have to write them all at once. You can slowly add coverage over time and improve the quality of the whole project at every step. We'll spend all of chap-

ter 6 writing unit tests for the front-end and back-end; for now we can run through some test cases you can add to any project.

User log in flow

If you have a screen where users log in, there should be a test around the whole process. You should have mock data to mimic the expected response from the server. It should account for any error messages that might be returned for a user name, password, or two-factor authentication code. Writing tests for the authentication flow can help you check for potential ways that users might mess up so you can write code to help prevent it.

Error message handling

Sometimes you make requests to the back-end and all you get is an error message. You can test to make sure the correct components are rendered based on the error returned. It's important to make sure that your error handling is working everywhere because the app can crash if you miss a scenario.

Data validation

The APIs we work with can change responses unexpectedly. You know the data structure and types you expect from the back-end so

having tests around user actions that lead to API requests is essential. This lets you know if you need to make updates to views or components you aren't directly working on. This one big benefit of writing unit tests because it gives you coverage across areas that haven't been updated.

Edge cases

There are some odd situations that come up during your development or QA testing that you can write unit tests for. Anywhere that you have a complex workflow, check for edge cases that you can test. You'd be surprised how many times these cases get triggered by unrelated changes.

Back-end testing

There are a different set of concerns for the back-end because we don't have to worry about what's rendered on a page. You're more focused on permissions around requests, the data in responses, and interactions with third party services. From what I've seen, writing tests on the back-end can be even more simple than writing tests on the front-end because the scenarios you run are more focused on code interactions instead of user behaviors.

Let's go through a few common test cases.

Authorization flows

We included authentication testing on the front-end and it can and should be included on the back-end as well. Even so, we need to have tests that make sure users only have access to the data their permissions allow. For example, we don't want unpaid users to have access to the same features as paid users. You'll likely be working with a project manager to help define the different levels of access for users.

NOTE

If you're working on an app that doesn't have a good authorization system, lead the change for that. Many companies give users more access than they need, which opens their apps to a number of attacks.

Third-party services

If you're expecting a response from a third-party service, you want to make sure that you're getting the data you expect. Sometimes third-parties make unnanounced updates to their responses which can break your application. So anywhere there is a call to another service, there should also be a test in place.

Request parameter validation

This is an area where you'll work closer with the front-end because it has to know what errors to expect so it can determine what message to show users as a security measure. Even though the front-end will do validation on user inputs, you are accounting for times when someone accesses your API directly, like through an endpoint. Any request parameters that affect something in the database should always have validation around them to help prevent security attacks, like SQL injection.

Database transactions

Since the back-end interfaces directly with the database, we want to make sure the data we're querying has the values we need. There might be jobs or stored procedures that execute at set times of the day and we want to test to check that those values are up to date. There might be some data we need to update or delete that should be validated before any actions are made. The database is where all changes become permanent and propagate to anything else that needs that data.

Tagging releases

Think of a tag as a branch that can't be changed. It's a version of your code that has been locked in place so that it's stable for future use.

Many times, you'll see tagged releases for different versions of packages. It's why you're able to downgrade a package to a previous version if you need to.

By tagging releases for your applications, you can also take advantage of semantic versioning. *Semantic versioning* is a number that has three components. The first number indicates the major version to alert everyone of major, breaking changes. The second number indicates the minor version to show when new features are added, but don't introduce breaking changes. The third number is for patches, like bug fixes.

[[fig 5.7-1]] .Semantic versioning diagram image::images/image-to-come.png["semantic versioning diagram"]

This not only helps communicate changes to other developers, it also helps with versioning in the CI/CD pipeline. When it's time to release changes and run your CI/CD pipeline, you have to decide what action will kick off the process. You'll notice a lot of open source tools have different tagged releases in their GitHub repos. That's because tagging is a great release strategy.

By tagging a certain branch to be released, you tie each of your deploys to a specific code artifact that you can refer to later. This gives you flexibility in which changes execute a CI/CD run and where those

changes automatically get deployed, while documenting all of the changes bundled in release.

This is another area for you to shine. Work with your DevOps team to decide which branches in the repo will correspond to the different deploy environments. Then you can choose naming conventions for branches and tags that make sense within the CI/CD process. The important part is the developers and the DevOps team agree on a release strategy and document it well.

These are the strategies behind the scenes that keep applications in a stable state and deploying on time.

Rolling back deploys

There will inevitably be times when code changes bring staging or production down and cause a panic. Before you start trying to push hotfixes or start a super deep dive into the problem, see if you can do a quick rollback of your changes. The quickest way to figure out if a rollback is the right approach is to do a rollback locally.

It sounds obvious, but you'd be surprised how many companies have other strategies in place. If you're following the tagging strategy, you can just deploy the previous version to get staging back up as soon

as possible. That's why it's important to have a release strategy: so that you can handle any issues that get deployed.

What to do without tagged releases

If you aren't using tagged releases, you're probably manually merging changes to different branches. This get a little hairy depending on the branch deploy strategy in place, because you can end up with several environments in different states. Here's a potential way to untangle things and get a clean rollback to the right environment.

Look at the latest changes deployed to the broken environment

When staging breaks immediately after release gets deployed, check the console to see where the errors are coming from or check the logs. If it's clearly connected to the code that just went out, find the pull request that was merged related to it and create a revert PR for it.

Update the branches related to other environments effected by the revert

Once you've gotten staging back up, update any branches that have those changes. Propagating the changes across the board keeps everything consistent so you don't end up with those odd states. Make sure that you check that all of the environments work correctly so you know you have a good reset.

Get a new fix ready

Fix the issue that brought down staging locally and get a new release branch up. When you get the fix up, double check that you have any other changes that got rolled back. Take your time going through this process. This is also something senior developers do. It's ok if it takes time to thoroughly check your code. It's much better to do your double checks and get a solid deploy out to staging.

We've covered many of the considerations that go into deploying code to staging from cloud platforms to data handling. These are some of the concerns you'll encounter regardless of the type of app or industry you're working in. There will be some fine-tuning to get the process down, but this is great starting point.

Chapter 5. Testing

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at milecia.mcgregor@gmail.com.

One of the best things you can do to help yourself and future developers on a project is write tests. Unit tests and automated tests will catch *so* many unexpected errors. The best projects I've ever worked on had some great test suites written up that were constantly maintained. They would catch some of bugs that weren't always directly tied to changes in one particular place. For example, changing the modal behavior on one page revealed that another page was referencing the wrong modal.

Well-written tests will also give you and everyone else a lot more confidence when it's time to deploy changes to users. There is a cost to implementing tests though and it comes in feature development time. It's much faster to get a feature or bug fix out by skipping tests. Sometimes this is unavoidable due to critical patches or looming deadlines. That doesn't mean to skip testing entirely.

Some amount of code coverage is better than none at all.

Why bother testing

I'm a huge advocate of writting tests as you write code, but it does take time to write those tests that could be spent on other tasks. The reason a senior developer needs to know about testing, how to do it, and why it's important is because many projects need an advocate for this. Many applications that don't have some type of code testing in place have consistent, constant issues with deploying changes to users.

Here are some of the key reasons that you should push for more code coverage.

They catch unexpected bugs

This is helpful for projects at any stage, even legacy projects. As you start adding code coverage, you'll catch things like un-

expected functions being called, rendering components in odd states, or not getting the correct pagination results based on a search query. You'll see more of a benefit with more tests because it will start to reveal interesting connections in places you don't check for.

They act as your second pass

Even if you are lucky enough to have a QA team to work with, there will be functionality that can always be checked automatically. This can give QA more time to focus on finding edge cases and testing larger releases. These tests are also run earlier in the SDLC so you can see the results locally before changes go to any other environment.

They make you think about the code from different angles

The process of writing tests makes you come up with different scenarios your code might face. You can keep the scenarios as high level or as granular as you need. Regardless, it will help you get a better understanding of how components and features are supposed to work.

They save time and money

Think of all the time you save by not having downtime in production. Depending on the type of application you're working

on, this could also save you and your users a lot of money. Because issues are caught very early in the development process, less time is spent trying to fix issues. It brings those out early so that you can talk with the product team about functionality expectations.

Now let's take a look at writing some common test scenarios on a React application for the front-end and an Express application for the back-end.

Front-end testing in React

We're going to write tests for a few scenarios that many apps have in common.

Authentication flow

You'll definitely need to make sure the way users log in stays functional. Some of the things we'll include in this test are making sure the right components are rendered and checking any changes in the responses we get. Here are what some tests written in Jest might look like.

```
describe('Authentication', () => {
  beforeEach(() => {
```

```
    moxios.requests.reset()

    moxios.stubRequest(/.*sign_in*/, {
      status: 200,
      response: FIXTURES.SIGN_IN_RESPONSE
    })

    moxios.stubRequest(/.*verify_otp/, {
      status: 200,
      response: FIXTURES['2FA_RESPONSE']
    })
  })

  it('allows a user to log in', (done) => {
    const { getByText, getByPlaceholderText } = r

    const emailInput = getByPlaceholderText('emai
    const passwordInput = getByPlaceholderText('p
    const btn = getByText('Log In')


    fireEvent.change(emailInput, { target: { valu
    fireEvent.change(passwordInput, { target: { v

    fireEvent.click(btn)

    moxios.wait(async () => {
      const req = moxios.requests.at(0)
      const body = JSON.parse(req.config.data)
```

```
      expect(body.email).toBe('test@example.com')
      expect(body.password).toBe('password')

      await waitForElement(() =>
        getByText('Login with two-factor authenti
      )

      done()
    })
  })

  it('allows a user to request a password reset',
    const { getByText, getByPlaceholderText } = r

    const btn1 = getByText('Forgot Password?')

    fireEvent.click(btn1)

    const input = await waitForElement(() =>

      getByPlaceholderText('Email Address')
    )
    const btn2 = getByText(
      'Send me password reset instructions.'
    ) as HTMLButtonElement

    fireEvent.change(input, { target: { value: 't
    fireEvent.click(btn2)
```

```
      moxios.wait(() => {
        const req = moxios.requests.mostRecent()
        const body = JSON.parse(req.config.data)

        expect(body.email).toBe('test@example.com')

        done()
      })
    })
  })
```

Rendering error messages

When we're handling responses from APIs, which is almost always, there will be errors that come up that we need to handle to keep a smooth experience for users. This is one area of testing that you can work with the back-end and learn what to expect from it in different cases. You'll need to know what errors to expect so you can determine what messages to show users.

NOTE

Never display raw server errors to end users. This can open some security vulnerabilities depending on the level of information returned from the back-end. It's a best practice to show more user-friendly messages that don't reveal anything about the underlying system. For example, it might be better to tell a user something is wrong with their login credentials compared to telling them whether it's the user name or password.

Here are some examples of error message testing.

```
describe('Countries > Error', () => {
  beforeEach(() => {
    moxios.install()

    moxios.stubRequest(/.*countries.*/, {
      status: 500
    })
  })

  afterEach(() => {
    moxios.uninstall()
  })

  it('displays an error notice', async () => {
    const { getByText } = renderChunk(
      <CountryReport siteId={1} dateParams={PARAM
    )
```

```
      await waitForElement(() => getByText('Retry?
    })

    it('should handle API GET errors correctly', (d
      const rendered = renderChunk(<Users />)

      moxios.wait(async () => {
        const req = moxios.requests.mostRecent()

        req.respondWith({
          status: 400,
          response: {}
        })

        await waitForElement(() => rendered.getByTe

        done()
      })
    })
  })
```

Checking API responses

There's a shape of data you're expecting from every back-end request. It's important to make sure you're still getting the results you

expect each time. These are probably the tests that have saved me the most time on projects when doing full-stack development. Sometimes you rename a variable and having that extra check catches things like this.

Here are a few tests for some API responses.

```
describe('Countries Report', () => {
  beforeEach(() => {
    moxios.install()

    moxios.stubRequest(/.*summary.*/, {
      status: 200,
      response: COUNTRY_SUMMARY_RESPONSE

    })
    moxios.stubRequest(/.*countries.*/, {
      status: 200,
      response: COUNTRY_RESPONSE
    })
  })

  afterEach(() => {
    moxios.uninstall()
  })

  it('displays data', async () => {
    const { getByText, getAllByText } = renderChu
```

```
      <CountryReport siteId={1} dateParams={PARAM
    )

    await waitForElement(() => getByText('Shaolin

    await waitForElement(() => getByText('$1,000.

    getByText('1,000')

    getAllByText('80.0%')

    getByText('$1.20')

    getByText('$1.10')


    getByText('95.0%')

    getByText('92.0%')
  })

  it('displays pageview data', async () => {
    const { getByText, getAllByText } = renderChu
      <CountryReport siteId={1} dateParams={PARAM
    )

    await waitForElement(() => getByText('1,200',

    getAllByText('75.0%')
```

```
    })
})
```

Rendering components

You can test business flow with unit tests and that can help detect things like project drift. Instead of just making sure things work, you can check that components are rendered based on different actions. For example, you might have a workflow for users to create new games that involves different modals and API requests. Making sure those execute in the right order with the expected results is a good check to have.

Here is a test for going through a game creation workflow.

```
describe('existing sites', () => {
    beforeEach(() => {
        moxios.install()

        moxios.stubRequest(/.*earnings.*/, {
        status: 200,
        response: { earnings: [] }
        })
    })

    it('displays a notice if a site has given te
```

```
            const { getByText } = renderGivenNotice(<

            await waitFor(() => getByText('We have re
        })
    })
```

Back-end testing in Express

Now we're going to turn our attention to the back-end and write tests for a few scenarios that many apps have in common.

Authorization flow

Permissions around data are handled on the back-end. That way there's a much lower chance of sensitive information leaking to the front-end. For example, a user can log into their account without two-factor authentication, but they aren't able to edit their profile information. Or a user isn't able to see the same data that a company employee can see because of their admin priveleges.

Here is what some authorization tests might look like in an Node app using Jest.

```
describe('campaign tests', () => {
    let client: ApolloClient<any>;
```

```
    let adminClient;
    let shopId;
    let productId;

    before(async () => {
        adminClient = createApolloClient({
            token: (await getToken('admin@sass.te
        });
    });

    beforeEach(async () => {
        const {product, shop, shopOwnerClient} =
        client = shopOwnerClient;
        shopId = shop.id;
        productId = product.id;
    });

    it('should create video campaign', async () =
        const input = fakeVideoCampaignInput(shop
        const campaign = await createCampaign(cli

        expectBaseCampaign(input, campaign);

        expect(campaign).has.property('budget').t
            maxSpend: input.budget.maxSpend,
            maxDailySpend: input.budget.maxDailyS
            limitOfImpressions: input.budget.limi
        });
```

```
            expect(campaign).has.property('video');
            expect(campaign.video.videoUrl).equals(in
        });
    });
```

Calculation checks

Any data-heavy calculations should be handled on the back-end to keep the front-end running as fast as possible. This is another great place to check business logic. What *are* the equations for the calculations we do? This is a good place to record this type of information.

Here are some calculation tests. Hopefully you can see why we might not want to do this on the front-end.

```
describe('breakdown service tests', () => {
    it('should correctly calculate items', async
        process.env.MERCHANT_REFERRAL_FEE = '0.15
        process.env.MERCHANT_PROCESSING_FEE = '0.
        process.env.MERCHANT_HANDLING_FEE = '0.5
        process.env.MERCHANT_EBRIDGE_FEE = '0.50

        const shippingServiceMock = mock(Shipping
        when(shippingServiceMock.calculateCost(an
```

```
        const taxServiceMock = mock(TaxService);
        when(taxServiceMock.calculateTax(anythin
            tax: {
                amount_to_collect: 3,
            },
        } as TaxForOrderRes);

        const rewardServiceMock = mock(RewardServ
        when(rewardServiceMock.calculateSalesEnha

        const breakdownService = new BreakdownSer
            instance(shippingServiceMock),
            instance(taxServiceMock),
            instance(rewardServiceMock),
        );

        const shop1 = getShop();
        const shop2 = getShop();
        const game = getGame();
        const shippingInfo = getShippingInfo();

        const items: ICheckoutItemBase[] = [
            getBreakdownItem(shop1, game, shippi
            getBreakdownItem(shop1, game, shippi
            getBreakdownItem(shop2, game, shippi
        ];
        const checkoutItems = await breakdownServ
        expect(checkoutItems).lengthOf(items.leng
```

```
        expect(checkoutItems[0]).includes({
            shippingCostAmount: 0,
            taxAmount: 3,
            salesEnhancementAmount: 7,
            retailAmount: 66,
            amount: 69,
            paymentProcessingAmount: 2.33,
            referralAmount: 4.77,
            gameOwnerAmount: 4.77,
            vendorAmount: 46.62,
        });
        expect(checkoutItems[1]).includes({
            shippingCostAmount: 0,
            taxAmount: 3,
            salesEnhancementAmount: 7,
            retailAmount: 66,
            amount: 69,
            paymentProcessingAmount: 2.33,
            referralAmount: 4.77,
            gameOwnerAmount: 4.77,
            vendorAmount: 46.62,
        });
        expect(checkoutItems[2]).includes({
            shippingCostAmount: 0,
            taxAmount: 3,
            salesEnhancementAmount: 7,
            retailAmount: 66,
            amount: 69,
```

```
                paymentProcessingAmount: 2.17,
                referralAmount: 4.79,
                gameOwnerAmount: 4.79,
                vendorAmount: 46.76,
            });
        });
    });
```

Validating inputs

Even though there will be input validation on requests coming from the front-end, we still have to account for times someone accesses an API directly. That's why we include that same validation on the back-end.

Here are a few tests showing how to validate inputs. These are some you can put in for quick wins.

```
describe('basic tests', () => {
    it('should throw invalid credentials', async
        let token;
        try {
            token = await getToken('bad@email.tes
        } catch (e) {
            expect(e.error.statusCode).eqls(401);
        }
        expect(token).eqls(undefined);
```

```
        expect(token).eqls(undefined);
    });
});
```

Updating data

Users will make requests to update their data in some way, even if it's deleting things. We want to make sure the right data is going to the database. So we check for types, any transformations that need to happen, and schema changes. If a column name gets changed, that could break the app and these tests can help catch that before it even gets to QA.

Here are some tests for database transactions.

```
describe('game tests', () => {
    const EMAIL = 'game-owner@sass.test';
    const PASSWORD = 'Password1';

    let client: ApolloClient<any>;

    beforeEach(async () => {
        client = createApolloClient({
            token: (await getToken(EMAIL, PASSWO
        });
    });
```

```
    it('should create game for organization', asy
        const adminClient = createApolloClient({
            token: (await getToken('admin@scusass
        });

        const OWNER_EMAIL = faker.internet.email(
        const OWNER_FULL_NAME = `${faker.name.fir

        const organization = await createOrganiza
            ...fakeOrganizationInput(),
            owner: {
                email: OWNER_EMAIL,
                fullName: OWNER_FULL_NAME,
            },
        });

        const input: GameInput = {
            ...fakeGameInput(),
            organizationId: organization.id,
        };

        const {id} = await createGame(adminClient

        await verifyEmail(OWNER_EMAIL);
        await setPassword(OWNER_EMAIL);

        const ownerClient = createApolloClient({
```

```
            token: (await getToken(OWNER_EMAIL)).
        });

        const game = await getGame(ownerClient, i

        expectGame(game, input, {email: OWNER_EMA
    });

    it('should create my game', async () => {
        const input: BaseGameInput = fakeGameInpu

        const {id} = await createGame(client, inp

        const game = await getGame(client, id);

        expectGame(game, input, {email: EMAIL.toL
    });

    it('should create and update game', async ()
        const input: BaseGameInput = fakeGameInpu
        const game = await createGame(client, inp
        const updatedGame = await updateGame(clie
            ...input,
            id: game.id,
        });
        expect(game).eqls(updatedGame);
    });
});
```

You'll probably organize your mock functions and fake data differently, but these are some tests you can look at to get thoughts on how to start writing your own.

Other types of automated tests

Unit tests are awesome and they catch a lot of issues early in the development process. There are other tests we can add to catch issues at a different stage though. You might want to run automated integration tests as well. This is something that can be handled by a QA team or it might be handled by a Software Development Engineer in Test (SDET).

SDETs are usually developers that have had a lot of experience with writing tests in different suites. So they are familiar with why tests are important, how the code works together, and they usually work with JavaScript or Python. You probably won't write a lot of these tests because they take more time than unit tests.

As a senior developer though, knowing how to write a few of these will be useful.

End-to-end tests with Cypress

There are a couple of tools you'll hear commonly used in end-to-end (E2E) testing: Cypress and Selenium. While Selenium will let you write in JavaScript, it's more commonly written in Python. Cypress is JavaScript only and that's the tool we'll be writting a few tests with.

It tests actually user flows through the UI by targeting elements and interacting with them like users through click actions and typing. You can automate tests for components in any of the JavaScript frameworks and you can test APIs as well. Most E2E testing is built on top of Selenium, but Cypress isn't so it executes differently. Let's take those tests now.

The first test will check that a user can go to a form, select an option from a dropdown, and submit that form.

```
import { User } from "../../../src/models";
import { isMobile } from "../../support/utils";

type NewTransactionTestCtx = {
  allUsers?: User[];
  user?: User;
  contact?: User;
};

describe("New Transaction", function () {
  const ctx: NewTransactionTestCtx = {};
```

```
  beforeEach(function () {
    cy.task("db:seed");

    cy.intercept("GET", "/users*").as("allUsers"

    cy.intercept("GET", "/users/search*").as("use

    cy.intercept("POST", "/transactions").as("cre


    cy.intercept("GET", "/notifications").as("not
    cy.intercept("GET", "/transactions/public").a
    cy.intercept("GET", "/transactions").as("pers
    cy.intercept("PATCH", "/transactions/*").as('

    cy.database("filter", "users").then((users: U
      ctx.allUsers = users;
      ctx.user = users[0];
      ctx.contact = users[1];

      return cy.loginByXstate(ctx.user.username);
    });
  });

  it("navigates to the new transaction form, sele
    const payment = {
      amount: "35",
      description: "Sushi dinner ^弦
    };
```

```
    cy.getBySelLike("new-transaction").click();
    cy.wait("@allUsers");

    cy.getBySel("user-list-search-input").type(ct
    cy.wait("@usersSearch");
    cy.visualSnapshot("User Search First Name Inp

    cy.getBySelLike("user-list-item").contains(ct

    cy.visualSnapshot("User Search First Name Lis

    cy.getBySelLike("amount-input").type(payment.
    cy.getBySelLike("description-input").type(pay
    cy.visualSnapshot("Amount and Description Inp
    cy.getBySelLike("submit-payment").click();
    cy.wait(["@createTransaction", "@getUserProfi
    cy.getBySel("alert-bar-success")
      .should("be.visible")
      .and("have.text", "Transaction Submitted!")

    const updatedAccountBalance = Dinero({
      amount: ctx.user!.balance - parseInt(paymen
    }).toFormat();

    if (isMobile()) {
      cy.getBySel("sidenav-toggle").click();
    }
```

```
      cy.getBySelLike("user-balance").should("conta
      cy.visualSnapshot("Updated User Balance");

      if (isMobile()) {
        cy.get(".MuiBackdrop-root").click({ force:
      }

      cy.getBySelLike("create-another-transaction"
      cy.getBySel("app-name-logo").find("a").click(

      cy.getBySelLike("personal-tab").click().shou
      cy.wait("@personalTransactions");

      cy.getBySel("transaction-list").first().shou

      cy.database("find", "users", { id: ctx.contac
        .its("balance")
        .should("equal", ctx.contact!.balance + par
      cy.getBySel("alert-bar-success").should("not
      cy.visualSnapshot("Personal List Validate Tra
    });
  });
```

This next test will check the authentication flows that a user can go through.

```
import { User } from "../../../src/models";
import { isMobile } from "../../support/utils";

const apiGraphQL = `${Cypress.env("apiUrl")}/grap

describe("User Sign-up and Login", function () {
  beforeEach(function () {
    cy.task("db:seed");

    cy.intercept("POST", "/users").as("signup");
    cy.intercept("POST", apiGraphQL, (req) => {
      const { body } = req;

      if (body.hasOwnProperty("operationName") &&
        req.alias = "gqlCreateBankAccountMutation
      }
    });
  });

  it("should allow a visitor to sign-up, login, a
    const userInfo = {
      firstName: "Bob",
      lastName: "Ross",
      username: "PainterJoy90",
      password: "s3cret",
    };

    // Sign-up User
```

```
    cy.visit("/");

    cy.getBySel("signup").click();
    cy.getBySel("signup-title").should("be.visibl
    cy.visualSnapshot("Sign Up Title");

    cy.getBySel("signup-first-name").type(userInf
    cy.getBySel("signup-last-name").type(userInfo
    cy.getBySel("signup-username").type(userInfo.
    cy.getBySel("signup-password").type(userInfo.
    cy.getBySel("signup-confirmPassword").type(us
    cy.visualSnapshot("About to Sign Up");
    cy.getBySel("signup-submit").click();
    cy.wait("@signup");

    // Login User
    cy.login(userInfo.username, userInfo.passwor

    // Onboarding
    cy.getBySel("user-onboarding-dialog").should(
    cy.getBySel("list-skeleton").should("not.exis
    cy.getBySel("nav-top-notifications-count").sh
    cy.visualSnapshot("User Onboarding Dialog");
    cy.getBySel("user-onboarding-next").click();

    cy.getBySel("user-onboarding-dialog-title").s

    cy.getBySelLike("bankName-input").type("The B
```

```
      cy.getBySelLike("accountNumber-input").type('
      cy.getBySelLike("routingNumber-input").type('
      cy.visualSnapshot("About to complete User Onb
      cy.getBySelLike("submit").click();

      cy.wait("@gqlCreateBankAccountMutation");

      cy.getBySel("user-onboarding-dialog-title").s
      cy.getBySel("user-onboarding-dialog-content")
      cy.visualSnapshot("Finished User Onboarding")
      cy.getBySel("user-onboarding-next").click();

      cy.getBySel("transaction-list").should("be.vi
      cy.visualSnapshot("Transaction List is visibl

      // Logout User
      if (isMobile()) {
        cy.getBySel("sidenav-toggle").click();
      }
      cy.getBySel("sidenav-signout").click();
      cy.location("pathname").should("eq", "/signin
      cy.visualSnapshot("Redirect to SignIn");
    });
  });
```

The last example test will cover an API test where we need to update some user data.

```
import { faker } from "@faker-js/faker";
import { isEqual } from "lodash/fp";
import { User, NotificationType, Transaction, Ban

type TestTransactionsCtx = {
  receiver?: User;
  authenticatedUser?: User;
  transactionId?: string;
  notificationId?: string;
  bankAccountId?: string;
};

const getFakeAmount = () => parseInt(faker.financ
const apiTransactions = `${Cypress.env("apiUrl")}

describe("Transactions API", function () {
  let ctx: TestTransactionsCtx = {};

  const isSenderOrReceiver = ({ senderId, receive
    isEqual(senderId, ctx.authenticatedUser!.id)

  beforeEach(function () {
    cy.task("db:seed");

    cy.database("filter", "users").then((users: U

      ctx.authenticatedUser = users[0];
      ctx.receiver = users[1];
```

```
      return cy.loginByApi(ctx.authenticatedUser.
    });

    cy.database("find", "transactions").then((tra
      ctx.transactionId = transaction.id;
    });

    cy.database("find", "notifications").then((no
      ctx.notificationId = notification.id;
    });

    cy.database("find", "bankaccounts").then((bar
      ctx.bankAccountId = bankaccount.id;
    });
  });

  context("PATCH /transactions/:transactionId", f
    it("updates a transaction", function () {
      cy.request("PATCH", `${apiTransactions}/${c
        requestStatus: "rejected",
      }).then((response) => {
        expect(response.status).to.eq(204);
      });
    });


    it("error when invalid field sent", function
      cy.request({
        method: "PATCH",
```

```
          url: `${apiTransactions}/${ctx.transactio
          failOnStatusCode: false,
          body: {
            notATransactionField: "not a transactio
          },
        }).then((response) => {
          expect(response.status).to.eq(422);
          expect(response.body.errors.length).to.eq
        });
      });
    });
  });
```

E2E tests are incredibly nice to have. If you can squeeze in the time on a project to write these, they can be game changers.

Chapter 6. Performance Enhancement

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 7th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at milecia.mcgregor@gmail.com.

One goal for every app is to be as fast as possible. Optimizations can be made at almost every layer of the app from the front-end to the database. At the database layer, you might look into creating views and writing stored procedures that keeps them up to date depending on how the schema is defined. At the API layer, you might run tasks concurrently to process data faster. On the front-end, you might de-

cide to look at the libraries you have and what you decide to store in the cache.

These are the things we'll explore in this chapter. This is one of those other areas that senior developers have on their checklist of things to always check for. One project I worked on had API responses that would take upwards of 10 seconds. A quick audit of the front-end and back-end showed that a lot of large calculations were happening in the browser. Moving them to the back-end dropped that response time to 2 seconds.

Finding places that can be improved has bigger effects than knowing your code is well-written, even though that's a great feeling. It also means that users have a better experience, it increases the profit for a company over the long-term, and it make the developer experience better.

We're going to cover a few ways that you can improve the performance of any JavaScript app.

Bundling

This is one of the first things you'll hear about when it comes to front-end performance optimization. A bundle is a set of the CSS and JavaScript files it takes to run your project. Over the years, web apps

have become much more than just a few JavaScript files you import into an HTML file. A project could have hundreds of files it depends on to run, which means it could have multiple bunles.

The front-end is the part of the app that accounts for most of the load time. Any JavaScript code needs to be executed *and* the page has to be rendered. A leading cause for front-end apps to load slow is because they spend most of the time trying to execute all of the JavaScript files on the network. This means that the initial time to load a page can take forever.

All of those dependencies that make development easier also increase that bundle size and the initial load time of a page. I've seen some projects that have had bundle sizes of over 500KB which is huge! Those apps took forever to load in both production and development. Handling bundle size can be challenging as your app grows, but there are a few methods that make it more manageable.

Code splitting

This is the primary way we try to manage bundle sizes. Code splitting lets you split your code into multiple bundles that are either loaded in parallel or as the code is needed. The problem with large bundle sizes is that upfront requirement to download everything for the app

all at the same time, even if a user never goes to a page. Code splitting gives us a way to only load the code that is necessary for the current view.

A couple of tools you'll see used for this are Webpack and Browserify. They can be used to make multiple bundles that get loaded when needed. No matter how many bundles you have, code splitting doesn't change that. It just reorganizes things. When you use tools like these to split code, be aware that it might increase the total amount of code you have by a few bytes.

Once you have something like Webpack configured, you can start splitting the code up. You can do this in a few ways.

Avoid bundle duplication

Sometimes you have packages that are used in multiple places throughout the app and this can make code splitting tricky. Remember, everything we do has a trade off. In this case, we can reuse the same code in different areas of the app without having multiple areas trying to load the same code when it can be loaded once.

Have multiple entry points

An entry point is a way to split code based on the files needed for each page. This is a more manual approach, but it gives you a way to

only load the packages and code for the current page without worrying about the others. A drawback to this approach is that it doesn't let you dynamically split the code. Everything is split up in advanced no matter how the app changes. So you'll need to manually update the way code is split in this case.

Dynamically load bundles

It's very common to see React projects taking advantage because this framework helped make code splitting popular. In React, you can split everything if you want although that's not a good strategy. Some people tend to go overboard when they start code splitting for the first time and try to split out as many components as they can.

To find that balance of when and where to split code, start by looking at these areas.

- Large third-party packages or components
- Individual pages based on your routes
- Large conditionally rendered components like modals or dropdowns

Once you have the code split between these three areas, you can start the normal flow of dynamically loading code throughout your project with these steps.

- Use a default export for a component that we want to code split
- Import the component with `React.lazy`
- Render the component as a child of `React.Suspense`
- Provide a fallback component to `React.Suspense` for when the main component is loading

Here's what that might look like programatically:

```
import React, { Suspense } from 'react';

const Report = React.lazy(() => import('./Report

function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <Report />
      </Suspense>
    </div>
  );
}
```

Another consideration is grouping dependencies by the frequency that they update. Some dependencies you have in your project, like React and React DOM, update significantly slower than others.

Those can be bundled together and kept separate from the other dependencies that change more often.

Bundle size also effects other metrics that are harder to measure but just as important like power usage, memory usage, execution time, and compile time. These are also metrics you should try to observe so you can find even more areas of the app to optimize.

Splitting the back-end into microservices

Most of the performance optimization for an app will happen on the front-end because that's the part that effects users the most. However, there will be times when you find the problem isn't with the front-end rendering, but the response time from an API. Some APIs are responsible for a lot of data manipulation and depending on how the project has been architectured, it might be hard to trim the response time.

One option you have is to start spinning up mircroservices to take out some of the bloat from an API. A microservice is almost like a standalone endpoint. This could help speed up the repsonse time by trimming out some of the other background things happening in an API. It

can also help make debugging a little easier because you don't get lost in a tangle of code.

This is what makes microservices so resilent. Because they are so simplified, when any issues come up the downtime usually isn't as long. You're also able to have more confidence that your hotfixes won't introduce bugs in other unexpected areas. Microservices also solve the problem of getting need functionality out quickly without needing to redeploy an entire system.

Now instead of the front-end waiting for the back-end to go through many different checks and workflows, it can directly get the data it needs for users. You can even use different languages when you split functionality out into microservices. I remember one client I worked with had Python and Node microservices. The Python one was used for machine learning predictions and the Node one was used for other data requests.

Another thing mircroservices allow you to do is scale specific bits of functionality. With a full back-end app, you have to scale all of the endpoints equally because they're all on the same server. Microservices allow you to break from that. So if you know you have a reporting endpoint that gets 3x the number of requests as the others, this is a great candidate for a microservice.

By giving the most requested functionality the ability to scale independently, you can start to see performance gains quickly. There are some drawbacks to working with microservices though. You have to really make sure the back-end team understands how all of the functionality impacts each other. Without this core understanding, the scope of microservices can start to overlap and introduce strange side effects that take much longer to track down.

That's why a common strategy may include starting with an API and monitoring which endpoints get the most requests or finding which endpoints have the longest response times. Those could be parts that you extract out of the whole API and make microservices. You already know how they fit into the overall app and you already know these are areas that needed optimization.

Microservices definitely shouldn't be jumped into until you've tried performance optimization on your APIs. However, if you've already tried everything else, implementing a few microservices is another path to explore. The upfront work will take time because you still have to factor in things like security and monitoring, but once it's in place it can drastically improve your response time.

Data fetching

Another area that can be optimized is the data layer. This is something that JavaScript developers probably won't touch, but if you're working on the back-end, you could be responsible for new migrations. You can look for ways you can offload some of the calculations to the database. The main thing you're trying to speed up will likely be large queries.

Some of the strategies we'll cover include:

- Writing stored procedures to create views
- Writing better SQL queries
- Database sharding

Write stored procedures

A lot of the calculations for an app are handled on the back-end. One way to speed this up is to move intensive calculations back to the database. When you write a stored procedure, you can generate a new table based on the calculations with new column names called a view. Views can be updated on a scheduled interval, whether it's daily, weekly, or some other schedule.

Creating views can also help limit the number of tables you have to query. Here's an example of a stored procedure that creates a view based on a couple of tables.

```
CREATE PROCEDURE uspCreateView
   AS EXEC ('
       CREATE VIEW vwDataLayoutFileAssignment
       AS
       better
       SELECT b.FileID, d.FieldID
       FROM [File] b, DataLayoutFileAssignment c
       WHERE b.DriverFileID = c.FileID
       AND C.DataLayoutID = d.DataLayoutID
   ')
```

Write more specific SQL queries

Something else that can help speed up those queries is writing them more efficiently. Instead of using `*` in your queries, only fetch the specific columns you need. That way the query doesn't need to run through the entirety of the table and can pull only the required data. This also helps to eliminate `DISTINCT` queries. Instead of using `SELECT DISTINCT` to get unduplicated records, you can define a combination of fields that will uniquely identify a row.

For example, say we have a `Customer` table and we're running the following query to get unique data.

```
SELECT DISTINCT FirstName, LastName, State
FROM Customer
```

We could rewrite this and save some query time with the following update.

```
SELECT FirstName, LastName, Address, City, State,
FROM Customer
```

Having a few more fields to fully define the record your looking for can be faster and more accurate than the underlying `GROUP` action that runs on `DISTINCT` queries.

The database schema

Before we talk about database sharding, it's important to really take a look at the database schema and how large your database really is. If you have a straight forward schema where the tables are relatively isolated from each other, breaking the database apart might be worth the upfront cost of this migration. If you're working with tables that are heavily related as defined by the schema, splitting the database with sharding might be pretty hard.

This isn't something that will come up often and it'll only happen on extremely large databases where performance has really become a

problem. Sharding is a last resort after every other optimization strategy has been tested.

Database sharding

If you can confidently spread your data across multiple database instances, this is sharding. Each table in each database will have the exact same column names. The difference is that they will all have different row data. When you're working with really big databases, this is an option to consider. It is a huge undertaking and will likely fall on the operations team.

It's just something good to know about as a developer. If you get to sit in on those discussion, ask questions about the architecture for this sharded database. This is a crucial part to get right because if it's not handled correctly, there's a risk of data being lost or having corrupted tables.

Package bloat

Since you know about bundling and how the amount of code we have has a direct impact on how well the front-end of the app runs, let's talk about how that becomes an issue. There's a package for everything these days. Do you need to parse XML responses from a SOAP server? There's a package for that. Have you ever needed to make a

machine learning model using something other than Tensorflow.js? Of course there's a package for that.

Any functionality you can think of probably has a handful of packages already available for you. That's what makes it so tempting to install a bunch of packages and hobble together something good enough for production. The problem is that projects end up with developers that like the packages they're familiar with and they install what they use, slowly adding on to that bundle size.

I've even seen projects that used both Material UI and Tailwind CSS for their UI which was pretty wild. At some point, installing packages became a sort of default for adding more complex functionality. Many times this is fine, but it's worth doing some analysis to see if installing a new package is better than making something internally. Usually internally built functionality will have a smaller code footprint than a package.

The main trade-off between choosing between a package and internally developed functionality is development time. It's much faster to go with something that already exists and will be maintained for the foreseeable future. The problem is that you could find yourself limited in what you can customize as your app grows and potentially changes use.

That's usually when the package boat starts because you need a tool to work around a different tool. There are some ways to combat this and even trim down any existing bloat.

Managing package bloat

Something that you can do periodically is look at some of your larger packages and see if there are smaller, alternative versions that offer similiar functionality. I've seen this strategy trim bundles from 253kb to 37kb. When comparing packages, look for how many dependencies it will need. It's surprising how many things get installed in the `node_modules` folder that aren't really used in the app because of this.

An example of a package that adds a lot of bloat is Jest. Mocha is another testing package that has a lot of the same functionality as Jest, but it uses less dependencies. On the other end, you can also do an audit of your smaller packages to see if it's something that might be better implemented in the project now because things change. Maybe you only use one method from an entire package.

This is a good candidate to extract out and put in your project. A good example for this is the Lodash package. There are a lot of JavaScript developers that use a few of the functions from this package and they could be ripped out and made into project helper functions. These are

just a few of the ways you can watch out for package bloat and trim it down when you're doing performance optimization.

Front-end performance optimization

So far we've covered some of the bigger topics in performance optimization from the front-end to the database layer. Let's cover some specific tools that you can use to find common, fast improvements that can be made on the front-end.

The most used tool is Chrome Lighthouse. You can find this in the Chrome Dev Tools of the browser so there's nothing you need to install. It will go through and check five metrics for your app: performance, accessibility, best practices, SEO, and PWA. You can run it for mobile and desktop versions of your app and it will give you a score for each metric and a list of suggestions you can use to improve the areas.

Some suggestions to improve performance include:

- Minify your CSS and JavaScript, if you haven't already
- Size images properly and lazy load them where possible
- Delete all unused code, especially if things have been commented out
- Avoid using a lot of page redirects

- Cache API responses when possible
- Avoid big layout shifts on a page
- Limit the number of requests you need to make
- Use composited animations

If you've already exhausted these suggestions and you're looking at the other options we have available, `webpack-bundle-analyzer` will take you down the bundle/code splitting rabbit hole. You might also check out Benchmark.js. Even though it hasn't been updated in a while, it's still a commonly used library to see how fast your app is running and where improvements can be made.

Another thing to note is that it's not good to create functions within functions if closures aren't needed because it'll affect processing speed and memory consumption performance. Also, using `async/await` can slow down performance by blocking code execution in async blocks. Just a few programmatic things to be aware of as you work in the code.

Back-end performance optimization

While the bulk of performance optimization for an app will happen on the front-end, the back-end shouldn't be overlooked. The front-end has a lot of the low-hanging optimization fruit because much of it fol-

lows common best practices. The back-end on the other hand can be trickier to optimize because performance is dependent on how many concurrent users are making requests at a time.

You can't confidently say how an API will perform with 100 concurrent users or $10{,}000$ concurrent users. Although you *can* be confident that at some point, having enough users will make the back-end a bottleneck because the responses will be extra slow. As a JavaScript developer you will be consulted with on how to test the back-end performance so knowing some of the steps you can take will have you ready for most scenarios.

NOTE

Even the most experienced back-end developers, it's hard to predict what the bottlenecks in an app might be or how many users it can handle before needing to scale up. If you don't know the answer, ask for time to research the problem further so you can come up with a well-formulated plan.

Test plan

You'll need a plan going into back-end performance testing because it's not as straight-forward as the front-end. You need to know the metrics you're looking for, the environment conditions you're working

with, and the tests you need to run. Here's a quick outline of a plan you can use.

Decide what the testing environment is

The best test environment would actually be production, but it's unlikely you'll convince anyone to let you potentially crash the same servers users are on. See if you can get a replica of the production environment. This includes using the same hardware, network tools and configurations, and any other software.

Performance testing results in a different environment, like staging or dev, don't mean much. Those environments have completely different configurations than production so any performance metrics you get aren't going to reflect what real users experience.

Decide what the key performance metrics are

Some metrics you might look at are: peak response time, error rate, throughput, concurrent users, and requests per second. When you load test an app (use a tool to simulate any number of users interacting with the back-end), these metrics will help you identify any bottleneck areas.

Design the performance tests

Write some test scenarios that users can actually put the app through. Having a sale? See what happens when you have a 10x jump in traffic. Adding some new features? Monitor which endpoints get the most use. You will get a lot of insight about how users interact with your app after you go through these scenarios with product teams.

Run your tests

You have the test environment ready and you have a list of tests and metrics you need, so all that's left is running the tests. Make sure you have logging and monitoring in place here. Knowing things like how much memory and disk space are being used during these tests can help you figure out how to handle any bottlenecks you find.

Write a report and start identifying bottleneck solutions

Once you've run each test at least 3 times, you can write up a report outlining your findings and some solutions. The whole process of performance testing the back-end isn't one you should rush through. So if this takes a few weeks, you're fine. By the time you finish, you will have found areas of the back-end that can be improved that no one expected.

Chapter 7. Next Level JavaScript

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 12th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at milecia.mcgregor@gmail.com.

You're likely already familiar with a lot of concepts in JavaScript. Things like the different array methods and how to use them, DRY principles, and writing effecient code. These are all important to a JavaScript developer at any level.

While it's great to be familiar and comfortable with those concepts, there are a few things that take you deeper into JavaScript and might help you understand how and why we write code the way we do.

In this chapter, we'll spend some time on more advanced JavaScript concepts. You probably work with these all the time and so some of them will be familiar. Others might have never come up in your day to day programming tasks, even though they're always in the background.

We're going to clear up some concepts that will help you understand code better and help you to be able to explain how things work to more junior developers. By the time you finish this chapter, you should:

- Understand and be able to explain some of the more confusing concepts in JavaScript
- See the use cases for these concepts
- Be able to implement them in appropriate situations

We'll start off by talking about closures since this is something we commonly use.

What is a closure

A closure is a is a function wrapped inside an outer function that references the variables in the outer function's scope and it keeps the values in the outer function perserved in the inner function.

That's a quick definition of what a closure is. To really understand the concepts behind what's going on in the closure, let's take a look at some scoping specifics.

A few things about scope

It helps to have a strong grasp of how scoping works in JavaScript. There are different levels of scopes. There's the global scope, local scope, and in some cases, a nested scope. Let's start by taking a look at the global scope.

When a variable or function is in the global scope, that means it can be used in any other functions. Wherever you decide to reference the value, it will be accessible. Here's an example of a globally scoped variable.

NOTE

You'll be able to run all of the code examples here directly in a browser console if you want to see them in action.

```
let item = "popcorn"

function printItem() {
    console.log(item)
```

```
}

printItem() // output: popcorn
```

You see how you're able to reference the `item` value even though it wasn't declared inside of the `printItem` function? That's an example of how the global scope works. You'll be able to access *and* update that value from anywhere in the code.

That's why scoping is so important when we're writing maintainable code. JavaScript gives us the ability to do pretty much anything we want, so we have to keep values in the correct scope to make sure our code works like we expect consistently.

Now let's take a look at a locally scoped variable. We'll expand on the previous example. Since we're printing the item to the console, we might want to include something else in the message we send.

Inside of the `printItem` function, we'll add a new variable called `price`.

```
let item = "popcorn"

function printItem() {
    let price = 5.99
    console.log(+${item} is ${price}+)
```

```
}

printItem() // output: popcorn is 5.99
```

This is where we can see scoping in action. If you try to use the value of `price` outside of the `printItem` function, you'll get a reference error. If you tried something like the following snippet of code, you would see that error.

```
let item = "popcorn"

function printItem() {
    let price = 5.99
    console.log(+${item} is ${price}+)
}

console.log(price) // output: Uncaught ReferenceE
```

With locally scoped variables, you don't have to worry about the value being changed outside of the block the variable was declared in. In this example, no function outside of `printItem` could ever directly update the value of `price`.

let vs. var

One thing I want to note before we continue talking about scope is the use of `let`. There are constant discussions on when to use `var`, `let`, and `const`. When you're working with closures, using `let` to declare variables keeps their scope in the block they're declared in and any sub-blocks.

The main difference between `var` and `let` is that the scope of a variable declared with `var` is the whole function block. That means you can't use the same variable names within nested functions. This kind of behavior will definitely cause some unexpected behavior in closures.

Here's an example of a `var` variable's scope.

```
function printItem() {
    var price = 5.99

    {
        var price = 14.99

        console.log(price)
    }

    console.log(price)
}

printItem() // output: 14.99 14.99
```

You'll notice that if you run this code, the value of `price` will be overwritten with the value in the inner block. This won't happen if we use `let` instead.

```
function printItem() {
    let price = 5.99

    {
        let price = 14.99

        console.log(price)
    }

    console.log(price)
}

printItem() // output: 5.99 14.99
```

Another important difference between `var` and `let` is that if you create a global variable, `let` doesn't create a property on the global object. That means you won't be able to reference any `let` declared variables with the keyword `this`. Here's a quick example of that.

```
var quantity = 7
let price = 3.99
```

```
console.log(this.quantity) // output: 7
console.log(this.price)  // output: undefined
```

That's enough about variable declarations for now, but this is an important difference to note when you're deciding how to declare variables as you write your code in different code blocks.

Turning back to local and global scopes, hopefully these examples helped explain the how the two scopes work. Now we can dig into how closures work.

How it works

A closure takes the values from its outer block and uses them the inside the inner function. So here's an example of a closure.

```
function getPrice() {
    let price = 5.99

    function calculatePrice() {
        console.log(price)
    }

    calculatePrice() // output: 5.99
}
```

```
getPrice() // output: 5.99
```

This is a basic closure. It's a function wrapped inside another function that uses the variable from the outer function. You see that we call the `calculatePrice` function inside the `getPrice` function and it references the `price` value.

When you call the `getPrice` function at the global level, you'll get the `price` in the console. This is all a closure is. It's a nested function that uses variables in within the current scope. Here's another example of a closure with a little more functionality.

```
function getPrice() {
    let price = 5.99

    function calculatePrice() {
        let taxes = 0.085
        let withTaxes = price + (price * taxes)

        console.log(withTaxes.toFixed(2))
    }

    calculatePrice() // output: 6.50

    console.log(price) // output: 5.99
}
```

```
getPrice() // output: 6.50
```

You see how the calculation happens within the `calculatePrice` function and the `price` value doesn't change? That's how closures work! We're able to take the variables from an outer scope, do something with them, and return a new value without altering the original.

There's another way you might use a closure. Let's say you don't want to execute `calculatePrice` within the `getPrice` function because you want the execution to happen at a different time. That would make your code block look like this.

```
function getPrice() {
    let price = 5.99

    function calculatePrice() {
        let taxes = 0.085
        let withTaxes = price + (price * taxes)

        console.log(withTaxes.toFixed(2))
    }

    return calculatePrice
}

let getUserPrice = getPrice()
```

```
let getUserPrice = getPrice()

getUserPrice() // output: 6.50
```

We create the `getUserPrice` variable that executes the `getPrice` function which returns the `calculatePrice` function. Then we call our `getUserPrice` function, which gives us the value we expect.

You might also run into a situation where you need to pass a value directly to your closure function. That's where we get into higher order functions like you might see in React, Angular, or any of the other frameworks. Here's an example of this.

```
function getPrice() {
    let price = 5.99

    function calculatePrice(quantity) {
        let taxes = 0.085
        let quantityPrice = price * quantity
        let withTaxes = quantityPrice + (quantity

        return withTaxes.toFixed(2)
    }

    return calculatePrice
}
```

```
let getUserPrice = getPrice()

getUserPrice(3) // output: 19.50
```

We updated the `calculatePrice` function to accept a `quantity` parameter. In order to call this, we need to do a couple of things that you see. Since `getPrice` returns the `calculatePrice` without executing it, we need to have a variable that calls the `getPrice` function. That's where `getUserPrice` comes in.

Now `getUserPrice` is the `calculatePrice` function because that is what is returned from `getPrice` when we call it. Then you see we call `getUserPrice` and pass it the value `3`. This is the `quantity` that `calculatePrice` is exepecting. Then we finally get the price returned.

The flexibility of closure behavior will let you do a lot of useful things with closures and we'll take a look at some of those use cases.

When you would use one

Private functions

One of the most common use cases for closures is creating private functions and variables in JavaScript. Unlike with strongly typed languages like C# or Java, JavaScript doesn't have an explicit way to declare private functions and variables. So we use closures to fill this gap.

Let's say you have an app where the user needs to know how much they spent the previous day. Here's an quick implementation of that using a closure.

```
function getPreviousSpendData(userId) {
    // this data would likely come from an API ca

    let userData = {
        previous: {
            spent: 151.87,
            made: 459.23
        }
    }

    function calculatePreviousSpendData() {
        let remaining = userData.previous.made -

        return remaining
    }

    return calculatePreviousSpendData()
}
```

```
}

getPreviousSpendData("fwjrigr-2904g09") // output
```

With this example, you might not want the user's data accessible outside of this particular function. That's why we're using it inside of `getPreviousSpendData`. Nothing outside of this code block will be able to reference the user's data and they still get the information they needed.

Function factories

You can also use closures to create function factories. A function factory is a way that you can use a function to create other functions. Here's an example of what a function factory using closures could look like.

```
function priceByQuantity(quantity) {
    let price = 7.99

    return function totalPrice(taxRate) {
        let total = (price * quantity) + (price *

        return total.toFixed(2)
```

```
    }
}

let quantityPrice = priceByQuantity(7)

let priceInOk = quantityPrice(0.085) // output: 6
let priceInCa = quantityPrice(0.0775) // output:
```

This is one of the ways you can use closures to make multiple functions. In this example, we make the `priceByQuantity` function and it takes a `quantity`. Then it has a private variable called `price` and an inner function called `totalPrice` that takes a `taxRate` parameter.

Then you'll notice we call the `priceByQuantity` with a value of `7` in the `quantityPrice` variable. This gives us a function that we can pass tax rates to in order to see what the total price would be in different locations.

Callbacks

Probably the most common use of closures is in callbacks. This is when you pass a function to another function. This happens all the time with array methods that use a function to work with the individual

values in the array. Here's an example of a closure you may have seen before.

```
let items = [
    {
        name: "kumquat",
        price: 2.99
    },
    {
        name: "pineapple",
        price: 4.99
    },
    {
        name: "papaya",
        price: 5.99
    },
]

let itemMessages = items.map(function itemMsg(ite
    return `The ${item.name} costs $${item.price}
})
```

The `itemMsg` function is a callback function and this is also a closure because it takes the current item value and works with it inside the inner function.

These are some of the common use cases for closures, but you'll encounter them in a lot of other implementations. Closures are a core concept in JavaScript and being able to explain it to newer developers can be a little tricky. Just make sure you can recognize when you need them.

NOTE

Make sure that you go through the examples and look for a few places you see closures in your daily coding activities!

The JavaScript event loop

JavaScript is really good at pretending to be multi-threaded even though it runs on a single thread. That means it can only execute one thing at a time. It doesn't matter whether you're on the front-end or back-end, JavaScript only has one thread and it's managed with an event loop.

The reason this is important to understand is because it directly effects when your functions run. The problem with a single threaded application is that whenever a function takes a long time to finish running, it blocks any other actions.

Users won't be able to click anything on the page, no other functions can run, and it might make your app crash. That's why the event loop exists. It's the magic behind the asynchronous behavior in JavaScript.

What makes an event loop

There are a few elements you need to understand before we jump into the full picture of how event loops work. An event loop is made of the heap, call stack, the callback queue, and web APIs.

The **heap** is typically a large, unstructured area of memory. This is where the objects for your app go in the JavaScript runtime.

The **call stack** is what the JavaScript runtime engine uses to manage when functions get executed. It keeps track of all of the functions that have been called. It works on the LIFO (last-in-first-out) principle which means that it executes the last function that was added to the stack first.

The **web API** are threads that you can only make calls to. You don't have any other access to them. These include the DOM(Document Object Model), AJAX (Asynchronous JavaScript And XML), and time-out calls. This is where the any functions that take a while get held until their values are ready. If you work in Node, these are the C++ APIs.

The **event queue** is where the functions that were sitting in the web API get pushed when they're waiting to be called. This is where your async functions sit when you're waiting for a value to return.

How the event loop works

The **event loop** is what orchestrates all of these different elements. It constantly checks the call stack to see if there's anything that needs to be executed. Then it checks the event queue to see if there are any functions awaiting execution.

Let's take a look at a code example to show off how the event loop really works.

```
console.log("This is the first function.")
setTimeout(function asyncMessage() {
    console.log("This is the async function.")
}, 3000)
console.log("This is the last function.")
```

This is what's happening in the event loop.

Figure 7-1. Picture of the complete event loop

You see the order the functions are executed in and how longer functions are managed.

What effects it can have on your code

This is where we get into async calls in our code. You've probably written code where you're waiting for the data from an API request or your waiting for some data to be returned from the database. Then you get an undefined result.

That's because of the way the event loop handles AJAX execution. Your function call will get executed and then sent to the web API stack where it'll wait for the value to get returned. Then it goes to the event queue where it waits on the event loop to add it to the call stack and then execute it.

We'll talk all about how to handle this with promises and with aysnc/await.

Promises

Promises were one of the first clean ways we started handling asynchronous code in JavaScript. Before promises were created, we would end up crazy callback loops affectionately called “callback hell”. These loops would end up looking like this.

```
outerFunc(function(result) {
  innerFunc(result, function(nextResult) {
    doubleInnerFunc(nextResult, function(lastResu
      console.log('Got the last result: ' + lastR
    }, failureCallback);
  }, failureCallback);
}, failureCallback);
```

This is where things tend to get hard to follow and it’s hard to trace where errors are coming from. Which is why promises were introduced.

What is a promise

A promise is just an object that returns a value in the future. If you remember when we discussed event loops, there is some code execution that leads to a function being added to the web API list and then it gets added to the event queue when its ready to be called.

While we're waiting on the function call to move from the event queue to the call stack and get executed, this is where promises come in. There are several states that a promise object can have: pending, fulfilled, rejected.

Every promise object starts off as pending. It's like when you figure out you're hungry and you take some time to think about what you want to eat. This pending state represents the time it takes for you to make a decision. With promises, the pending state is how long it takes to get a response back. The response is usually data from an API, but it could be a number of other values.

Just like real life promises, there are only two outcomes. Either the promise is kept and the data is returned successfully or the promise is broken and an error is returned. Regardless of the result, every promise ends in the settled state that means the promise is finished.

There are a few benefits to working with promises:

- it's easier to read and maintain code
- handling asynchronous functions is better than with callbacks
- there's better error handling available

These are a few of the reasons promises are better than handling async functionality with callbacks. One important thing to remember is that promises still use callbacks. The differences is that with

promises, we attach a callback to it instead of passing the callback in. This is called chaining and we'll get to that a bit later.

Creating a promise

If you're working with APIs, you'll likely get a promise returned as a result at some point. The Fetch API uses promises to handle data requests from other APIs. That's one way to get a new promise. The other way is to create a new promise object instance.

Here's an example of a new promise.

```
let promise = new Promise((resolve, reject) => {
    let message = "just sitting in a promise"
    return message
})
```

This is how you can make a new promise. The arrow function inside of the promise is called the *executor*. This is the function we write to get our data and it automatically gets run when the promise is created. The `resolve` and `reject` functions come directly from JavaScript. They aren't functions that you have to implement.

When the *executor* gets the result, it will call the `resolve` or `reject` method depending on the response. Here's what the meth-

ods look like and how they behave.

```
resolve(value) // if the promise finishes succes

reject(error) // if the promise fails, you'll get
```

Since a promise can only return a value or an error, only the `resolve` or `reject` function will be called. You won't see both of these methods used in the same executor and if you do, only the first method will be acknowledged. Any other `resolve` or `reject` calls will be ignored.

To give you a better idea of this, here's what the flow of this promise looks like.

Figure 7-2. Promise flow from pending to either rejected or resolved

It's a good practice to have some way of handling both the `resolve` and `reject` methods because you don't know what will get re-

turned from the promise. This is especially true if you're working with Node.

In order to handle this, you can use a conditional statement like this one.

```
let hasFetched = false

let finishedPromise = new Promise((resolve, rejec
  if (hasFetched) {
    const fetchingDone = "Here's that stuff I wa
    resolve(fetchingDone)
  } else {
    const notFetched = "Still working on it...""
    reject(notFetched)
  }
})
```

Now that you know how to create a promise, let's look at how we actually get the data we need from them.

Consuming a promise

There are three different methods that us consume the values from the promises we create: then, catch, and finally.

Let's start with an example. You've likely worked with the Fetch API, so a call like this might look familiar.

```
fetch('https://jsonplaceholder.typicode.com/todos
```

If you run this in a console right now and check the value of `promise`, you see that your promise is in a pending state and there is no promise result. That means we don't have the data or an error available yet.

To fix that, we're going to attach a `then` method to our initial promise.

```
fetch('https://jsonplaceholder.typicode.com/todos
  .then(res => res.json())
```

The response for that API call is now the `res` value inside the `then` call. This is how we start working with the results returned from the initial promise. The `then` method makes it so we can attach callbacks to the promise so they will have access to the results we expect when they're available.

NOTE

The `then` method can take two parameters, a function for if the promise is fulfilled and a function for if the promise is rejected. You'll usually see the rejected state handled by the `catch` method which we'll get to in just a bit.

Promise chaining

We can have multiple `then` statements attached to a promise and this is called chaining. Let's take a look at a quick example and then get into the details.

```
fetch('https://jsonplaceholder.typicode.com/todos
  .then(res => res.json())
  .then(json => console.log(json))
```

If you run this code in a console, you'll be able to see the results of that promise. We did this by chaining another `then` statement to the promise. The first `then` give us the data in JSON format. The second `then` uses the value returned from the first `then` to print to the console.

NOTE

Make sure that you notice how the `then` statements are written. In the examples, we're using arrow functions and directly returning the result. Just remember that the arrow functions can still work if you define them like this: `res ⇒ { return res.json() }` All that matters is that something is being returned.

You can do all kinds of things inside these promise chains to get the data you need. When you add a callback using `then`,it will never be executed before the current run of the JavaScript event loop finishes. That happens with regular callbacks and can lead to weird behavior due to race conditions or data not being available when it needs to be.

Now we can introduce the other consumables as parts of the chain.

Handling errors in promises

Similiar to a `try-catch` statement, the `catch` method of a promise is used to handle any errors that arise. The `catch` method actually breaks down to this: `then(null, handleError)`. So it's just a `then` that doesn't receive a function to call for a fulfilled promise and calls the rejected promise function.

Here's an example of what a promise chain might look like with a `catch` statement.

```
fetch('https://jsonplaceholder.typicode.com/todos
  .then(res => res.json())
  .then(json => console.log(json))
  .catch(err => console.log(error))
```

When anything in your promise chain rejects the promise or throws an error, the promise goes to the closest `catch` statement in the chain.

NOTE

You always want to include some kind of error handling for promises. Even if it's just a console log, you need something to keep your app from crashing and to tell you where the issue is.

`catch` statements can also be chained. For example, you might get a very specific error from your API that has details you wouldn't want to print to the console. So you take the error and create a user-friendly message for it.

```
fetch('https://jsonplaceholder.typicode.com/todos
  .then(res => { throw new Error("Forced breakage
  .then(json => console.log(json))
  .catch(err => { throw new Error("Hey user. Some
  .catch(err => console.log(err))
```

If you run this in a console, you'll get the user-friendly message returned from the chained `catch` statement instead of the original error message *"Forced breakage."*

Now that you know how to handle errors in a promise chain, let's finish off the consumables with `finally`.

Executing code regardless of promise state

Just like how a `try-catch` has a `finally` method, so do promises. When you want to run the same code regardless of whether the promise is fulfilled or rejected, you'll reach for `finally`. Here's a quick example.

```
fetch('https://jsonplaceholder.typicode.com/todos
  .then(res => res.json())
  .then(json => console.log(json))

  .catch(err => console.log(error))
  .finally(() => console.log("Doesn't matter what
```

No matter what happens in the promise, the callback in the `finally` statement will always run. Since you know all of the con-

sumable methods, let's take a look at what happens when you're working with multiple promises at the same time.

Working with multiple promises

In more complex systems, you might need to make calls to different APIs at the same time, creating multiple promises. Sometimes you need to sync all of these promises to make sure you have a list to work with when they are all resolved. Here's an example with a few API calls.

```
let todos = fetch('https://jsonplaceholder.typic
let comments = fetch('https://jsonplaceholder.ty
let users = fetch('https://jsonplaceholder.typic

Promise.all([todos, comments, users])
    .then(res => {
        console.log("All of the results: ", res)
        return res.map(result => result.json())
    })
    .then(json => console.log(json))
    .catch(err => console.log(err))
```

We're calling three different endpoints that will return in different amounts of time. By using the `Promise.all` method, we're able to

get all of the resolved promises at the same time and then pass their values to the next part of the chain.

If you run that code snippet in a console, you'll be able to see the resolved results for each of these promises. This isn't restricted to just `fetch` calls. It can be used with any type of promise.

There are a couple of other methods that let you work with multiple promises: `Promise.any` and `Promise.race`.

If you need to take the data from whichever promise responds first and you can ignore the rest, `Promise.any` will do that for you. This might come up if you have the same data coming from different endpoints, but in slightly different formats. Here's a quick example of this.

```
let todos = fetch('https://jsonplaceholder.typic
let comments = fetch('https://jsonplaceholder.typ
let users = fetch('https://jsonplaceholder.typic

Promise.any([todos, comments, users])
    .then(res => {
        console.log("All of the results: ", res)
        return res.json()
    })
    .then(json => console.log(json))
    .catch(err => console.log(err))
```

If you want to execute the callback of the first promise that resolves or rejects one time, you'll use `Promise.race`. This might happen when you have some data that needs to get out quickly and you need to send something as soon as you can. Here's a little example.

```
let commentsA = fetch('https://jsonplaceholder.ty
let commentsB = fetch('https://jsonplaceholder.ty
let commentsC = fetch('https://jsonplaceholder.ty

Promise.race([commentsA, commentsB, commentsC])
    .then(res => {
        console.log("All of the comments: ", res)
        return res.json()
    })
    .then(json => console.log(json))
    .catch(err => console.log(err))
```

NOTE

Remember that when you're working with `Promise.any` or `Promise.race`, the returned values could be different on each execution. It just depends on which promise resovles the fastest.

Common pitfalls

There are a few things you really want to watch out for when working with promises.

While you *can* nest promises, it can lead to some strange behavior if not implemented precisely. There are some use cases for nesting promises if you need more granular details for error recovery. Typically though, you want to keep your promises flat.

The next thing is to forget to terminate your chains with a `catch`. You don't want to send uncaught promises rejections to the browser. Remember that `then` statements only handle a resolved promise.

Recap over promises

- A promise is just an object that returns a value in the future.
- All promises start in a pending state and are either fulfilled or rejected.
- The `then` method is used for fulfilled promises, the `catch` method is used for rejected promises, and the `finally` method executes regardless of the promise's final state.

Async/await

Now that we've covered promises, let's talk about how we can make them even cleaner with `async/await`. The `async` and `await`

keywords give us a more synchronous way to write our async functions. This makes code easier to read and maintain because it gets rid of a lot of `then` statements.

NOTE

It took a while for the `async/await` stuff to stick with me, but as long as you understand promises, you'll get it faster.

The `async` keyword goes before a function. All this means is that the function will always return a promise. Here's an example of an `async` function.

```
async function fetchData() {
    return {
        message: "Yep. It's in a promise."
    }
}
```

This is the same as if we wrote this code.

```
function fetchData() {
    return Promise.resolve({
        message: "Yep. It's in a promise."
    })
}
```

Check these out in the console and you'll see that the return value is a promise! You can call this function and add a `then` to it and get your value.

```
fetchData().then(data => console.log(data))
```

The second part that makes this new syntax so useful is the `await` keyword. The `await` keyword only works inside of `async` functions. This is used to make JavaScript wait until the promise settles and results a result. Let's take a look at how this works.

```
async function fetchData() {
    let res = await fetch('https://jsonplaceholde
    let parsedRes = await res.json()

    return parsedRes.title
}
```

The `await` keyword pauses the execution of the `async` code block until the promise is settled. Instead of writing all of this with `then` statements, we can write the code in a more intuitive way. One thing to note is that you can't use `await` in regular functions. The `await` keyword only works inside of `async` functions.

NOTE

We can do some weird JavaScript magic and make async code run synchronously, even though it's still asynchronous, even though it's all happening in a single thread.

Rewriting promises with async/await

Let's take a look at what it's like to rewrite a promise chain using `async/await`. First, here's the origin promise chain.

```
fetch('https://jsonplaceholder.typicode.com/todos
  .then(res => res.json())
  .then(json => console.log(json))
  .catch(err => console.log(error))
```

Here's how we would rewrite this with `async/await`.

```
async function fetchData() {
    let res = await fetch('https://jsonplaceholde
    let parsedRes = await res.json()

    console.log(parsedRes)

    return parsedRes
}
```

```
fetchData()
    .catch(err => console.log(err))
```

Notice how we handled the error here.

Error handling

Remember that `async` functions are still promises, so we can add a `catch` to the chain. You could also wrap the function in a `try-catch` block like this.

```
async function fetchData() {
    try {
        let res = await fetch('https://jsonplaceh
        let parsedRes = await res.json()

        console.log(parsedRes)

        return parsedRes

    } catch(err) {
        console.log(err)
    }
}

fetchData()
```

The `async/await` keywords still require us to do some kind of error handling. We always want to make sure we have some kind of `catch` statement in place. There's just one more thing to go over with `async/await`.

Using `Promise.all`

Just like there are times you get data from multiple endpoints across multiple promises, the same thing happens with `async/await`. Let's take a look at a code example.

```
async function fetchWrapper(endpoint) {
    let res = await fetch(endpoint)

    if (!res.ok) {
        throw new Error("You'll have to check the
    } else {
        let data = await res.json()

        return data
    }
}

async function fetchData() {
    let todos = fetchWrapper('https://jsonplaceho
    let users = fetchWrapper('https://jsonplaceho
```

```
    let users = fetchWrapper('https://jsonplaceh
    let posts = fetchWrapper('https://jsonplaceho

    let results = await Promise.all([todos, users

    console.log("set 1: ", results[0])
    console.log("set 2: ", results[1])
    console.log("set 3: ", results[2])

    return results
}

fetchData()
    .catch(err => console.log(err))
```

We have two `async` functions working with different APIs. The `fetchWrapper` returns the data we need by handling some `await` operations. Then `fetchData` waits for all of the promises to resolve by using the `await` keyword. We handle the error by chaining the `catch` statement. Now we have a fully functional `async/await` in place.

Iterators

There are a few concepts in JavaScript that don't come up very often, but when they are needed they're super powerful. One of those con-

cepts is iterators.

You've already worked with iterators if you ever iterated over an array. An iterator is an object that lets us iterate over a list or collection. This can be useful when you have complex data structures that you need to get values out of.

Of course you could nest loops, but that can lead to some unexpected behavior and it's harder to maintain. To really understand how this object works, we need to get into some details.

The iteration protocol

Iterators were added to JavaScript to help standardize the process of looping over custom objects. To do this, the iteration protocol was created. That gives us an *iterable* and an *iterator*. Any object can use this protocol by following the right conventions.

Figure 7-3. The iteration protocol

The *iterable* is an object that contains the `Symbol.iterator` symbol. `Symbol.iterator` is how we define the `@@iterator` method needed to handle any iterations for the object. This always returns an object called an *iterator*.

The *iterator* is responsible for letting us know if we still have values to iterate over and for returning the current element. It will have a `next` method that returns an object with the `value` and `done` keys. The `value` key will have the value for the current iteration state and it can have any data type. The `done` value is a boolean that lets us know when we've gone through all of the values in the object.

Any object can be an iterator as long as it implements a `next` method that returns an object with `value` and `done` keys. Let's take a look at a quick example.

```
let iterableObj = {
    [Symbol.iterator]() {
        let interval = 0


        let iterator = {
            next() {
                interval++

                if (interval < 5) {
                    return {
```

```
                        value: `This is step ${i
                        done: false
                    }
                } else {
                    return {
                        value: "The iterator is
                        done: true
                    }
                }
            }
        }

        return iterator
    }
}
```

This is a regular object like you've worked with many times before. The only difference is that the first property we define is the `Symbol.iterator`. You see that this is just a symbol and we define the object for it like we would anything else.

Inside of the *iterable* we just made, we have a state variable available for our *iterator*. The `iterator` object is another regular object that defines a `next` method. This method increments the state variable and then checks to see if it's less than five.

Depending on the result of that conditional check, we'll return one of two objects. Either the object with a `value` telling us which step we're on and that the iterator still has some values left. Or we get the object with a `value` telling us the iterator has gone through all of the possible values and it's finished.

Using an iterator

Now let's look at some ways to actually get values from the iterator. You can run this in a console after the above code example.

```
for(let obj of iterableObj) {
    console.log(obj)
}
```

The `for-of` loop is one of the ways we can loop through all of the values in an iterable. One important thing to note is that when you use this type of loop, it stops returning values as soon as it detects `done` is true. So with our `iterableObj`, we will get the following output.

```
This is step 1 of the iterator.
This is step 2 of the iterator.
This is step 3 of the iterator.
```

```
This is step 4 of the iterator.
undefined
```

You'll notice that we don't get that last message. That's because of what we just mentioned with the `for-of` loop. Once `done` returns true, it stops returning values.

If you want to get the last value from the iterator, then running the `next` method until `done` is true will give you that. If you clear the console and recreate the `iterableObj`, you can run the following code and get the results shown below.

```
var it = iterableObj[Symbol.iterator]()
it.next()
it.next()
it.next()
it.next()
it.next()
it.next()
it.next()
```

```
This is step 1 of the iterator.
This is step 2 of the iterator.
This is step 3 of the iterator.
This is step 4 of the iterator.
The iterator is finished
```

```
The iterator is finished
The iterator is finished
```

Optional return statement

With iterators, we don't have to return any values. The `next` method handles all of the operations and results for the iterator. We *do* have the ability to add an optional return method to handle any situation that ends the iteration early. Let's add a `return` method to this iterator.

```
let iterableObj = {
    [Symbol.iterator]() {
        let interval = 0

        let iterator = {
            next() {
                interval++

                if (interval < 5) {
                    return {
                        value: `This is step ${i
                        done: false
                    }
                } else {
                    return {
                        value: "The iterator is
```

```
                        done: true
                    }
                }
            },
            return() {
                console.log("Yeah... Something de
                return {
                    value: "Who knows what happer
                    done: true
                }
            }
        }

        return iterator
    }
}
```

Now let's say that we're going through the values in the iterator and we're processing the values and an error throws. If you run the following code in a console after with the iterator above, you'll get the console log message defined in the `return` method.

```
for(let obj of iterableObj) {
    if (obj.contains(3)) {
        throw new Error("Ope");
    }
```

```
    console.log(obj)
}
```

You see iterables in action all the time. Any time you've used an array method to loop over values or you've used the spread operator, you've worked with an iterator.

Generators

Most functions follow the run-to-completion model, meaning they can't be stopped before they execute the last line in the code block. If you exit a function with a return statement or by throwing an error, the next time you call it, execution will begin from the top of the code block again. They also only return one value or nothing at all.

A generator is a lot different from the regular functions we work with. Generators are functions that can return multiple values and they can also be exited and re-entered later and still work with the values you left off with.

How generators work

Let's start by looking at the code to define a generator function.

```
function* generateMessages() {
```

```
    yield "This is the first generated message."
    yield "This doesn't get returned immediately.
    yield "You must have called this a third time
}
```

There are quite a few things going on that are different from a regular function. To start with, the way we declare a generator function is slightly different. There's an asterisk right next to the function keyword.

There are a few different ways you might see a generator declaration written.

```
function* generateMessages() { }
function * generateMessages() { }
function *generateMessages() { }
```

All of these are the same thing. The one you choose to use depends on any conventions you decide to go with or your personal preference. With the generator function declared, let's take a look inside that code block.

Unlike with regular functions, we have `yield` statements instead of `return` statements. A `yield` statement pauses the function's execution and sends a value back to where it was called from and it

keeps the state in order for the function to pick up where it left off when it's called again.

Figure 7-4. The generator process

Executing generator functions

Since generators let us pause a function and pick up where we left off later, let's take a look at how executing generator functions work.

```
function* generateMessages() {
    yield "This is the first generated message."
    yield "This doesn't get returned immediately.
    yield "You must have called this a third time
}


let gm = generateMessages()

gm.next() // output: {value: 'This is the first g
```

```
  console.log("This is something happening in betwe

  gm.next() // output: {value: "This doesn't get re

  console.log("Something else happening in between

  gm.next() // output: {value: 'You must have calle

  gm.next() // output: {value: undefined, done: tru
```

We have a generator method defined that returns three different messages. The interesting thing starts outside of the generator though. When we create the `gm` variable, we call the `generateMessages` function. Normally, we'd expect this to return some value.

Since this is a generator function, when we initially call it we don't get a message like we'd expect. Instead we get a generator object. A generator object is the same as an iterator and it has the exact same keys we manually define in our iterable objects.

When we call the `generateMessage` function, the `gm` variable gets set to this generator object. The generator object has a `next` method and it returns an object with the `value` and `done` properties. If this feels a lot like working with iterators, that's because it is!

Similar to how `async/await` helped us clean up promises, generators help us write cleaner iterators along with other things. Like you see in the example above, we can call the `next` method on our generator object until there aren't any more values yielded by the generator function.

NOTE

We use slightly different terminology when describing values we get from generators. A generator yields values instead of returning them. So we get yielded results from generator functions.

Just like with iterators, values aren't returned until you call the `next` method. The big difference here is that instead of us manually creating return statements with object values, the `yield` keyword does all of that for us. The generator handles the `next` method implementation and `yield` handles the results that need to be returned.

Rewriting an iterator as a generator

One of the use cases for generators is making iterators easier to write and maintain. Let's convert an iterator to a generator. Here's the code for the original iterator.

```
let iterableObj = {
    [Symbol.iterator]() {
```

```
    [Symbol.iterator]() {
        let interval = 0

        let iterator = {
            next() {
                interval++

                if (interval === 1) {
                    return {
                        value: `This is step ${in
                        done: false
                    }
                } else if (interval === 2) {
                    return {
                        value: `Step ${interval}
                        done: false
                    }
                } else if (interval === 3) {
                    return {
                        value: `There's somethin
                        done: false
                    }

                } else {
                    return {
                        value: "The iterator is f
                        done: true
                    }
                }
            }
```

```
                }
            }

            return iterator
        }
    }
```

Here's what this same iterator looks like, but written as a generator.

```
function* iterableObj() {
    yield "This is step 1 of the iterator."
    yield "Step 2 is a good one."
    yield "There's something strange about 3..."
}
```

This code is a lot more concise and easier to read. The generator and `yield` handle everything for us so we don't have to manage state as deeply as we do with iterators.

Using return statements in generators

You can use `return` statements in generators and they will make the generator finish by setting the `done` property returned by `next`

to true. Anything that you place in a generator after `return` will be ignored. Here's an example.

```
function* iterableObj() {
    yield "This is step 1 of the iterator."
    yield "Step 2 is a good one."
    return "We're wrapping this up now."
    yield "There's something strange about 3..."
}
```

Once the return statement has happened, any other `next` calls will return `{value: undefined, done: true}` because your generator has finished all of its statements.

NOTE

One subtle difference between iterators and generators is that an iterator can continue to return the last value whereas a generator will always return an undefined value after all of the `next` calls have been made.

Another way that you can work with return statements on generators is to call `return` on it directly. Here's a quick example of that.

```
function* iterableObj() {
    yield "This is step 1 of the iterator."
```

```
    yield "Step 2 is a good one."
    yield "There's something strange about 3..."
}

let iterOb = iterableObj()

iterOb.next() // output: {value: 'This is step 1

iterOb.return("Just going to stop here.") // outp

iterOb.next() // output: {value: undefined, done:
```

Even though we've only called the `next` method once, any other `next` calls we make after the `return` will have an undefined value because the generator is done. You will be able to get the value passed to `return` on that call though.

Using yield*

Those are the ways you can handle return statements in generators and how they work. Now let's go back to the `yield` keyword. There's another form of this that we can use to iterate over other iterables. The `yield*` operator allows us to do that. Let's take a look at an example.

```
function* messageGenerator() {
```

```
    yield "This is step 1 of the iterator."
    yield "Step 2 is a good one."
    yield "There's something strange about 3..."
}

function* userGenerator() {
    yield "John"
    yield* messageGenerator()
    yield "Genji"
    yield "Jerome"
}

for (let value of userGenerator()) {
    console.log(value)
}
```

Here we have two generator functions. One that we have a user generator that we want to add some messages to, so we use `yield*` to do that for us. If you run this code snippet in a console, you'll see the following output.

```
John
This is step 1 of the iterator.
Step 2 is a good one.
There's something strange about 3...
Genji
```

```
Jerome
undefined
```

This `yield*` operator lets us act as if the values from the other generator were created natively in the generator we called. This might be useful if you ever need a private generator for some values.

Passing arguments to generators

There's another interesting thing about `yield`. Unlike with return statements in regular functions, you can pass a value back the generator through the `next` call. This will return a value to the line for the yield statement that was executed. This is known as passing arguments into the generator and it makes generators way more powerful and flexible than iterable objects.

Let's take a look at how this works.

```
function* iterableObj() {
    let first = yield "You tell me what the first
    let second = yield `You said ${first} was the
    yield `Now you're saying ${second} is the val
    yield `Here are all the values: ${first}, ${s
}

let itOb = iterableObj()
```

```
itOb.next() // output: {value: 'You tell me what
itOb.next(8) // output: {value: 'You said 8 was t
itOb.next(9) // output: {value: "Now you're sayi
itOb.next() // output: {value: 'Here are all the
itOb.next() // output: {value: undefined, done: t
```

Walking through the steps in this code, first we declare the generator function. Inside of it, we create a couple of variables called `first` and `second`. These hold the values returned for the yield statements. We use the values passed into the generator to create dynamic messages we yield when the generator is called again.

The first time we call `next`, we don't need to pass a value. The second time we call it, we'll pass a value and this gets assigned to the `first` variable. Then we'll call `next` again and pass the `second` value. Since the generator perserves the state of the function in between pauses, we can keep referencing these variables in later yield statements.

This can be very useful if you have to handle complex data transformations efficiently.

Generator advantages

You can run a generator to get some data you need, do some processing, render it in the browser, and then go do a bunch of other stuff until you need some new data. Then the generator will pick up exactly where it left off. One great use case for this is implementing an id generator.

As an example, you might have to account for product names or numbers for orders created and they're dependent on the previous value and you only do this every now and then. A generator will give you a new id each time you call `next` and it'll sit and wait for you to need another id.

When you have a generator without a return statement and you never set its `done` value to false, you can create an infinite stream. This is a generator that you can call forever. This could be used to stream data when you need it.

It also gives you lazy evaluation. That means the evaluation of an expression is delayed until the value is needed. This makes generator functions memory efficient. We only generate the values that are needed so we're not taking up extra space in the heap.

NOTE

Iterators and generators are concepts you'll rarely have to use. Although if you do have to use them, it's invaluable to already have a strong understanding of what they are and how they work.

Observables

This is the last of the advanced topics we'll cover on JavaScript, but it's an important one. If you've ever worked with Angular, you've probably heard of observables, but they aren't specific to this framework. Observables are actually made of two different parts.

Observables are just functions that throw values and **observers** subscribe to these values. This follows the observable design pattern and creates a pub-sub system. There are a lot of words that could describe this concept, but it might be easier to understand if we build an observable class.

Building an observable class from scratch

Let's walk through how this works.

```
class Observable {
    constructor(observerFunction) {
        this._observerFunction = observerFunctio
```

```
    }

    subscribe(observer) {
        return this._observerFunction(observer)
    }
}

let guestObservable = new Observable(observer =>
    setTimeout(() => {
        observer.next("This observer actually wo
        observer.complete()
    }, 1000)
})

let guestObserver = {
    next(data) {
        console.log(data)
    },
    error(err) {
        console.log(err)
    },
    complete() {
        console.log("The request is finished!")
    }

}

guestObservable.subscribe(guestObserver)
/*
```

```
  This observer actually works?
  The request is finished!
  */
```

The `Observable` class is where we start. There's a constructor that takes in an `observerFunction` function as the argument. Then we have a `subscribe` function that takes an `observer` that gets passed to the function we made in the constructor. This is really all an observable is at its heart.

Then we create a new instance of this `Observable` called `guestObservable`. It has the `observerFunction` defined and you can see how the `observer` values are called. We're using a `setTimeout` to simulate us waiting on a response from an observable.

Next, we'll create an observer object called `guestObserver`. This object will have a few functions that define the notification types that handle how the observable processes. An observer is an object that has a `next` method, a `complete` method, and an `error` method that define the notification types the observable can send.

Lastly, we execute the `guestObservable` by calling the `subscribe` method. This passes our `guestObserver` into the `subscribe` method we defined in the `guestObservable`. That's

why we're able to call the `next` and `complete` methods inside of the observable `subscribe` function.

Luckily for us, we don't have to implement observables from scratch in practice. It's pretty common to use the RxJS library to handle all of the details behind observables.

NOTE

I highly suggest you take a look at the RxJS documentation as we will reference some of the functionality in the rest of this section.

What problem do observables solve

Observables came about from the quest to handle asynchronous activity in JavaScript. If you have more experience with promises, you might wonder why you should bother learning about observables. There are some key concepts about observables that are important for you to understand.

NOTE

You may hear people draw comparisons between observables and promises or observables and event emitters. Observers are not like either of these. They might behave like promises or event emitters depending on how you implement them, but they don't actually share any commonalities. They're just super flexible.

Observables can return multiple values over time

Most normal functions and promises only return a single response one time. This is great for things like API calls because you don't have to worry about there being multiple responses to the same request. When a function returns a value, it's finished. When a promise resolves, it's finished.

When an observer returns a value though, it can return more values afterwards. Let's take a look at a couple of examples.

```
function pageDog() {
    console.log("Looking for the dog.")

    return "Still looking."
    return "Don't know where they went."
}
```

The second return statement in the function above will never be executed. Now let's take a look at how this works in an observable.

```
import { Observable } from "rxjs"

const findDog = new Observable(subscriber => {
  console.log("Not sure where he went.")

  subscriber.next("Still looking.")
```

```
    subscriber.next("Still looking.")
    subscriber.next("Maybe upstairs?")
    subscriber.next("I'll call him for a treat.")
  });

  console.log("My dog has been too quiet.")


  findDog.subscribe(msg => {
    console.log(msg)
  })

  console.log("Of course he came for a treat.")
```

With an observer, we can return multiple values. This is called emitting values from an observable.

Observables can be asynchronous sometimes instead of always

The code executes synchronously, but you could also include async emit values too.

```
import { Observable } from "rxjs"

const findDog = new Observable(subscriber => {
  console.log("Not sure where he went.")
```

```
    subscriber.next("Still looking.")
    setTimeout(() =. {
      subscriber.next("Maybe upstairs?")
    }, 2500)
    subscriber.next("I'll call him for a treat.")
  });

  console.log("My dog has been too quiet.")

  findDog.subscribe(msg => {
    console.log(msg)
  })

  console.log("Of course he came for a treat.")
```

This is true for any observers that are subscribed to this observable. They can use async or sync emitted values. Which brings us to the next cool thing about observables.

Observables can have multiple subscribers

Sometimes you need to share updates in real-time. Promises are limited to one call at a time. That means if you call a promise with one value and then you call that promise with a different value, you'll only get one of them instead of both. Since timing is shared between both promises, they collide which leads to one value being returned.

When we use observables, there can be as many observers as we need. This is because similar to promises, observables work on a push system. That means they push values out to functions that aren't sure when they will receive that value.

Each observer gets its own connection to the observable. That lets us execute the same observable with different functions at the same time and they all get the updated values because they are pushed or emitted to all of the observers.

So we say that observables are unicast. Each observer that subscribes to an observable gets a separate instance of the observable function. Here's an example of what that would look like.

```
import { Observable } from "rxjs"

const handleMessages = new Observable(subscriber
  console.log("Are you in the chat?")

  subscriber.next("No, just your DMs.")
})

handleMessages.subscribe(user1 => {
  console.log(user1)
})

handleMessages.subscribe(user2 => {
```

```
  console.log(user2)
})
```

Both of these observers will get the same output at the same time. Subscribing to an observable is like calling a function that accepts a function as the argument.

Observables are cancellable

There might come a point in your code where you want to make an observer unsubscribe from an observable. This is something that's built into the RxJS library. It works by returning a `Subscription` object when we call `subscribe` on an observable. So when we subscribe, we also have an `unsubscribe` method available. Here's a quick example.

```
import { Observable } from "rxjs"

const handleMessages = new Observable(subscriber
  console.log("Are you in the chat?")

  subscriber.next("No, just your DMs.")
})

let userSub = handleMessages.subscribe(user => {
```

```
    console.log(user1)
  })

  console.log("The user is subscribed right now.")

  userSub.unsubscribe()

  console.log("And now they aren't.")
```

This ends the subscription of that observer and you can free up some resources.

Observables are lazy

An observable only emits values when it has observers. Otherwise it just sits there. The function we define in the constructor doesn't get called until something calls the `subscribe` method. This is different behavior than a promise. It gets called as soon as it's created.

Think about an observable as a singer waiting to go on stage. They only start singing when enough people are there to listen. If there isn't anyone there yet, they will stay backstage and continue warming up.

When would you use observables

If you've ever used a live chat, you've already seen how observables work and this is one of the use cases for them. Anything that has to do with results being sent through web sockets or DOM events can be handled nicely with observables.

You could use observables to update a web page with real-time financial data or weather data. You can use them to broadcast a countdown to users across different timezones. Almost any where you want to work with real-time data is also a good time to use observables.

Chapter 8. Shifting to TypeScript

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

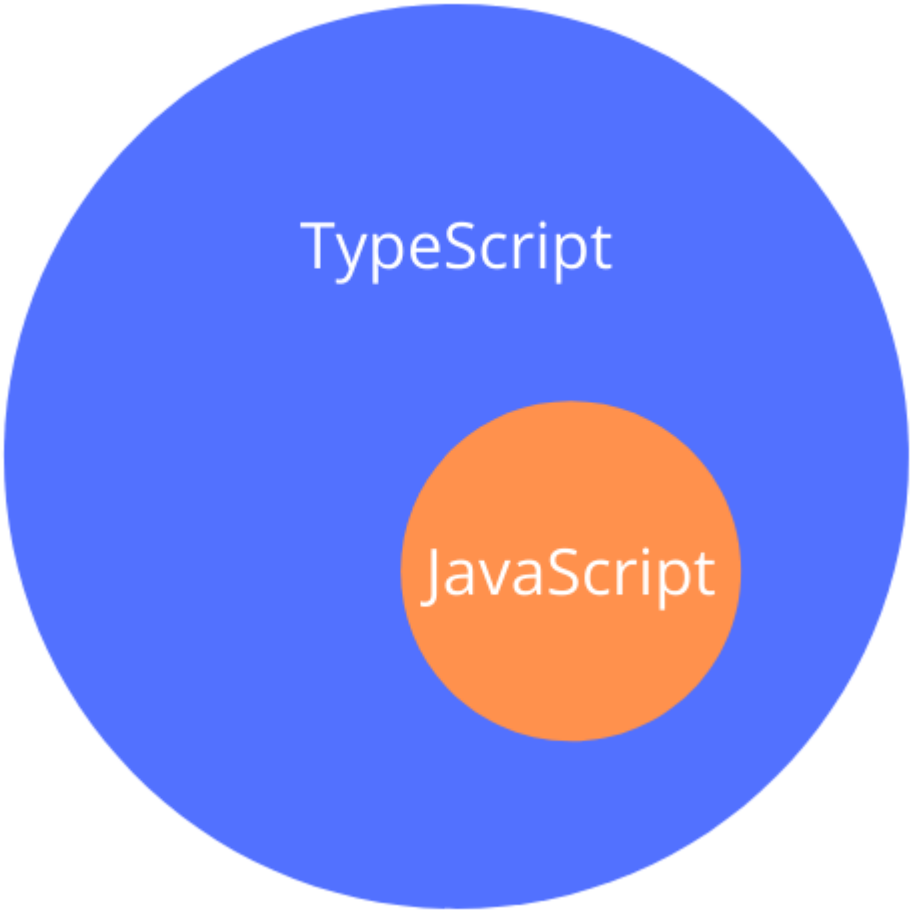
This will be the 13th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at milecia.mcgregor@gmail.com.

One big complaint in the JavaScript community is that it is a weakly-typed language. In JavaScript, your variables can be anything at anytime. This leads to issues with values dynamically changing while an app is running which can lead to unexpected errors, causing your applications to crash for users.

What is TypeScript?

That's where TypeScript comes in. TypeScript is a language developed by Microsoft that's an extension of JavaScript that adds the ability to use types. It can be used on the front-end and the back-end, regardless of any frameworks you choose to work with.



Types let you define the data a variable holds. If you're using JavaScript, you likely define variables using one of the following keywords:

```
let price = 14.99
const name = 'Battery charger'
var inStock = true
```

All of these variables have different types and you can figure that out from the values that are hard-coded. What happens when you have dynamic values coming from some other source?

```
let quantity = userQuantity
const customerName = name
var isShippingFree = orderTotal > 50
```

With these new values coming from other places, we have no idea what the data type will actually be. Unless we wrap everything in conditional statements, the incoming data types could be anything. This is the problem TypeScript solves.

It will help you catch many errors before you even run the code because a large number of JavaScript errors are due to type mismatches. This isn't a a glaring problem in smaller apps, but as you scale and you're working with thousands of lines of code it's important to have this kind of type-checking available.

Throughout your entire development process, TypeScript can help you keep track of values being returned from APIs and what parame-

ters you should pass into functions. It's a static type checker, so it runs your code before it gets to the runtime engine. This is when it makes sure that all of the types throughout your codebase are good to go.

Now that you know TypeScript is basically strong-typed JavaScript, let's review some of the specific differences betweent the two languages.

Differences between TypeScript and JavaScript

TypeScript

- released by Netscape in 1995
- static-typed language
- built for large, complex projects
- supports modules for projects
- functions can have optional parameters
- finds errors before runtime
- can have slower performance
- slightly longer learning curve
- supports object-oriented features like classes and inheritance
- supports ES3, ES4, ES5, and ES6 features

JavaScript

- released by Microsoft in 2012
- dynamic-typed language
- subset of TypeScript
- still valid in TypeScript
- great for small projects
- allows more flexibility in projects
- no compilation step
- large, active community of developers
- extra packages for types are unnecessary
- developers can work with it directly in HTML

Why Bother Using It

As you can see, there are a lot of pros and cons to using either language. It boils down to the type of project you're working on. When you have a complex app that handles large amounts of data across several files, this is a perfect use case for TypeScript.

Projects like user dashboards, finance applications, and medical systems are just a few scenarios that would benefit from the static-typing TypeScript offers. This saves you a lot of panic later on when your

code is on production and you start getting type errors because users find new ways to break things.

Primary benefits

The main benefit is that you have static types with TypeScript. That gives you and your team the ability to catch bugs long before they hit production or any other environment. Your app won't be able to run as long as there are type errors.

This gives you a chance to account for a number of issues that could come up over time by making you include types for your parameters, variables, and incoming data. You won't have to worry about your app crashing from small errors like this anymore.

By adding types, your code becomes more maintainable because it's readable. Any new developers to a project will be able to understand what they should be sending to different functions and what they can expect back. This will save time ramping up on projects and it will also help developers make more informed decisions.

If you're writing code in VSCode, you also get help from Intellisense. When you write your functions with TypeScript, you'll be able to see the expected parameter types and the return type. So you can cut down on the number of times you have to switch between files to remember what your functions need and return.

Modules are another powerful benefit of TypeScript. For larger apps, you probably have code that you reuse throughout the project. Being able to import and export code as modules makes it more manageable to maintain functions and constants used across different files. The JavaScript alternative is to rewrite the same functionality in each file you need it.

This leads to improved testing. With modules, you can test that your functions work in isolation from the complete app. Another advatage of TypeScript is this modularity for testing purposes. When you can test specific functionality in your app in isolation, you can be more sure of where the real issues are.

One of the features that leads to teams adopting TypeScript over time is that you can gradually update your codebase. You don't have to immediately convert all of your code to a new language. Since TypeScript is a superset of JavaScript, it still supports dynamically-typed code. So your current code is still supported as you upgrade. This has led to many teams converting projects to TypeScript as they grow.

This is especially apparent on server-side apps. When you're handling a lot of data requests and responses from and to different sources, having static types helps keep the code making sense. You

can handle some complex data structures without worrying about your types changing right before an important database update.

TypeScript is also flexible enough to work on the front-end. You could have you're entire stack be one programming language that both the front-end and back-end engineers will speak the same language and expect the same things so this can make cross-team communication smoother.

When not to use TypeScript

As great as TypeScript is, there are some cases where you may want to consider using regular JavaScript. If you're working on a small prototype, TypeScript might be overkill to start with. Another scenario would be if you're working with code directly in the HTML for performance reasons.

Since TypeScript has to be trans-compiled, we aren't able to write it directly inside HTML `<script>` tags. If you have a simple web app, like an income calculator, you can likely get by with just JavaScript.

The main use cases for not using TypeScript involve small projects that don't need to support any complexity. As soon as you start to scale a project though, the benefits of static-typing start to outweigh the upfront investment of setting up a TypeScript project.

We've covered all of the background on TypeScript so let's take a look at a few code examples.

TypeScript Code Examples

We'll take a look at a few snippets that show some of the common uses for TypeScript in production projects.

Function definitions

Probably the most common use will be for function and variable definitions. Let's set up the scenario. We have a function that will take in a few parameters from a user and then return a value that determines which message the user is shown.

```
interface UserAuthenticationInput {
  userId: number
  password: string

  mfaCode: number
}

interface UserAuthenticationResponse {
  confirmationCode: number
}

async function isAuthenticated(input: UserAuthent
```

```
async function isAuthenticated(input: UserAuthen
  const url = process.env.AUTH_API

  let res = await fetch(url, {
    method: 'POST',
    body: JSON.stringify({
      userId: input.userId,
      password: input.password
    })
  })

  return res.json()
}

let isUserLoggedIn = isAuthenticated({ userId, pa
  console.log(data) // JSON data parsed by `res.j
})
```

There are a couple of interfaces that define the input the function expects and the types associated with each value and the expected response value. These could be more complex objects if that would fit your project better.

Data structures

Let's take a look at a more complex data interface. This comes in handy when you're working with APIs that return a lot of values. You

not only know what you have access to before the API returns the response, you'll also know when an API has been changed.

If the types in your code no longer match what the API returns, you've saved yourself a substantial amount of debugging time because you will get an error quickly instead of the app crashing in some loosely connected place.

Here's that example.

```
interface Address {
  country: string
  street: string
  zipCode: number
}

interface UserProfile {
  name: string
  imageUrl: string
  age: number
  address: Address
}

interface AccountSummary {
  balance: number
  hasOverdraftProtection: boolean
  dependents: number
}
```

```
}

interface AccountReport extends AccountSummary {
  expenses: number
  income: number
  accountOwners: number
  accountId: string
  accountName: string
}

interface UserAccountResponse {
  profile: UserProfile
  summary: AccountSummary

  report: AccountReport
}
```

Here's where some of the true benefits of TypeScript shines. When you have data like this coming from some API, you want to know exactly what you should expect. Having nested values is something that happens all the time. You see that TypeScript lets you use your interfaces to define the types for other values.

You can also use some object-oriented programming and do things like extend interfaces or use inheritance. This is something that JavaScript doesn't let you do.

Modules for helper types

If you've ever use an `npm` package, you're may have used some type modules. In larger projects, you may need to create type modules to use helper functions throughout your code. Here's an example of some helper types that seems to come up in a lot of projects.

```
// UserProfile.ts
export const API_ENDPOINT: string = "http://local

export interface UserProfile {
    name: string
    userId: number
    isLoggedIn: boolean
}
```

This module lets us export a value and a type interface. Now if we want to import these into a different file and use them, that would look something like this:

```
import { API_ENDPOINT, UserProfile } from './User

fetch(API_ENDPOINT)
  .then((res): Promise<UserProfile> => res.json()
  .then((data): UserProfile => console.log(data)
```

You can import all of the values from a module or just the ones you need. These are just some of the ways TypeScript can make your code more maintainable and less error prone.

That's all for TypeScript. If you're already doing JavaScript development, it's something you should strongly consider learning. TypeScript developer jobs tend to pay more than the JavaScript equivalent. Now we can move on to some of the things that no one really teaches you as you gain experience at work.

About the Author

Milecia McGregor is a senior software engineer that's worked with JavaScript, Angular, React, Node, PHP, Python, .NET, SQL, AWS, Heroku, Azure, and many other tools to build web apps. She also has a master's degree in mechanical and aerospace engineering and has published research in machine learning and robotics. She started Flipped Coding in 2017 to help people learn web development with real-world projects and she publishes articles covering all aspects of software on several publications, including freeCodeCamp. In her free time, she spends time with her husband and dogs while learning to play the harmonica and trying to create her own mad scientist lab.