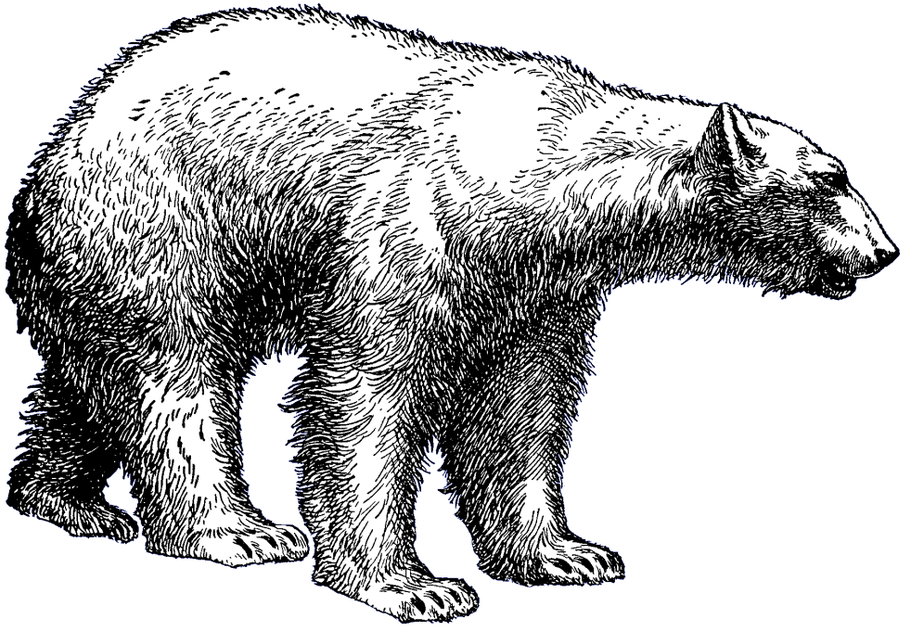
Build your own full stack SaaS product in days!



Full Stack SaaS Product Cookbook

From Soup to Nuts

Full Stack Craft LLC™

Christopher J. Frewin

Full Stack SaaS Product Cookbook

From Soup  to Nuts  - Create a Profitable SaaS Product as a Solo Developer

Christopher J. Frewin

https://chrisfrew.in

May 25, 2021

Contents

List of Figures

List of Listings

Foreword

> If I have seen further it is by standing on the shoulders of Giants.
>
> *(Isaac Newtown, 1675)*

What's a SaaS?

SaaS Products. Such a massively overused buzzword in today's internet culture.

Everyone seems to *want* a profitable SaaS product, but rarely is a complete in-depth discussion taken on what exactly that entails. Typically, the technical bare minimum for a 2020s SaaS product includes the following:

- User authentication, authorization, and management
- A custom backend API
- A nice looking and easy-to-use UI
- Email flow and service for welcoming new customers, password resets, etc.
- Logging and alerts throughout the entire stack
- Last and most importantly, *what value the product itself provides*.

These design minutia and decisions don't fit into our 280

character Tweet world. A huge majority of the resulting noise online surround SaaS development therefore devolves into the incessant framework vs. framework or language vs. language battles - or worse - paraphrased guru or meme-like slogans that have nothing to do with actually putting in the hard work to build the product itself.

In this book, I cut through all that noise, describing in extreme detail, step-by-step, from frontend to backend, with all configuration in between, how to build all parts of your next profitable SaaS product. The final product will be highly maintainable while at the same time highly customizable. After 10+ years of building my own solo side products, wasting literally *thousands* of hours making countless of mistakes, I've finally arrived at an extensively reusable, fast, and very lean stack that works for solo developers. This book is the refined culmination and best practices of my decade long experience.

Who this Book is For

This book is targeted at solo developers, creators, and makers who want to have full control over their own SaaS Products and know the inner workings at all parts of the stack. It's for those who want to ultimately automate nearly all aspects their product or service with small exceptions like communicating with customers, or personal interactions promoting the product (all of which are *extremely* important, as I'll get to in later sections of the book.)

If you are a solo software developer looking to move into the SaaS landscape and not waste time asking yourself and answering complex questions like:

- What database to use
- What authentication or authorization service to use

- What type of API to use
- How to implement full stack logging, monitoring, and alerts through the entire application
- How to create and automate frontend and backend builds with CI and CD

Then look no further. This book will provide answers to all those questions and more with full code solutions. Note that this book *is* highly opinionated. I do use specific frameworks and services throughout the entirety of the book.

Like I've said, after searching for 10 years for the holy grail of SaaS product generators, I believe I've found it, at least for web-based SaaS products. If you are looking for more theoretical or fundamental-minded books on building apps, this book is not for you, and there are plenty of those out there.

Book GitHub Organization and Repository

I have created an **entire GitHub Organization for this book**. It includes all milestone repositories as well as **the repository for this book's source**! (Too meta, right?). While I encourage you to understand and write your own code as you follow along, I also totally understand if you've missed a section or something small and clone the code just to see how it works. Enjoy!

Highlight Boxes

Throughout this book, you'll encounter a variety of highlighted boxes, which are colored coded to provide specific types of information. Examples are as follows:

✅ Green Highlight Boxes ✅

Green highlight boxes have green check emojis and will offer links to various repositories which act as milestones

of the codebase we will be building together.

Blue Highlight Boxes

Blue highlight boxes have blue information emojis and are more of aside details about my opinions on languages, methodologies, and tools. They aren't essential to the workflow of building the product, but offer some nice insights (in my opinion) into the careful thought process I put into my stack.

Yellow Highlight Boxes

Yellow highlight boxes have yellow warning emojis are warnings of what could go wrong with a particular piece of code, the stack, or a methodology. Take note of these far and few between warning highlights!

Use the Index, Listings, Recipes, and Figures to Your Advantage

By the power of LaTeX, a variety of helpful references have been built into this book:

The Index includes all references to all packages, files, and keywords used throughout this book. Typically files are referenced in cronological order, so you can observe changes made to specific files throughout the production of ReduxPlate.

The list of listings also includes every code snippet in the entire book with a detailed description. Use it to jump to whatever snippet you'd like to look at.

Likewise, the list of Recipes is a custom listing of reusable style code that shouldn't need to be refactored away from ReduxPlate - these recipes are generic snippets or files that

can be reused in any SaaS product.

Finally, the list of figures

Are You Ready?

I'm proud of how this book came out, and I frequently reap the rewards of my own labor, using it as a handbook myself for each new SaaS product I build. I hope that I've piqued your interest, and that you'll join me on this full stack adventure!

- Christopher Frewin

Feldkirch, Austria, April 2021

1

The Product

> It's really rare for people to have a successful start-up in this industry without a breakthrough product. I'll take it a step further. It has to be a radical product. It has to be something where, when people look at it, at first they say, 'I don't get it, I don't understand it. I think it's too weird, I think it's too unusual.
>
> *(Marc Andreessen)*

1.1 The Product We'll Be Building

The product we'll be making in this book is a product I call 'ReduxPlate'. It's a real, full fledged, profit generating product I own, currently live at **https://reduxplate.com**. It's a $60 / year subscription service that builds the entire Redux code boilerplate from the state of an application alone, in addition to many other time-saving features! In short, it's a one stop shop for Redux code management, generation, and maintenance.

For those who use with , you may know how much code needs to be written after adding just one new part of state. (Read: it's even more than the boilerplate required with vanilla JavaScript!) I had long wanted to build a SaaS product like this, and the motivation to write the book finally

spurred me to build it, since it is a good example for a full stack SaaS product.

Don't worry, we'll get into the nitty-gritty of how it actually works, writing all the code step-by-step throughout this book. But more on those details will come later.

My Challenge to You

If you're motivated, I suggest to copying only the *nature* of each of the tutorials throughout the book, modifying code where it is needed, ultimately coming out with your own SaaS product by the end of the book. This is especially useful in the "recipe" sections in the second half of the book - they are actually product agnostic, and should be able to be included for *any* type of SaaS Product.

It's also completely acceptable to work through the tutorials exactly step-by-step - you'll come out with an exact clone of what ReduxPlate looks like today! Even if you take this mimicry style of workflow, at the end, you'll still have this book as a reference and can do it all again, already knowing all the steps, for your next profitable SaaS product!

2

The Frontend - Getting Started

> You've got to start with the customer experience and work backwards to the technology.
>
> *(Steve Jobs, 1997)*

2.1 Introduction to the Frontend

Chapter Objectives

- Some notes on naming conventions you'll see throughout the book

- A few of my own personal style techniques when writing frontend code with React and TypeScript
- Define the framework and tool versions used on the frontend

We're going to start off building the frontend, as that side of the stack gives us some immediate visual feedback, and as Steve's quote above touts, we can then work backwards to figure out what sort of technologies we'll need to complete our SaaS product.

A Word On Naming Conventions

As mentioned in Section I, I'll be going step by step through what I did to build ReduxPlate (**https://reduxplate.com**) Indeed, this book was written *while* I built ReduxPlate! The repositories we'll create for the project will key into the naming convention I will use throughout the book. In fact, the only two repositories we'll need for our entire complete SaaS product will have the following names:

`reduxplate.com` (For the frontend repository, AKA the client. In the case of a web app, which ReduxPlate is, I typically choose the root domain name for the the name of the repository.)

`ReduxPlateApi` (For the backend repository, AKA the API. This is standard capitalized camel case notation that is standard in C#, and will make our namespaces play nice in our .NET code.)

So, we will see this `reduxplate` or `ReduxPlate` moniker

over and over again throughout this book. In the case of things like secrets and constants, we will see this moniker used instead in all caps and with an underscore as a space, i.e. `REDUX\_PLATE`. In some cases for readability, I will use it lower case with a hyphen, .i.e. `redux-plate`.

If you are going the option of tailoring each step in this book to your own project, whenever you see `reduxplate` or `ReduxPlate`, take it as a signal to rename variables with those monikers to your own product's name. Take a deep breath, there's going to be **a lot** of them.

Some Notes on My Frontend Style

I also have developed my own specific code style. Some of my most important rules, though not all of them, include:

- Avoid `var` and `let` wherever possible; this should almost always be possible.
- Always de-structure `props`
- Keep as much logic out of components as possible - components should generally be only for rendering jsx-style markup
- Use TypeScript
- Use **Redux** with **Redux Toolkit**

Frontend Frameworks and Tools Versioning

On the frontend, I will be using these versions of the following tools and frameworks:

- npm 7.6.13
- Node 14.16.0

- Gatsby 3.0.0
- React (and React DOM) 17.0.2
- Bootstrap 5.0
- TypeScript 4.2

Installation and setup of all these frameworks, including code editor plugins and so on are outside of the scope of the book (excluding Ubuntu 20.04 - I will be going over in detail how to start a Ubuntu 20.04 box with Digital Ocean). There are plenty of awesome resources online for everything else, and for the packages themselves, **it's always best to start with their respective documentation first.**

Everything still okay? Let's finally start building this product!

2.2 Bootstrap the Frontend With Gatsby V3

Chapter Objectives

- Bootstrap the frontend with Gatsby's official starter, `gatsby-starter-default`

With some housekeeping done, let's jump right into code. We'll start by cloning one of the official Gatsby starters, in fact, the default one, `gatsby-starter-default`, and I'll name my project `reduxplate.com`. This will also be the folder that Gatsby creates for us.

So, you'll also need to install Gatsby if you don't have it installed yet:

```
terminal
npm install gatsby
```

Listing 1: Installing Gatsby via npm.

Then, the command to create our frontend Gatsby project is:

Listing 2.1: terminal

```
gatsby new reduxplate.com
https://github.com/gatsbyjs/gatsby-starter-default
```

We'll cd into the directory:

Listing 2.2: terminal

```
cd reduxplate.com
```

and get started with the **develop** command:

Listing 2.3: terminal

```
npm run develop
```

You should see the Gatsby starter spool up at **localhost:8000** in your browser, or a different port if you already had something running at 8000:

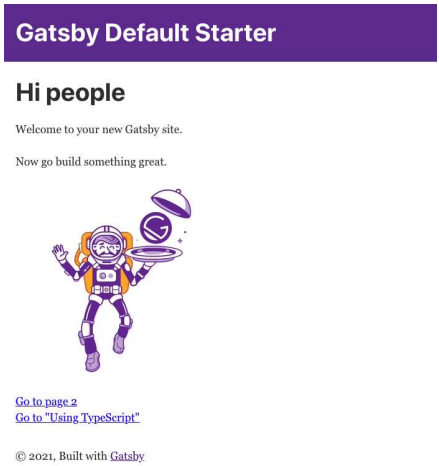


Figure 2.1.: Screenshot of the unmodified default Gatsby starter.

2.3 Clean Up the Gatsby Default Starter

Let's now do some simple house cleaning on this project Gatsby has just scaffolded for us. While doing all of these steps, you should be able to keep running the site in development mode, and see the warnings provided in the terminal. The step by step process to get down to a no-fluff skeleton is as follows:

- hop into `package.json` and modify all the values to fit your project. This likely includes the `name`, `description`, `author`, and `keywords` fields.
- Then take a look in `gatsby-config.js`, and follow a similar pattern, modifying the `title`, `description`, and `author` fields. You can also scroll down and active the `gatsby-plugin-offline` plugin and delete the comments about it. Also be sure to update the values under the `gatsby-plugin-manifest`: update both the

name and **short_name** fields.

- In the **src/** folder, within **pages/**, delete the **page-2.js** and **using-typescript.tsx** files. You can then delete the two **Link** components to each of those pages from **index.js**, as well as the **Link** import there.
- Delete the comments in **gatsby-browser.js**, **gatsby-node.js**, and **gatsby-ssr.js**.
- In the **components** folder, delete **layout.css** (and where it is imported in **layout.js**).
- Delete the **gatsby-astronaut.png** image in the **images/** folder and delete the **<StaticImage>** component from **index.js**.
- Delete the comment fluff on the top of each of the remaining components **header.js**, **layout.js**, and **seo.js**
- Convert all the remaining component files to **.tsx** files, since they are all React components. Also capitalize all the files in the **components/** folder, i.e. **Header**, **Layout**, and **Seo** - we do this as the standard TypeScript pattern for files to match their export names. We won't capitalize the names of the files within the **pages/** folder, since these file names will reflect the actual URL of the page that is produced.
- Remove all references to **propTypes** and **defaultProps** in the codebase.
- After doing that, you'll need to clean up what is now the **Seo.tsx** file. We'll create an **ISeoProps** to use as our props instead. The full resulting component that makes TypeScript happy looks like this:

Listing 2.4: </> Seo.tsx

```
import * as React from "react"
import { Helmet } from "react-helmet"
import { useStaticQuery, graphql } from "gatsby"
import { siteMetadata } from "../../gatsby-config"

export interface ISeoProps {
  title: string
  description?: string
}

function Seo(props: ISeoProps) {
  const { description, title } = props
  const { site } = useStaticQuery(
    graphql`
      query {
        site {
          siteMetadata {
            title
            description
            author
          }
        }
      }
    `
  )
```

it's not as detailed as an SEO component could be, but we'll be revisiting and boosting the **Seo** component later in the book.

Also update the **README.md**. I typically set the title of the README as the name of the repository itself, and then add a small description, something like this:

Listing 2.5: </> README.md

```
# reduxplate.com

The website source for ReduxPlate - never write a
line of Redux again.
```

Since this repository is private, we won't be adding any more information to the README. If you are open-sourcing your project it's wise to include things like install steps, environment variables, and any other ex-

```
        siteMetadata.description || ""}
      />
      <meta property="og:title" content={title} />
      <meta
        property="og:description"
        content={description ||
        siteMetadata.description || ""}
      />

      {/* Twitter Card tags */}
      <meta name="twitter:card"
      content="summary_large_image" />
      <meta name="twitter:creator"
```

amples or requirements to get the product running.

- Finally, update the LICENSE file. You can keep the BSD license, but be sure to change the company name to your company or your own name. I prefer the MIT license. When formatted for my own company, Full Stack Craft LLC, the MIT license looks like this:

Listing 2.6: </> LICENSE

```
MIT License

Copyright (c) Full Stack Craft LLC and its
affiliates.

Permission is hereby granted, free of charge, to
any person obtaining a copy
of this software and associated documentation
files (the "Software"), to deal
in the Software without restriction, including
without limitation the rights
to use, copy, modify, merge, publish, distribute,
sublicense, and/or sell
copies of the Software, and to permit persons to
whom the Software is
furnished to do so, subject to the following
conditions:

The above copyright notice and this permission
notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT
WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE
WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT
OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE
USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

Also remember to update the **license** key in

package.json appropriately if you choose also to switch to a different license, here following my MIT license example:

Listing 2.7: </> package.json

```
"license": "MIT",
```

So far so good. Right now, the folder structure of the skeleton of the Gatsby default starter should look like this:

Listing 2.8: </> terminal

```
.
├── LICENSE
├── README.md
├── gatsby-browser.js
├── gatsby-config.js
├── gatsby-node.js
├── gatsby-ssr.js
├── package-lock.json
├── package.json
└── src
    ├── components
    |   ├── header.tsx
    |   ├── layout.tsx
    |   └── seo.tsx
    ├── images
    └── pages
        ├── 404.tsx
        └── index.tsx
```

We've got only two pages, the home page (`index.tsx`) and a 404 (`404.tsx`) page, and a handful of components: a layout, a header, and an seo utility component.

A Word On Typescript

For a Full Stack SaaS Product, I would argue that using TypeScript is nearly a necessity. It speeds up development, maintainability, and will help you catch any type errors before you even run your code. We will be using it all across the frontend, including our serverless functions, as we'll see later.

For the Gatsby project, every file we write within the `src` directory will have either a `.tsx` extension, when JSX syntax is needed for React, or a `.ts` extension, for any other non-React code. Luckily, Gatsby supports TypeScript out of the box, so all we need to do is convert the existing files to their respective `.ts` and `.tsx` extensions, and we are all set.

Recap of the Frontend Bootstrapping

We're nearly reading to start actually coding and building our frontend. We've bootstrapped our project with the Gatsby CLI. We've edited our `package.json` and `gatsby-config.js` to reflect our project, converted all components to `.tsx` files, and removed all fluff from all files and code. We also made a few changes to get the codebase to jive nicely with TypeScript. All that is left to get started is to creating a proper git repository so we can start pushing our changes!

✅ Milestone Code #1 ✅

We've reached the first milestone repository of this book: **the skeleton Gatsby repository which we've just finished crafting**! There's not much in it, but it is a perfect minimalist and TypeScript-minded Gatsby boilerplate to start your future SaaS products with.

2.4 Setup a Bitbucket Repository for the Frontend

Chapter Objectives

- Creating a BitBucket repository for the SaaS app's frontend.

Since this will be a private SaaS product, I will be creating a Bitbucket repository for it. Feel free to start yours in a private (or even public!) repository on GitHub. Just keep in mind that further on in this book you will have to take care of things like API secrets and keys in an environment like GitHub by yourself. This is still possible and the workflow is very similar to Bitbucket.

Create the Repository

Create an account on BitBucket if you don't have one already. Then, from your overview dashboard, click the '+' icon in the top left of the screen:

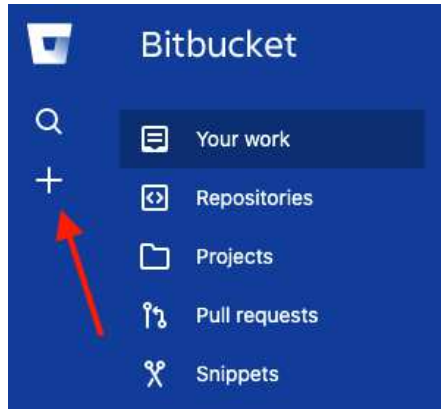


Figure 2.2.: Screenshot of adding a repository on Bitbucket.

then select ‘Create’ > ‘Repository’:

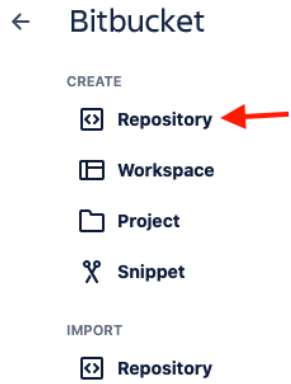


Figure 2.3.: Screenshot of the ’create repository’ on Bitbucket.

On the resulting page, apply the following:

- Workspace: can just be your workspace, or your team’s if you have one.

- Project: I created a new project called ‘ReduxPlate’, you can choose whatever project you’d like here
- Repository name: should match the folder name that Gatsby made for us, in my case **reduxplate.com**
- Include a README? > No
- Default branch name > Leave blank
- Include .gitignore > No (Gatsby includes one for us!)

All configured, the repository you are about to create should look something like this:

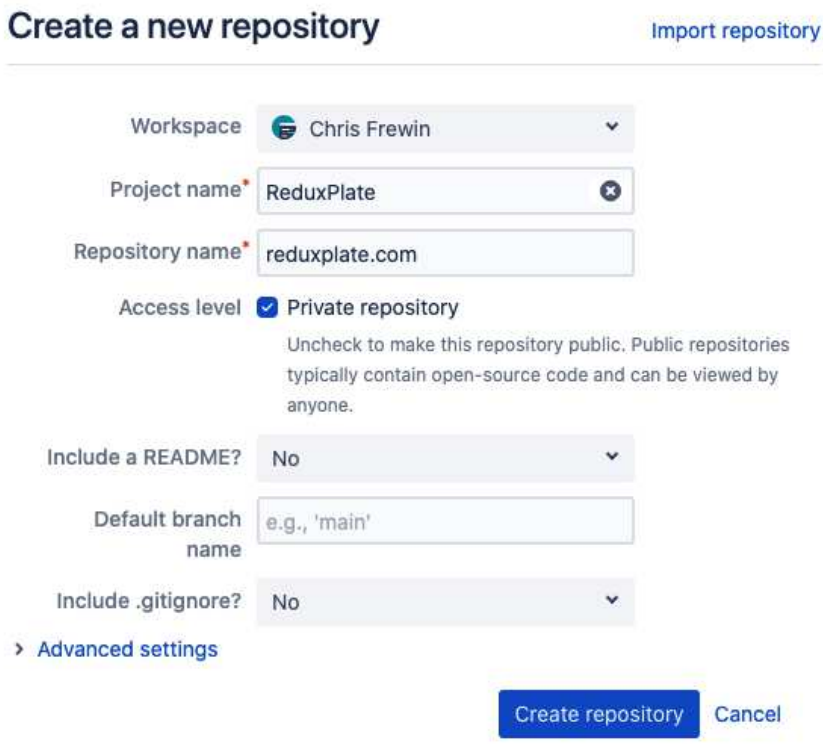


Figure 2.4.: Screenshot of repository fields for your SaaS product.

Go ahead and click the blue ‘Create repository’ button. You should be redirected to your repository’s homepage.

Add the Repository URL to Project and Push the Code
Nice, so we’ve successfully created out Bitbucket repository. Let’s do the signature ‘initial commit’ with our current scaffolded project as a sanity check to make sure things are working.

To achieve this, first make sure your **package.json** reflects the new repository you've just created. As an example, here is the **url** key with my own Bitbucket git URL:

Listing 2.9: </> package.json

```
"repository": {
  "type": "git",
  "url": "https://princefishthrower@bitbucket.org/prin
  cefishthrower/reduxplate.com.git"
},
```

We also need to update the git origin url from the Gatsby starter to our new repository:

Listing 2.10: </> terminal

```
git remote set-url origin https://princefishthrower@bi
tbucket.org/princefishthrower/reduxplate.com.git
```

We are ready to push. Do that with:

Listing 2.11: </> terminal

```
git add .
git commit -m "initial commit"
git push
```

Don't worry about adding files or patterns to the **.gitignore** file, the Gatsby starter has already included one for us!

2.5 Use Netlify for the Frontend DevOps Framework

Chapter Objectives

- Using Netlify and the Netlify CLI to build and deploy our site to a live URL whenever we push to the `master` branch

Alright. So we've got our skeleton Gatsby project and a Bitbucket git repository to track our changes as we build the project. Let's connect Netlify now for automatic builds and publishes to master.

Log In or Create an Account for Netlify

Like Bitbucket, Netlify accounts are free for individuals on the most basic plan. From you dashboard, navigate to the 'Sites' section and click the green button 'New site from Git':

Figure 2.5.: Screenshot of the 'New site from Git' button.

On the resulting page, click the 'Bitbucket' button:

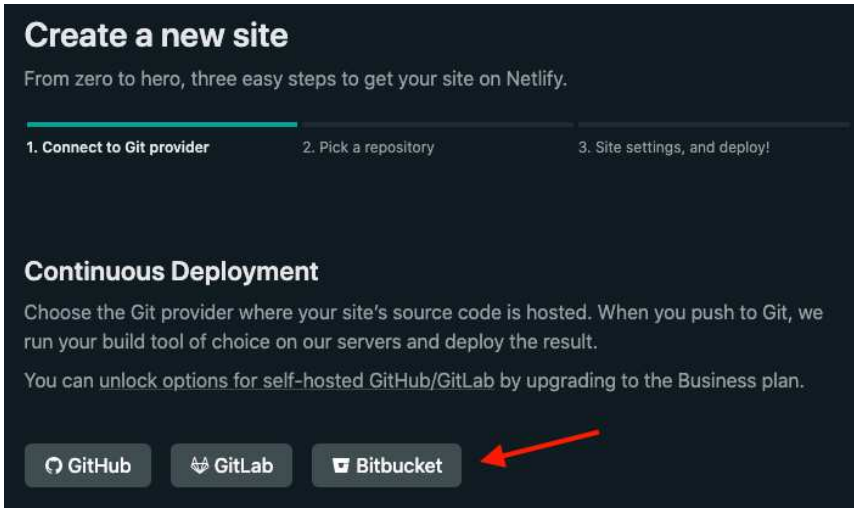


Figure 2.6.: Screenshot of the 'Bitbucket' button.

You'll then be guided through the OAuth process to connect your Bitbucket account to Netlify. After authenticating, you'll be redirected back to Netlify, where you should see a list of all your Bitbucket repositories. You can scroll through or search for the repository you want to connect. In my case that is 'reduxplate.com'. Then click that repository.

Netlify needs just two final variables to start building the site: the build command itself, and then the "publish" folder, in which the artifacts for the site are placed. Since we are using Gatsby, the build command is `npm run build` and the publish folder is `public/`:

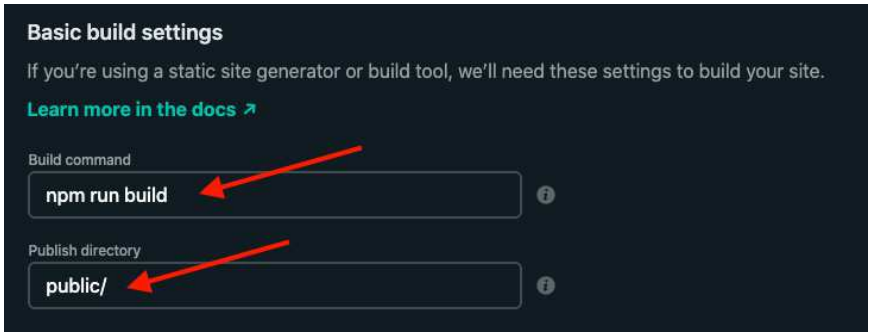


Figure 2.7.: Screenshot of Netlify build settings for a Gatsby site.

Confirm these two variables and feast your eyes as your first build takes off!

Monitoring Your First Build

You can monitor the build log in real time by clicking the specific deploy (in our case so far, the only one under the 'deploys' section):

Figure 2.8.: Screenshot of where to find the detailed deploy log per deploy.

In the deploy log, if you see something like:

Listing 2.12: </> terminal

```
10:50:06 AM: Site is live
10:50:08 AM: Build script success
10:50:37 AM: Finished processing build request in
2m3.485934857s
```

at the bottom of the log, your site was built successfully!

Changing the Randomly Assigned URL

Netlify will go right ahead and assign you a random URL for your site. In my case, I was assigned **sleepy-easley-bb9e3d.netlify.app**.

I like to rename the randomly assigned URL to a name closer to the project at hand, and again, Netlify shines through, allowing us to do that for free. Click the 'Domain settings' button with the gear icon first:

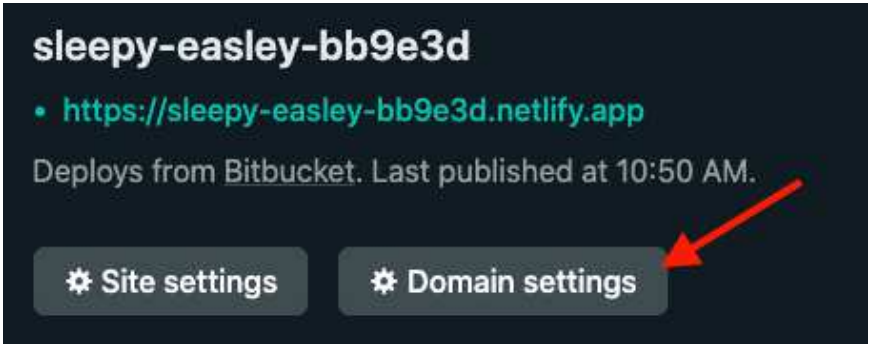


Figure 2.9.: Screenshot of the 'Domain settings' button.

In the resulting page, you should see a list of domains for your project. So far there should only be one: the randomly

assigned url. Navigate to the ‘Options’ dropdown and select ‘Edit site name’:

Figure 2.10.: Screenshot of the ‘Edit site name’ domain option.

Fortunately, the site name `reduxplate` was available, so I used that:

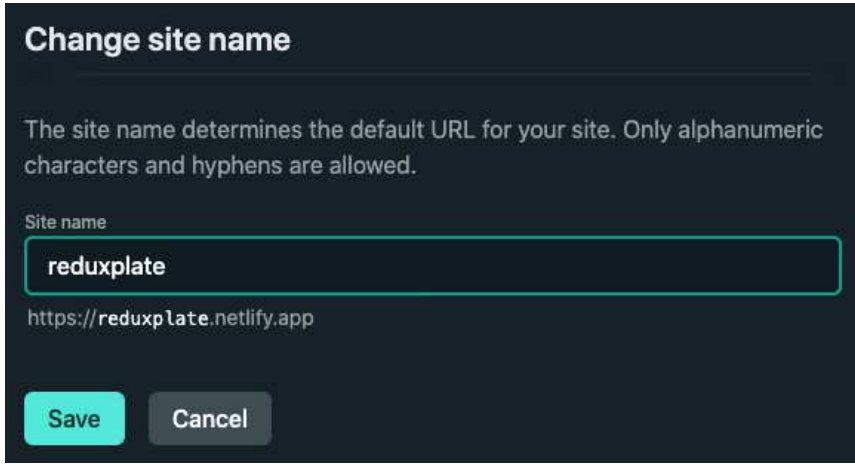


Figure 2.11.: Screenshot of the ‘Edit site name’ domain option.

Great. Your domain should now be at whatever custom name you’ve provided!

ℹ️ A Word On Netlify ℹ️

I've only been using Netlify since February 2021, but I am already hooked on it as a service, and it deserves it's own section here. I find the tiers of their service so generous, that sometimes, it almost feels like *stealing*. On the free plan, you immediately receive 100 GB of bandwidth, 300 build minutes, *and both quotas complete reset each month*. İt truly is an incredible service.

As we'll see later, even with their authentication / authorization service, known as Netlify Identity, you don't pay until you have over *1000 active users*, and if we get that many users on our SaaS product, we don't have to worry about a few additional service fees that'll we'll be more than happy to pay Netlify for! 😉

So, hats off to you, Netlify team, your service is awesome! 👍

Rename the Assigned Netlify Domain Name

Netlify supplies us with a random name for our subdomain, but we can rename this to whatever we'd like! If you've picked a unique enough name for your product, chances are it will be available for your Netlify site domain. If it's already taken, consider adding an hyphen or other small changes so it is a human readable reminder of what the product or project name is. In my case, **`reduxplate`** was available.

2.6 Add a Primary Domain to Netlify via Namecheap

Chapter Objectives

- Buying a domain via Namecheap, and setting that as our primary domain on Netlify

It's great that Netlify right away gives us a live domain (now `reduxplate.netlify.app` in my case), but a custom domain is always better, right? Luckily, Netlify shines through yet again, allowing us to add our own custom domain.

I've already purchased `reduxplate.com` as my primary domain, so I'll use that as an example here.

A Word on Namecheap

While Namecheap does not have perhaps the best UI or services, they are true to their word in that they are *cheap*. For DNS setups outside of what we will need for Netlify, their DNS manager UI has a few quirks that takes some getting used to, but that is outside of the scope of this book. As an overall rule of thumb I *do* recommend Namecheap, as their domain prices are quite competitive.

Domain Name Shopping

Choosing and purchasing a domain is an important step to consider *before* you even start writing code for your product - you don't want to get in the classic trap of building out a brand and logo without an applicable domain to use first! Nowadays you can always find a `.app`, `.us` or similar top level domains for whatever domain name you are looking for, but the classic top level domain `.com` is what I recommend you try and get a hold of. Also realize

that this may take some compromising and / or creativity, and that shorter domain names can be rather expensive!

Adding Netlify DNS

To add your custom domain, first start on your site's dashboard in Netlify. Netlify introduces it as their 'Step 2' for building a site:

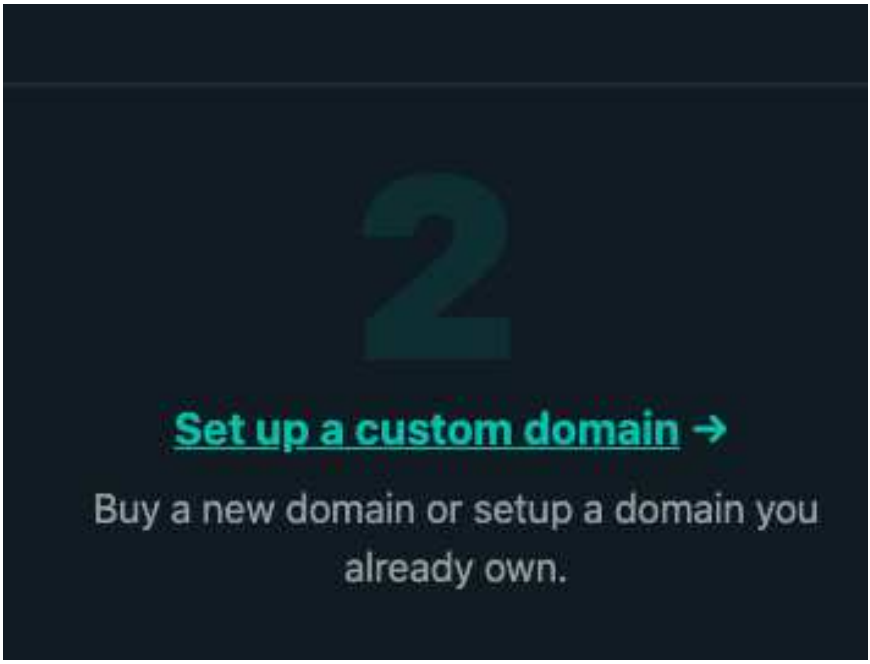


Figure 2.12.: Screenshot of the 'Set up a custom domain' domain step.

in the resulting screen, provide your domain name:

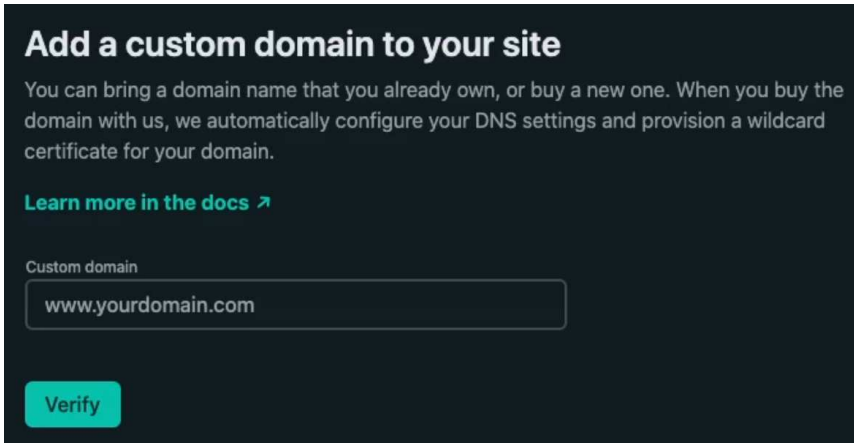


Figure 2.13.: Screenshot of the custom domain input.

you'll then be redirected to your domain lists. Below the original **`netlify.app`** domain, Netlify adds your custom domain, as well as the **`www`** subdomain for it, automatically. However, for both of the new domains, you may notice a warning symbol that says 'Check DNS configuration', in my case for my **`reduxplate.com`** and **`www.reduxplate.com`** domains:

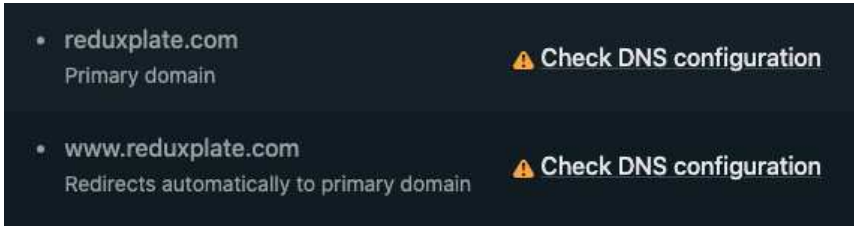


Figure 2.14.: Screenshot of the DNS warnings on the custom domains.

Go ahead and click either of those warning messages. You will have the option of using an A record to point to a Netlify-owned IP address, or the option to use Netlify's DNS. We will be using Netlify's DNS, as they claim that it provides the best possible performance and allows easier use of the branch subdomain feature (which we will be discussing and utilizing later in the book). Go ahead and click the 'Set up Netlify DNS for <your URL here>':

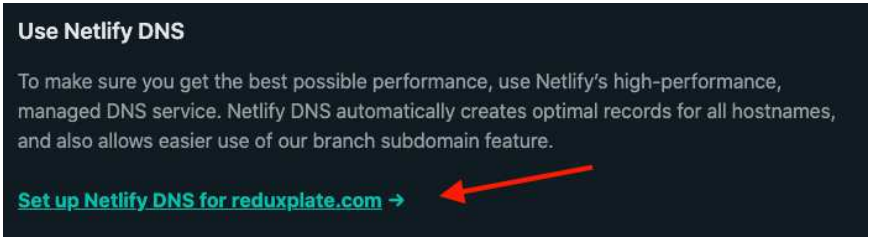


Figure 2.15.: Screenshot of the 'Use Netlify DNS link'.

In the resulting flow, you'll first need to click to 'verify' your domain once again, and in the second step, Netlify will ask if you want to add any custom domain records. We don't need to add any additional DNS entries at this point, so go right ahead to the last step in the flow labeled 'Activate Netlify DNS'. Here, Netlify provides us with a handful of name servers that we need to add to our domain provider:

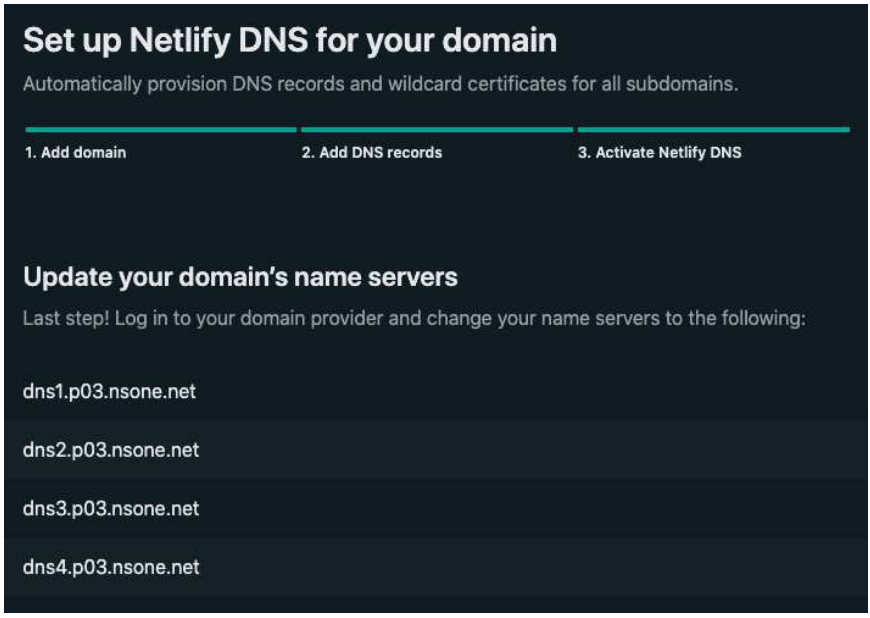


Figure 2.16.: Screenshot of the final step in the Netlify DNS setup, 'Activate Netlify DNS'.

You will then have to navigate to your domain provider to maintain these name server entries. As previously stated, my domain provider is Namecheap, so I can go to my site's dashboard on Namecheap, and click the dropdown for the 'Nameservers' tab, and click the 'Custom DNS' option:

Figure 2.17.: Screenshot of the nameservers dropdown on the Namecheap site dashboard.

In the fields that appear, apply the handful of values that Netlify provided us with:

Figure 2.18.: Screenshot of the Netlify nameservers applied to the custom DNS configuration on Namecheap.

⚠ Domain Name Shopping ⚠

Depending on a variety of factors, like your domain name, your provider, and the nameservers that Netlify gives you, this custom DNS setup could unfortunately take *days* to propagate around the world. As an anecdotal story, when I published my product **The Wheel Screener**, my friends in the United States were able to see the live site within a few hours of me setting up the

custom DNS on Namecheap. Here on my internet in Austria, it took about *two days*, and even a bit longer to show up on the cellular network here. So just be prepared for a definitely non-zero lag time for the DNS propagation. It shouldn't be a problem, you'll have plenty to build in the meantime. 😉

Netlify Domain Recap

We first modified our custom assigned URL to a more memorable and project-relatable one. We then added a custom domain entirely and then leveraged Netlify's DNS, maintaining the name server values in our domain provider (in my case, Namecheap).

We can even see that Netlify has automatically added a redirect from the `www` subdomain to our main domain - this is a nice modern touch that is implemented by many sites today.

With our custom domain set up, and a build being triggered every time we push to the `master` branch, in the next chapter, we will finally start focusing on some client code to get our site looking nice.

3

The Frontend - Implementation

> Design is a funny word. Some people think design means how it looks. But of course, if you dig deeper, it's really how it works. The design of the Mac wasn't what it looked like, although that was part of it. Primarily, it was how it worked. To design something really well, you have to get it. You have to really grok what it's all about. It takes a passionate commitment to really thoroughly understand something, chew it up, not just quickly swallow it. Most people don't take the time to do that.
>
> *(Steve Jobs, 1994)*

3.1 Running the Frontend via the Netlify CLI

From the previous section, we've managed to put together a working continuous integration process using Netlify. We

should start running our client project as if it were on Netlify. Netlify makes this easy for us by providing us with a Netlify CLI tool, which can simulate the Netlify build environment. This will be useful later when we start increasing hte complexity of our frontend, adding things like environment variables, and working with Netlify's serverless functions.

Install the Netlify CLI

We will be following **the official Netlify documentation on how to install and use the Netlify CLI**. Ensure you have the netlify CLI installed globally with:

Listing 3.1: </> terminal

```
npm install -g netlify-cli
```

The Netlify CLI should now be available through either the `netlify` or `ntl` commands in the terminal.

ntl Command

Throughout the remainder of this book, I will use only the shorter `ntl` Netlify CLI command.

Before using the CLI for anything, we should first authenticate with Netlify:

Listing 3.2: </> terminal

```
ntl login
```

This will open up a browser window and prompt you to authenticate with Netlify.

Linking the Frontend Project to Netlify

Once you are authenticated, navigate to the root folder of your frontend repository and issue:

Listing 3.3: </> terminal

```
ntl link
```

This links our local project with all the settings and configurations we've made on the Netlify UI. From now on, whenever working on the frontend repository, instead of issuing **`npm run dev`** commands, issue **`ntl dev`**. This ensures the proper Netlify environment is loaded, and later, that we will be able to use our serverless functions properly.

Environment Variable Example

To illustrate how Netlify injects environment variables into your local machine, head to your Netlify site UI, and go to the 'Deploy settings' screen. Scroll down a bit to the 'Environment' panel and click the 'Edit variables' button:

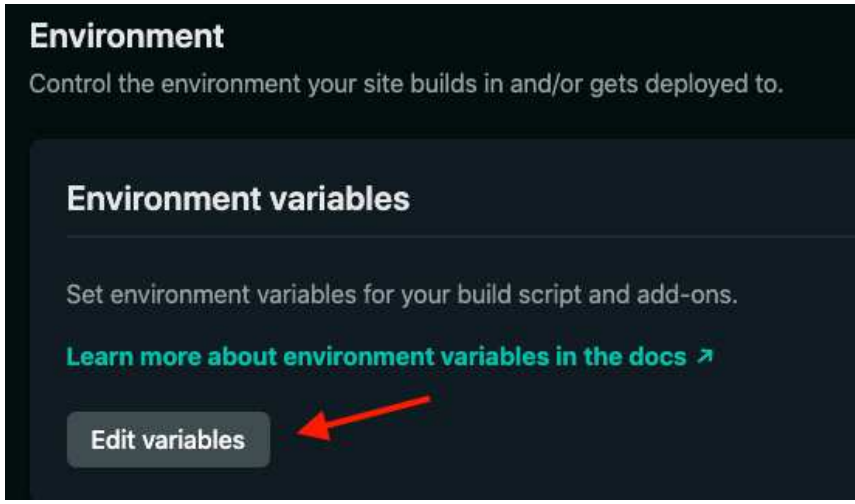


Figure 3.1.: Screenshot of the Netlify environment variables panel and the 'Edit variables' button.

You should have an empty panel with just two inputs that pop up with 'Key' and 'Value'. Here we'll add our first environment variable, and one that I like to maintain myself in all my Netlify projects: `NODE\_VERSION`. Let's set it to the latest LTS release of Node. (As of June 2021 when this edition was first published, that was `14.16.0`). Maintain the 'Key' as `NODE\_VERSION`, and it's value as `14.16.0`, and click 'Save':

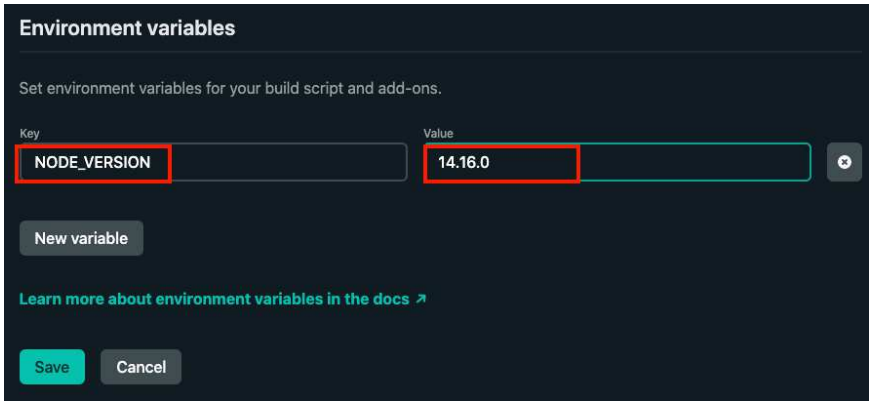


Figure 3.2.: Screenshot of the opened Netlify Environment variables panel, with it's key-value style interface. Here we are adding the NODE_VERSION variable.

⚠ Netlify Configuration Variables and Read-only Variables ⚠

`NODE\_VERSION` is also what is known as a 'Netlify configuration variable' - it will not only be used by us, but also Netlify itself during the build process. You can look at the list of these special configuration variables **on Netlify's official documentation**. There are a few read-only variables as well, so it may be prudent to take a look at that list to make sure you are not trying to overwrite any of them. We can of course also maintain any custom environment variables here, such as API keys and secrets, as we will see shortly when configuring our connection to Stripe.

For the most part, however, you likely won't run

into any issue using realistic names for your environment variables and have to worry about collisions with reserved or special configuration variables in Netlify.

Issue ntl dev

We can now issue **ntl dev**, which will simulate our Netlify environment for us on our local machine. We see in the terminal that Netlify is injecting the **NODE_VERSION** variable for us automatically:

Figure 3.3.: Screenshot of the terminal output after running the Netlify CLI in development mode. The environment variable NODE_VERSION we defined in the Netlify UI is being used in our local environment.

However, if you were to modify your local environment, for example if we wanted to define a different value for **NODE_VERSION** locally, for example to **14.15.0** by issuing:

Listing 3.4: </> terminal

```
export NODE_VERSION=14.15.0
```

Then we would see, after issuing **ntl dev** again, the following message from Netlify:

Figure 3.4.: Screenshot of the terminal output after running the Netlify CLI in development mode. The environment variable NODE_VERSION we defined in the Netlify UI is now being ignored, as the local NODE_VERSION defined in process takes precident.

In summary, Netlify looks for build variables first in our Node `process` before taking the ones we have defined in the Netlify UI. This is an important takeaway which we will leverage later, for example when we have variables that we only wish to use in development mode, such as testing or sandbox API keys.

3.2 Adding SCSS and Bootstrap as the Styling Framework

It's finally time to get into some code!  We'll be doing some styling here, so get your CSS hats on!

Chapter Objectives

- Use scss as our main styling language.
- Add the `gatsby-plugin-sass` plugin, which allows us to import our .scss files directly and will compile our styles inline during the build process.
- Add Bootstrap as our main styling framework.

Remove all Inline Styles

Despite our cleanup in the previous section, there are still a few inline styles hidden throughout the codebase. They are in `Layout.tsx` and `Header.tsx`. Delete all of those now.

A Word on CSS in JS

Though it is still a controversial issue in the frontend community nowadays with 'CSS-in-JS' solutions like styled components and JSS, I like keeping as much styling content as possible in .scss files, and avoid inline styles.

Getting Started with Styling in Gatsby

Following the official Gatsby documentation on **how to add SASS to Gatsby**, first install both the **sass** package and the **gatsby-plugin-sass** plugin:

Listing 3.5: </> terminal

```
npm install sass gatsby-plugin-sass
```

also include the **gatsby-plugin-sass** plugin in your **gatsby-config.js**:

Listing 3.6: </> gatsby-config.js

```
plugins: [
...
`gatsby-plugin-sass`
]
```

Go ahead and create a **styles/** folder within the **src/** folder, and then add a file called **styles.scss**. This will be our root SASS file, and we'll import all custom modules we write into it, including Bootstrap.

Installing and Including Bootstrap

Install Bootstrap with:

Listing 3.7: </> terminal

```
npm install bootstrap@next
```

We will follow **the Bootstrap official documentation on how to add Bootstrap to our SASS styles**. For now, we can import the Bootstrap SASS directly in our **styles.scss** file:

Listing 3.8: </> styles.scss

```
@import "../../node_modules/bootstrap/scss/bootstrap";
```

As the official docs state, this import should be the first one, excluding theming variable changes we will make. All other imports to custom style sheets should follow it. In a later section we will work on tree shaking out only the CSS classes which are used in our project to keep our CSS footprint low. For now, importing the entire Bootstrap library will work for our needs.

Theming For Bootstrap

Again, following the official documentation, we will create our own **_variables.scss** file and import that ahead of the bootstrap import. For my ReduxPlate project, I've settled on using the Redux purple (hex code **#764abc**). For fonts, think that the Montserrat font (weight 500 for normal text and 700 for bold) looks nice for all titles and text, and Fira Code for monospace fonts. Bootstrap exposes all of these various theming elements via SASS variables. A nice tool which can help you visualize how various Bootstrap components will look is **Bootstrap Build**. Ultimately,

our **_variables.scss** file will look like this:

Listing 3.9: </> _variables.scss

```
@import url("https://fonts.googleapis.com/css2?family=
Montserrat:wght@500;700&display=swap");
@import url("https://fonts.googleapis.com/css2?family=
Fira+Code:wght@500;700&display=swap");

$purple: #764abc;
$primary: $purple;
$font-family-sans-serif: "Montserrat",-apple-system,
BlinkMacSystemFont, "Segoe UI", Roboto, "Helvetica
Neue", Arial, "Noto Sans", sans-serif, "Apple Color
Emoji", "Segoe UI Emoji", "Segoe UI Symbol", "Noto
Color Emoji";
$font-family-monospace: "Fira Code",SFMono-Regular,
Menlo, Monaco, Consolas, "Liberation Mono", "Courier
New", monospace;
```

and then we have to import that before the Bootstrap import in **styles.scss**, such that the whole file looks like this:

Listing 3.10: </> styles.scss

```
@import 'variables';
@import "../../node_modules/bootstrap/scss/bootstrap";
```

Import styles.scss into gatsby-browser.js

The **src/styles/styles.scss** file is our one source of truth for all styling in the app. This will be the file we import into **gatsby-browser.js**:

Listing 3.11: </> gatsby-browser.js

```
import "./src/styles/styles.scss"
```

We should now see our theming applied site-wide.

3.3 Creating React Components for Your Site's Layout

We have some nice looking theming for our SaaS product. Let's now start creating React components across our site!

Chapter Objectives

- Create a navigation component
- Create a footer component
- Improve the header component

Creating a Navigation Component

Under **components/**, create a new folder called **layout/**, and then create a new file called **Nav.tsx**. Add the following code to **Nav.tsx**:

Listing 3.12: </> Nav.tsx

```
import { Link } from "gatsby"
import { StaticImage } from "gatsby-plugin-image"
import * as React from "react"

export interface INavProps {
  siteTitle: string
}

export function Nav(props: INavProps) {
  const { siteTitle } = props
  return (
    <nav className="navbar bg-primary">
      <Link className="navbar-brand text-light" to="/">
        <StaticImage
          src="../../images/gatsby-icon.png"
          className="d-inline-block align-top mx-3"
          alt=""
          layout="fixed"
          width={30}
          height={30}
        />
        {siteTitle}
      </Link>
    </nav>
  )
}
```

Here, we use some helpful Bootstrap classes to style our nav, and are currently using **gatsby-icon.png** as a placeholder for our site's logo. We also pass down a **siteTitle** prop so that if we change the title in the **gatsby-config**, it will be changed everywhere.

You can also move **Layout.tsx** into the **layout** folder. If you are using Visual Studio Code and TypeScript, the imports should automatically be updated for you - just be sure to save after dragging and dropping **Index.tsx**!

Be sure to then import and use the **Nav** component in **Layout.tsx**:

Listing 3.13: </> Layout.tsx

```
...
import { Nav } from "./Nav" // new
...
return (
  <>
    <Nav siteTitle={data.site.siteMetadata.title} />
    // new
    <main>{children}</main>
  </>
)
...
```

Creating a Footer Component

Just as with **Nav.tsx**, create a **Footer.tsx** file under **Layout.tsx**. Add this code to it:

Listing 3.14: </> Footer.tsx

```
import * as React from "react"

export function Footer() {
  return (
    <footer className="fixed-bottom bg-primary
    text-light text-center text-lg-start">
      © {new Date().getFullYear()} <a
      className="link-light"
      href="https://fullstackcraft.com">Full Stack
      Craft</a>
    </footer>
  )
}
```

here we leverage the **link-light** utility class from Bootstrap so we can easily see it against the purple (primary) colored background. Pretty basic, but good enough for now.

As with **Nav.tsx**, import and place the **Footer** component in codewordLayout.tsx:

Listing 3.15: </> Footer.tsx

```
...
import { Footer } from "./Footer" // new
...

return (
  <>
    <Nav siteTitle={data.site.siteMetadata.title} />
    <main>{children}</main>
    <Footer/> // new
  </>
)

export default Layout
```

Improve the Header Component

Move the **Header** component into the **layout/** folder as well. We will now begin filling it out, including a nice little SCSS widget I thought up which leverages CSS pseudo elements.

First, **Header.tsx** can be replaced with the following code:

Listing 3.16: </> Header.tsx

```
import { useStaticQuery, graphql } from 'gatsby';
import * as React from 'react';
import * as styles from
'../../../styles/modules/header.module.scss'
import { PlateWidget } from
'../../widgets/PlateWidget';

export function Header () {
  const data = useStaticQuery(graphql`
    query HeaderQuery {
      site {
        siteMetadata {
          description
        }
      }
    }
  `)
  return (
    <header className={styles.header}>
    <h1 className={styles.title}>
      <span className="text-primary">Redux</span>
      <span className="text-light">Plate</span>
      <PlateWidget />
    </h1>
    <h2 className={styles.subtitle}>{data.site.siteMet
    adata.description}</h2>
  </header>
  );
}
```

Where **header.module.scss** contains the following:

Listing 3.17: </> header.module.scss

```
@import "../variables";

.header {
  display: flex;
  flex-wrap: wrap;
  flex-direction: column;
  text-align: center;
}

.title {
  display: flex;
  flex-wrap: wrap;
  flex-direction: row;
  justify-content: center;
  font-size: 70px;
}

.plateText {
  position: relative;
  z-index: 1;
}

.subtitle {
  font-size: 35px;
  color: $primary;
}
```

The **PlateWidget** component actually holds the markup of the background pseudo element:

Listing 3.18: </> PlateWidget.tsx

```
import * as React from 'react';
import * as styles from
'../../styles/modules/plate-widget.module.scss'

export function PlateWidget () {
  return (
    <span className={styles.plate}>
      <span className={styles.topScrews}>
        <span className={styles.bottomScrews}></span>
      </span>
    </span>
  );
}
```

Where **plate-widget.module.scss** contains the following:

Listing 3.19: </> plate-widget.module.scss

You'll notice as a little easter egg, when hovering on the plate that there is a kind of 'unscrew' animation, where the plate appears to lift off the page! Worried about responsive styling? The way these elements are arranged and marked up with flex styling allows them to be quite responsive! Go ahead in exploring how various widths look. There should be no troubles.

3.4 Creating an Interactive Code Editor Widget

I decided to put a code editor immediately on the home page, as I think an interactive example draws interest and converts to the most customers. Later, this editor will actually interact with our custom .NET API, but for now, let's focus on it's appearance. We will be using the `\at monaco-editor/react`, which is a React wrapper for the `monaco-editor`, which is the Microsoft-owned open source repository for their powerful Intellisense editor, used in Visual Studio Code, Microsoft's online TypeScript Playground, and CodeSandbox.

Layout Considerations

We can begin to imagine for our SaaS product that we would like a typical code editor UI - a tabbed interface with file names, which, when clicked, reveal the code in those files. The code can then be edited in the editor, and we should (thus the choice of using `monaco-editor`)

Naming and Building the Code Editor Component

Ultimately, I decided on the name of `EditorWidget` for the component. Create a new folder called `utils` under `src/components/`, and create a new TypeScript React component file called `EditorWidget.tsx`. I put this component

under the **utils/** folder because it will be used on multiple pages and locations in our app.

Props for the EditorWidget component

Our editor widget can have an option title above the editor, then we need a series of 'files' to render. These 'files' should have both a label for the file name, the code contents of that 'file', and if it is active or not. We can define an interface IEditorSettings as a helper to store these two values per file, and then we can pass an array of this settings interface to the **EditorWidget** component. I called it **IEditorSettings**. It's actually the first non-prop interface we're creating so far on the frontend, so let's create a new folder under the **src/** folder called interfaces/. Go ahead and create a new file **IEditorSettings.ts**, and add this:

Listing 3.20: </> IEditorSettings.ts

```
export default interface IEditorSetting {
  fileLabel: string
  code: string
}
```

A Word on the 'interfaces' Folder

Some developers like including interfaces and types close to where they are used in various React components, so they act as more of a namespace. I typically do not follow this pattern for two reasons:

1. Many custom interfaces we define will be used in more than one component
2. JavaScript and TypeScript do not natively have a

> concept of namespaces, and so I generally organize all interfaces, enums, and types into respective folders labeled so
>
> No matter what style you decide, Visual Studio Code's Intellisense doesn't care either way and will automatically update the import locations for you as you rearrange and drag and drop files. Just know that it is my preference to put all non-prop interfaces in the **src/interfaces** folder.

With **IEditorSettings** defined, we can now fully define the props for our component, **IEditorWidgetProps**:

Listing 3.21: </> IEditorWidgetProps.ts

```
export interface IEditorWidgetProps {
  editorTitle?: string
  editorSettings: Array<IEditorSetting>
}
```

Whew. All done with props. Let's get on to the body of the component.

State Management and Setup

As is my style, I destructure all props out from **props**. We'll then immediately make the **editorSettings** prop stateful - we'll need to track all changes to it for each editor.

Listing 3.22: </> EditorWidget.tsx

```
...
const { editorTitle, editorSettings } = props
const [editorSettingsState, setEditorSettingsState] =
useState<
  Array<IEditorSetting>
>(editorSettings)
...
```

That's all the React state variables we should need for our component to be functional. Let's move on to what we will render for the component.

Rendering the Code Editor and File Tabs

The `@monaco-editor/react` package makes rendering the editor part of our`EditorWidget` component rather easy. First install the package:

Listing 3.23: </> terminal

```
npm install @monaco-editor/react
```

Then include it in `EditorWidget`:

Listing 3.24: </> EditorWidget.tsx

```
return (
  ...
  <Editor
    height="500px"
    defaultLanguage="typescript"
    defaultValue={"// a comment"}
    options={{
      minimap: { enabled: false },
      scrollBeyondLastLine: false
    }}
  />
  ...
)
```

As mentioned, we should also make a way to have multiple tabs above the editor. To have such a display, I'm going to leverage some Bootstrap nav tab styles on an `<ul>` element. The active tab will then simply need the 'active' class applied:

Listing 3.25: </> EditorWidget.tsx

```
  <ul className="nav nav-tabs">
  {editorSettingsState
    .map(editorSettings => {
      const { fileLabel } = editorSettings;
      const className =
      editorSettings.isActive
          ? "nav-link active font-monospace"
          : "nav-link font-monospace"
      return (
        <li className="nav-item" onClick={() =>
        onChangeTab(fileLabel)}>
          <button className={className}>
            {fileLabel}
          </button>
        </li>
      )
    })}
</ul>
```

So far, you may have noticed two helper functions being used in our render, **onChangeCode** and **onChangeTab**. Those are defined as follows:

Listing 3.26: </> EditorWidget.tsx

```
const onChangeCode = (code: string) => {
  // only modify the code string of the file which is
  active
  setEditorSettingsState(editorSettingsState.map(edito
  rSetting =>
  {
    if (editorSetting.isActive) {
      editorSetting.code = code
    }
    return editorSetting
  }))
}

const onChangeTab = (fileLabel: string) => {
  setEditorSettingsState(editorSettingsState.map(edito
  rSetting =>
  {
      editorSetting.isActive = editorSetting.fileLabel
      === fileLabel
    return editorSetting
  }))
}
```

But there's something a bit repetitive about what we are doing in each of these handlers: both are using `map` to return a new array where a value has been updated based on a test criteria. This should be refactored to make our codebase even cleaner.

Updating an Array the TypeScript Way

`onChangeCode` and `onChangeTab` are really doing the same thing: they're changing parts of an object array based on a test criteria. We will likely be using similar functionality in multiple locations around the app, and so we should build a fancy TypeScript function that will do this type of array

manipulation for us. I call it simply **updateArray**. This will be the first **util** function we create, so, create a new folder called **utils**, and the file **updateArray.ts**, with this in it:

Listing 3.27: </> updateArray.ts

```
// Updates an object array at the specified update key
with the update value,
// if the specified test key matches the test value.
// Optionally pass 'testFailValue' to set a default
value if the test fails.
export const updateArray = <T, U extends keyof T, V
extends keyof T>(options: {
  array: Array<T>
  testKey: keyof T
  testValue: T[U]
  updateKey: keyof T
  updateValue: T[V]
  testFailValue?: T[V]
}): Array<T> => {
  const {
    array,
    testKey,
    testValue,
    updateKey,
    updateValue,
    testFailValue,
  } = options
  return array.map(item => {
    if (item[testKey] === testValue) {
      item[updateKey] = updateValue
    } else if (testFailValue !== undefined) {
      item[updateKey] = testFailValue
    }
    return item
  })
}
```

We can then refactor **onChangeCode** and **onChangeTab** to look like this:

Listing 3.28: </> EditorWidget.tsx

```
const onChangeCode = (code: string) => {
  // only modify the code string of the file which is
  active
  setEditorSettingsState(updateArray<
    IEditorSetting,
    "isActive",
    "code"
  >({
    array: editorSettingsState,
    testKey: "isActive",
    testValue: true,
    updateKey: "code",
    updateValue: code,
  }))
}

const onChangeTab = (fileLabel: string) => {
  setEditorSettingsState(updateArray<
    IEditorSetting,
    "fileLabel",
    "isActive"
  >({
    array: editorSettingsState,
    testKey: "fileLabel",
    testValue: fileLabel,
    updateKey: "isActive",
    updateValue: true,
    testFailValue: false,
  }))
}
```

this solution is rather verbose, but we don't have to

think about write any **map** logic or **if** statement checks - **updateArray** does that all for us *and* it is strongly typed.

3.5 Some Styling for the Editors

The **width** property for the Monaco Editor is **100\%** by default. This value messes with our side-by-side layout when using flexbox. We should also consider more narrow screens like iPads, where the editors should become single column and take the full width of the screen. To handle this, we'll create a new Sass module, **editor.module.scss**:

Listing 3.29: </> editor.module.scss

```
@import
"../../../node_modules/bootstrap/scss/functions";
@import
"../../../node_modules/bootstrap/scss/variables";
@import "../../../node_modules/bootstrap/scss/mixins";

.editorWrapper {
    flex-grow: 1;
    flex-shrink: 1;
    flex-basis: 0;
}

@include media-breakpoint-down(lg) {
    .editorWrapper {
        width: 100% !important;
    }
}
```

3.6 Adding a Custom Theme to the Editor

The default Monaco Editor theme is **vs-light**, but it is a bit *too* bright for our application - there's no contrast with the background of our site, which is white as well. We're going to introduce the GitHub theme, which is still a nice looking theme, but will demarcate the borders of the editor clearly. (Later, in the advanced frontend implementation, we will look at how to dynamically change this when we introduce dark mode).

First we'll install the package **monaco-themes** with npm:

Listing 3.30: </> terminal

```
npm install monaco-themes
```

We will need to access the **monaco** object to define and set a new theme. Luckily, the **\at monaco-editor/react** package includes a **beforeMount** callback in their component which includes the **monaco** object as an argument. We can load and set the GitHub theme there:

Listing 3.31: </> EditorWidget.tsx

```
// any time an editor is about to mount, set the theme
to github
const handleBeforeMount = (monaco: Monaco) => {
  import("monaco-themes/themes/Github.json").then(data
  => {
    monaco.editor.defineTheme("github", data)
    monaco.editor.setTheme("github")
  })
}
```

Excellent. We are finished with our editor widget component. The full contents we arrive at for EditorWidget are:

Listing 3.32: </> EditorWidget.tsx

```
import * as React from "react"
import * as styles from
"../../styles/modules/editor.module.scss"
import EditorID from "../../enums/EditorID"
import { codeEdited, tabClicked } from
"../../store/editors/editorsSlice"
import { useAppDispatch, useAppSelector } from
"../../hooks/redux-hooks"
import Editor from "@monaco-editor/react"

export interface IEditorWidgetProps {
  editorID: EditorID
}

export function EditorWidget(props:
IEditorWidgetProps) {
  const { editorID } = props
  const { editorTitle, editorSettings } =
  useAppSelector(
    state => state.editors.editors[editorID]
  )
  const dispatch = useAppDispatch()

  const onChangeCode = (code: string) => {
    dispatch(
      codeEdited({
        editorID,
        code,
      })
    )
  }

  const onChangeTab = (fileLabel: string) => {
```

The code editor to modify what state the user would like to generate looks like this:

Figure 3.5.: Screenshot of the single tab code editor.

It's just the single file that I call **`state.ts`**, so we see that one tab and the one editor. A traditional generation output from this state should ultimately result in three files, a **`types.ts`** File, a **`actions.ts`** file, and a **`reducers.ts`** file (at least when not using **`@reduxjs/toolkit`**). Such a config results in our **`EditorWidget`** to look like this:

Figure 3.6.: Screenshot of the multi tabbed code editor.

We will again reuse this component later on the 'App' page in just a few sections.

Creating a 'Try It' Widget

Now that we've got our Editor Widget, we can create the desired 'Try it' component for visitors to immediately see the power of ReduxPlate for themselves. Create a new file `TryItWidget.tsx` under the same `home/` folder, and add this to it:

Listing 3.33: </> TryItWidget.tsx

```
import * as React from "react"
import { EditorWidget } from "./EditorWidget"
import { TryItButtons } from "./TryItButtons"

export function TryItWidget() {
  return (
    <div className="container text-center">
      <div className="d-flex flex-wrap
      justify-content-center">
        <EditorWidget
          editorTitle="Desired Redux State"
          editorSettings={[
            {
              fileLabel: "state.ts",
              code: `// Feel free to edit with
              whatever state you need.
// Then click 'Generate!' below!
export interface ReduxPlateState {
  firstName: string
  lastName: string
  isLoggedIn: boolean
  roles: Array<string>
}`,
              isActive: true
            },
          ]}
        />
        <EditorWidget
          editorTitle="Generated Code"
          editorSettings={[
            {
              fileLabel: "types.ts",
              code: `// types.ts
// Nothing here yet.
// Click 'Generate!' below!`,
```

It's two of our **EditorWidget** components side by side in a flex box.

What is the TryItButtons component?

I decided to put two buttons under our side-by-side editors: one says 'Generate!' which will actually kick off a real example functionality of what ReduxPlate can do, and another button to prompt visitors to preview the full app, labeled 'Try Full App'. I organized those into a component called <TryItButtons/>. Create a new file **TryItButtons.tsx** under the same **home/** folder, and add this to it:

Listing 3.34: </> TryItButtons.tsx

```
            },
            {
              fileLabel: "reducer.ts",
              code: `// reducer.ts
// Nothing here yet.
// Click 'Generate!' below!`,
isActive: false
            },
            {
              fileLabel: "actions.ts",
```

```
import { Link } from "gatsby"
import * as React from "react"

export function TryItButtons() {
  return (
    <div className="d-flex justify-content-center">
      <button className="btn btn-outline-primary m-3">
        Generate!
      </button>
      <Link to="/app" className="btn btn-primary m-3">
        Try Full App
      </Link>
    </div>
  )
}
```

Take note of the class names on each - Bootstrap helps us out a lot with the style of these buttons. Even though the 'Try Full App' is actually a Gatsby **Link** component (which ultimately becomes an anchor tag), it will still appear identical to a button - nice!

Refactoring Button Styles

The default Bootstrap button styles look a little too playful for what will be a serious developer tool. To make it more serious looking, let's set all border radius throughout the Bootstrap styles to 0. To do that, add the following variables to **_variables.sccs**:

Listing 3.35: </> _variables.scss

```
$border-radius: 0;
$border-radius-lg: 0;
$border-radius-sm: 0;
$badge-pill-border-radius: 0;
```

I then decided to use the class **btn-outline-primary** on the 'Generate!' button instead of **btn-primary** so it can contrast the 'Try Full App' button. The call to action button, 'Try Full App', retains the 'primary' styling.

Also note that the 'Generate!' button has no **onClick** handler yet - it won't do anything if you click it. Likewise, the 'Try Full App' button will take us to the **app/** page, but that page doesn't exist yet, so we'll get the Gatsby development 404 page.

Extending the Index (Home) Page

Now that we've got our **TryItWidget** component, start by creating a **pages/** folder under **components/**, and then another folder **home/** under that. Then create the file **Home.tsx**.

A Word on React Component Organization in Gatsby

When using Gatsby, I like to keep the components in the **pages** folder as simple as possible. This is to signify that these are actual HTML pages that will be created, and their actual content can be abstracted away into various components under the **components/** folder.

We can now add the **Header** and **TryItWidget** components to the **Home.tsx** file:

Listing 3.36: </> Home.tsx

```
import * as React from "react"
import { Header } from "./Header"
import { TryItWidget } from "./TryItWidget"

export function Home() {
  return (
    <>
      <Header />
      <TryItWidget />
    </>
  )
}
```

3.7 Recipe: Creating a Production-Ready SVG

It's now time to create the logo for our SaaS Product. In this section, I'll teach you how to make a low footprint SVG, ready to use simply as a static SVG, or to build as a React component to be animated or otherwise dynamically modified.

My typical process of producing a production-ready SVG looks something like this:

1. Create the SVG, either by hand or in code. (If it is simple enough, you may find you can create it in a code-based fashion. I typically use Inkscape as my SVG weapon of choice.)

2. Load the SVG into the amazingly powerful SVGOMG

3. Export the SVG markup after SVGOMG does its thing

4. Examine at the SVG code - there are usually some

manual steps that can be taken to even *further* simply the SVG

5. For use as a React component, convert the SVG markup to JSX syntax using the HTML to JSX Compiler
6. Create custom styles for the SVG
7. Implement dynamic capabilities of the SVG

Getting Started

I started creating my logo in Inkscape, reusing the purple hex from the `\_variables.scss` we've already defined. No matter what way you build your logo, since this logo will be used both as the favicon, I recommend you frequently look at how it appears at multiple zoom levels, from very far out (to simulate it's appearance as a favicon), to further in (to simulate it's appearance on your homepage, header, or footer). Ultimately, for ReduxPlate, I arrived at this logo:

Figure 3.7.: The square plate-like logo for ReduxPlate.

I recommend paths over SVG shape objects like **`circle`** or **`rect`**, as paths can be joined and optimized with a tool like SVGOMG, as we will seen in the next part. I also recommend sizing the SVG to a nice round number in pixels. I settled on 250px x 250px. Furthermore, for Inkscape users, I recommend saving two copies of the logo you build - first as an 'Inkscape' SVG, which includes additional markup that Inkscape uses, but also as a 'plain' SVG - this has all the additional Inkscape markup removed, and is the one we will be importing to SVGOMG.

Using SVGOMG to Optimize the Logo

Once you have designed an SVG that you like, head over to **SVGOMG**. In the sidebar, either paste in or upload your SVG:

Figure 3.8.: Screenshot of the sidebar options in SVGOMG.

A word on SVGOMG

SVGOMG is an amazing tool that I've been using for years now. It's my one stop shop for trimming down and optimizing SVGs. No matter where you get your SVG, whether it is from your design team, an asset pack, or you built it yourself, I recommend you run it through SVGOMG. You can almost *always* reduce the footprint of an SVG at no visual cost!

Once the SVG has been imported, you should see a preview of it directly in your browser. The first and largest footprint saver will likely come from the 'Precision' bar, where sliding to the left will produce less precise paths and sliding

to the right will produce more precise paths:

Figure 3.9.: Screenshot of the sidebar options in SVGOMG.

Typically, I have found you can get quite close to a precision of '0' before noticing visible differences in the SVG. While tuning the 'Precision', be sure to monitor the savings in the icons on the bottom right:

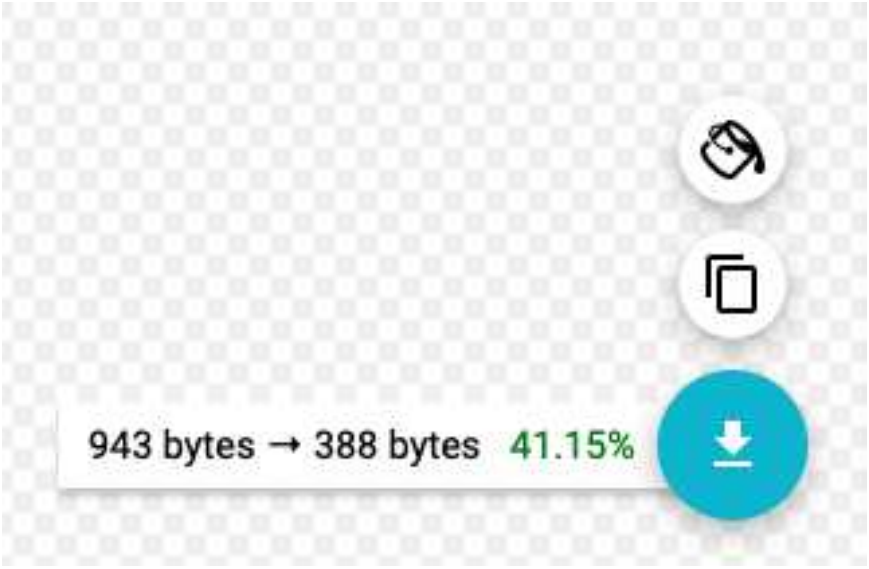


Figure 3.10.: Screenshot of the option buttons in SVGOMG (Background toggle, copy to clipboard, and export).

In addition to using the ‘Precision’ bar, I typically check the option ‘Prefer viewBox to width/height’. You can explore and toggle the other numerous options in SVGOMG, but I find that the default values work quite well.

When you are done, click either the export button to trigger a browser download of the SVG, or the copy button, to copy the SVG markup to your clipboard (both of these buttons are shown in 3.10). Typically I simply copy the markup to the clipboard and paste it into an empty Visual Studio Code. The SVG markup produced by SVGOMG is as follows:

Listing 3.37: </> logo.svg

```
<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0
161 161">
      <g paint-order="fill markers stroke">
            <path d="M35.5 0h90A35.5 35.5 0 01161
            35.5v90a35.5 35.5 0 01-35.5
            35.5h-90A35.5 35.5 0 010
            125.5v-90A35.5 35.5 0 0135.5 0z"
            fill="#bba4de" />
            <path d="M72.7 38.4a31.4 31.4 0
            01-31.4 31.4A31.4 31.4 0 0110 38.4
            31.4 31.4 0 0141.3 7a31.4 31.4 0
            0131.4 31.4z" fill="#8468b2" />
            <path d="M51.9 22.2L41.3 32.8 30.8
            22.2l-5.6 5.6 10.5 10.6L25.2 49l5.6
            5.6L41.3 44 52 54.6l5.6-5.6-10.6-10.6
            10.6-10.6z" />
            <path d="M151 38.4a31.4 31.4 0 01-31.3
            31.4 31.4 31.4 0 01-31.4-31.4A31.4
            31.4 0 01119.7 7 31.4 31.4 0 01151
            38.4z" fill="#8468b2" />
            <path d="M130.2 22.2l-10.5 10.6L109
            22.2l-5.6 5.6 10.6 10.6L103.5 49l5.6
            5.6L119.7 44l10.5 10.6L136
            49l-10.6-10.6 10.6-10.6z" />
            <path d="M72.7 122.6A31.4 31.4 0
            0141.3 154 31.4 31.4 0 0110 122.6a31.4
            31.4 0 0131.3-31.3 31.4 31.4 0 0131.4
            31.3z" fill="#8468b2" />
            <path d="M51.9 106.4L41.3
            117l-10.5-10.6-5.6 5.7 10.5 10.5-10.5
            10.6 5.6 5.6 10.5-10.6L52
            138.8l5.6-5.6-10.6-10.6 10.6-10.5z" />
            <path d="M151 122.6a31.4 31.4 0
            01-31.3 31.4 31.4 31.4 0 01-31.4-31.4
            31.4 31.4 0 0131.4-31.3 31.4 31.4 0
            0131.3 31.3z" fill="#8468b2" />
            <path d="M130.2 106.4L119.7 117 109
            106.4l-5.6 5.7 10.6 10.5-10.6 10.6 5.6
            5.6 10.6-10.6 10.5 10.6
            5.7-5.6-10.6-10.6 10.6-10.5z" />
      </g>
</svg>
```

In this particular case, we see that SVGOMG has produced a group node (i.e. <g paint-order="fill markers stroke">) around all

of our paths. This appears to be an artifact from Inkscape's version of the SVG, and likely won't affect the visual appearance of the logo. As a sanity check, let's remove that group node and re-paste the entire SVG back into SVGOMG to check that it has remained visually the same. Indeed, we see that there has been no change. (Make sure to refresh SVGOMG entirely before pasting in the markup, so you can be sure you are working with a blank slate in SVGOMG!) You may need to repeat this process multiple times for other nodes on your SVG, massaging it until the markup is as clean as possible.

Finally, one pet peeve of mine is that the `fill` property is left of the `d` property in all of the nodes. In an editor, it is a bit annoying to scroll all the way to the right to see what the `fill` property is for each `path`. I will move all of these fills to the left, as the first property. We can also see for each of the screws groove path that the `fill` property has been omitted, since black is the default fill for a path. But what if we want to modify this color later? In this case it is best to explicitly provide the fill color, so we can change it later if we wish, and also so that we can see in code and recognize immediately that this path represents the screw grooves.

So, with not *too* much trouble, we have arrived at our SaaS product's logo final SVG markup:

Listing 3.38: </> logo.svg

```
<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0
161 161">
    <path fill="#bba4de" d="M35.5 0h90A35.5 35.5 0
    01161 35.5v90a35.5 35.5 0 01-35.5 35.5h-90A35.5
    35.5 0 010 125.5v-90A35.5 35.5 0 0135.5 0z"/>
    <path fill="#9877cd" d="M43 29.1A12.9 12.9 0
    0130.3 42 12.9 12.9 0 0117.4 29a12.9 12.9 0
    0112.8-12.8A12.9 12.9 0 0143.1 29zM143.6 29.1A12.9
    12.9 0 01130.8 42 12.9 12.9 0 01117.9 29a12.9 12.9
    0 0112.9-12.8A12.9 12.9 0 01143.6 29zM143.6
    132a12.9 12.9 0 01-12.8 12.8 12.9 12.9 0
    01-12.9-12.9 12.9 12.9 0 0112.9-12.8 12.9 12.9 0
    0112.8 12.8zM43 132a12.9 12.9 0 01-12.8 12.8 12.9
    12.9 0 01-12.8-12.9 12.9 12.9 0 0112.8-12.8 12.9
    12.9 0 0112.9 12.8z" />
    <path fill="#000000" d="M34.2 20.8l-4 4-3.9-4-4.3
    4.4 3.9 3.9-4 4 4.4 4.3 4-4 3.9 4 4.3-4.4-4-3.9
    4-4zM134.7 123.6l-4 4-3.8-4-4.4 4.4 4 4-4 3.8 4.4
    4.4 3.9-4 3.9 4 4.3-4.4-3.9-3.9 4-3.9zM34.2
    123.6l-4 4-3.9-4L22 128l3.9 4-4 3.8 4.4 4.4 4-4
    3.9 4 4.3-4.4-4-3.9 4-3.9zM134.7 20.8l-4
    4-3.8-4-4.4 4.4 4 3.9-4 4 4.4 4.3 3.9-4 3.9 4L139
    33l-3.9-3.9 4-4z" />
</svg>
```

Add the Optimized Logo to the Gatsby Project

Go ahead and paste the finalized SVG into a new file **logo.svg**, under the **src/images** folder. Then don't forget to update the value of the icon under the **gatsby-plugin-manifest** in **gatsby-config.js**:

Listing 3.39: </> gatsby-config.js

```
...
{
  resolve: `gatsby-plugin-manifest`,
  options: {
    ...
    icon: `src/images/logo.svg`, // updated
  },
},
...
```

Luckily, the **gatsby-plugin-manifest** can handle SVG files for the icon file - and it will work well with this plugin, as the plugin rasterizes any other icon sizes needed for various devices and icons, like for Apple home screens and Windows icons.

You will also need to update the icon in **Nav.tsx**:

Listing 3.40: </> Nav.tsx

```
...
export function Nav(props: INavProps) {
...
<StaticImage
  src="../../images/logo.svg" // updated
  className="d-inline-block align-top mx-3"
  alt=""
  layout="fixed"
  width={30}
  height={30}
/>
...
```

You can now delete the **gatsby-icon.png** from the **src/images/** folder.

Great. Now we should be seeing our newly fashioned logo

in both the nav:

Figure 3.11.: Screenshot of the logo in the nav.

and as the site's favicon:

Figure 3.12.: Screenshot of the logo as a favicon.

For the logo displayed in the nav, I'd like to do a little something extra and fancy. We're going to add some animations to it! Since these animations should be dynamic based on location in the application (we don't want to distract customers with it on the 'App' part of our site), we'll be transforming it into a reusable React component.

Create a React Component for the Logo

Now that we have our clean SVG markup, it is also possible for us to build a React component instead of a static SVG file. Using our React component, let's animate the screws in our logo to rotate continuously and in varying speeds and directions.

Why a React Component?

You may be wondering why I have opted to create an entire React component for this svg instead of using the perfectly good. Indeed, we could just write some css and

it would work just fine. However, since the logo is visible in the nav, it will be visible in all places on the site. Many people do not like too many animations as they are distracting, so I will only be playing these animations on the homepage, as a bit of a tiny easter egg in our nav. We will get into how to dynamically control animations this later in the advanced implementation section of the book

First we'll utilize a tool to ensure our SVG markup is compatible with React. React will not recognize the various hyphenated properties of vanilla SVG markup (ex. `paint-order`), only understanding their camel case versions (ex. `paintOrder`). The **HTML to JSX Compiler** will take care of all of that for you, also converting other common attribute gotchas like `class` to `className`. In the case of the example SVG I am working with, there are not too many changes. but if your SVG is more complex, you may find the HTML to JSX Compiler to be very helpful.

Unfortunately, the HTML to JSX Compiler produces the old school `React.createClass()` style markup, so we won't be copying the entirety of the output, but that's fine, we can copy everything the contents of the `return()` statement. After copying that, create a new file under the `utils/` folder called `Logo.tsx`.

We'll need to migrate the `className`, `width`, and `height` properties to the SVG node in `Logo.tsx`. Then of course it is time to add animations. Create a new scss module under `src/styles/modules` called `logo.module.scss`. I decided to make a variety of animations, and applied them to each of the four screws. The contents of `logo.module.scss` looks like this:

Listing 3.41: </> logo.module.scss

```
.topLeftScrew,
.topRightScrew,
.bottomLeftScrew,
.bottomRightScrew {
  transform-origin: center;
  transform-box: fill-box;
}

.topLeftScrew {
  animation: turnClockwise 1s ease-in-out infinite;
}
.topRightScrew {
  animation: turnCounterClockwise 5s ease-in infinite;
}
.bottomLeftScrew {
  animation: turnHalfThenBack 3s ease-out infinite;
}
.bottomRightScrew {
  animation: turnClockwise 10s ease-in-out infinite;
}

@keyframes turnClockwise {
  0% {
    transform: rotateZ(0deg);
  }
  100% {
    transform: rotateZ(360deg);
  }
}

@keyframes turnCounterClockwise {
  0% {
    transform: rotateZ(0deg);
  }
  100% {
```

Listing 3.42: </> Logo.tsx

See what fun animations you can think up for your own logo!

With these styles complete, import what you had from the HTML to JSX Compiler, which results in the following to `Logo.tsx`:

```
@keyframes turnHalfThenBack {
  0% {
    transform: rotateZ(0deg);
  }
  50% {
    transform: rotateZ(180deg);
```

```
import * as React from 'react';
import * as styles from
'../../styles/modules/logo.module.scss'

export function Logo () {
  return (
    <svg xmlns="http://www.w3.org/2000/svg" viewBox="0
    0 161 161" width="30" height="30"
    className="d-inline-block align-top mx-3">
        <path fill="#bba4de" d="M35.5 0h90A35.5 35.5 0
        01161 35.5v90a35.5 35.5 0 01-35.5
        35.5h-90A35.5 35.5 0 010 125.5v-90A35.5 35.5 0
        0135.5 0z" />
        <path fill="#8468b2" d="M72.7 38.4a31.4 31.4 0
        01-31.4 31.4A31.4 31.4 0 0110 38.4 31.4 31.4 0
        0141.3 7a31.4 31.4 0 0131.4 31.4z" />
        <path className={styles.topLeftScrew}
        fill="#000000" d="M51.9 22.2L41.3 32.8 30.8
        22.2l-5.6 5.6 10.5 10.6L25.2 49l5.6 5.6L41.3
        44 52 54.6l5.6-5.6-10.6-10.6 10.6-10.6z" />
        <path fill="#8468b2" d="M151 38.4a31.4 31.4 0
        01-31.3 31.4 31.4 31.4 0 01-31.4-31.4A31.4
        31.4 0 01119.7 7 31.4 31.4 0 01151 38.4z" />
        <path className={styles.topRightScrew}
        fill="#000000" d="M130.2 22.2l-10.5 10.6L109
        22.2l-5.6 5.6 10.6 10.6L103.5 49l5.6 5.6L119.7
        44l10.5 10.6L136 49l-10.6-10.6 10.6-10.6z" />
        <path fill="#8468b2" d="M72.7 122.6A31.4 31.4
        0 0141.3 154 31.4 31.4 0 0110 122.6a31.4 31.4
        0 0131.3-31.3 31.4 31.4 0 0131.4 31.3z" />
        <path className={styles.bottomLeftScrew}
        fill="#000000" d="M51.9 106.4L41.3
        117l-10.5-10.6-5.6 5.7 10.5 10.5-10.5 10.6 5.6
        5.6 10.5-10.6 10.6 10.6 5.6-5.6-10.6-10.6
        10.6-10.5z" />
        <path fill="#8468b2" d="M151 122.6A31.4 31.4 0
        01-31.3 31.4 31.4 31.4 0 01-31.4-31.4A31.4
        31.4 0 01119.7 91 31.4 31.4 0 01151
        122.6z" />
        <path className={styles.bottomRightScrew}
        fill="#000000" d="M130.2 106.4L119.7 117 109
        106.4l-5.6 5.7 10.6 10.5-10.6 10.6 5.6 5.6
        10.6-10.6 10.5 10.6 5.7-5.6-10.6-10.6
        10.6-10.5z" />
    </svg>
  );
}
```

Within `Nav.tsx`, we'll now replace the `StaticImage` component with our component `Logo`.

Nice, same looking nav, new fun animations! Looking good. If you've followed all the steps so far, the homepage should now look like this:

Figure 3.13.: A screenshot of the homepage we've built so far.

Review of Initial Components and Layout

✅ Milestone Code #2 ✅

Nice! You've made it to the second milestone code repository, **`themed-frontend-with-components`**

So far so good. We've refactored a few file locations, namely creating a **`pages/`** and **`layout/`** folder to organize our **`components/`** folder a bit more. So far, the layout of your code base should look something like this:

Listing 3.43: </> terminal

```
.
├── LICENSE
├── README.md
├── gatsby-browser.js
├── gatsby-config.js
├── gatsby-node.js
├── gatsby-ssr.js
├── package-lock.json
├── package.json
└── src
    ├── components
    │   ├── Seo.tsx
    │   ├── layout
    │   │   ├── Footer.tsx
    │   │   ├── Layout.tsx
    │   │   └── Nav.tsx
    │   ├── pages
    │   │   ├── app
    │   │   │   ├── App.tsx
    │   │   │   └── home
    │   │   │       ├── EditorWidget.tsx
    │   │   │       ├── Header.tsx
    │   │   │       ├── Home.tsx
    │   │   │       └── TryItWidget.tsx
    │   ├── utils
    │   │   └── Logo.tsx
    │   └── widgets
    │       └── PlateWidget.tsx
    ├── images
    │   └── logo.svg
    ├── interfaces
    │   └── IEditorSettings.ts
    ├── pages
    │   ├── 404.tsx
    │   ├── app.tsx
    │   └── index.tsx
```

It's been pretty basic React so far, and all cosmetic oriented changes. Now we're going to get into some more complex functionality, involving helper functions and actual business functions that form the bedrock of ReduxPlate.

3.8 Recipe: Adding API Helper Functions

We will now add the first in a series of API Helper functions which I believe are as optimized as they can be. They are both as flexible as possible in terms of typings, but able to be called from any React component as a single function call, *including* side effect callbacks. This set of functions will be completely standalone, so you can reuse them for *any* SaaS product. The first we will be adding is a **post**

method to call our netlify function.

Listing 3.44: </> ApiHelpers.ts

```
import IApiConnectorParams from
"../../interfaces/IApiConnector"
import IApiError from "../../interfaces/IApiError"

export const post = async <T = undefined, U =
undefined>(
  success: (model: U) => void,
  failed: (model: IApiError) => void,
  body?: IApiConnectorParams & T
): Promise<void> => {
  try {
    const response = await
    fetch(`/.netlify/functions/api-connector`, {
      method: "POST",
      body: JSON.stringify(body),
    })
    const data = await response.json()
    if (response.ok) {
      return success(data)
    }
    return failed(data)
  } catch (error) {}
}
```

This is a lot to take in at first, but the complexity here will abstract away a lot of effort when we write code in our components. The generic types **T** and **U** respectively allow for a type signature of something like this:

Listing 3.45: </> pseudo code

```
post<InputType, OutputType>()
```

so it will be clear in our components calling it what the required input and expected output types are.

I've utilized two helper interfaces here. One is **IApiConnectorParams**, which is made in a union via the **\&** operator,with the generic type **T**. The **IApiConnectorParams** interface is defined as:

Listing 3.46: </> IApiConnectorParams.ts

```
export default interface IApiConnectorParams {
  endpoint: string;
}
```

This union type will enforce that we always send an endpoint parameter to the this endpoint. For now, like **IApiConnectorParams**, interface **IApiError** includes only a single parameter as well:

Listing 3.47: </> IApiConnectorParams.ts

```
import ApiErrorMessage from "../enums/ApiErrorMessage";

export default interface IApiErrorMessage {
    apiErrorMessage: ApiErrorMessage;
}
```

Where ApiErrorMEssage is an enum signifying what error the API has returned:

Listing 3.48: </> ApiErrorMessage.ts

```
enum ApiErrorMessage {
    GENERATOR_ERROR = 'GENERATOR_ERROR'
}

export default ApiErrorMessage
```

Finally, I define callbacks **onSuccess** and **onError**, so we can write complex side effect code without cluttering the API call itself.

For now, the **post** method is actually incomplete, as we have left the catch block empty. We'll get to that shortly after adding some messaging and toast functionality to our app. Let's look a little bit further into the enum **ApiErrorMessage**.

3.9 Recipe: Robust API Error Message Handling

We expect, in the case of a non-200 level API response, the API to return an **ApiErrorMessage**.

So far, **ApiErrorMessage** includes the single message key 'GENERATOR_ERROR', as a catch all, since we haven't written the logic behind our API endpoint yet. Note that while a solid start to a clean messaging system, this enum is only half the battle: we can't just show the string 'GENERATOR_ERROR' to the customer. We need to have a human readable message associated to each value in enum **ApiErrorMessage**. To do that, we'll create a message configuration to store these human readable messages. Create a new folder called **config/**. Then create the file **ApiErrorMessages.ts**, and add the following:

Listing 3.49: </> ApiErrorMessages.ts

```
import ApiErrorMessage from "../enums/ApiErrorMessage";

export const apiErrorMessages = {
    [ApiErrorMessage.GENERATOR_ERROR]: 'Error
    generating Redux code!',
}
```

This will work fine, but **apiErrorMessages** currently has no strict typing associated with it; to TypeScript, it's just an object. For now this may appear to be harmless, but as the application grows with a variety of API message types, we will need to make sure we don't incorrectly typo any key names, and more importantly, that we're not *forgetting to include any message IDs*. To accomplish these requirements, we will define a new type to associate to our configuration variable **apiErrorMessageConfig**. Create a new folder **types/**, and add this type:

Listing 3.50: </> ApiErrorMessageConfigEntries.ts

```
import ApiErrorMessage from "../enums/ApiErrorMessage";

export type ApiErrorMessageConfigEntries = {
    [key in ApiErrorMessage]: string
}
```

Now we can import that type and associate it with our config:

Listing 3.51: </> ApiErrorMessages.ts

```
import ApiErrorMessage from "../enums/ApiErrorMessage";
import { ApiErrorMessageConfigEntries } from
"../types/ApiErrorMessageConfigEntries";

export const apiErrorMessageConfig:
ApiErrorMessageConfigEntries = {
    [ApiErrorMessage.GENERATOR_ERROR]: 'Error
    generating code!',
}
```

⚠ Why is ApiErrorMessageConfigEntries a Type and Not an Interface? ⚠

We are unable to use the TypeScript **key in** syntax for interfaces. Since we explicitly want our keys to be the keys of **ApiErrorMessage** enum, we must use a type. Not only will this type help us select the values from the **ApiErrorMessage** enum, it will also remind us when a value is missing, since we want *each* key **in** the *ApiErrorMessage* enum!

Why All the Trouble?

This seems like a lot of trouble to go through just for some simple messaging functionality. However, this method forces us to collect all message strings into a single file, making things like translations and localization much easier later on. It also makes any code which uses these messages cleaner. We won't have any hardcoded message strings anywhere else in our app except for our config files. We can also later easily extend these types to include perhaps a

slice of the app, or endpoint that certain messages are associated with, to keep the configs even shorter and more maintainable.

3.10 Setting Up a Contract-Based API Call

As we saw in our `post` API helper function, the calls to our API accept a generic type and return a generic type. In this section, we'll be setting up an interface that matches what we will build in our .NET API, so that on both the frontend and backend, we can expect what shapes of data will be receiving and sending.

Later, we will see on the `/app` page of ReduxPlate that features available to non-subscribed customers or visitors to the site will be very limited. In any case, as we build out features for paying customers, it's clear that the `/CodeGenerator` endpoint will eventually have a complex set of options that our API and client will need to robustly follow and understand. The best way to do this is define the two interfaces that the `post` function expects - one for the shape of the POST body, and one for the expected shape of the JSON data returned (in the case of a successful call, otherwise we expect the shape of `IApiError` as previously defined). This approach is powerful and will save us time later: as we build complexity and features to our SaaS product, we can always return to these two contracts and extend them as needed. Even better, we can repeat this contract pattern for each new endpoint that we may need!

For starting off this pair contracts, let's first think about what we need to *send* to the API. Since this is the `/CodeGenerator` endpoint, I call the POST body contract `IGenerateOptions`, and the return contract `IGenerated`.

Let's create the POST body contract first.

Creating the IGenerateOptions Interface

Let's create a new file, **IGenerateOptions.ts** under the **src/interfaces/** folder. Our initial **IGenerateOptions** interface will look like this:

Listing 3.52: </> IGenerateOptions.ts

```
import ITypeScriptProperty from
"./ITypeScriptProperty";

export default interface IGenerateOptions {
  data: {
    stateCode: string
  }
}
```

Why do I wrap the **stateCode** property in an object with key **data**? Recall that this interface is made in a union with the **IApiConnectorParams**, which holds the **endpoint** parameter for our .NET API. Wrapping the actual parameters for our API this way will make the call in our serverless function cleaner, as we'll see shortly - we'll end up forwarding the entire contents of the **data** object into the **body** of the .NET call.

For now, this **stateCode** string should be all we need (for the free and public version of the generator!) to generate the code for a Redux boilerplate codebase. Because generating and doctoring Redux code is the whole point of our SaaS product, the generation step must be done on our server.

It is now time to define the expected type that should be returned from our API. We expect the generator end-

point to return nothing more than an array of files. Each file should have a label, and the code contents of that file. In fact, this looks like something which we already have in our application: the first two properties of interface IEditorSettings: **fileLabel** and **code**! let us redefine the existing **IEditorSettings** interface to accommodate this new contract. Create a new interface under **interfaces/** called **IFile.ts**, and move both **fileLabel** and **code** from **IEditorSetting.ts** to **IFile.ts**:

Listing 3.53: </> IFile.ts

```
export default interface IFile {
  name: string
  type: string
}
```

We can then have **IEditorSetting.ts** extend **IFile**, which looks like this now:

Listing 3.54: </> IEditorSetting.ts

```
import IFile from "./IFile";

export default interface IEditorSetting extends IFile {
  isActive: boolean
}
```

Creating the IGenerated Interface

We are now ready to create the return type interface for the **/CodeGenerator** endpoint. Under **interfaces/**, create a new file **IGenerated.ts**:

Listing 3.55: </> IEditorSetting.ts

```
import IFile from "./IFile";

export default interface IEditorSetting extends IFile {
  isActive: boolean
}
```

With the API contracts (interfaces) done, we can *finally* craft the call to **post** in **TryItButton.tsx**. We can hard code the most of settings for **IGenerateOptions**, since this button is for the free and public version of the endpoint, and the user won't have access to any of the more advanced options.

Listing 3.56: </> TryItButton.ts

```
// TODO: uh oh - how to get at the current value of
variable 'code' here? That's buried in an adjacent
component!
const onClickGenerate = async () => {
  await post<IGenerateOptions, IGenerated>(
    generated => {
      dispatch(
        codeGenerated({
          editorID: EditorID.TRY_IT_RESULTS,
          files: generated.files,
        })
      )
    },
    apiError => {
      console.log(apiError)
    },
    {
      endpoint: "/CodeGenerator",
      data: {
        stateCode: code // Uh oh! 'code' is not
        defined :(
      },
    }
  )
}
```

We've run into a new issue. We don't immediately have access to the string value of what code is in our editor! Getting at it in a traditional React way would involve a series of parent-to-child callbacks, and then back again, which is not readable or maintainable. It's time we introduce Redux into the app, and build a slice of state specifically for storing the state of the various code editors that will exist around the app.

3.11 Adding Redux

We saw in the previous section that when we went to add the call to our `onClickGenerate` function we didn't have immediate access to the current value of what the code was in the editor. To do this in a clean and maintainable way, we will add Redux to the frontend. (I know, I know, using Redux in a developer tool built *for* Redux is a bit meta, but it won't be too bad to understand 😉).

Getting Started

Before writing code, let's install all the packages we will need to use Redux. We'll need `redux` itself, `react-redux` for a variety of React hooks to use Redux in our components, and finally `\at reduxjs/toolkit` for Redux toolkit, which helps making slices of state a breeze. All together, this install with `npm` is:

Listing 3.57: </> terminal

```
npm install redux react-redux @reduxjs/toolkit
```

Add Redux Scaffolding to Make Redux Compatible with Gatsby

There is a **nice example on GitHub on how to use Redux in a Gatsby app**. We will be following the same pattern here, but with a few additions to support `@reduxjs/toolkit`. First start by creating a **JavaScript** file in the project root called `wrap-with-provider.js`, and add the following:

Listing 3.58: </> wrap-with-provider.jsx

```
import React from "react"
import { Provider } from "react-redux"
import createStore from "./src/store/index"

export default ({ element }) => {
  const store = createStore()
  return <Provider store={store}>{element}</Provider>
}
```

Then import it in **gatsby-ssr.js**:

Listing 3.59: </> gatsby-ssr.js

```
import wrapWithProvider from './wrap-with-provider'

export const wrapRootElement = wrapWithProvider
```

Also add these two lines to **gatsby-browser.js**, such that it results with:

Listing 3.60: </> gatsby-browser.js

```
import "./src/styles/styles.scss"
import wrapWithProvider from './wrap-with-provider'

export const wrapRootElement = wrapWithProvider
```

createStore is defined in the **index.ts** of our store, under **src/store/**. This file contains the function **createStore**, as well as a few helper types that are recommended by Redux Toolkit's official Typescript Quick Start documentationhttps://redux-

toolkit.js.org/tutorials/typescript:

Listing 3.61: </> index.ts

```
import { configureStore } from "@reduxjs/toolkit"
import editorsReducer from './editors/editorsSlice'

const createStore = () => configureStore({
  reducer: {
    editors: editorsReducer,
  },
})

type ConfiguredStore = ReturnType<typeof createStore>;
type StoreGetState = ConfiguredStore["getState"];
export type RootState = ReturnType<StoreGetState>;
export type AppDispatch = ConfiguredStore["dispatch"];
export default createStore
```

While **createStore** is used in our **wrap-with-provider.js**, we can also employ the two exported types **RootState** and **AppDispatch**. Create a new folder under **src/** called **hooks/** and add a new file called **redux-hooks.ts**:

Listing 3.62: </> redux-hooks.ts

```
import { TypedUseSelectorHook, useDispatch,
useSelector } from 'react-redux'
import { AppDispatch, RootState } from '../store'

export const useAppDispatch = () =>
useDispatch<AppDispatch>()
export const useAppSelector:
TypedUseSelectorHook<RootState> = useSelector
```

This another recommended pattern also derived from the

TypeScript Quick Start documentation from Redux Toolkit. We can use these typed Redux hooks throughout our application, instead of the standard `useSelector` and `useDispatch` Redux hooks. Then, no matter how many slices of state we add, we can expect them to be typed properly whenever we call these hooks across our app.

The hooks Folder

I typically add all custom hooks into a `hooks` folder. While the typed Redux hooks are the first hooks we have seen so far, we will be revisiting the `hooks/` folder many times throughout this book.

Adding a Slice of State for the Editors

Add a new folder called `store/` under the `src/` folder. Then, create a folder for our first slice called `editors`. Finally, create a file called `editorsSlice`. Add the following to it:

Listing 3.63: </> editorsSlice.ts

```
import { createSlice, PayloadAction } from
"@reduxjs/toolkit"
import Editor from "../../enums/Editor"
import IEditorSetting from
```

I've moved most of the **editorSettings** update logic into the reducer action logic - but note we're still leveraging our utility function, **updateArray**, within the redux-toolkit form of the reducer. I also renamed the names **onChangeCode** and **onChangeTab** actions to **codeEdited** and **tabClicked**, respectively, as **Redux recommends making actions have the most meaningful names as possible**, and avoid generic names. We can also remove the state variable from **EditorWidget**. The initial state and props can also be eliminated from **EditorWidget** component. These changes ultimately results in the **EditorWidget** and **TryItWidget** being much cleaner and easier to read. Most of the complexity was really in the initial props of the editors, now refactored to be in the Redux initial state.

After refactoring, the **TryItWidget** component now looks like this:

Listing 3.64: </> TryItWidget.tsx

```
import * as React from "react"
import Editor from "../../../enums/Editor"
import { EditorWidget } from "./EditorWidget"
import { TryItButtons } from "./TryItButtons"

export function TryItWidget() {
  return (
    <div className="container text-center">
      <div className="d-flex flex-wrap
      justify-content-center">
        <EditorWidget editor={Editor.TRY_IT_STATE} />
        <EditorWidget editor={Editor.TRY_IT_RESULTS} />
      </div>
      <TryItButtons />
    </div>
  )
}
```

likewise, the **EditorWidget** component has also become much cleaner:

Listing 3.65: </> EditorWidget.tsx

```
import * as React from "react"
import AceEditor from "react-ace"
import "ace-builds/src-noconflict/mode-typescript"
```

3.12 Completing the API Call Setup

```
import { useEffect, useRef } from "react"
import * as styles from
"../../../styles/modules/editor.module.scss"
import Editor from "../../../enums/Editor"
import { codeChanged, tabClicked } from
"../../../store/features/editorWidget"
import { useAppDispatch, useAppSelector } from
"../../../hooks/redux-hooks"

export interface IEditorWidgetProps {
  editor: Editor
}
```

Returning to **TryItButtons.tsx**, we can finally complete the call by adding that missing **code** variable. Adding a **useSelector** hook, we can get at the current value of the editor's code within the **EditorWidget** component:

Listing 3.66: </> TryItWidget.tsx

```
const code = useAppSelector(
  state => state.editors.editors[Editor.TRY_IT_STATE].
  editorSettings[0].code
)
```

We can then submit the processed code (thanks to the **parseTypeScript** function we wrote), along with the hard-coded values, to the **post** function from ApiHelpers!

3.13 Recipe: Toast Helper Functions

Following the inclusion of our **ApiHelpers** functions, we saw the need for side effects to inform the customer. A typical pattern is to include an animated feedback that appears on screen. It is up to you to have these appear directly on the site, for example, right near the button after it is clicked, or in a floating div somewhere else on the page. I typically use the floating pattern of. Just as we did with an API connector, we'll create a file **ToastHelpers** inside the **helpers/** directory with a variety of helper functions which can create these toasts.

Getting Started

First we need to install the **react-toastify** library via **npm**:

Listing 3.67: </> terminal

```
npm install react-toastify
```

We should also immediately include the default styles for the library in **gatsby-rowser.js**:

Listing 3.68: </> gatsby-browser.js

```
require('react-toastify/dist/ReactToastify.css')
```

We also need to include the required **ToastContainer** component. We can add that to our **Layout** compenent, so toasts will be available to show on any page we make with Gatsby:

Listing 3.69: </> Layout.tsx

```
...
return (
  <>
    ...
    <ToastContainer />
    ...
  </>
)
```

Add a new file ToastHelpers.tsx to the **helpers/** folder, and add this:

Listing 3.70: </> ToastHelpers.ts

```
import {
  toast,
  ToastPosition,
} from "react-toastify"

export const showSimple = (
  message: string,
  position: ToastPosition = "top-center"
): void => {
  toast(message, { position })
}
```

showSimpleToast function is nothing more than a small wrapper which accept a string as the toast's message, and an optional position defaulting to the top-center of the page. This helper function helps us cleanly call up a toast wherever we may need it in our app. We can now add a few calls to **showSimpleToast** by replacing the two message placeholder **console.log** calls in **TryItButtons.tsx**:

Listing 3.71: </> TryItButtons.tsx

```
const generate = async (typeScriptProperties:
Array<ITypeScriptProperty>) => {
  await post<IGenerateOptions, IGenerated>(
    generated => {
      dispatch(
        codeGenerated({
          editorID: EditorID.TRY_IT_RESULTS,
          files: generated.files
        })
      )
    },
    apiError => {
      // added
      if (
        Object.values(apiErrorMessageConfig).includes(
          apiError.apiErrorMessage
        )
      ) {
        showSimpleToast(apiErrorMessageConfig[apiError
        .apiErrorMessage])
      } else {
        showSimpleToast(apiErrorMessageConfig[ApiError
        Message.UNKNOWN_ERROR])
      }
    },
    {
      endpoint: "/CodeGenerator",
      data: {
        typeScriptProperties,
        useReduxToolkit: false,
        useTypeScript: true,
        singleFile: false,
      }
    }
  )
}
```

Here we leverage all the scaffolding effort we made for our messaging system - using only the enum key values to reference what message we want to show. In the case of **onError** callback from **post**, this is an **apiErrorMessage**, as it will originate from the API.

3.14 Styling Toasts to Match the Application Styles

If we were to see one of these toasts right now, we would see the timing bar takes on a rainbow gradient. This is neat, but a little too flashy for our application. Let's style the toasts so that the time indicator bar takes on the Redux purple color we've already been using throughout our app. Create a new Sass partial file **_toasts.scss** under the **styles/** folder, and add this single rule:

Listing 3.72: </> _toasts.scss

```
.Toastify__progress-bar--default {
    background: $primary !important;
}
```

don't forget to include this partial into the global styles file, **styles.scss**:

Listing 3.73: </> styles.scss

```
@import "variables";
@import "../../node_modules/bootstrap/scss/bootstrap";
+ @import "toasts.scss";
```

3.15 New Action to Set Code Returned by API

There's one placeholder `console.log` still left in `TryItButtons` component. We need to actually set the generated code into our editors, not just log it to the console!

Add a New Action to Editors Slice of State

We will get started by define a new action to actually set the code in the editor once it is returned by the API. Go into `editorsSlice.ts` and add the following action, `codeGenerated`:

Listing 3.74: </> editorsSlice.ts

```
codeGenerated: (
  state,
  action: PayloadAction<{ editor: Editor; files:
  Array<IFile> }>
) => {
  const { editor, files } = action.payload
  state.editors[editor].editorSettings =
  files.map(file => {
    const existingFile =
    state.editors[editor].editorSettings.find(
      editorSetting => editorSetting.fileLabel ===
      file.fileLabel
    )
    return {
      ...file,
      isActive: existingFile ? existingFile.isActive :
      false,
    }
  })
}
...
export const { codeEdited, tabClicked, codeGenerated }
= editorsSlice.actions
```

Here, we merge the returned files with the existing **isActive** property for all the files. This will update the code in each of the tabs without resulting in unexpected changes in which tab is open. In the unexpected case where we can't find the previous **isActive**, we set **isActive** to false. Also don't forget to add **codeGenerated** to the actions export!

Add Event to TryItButtons

We can now call **dispatch** on our **codeGenerated** action in **TryItButtons**:

Listing 3.75: </> TryItButtons.tsx

```
generated => {
  dispatch(
    codeGenerated({
      editor: Editor.TRY_IT_RESULTS,
      files: generated.files
    })
  )
}
```

We can use `Editor.TRY\_IT\_RESULTS` explicitely here, since we expect this TryItButtons to only be used with this editor.

Rejoice! That should *finally* just about do it for it for `TryItButtons`. Whew.

3.16 Add Netlify Functions with TypeScript

In the last section, we saw that our API call to `/CodeGenerator` is ready to go. But there's a small problem right now: the Netlify function at the URL `./netlify/functions/api-connector` that we are trying to call doesn't exist yet! In this section, we'll build our first Netlify function, complete with TypeScript builds so we can write our serverless functions to use TypeScript as well.

Getting Started

To get started, we'll first need in a `functions` folder. Go ahead and make one right in the root of your Gatsby project. Next, we should make a separate `package.json` within that folder. To start that process of, issue `npm init` in the root

of the **functions** folder:

Listing 3.76: </> terminal

```
cd functions/
npm init
```

Go through the prompts and fill them out as you best see fit. For example, my responses led to the following initial **package.json**:

Listing 3.77: </> package.json

```
{
  "name": "reduxplate-functions",
  "version": "1.0.0",
  "description": "Netlify serverless functions for
  ReduxPlate",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit
    1"
  },
  "repository": {
    "type": "git",
    "url": "git+https://princefishthrower@bitbucket.or
    g/princefishthrower/reduxplate.com.git"
  },
  "keywords": [
    "developer",
    "tool",
    "redux",
    "redux-toolkit",
    "react-redux",
    "saas",
    "product"
  ],
  "author": "Chris Frewin",
  "license": "MIT",
  "homepage": "https://bitbucket.org/princefishthrower
  /reduxplate.com#readme"
}
```

Define tsconfig.json for the Serverless Functions

Create a **tsconfig.json** file in the root of the **functions/** folder, and put this in it:

Listing 3.78: </> tsconfig.json

```
{
  "compilerOptions": {
    "strict": true,
    "isolatedModules": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "removeComments": false,
    "preserveConstEnums": true,
    "resolveJsonModule": true,
    "outDir": "./dist"
  },
  "include": ["./src/**/*"]
}
```

In this **tsconfig.json** file, we can see that all files in the subfolder **src/** will be compiled to **dist/**.

Define a build command

Within our newly created **package.json**, remove the "test" script (don't worry, we will be adding tests later), and add a new "build" script, which is just "tsc". The TypeScript compiler will take care of the rest, following the rules we defined in our **tsconfig.json**.

Listing 3.79: </> package.json

```
"scripts": {
-  "test": "echo \"Error: no test specified\" && exit
1"
+  "build": "tsc"
}
```

With the addition of functions and using TypeScript to

build them, our build command has now grown too complex to maintain using the Netlify UI. Luckily, Netlify offers us a way to maintain variables like the build command and functions path using code, and that is with a **netlify.toml** file. Create a **netlify.toml** in the project root (*not* in the functions folder). The **netlify.toml** file should include the following:

Listing 3.80: </> netlify.toml

```
[build]
  command = "cd functions && npm install && npm run
  build && cd .. && npm install && npm run build"
  publish = "public"
  functions = "functions/dist"

[dev]
  command = "npm run develop"
  functions = "functions/dist"
```

The build command may look rather scary at first, but it is simply telling Netlify to:

1. Move into the functions folder
2. Install dependencies (using whatever is defined with the new package.json there)
3. Run the build process (**tsc** as we defined)
4. Move back to the root
5. Install dependencies for the Gatsby project
6. Build the Gatsby project

We will be returning to **netlify.toml** later as our app grows in complexity.

Creating Our First Serverless Function

First, within the **functions/** folder, create the **src/** folder where we will store all our source TypeScript functions.

Note that the name of the functions must match the /.netlify/functions/api-connector exactly. Thus our function file should be **api-connector.ts**. Before writing code in **api-connector.ts**, we need to install a type library as recommended by linkthe official Netlify documentation on building serverless functions with TypeScripthttps://docs.netlify.com/functions/build-with-typescript/:

Listing 3.81: </> terminal

```
npm install @netlify/functions
```

We'll also be using the **node-fetch** package, which brings the **fetch** api to Node.js:

Listing 3.82: </> terminal

```
npm install node-fetch
```

Now we can write the connector function:

Listing 3.83: </> api-connector.ts

```
import { Handler } from "@netlify/functions"
import { Event } from
"@netlify/functions/src/function/event"
import fetch from 'node-fetch'

const handler: Handler = async (event: Event) => {
  if (event.body === null) {
    return {
      statusCode: 400,
      body: JSON.stringify({ apiErrorMessage:
      "UNSPECIFIED_BODY" }),
    }
  }
  const { endpoint, data } = JSON.parse(event.body)

  try {
    const response = await fetch(
      `${process.env.REDUX_PLATE_API_URL}${endpoint}`,
      {
        method: "POST",
        headers: {
          'Content-Type': 'application/json'
        },
        body: JSON.stringify(data),
      }
    )
    const json = await response.json()
    if (response.ok) {
      return {
        statusCode: 200,
        body: JSON.stringify(json),
      }
    }
    return {
      statusCode: response.status,
      body: JSON.stringify(json),
    }
  } catch (error) {
    // unknown error occurred, return 500 with
    UNKNOWN_ERROR message
    return {
      statusCode: 500,
      body: JSON.stringif piErrorMessage:
      error.message }),
    }
  }
```

As long as we continue to define our client-side API functions in a contract-based way as we saw in 3.10, this `api-connector` serverless function will work for all calls we need to make to our .NET API. We always expect both a `endpoint` and `data` parameter in the calls to, and likewise, we expect that our .NET API returns JSON of the specified type, or in the case of error, with a the `ApiErrorMessage` type. (In the case of errors at the Netlify serverless layer, we provide this `apiErrorMessage` explicitly.)

Create the first build

We need to compile this function now to it's JavaScript version, since we defined in **netlify.toml** the functions path to be **functions/dist**. **Note that you should do this every time after modifying any of your serverless functions.**

Defining the Environment Variable REDUX_PLATE_API_URL

You may have noticed in **api-connector.ts** the usage of an environment variable **REDUX_PLATE_API_URL** - this has to be an environment variable so we can set it to various environments as we test our serverless functions - whether we are in the development, staging, or production environments. For our local development environment, the .NET API will be available at both an HTTP and HTTPS endpoint, **http://localhost:5000** and **https://localhost:5001** respectively, by default. Because netlify will complain about the endpoint at **https://localhost:5001** because of it's self signed certificate, we must use the **http://localhost:5000** endpoint. For those of us on UNIX or UNIX-like systems, that is as easy as adding the following to your shell's profile file. I use **zsh** as my shell, and so I can add teh following to my **.zprofile**:

Listing 3.84: </> .zprofile

```
export REDUX_PLATE_API_URL='http://localhost:5000'
```

Remember to source your profile or save the changes, and restart the development process by stopping the **ntl** process and reissuing it with **ntl dev**.

This is enough for now, but will be adding the `REDUX\_PLATE\_API\_URL` variable to the netlify UI later in the book when we connect our production API.

Chapter Review

For our serverless functions we've:

- Installed the @netlify/functions type library
- Built a robust api-connector function

3.17 Building an App Page

We've nearly gotten to an MVP stage with our client. There is one small aspect that we should finish before diving in to the backend to build the `/CodeGenerator` endpoint, and that is to ensure an actual page exists when we click the 'Try Full App' button on our homepage. We've coded it in using a Gatsby `Link` component to `/app`, but that page actually doesn't exist. We should at least fill it out with a basic 'Coming Soon' banner for our MVP, which is much better to show our potential customers than an unsightly 404 page.

Utilizing Gatsby to Build Static Pages

So far, you may be questioning why I chose to use the Gatsby framework for building the client. We haven't really used any of its features yet, aside from a few Gatsby plugins, GraphQL imports and a `StaticImage` component, which we anyway removed, opting for our fancy logo SVG. In this section, we're finally going to use a powerful feature of Gatsby to build a new static page under the path `/app`. The Gatsby core automatically turns any React component found in `src/pages` into it's correspondingly named static page, as stated **by the official Gatsby docs**.

Getting Started

We'll get started by creating an **app.tsx** file under the **src/pages/** folder. Note that this is very different from what you might see in **create-react-app**, where an **App.tsx** is the root component of a single page application. This *lowercase* **app.tsx** will be a page created at redux-plate.com/app.

Following the pattern from the **index.tsx** page, we will keep the **app.tsx** file as minimal as possible, adding only the **<Seo/>** component, and abstracting the actual content of the page into the components folder, where we will make another folder under **src/components/pages/** called **app/**. Within this folder create a capitalized **App.tsx**. For now, we can leave just a simple placeholder render:

Listing 3.85: </> App.tsx

```
import * as React from "react"

export function App() {
  return <h1>Full App Coming Soon</h1>
}
```

If this name **App.tsx** confuses you too much with frameworks like **create-react-app**, feel free to call this page and component **dashboard** or something similar. In the end, the *lowercase* page component **app.tsx** should look like this:

Listing 3.86: </> app.tsx

```
import * as React from "react"
import Layout from "../components/layout/Layout"
import { App } from "../components/pages/app/App"
import Seo from "../components/Seo"

const AppPage = () => (
  <Layout>
    <Seo title="ReduxPlate - App"/>
    <App/>
  </Layout>
)

export default AppPage
```

Now when we click the 'Try Full App' button on our homepage, we'll get an nearly empty (but at least existing) app page, just with our 'Coming Soon' message, instead of the Gatsby development 404 page.

3.18 Add a Mailchimp Signup Form to the App Page

As a nice capstone to the reaching the end of our frontend MVP, we're going to add a Mailchimp signup form on the page at **/app** we just built. This will be a minimalistic way to collect interest in the product.

⚠ Traction and Sign Ups ⚠

I'm a very large proponent of releasing an MVP as soon as possible, and getting feedback or ideas on it as soon as possible. When starting a new SaaS product, you'll never know *exactly* what niche your end product will fill, or be able to predict what customers will say about your application, or what ideas they will have and what direction it will fall into. One thing is universal however: **if after**

releasing your MVP to the world, you don't see any traction or interest in your product's MVP, it's time to add the SaaS product to your portfolio and move on to the next one. This can hurt - beleive me, I've had to do it more than once. However, if this is the unfortunate case, think about the totality of what was really lost: with what we have done so far in the book, and the steps we'll need to make a few API endpoints (in the next section of the book), I would geuess an advanced developer would only spend a total of 6-8 hours, or about one working day, building out a full stack MVP. (To be fair, the ideation and conceptual scaffolding could take much longer, this I concede.) In any case, it's yet another full stack iteration - and certainly you learn something new each time. So don't regret a flopped launch - rinse and repeat!

Create an Account or Sign In to Mailchimp

Mailchimp has a free level, which provides up to 2000 contacts - plenty for MVP stage of our product. Once you've signed in or created an account, click the 'Create' button on the sidebar and then scroll to the 'Signup Form' option. The default 'Embedded form' option is fine for us.

Within the resulting page, within the 'Copy/paste ontor your site section', take note of both the `form action` parameter and the value of the `name` parameter in the `input` with type `text` - we'll need both those values in the next section when we build a React for the sign up form. It may be easier to find these two values in the form's source code by choosing the the 'Unstyled' option tab, and if you uncheck all extraneous fields:

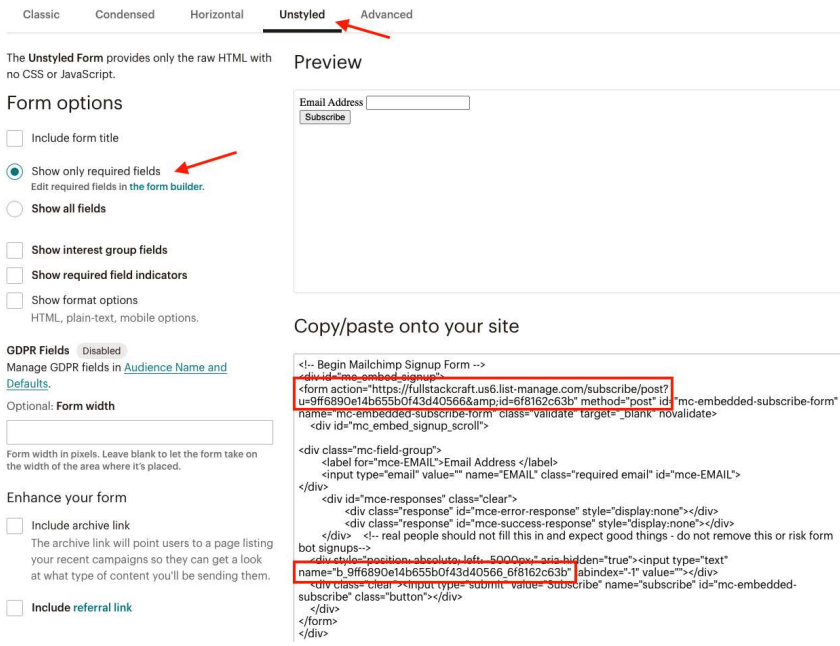


Figure 3.14.: Copying Mailchimp's form action URL and validation name value.

Returning to the Embedded Form Code

I find Mailchimp's UI to be a bit confusing to say the least. If you need find these two values at a later time, you'll need to look into Mailchimp's embedded forms. On the Mailchimp dashboard, on the sidebar, click the audience icon, then the 'Signup forms' tab, and in the resulting page, the 'Embedded forms' row:

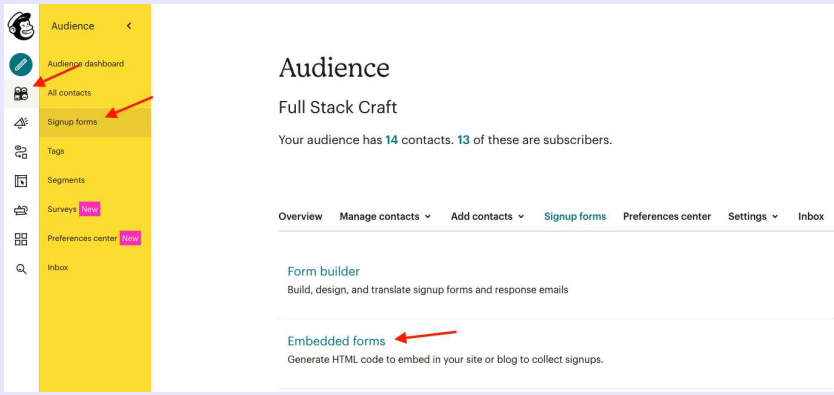


Figure 3.15.: Navigation to Mailchimp's embedded forms.

Building a Signup Component

Under the `components/utils/` folder, create a new file `SignUpWidget.tsx`. I've made things easy, and provide the following component which only needs to accept a `formActionURL` and `formValidationValue` props:

Listing 3.87: </> SignUpWidget.tsx

```
import * as React from "react"

export interface ISignUpWidgetProps {
  formActionURL: string
  formValidationValue: string
}

export function SignUpWidget(props:
ISignUpWidgetProps) {
  const { formActionURL, formValidationValue } = props

  return (
    <form action={formActionURL} method="post">
      <div className="row g-3 align-items-center">
        <div className="col-auto">
          <input
            type="email"
            name="EMAIL"
            placeholder="dev@company.com"
            className="form-control"
          />
        </div>
        <div
          style={{ position: "absolute", left:
          "-5000px" }}
          aria-hidden="true"
        >
          <input type="text"
          name={formValidationValue} tabIndex={-1} />
        </div>
        <div className="col-auto">
          <input
            type="submit"
            value="Get Notified"
            name="subscribe"
            className="btn btn-primary"
          />
        </div>
      </div>
    </form>
  )
}
```

Extending the App Page

In the app page component, I added a few `<p/>` tags and the `SignUpWidget`, so that it now looks like this:

Listing 3.88: </> app.tsx

```
import * as React from "react"
import { SignUpWidget } from "../../utils/SignUpWidget"

export function App() {
  return (
    <>
      <div className="d-flex flex-column
      align-items-center m-5">
        <h1>Full App Coming Soon</h1>
        <p>
          Are you a developer or company that has been
          waiting for a service
          like ReduxPlate?
        </p>
        <p className="m-0">
          Sign up to be the first to know when the
          full product is released!
        </p>
        <SignUpWidget
          formActionURL="https://fullstackcraft.us6.li
          st-manage.com/subscribe/post?u=9ff6890e14b65
          5b0f43d40566&id=6f8162c63b"
          formValidationValue="b_9ff6890e14b655b0f43d4
          0566_6f8162c63b"
        />
        <p className="fw-bold">
          I take email spam seriously and will only
          email you once when ReduxPlate is released.
        </p>
      </div>
    </>
  )
}
```

Be sure to provide your own Mailchimp **formActionURL**

and `formValidationValue` from the previous section!

3.19 Define a Custom Subscription Success Redirect URL

The default action for the Mailchimp post URL is to redirect to a Mailchimp-owned URL as a confirmation page. I find this unnecessary. Luckily, Mailchimp offers us a way to redirect on subscription success, instead of showing their boilerplate 'thank you' page. To do this, head in to Mailchimp, and on the sidebar, click the audience icon, then the 'Signup forms' tab, and in the resulting page, the 'Form builder' row:

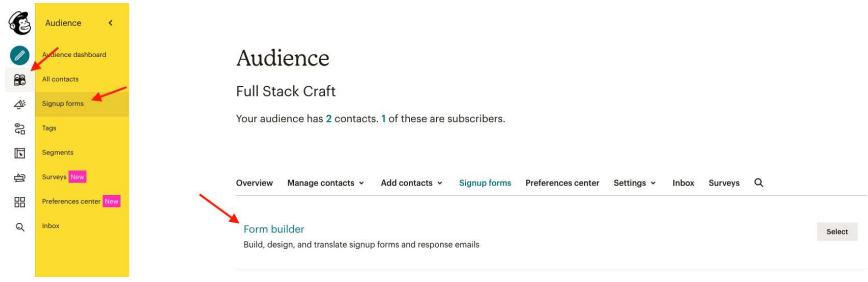


Figure 3.16.: Navigation to Mailchimp's form builder.

In the resulting page, you should see a dropdown near the top with a wide variety of options as to what form you would like to edit. We want the 'Confirmation thank you page':

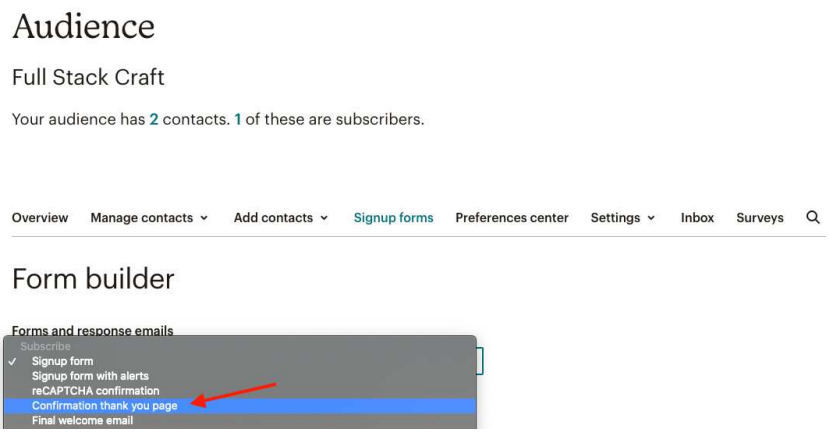


Figure 3.17.: Selecting 'Confirmation thank you page' from the dropdown.

Under the 'Build it' tab, we finally find the URL field that we can maintain. In my case, I would like the page to be the same one as site as our signup form, **`https://reduxplate.com/app`**, but with a URL parameter **`from`**, resulting in a full URL of **`https://reduxplate.com/app?from=mailchimp-subscription-succ`**

In the next section, we'll learn how to robustly handle these URL parameters, and our frontend MVP will be complete!

3.20 Recipe: Handling URL Search Parameters Robustly

To get started, let's define a pair of enumerations, which will define our allowable URL search parameter keys and values. For the keys, create a new enum under the **`enums/`** folder, called **`URLSearchParamKey.ts`**

Listing 3.89: </> URLSearchParamKey.ts

```
enum URLSearchParamKey {
  FROM = 'from'
}

export default URLSearchParamKey
```

Listing 3.90: </> URLSearchParamValue.ts

```
enum URLSearchParamValue {
  MAILCHIMP_SUBSCRIPTION_SUCCESSFUL =
  'mailchimp-subscription-successful',
}

export default URLSearchParamValue
```

We'll then need a config to determine what to do in the case that a proper value is found. As we saw with **ApiErrorMessageConfig** and it's **ApiErrorMessageConfigEntries**, this config will have a type which is an array of "Entry" types. I called this **Entry** type **SearchParamConfigEntry**:

Listing 3.91: </> SearchParamConfigEntry.ts

```
import URLSearchParamKey from
"../enums/URLSearchParamKey";
import URLSearchParamValue from
"../enums/URLSearchParamValue";

export type SearchParamConfigEntry = {
    key: URLSearchParamKey,
    value: URLSearchParamValue,
    action: () => void
}
```

Semantics Are Important!

Why is the type for the API error messages, **ApiErrorMessageConfigEntries** suffixed with 'Entries', but type for the search param configuration, **SearchParamConfigEntry** suffixed with the singular 'Entry'? By the nature of their structure, the exported **apiErrorMessageConfig** is a simple key-value object, and therefore it's type represents all entries, which is plural. As for the nature of the exported **searchParamConfig**, it really is an array of entries, and thus the typing for each entry is just that - a single entry, non-plural.

We can now maintain the values of the search parameter config itself. Create a new file **SearchParamConfig.ts** under the **config/** folder, and add the following:

Listing 3.92: </> SearchParamConfig.ts

```
import AppMessage from "../enums/AppMessage"
import URLSearchParamKeys from
"../enums/URLSearchParamKey"
import URLSearchParamValues from
"../enums/URLSearchParamValue"
import { showSimpleToast } from
"../helpers/ToastHelpers"
import { SearchParamConfigEntry } from
"../types/SearchParamConfigEntry"
import { appMessageConfig } from "./AppMessageConfig"

export const searchParamConfig:
Array<SearchParamConfigEntry> = [
  {
    key: URLSearchParamKeys.FROM,
    value: URLSearchParamValues.MAILCHIMP_SUBSCRIPTION
    _SUCCESSFUL,
    action: () =>
      showSimpleToast(
        appMessageConfig[AppMessage.MAILCHIMP_SUBSCRIP
        TION_SUCCESSFUL]
      ),
  },
]
```

Again, a lot of work, but these types and patterns will save us so much time later when it comes to internationalizing our app.

Adding App Messages to the Client

To prevent hardcoding of message strings, just as with the API error messages, I've done the same now for the frontend - as seen by my reference to `appMessageConfig[AppMessage.MAILCHIMP\_SUBSCRIPTION\_SUCCE` in `SearchParamConfig.ts`

To do this you'll need to add three new files, just as we did

for the API Error messages: an enum, a config type, and the configuration itself. Since we've walked through this process already with the API error messages, I will simply provide the source files here in quick succession.

First the enum to hold the message keys for the app messages, **AppMessage**:

Listing 3.93: </> AppMessage.ts

```
enum AppMessage {
  MAILCHIMP_SUBSCRIPTION_SUCCESSFUL =
  'MAILCHIMP_SUBSCRIPTION_SUCCESSFUL',
}

export default AppMessage
```

Then the config entry type, **AppMessageConfigEntry**, which uses the **key in** keywords from TypeScript so that no messages are forgotten to be maintained:

Listing 3.94: </> AppMessageConfigEntry.ts

```
import AppMessage from "../enums/AppMessage";

export type AppMessageConfig = {
    [key in AppMessage]: string
}
```

And finally, the config itself, **AppMessageConfig**:

Listing 3.95: </> AppMessageConfig.ts

```
enum AppMessage {
  MAILCHIMP_SUBSCRIPTION_SUCCESSFUL =
  'MAILCHIMP_SUBSCRIPTION_SUCCESSFUL',
}

export default AppMessage
```

This configuration should abstract a large amount of the complexity away from our code. We can now try and write a function that will check all values in our **searchParamConfig**, and call the **action** function if a key / value match is found. I called it

Listing 3.96: </> URLSearchParramHelpers.ts

```
import { searchParamConfig } from
"../config/SearchParamConfig"
import { isSearchParamValid } from "./WindowHelpers"

export const runSearchParamLogic = () => {
  searchParamConfig.forEach(config => {
    if (isSearchParamValid(config.key, config.value)) {
      config.action()
    }
  })
}
```

Where **isSearchParamValid** is a helper function, which we have yet to create. Create a new file **WindowHelpers.ts** under **helpers/**, and add the following:

Listing 3.97: </> URLSearchParramHelpers.ts

```
export const isSearchParamValid = () => {
  const fromValue =
  getSearchParamByKey(URLSearchParamKey.FROM)
  return fromValue && Object.values(URLSearchParamValu
  e).includes(fromValue as URLSearchParamValue) ? true
  : false
}
```

`WindowHelpers.ts` will house all functions which need to access the `window` object. Unfortunately, because Gatsby is a server side rendered framework, we always need to check. Any functions that need to do that will be written in `WindowHelpers.ts`.

Let's return to looking at `runSearchParamLogic`. `runSearchParamLogic` is a check to see if there exists a configured key and value properties in the window's search parameters, and if there is a match, we run the function defined by the `action` parameter in the respective configuration entry. In our case so far, this is the `showMailchimpSuccessToast` function, which displays a toast telling the customer their signup was successful.

Clearing Search Parameters

As a final micro-optimization, after the `forEach` runs in `runSearchParamLogic`, we can add a cleanup function which removes the search parameter, to prevent confusing behaviour if the customer was to refresh their browser for example. Like the `isSearchParamValid` valid function, we will once again need to access the `window` object to do this, so this should live in our helper file `WindowHelpers.ts`:

Listing 3.98: </> WindowHelpers.ts

```
export const clearSearchParams = (): void => {
    if (typeof window !== 'undefined') {
        window.history.replaceState('', '',
        `${window.location.pathname}`)
    }
}
```

Calling runSearchParamLogic from the Layout Component

The search parameter logic should be run as soon as any page on our application mounts. The best place for this is an on mount hook right in the **Layout** component, **Layout.tsx**:

Listing 3.99: </> Layout.tsx

```
...
useEffect(() => {
  runSearchParamLogic();
}, [])
...
```

That should do it! We should expect to see a confirmation toast to appear on the app page after the user is redirected to our success URL. The search parameter will also be removed after that, so the customer should only see the toast once.

Chapter Review

3.21 Review of the Frontend Implementation

Great work so far! If you've been following along, our landing page has all the trimmings to soon be a fully functioning MVP. In fact, as of this moment, the frontend side of things for MVP is complete! In this section, we've:

- added automatic deploys to our live custom URL every time we push to the `master` branch
- added custom styling and theming to our website via Bootstrap
- added a custom (and production optimized) SVG logo and favicon, as well as a fun CSS pseudo element plate for decoration
- added a useful ApiHelpers file, allowing for API calls including callbacks for customer feedback and error handling
- set up a robust key-value message system for both client and API originating errors
- added a ToastHelpers file, which provides an easy-to-use utility function for the `react-toastify` package
- added netlify functions with typescript, and the tooling needed to automatically build the functions each time we deploy
- created a page under the `/app` path
- added a Mailchimp signup form, redirecting customers back to the app page and showing a subscription successful toast
- learned how to robustly handle URL search parameters, the first of which being handling a subscription success redirect url from Mailchimp

Right now, our app *looks* really great when viewed in a browser - it's all responsive as well. But there's one ma-

jor problem - our product doesn't actually *do* anything yet! 😂 Remember those calls to the `/CodeGenerator` endpoint? Yeah. That endpoint doesn't exist yet - our toast will continually show the 'UNKNOWN_ERROR' message, as it will continually create a 500 error in our Netlify functions layer.

✅ Milestone Code #3 ✅

We've reached another milestone in the book, the completely MVP-ready frontend codebase!

The next step then is to build this `/CodeGenerator` endpoint on our custom API - and then we'll have a full stack MVP working which we can ship to the world. See you in the next section!

4

The Backend - Getting Started

4.1 Introduction to the Backend

Chapter Objectives

- A few of my own opinions when writing backend code with .NET
- Define the framework and tool versions used on the backend

Some Notes on My Backend Style

- Use Repositories
- Use Service Classes
- Use the EF Framework for all database migrations and modifications

Backend Frameworks and Tools Versioning

On the backend, I will be using the following versions of the following tools and frameworks:

- .NET 5.0
- PostgreSQL 13.2
- Nginx 1.17
- Ubuntu 20.04 (Focal Fossa)

4.2 Bootstrap the Backend With the .NET CLI

Getting Started

We'll start scaffolding of our API using the **`dotnet new`** command:

Listing 4.1: </> terminal

```
dotnet new webapi -n ReduxPlateApi
```

Here, we want the project to be using the **`webapi`** template, and created with the name **`ReduxPlateApi`**. When .NET has finished scaffolding the project, go ahead and **`cd`** into the newly created **`ReduxPlateApi/`** folder, and issue 'open' on the **`.csproj`** file, which should launch Visual Studio:

Listing 4.2: </> terminal

```
cd ReduxPlateApi/
open ReduxPlateApi.csproj
```

4.3 Clean Up the Backend Boilerplate Code

Just as we did for the frontend, we'll clean up some of the extra fluff that .NET has created in our API codebase:

- Delete both **WeatherForecast.cs** in the project root, and **WeatherForecastController.cs** under the **Controllers/** folder
- In Program.cs, remove the following unused imports:

Listing 4.3: </> Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;
```

- In **Startup.cs**, remove the following unused imports:

Listing 4.4: </> Startup.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.HttpsPolicy;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
\item Find and remove the line
\codeword{app.UseHttpsRedirection();} from
\codeword{Startup.cs} - this is necessary for
local development as Netlify won't let us access
endpoints from it's serverless functions which
have a self signed certificate.
```

Nice. We're at a super clean standpoint to start writing code for our API.

✅ Milestone Code #4 ✅

We've reached another milestone in the book, the boilerplate backend codebase!

4.4 Setup a Bitbucket Repository for the Backend

Create the Repository

Just as we used Bitbucket for the frontend, we will do the same for the backend. Sign into Bitbucket and create a new repository. I called my repository **`ReduxPlateApi`**, the same as I named the .NET project. Let's set this repository as our origin in the .NET project we just created.

Initialize Git

We'll need to first initialize git in the .NET project:

Listing 4.5: </> terminal

```
git init
```

Then, we can set the origin to our Bitbucket git url with:

Listing 4.6: </> terminal

```
git remote set-url origin https://princefishthrower@bi
tbucket.org/princefishthrower/reduxplateapi.git
```

Add a .gitignore File

Before commiting anything, we should make sure that we have a proper **`.gitignore`** file. Unlike Gatsby, the .NET

framework won't automatically include a **.gitignore** for you in the boilerplate. With .NET, there are quite a few artifact and build files that we don't need to track in the repository. Luckily however, the **dotnet** CLI tools provide us an easy enough command which will generate a .gitignore file for us. In the root of your .NET project, simply issue:

Listing 4.7: </> terminal

```
dotnet new gitignore
```

Now we can create our initial commit:

Listing 4.8: </> terminal

```
git add .
git commit -m "initial commit"
git push
```

4.5 Create a Digital Ocean Droplet

While our client code lives on Netlify's CDN, our custom .NET API will live on a Digital Ocean 'Droplet', which will be nothing more than a Linux Ubuntu instance. Head over to Digital Ocean and create an account if you don't have one already. Once you're logged in, click the 'Droplets' tab in the sidebar:

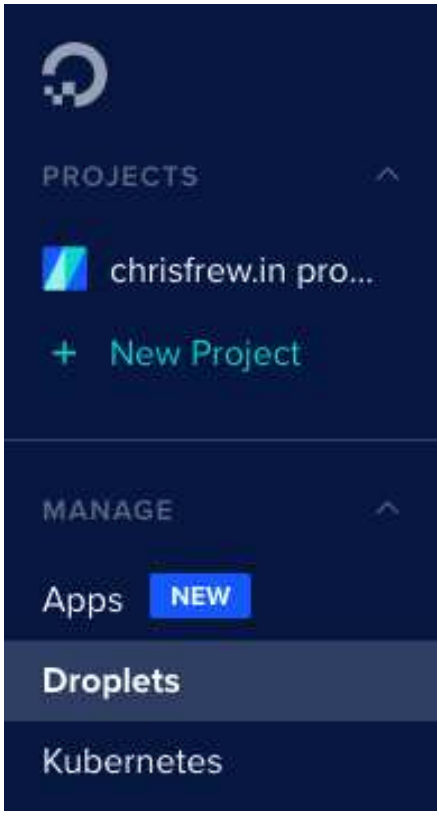


Figure 4.1.: Screenshot of the droplets tab.

In the new page that opens, click the big green ‘Create Droplet’ button:

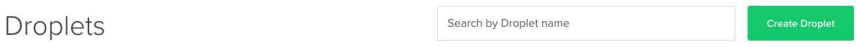


Figure 4.2.: Screenshot of the new droplet button.

On the resulting page, choose the following settings:

- Image > Distributions > Ubuntu 20.04 (LTS) x64
- Plan > Shared CPU > Basic
- CPU Options > Regular Intel with SSD > $5 / month
- Datacenter Region > Choose the option that is closet to where you think most of your customers will be!
- Authentication > SSH Keys > If you have an SSH key registered, that's great. If not read on below.
- Choose a hostname > Pick a hostname that matches your project. Following the naming convention we've been using throughout the book I will be using `reduxplate`

Generating a New SSH Key

If you don't have an SSH key saved with Digital Ocean yet, no worries. Click the 'New SSH Key' button to get started:

Take note of this IP address, as we'll need it in the next step as part of our continous integration pipeline.

⚠ ReduxPlates's Droplet IP ⚠

Though adept developers will be able to determine this value anyway with a one-liner, in an attempt to prevent snooping and attacks on ReduxPlate, I'll be using the placeholder IP of `123.456.789.0` throughout the remainder of the book. This will anyway also serve as a reminder to readers that `123.456.789.0` should be replaced with their own server IP in any commands that use it.

Chapter Review

We've put together a Digital Ocean Droplet, which runs at the insane price of just $5 / month! Don't think our app will be able to run on such a tiny little instance? Just wait and see!

4.6 Use Bitbucket Pipelines for the DevOps Framework

Just as we used Netlify for automatic builds on the frontend, let's set up Bitbucket Pipelines to automatic build and ship our API. To get started with Bitbucket Pipelines, create a **`bitbucket-pipelines.yml`** file in the root of your .NET project:

Listing 4.9: </> terminal

```
touch bitbucket-pipelines.yml
```

Add the following code to our pipeline:

Listing 4.10: </> bitbucket-pipelines.yml

```
pipelines:
  branches:
    master:
        - step:
            name: Run .NET Core publish
            image: mcr.microsoft.com/dotnet/sdk:5.0
            caches:
                - dotnetcore
            script:
                - dotnet publish --configuration
                Release -p:EnvironmentName=Production
            artifacts:
                - bin/Release/net5.0/publish/**
        - step:
            name: Deploy .NET artifacts using SCP to
            server
            deployment: production
            script:
                - pipe: 'atlassian/scp-deploy:0.3.3'
                  variables:
                    LOCAL_PATH:
                    'bin/Release/net5.0/publish/**'
                    REMOTE_PATH:
                    '/var/www/ReduxPlateApi'
                    SERVER: $SERVER
                    USER: $USER
        - step:
            name: SSH into server and issue schema
            update and restart Kestral
            script:
                - ssh $USER@$SERVER '/bin/bash
                /root/scripts/api_postbuild.sh'
```

This may appear daunting at first, so let's break it down step by step:

Step 1: We use the .NET 5.0 image (which will be cached for

speed after all subsequent builds) to make a production build of our .NET project, and define the artifacts produced by the build

Step 2: We use Secure Copy Protocol (SCP) to deploy those production artifacts to the server, under the traditional Linux path for web artifacts at `/var/www/`, with a custom folder named `ReduxPlateApi`

Step 3: We issue a script called `api\_postbuild.sh`, which will foreseeably restart the API process and do any other chores needed to be done per-release

There are a few things which we'll now need to complete. First, there are two variables in our pipeline, `\$SERVER` and `\$USER`, which we'll need to maintain on our Bitbucket for the repository. Then, we can see there is a script that will be called on the Droplet that we'll have to implement.

Adding Repository Variables to the Bitbucket Pipeline

To add the `\$SERVER` and `\$USER` variables so they can be used in your pipeline, head to your project's Bitbucket repository dashboard, and on the sidebar, click the 'Repository Settings' tab, then scroll down to the 'Pipelines' section of the sidebar and click 'Repository variables':

The resulting page will be two inputs with a name and a value. Let's maintain the two variables we need as follows:

- Name: `USER` Value: `root`
- Name: `SERVER` Value: `123.456.789.0`

When you are done, your repository variables page should look like this:

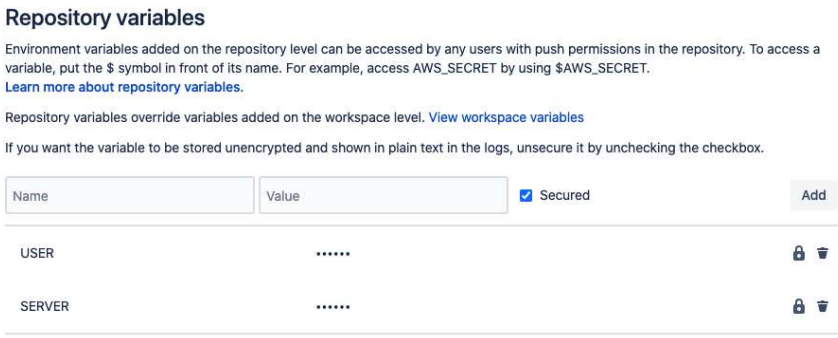


Figure 4.3.: The repository variables page with the USER and SERVER variables.

Here, note that the variables names *do not* include the `\dollar` symbol. The dollar symbol is a special character used in the `bitbucket-pipelines.yml` file to indicate it is a repository variable.

Adding Scripts and Scaffolding on the Digital Ocean Droplet

Log into your Digital Ocean Droplet via SSH with:

Listing 4.11: </> terminal

```
ssh root@123.456.789.0
```

An SSH Alias

Since I don't like typing this long SSH command every time I want to get onto my droplet (and can't be bothered to look up the IP of each droplet I own every time I want to access them), I typically make a rememberable alias in my shell profile which issues the command for me.

For example, for ReduxPlate, I have defined the following alias: `alias reduxplatessh=ssh root@123.456.789.0`

Create the API Post Build Script

Once logged in to your Droplet, create a folder in the root called **scripts**, and create a new shell file **api_postbuild.sh**:

Listing 4.12: </> terminal

```
mkdir scripts
cd scripts/
touch api_postbuild.sh
```

Add the following Bash code to **api_postbuild.sh**:

Listing 4.13: </> api_postbuild.sh

```
#! /bin/bash
source .bashrc &&
systemctl restart ReduxPlateApi.service &&
curl -X POST --data-urlencode
'payload={"text":"ReduxPlateApi Production CI
successfully completed!"}'
$REDUX_PLATE_SLACK_WEBHOOK_URL
```

You'll see here we make an HTTP POST request to a Slack webhook URL defined in the environment as **REDUX_PLATE_SLACK_WEBHOOK_URL**. We need to create a Slack bot which will send messages on our behalf when we POST to that URL.

4.7 Create a Slack Bot and Enable Webhooks

If you don't have a Slack account yet, go ahead and create one, they are free. Then, head to **https://api.slack.com/apps**, and click the 'Create New App' app button:

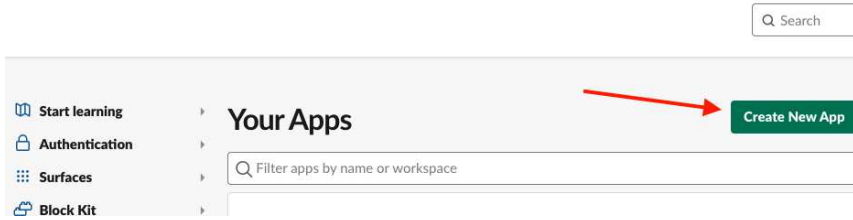


Figure 4.4.: Creating a new app on the Slack API site.

'From scratch' should be fine:

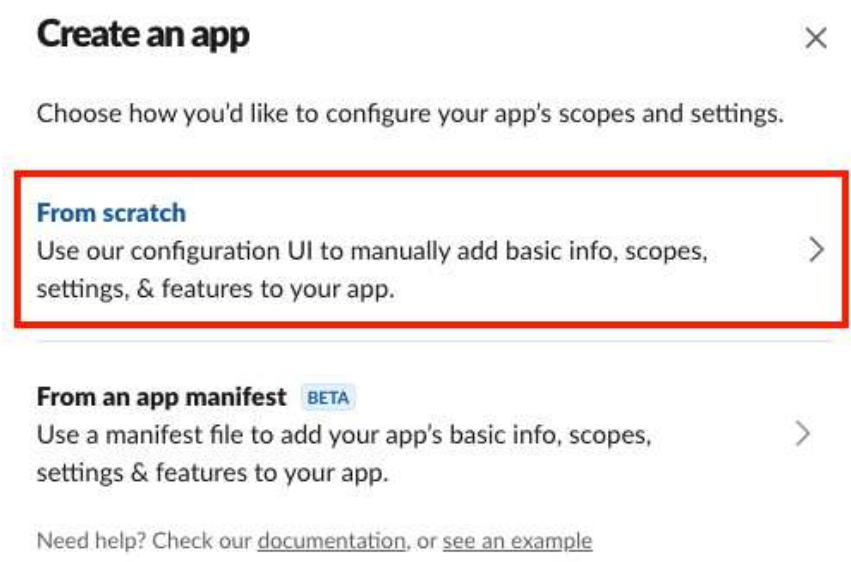


Figure 4.5.: Selecting the 'from scratch' option in the Slack API UI.

and provide a name. I called my application 'ReduxPlate Deploy Bot'. Select your workspace as well, and click 'Create App':

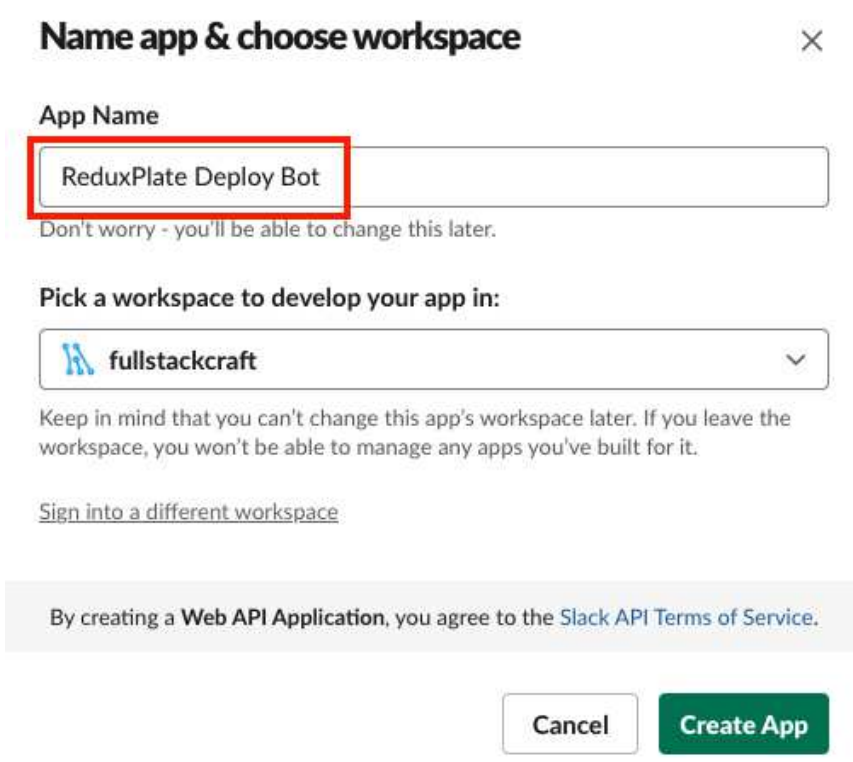


Figure 4.6.: Selecting the 'from scratch' option in the Slack API UI.

In the resulting screen, select 'Add features and functionality', and then the 'Incoming Webhooks' tab:

Activate the incoming webhooks

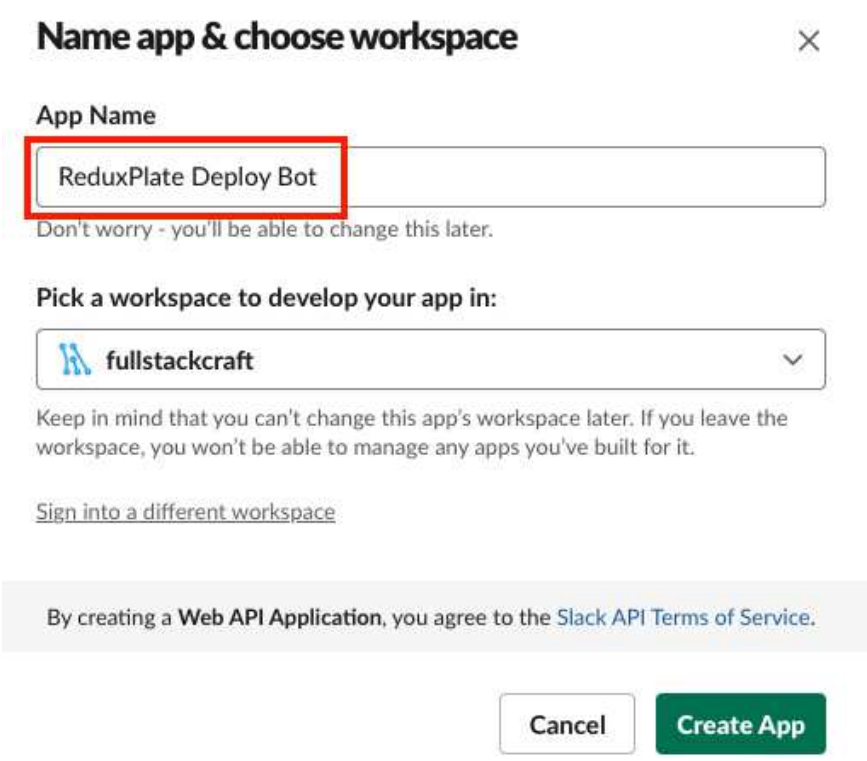


Figure 4.7.: Selecting the 'from scratch' option in the Slack API UI.

Finally, maintain this environment variable in the `.profile` file located in the root of the Droplet:

Listing 4.14: </> .profile

```
export REDUX_PLATE_SLACK_WEBHOOK_URL=https://hooks.sla
ck.com/services/ABCDEFG/HIJKLMNOP/qRsTuVwxyz
```

4.8 Create the ReduxPlateApi folder

As we saw in the build script, we put the artifacts in the **/var/www/ReduxPlateApi** folder. We need to ensure this folder exists, or otherwise the SCP command in our pipeline will fail. From the Droplet, issue:

Listing 4.15: </> terminal

```
cd /var/www/
mkdir ReduxPlateApi
```

4.9 Try Out the Continous Integration Pipeline

We should now have all we need on our production server to kick off our first API build. On your development machine, add and commit all files:

Listing 4.16: </> terminal

```
git add .
git commit -m "Bitbucket pipelines ready to go!"
git push
```

We haven't made any branches other than **master**, so this commit should fire off the build process right away. Within a few minutes, you should see on your new application emit a message on the Slack channelyou chose, something like this:

Awesome. Our continuous integration pipeline is working!

5

The Backend - Implementation

5.1 Writing the First Endpoint for the Custom API

Great. Our .NET API is up and running, and, like the frontend, it is automatically built and deployed upon pushing to the `master` branch. Typically, when I get to this point, as a sanity check, I create a `Root` controller which just returns some plain text of an API version string, or really whatever you'd like to have as a public facing endpoint for your API.

Getting Started

First we'll create a new API controller. In Visual Studio, the easiest way to do this is to let Visual Studio code template the controller for us. First right click on the `Controllers/`

folder and select ‘Add’ > ‘New Class...’, and in the resulting dialog, select ‘ASP.NET Core’ in the left most list, and then at the very bottom ‘Web API Controller Class’. Don’t forget to provide the name ‘RootController’ for the file name:

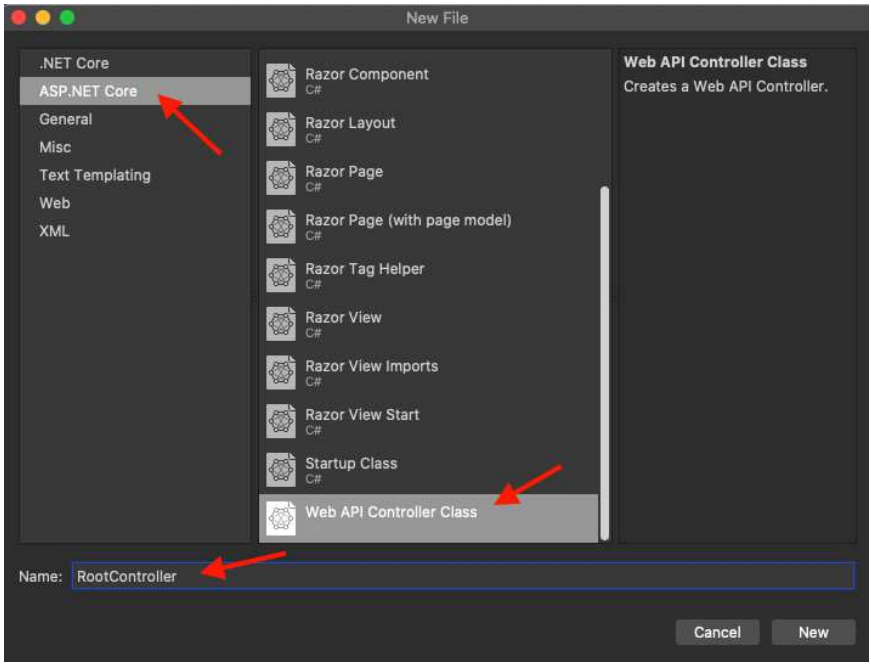


Figure 5.1.: Selecting the ‘Web API Controller Class’ template choice in Visual Studio.

There are few code modifications we need to make to this template:

- First, we want to extend `ControllerBase`, not `Controller`.
- Delete all example class methods except for `Get`
- Remove the unused packages and the comments all around the class.

- Change the signature of the `Get` method from `public string` to `public ActionResult<string>`. You'll also need to wrap the returned string with the `Ok()` method.
- In this special case, remove the endpoint Attribute `[Route("api/[controller]")]` - we don't want any controller name in the route name since this is the root controller
- Also modify the `[HttpGet]` attribute to `[HttpGet("/")]` - this will tell Swagger where the endpoint is, so it will actually show up in the API documentation

Finally, fill the `return` value with whatever string you'd like. With these changes, your `RootController` should be quite a short source file and look something like this:

Listing 5.1: </> Startup.cs

```
using Microsoft.AspNetCore.Mvc;

namespace ReduxPlateApi.Controllers
{
    public class RootController : ControllerBase
    {
        [HttpGet("/")]
        public ActionResult<string> Get()
        {
            return Ok("ReduxPlate API v1.0.0");
        }
    }
}
```

For the first time, we're ready to spool up our API! Go ahead and click the run project button at the top left corner

of Visual Studio, which looks like a play button:

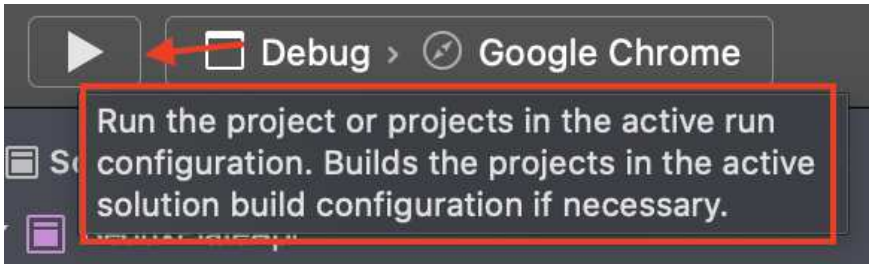


Figure 5.2.: The run project button and it's description in Visual Studio.

A browser should open immediately at **https://localhost:5001/swagger/index.html** and you should see a resulting screen with our single endpoint at '/':

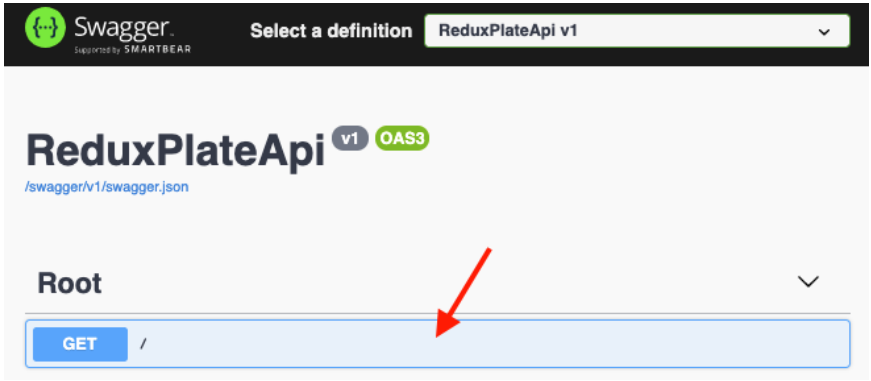


Figure 5.3.: Initial Swagger screen with expandable endpoint method bars.

Go ahead and click this blue bar to expand it, then the 'Try

it out' button. As we will see later, with endpoints that require parameters, swagger provides inputs for each, along with type requirements on those fields. For now, our simple GET endpoint can be called immediately by clicking the the 'Execute' button:

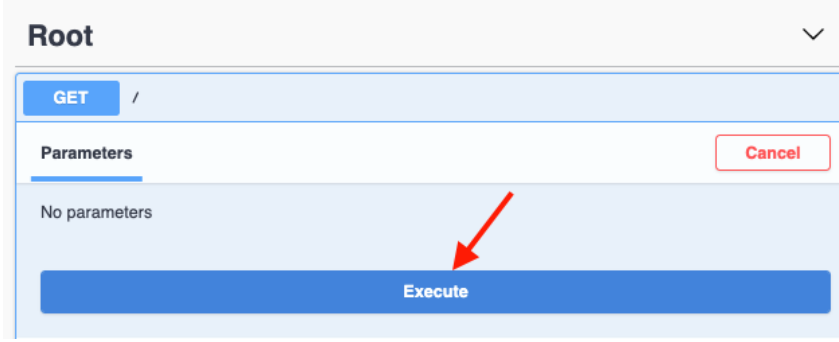


Figure 5.4.: Expanded method bar with 'Execute' button shown

We immediately see a detailed 'Responses' panel appear with the expected response body. Swagger also has a panel for the response headers:

Figure 5.5.: Detailed response panel showing both the response body and response headers.

As we write more endpoints, this Swagger documentation page will become invaluable. It lists all endpoints, their HTTP method, and even generates example responses, without us even having to do the 'Try it out' / 'Execute' workflow.

5.2 Writing the Generate Endpoint

Following the same pattern as the previous section, create a new controller called `CodeGeneratorController`.

Just as we did for the client, we need to define two contracts that define the models we expect to recieve and

return in this endpoint. Create a folder in the project root called **Models**. Following the same names as what was defined in the client, but using .NET naming patterns, we'll create corresponding class files **GeneratorOptions**, **Generated**, and **File**:

Listing 5.2: </> GeneratorOptions.cs

```
using System.Collections.Generic;

namespace ReduxPlateApi.Models
{
    public class GeneratorOptions
    {
        public string StateCode { get; set; }
    }
}
```

Listing 5.3: </> Generated.cs

```
using System.Collections.Generic;

namespace ReduxPlateApi.Models
{
    public class Generated
    {
        public List<File> Files { get; set; }
    }
}
```

Listing 5.4: </> File.cs

```
namespace ReduxPlateApi.Models
namespace ReduxPlateApi.Models
{
    public class File
    {
        public string FileLabel { get; set; }

        public string Code { get; set; }
    }
}
```

Note that all these models are identical to their client counterparts in names and typing, except for the fact that they take on the .NET capatalized naming convention. A great feature of .NET APIs is that we don't need to worry about the differences between the conventions either - .NET will automatically understand and associate properties of our models regardless of casing, in both serialization and deserialization in our endpoints.

5.3 Building a Code Generator Service Class

To keep our `CodeGeneratorController` clean, we're now going to build a service class called `CodeGeneratorService`. Create a new folder in the project root called `Services`, and create a new class `CodeGeneratorService`. You can leave the boilerplate code there fore now.

To properly incorporate this into .NET's dependency injection, we should also create an interface for this service to implement. For now, it will just have a single method we should implement called `Generate`. First create yet another folder called `Infrastructure`, and under that folder

a folder called **Services**. Create a new interface called **ICodeGeneratorService**:

Listing 5.5: </> ICodeGenerateService.cs

```
using System.Threading.Tasks;
using ReduxPlateApi.Models;

namespace ReduxPlateApi.Infrastructure.Services
{
    public interface ICodeGeneratorService
    {
        Task<Generated> Generate(GeneratorOptions
        generatorOptions);
    }
}
```

You may have noticed that I do not return the **Generated** model, but a **Task<Generated>** model. For reasons that we will soon see, the **Generate** method will have to be asynchronous.

With **ICodeGenerateService** complete, don't forget to implement it in **CodeGeneratorService**, with a **: ICodeGeneratorService**

5.4 Parsing the Code Editor's Source Code with the Type-

We'll now be writing a microservice that will be a key part of our SaaS product: the service that generates the Redux code for us. While with some effort this could be conceivable done using fancy regular expresions and template strings directly in the .NET project, that way of solving hte problem is fighting against a powerful tool that already ex-

ists, is maintained by hundreds of developers, and is proven to work time and time again: the itself! Accessing the (AST) from the TypeScript Compile API, it should be a breeze to parse and navigate the code sent from the client to the server, and generate the files to return to the client.

Where's the .NET code?

The code generation process on the server is going to be a bit non-traditional in terms of a .NET API, in that the code to generate the Redux boilerplate code *will not* be written in native C# code, but in a Node.js project. We require running the TypeScript compiler and AST services, and by definition TypeScript will need to be run in it's native environment. We will be building a Node.js microservice which will be called from our .NET project.

Scaffold the the Node.js Project

First create a directory in the .NET root called **`Microservices`**. Then create yet another folder within **`Microservices/`** called **`redux-plate-code-generator`**. Move into this directory with the terminal, and as we saw for the serverless functions, initialize a new Node.js project:

Listing 5.6: </> terminal

```
cd Microservices/redux-plate-code-generator/
npm init
```

After answering the prompts, my **`package.json`** resulted in the following:

Listing 5.7: </> package.json

```
{
  "name": "redux-plate-code-generator",
  "version": "1.0.0",
  "description": "Uses the TypeScript compiler API to
  generate Redux code from state alone.",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit
    1"
  },
  "repository": {
    "type": "git",
    "url": "git+https://princefishthrower@bitbucket.or
    g/princefishthrower/ReduxPlateApi.git"
  },
  "keywords": [
    "code",
    "generator",
    "code",
    "printer",
    "typescript",
    "ast",
    "typescript",
    "compiler",
    "typescript"
  ],
  "author": "Chris Frewin",
  "license": "MIT",
  "homepage": "https://bitbucket.org/princefishthrower
  /ReduxPlateApi#readme"
}
```

Install the ts-morph package

We will be parsing out the names and types of the state interface using the package **ts-morph**. **ts-morph** is a developer-friendly wrapper around Typescript's compiler API. With it, we can rather quickly access all parts of TypeScript's abstract syntax tree (AST) API - to parse out what we need from a given string of TypeScript source code and be on our way.

First install **ts-morph** with **npm**:

Listing 5.8: </> terminal

```
npm install ts-morph
```

Scaffold the Node.js project for builds with TypeScript

There's some initial housekeeping we have to do before writting any code for our microservice. First we want to allow writing code with TypeScript. Create a **tsconfig.json** file in the project root with the following:

Listing 5.9: </> tsconfig

```
{
  "compilerOptions": {
    "strict": true,
    "isolatedModules": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "removeComments": false,
    "preserveConstEnums": true,
    "resolveJsonModule": true,
    "outDir": "./dist",
    "lib": ["es2016", "dom"],
    "downlevelIteration": true
  },
  "include": ["./src/**/*"]
}
```

As we did for the serverless functions, everything in the `src/` folder will be compiled into the `dist/` folder. Let's create the `src/` now, and start writing code!

Writing the Generator Service

Inside the `src/` folder, create a new folder called `services`, and create the a file called `GeneratorService.ts`. This file will hold our main service class to generate code with.

Listing 5.10: </> CodeGeneratorService.ts

This is a rather complex class. Let's unpack it one step at a time. In the constructor, we start creating the variety of **SourceFile** objects in **ts-morph** - which represent real TypeScript files. We also call the extensive **runValidations** function - which requires that the code pass a variety of checks - no syntax errors. Many of these are design decisions, but some also insure that all following code execution.

The **generate** function then houses the actual logic of building the files - adding code to the **types.ts** file, and building from scratch the **reducers.ts** and **actions.ts** file.

Note here that I've employed the same type of messaging pattern as I did for the app messages on the client. I've defined an **ApiErrorMessage** enum that includes all the various error codes we throw from the **runValidations()** function:

Listing 5.11: </> ApiErrorMessage.ts

```
  enum ApiErrorMessage {
    FIX_SYNTAX_ERRORS = 'FIX_SYNTAX_ERRORS',
    ONE_INTERFACE_LIMIT = 'ONE_INTERFACE_LIMIT',
    STATE_IDENTIFIER_IN_INTERFACE_REQUIRED =
    'STATE_IDENTIFIER_IN_INTERFACE_REQUIRED',
    ONLY_CERTAIN_PRIMITIVES_SUPPORTED_IN_STATE =
    'ONLY_CERTAIN_PRIMITIVES_SUPPORTED_IN_STATE',
    STATE_NAME_MUST_BE_CAPITALIZED =
    'STATE_NAME_MUST_BE_CAPITALIZED',
    MAX_FIVE_PROPERTIES_ALLOWED_IN_STATE =
    'MAX_FIVE_PROPERTIES_ALLOWED_IN_STATE'
}

export default ApiErrorMessage
```

With these messages, we don't need to immediately in-

clude a custom type or a config with actual message values as we did on the client with **AppErrorMessages**. These **ApiErrorMessages** will be parsed in the client. This anyway *must* be the responsiblity of the client, as this would include information about the language and locale the customer is using so throwing an error with just the keyed code from the server is perfect.

There are also a series of string converting helper functions which I've placed in **StringHelpers.ts**:

Listing 5.12: </> StringConversionHelpers.ts

```
import { isLowerCase } from "../utils/isLowerCase";

export const convertCamelCaseToCapsCamelCase = (str:
string): string => {
  return `${str[0].toUpperCase()}${str.slice(1,
  str.length)}`;
};

export const convertCamelCaseToCapsUnderscore = (str:
string): string => {
  const capsStr = convertCamelCaseToCapsCamelCase(str);
  return [...capsStr].reduce((acc, cur) => {
    return `${acc}${isLowerCase(cur) ?
    cur.toUpperCase() : `_${cur}`}`;
  });
};

export const convertPropertyNameToActionConstName =
(propertyName: string): string => {
  return `SET_${convertCamelCaseToCapsUnderscore(prope
  rtyName)}`;
};

export const convertPropertyNameToActionInterfaceName
= (propertyName: string): string => {
  return `Set${convertCamelCaseToCapsCamelCase(propert
  yName)}Action`;
};

export const convertPropertyNameToActionFunctionName =
(propertyName: string): string => {
  return `set${convertCamelCaseToCapsCamelCase(propert
  yName)}`;
};
```

which in turn uses the utility function **isLowerCase**:

Listing 5.13: </> isLowerCase.ts

```
export const isLowerCase = (str: string) => {
  return str === str.toLowerCase() && str !=
  str.toUpperCase();
};
```

Testing the Code

To create a way to test all this code, we can create an **index.ts** file with the following:

Listing 5.14: </> index.ts

```
import CodeGeneratorService from
"./services/CodeGeneratorService"

const stateCode = `export interface ReduxPlateState {
    myString: string
}`

const run = async () => {
    const codeGeneratorService = new
    CodeGeneratorService(stateCode);
    const files = await codeGeneratorService.generate()
    files.files.forEach(file => {
        console.log(`------${file.fileLabel}------`)
        console.log(file.code)
    })
}
run();
```

Feel free to change the value of the code in **stateCode** to any valid (or invalid!) TypeScript interface of defining a Redux slice of state. We should add both **build** and **develop**

scripts to our **package.json** to both compile and then run the JavaScript emitted version of **index.ts**:

Listing 5.15: </> package.json

```
...
"scripts": {
  "build": "tsc",
  "develop": "npm run build && node dist/test.js"
},
...
```

Go ahead and issue **npm run develop** and behold as beautiful TypeScript code is printed to the console! You should see something like this printed:

Listing 5.16: </> terminal

```
------types.ts------
export interface ReduxPlateState {
    myString: string
}
export const SET_MY_STRING = 'SET_MY_STRING'

export interface SetMyStringAction {
  type: typeof SET_MY_STRING
  payload: {
    myString: string
  }
}
export type ReduxPlateActionTypes = SetMyStringAction

------reducers.ts------
import { ReduxPlateActionTypes, ReduxPlateState,
SET_MY_STRING } from "./types"

export const initialReduxPlateState: ReduxPlateState =
{
  myString: '',
}

export function ReduxPlateReducer(state =
initialReduxPlateState, action:
ReduxPlateActionTypes): ReduxPlateState {
  switch (action.type) {
    case SET_MY_STRING:
      return {
        ...state,
        myString: action.payload.myString
      }
    default:
      return state
  }
}

------actions.ts------
import { ReduxPlateActionTypes, SET_MY_STRING } from
"./types"

export function setMyString(myString: string):
ReduxPlateActionTypes {
  return {
    type: SET_MY_STRING,
```

You can start to feel that we are really getting close to our MVP now! 😉 We just need to complete the full stack connection flow from client to microservice now.

Adding Code Generator Microservice Artifacts to .gitignore

Now that our micoservice is running well, let's do some housekeeping to make sure unwanted files are not included in the repository. In the root of the .NET project, add the following lines to your **.gitignore**:

Microservice Review

When complete, the source code organization of our microservice's `src/` folder should look like this:

Listing 5.17: </> terminal

```
src
├── constants
|   └── Constants.ts
├── enums
|   ├── ApiErrorMessage.ts
|   └── Primitive.ts
├── helpers
|   └── StringHelpers.ts
├── index.ts
├── interfaces
|   ├── IFile.ts
|   ├── IGenerated.ts
|   └── ITypeScriptProperty.ts
├── services
|   └── CodeGeneratorService.ts
├── types
|   └── ApiErrorMessageConfigEntries.ts
└── utils
    └── isLowerCase.ts
```

5.5 Implementing CodeGeneratorService

With our microserve written, succesfully compiling to JavaScript, and succesfully creating some Redux code, let's get back to the .NET codebase and implement the

`CodeGeneratorService` class, which will be the class calling our TypeScript service class in the first place!

Install the Jering.Javascript.NodeJS Nuget package

We will be using the `Jering.Javascript.NodeJS` package. Open the NuGet window in Visual Studio via the Project > Manage NuGet Packages... option:

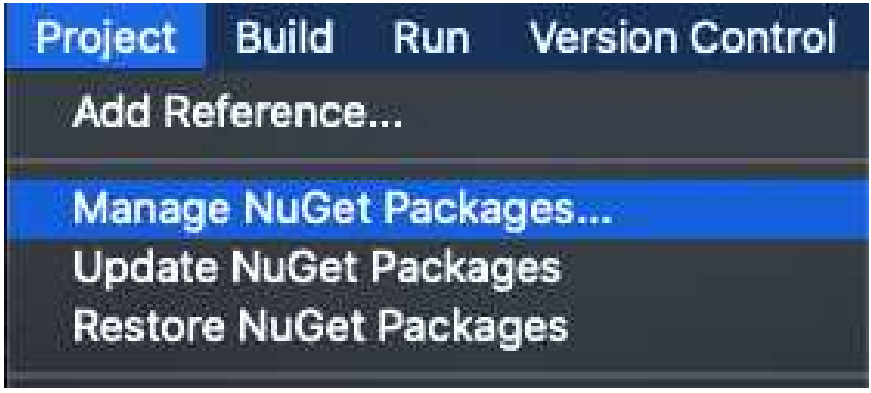


Figure 5.6.: The manage NuGet packages menu.

Then search for 'Jering', and one of the top (if not the top) result should be the `Jering.Javascript.NodeJS` package. then click 'Add Package':

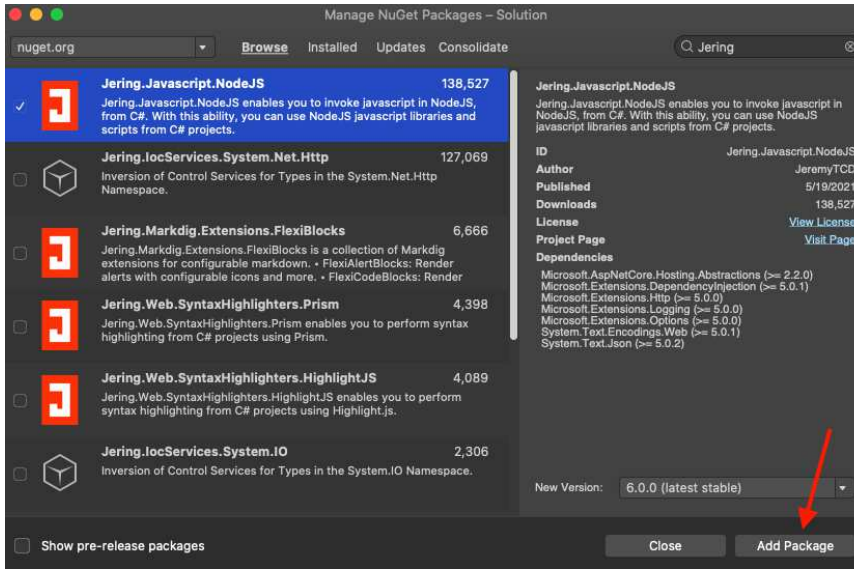


Figure 5.7.: The NuGet window, searching for 'Jering'.

To use this in dependency injection across our app, we need to add it to our services. In **Startup.cs**, add the following to the **ConfigureServices** method:

Listing 5.18: </> Startup.cs

```
services.AddNodeJS();
```

Using **Jering.Javascript.NodeJS**, we can call Node.js code directly from our .NET project. First we will have to get our Node.js function in a format that **Jering.Javascript.NodeJS** expects, which is the **module.exports** format, and with a callback function to be called. So far we've been testing our code in **index.ts**. Move that source code to a new file called **test.ts**. We

should probably update our **develop** script to reflect that change as well:

Listing 5.19: </> package.json

```
...
"develop": "tsc; node dist/test.js"
...
```

So now when we run develop, we'll really be running our 'test' script after compiling the project. Now we can replace **index.ts** with the following:

Listing 5.20: </> ApiErrorMessage.ts

```
import IApiErrorMessage from
"./interfaces/IApiErrorMessage";
import IGenerated from "./interfaces/IGenerated";
import CodeGeneratorService from
"./services/CodeGeneratorService";

module.exports = (
  callback: (_: null, result: IGenerated |
  IApiErrorMessage) => void,
  stateCode: string
) => {
  try {
    const codeGeneratorService = new
    CodeGeneratorService(stateCode);
    const generated = codeGeneratorService.generate();
    callback(null, generated);
  } catch (error) {
    throw error.message
  }
};
```

TypeScript will complain that it cannot find the name **module**. Follow TypeScript's suggestion and install **@types/node** as a development dependency:

Listing 5.21: </> terminal

```
npm install --save-dev @types/node
```

There are a few things to note with this new **index.ts** file. First, I am doing something very special with the try / catch block, throwing only the **message** property of the **error**. This is intentional, as we only want to return the **ApiErrorMessage** enum value. A standard JavaScript **Error** object will include the entire stack trace in the **message**.

If we issue **npm run build** now, we should see the **dist/index.js** file populated now with our **module.exports** code. Likewise, the our testing script will be written to **dist/test.js**. In the end, it is this **dist/index.js** we will need for our .NET code. **Jering.Javascript.NodeJS** offers a way to call a JavaScript file asynchronously with **InvokeFromFileAsync**. We need only to wrap this call in a try catch, since we know our JavaScript file can throw exceptions, and our .NET **CodeGeneratorService** is completed in a rather succint fashion:

Listing 5.22: </> CodeGeneratorService.cs

5. The Backend - Implementation

```
using System;
using System.IO;
using System.Threading.Tasks;
using Jering.Javascript.NodeJS;
using Microsoft.Extensions.FileProviders;
using ReduxPlateApi.Infrastructure.Services;
using ReduxPlateApi.Models;

namespace ReduxPlateApi.Services
{
    public class CodeGeneratorService :
    ICodeGeneratorService
    {
        private readonly INodeJSService nodeJSService;

        public CodeGeneratorService(INodeJSService
        nodeJSService)
        {
            this.nodeJSService = nodeJSService;
        }

        public async Task<Generated>
        Generate(GeneratorOptions generatorOptions)
        {
            try
            {
                var physicalProvider = new
                PhysicalFileProvider(Directory.GetCurr
                entDirectory());
                var filePath =
                Path.Combine(physicalProvider.Root,
                "Microservices",
                "redux-plate-code-generator-cli",
                "index.js");
                return await nodeJSService.InvokeFromF
                ileAsync<Generated>(filePath, args:
                new[] { generatorOptions.StateCode });
            }
            catch (Exception exception)
            {
                throw new Exception(exception.Message.
                Replace("\n",
                ""));
            }
        }
    }
```

Note here that I am using the built-in **System.IO.Path** and **Microsoft.Extensions.FileProviders.PhysicalFileProvider** classes to get at the produced JavaScript artifact in a OS-independent way - it's never a good idea to hardcode a filepath in code!

5.6 Use CodeGeneratorService in CodeGeneratorCon-

With the implementation complete, let's use **CodeGeneratorService** in **CodeGeneratorController**. **CodeGeneratorService** abstracts away all the code generation, and so the complete **CodeGeneratorController** is as simple as:

</> CodeGeneratorController.cs </>

```
{The completed CodeGeneratorController.}
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using ReduxPlateApi.Infrastructure.Services;
using ReduxPlateApi.Models;

namespace ReduxPlateApi.Controllers
{
    [Route("/[controller]")]
    public class CodeGeneratorController :
    ControllerBase
    {
        private readonly ICodeGeneratorService
        codeGeneratorService;

        public
        CodeGeneratorController(ICodeGeneratorService
        codeGeneratorService)
        {
            this.codeGeneratorService =
            codeGeneratorService;
        }

        [HttpPost]
        public async Task<ActionResult<Generated>>
        PostAsync([FromBody] GeneratorOptions
            generatorOptions)
        {
            try
            {
                var generated = await this.codeGenerat
                orService.Generate(generatorOptions);
                return Ok(generated);
            }
            catch (Exception exception)
            {
                var apiErrorMessage = new
                ApiErrorMessage
                {
                    Message = exception.Message
                };
                return St ode(StatusCodes.Status50
                0InternalServerError,
                apiErrorMessage);
            }
```

We return the expected `Generated` model if all goes well, and return an `ApiErrorMessage` model with the `Message` property set to the exception message - which *should* be one of the enum values from `ApiErrorMessage` enum from our TypeScript microservice. Even if something far worse happens on the server side and this error message is empty, null, undefined, or something else entirely, we still have the fallback unknown error on the `switch` statement on the frontend.

5.7 Recap

So far so good. Currently, your project structure (omitting the contents of Microservices) should look like this:

terminal

```
{Current .NET project structure.}
.
├── Controllers
|   ├── CodeGeneratorController.cs
|   └── RootController.cs
├── Infrastructure
|   └── Services
|       └── ICodeGeneratorService.cs
├── Microservices
├── Models
|   ├── File.cs
|   ├── Generated.cs
|   ├── GeneratorOptions.cs
|   └── TypeScriptProperty.cs
├── Program.cs
├── Properties
|   └── launchSettings.json
├── ReduxPlateApi.csproj
├── ReduxPlateApi.sln
├── Services
|   └── CodeGeneratorService.cs
├── Startup.cs
├── appsettings.Development.json
└── appsettings.json
```

We should be ready to handle the hardcoded 'free' version of the request we create on the client side. Let's start the .NET API! As soon as the project is done building, we should now see in the Swagger dashboard both the initial root endpoint (at '/') we wrote, and the new endpoint we've just finished, at '/CodeGeneator'. Note as well that this is a POST endpoint, which shows up in Swagger as green as apposed to GET's blue. You can test the endpoint in the Swagger page to check that the endpoint is working. But it's going to be a lot more fun to test right from the client, right? Let's give it a go!

5.8 Calling the new Generate endpoint from the Client

.

6

Building a Staging (or Testing) Environment

So far we've focused on building out the frontend and custom backend API for ReduxPlate. We write code in our `develop` git branch, but every time we merge to the `master` branch in either our frontend or backend repositories, the continuous integration process is fired off and shipped to our live SaaS product immediately. Our continuos integration tool for the frontend is Netlify, and with the backend

it is Bitbucket pipelines. That's been great so far for prototyping our MVP, but it's fairly risky once we start having customers.

In this section of the book, we'll get into building out what is known as a staging environment. With all of the tooling available in netlify on the frontend side, and .NET on the backend side, the challenge is not too great, but there will be some important considerations and distinctions which we'll look at in detail.

6.1 The Essential need for a Testing Environment

A staging environment is important, because it mimics your live product almost exactly. As we'll see in this section, in comparison to your live product, the staging version of your product will differ only in small configuration changes. Perhaps the exact quality of what certain API endpoints return may differ, but other than that, your staging site is essentially a production-like, risk-free playground where you can test new features, or catch bugs before they ship to production.

6.2 Staging CI / CD for the Frontend

We'll get the client side of things out of the way first. Again, Netlify's powers come to the rescue and setting up a staging version of the frontend is absolute peanuts.

6.3 Create a Staging Branch for the Frontend

To get started, we'll branch off our develop branch into a new:

6.4 Configure Netlify to Build According to the Staging

On Netlify, head to your product's DNS.

The staging site is up and running! We've got the correct staging environment variables up, builds are firing when we merge to staging; all is well. But if we open a console while looking - we can see . We're getting a bunch of 404 errors when we try to call the staging API endpoint we defined at staging.api.reduxplate.com. Let's switch gears into backend mode and rectify this issue.

6.5 Staging CI / CD for the Backend

Our .NET application will unfortunately be a bit more involved than what it took with Netlify due to it's custom nature. But, .NET and BitBucket offer a lot of powerful features which make the process not too difficult.

6.6 Create a Staging Branch for the Backend

As we did with the frontend, branch of of the development repository for the backend:

Staging Environment Recap

Perfect. We've successfully built out a staging environment from layers as deep as the database, all the way to the frontend. Tools like Bitbucket Pipelines and Netlify's branch builds made this a relatively painless task as well, since we already had the production environments working.

✅ Milestone Code #6 ✅

We've reached the 6th milestone in the book, the completion of our CI and CD process for the backend and frontend! Because this section included efforts on both the frontend and backend, it will include two milestone repositories for the frontend and backend respectively, `frontend-full-ci-cd-complete` and `frontend-full-ci-cd-complete`.

7

Building Full Stack Testing Suite

Admiteddly, as a solo developer, they often take a backseat. But in the long run they save time, and with a few , we can integrate them into both our backend and frontend CI and CD.

7.1 Frontend - Installing Cypress

For the frontend, we will be using Cypress as our testing library.

7.2 Backend - Installing xUnit

For the frontend, we will be using xUnit as our testing library.

8

The Frontend - Advanced Implementation

> Any intelligent fool can make things bigger, more complex, and more violent. It takes a touch of genius—and a lot of courage to move in the opposite direction.
>
> *(E.F. Schumacher, 1973)*

The remainder of the frontend implementation that will be discussed in this book will including handling user authenticationa and authorization, adding stripe and a Fauna database to handle user data, building out features of the `/app` page, and other various advanced features around the app.

8.1 Building the App Page

So far on the page at `/app`, we've only created a title with a signup form to build interest from our VIP staged product. The MVP stage is over: it's time to build out our app page with it's full feature set.

8.2 Extending the Code Generator API Contract

Currently, the interface used to call the `CodeGenerator` endpoint, `IGenerateOptions`, has only a single property `stateCode`.

8.3 Add Netlify Identity as the Authentication and Autho-

We've got a decent UI to work with, including now both a ToastHelpers and ApiHelpers class. It's time to add all the code to allow users to sign up, log in and log out.

Chapter Objectives

- Install the `netlify-identity-widget` package
- Scaffold main functions to modify state in the app when the user logs in or logs out

Getting Started

We'll be using the official **netlify-identity-widget**. This package markets itself as 'A zero config, framework free Netlify Identity widget'.

Install the netlify-identity-widget Package

Get started by install **netlify-identity-widget**:

Listing 8.1: </> terminal

```
npm install netlify-identity-widget
```

Then create a file under **src/helpers/** called NetlifyIdentityHelpers.ts. This is the single place where we will import and use **netlify-identity-widget** package.

8.4 Adding Netlify State to Redux

As we saw in the previous section, we need to manage the user's Netlify state across the app. We'll need to add the Netlify state to Redux to accomplish this. As an added bonus, we'll even be using **redux-persist** to persist the user state within **localStorage**.

Resolving User Roles

It's clear we need a utility to determine the user's role across the application - whether they are a public visitor or have bought a premium subscription. While it may be tempting to build a simple if else utiltiy function, we're going to leverage some powerful TypeScript features to make a robust solution.

Note that this style of solution is future proof as well: it doesn't matter if we add additional roles later, for exam-

ple a ‘deluxe’ or ‘corporate’ plan. To achieve this future-proofing, we don’t use any `switch`, `if`, or `else if` logic on the user’s role. Instead we opting for the `Object.values()` of the available roles, and compare them against every role with `roles.find()`.

8.5 Use Stripe for the First Payments Platform

Chapter Objectives

- Setting up Stripe to accept subscriptions

8.6 Use Netlify Serverless Functions

From the last section, we saw that we need to use a protected, mainly because we need to access the Stripe secret key to generate a sessions for the customer who wishes to subscribe.

8.7 Set Up Fauna DB for User Management

In the last section, the need arose to . It’s easiest to assign a stripe ID as soon as a customer signs up. This ID will then be tied to their Netlify ID.

8.8 Building a Pricing Section

ReduxPlate has a rich set of features for Premium subscribers. We should showcase that right on the homepage with a pricing component, which will tease some of the features. It will also summarize the .

✅ Milestone Code #5 ✅

We've reached the nth milestone in the book, the advanced implementation Part I: frontend! This milestone code includes the full working code, **advanced-implementation-part-i**

8.9 Dynamically Setting Animations

While creating our fancy animated logo, I mentioned that we would create a way to deactivate the animations on any page other than the homepage. We will do this by creating a custom hook. In the **hooks** folder, create a new file **useShouldAnimate.ts**. This file should include the following:

Listing 8.2: </> useSSRSafeWindowLocation.ts

```
export const useSSRSafeWindowLocation = (): string => {
  const [location, setLocation] = useState<string>()

  // every time didMount changes, attempt to set the
  location
  useEffect(() => {
    if (didMount && typeof window !== "undefined") {
      setLocation(window.location)
    }
  },[didMount])
}
```

Listing 8.3: </> useShouldAnimate.ts

```
export const useShouldAnimate = (): boolean => {
  const location = useWindowLocation()
  const [shouldAnimate, setShouldAnimate] =
  useState<boolean>(true)

  // every time location changes, set the
  useEffect(() => {
    setShouldAnimate(location === "/")
  },[location])
}
```

Via location, it returns a boolean if the visitor is on the homepage or not.

✅ Milestone Code #8 ✅

We've reached the 6th milestone in the book, the advanced implementation Part II: frontend completed! This milestone code includes the full working code, **advanced-implementation-part-ii**

9

The Backend - Advanced Implementation

As we have seen from the advanced frontend section, there are a few advanced tasks we need to complete on the backend.

9.1 Further Options for the CodeGenerate endpoint

9.2 Building the ReduxDoc Endpoint

✅ Milestone Code #9 ✅

We've reached the 6th milestone in the book, the advanced implementation Part II: frontend completed! This milestone code includes the full working code, `advanced-implementation-part-ii`

What's Next?

Our SaaS app is in a pretty good position right now: we have staging and production environments running successfully side by side (on both the frontend and backend), and we have fully working user onboarding flow thanks to Netlify and Fauna DB, and are able to process payments and subscriptions with Stripe. We've also built out some advanced functionality on both the front and backends.

The remainder of this book takes will take our SaaS app to the next level. The remaining sections consist of a variety of "recipes" on how to integrate things like additional payment providers, application-wide logging, and examples of automation tasks you may want to add to your application. I would recommend trying to implement them *all*, as they will bring your SaaS app above and beyond intergalactic standards! 🚀

10

Recipe: Additional Payment Platform Integrations

10.1 Introduction

Payment integrations are an essential part of any SaaS production. In this chapter, we'll learn how to connect Stripe, PayPal, and Gumroad into the frontend flow, be notified of both new subscriptions and unsubscriptions, and automati-

cally update the role in the user's netlify Identity user automatically.

11

Recipe: Add Application-Wide Logging

12

Recipe: Adding Custom Emails

While Netlify takes care of the user email flow (welcome emails, reset password, forgot password)

13

Recipe: Adding Automation

14

Recipe: SEO Optimization

Background

Using the Gatsby framework, we should be able to to get 100s across the board in google's Lighthouse tool. Google rewards sites which score highly with Lighlhouse, meaning better search results. In this section of the book, we'll look at the initial score for Lighthouse how ReduxPlate is now, and I'll walk through all optimizations we can make to ReduxPlate, getting it to 100s across the board with Lighthouse.

Chapter Objectives

- Learn how to use Lighthouse in Chrome debugger
- Learn how to solve problems and issues with our site

that Lighthouse finds

To start up lighthouse, open up developer tools with Cmd+Option+I. Typically, Lighthouse can be found as one of the right most tabs within. If you don't see it right away, click the double arrow symbol and select it from the drop-down.

Then click 'Generate report':

14.1 Two Final SEO Quick Wins

Two quick wins we can get from Gatsby plugins are for creating a `robots.txt` file and a `sitemap.xml`. Lighthouse doesn't score for either of these, but they are important for SEO nonetheless.

Afterword

You've Done It!

Well, that was quite an adventure. We've both made it out alive! I hope you've found this book immensely useful, and that you're ready to refine your SaaS building skills even further.

Cheers! 

-Chris

Credits and Thanks

Credit where credit is due! (Note that I am not sponsored or supported by any of these platforms or individuals in anyway):

1. Netlify, for their awesome "feels like stealing" free tier
2. Bitbucket, for their great UI and tooling, including Bitbucket Pipelines
3. Digital Ocean, for the sheer ease of to start up a Linux instance with a few clicks
4. .NET, for just being an absolute joy of a framework to write and run code in
5. Jason Lengstorf, **@jlengstorf**, who ultimately was responsible in sending me down the Netlify / Identity / Stripe rabbithole with his free course video, **Sell Products on the JAMstack**
6. Josh W. Comeau, **@JoshWComeau** who also released a book independently which inspired me to do the

same (somewhat unrelated: I consider him my blog rival, though I suppose that feeling is not mutual 😂)

7. **Dabolus on DeviantArt**, for all of those juicy hi-res emoji PNGs that I've used generously throughout the book!

8. All my family and friends, who had to deal with my near daily spamming of PDF drafts of this book, probably overloading all their memory on all their devices. (It's addicting and too easy to do when you're working with LaTeX!)

Appendices

A

Installing Node.js and npm

Though the process of install Node.js has become ever more easy over the years, I still find it is a challenge to maintain the ever evolving versions of both Node.js and npm (and cheers to the team for producing releases so rapidly!)

I believe a tool like nvm is essential, as it manages and partitials all environment seperately, so you will never end up with a missing version or incompatible globally install module. You can also permanantly uninstall older version of node, or install newer ones with a one-liner CLI command.

B

Installing .NET 5.0

Head to **the official .NET download site to download .NET 5.0**. The site will attempt to detect your OS, but make sure that it is suggesting the correct one for your machine. The click the 'Download .NET SDK' button, and open it as soon as it downloads.

The resulting download file installer will include everything you need: the runtime, all intellisense, and the CLI command `dotnet`.

Index

About the Author

Figure 14.1.: Christopher Frewin

Christopher Frewin is a Senior Full Stack Developer with over 10 years of programming experience, the last 7 of which were in industry. When he's not writing code, building SaaS Products, or teaching full stack software engineering, he can be found doing any of the following: hiking, skiing, taking pictures, losing money on options, dropping into warzone with the boys, spoiling homebrew, writing music, and creating art. He is originally from Burnt Hills, New York, United States but currently resides in Feldkirch, Vorarlberg, Austria.