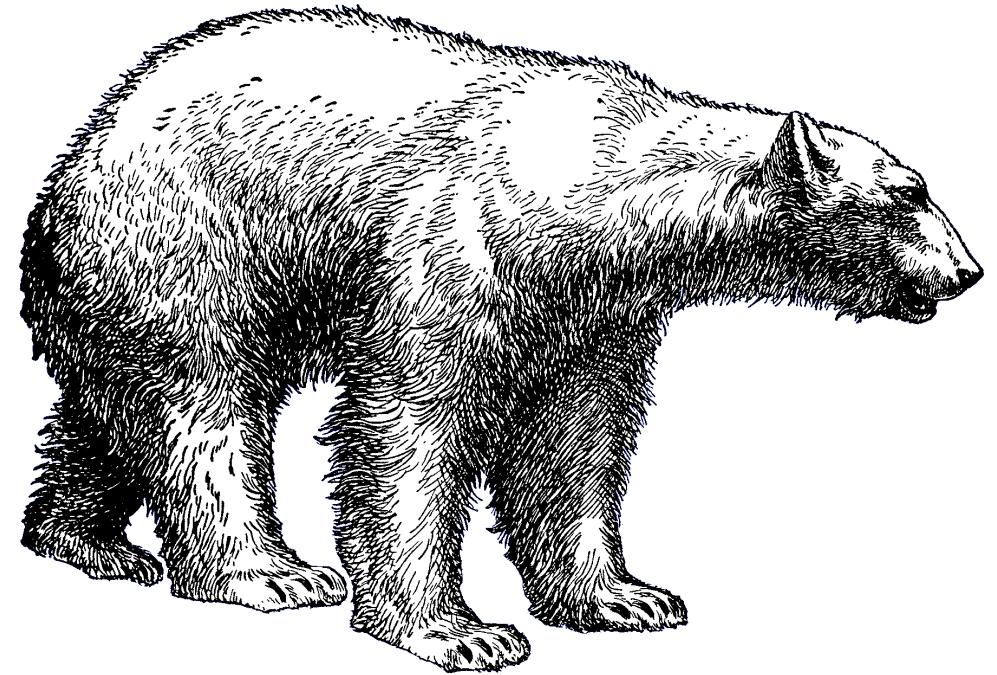


Build your own full stack SaaS product in days!



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 14, 2021

Contents

List of Figures	9
List of Listings	14
Foreword	15
1 The Product	21
1.1 The Product We'll Be Building	22
2 The Frontend - Getting Started	25
2.1 Introduction to the Frontend	25
2.2 Bootstrap the Frontend With Gatsby V3	28
2.3 Clean Up the Gatsby Default Starter	30
2.4 Setup a Bitbucket Repository for the Frontend	38
2.5 Use Netlify for the Frontend DevOps Framework	43
2.6 Add a Primary Domain to Netlify via Namecheap	49
3 The Frontend - Implementation	57
3.1 Running the Frontend via the Netlify CLI	58
3.2 Adding SCSS and Bootstrap as the Styling Framework	64
3.3 Creating React Components for Your Site's Layout	68

Contents

3.4	Creating an Interactive Code Editor Widget . .	76
3.5	Recipe: Creating a Production-Ready SVG . . .	89
3.6	Recipe: Adding API Helper Functions	104
3.7	Setting Up a Contract-Based API Call	105
3.8	Parsing the Redux State Interface	106
3.9	Adding Redux	108
3.10	Recipe: Toast Helper Functions	114
3.11	Add Netlify Identity as the Authentication and Authorization Platform	114
3.12	Adding Netlify State to Redux	115
3.13	Use Stripe for the First Payments Platform . .	116
3.14	Use Netlify Serverless Functions	116
3.15	Setup Fauna DB for User Management	116
3.16	Building an App Page	116
3.17	Review of the Frontend Implementation	118
4	The Backend - Getting Started	121
4.1	Introduction to the Backend	121
4.2	Bootstrap the Backend With the .NET CLI . . .	122
4.3	Clean Up the Backend Boilerplate Code	122
4.4	Setup a Bitbucket Repository for the Backend	122
4.5	Use Bitbucket Pipelines for the DevOps Frame- work	122
4.6	Create a Digital Ocean Droplet	123
4.7	Creating a Droplet	123
4.8	Using Secrets	125
5	The Backend - Implementation	127
5.1	Writing the First Endpoint for the Custom API	127
6	Building a Staging (or Testing) Environment	129
6.1	The Essential need for a Testing Environment .	130
6.2	Staging CI / CD for the Frontend	130

6.3 Create a Staging Branch for the Frontend . . .	130
6.4 Configure Netlify to Build According to the Staging Branch	131
6.5 Staging CI / CD for the Backend	131
6.6 Create a Staging Branch for the Backend . . .	131
7 The Frontend - Advanced Implementation	133
7.1 Dynamically Setting Animations	134
8 The Backend - Advanced Implementation	137
9 Recipe No. #1: Additional Payment Platform Integrations	139
9.1 Introduction	139
10 Recipe No. #2: Add Application-Wide Logging	141
11 Recipe No. #3: Adding Custom Emails	143
12 Recipe No. #4: Adding Automation	145
13 Recipe No. #5: SEO Optimization	147
13.1 Two Final SEO Quick Wins	148
Afterword	149
Credits	151
Index	153

List of Figures

2.1 Screenshot of the unmodified default Gatsby starter.	30
2.2 Screenshot of adding a repository on Bitbucket.	39
2.3 Screenshot of the 'create repository' on Bitbucket.	39
2.4 Screenshot of repository fields for your SaaS product.	41
2.5 Screenshot of the 'New site from Git' button.	43
2.6 Screenshot of the 'Bitbucket' button.	44
2.7 Screenshot of Netlify build settings for a Gatsby site.	45
2.8 Screenshot of where to find the detailed deploy log per deploy.	45
2.9 Screenshot of the 'Domain settings' button.	46
2.10 Screenshot of the 'Edit site name' domain option.	47
2.11 Screenshot of the 'Edit site name' domain option.	47
2.12 Screenshot of the 'Set up a custom domain' domain step.	50
2.13 Screenshot of the custom domain input.	51
2.14 Screenshot of the DNS warnings on the custom domains.	51
2.15 Screenshot of the 'Use Netlify DNS link'.	52

List of Figures

2.16	Screenshot of the final step in the Netlify DNS setup, 'Activate Netlify DNS'.	53
2.17	Screenshot of the nameservers dropdown on the Namecheap site dashboard.	54
2.18	Screenshot of the Netlify nameservers applied to the custom DNS configuration on Namecheap.	54
3.1	Screenshot of the Netlify environment variables panel and the 'Edit variables' button.	61
3.2	Screenshot of the opened Netlify Environment variables panel, with it's key-value style interface. Here we are adding the <code>NODE_VERSION</code> variable.	62
3.3	Screenshot of the terminal output after running the Netlify CLI in development mode. The environment variable <code>NODE_VERSION</code> we defined in the Netlify UI is being used in our local environment.	63
3.4	Screenshot of the terminal output after running the Netlify CLI in development mode. The environment variable <code>NODE_VERSION</code> we defined in the Netlify UI is now being ignored, as the local <code>NODE_VERSION</code> defined in process takes precedent.	64
3.5	Screenshot of the single tab code editor.	84
3.6	Screenshot of the multi tabbed code editor.	84
3.7	The square plate-like logo for ReduxPlate.	90
3.8	Screenshot of the sidebar options in SVGOMG.	91
3.9	Screenshot of the sidebar options in SVGOMG.	92
3.10	Screenshot of the option buttons in SVGOMG (Background toggle, copy to clipboard, and export).	93

3.11 Screenshot of the logo in the nav.	99
3.12 Screenshot of the logo as a favicon.	99
3.13 A screenshot of the homepage we've built so far.	102
4.1 Screenshot of the droplets tab.	124
4.2 Screenshot of the new droplet button.	124

List of Listings

1	Installing Gatsby via npm.	29
2.1	Creating a new Gatsby project from gatsby-starter-default.	29
2.2	Moving into frontend project directory.	29
2.3	Starting develop mode via npm.	29
2.4	A baasic SEO React component.	32
2.5	Basic README for the frontend project	33
2.6	An example MIT license for the frontend project.	34
2.7	Modifying the license field in package.json	36
2.8	Directory tree after initial cleanup of the Gatsby default stater.	36
2.9	Modifying the git repository fields in package.json	42
2.10	Setting the git original URL to the new Bitbucket repository.	42
2.11	Adding committing and pushing all code changes to the remote repository.	42
2.12	Example successful deploy log output in the Netlify UI.	46
3.1	Installing the Netlify CLI via npm.	59
3.2	Logging in to Netlify via the Netlify CLI.	59

3.3 Linking the frontend project with Netlify via the Netlify CLI.	60
3.4 Example of overriding the <code>NODE_VERSION</code> environment variable in a terminal.	63
3.5 Installing the sass and gatsby-plugin-sass packages via npm.	65
3.6 Adding gatsby-plugin-sass to gatsby-config.js .	65
3.7 Installing Bootstrap via npm.	66
3.8 Importing Bootstrap Sass into styles.scss . . .	66
3.9 Initial creating of <code>_variables.scss</code>	67
3.10 Adding <code>_variables.scss</code> to styles.scss	67
3.11 Importing styles.scss into gatsby-browser.js. .	68
3.12 The contents of Nav.tsx.	68
3.13 Adding the Nav component to Layout.tsx. . . .	70
3.14 The contents of Footer.tsx	70
3.15 Adding Footer.tsx to Layout.tsx.	71
3.16 The contents of Header.tsx.	71
3.17 The contents of header.module.scss.	72
3.18 The contents of PlateWidget.tsx.	73
3.19 The contents of plate-widget.module.scss. . . .	74
3.20 The IEditorSettings interface.	77
3.21 The IEditorWidgetProps interface.	78
3.22 State management in EditorWidget.tsx	78
3.23 Rendering an Ace editor in EditorWidget.tsx .	79
3.24 Rendering Bootstrap styled nav tabs in Editor-Widget.tsx	80
3.25 The two helper functions <code>onChangeCode</code> and <code>onChangeTab</code> in EditorWidget.tsx	80
3.26 Adding an on mount side effect to move the cursor of the editor to the start of the code. . .	81
3.27 The full contents of EditorWidget.tsx	82
3.28 The full contents of TryItWidget.tsx.	85

3.29	The full contents of <code>TryItButtons.tsx</code>	86
3.30	Adding additional bootstrap variables to <code>_variables.scss</code>	87
3.31	The contents of <code>Home.tsx</code>	88
3.32	SVG markup as returned by <code>SVGOMG</code>	94
3.33	Final SVG markup for the application's logo.	96
3.34	Modifying the path for the icon in <code>gatsby-plugin-manifest</code>	97
3.35	Modifying the path for the nav component logo.	98
3.36	Animation styles for the logo component.	100
3.37	The contents of <code>Logo.tsx</code>	101
3.38	Directory tree of further expanded frontend.	103
3.39	The series of API Helper functions <code>ApiHelpers.ts</code>	105
3.40	The initial shape of interface <code>IGenerateOptions</code>	106
3.41	The initial shape of interface <code>IGenerateOptions</code>	106
3.42	Installing the <code>ts-morph</code> package.	107
3.43	The contents of <code>ASTHelpers.ts</code>	107
3.44	Installing <code>redux react-redux</code> and <code>@reduxjs/toolkit</code>	108
3.45	The contents of <code>wrap-with-provider.js</code>	108
3.46	Adding <code>wrapWithProvider</code> to <code>gatsby-ssr.js</code>	109
3.47	Adding <code>wrapWithProvider</code> to <code>gatsby-browser.js</code>	109
3.48	The 'editors' slice of the Redux state <code>editorsSlice.ts</code>	110
3.49	The <code>TryItWidget</code> component after refactoring the frontend application to use Redux	112
3.50	The <code>EditorWidget</code> component after refactoring the frontend application to use Redux.	112
3.51	Installing the <code>netlify-identity-widget</code> via <code>npm</code>	115
3.52	The contents of an initial <code>App.tsx</code>	117
3.53	The contents of our new app page component <code>app.tsx</code>	117

List of Listings

4.1	Creating the bitbucket-pipelines.yml file. . . .	123
7.1	The contents of useSSRSafeWindowLocation.ts	134
7.2	The contents of useShouldAnimate.ts	134

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 chronological 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

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! It's a one stop shop for Redux code management.

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

2 The Frontend - Getting Started

- » 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

2 The Frontend - Getting Started

- » 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, `\index{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, `\index{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:

2 The Frontend - Getting Started

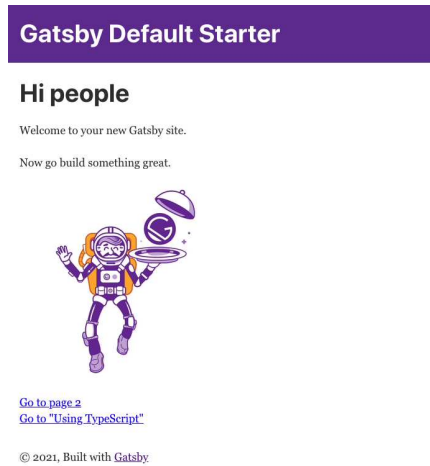


Figure 2.1: Screenshot of the unmodified default Gatsby starter.

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 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 plugin and delete the comments about it. Also be sure to update the values under the **:** 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:

2 The Frontend - Getting Started

Listing 2.4: `</>`

`Seo.tsx`

2.3 Clean Up the Gatsby Default Starter

```
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.
```

```
    siteMetadata.description || ""}
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
```

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

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

2 The Frontend - Getting Started

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

2.3 Clean Up the Gatsby Default Starter

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 appro-

2 The Frontend - Getting Started

priately 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 ready to start actually coding and building our frontend. We've bootstrapped our project with the Gatsby CLI. We've edited our **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

We've reached the first milestone repository of this book: **the skeleton Gatsby repository which we've just fin-**

ished 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:

2.4 Setup a Bitbucket Repository for the Frontend

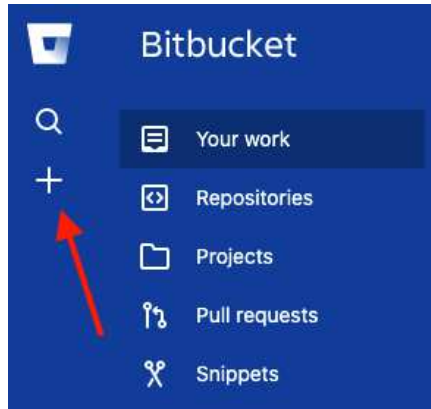


Figure 2.2: Screenshot of adding a repository on Bitbucket.

then select 'Create' > 'Repository':

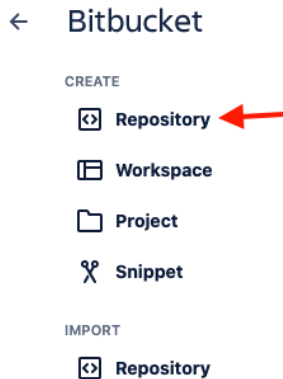


Figure 2.3: Screenshot of the 'create repository' on Bitbucket.

On the resulting page, apply the following:

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

2 The Frontend - Getting Started

- » 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:

2.4 Setup a Bitbucket Repository for the Frontend

Create a new repository [Import repository](#)

Workspace Chris Frewin ▼

Project name* ReduxPlate ✕

Repository name* reduxplate.com

Access level ☒ Private repository
Uncheck to make this repository public. Public repositories typically contain open-source code and can be viewed by anyone.

Include a README? No ▼

Default branch name e.g., 'main'

Include .gitignore? No ▼

[> Advanced settings](#)

[Create repository](#) [Cancel](#)

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 reflects the new

2 The Frontend - Getting Started

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/princefishthrower/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@bitbucket.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:

2 The Frontend - Getting Started

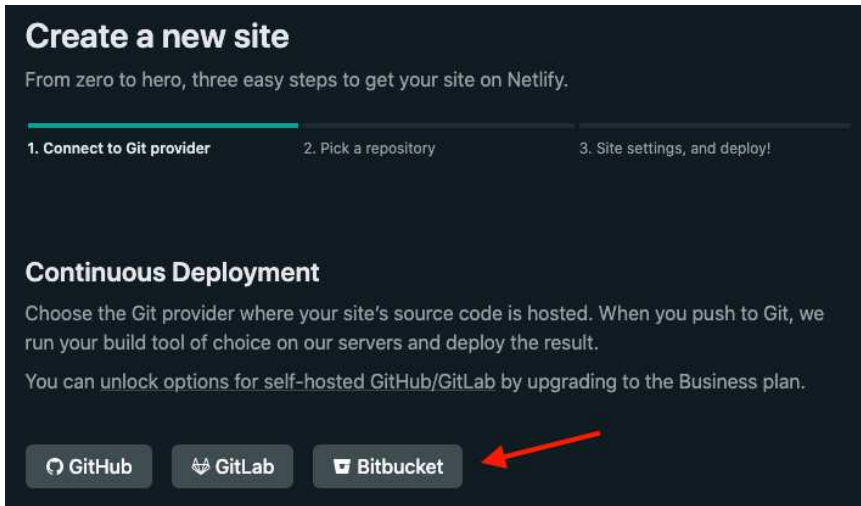


Figure 2.6: Screenshot of the 'Bitbucket' button.

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/**:

2.5 Use Netlify for the Frontend DevOps Framework

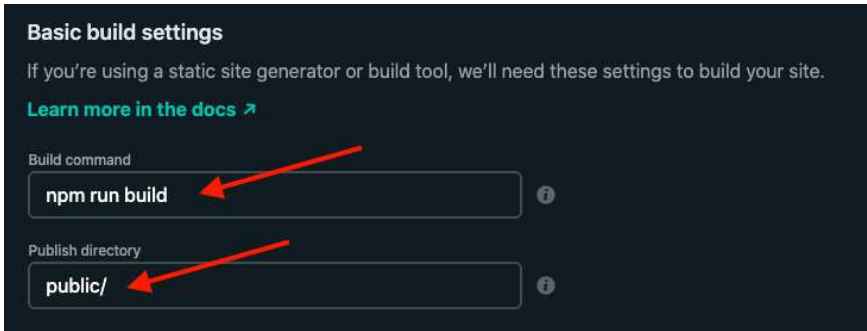


Figure 2.7: Screenshot of Netlify build settings for a Gatsby site.

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):

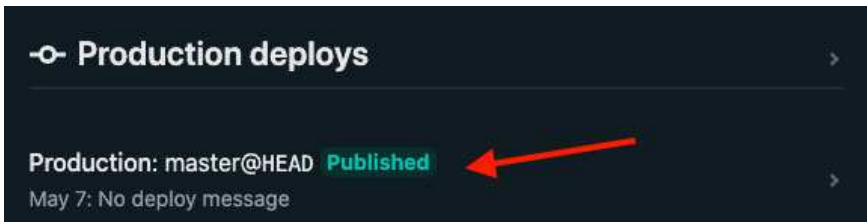


Figure 2.8: Screenshot of where to find the detailed deploy log per deploy.

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:

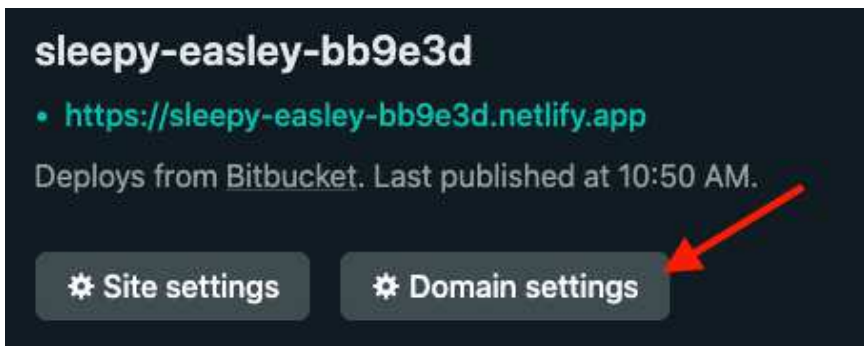


Figure 2.9: Screenshot of the 'Domain settings' button.

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

2.5 Use Netlify for the Frontend DevOps Framework

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:

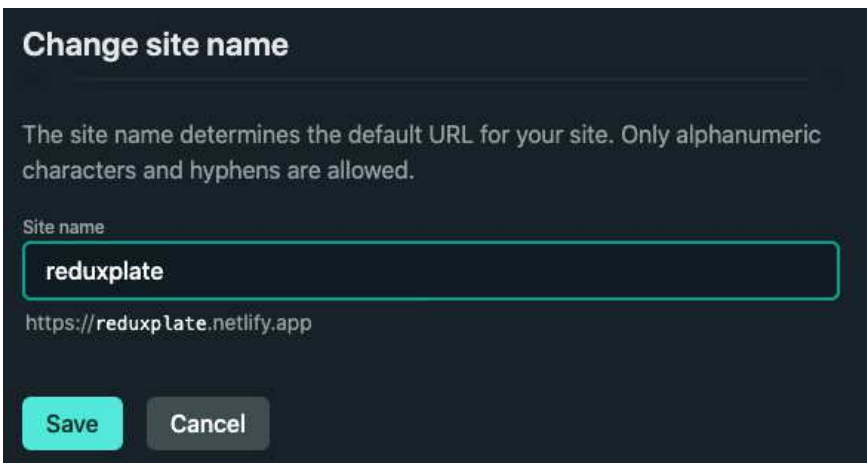


Figure 2.11: Screenshot of the 'Edit site name' domain option.

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

A Word On Netlify

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*. It 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:

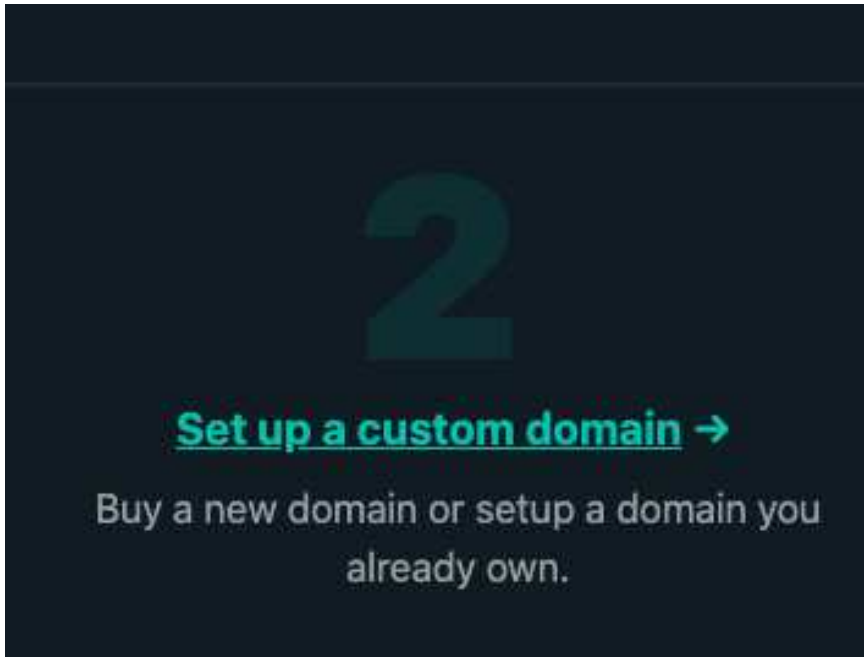
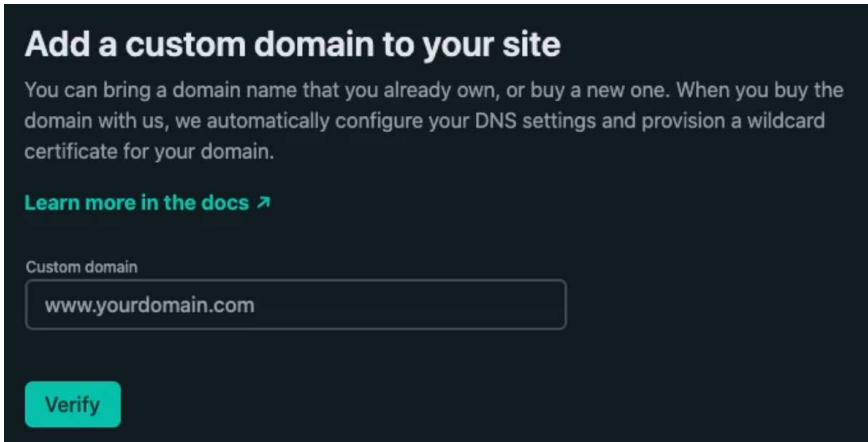


Figure 2.12: Screenshot of the 'Set up a custom domain' domain step.

in the resulting screen, provide your domain name:

2.6 Add a Primary Domain to Netlify via Namecheap



Add a custom domain to your site

You can bring a domain name that you already own, or buy a new one. When you buy the domain with us, we automatically configure your DNS settings and provision a wildcard certificate for your domain.

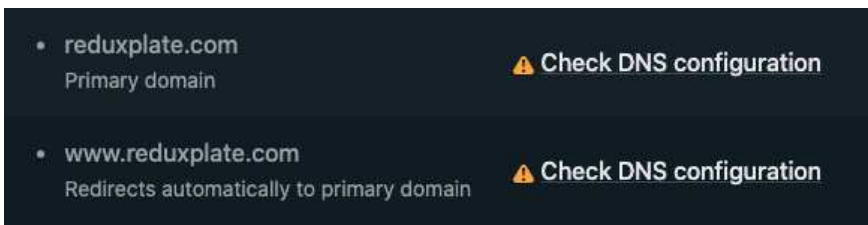
[Learn more in the docs ↗](#)

Custom domain

[Verify](#)

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:



- **reduxplate.com**
Primary domain ⚠️ [Check DNS configuration](#)
- **www.reduxplate.com**
Redirects automatically to primary domain ⚠️ [Check DNS configuration](#)

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

2 The Frontend - Getting Started

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>':

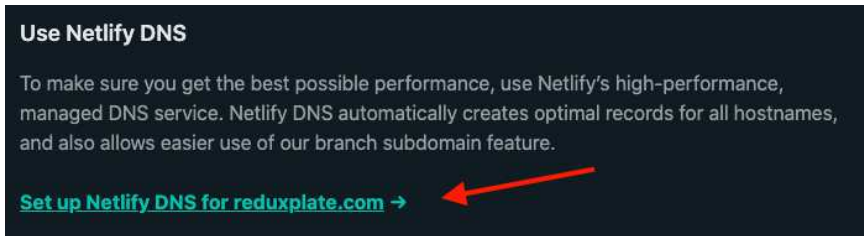


Figure 2.15: Screenshot of the 'Use Netlify DNS link'.

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:

2.6 Add a Primary Domain to Netlify via Namecheap

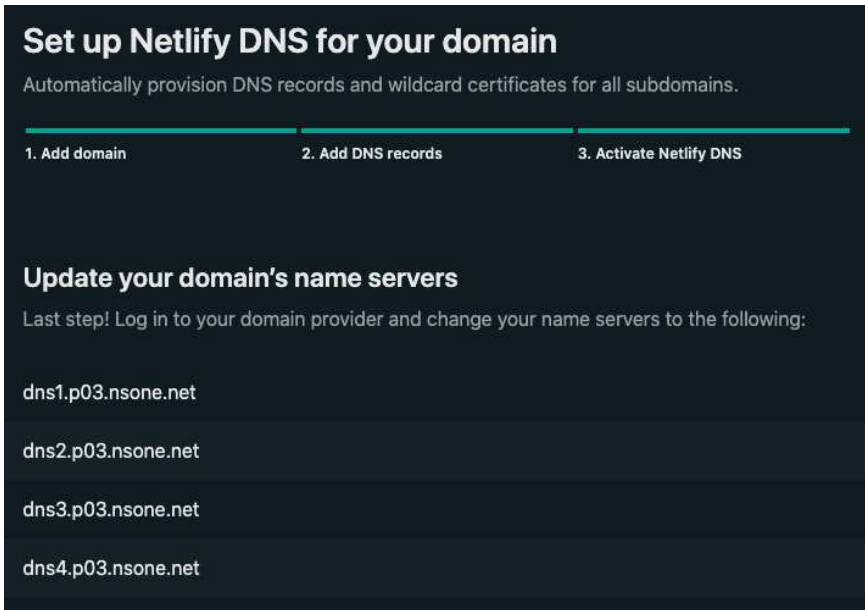


Figure 2.16: Screenshot of the final step in the Netlify DNS setup, 'Activate Netlify DNS'.

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:

2 The Frontend - Getting Started



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

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 the 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:

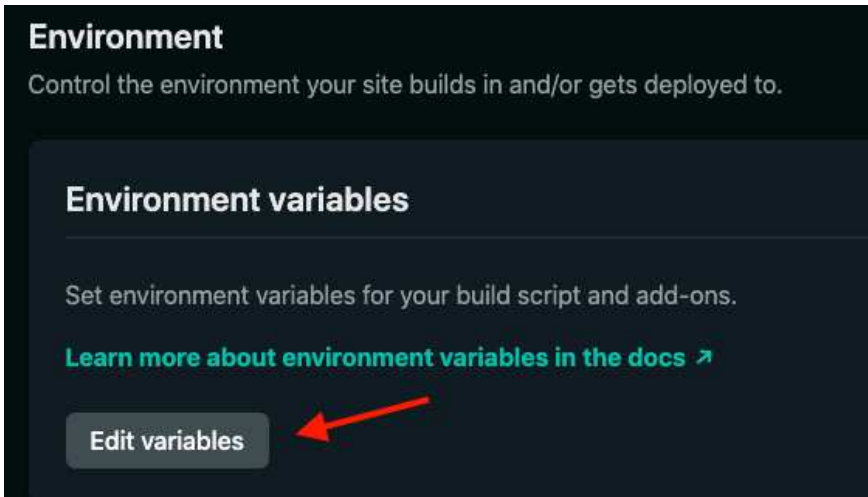


Figure 3.1: Screenshot of the Netlify environment variables panel and the 'Edit variables' button.

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 its value as **14.16.0**, and click 'Save':

3 The Frontend - Implementation

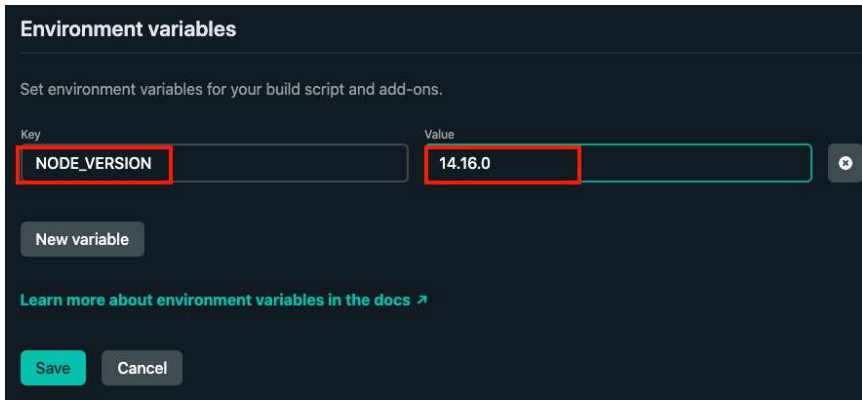


Figure 3.2: Screenshot of the opened Netlify Environment variables panel, with it's key-value style interface. Here we are adding the `NODE_VERSION` variable.

⚠ 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:

A terminal window with a dark background. The text "Injected build settings env var: NODE_VERSION" is displayed in a light blue/cyan monospace font. A small diamond icon is visible at the start of the line.

```
◆ Injected build settings env var: NODE_VERSION
```

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

A terminal window with a light gray background. The text "export NODE_VERSION=14.15.0" is displayed in a green monospace font.

```
export NODE_VERSION=14.15.0
```

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

3 The Frontend - Implementation

```
✦ Ignored build settings env var: NODE_VERSION (defined in process)
```

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 precedent.

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 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 `index.html` and `styles.css`. 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 plugin:

Listing 3.5: `</>`

terminal

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

also include the 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:

3 The Frontend - Implementation

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**), and think that the Montserrat font 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:

3.2 Adding SCSS and Bootstrap as the Styling Framework

Listing 3.9: `</>`

`_variables.scss`

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

Listing 3.12: `</>`

`Nav.tsx`

3.3 Creating React Components for Your Site's Layout

```
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 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**!

3 The Frontend - Implementation

Be sure to then import and use the `<Nav/>` component in :

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 , create a file under . Add this code to it:

Listing 3.14: `</>`

Footer.tsx

```
import * as React from "react"

export function Footer() {
  return (
    <footer className="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>
  )
}
```

3.3 Creating React Components for Your Site's Layout

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 , 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 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, can be replaced with the following code:

Listing 3.16: `</>`

`Header.tsx`

3 The Frontend - Implementation

```
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

3.3 Creating React Components for Your Site's Layout

```
@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

3 The Frontend - Implementation

```
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

3.3 Creating React Components for Your Site's Layout

3 The Frontend - Implementation

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 **react-ace** package, which provides an **AceEditor** component from the **ace** package. We can begin to imagine for this 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.

Ultimately, I decided on the name of **EditorWidget** for the component. Create a new folder called **widgets** under **src/componentnts/**, and create a new TypeScript React component file called **EditorWidget.tsx**.

Props for <EditorWidget/>

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

No matter what style you decide, Visual Studio Code's Intellisense won't care 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. Then, as we'll see below, we will need a ref for the editor to make some changes with it.

Listing 3.22: </>

EditorWidget.tsx

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

That's all the React state and ref 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 **react-ace** package makes rendering the editor part of our **EditorWidget** component rather easy:

Listing 3.23: `</>`

EditorWidget.tsx

```
return (
  ...
  <AceEditor
    className={styles.editor}
    ref={aceEditorRef}
    mode="typescript"
    theme="kuroir"
    onChange={onChangeCode}
    showPrintMargin={true}
    showGutter={true}
    highlightActiveLine={true}
    name={editorTitle.split('
').join('-').toLowerCase()}
    value={editorSetting.code}
    setOptions={{
      enableBasicAutocompletion: true,
      enableLiveAutocompletion: true,
      enableSnippets: false,
      showLineNumbers: true,
      tabSize: 2,
    }}
  />
  ...
)
```

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

3 The Frontend - Implementation

Listing 3.24: `</>`

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 3.25: `</>`

EditorWidget.tsx

3.4 Creating an Interactive Code Editor Widget

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

Finally, as a micro-optimization, using the **useEffect** hook, we can add a side effect that when **<EditorWidget/>** mounts, we can put the editor cursor location at the beginning of the file - the default is the very end, which feels a bit weird. This is where we will employ our **ref** for the Ace editor:

Listing 3.26: **</>**

EditorWidget.tsx

3 The Frontend - Implementation

```
// on mount: go to the top left of the editor  
useEffect(() => {  
  const editor = aceEditorRef.current.editor  
  editor.gotoLine(0, 0, false)  
}, [])
```

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

Listing 3.27: `</>`

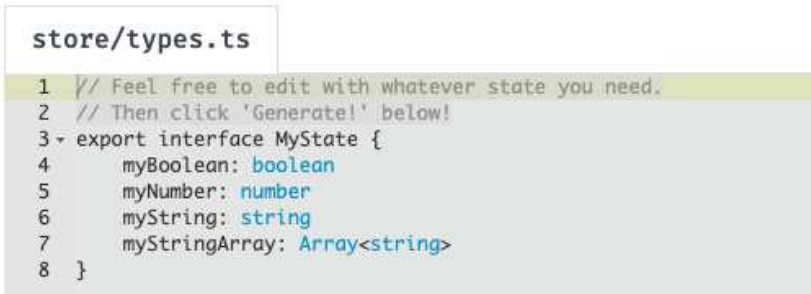
EditorWidget.tsx

3.4 Creating an Interactive Code Editor Widget

```
import React from "react"
import AceEditor from "react-ace"
import "ace-builds/src-noconflict/mode-typescript"
import "ace-builds/src-noconflict/theme-kuroir"
import { useEffect, useRef, useState } from "react"
import IEditorSetting from 83
    "../.././././interfaces/IEditorSettings"
import * as styles from
```

3 The Frontend - Implementation

to generate looks like this:



```
store/types.ts
1 // Feel free to edit with whatever state you need.
2 // Then click 'Generate!' below!
3 export interface MyState {
4   myBoolean: boolean
5   myNumber: number
6   myString: string
7   myStringArray: Array<string>
8 }
```

Figure 3.5: Screenshot of the single tab code editor.

It's just the single **store/types.ts** file, so we see that one tab and the one source. A traditional generation output would ultimately result in three files, two in addition to the **types.ts**, a **actions.ts** and **reducer.ts** file (at least when not using **@reduxjs/toolkit**). Such a config results in our **EditorWidget** to look like this:



```
store/types.ts  store/reducer.ts  store/actions.ts
1 // Feel free to edit with whatever state you need.
2 // Click 'Generate!' below!
3 export interface MyState {
4   myBoolean: boolean
5   myNumber: number
6   myString: string
7   myStringArray: Array<string>
8 }
```

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 3.28: `</>`

TryItWidget.tsx

3 The Frontend - Implementation

```
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: "store/types.ts",
              code: `// Feel free to edit with
                whatever state you need.
// Then click 'Generate!' below!
export interface MyState {
  myBoolean: boolean
  myNumber: number
  myString: string
  myStringArray: string[]
}
isActive: true
`,
            },
          </EditorWidget>
          <TryItButtons />
        </div>
      </div>
    )
  }
}
```

It's two of our `<EditorWidget>` components side by side in a 'flex box'. I then 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:

```
fileLabel: "store/types.ts",
code: `// types.ts
// Nothing here yet.
// Click 'Generate!' below!`,
isActive: false
```

Listing 3.29: `</>`

`TryItButtons.tsx`

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

Listing 3.30: `</>`

`_variables.scss`

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


3 The Frontend - Implementation

I then decided to use the class **btn-outline-primary** on the ‘Generate!’ button instead of **btn-primary** as it seems to fit the style of the home page a bit better. The call to action button, ‘Try Full App’, retains the ‘primary’ styling, however.

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 3.31: **</>**

Home.tsx

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

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

3.5 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 weapon of choice.)
2. Load it into the amazingly powerful and useful SV-GOMG
3. Convert the SVG markup to JSX syntax using the HTML to JSX Compiler

3 The Frontend - Implementation

4. Look at the code - there are usually some manual steps that can be taken to even *further* simplify the SVG
5. Create a Style and consider dynamic capabilities of your vector

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 shapes, as these paths can be joined and optimized with a tool like SVGOMG, as we will see 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.

i A word on SVGOMG i

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:

Precision



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:

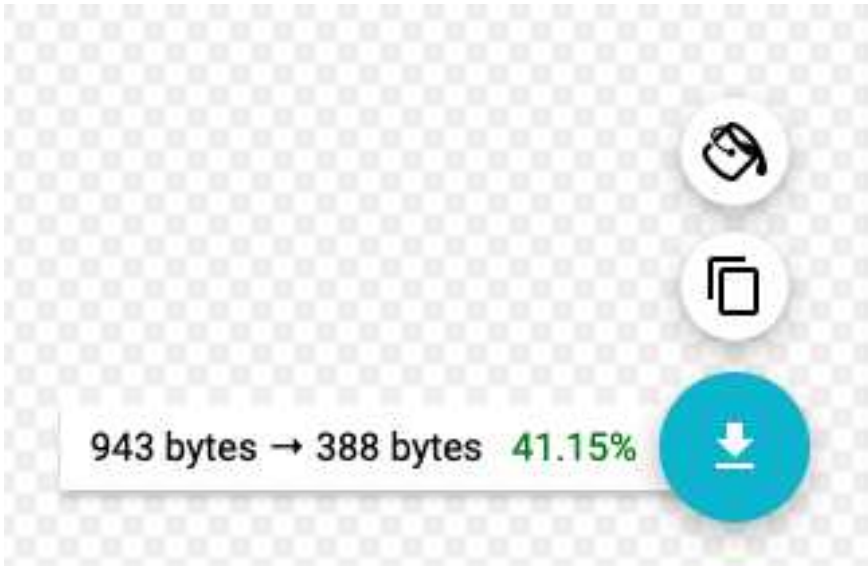


Figure 3.10: Screenshot of the option buttons in SVGOMG (Background toggle, copy to clipboard, and export).

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:

3 The Frontend - Implementation

Listing 3.32: `</>`

`logo.svg`

3.5 Recipe: Creating a Production-Ready 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 SVG-GOMG has produced a group node (i.e. `<g paint-order="fill markers stroke">`) around all

3 The Frontend - Implementation

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 3.33: `</>`

logo.svg

3.5 Recipe: Creating a Production-Ready 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 in **gatsby-config.js**:

Listing 3.34: </>

gatsby-config.js

3 The Frontend - Implementation

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

```

Luckily, the 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 :

Listing 3.35: </>

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:

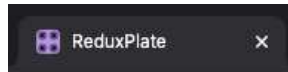


Figure 3.12: Screenshot of the logo as a favicon.

For the logo being displayed in the nav, I'd like to do a little something extra and fancy.

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 . Let's animate the screws in our logo to rotate continuously.

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

3 The Frontend - Implementation

First we'll utilize a tool to ensure our SVG markup is compatible with react. In the case of but if your SVG is more complex, React will not recognize the various hyphenated properties (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 snags like `class` to `className`.

unfortunately, the HTML to JSX Compiler produces the old school `React.createClass()` style markup, but that's fine, we only need to copy the contents of the `return()` statement. After copying that, create a new folder under `src/` folder called `utils`, and then a file `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 3.36: `</>`

`logo.module.scss`

3.5 Recipe: Creating a Production-Ready SVG

```
.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);
  }
}
```

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

Add the following to **Logo.tsx**:

Listing 3.37: </>

Logo.tsx

```
transform: rotateZ(-360deg);
```

Within **Nav.tsx**, we'll now replace the **<StaticImage/>** with our **<Logo/>** component.

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

3 The Frontend - Implementation

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:

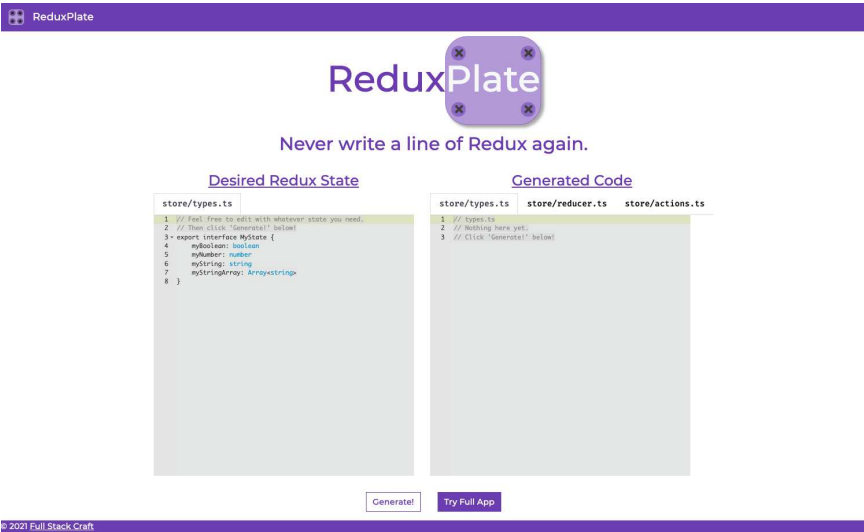


Figure 3.13: A screenshot of the homepage we’ve built so far.

Review of Initial Components and Layout

✓ Milestone #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:

3.5 Recipe: Creating a Production-Ready SVG

Listing 3.38: `</>`

terminal

3 The Frontend - Implementation

```
.
├── 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
```

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.

3.6 Recipe: Adding API Helper Functions

We will now add a series of API Helper functions which I believe are as optimized as it can be. They are both as flexible as possible in terms of typings, but able to be called from any React component as a one-liner, *including* side effect callbacks. This set of functions will be completely standalone, so you can reuse it for any SaaS product. Thus they earn their spot as the first official recipe listing in this

```
├── TryItWidget.tsx
├── utils
│   ├── Logo.tsx
│   └── widgets
│       ├── PlateWidget.tsx
│       ├── Images
│       │   ├── Logo.svg
│       └── interfaces
│           └── IEditorSettings.ts
├── pages
│   ├── 404.tsx
│   ├── app.tsx
│   └── index.tsx
```

book:

Listing 3.39: </>

ApiHelpers.ts

3.7 Setting Up a Contract-Based API Call

As we saw in our API Helper Functions, 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. Even later, we'll see even in the free preview of the **app/** page, we will limit features available. In any case, it's clear that the **/generate** endpoint will 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 an interface for the POST body. As we build complexity and features to our SaaS product, we can always return to this contract and extend it.

For starting off this contract, we'll need a few boolean flag options, and then the properties of the Redux state that we will be generating code for.

Creating the IGenerateOptions Interface

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

Listing 3.40: </>

IGenerateOptions.ts

```
export default IGenerateOptions {  
  typeScriptProperties: Array<ITypeScriptProperty>  
  useReduxToolkit: boolean;  
  useTypeScript: boolean;  
  singleFile: boolean;  
}
```

The interface **ITypeScriptProperty** is yet another new interface that we need to define. Create a new file **ITypeScriptProperty.ts** under the **src/interfaces/** folder, and add this to it:

Listing 3.41: </>

ITypeScriptProperty.ts

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

This is the type that will hold our array of properties from the state. It should be all we need from the client to generate a fully working 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. The state code itself is already public and editable , so parsing out the types

3.8 Parsing the Redux State Interface

We will be parsing out the names and types of the state interface using the package . **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 and be on our way. As a bonus, we'll be able to handle and show our SaaS customers any syntax or typing errors they may make - directly in the client!

Install **ts-morph**

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

Listing 3.42: </>

terminal

```
npm install ts-morph
```

Then, create a new folder called **helpers**, and create the a file called **ASTHelpers.ts**. This file holds a helper function **parseTypeScript** which accepts a string of TypeScript code and returns our desired **Array<ITypeScriptProperty>**:

Listing 3.43: </>

ASTHelpers.ts

If we go to our 'Generate!' button and try to call this function, we'll run into an annoying issue. We don't immediately have access to . Getting to it in a traditional React way would involve a variety of parent to child and vice versa callbacks, which is not readable, or maintainable. It's time we introduce Redux into the app, with a slice of state specifically for storing the state of this 'Try It' widget.

3.9 Adding Redux

We saw in the previous section that when we went to add the call to our **parseTypeScript** function, that we didn't have easy access to the current value of what the code was in the editor. To do this in a clean and maintainable way, we will at Redux to our 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 itself, for a variety of react style hooks to use redux in our components, and finally , for Redux toolkit, which helps making slices of state a breeze. All together, this install with **npm** is:

Listing 3.44: </>

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. First start by creating a **JavaScript** file in the project root called **wrap-with-provider.js**, and add the following:

Listing 3.45: </>

wrap-with-provider.js

```
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 3.46: `</>` `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 3.47: `</>` `gatsby-browser.js`

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

export const wrapRootElement = wrapWithProvider
```

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:

3 The Frontend - Implementation

Listing 3.48: `</>`

`editorsSlice.ts`

I've moved most of the `editorSettings` update logic into the reducer action logic. 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/>`

```

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

interface EditorsState {
  editors: {
    [Editor: string]: {
      editorTitle: string
      editorSettings: Array<EditorSetting>
    }
  }
}

```


3 The Frontend - Implementation

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 component now looks like this:

Listing 3.49: `</>`

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 3.50: `</>`

EditorWidget.tsx

Let's now finally hook in a call to our API (which doesn't exist yet, but we're getting very close to a point where we can start writing it). For the 'Generate!' button which triggers a call to our endpoint `/generate/`, we can actually hard code the settings for `GenerateOptions`, since this is the totally free and public version of the endpoint, and the user won't have access to the more advanced options. Adding a `useSelector` hook, we can now get at the current value of the editor's code within the `TextAreaButtons/` component. We can then submit this, along with the hardcoded values,

```

import * as React from "react"
import AceEditor from "react-ace"
import "ace-builds/src-noconflict/mode-typescript"
import "ace-builds/src-noconflict/theme-kuroir"
import { useEditor } from "react-ace"
import * as styles from "
  .. / .. / styles/modules/editor_module.scss"
import { useDispatch, useSelector } from "react-redux"
import Editor from "
  .. / .. / enums/Editor"
import { CodeChanged, EditorClicked } from "
  .. / .. / enums/Editor"

export interface IEditorWidgetProps {
  editor: Editor
}

export function EditorWidget(113 props:
  IEditorWidgetProps) {
  const { editor } = props

```

to the **post** function from ApiHelpers!

3.10 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

3.11 Add Netlify Identity as the Authentication and Au-

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 :

Listing 3.51: </>

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 package.

3.12 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 utility 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 example 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 opt for the **Object.values()** of the available roles, and compare them against every role with **roles.find()**.

3.13 Use Stripe for the First Payments Platform

Chapter Objectives

- » Setting up Stripe to accept subscriptions

3.14 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.

3.15 Setup 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.

3.16 Building an App Page

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/>`, which we anyway removed for our fancy logo SVG. In this case, we're finally going to use a powerful feature of Gatsby - to build a new static page under `/page`. The Gatsby core automatically turns any React component found in `src/pages` into a corresponding 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 3.52: **</>**

App.tsx

```
import * as React from "react"

export function App() {
  return <h1>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 3.53: **</>**

app.tsx

3 The Frontend - Implementation

```
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 home-page, 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.

3.17 Review of the Frontend Implementation

Great work so far! In this section, we’ve:

- » added custom styling and theming to our website via Bootstrap
- » added a useful ApiHelpers file, allowing for API calls including callbacks for customer feedback and error handling
- » added a ToastHelpers file, which provides some powerful utility functions for the **react-toastify** package
- » added authentication, authorization, via Netlify Identity
- » added automatic deploys to our live custom URL every time we push to the **master** branch
- » created a page under the /app path

Right now, our app *looks* really solid when viewed in a browser. But there's one major problem - our product doesn't actually *do* anything yet! 😂 Remember those calls to the **/generate** endpoint? Yeah. That endpoint doesn't exist yet - our toast will continually

The next step then is to build this **/generate** endpoint on our custom API and call it from the frontend to ensure that the full stack is working. 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

Start off using the

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.

4.4 Setup a Bitbucket Repository for the Backend

Just as we used Bitbucket for the frontend, we will do the same for the backend.

4.5 Use Bitbucket Pipelines for the DevOps Framework

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

Listing 4.1: `</>`

terminal

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

Great. It appears that our pipeline is working, but we have nowhere to send it to! In the next chapter, we'll configure a Digital Ocean droplet with Ubuntu 20.04, PostgreSQL 13.2, and .NET 5.0.

4.6 Create a Digital Ocean Droplet

Our custom .NET API will live on our Digital Ocean Droplet.

4.7 Creating a Droplet

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:

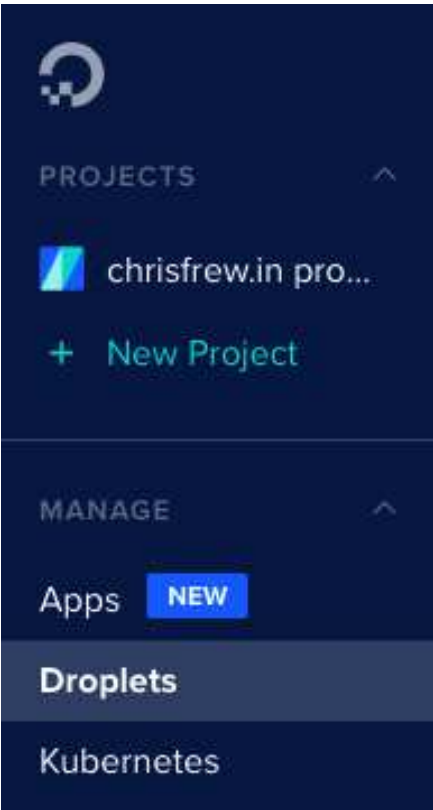


Figure 4.1: Screenshot of the droplets tab.

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 closest 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 of this book I will be using **redux-plate**

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:

So, our Digital Ocean Droplet is spooled up and 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!

4.8 Using Secrets

Secret keeping is always a major discussion when it comes to production environments. To handle our app secrets like our PostgreSQL connection string, we'll be using. SOMETHING

5

The Backend - Implementation

5.1 Writing the First Endpoint for the Custom API

Great. Our API is up and running, and, like the frontend, will automatically deploy upon pushing to the **master branch**. Typically, when I get to this point, as a sanity check, I create what I call a **Root** controller which just returns some plain text of an API version string, or really whatever you'd like to have.

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 continuous integration tool for the frontend is Netlify, and with the backend

6 Building a Staging (or Testing) Environment

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 its 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 off 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.

7

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 frontend that will be discussed in this book will be advanced features of the App page.

7.1 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 7.1: </>

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 7.2: </>

useShouldAnimate.ts

7.1 Dynamically Setting Animations

```
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.

8

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.

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! 🚀

9

Recipe No. #1: Additional Payment Platform Integrations

9.1 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-

9 Recipe No. #1: Additional Payment Platform Integrations

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

10

Recipe No. #2: Add Application- Wide Logging

11

Recipe No. #3: Adding Custom Emails

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

12

Recipe No. #4: Adding Automation

13

Recipe No. #5: 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 Lighthouse, 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':

13.1 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 any way):

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 rabbit hole 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!)

Index

Footer.tsx, 70
Header.tsx, 64, 71
Layout.tsx, 64, 69, 70
Nav.tsx, 68, 70, 71, 98
TryItWidget, 112
\at reduxjs/toolkit,
108
gatsby-plugin-manifest,
30, 97, 98
gatsby-plugin-offline,
30
gatsby-plugin-sass, 64,
65
netlify-identity-widget,
115
package.json, 30, 35, 37,
41
react-redux, 108
redux, 108
ts-morph, 106
Redux, 22
TypeScript, 22