FULLSTACK REACT WITH TYPESCRIPT

Learn Pro Patterns for Hooks, Testing, Redux, SSR, and GraphQL

MAKSIM IVANOV
ALEX BESPOYASOV

Fullstack React with TypeScript

Learn Pro Patterns for Hooks, Testing, Redux, SSR, and GraphQL

Written by Maksim Ivanov and Alex Bespoyasov
Edited by Nate Murray

Published by \newline

Contents

Introduction **1**
How To Get The Most Out Of This Book 1
What is TypeScript 5
Why Use TypeScript With React 8
A Necessary Word Of Caution 10

Your First React and TypeScript Application: Building Trello with Drag and Drop **11**
Introduction 11
What Are We Building? 12
Prerequisites 12
Preview The Final Result 13
How to Bootstrap React + TypeScript App Automatically 15
App Layout. React + TypeScript Basics 28
Remove The Clutter 28
Add Global Styles 30
How To Style React Elements 31
Using Separate CSS Files 31
Passing CSS Rules Through Style Property 32
Using External Styling Libraries 33
Prepare Styled Components 34
Install `styled-components`. Working with `@types` packages 34
Break the UI into components 35
Render Everything Together 38
Create Column Components 39
How to define props 40
How to accept `children` prop 41

Create Card Components 44
Render everything together 44
Component For Adding New Items 45
Styles For The Button 46
Create AddNewItem Component. Using State 47
Adding New Lists 49
Adding New Tasks 50
NewItemForm component 50
Styles For The Form 50
Create NewItemForm component 52
Automatically focus on input 54
Create the `useFocus` hook 54
Use the `useFocus` hook 57
Submit on enter 58
Add Global State And Business Logic. Using the useReducer 60
Using the useReducer 60
Implement Global State 67
Hardcode the data 67
Define the Context 69
Define the Context provider 69
Define the business logic 74
Create Actions 74
Define the appStateReducer 77
Adding Lists 78
Adding Tasks 79
Provide Dispatch Through The Context 81
Dispatching Actions 83
Moving Items 84
Define the `moveItem` helper function 84
Handling the `MOVE_LIST` action 86
Add Drag and Drop (Install React DnD) 87
Define The Type For Dragging 88
Store The Dragged Item In The State 89
Define The useItemDrag Hook 92
Drag Column 93
Hide The Dragged Item 96

- Styles For DragPreviewContainer . . . 96
- Implement The Custom Dragging Preview . . . 99
- Move The Dragged Item Preview . . . 104
- Hide The Default Drag Preview . . . 106
- Drag Cards . . . 108
- Update CustomDragLayer . . . 114
- Update The Reducer . . . 114
- Drag the Card To an Empty Column . . . 116
- Saving State On Backend. How To Make Network Requests . . . 118
- Loading The Data . . . 128

How to Test Your Applications: Testing a Digital Goods Store . . . 137

- Introduction . . . 137
- Get familiar with the application . . . 137
- Initial Setup . . . 142
- Writing Tests . . . 149
- Testing the App component . . . 150
- Mocking Components . . . 154
- Jest helper to test navigation . . . 156
- Global Helper With TypeScript . . . 157
- Testing navigation . . . 160
- Shared Components . . . 162
- Header . . . 162
- CartWidget . . . 164
- Loader Component . . . 168
- Home Page . . . 169
- ProductCard Component . . . 174
- Cart page . . . 178
- Cart component . . . 180
- Checkout Page . . . 188
- CheckoutList component . . . 188
- Testing The Form . . . 190
- Testing The FormField . . . 195
- Order summary page . . . 197
- Testing React Hooks . . . 201
- Testing useProducts . . . 202

Testing `useCart` . . . 206
Congratulations . . . 213

Patterns in React TypeScript Applications: Making Music with React . . . 214
Introduction . . . 214
What We’re Going to Build . . . 214
What We’re Going to Use . . . 216
First Steps and Basic Application Layout . . . 216
`Logo` component . . . 222
Combining Components . . . 223
A Bit of a Music Theory . . . 225
Third Party API and Browser API . . . 236
Patterns . . . 240
Adapter or Provider Pattern . . . 240
Creating a Keyboard . . . 243
Single Key on a Keyboard . . . 243
Styles for the `Key` . . . 244
Define the `Key` component . . . 248
Create the Keyboard component . . . 250
Update the `Main` component . . . 252
Adapter Hook . . . 253
Soundfont Adapter . . . 253
Connecting to a Keyboard . . . 259
Mapping Real Keys to Virtual . . . 265
Instruments List . . . 270
Instrument Selector . . . 272
Render Props . . . 278
What is a render prop . . . 278
Pros and Cons . . . 279
Creating Render Props With Functional Components . . . 280
Creating Render Props With Classes . . . 285
Higher-Order Components . . . 293
Higher-Order Functions . . . 294
Define a HOC . . . 296
When to Use . . . 297
Pros and Cons . . . 297

Caveats . . . 298
Instrument adapter as a Higher-Order Component . . . 299
Using HOC with Keyboard . . . 305
Passing Refs Through . . . 307
Static Composition . . . 313
Using Hooks with HOCs . . . 318
Conclusion . . . 320

Using Redux and TypeScript . . . 321
Introduction . . . 321
Preview The Final Result . . . 321
What is Redux? . . . 326
Why Can't We Use useReducer Instead of Redux? . . . 329
Initial Setup . . . 330
Redux Logger . . . 333
Prepare The Styles . . . 335
Update the App layout . . . 335
Working With Canvas . . . 337
Handling Canvas Events . . . 341
Define The Store Types . . . 341
Add Actions . . . 342
Add The Reducer Logic . . . 344
Dispatch Actions . . . 347
Draw The Current Stroke . . . 349
Define the `currentStrokeSelector` . . . 351
Update the `App` component . . . 352
Implement Selecting Colors . . . 354
Implement Undo and Redo . . . 359
Update the RootState type . . . 359
Create actions . . . 360
Update the reducer . . . 361
Create the EditPanel component . . . 363
Splitting Root Reducer And Using combineReducers . . . 367
Update the `App` component . . . 376
Join The Reducers Using combineReducers . . . 376
Exporting An Image . . . 377

Define the `getCanvasImage` 379
Create the `FilePanel` 380
Add the `FilePanel` to the `App` layout 382
Using Redux Toolkit 383
Configuring The Store 384
Fix Type Errors 385
Using createAction 386
Update the `App` component 391
Using createReducer 392
Using Slices 396
Remake The Imports 401
Add Modal Windows 403
Update the types 405
Add The Modal Manager Component 405
Define the modal windows 406
Define the `ModalLayer` component 407
Render the `ModalLayer` 408
Add `Save` and `Load` buttons 409
Prepare The Server 411
Save The Project Using Thunks 412
Define the API module 412
Handle saving the project 413
Define the `getBase64Thumbnail` function 415
Update the `ProjectSaveModal` 416
Load The Project 419
Update the types 419
Define the API module 420
Create a `projectsList` slice 420
Load the selected project 423
Show the list of projects 424
Update the `App` component 427

Static Site Generation and Server-Side Rendering Using Next.js 429
Introduction 429
What We’re Going to Build 429
Pre-Rendering 431

Next.js 433
Setting Up a Project 433
Creating A First Page 435
Basic Application Layout 436
Footer Component 440
Custom Document Component 441
Application Theme 446
Custom App Component 448
Front Page 450
Update the `Front` component 458
Page 404 458
Post Page Template 460
Backend API Server 461
Frontend API Client 465
Updating The Main Page 466
Pre-Render Post Page 471
Post API 471
Category Page 480
Category API 480
Adding Breadcrumbs 486
Comments and Server-Side Rendering 488
Components to render comments 491
API for Adding Comments 496
Adding comments to a page 498
Updating a statically generated page to use server-side rendering 500
Connecting Redux 501
Optimizing Images 512
Building Project 518
Deploying Project 519
Remaking API 520
Creating Client Requests 527
Updating Pages 529
Deployment with Serverless Functions 534
Summary 546

GraphQL, React, and TypeScript 547

Introduction 547
Is GraphQL better than REST? 550
What are we building? 552
Authenticate in GitHub and Preview The Final Result 556
Authenticating in GitHub 556
Previewing the final result 560
Setting up the project 561
Running TypeScript in the console 561
Add the .env file 563
Running the application 563
Get the auth code 565
Define the HTML page 565
Define the `getCode` 566
Auth Flow Link 569
Authentication context 572
GraphQL queries. Getting user data 574
Adding helper components 577
Define the `Button` component 578
Define the `List` component 579
Define the `Text` component 580
Define the `TextBox` component 581
Define the `Panel` component 582
Form helper components 584
Informationbal message components 587
Defining the WelcomeWindow layout 590
Getting GitHub GraphQL schema 592
Generating types 593
Adding routing 596
Define the resource screens 596
Define the routing scheme 597
Implement navigation 598
Define the `debounce` function 598
Define the `Header` 599
Render the `Header` 602
Repositories main component 603
Getting the list of repositories 605

GraphQL mutations. Creating repositories 611
Getting the repository ID 618
Working with GitHub issues 619
Getting the list of issues 622
Creating an issue 626
Generate the types for the mutation 627
Define the component 628
Define the layout 629
Create a new issue on form submit 630
Render the success and error results 632
Render the `NewIssue` component 633
Working with GitHub pull requests 634
Define the routing 636
Getting the list of pull requests 637
Update the root pull requests component 640
Creating a new pull request 642
Define the form 647
Add the navigation instructions 649
Use the component 649
Summary 650

Appendix **652**

Changelog **653**
Revision r12 (31-12-2021) 653
Revision r11 (26-03-2021) 653
Revision r10 (03-03-2021) 653
Revision r9 (26-02-2021) 653
Revision r8 (17-02-2021) 653
Revision r7 (01-12-2020) 654
Revision r6 (01-12-2020) 654
Revision r5 (10-11-2020) 654
Revision r4 (26-08-2020) 654
Revision 3p (07-30-2020) 654
Revision 2p (06-08-2020) 654
Revision 1p (05-20-2020) 655

Introduction

How To Get The Most Out Of This Book

Prerequisites

In this book we assume that you have at least the following skills:

- basic JavaScript knowledge (working with functions, objects, and arrays)
- basic React understanding (at least a general idea of component-based approach)
- some command line skills (you know how to run a command in the terminal)

We will mostly focus on the specifics of using TypeScript with React and some other popular technologies.

The instructions we give in this book are very detailed, so if you lack some of the listed skills, you can still follow along with the tutorials and be just fine.

Running Code Examples

Each section has an example app shipped with it. You can download code examples from the same place where you purchased this book.

If you have any trouble finding or downloading the code examples, email us at us@fullstack.io[1].

At the beginning of each section, you will find instructions on how to run the example app. In order to run the examples, you need a terminal app and NodeJS installed on your machine.

Make sure you have NodeJS installed. Run `node -v` to output your current NodeJS version:

[1] mailto:us@fullstack.io

```
$ node -v
v10.19.0
```

Here are the instructions for installing NodeJS on different systems:

Windows

To work with the examples in this book we recommend installing Cmder[2] as a terminal application.

We recommend installing node using nvm-windows[3]. Follow the installation instructions on the Github page.

Then run nvm to get the latest LTS version of NodeJS:

```
nvm install --lts
```

It will install the latest available LTS version.

Mac

Mac OS has a terminal app installed by default. To launch it toggle Spotlight, search for terminal and press `Enter`.

Run the following command to install nvm[4]:

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.33.11/inst\
all.sh | bash
```

Then run nvm to get the latest LTS version of NodeJS:

```
nvm install --lts
```

This command will also set the latest LTS version as default, so you should be all set.

If you face any issues follow the troubleshooting guide for Mac OS[5].

[2] https://cmder.net/
[3] https://github.com/coreybutler/nvm-windows
[4] https://github.com/nvm-sh/nvm
[5] https://github.com/nvm-sh/nvm#troubleshooting-on-macos

Linux

Most Linux distributions come with some terminal app provided by default. If you use Linux you probably know how to launch the terminal app.

Run the following command to install nvm[6]:

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.33.11/inst\
all.sh | bash
```

Then run nvm to get the latest LTS version of NodeJS:

```
nvm install --lts
```

In case of problems with installation follow the troubleshooting guide for Linux[7].

Code Blocks And Context

Code Block Numbering

In this book, we build example applications in steps. Sections have associated code examples:

```
01-first-app/
├── 01.02-how-to-bootstrap-react-typescript-app-automatically
├── 01.03-app-layout-react-typesctipt-basics
├── 01.05-prepare-styled-components
... // other steps
```

Their names match the names of the sections in the book.

If at some point in the chapter we achieve a state that we can run, you can run the version of the app from the particular step.

Some files in these folders can have numbered suffixes with `*.example`:

[6] https://github.com/nvm-sh/nvm
[7] https://github.com/nvm-sh/nvm#troubleshooting-on-linux

```
src/AddNewItem0.tsx.example
```

If you see this, it means that we are building up to something bigger. You can jump to the file with the same name but without a suffix to see a completed version of it.

Here the completed file would be `src/AddNewItem.tsx`.

Reporting Issues

We have done our best to make sure that our instructions are correct and code samples don't contain errors. There is still a chance that you will encounter problems.

If you find a place where a concept isn't clear or you find an inaccuracy in our explanations or a bug in our code, email us[8]! We want to make sure that our book is precise and clear.

Getting Help

If you have any problems working through the code examples in this book, email us[9].

To make it easier for us to help you, include the following information:

- What revision of the book are you referring to?
- What operating system are you on? (e.g. Mac OS X 10.13.2, Windows 95)
- Which chapter and which example project are you on?
- What were you trying to accomplish?
- What have you tried already?
- What output did you expect?
- What actually happened? (Including relevant log output.)

Ideally, please also provide a link to a git repository where we can reproduce the issue you are having.

[8] mailto:fullstack-react-typescript@newline.co
[9] mailto:fullstack-react-typescript@newline.co

What is TypeScript

> TypeScript is a typed superset of JavaScript that compiles to plain JavaScript - typescriptlang.org[10].

TypeScript allows you to specify types for values in your code, so you can develop applications with more confidence.

Using Types In Your Code

Consider this JavaScript example. Here we have a function that verifies that a password has at least eight characters:

```
function validatePasswordLength(password) {
  return password.length >= 8;
}
```

When you pass it a string that has at least eight characters it will return `true`.

```
validatePasswordLength("123456789") // Returns true
```

Someone might accidentally pass a numeric value to this function:

```
validatePasswordLength(123456789) // Returns false
```

In this case, the function will return `false`. Even though the function was designed to only work with strings you won't get an error saying that you misused the function.

It can cause nasty run-time bugs that might be hard to catch.

With Typescript we can restrict the values that we pass to our function to only be strings:

[10] https://typescriptlang.org

```
function validatePasswordLength(password: string) {
  return password.length >= 8;
}

validatePasswordLength(123456789) // Argument of type '123456789' is no\
t assignable to parameter of type 'string'.
```

If we call our function with the wrong type, TypeScript will give us an error.

TypeScript can tell if we have an error in our code just by analyzing the syntax. That means that you don't have to run your program. Most code editors support TypeScript so the error will be immediately highlighted.

Strings and numbers are examples of built-in types in TypeScript. TypeScript supports all the types available in JavaScript and adds some more. We will get familiar with a lot of them during the next chapters. But the coolest thing is that you can define your own types.

Defining Custom Types

Let's say we have a `greet` function that works with `user` objects. It generates a greeting message using provided first and last names.

```
function greet(user){
  return `Hello ${user.firstName} ${user.lastName}`;
}
```

How can we make sure that this function receives an input of the correct type?

We can define our own `User` type and specify it as a type of our function's `user` argument:

```
type User = {
  firstName: string;
  lastName: string;
}

function greet(user: User){
  return `Hello ${user.firstName} ${user.lastName}`;
}
```

Our function will only accept objects that match the defined `User` type.

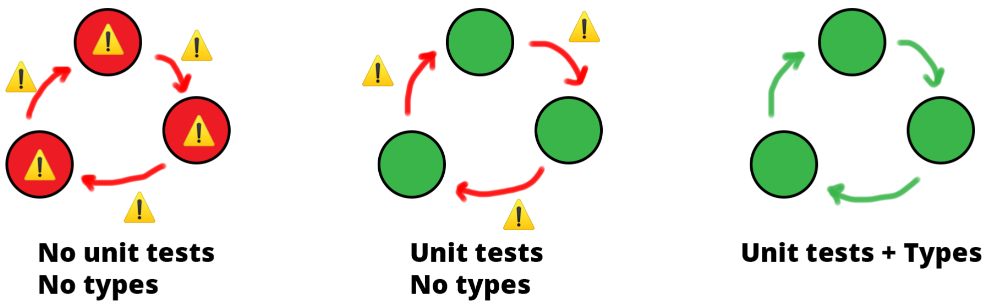
```
greet({firstName: "Maksim", lastName: "Ivanov"}) // Returns "Hello Maks\
im Ivanov!"
```

If we try to pass something else, we'll get an error.

```
greet({}) // Argument of type '{}' is not assignable to parameter of ty\
pe 'User'.
          // Type '{}' is missing the following properties from type 'U\
ser': firstName, lastName
```

Benefits Of Using TypeScript

Preventing errors. As you can see with TypeScript we can define the interfaces for parts of our program, so we can be sure that they interact correctly. It means they will have clear contracts of communication with each other which will significantly reduce the amount of bugs.



TypeScript contracts by which parts of your program communicate.

If on top of that we cover our code with unit tests - BOOM, our application becomes rock-solid. Now we can add new features with confidence, without fear of breaking it.

> There is a research paper[11] showing that just by using typed language you will get 15% fewer bugs in your code. There is also an interesting paper about unit tests[12] stating that products, where test-driven development was applied had between 40% and 90% reductions in pre-release bug density.

Better Developer Experience. When you use TypeScript you also get better code suggestions in your editor, which makes it easier to work with large and unfamiliar codebases.

Why Use TypeScript With React

The revolutionary thing about React is that it allows you to describe your application as a tree of components.

A component can represent an element, like a button or an input. It can be a group of elements representing a login form. Or it can be a complete page that consists of multiple simple components.

[11] http://ttendency.cs.ucl.ac.uk/projects/type_study/documents/type_study.pdf
[12] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.210.4502&rep=rep1&type=pdf

Components can pass the information down the tree, from parent to child. You can also pass down functions as callbacks, so if something happens in the child component it can notify its parent by calling the passed callback function.

This is where TypeScript becomes very handy. You can use it to define the interfaces of your components, so that you can be sure that your component only gets props with the correct types.

If you have worked with React before you probably know that you can specify a component's interface using `prop-types`.

```
import PropTypes from 'prop-types';

const Greeting = ({name}) => {
  return (
    <h1>Hello, {name}</h1>
  );
}

Greeting.propTypes = {
  name: PropTypes.string
};
```

If you can do this with `prop-types`, why would you need TypeScript?

There are several reasons:

- You don't need to run your application to know if you have type errors. TypeScript can be run by your code editor so you can see the errors as soon as you make them.
- You can only use `prop-types` with components. In your application you will probably have functions and classes that are not using React. It is important to be able to provide types for them as well.
- TypeScript is just more powerful. It gives you more options to define the types and then it allows you to use this type information in many different ways. We will demonstrate examples of this in the next chapters.

A Necessary Word Of Caution

TypeScript does not catch run-time type errors. It means that you can write code that will pass the type check, but you will get an error upon execution.

```
function messUpTheArray(arr: Array<string | number>): void {
    arr.push(3);
}

const strings: Array<string> = ['foo', 'bar'];
messUpTheArray(strings);

const s: string = strings[2];
console.log(s.toLowerCase()) // Uncaught TypeError: s.toLowerCase is no\
t a function
```

Try to launch this code example in TypeScript sandbox[13]. You will get an `Uncaught TypeError: s.toLowerCase is not a function` error.

Here we said that our `messUpTheArray` accepts an array containing elements of type `string` or `number`. Then we passed it our `strings` array that is defined as an array of `string` elements. TypeScript allows this because it thinks that types `Array<string | number>` and `Array<string>` match.

Usually, this is convenient because an array that is defined as having `number` or `string` elements can have only string items.

```
const stringsAndNumbers: Array<string | number> = ['foo', 'bar'];
```

In our case it allowed a bug to slip through the type checking.

It also means that you have to be extra careful with data obtained through network requests or loaded from the file system.

In this book, we will demonstrate the techniques that allow us to minimize the risk of such issues.

[13]https://www.typescriptlang.org/play/index.html?ssl=9&ssc=29&pln=1&pc=1#code/GYVwdgxgLglg9mABAWwKYGd0FUAOAVAC1QEEAnUgQwE8AKC8gLkTMqoB50pSYwBzRAD6IwIZACNUpAHwBKJgDc4MACaIA
fCE1HrzoYQBM6U4ucAA2qIZdcLyFhlBwADJwAO6SAMIU6Kg0MjLaQA

Your First React and TypeScript Application: Building Trello with Drag and Drop

Introduction

In this part of the book, we will create our first React + TypeScript application.

We will bootstrap the file structure using the `create-react-app` CLI. If you've worked with React before, you might be familiar with it. If you haven't heard about it yet - no worries, I will talk about it in more detail further in this chapter.

I will show you the file structure it generates and then I'll explain the purpose of each file there.

Then we'll create our components. You'll see how to use TypeScript to specify the props. We'll briefly discuss the difference between types and interfaces.

We will mostly work with functional components, because this is the most popular approach right now.

We'll talk about using JavaScript libraries in your TypeScript project. Some of them are compatible by default, and some require you to install special `@types` packages.

Our application will also store the state on the backend. So we will discuss how to use `fetch` with TypeScript.

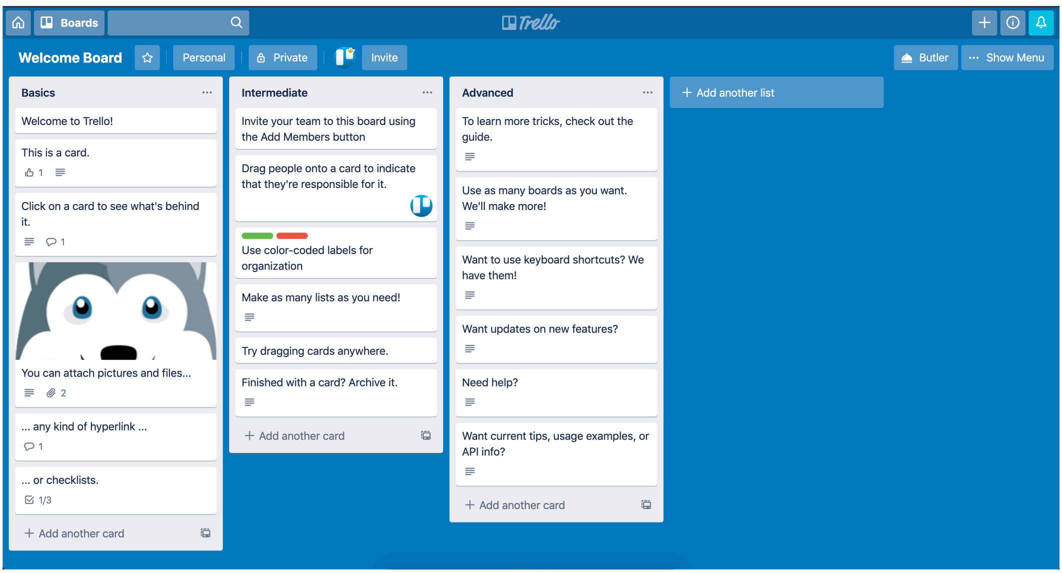
So in this chapter we'll cover:

- creating components
- defining props
- using state
- handling events

- working with refs
- styling components
- using external libraries
- making network requests

What Are We Building?

We will create a simplified version of a kanban board. A popular example of such an application is *Trello*.



Trello board

In Trello, you can create tasks and organize them into lists. You can drag both cards and lists to reorder them. You can also add comments and attach files to your tasks.

In our application we will recreate only the core functionality: creating tasks, making lists and dragging them around.

Prerequisites

There are a bunch of requirements before you start working with this chapter.

First of all, you need to know how to use the command line. On Mac, you can use `Terminal.app`, available by default. All Linux distributions also have some preinstalled terminal applications. On Windows I recommend using Cygwin[14] or Cmder[15]. If you are more experienced you can use Windows Subsystem for Linux[16].

You will need a code editor with TypeScript support. I recommend using VSCode, which supports TypeScript out of the box.

Make sure you have Node 10.16.0 or later. You can use nvm[17] on Mac or Linux to switch Node versions. For Windows there is nvm-windows[18].

You also need to know how to use node package managers. In this chapter's examples, I will use Yarn[19]. You can use npm[20] if you want.

> All the examples for this chapter contain `yarn.lock` files. Remove them if you want to use npm to install dependencies.

You need to have some React understanding. Specifically, you have to know how to use functional components and React hooks. In this example, we won't use class-based components. If you don't feel confident it might be worth visiting the React Documentation[21] to refresh your knowledge.

Preview The Final Result

We will build our app together from scratch, and I will explain every step as we go, but to get a sense of where we're going, it's helpful if you check out the result first.

This book has an attached `zip` archive with examples for each step. You can find the completed example in `code/01-first-app/completed`.

Unzip the archive and `cd` to the app folder.

---

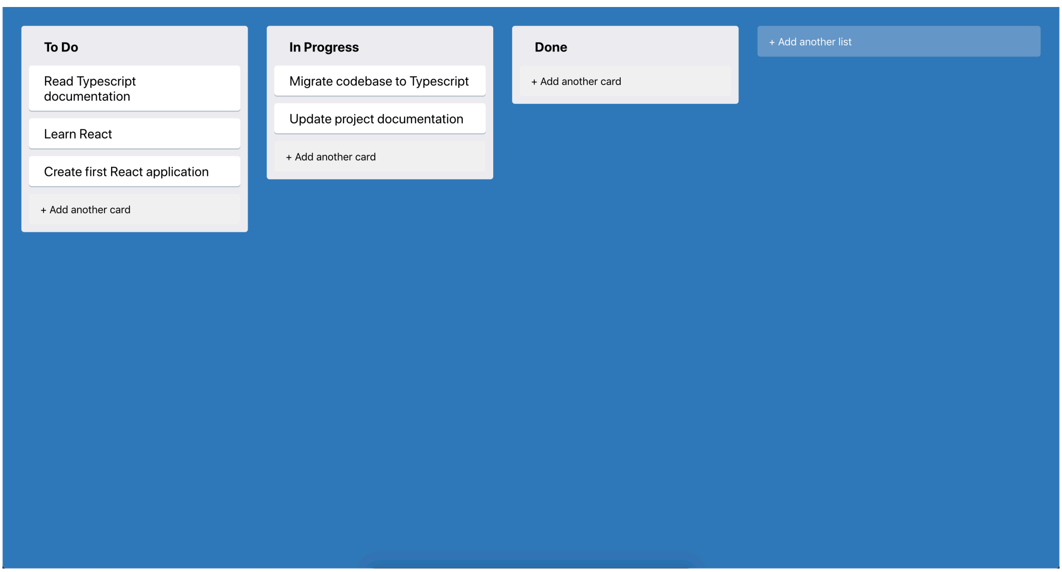
[14] https://www.cygwin.com/
[15] https://cmder.net/
[16] https://docs.microsoft.com/en-us/windows/wsl/install-win10
[17] https://github.com/nvm-sh/nvm#installing-and-updating
[18] https://github.com/coreybutler/nvm-windows#node-version-manager-nvm-for-windows
[19] https://yarnpkg.com/
[20] https://www.npmjs.com/
[21] https://reactjs.org/docs/getting-started.html

```
cd code/01-first-app/completed
```

When you are there, install the dependencies and launch the app:

```
yarn && yarn dev
```

This should open the app in the browser. If this doesn't happen, navigate to `http://localhost:3000` and open it manually.



Final result

Our app will have a bunch of columns that you can drag around. Each column represents a list of tasks.

Each task is rendered as a draggable card. You can drag each card inside a column and between columns.

You can create new columns by clicking the button that says "+ Add another list". Each column also has a button at the bottom that allows the creation of new cards.

Create a few more cards and columns and drag them around.

The state of the application is preserved on the backend. You can reload the page and all the lists and tasks will stay where you left them.

How to Bootstrap React + TypeScript App Automatically

In this chapter, we will use an automatic CLI tool to bootstrap our project's initial structure.

Why Use Automatic App Generators?

Usually, when you create a React application, you need to create a bunch of boilerplate files.

First, you will need to set up a transpiler. React uses `jsx` syntax to describe the layout, and also you'll probably want to use the modern JavaScript features. To do this we'll have to install and set up Babel[22]. It will transform our code to normal JavaScript that current and older browsers can support.

You will need a bundler. You will have plenty of different files: your components code, styles, maybe images and fonts. To bundle them together into small packages you'll have to set up Webpack[23] or Parcel[24].

Then there are a lot of smaller things. Setting up a test runner, adding vendor prefixes to your CSS rules, setting up linter and enabling hot-reload, so you don't have to refresh the page manually every time you change the code. It can be a lot of work.

To simplify the process we will use `create-react-app`. It is a tool that will generate the file structure and automatically create all the settings files for our project. This way we will be able to focus on using React tools in the TypeScript environment.

How to Use create-react-app With TypeScript

Navigate to the folder where you keep your programming projects and run `create-react-app`.

[22] https://babeljs.io/
[23] https://webpack.js.org/
[24] https://parceljs.org/

```
npx create-react-app --template typescript trello-clone
```

Here we've used `npx` to run `create-react-app` without installing it. This is the recommended way to use `create-react-app`. Read more in their getting started guide[25].

We specified an option `--template typescript`, so our app will have all the settings needed to work with TypeScript. The last argument is the name of our app. `create-react-app` will automatically generate the `trello-clone` folder with all the necessary files.

`cd` to `trello-clone` folder and open it with your favorite code editor.

Project Structure Generated By create-react-app

Let's look at the application structure.

If you've used `create-react-app` before, it will look familiar.

```
├── public
│   ├── favicon.ico
│   ├── index.html
│   ├── logo192.png
│   ├── logo512.png
│   ├── manifest.json
│   └── robots.txt
├── src
│   ├── App.css
│   ├── App.test.tsx
│   ├── App.tsx
│   ├── index.css
│   ├── index.tsx
│   ├── logo.svg
│   ├── react-app-env.d.ts
│   ├── reportWebVitals.ts
│   └── setupTests.ts
```

[25]https://create-react-app.dev/docs/getting-started/

```
├── node_modules
|   └── ...
├── .gitignore
├── README.md
├── package.json
├── tsconfig.json
└── yarn.lock
```

Let's go through the files and see why we need them. We'll do a short overview, and then go back to some of the files and talk about them a bit more.

Files In The Root

First, let's look at the root of our project.

README.md. This is a `markdown` file that contains a description of your application. For example, Github will use this file to generate an `html` summary that you can see at the bottom of projects.

package.json. This file contains metadata relevant to the project. For example, it contains the `name`, `version` and `description` of our app. It also contains the `dependencies` list with external libraries that our app depends on.

> You can find the full list of possible `package.json` fields and their descriptions on the npm website.[26]

Open the `package.json` file and check what packages are installed with `create-react-app`:

[26] https://docs.npmjs.com/files/package.json

```
  "dependencies": {
    "@testing-library/jest-dom": "^5.11.4",
    "@testing-library/react": "^11.1.0",
    "@testing-library/user-event": "^12.1.10",
    "@types/jest": "^26.0.15",
    "@types/node": "^12.0.0",
    "@types/react": "^17.0.0",
    "@types/react-dom": "^17.0.0",
    "react": "^17.0.2",
    "react-dom": "^17.0.2",
    "react-scripts": "4.0.3",
    "typescript": "^4.1.2",
    "web-vitals": "^1.0.1"
  },
```

Some packages that we use have a corresponding `@types/*` package.

> I'm showing only the `dependencies` block because this is where type definitions are installed when using `create-react-app`. Some people prefer to put types-packages in `devDependencies`.

Those `@types/*` packages contain type definitions for libraries originally written in JavaScript. Why do we need them if TypeScript can parse the JavaScript code as well?

The problem with JavaScript is that often it's impossible to tell what types the code will work with. Let's say we have a JavaScript code with a function that accepts the `data` argument:

```
export function saveData(data) {
  // data saving logic
}
```

TypeScript can parse this code, but it has no way of knowing what type the `data` attribute is restricted to. So for TypeScript, the `data` attribute will implicitly have type `any`. This type matches with absolutely anything, which defeats the purpose of type-checking.

If we know that the function is meant to be more specific, for instance, it only accepts the values of type `string`, we can create a `*.d.ts` file and describe it there manually.

This `*.d.ts` file name should match the module name we provide types for. For example, if this `saveData` function comes from the `save-data` module - we will create a `save-data.d.ts` file. We'll need to put this file where the TypeScript compiler will see it, usually in its `src` folder.

This file will then contain the declaration for our `saveData` function.

```
declare function saveData(data: string): void
```

Here we specified that `data` must have type `string`. We've also specified return type `void` for our function because it should not return any value.

We could create a package with this file and publish it to NPM. This is what all those `@types/*` packages are: they contain `*.d.ts` files with type definitions for libraries.

It is a convention that all the types-packages are published under the `@types` namespace. Those packages are provided by the DefinitelyTyped[27] repository.

When you install javascript dependencies that don't contain type definitions, you can usually install them separately by installing a package with the same name and `@types` prefix.

Versions for `@types/*` and their corresponding packages don't have to match exactly. Here you can see that `react-dom` has version `^17.0.1` and `@types/react-dom` is `^17.0.2`.

yarn.lock. This file is generated when you install the dependencies by running `yarn` in your project root. The file contains resolved dependencies versions along with their sub-dependencies. It is needed for consistent installations on different machines. If you use `npm` to manage dependencies, you will have a `package-lock.json` instead.

tsconfig.json. This contains the TypeScript configuration. We don't need to edit this file because the default settings work fine for us.

.gitignore. This file contains the list of files and folders that shouldn't end up in your `git` repository.

[27]http://definitelytyped.org/

These are all the files that we find in the root of our project. Now let's take a look at the folders.

public Folder

The `public` folder contains the static files for our app. They are not included in the compilation process and remain untouched during the build.

> Read more about the `public` folder in the Create React App documentation[28].

index.html. This file contains a special `<div id="root">` that will be a mounting point for our React application.

manifest.json. This provides application metadata for Progressive Web Apps[29]. For example, the file allows installation of your application on a mobile phone's home screen, similar to native apps. It contains the app name, icons, theme colors, and other data needed to make your app installable.

> You can read more about `manifest.json` on MDN.[30]

favicon.ico, logo192.png, logo512.png. These are icons for your application. There is `favicon.ico`, a small icon that is shown on browser tabs. Also, there are two bigger icons: `logo192.png` and `logo512.png`. They are referenced in `manifest.json` and will be used on mobile devices if your app will be added to the home screen.

robots.txt. This tells crawlers what resources they shouldn't access. By default it allows everything.

> Read more about `robots.txt` on the robotstxt website.[31]

[28] https://create-react-app.dev/docs/using-the-public-folder/
[29] https://web.dev/progressive-web-apps/
[30] https://developer.mozilla.org/en-US/docs/Web/Manifest
[31] https://www.robotstxt.org/robotstxt.html

src Folder

Take a look at the `src` folder. Files in this folder are processed by `webpack` and will be added to your app's bundle.

This folder contains a bunch of files with `.tsx` extension: `index.tsx`, `App.tsx`, `App.test.tsx`. It means that those files contain *JSX* code.

> *JSX* is an html-like syntax used in React applications to describe the layout. Read more about it in the React Docs.[32]

In a JavaScript React application, we could use either `.jsx` or `.js` extensions for such files. It would make no difference.

With TypeScript, you should use `.tsx` extensions on files that have JSX code, and `.ts` on files that don't.

This is important because otherwise there can be a syntactic clash. Both TypeScript and JSX use angle brackets, but for different purposes.

TypeScript has a *type assertion operator* that uses angle brackets:

```
const text = <string>"Hello TypeScript"
// text: string
```

You can use this operator to manually provide a type for your target variable. In this case, we specify that `text` should have type `string`.

Otherwise, it would have type `Hello TypeScript`. When you assign a `const` a `string` value, TypeScript will use this value as a type:

```
const text = "Hello TypeScript"
// text: "Hello TypeScript"
```

This operator can create ambiguity with *JSX* elements that also use angle brackets:

[32]https://reactjs.org/docs/introducing-jsx.html

```
<div></div>
```

You can read about it in the TypeScript Documentation[33].

index.tsx

The most important file in the `/src` folder is `index.tsx`. It is an entry point for our application. It means that `webpack` will start to build our application from this file, and then will recursively include other files referenced by `import` statements.

Let's look at this file's contents:

```
import React from "react"
import ReactDOM from "react-dom"
import "./index.css"
import App from "./App"
import reportWebVitals from "./reportWebVitals"

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById("root")
)

// If you want to start measuring performance in your app, pass a funct\
ion
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vit\
als
reportWebVitals()
```

First, we import React, because we have a `JSX` statement here.

[33] https://www.typescriptlang.org/docs/handbook/jsx.html#the-as-operator

```
ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById("root")
)
```

Babel transpiles `<App />` to `React.createElement(App, null)`. It means that implicitly we reference React in this file, this is why we import it.

Then we import `ReactDOM`. We use it to render our application to the `index.html` page. We find an element with an id `root` and render our `App` component to it.

We have the `index.css` import. This file contains styles relevant to the whole application, so we import it here.

We import the `App` component because we want to render it into the HTML.

After that we import `reportWebVitals`. This module can be useful if you want to measure your app performance. It is explained in more detail here[34].

As it is not specific to TypeScript, we are not going to focus on it.

Then we render the `App` using the `ReactDOM.render` method. Note that by default the `App` component is wrapped into the `React.StrictMode` component. This component mostly checks that no deprecated methods are being used. All those checks are performed only in development mode, and it is good practice to wrap your app into `React.StrictMode`.

> Check the documentation[35] for the updated list of the `StrictMode` functionality.

App.tsx

Let's open `src/App.tsx`. If you use modern `create-react-app`, this file won't be very different to the regular JavaScript version.

[34] https://create-react-app.dev/docs/measuring-performance/
[35] https://reactjs.org/docs/strict-mode.html

> Currently, in JavaScript apps generated with `create-react-app`, you don't need to import React at all. Read more here[36].

In older versions, React was imported differently.

Instead of:

```
import React from "react"
```

You would see:

```
import * as React from "react"
```

To explain this I will have to tell you a bit more about the default imports.

When you write `import name from 'module'` it is the same as writing `import {default as name} from 'module'`. To be able to do this the module should have the default export, which would look like this: `export default 'something'`.

React doesn't have the default export. Instead, it just exports all its functions in one object.

You can see it in React source code[37]. React exports an object full of different classes and functions:

```
export {
  Children,
  createRef // ... other exports
} from "./src/React"
```

So, strictly speaking `import * as React from 'react'` is the correct way of importing React.

But if you've used React with JavaScript before, you'll have noticed that React is always imported there as if it has the default export.

[36] https://reactjs.org/blog/2020/09/22/introducing-the-new-jsx-transform.html
[37] https://github.com/facebook/react/blob/master/packages/react/index.js

```
import React from "react"
```

This is possible for two reasons. First - JavaScript doesn't type check the imports. It will allow you to import whatever, and then if something goes wrong, it will only throw an error during *runtime*. Second - you most likely use React with some bundler like Webpack, and it's smart enough to check if no default property is set in the export, and where this is the case to just use the entire export as the default value.

When you use TypeScript, it's a different story. TypeScript checks that what you are trying to import has the matching export. If the default export doesn't exist, the default behavior of TypeScript will be to throw an error, something like this:

> TypeScript error in trello-clone/src/App.tsx(1,8): Module '"trello-clone/node_modules/@types/react/index"' can only be default-imported using the 'allowSyntheticDefaultImports' flag TS1259

Thankfully, since version *2.7*, TypeScript has the `allowSyntheticDefaultImports` option. When this option is enabled TypeScript will *pretend* that the imported module has the default export. So we'll be able to import React normally.

Modern versions of `create-react-app` enable this option by default. Read more about it in the TypeScript 2.7 release notes[38].

react-app-env.d.ts

Another file with an interesting extension is `react-app-env.d.ts`. Let's take a look.

Files with `*.d.ts` extensions contain TypeScript types definitions. Usually, these are needed for libraries that were originally written in JavaScript.

This file contains the following code:

```
/// <reference types="react-scripts" />
```

Here we have a special `reference` tag that includes types from the `react-scripts` package.

[38]https://www.typescriptlang.org/docs/handbook/release-notes/typescript-2-7.html#support-for-import-d-from-cjs-from-commonjs-modules-with---esmoduleinterop

> Read more about "triple slash directives" in the TypeScript documentation[39].

By default, this would reference the file `./node_modules/react-scripts/index.d.ts`, but `react-scripts` package contains a field `"types": "./lib/react-app.d.ts"` in its `package.json`. So we end up referencing types from:

```
./node_modules/react-scripts/lib/react-app.d.ts
```

> Instead of looking up the file in the `node_modules` folder you can check the react-scripts GitHub repo[40].

This file contains types for the Node environment and also types for static resources: images and stylesheets.

Why do we need type declarations for stylesheets and images?

TypeScript doesn't even see the static resources files. It is only interested in files with `.tsx`, `.ts`, and `d.ts` extensions. With some tweaking, it will also see `.js` and `.jsx` files.

Let's say you are trying to import an image:

```
import logo from "./logo.svg"
```

TypeScript has no idea about files with `.svg` extension so it will throw something like this: `Cannot find module './logo.svg'. TS2307`.

To fix it we can create a special module type. Or in our case it is already created.

One of the declarations in `react-app.d.ts` allows import of `*.svg` files:

[39] https://www.typescriptlang.org/docs/handbook/triple-slash-directives.html#-reference-types-
[40] https://github.com/facebook/create-react-app/blob/master/packages/react-scripts/package.json#L29

```
declare module '*.svg' {
  import * as React from 'react';

  export const ReactComponent: React.FunctionComponent<React.SVGProps<
    SVGSVGElement
  > & { title?: string }>;

  const src: string;
  export default src;
}
```

This declaration is a bit complex but bear with me.

First thing that happens here is the module declaration. We declare a wildcard module so that any import that would end with `svg` would use our type declaration.

Then inside this module we import `React` namespace because we'll need types from it.

Then we define a named export for `ReactComponent`. This is a "React component" representation of the SVG image that will be imported.

This code might be hard to understand before we discuss TypeScript generics and intersection types.

```
React.FunctionComponent<React.SVGProps<
  SVGSVGElement
> & { title?: string }>;
```

I suggest you go back here and check if you can understand this code after we discuss those topics.

For now I'll say that here we define `ReactComponent` as a functional component that receives the props of the SVG element, plus an optional `title` prop of type `string`.

It is done so that TypeScript knows that SVG images can be imported as React components. Read more about it in Create React App documentation[41].

Here I'll show you how it would look in your application:

[41] https://create-react-app.dev/docs/adding-images-fonts-and-files/#adding-svgs

```
import { ReactComponent } from './logo.svg';

function App() {
  return (
    <div>
      <ReactComponent />
    </div>
  );
}
```

In this case if you open the browser you'll see that the logo is rendered as inline SVG.

Check it yourself - open `src/App.tsx` and change the default import to named one:

```
import { ReactComponent as Logo } from './logo.svg';
```

For example like this. And then use it in the application layout instead of the `img` tag.

Back to our module declaration. There is another export after `ReactComponent`. This time it is default export of the `src` constant of type `string`.

In your app you would import it like this:

```
import image from "./foo.svg"
// image has type `string` here
```

In this case it would be treated as a path to some static file, that would look somewhat like this: `/static/media/foo.6ce24c58.svg`.

And Webpack dev server that Create React App is using is already set up to resolve static files to their paths in the `/static` folder.

App Layout. React + TypeScript Basics

Remove The Clutter

Before we start writing the new code, let's remove the files we aren't going to use.

Go to `src` folder and remove the following files:

- `logo.svg`
- `App.css`
- `App.test.tsx`

You should end up with the following files in your `src` folder:

```
src
├── App.tsx
├── index.css
├── index.tsx
├── react-app-env.d.ts
├── reportWebVitals.ts
└── setupTests.ts
```

Also open the `src/App.tsx`, remove the imports of the files that no longer exist and remove the layout:

```
export const App = () => {
  return null
}
```

For now the `App` component will just return `null`.

> If you use VSCode - configure it to use the Workspace TypeScript version. Otherwise if your global TypeScript version is older than 4.1 you will get an error: `'React' refers to a UMD global, but the current file is a module. Consider adding an import instead.ts(2686)`. Here is a relevant StackOverflow answer[42]

Then open the `src/index.tsx` and remove the `reportWebVitals`, we aren't going to use them anyway:

[42]https://stackoverflow.com/questions/50432556/cannot-use-jsx-unless-the-jsx-flag-is-provided

```
import React from "react"
import ReactDOM from "react-dom"
import "./index.css"
import { App } from "./App"

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById("root")
)
```

We also changed the default `App` export to named, so update the import in the `index.tsx` file to use the curly brackets.

> I prefer named exports over default exports mainly because they work better with refactoring tools in VSCode. if you default export a component and then rename that component, it will only rename the component in that file and not any of the other references in other files. With named exports it will rename the component and all the references to that component in all the other files.

Add Global Styles

Let's define the styles to apply to the whole application. Edit `src/index.css` and add some global CSS rules:

```css
html {
  box-sizing: border-box;
}

*,
*:before,
*:after {
  box-sizing: inherit;
}

html,
body,
#root {
  height: 100%;
}
```

Here we add `box-sizing: border-box` to all elements. This directive tells the browser to include `padding` and `border` elements in its `width` and `height` calculations.

We also make the `html` and `body` elements take up the whole screen vertically.

How To Style React Elements

There are several ways to style React elements:

- Regular CSS files, including CSS-modules.
- Manually specifying an element's `style` property.
- Using external styling libraries.

Let's briefly talk about each of the options.

Using Separate CSS Files

You can have styles defined in CSS files. To use them you'll need a properly configured bundler, like Webpack. Create React App includes a pre-configured Webpack that supports loading CSS files.

In our project, we have the `index.css` file. It contains styles that will be applied globally. We import this `index.css` file in the `index.tsx`.

React elements accept the `className` prop that sets the class attribute of the rendered DOM node.

```
<div className="styled">React element</div>
```

Passing CSS Rules Through Style Property

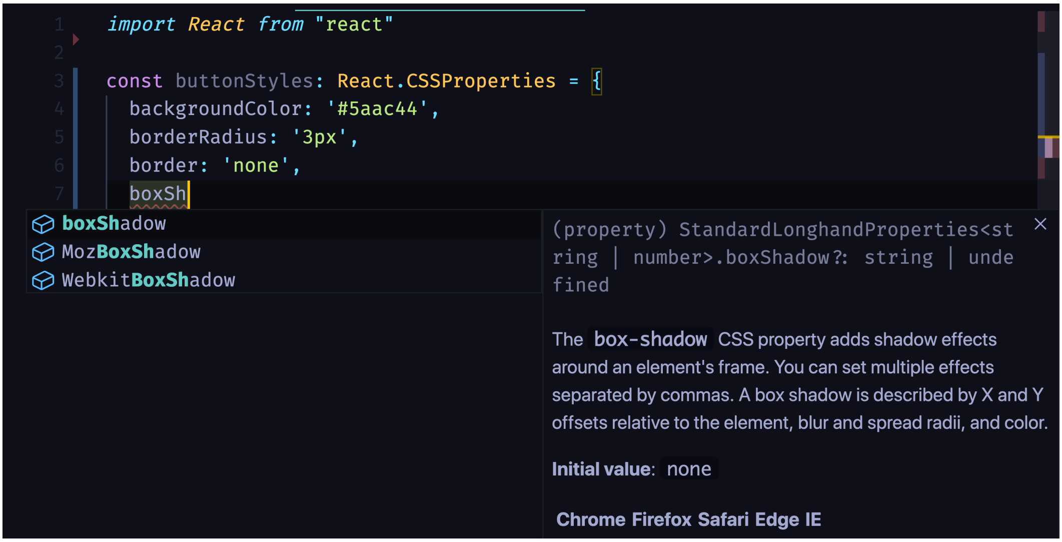
Another option is to pass an object with styling rules through the `style` property. You can declare the object inline, then you won't need to specify a type for it:

```
<div style={{ backgroundColor: "red" }}>Styled element</div>
```

A better practice is to define styles in a separate constant:

```
import React from "react"

const buttonStyles: React.CSSProperties = {
  backgroundColor: "#5aac44",
  borderRadius: "3px",
  border: "none",
  boxShadow: "none"
}
```

Here we set `buttonStyles` type to `React.CSSProperties`. As a bonus, we get autocompletion hints for CSS property names.



TypeScript provides nice CSS autocompletion

> We aren't using real CSS attribute names. In React the style properties are in camel case form. For example `background-color` is `backgroundColor` and so on.

Using External Styling Libraries

There are a lot of libraries that simplify working with CSS in React. I like to use Styled Components[43].

Styled Components allows you to define reusable components with attached styles like this:

[43] https://github.com/styled-components/styled-components

```
import styled from "styled-components"

const Button = styled.button`
  background-color: #5aac44;
  border-radius: 3px;
  border: none;
  box-shadow: none;
`
```

Then you can use them as regular React components:

```
<Button>Click me</Button>
```

At the time of writing, Styled Components has **28.4k** stars on Github. It also has TypeScript support.

Prepare Styled Components

Install `styled-components`. Working with `@types` packages

We'll begin by creating a bunch of styled components so that our application looks good from step one.

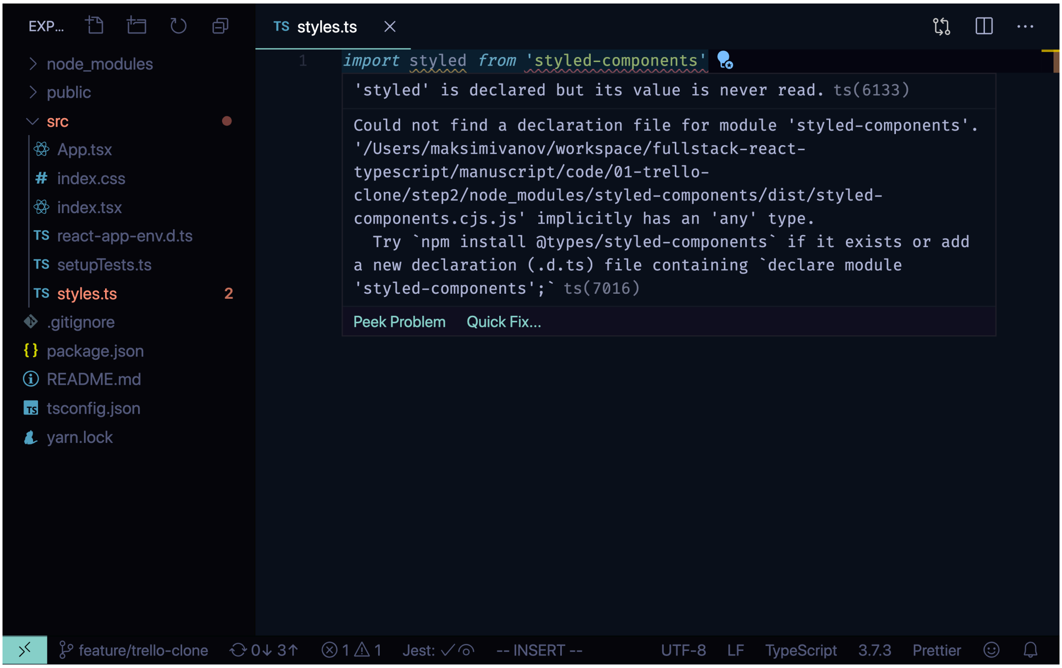
Install the `styled-components` library:

```
yarn add styled-components@^5.2.1
```

After it is installed we can define our first styled component. Create the `src/styles.ts` file and import `styled` from `styled-components`:

```
import styled from "styled-components"
```

You'll get a TypeScript error.



Missing @types for styled-components

TypeScript errors can be quite wordy, but usually, the most valuable information is located closer to the end of the message.

Here TypeScript tells us that we are missing type declarations for `styled-components` package. It also suggests that we install missing types from `@types/styled-components`.

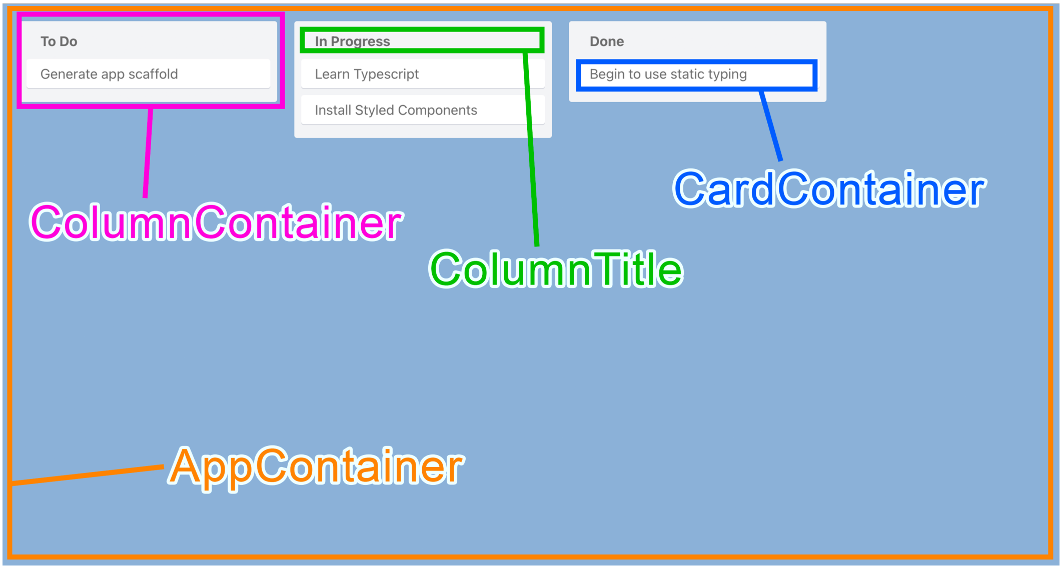
Install the missing types:

```
yarn add @types/styled-components@^5.1.9
```

Now we are ready to define our styled components.

Break the UI into components

Let's look at the app to decide what styled components will we define:



Application Components

- `AppContainer` - it will help us to arrange the columns horizontally. It is going to wrap the whole application.
- `ColumnContainer` - it is a visual representation of a column. It will have grey background and rounded corners.
- `ColumnTitle` - it will make the column title bold and add paddings to it.
- `CardContainer` - it will visually represent the card.

Styles For AppContainer

We want our app layout to contain a list of columns arranged horizontally. We will use flexbox to achieve this.

Create an `AppContainer` component in `styles.ts` and export it.

```
export const AppContainer = styled.div`
  align-items: flex-start;
  background-color: #3179ba;
  display: flex;
  flex-direction: row;
  height: 100%;
  padding: 20px;
  width: 100%;
`
```

Style component functions accept strings with *CSS* rules. When we use template strings, we can omit the brackets and just append the string to the function name.

Here we specify `display: flex` to make it use the flexbox layout. We set `flex-direction` property to `row`, to arrange our items horizontally. And we add a `20px` padding inside it.

Styles For Columns

Let's make our `Column` component look good. Create a `ColumnContainer` component in `src/styles.ts`.

```
export const ColumnContainer = styled.div`
  background-color: #ebecf0;
  width: 300px;
  min-height: 40px;
  margin-right: 20px;
  border-radius: 3px;
  padding: 8px 8px;
  flex-grow: 0;
`
```

Here we specify a grey background, margins, and paddings, and also specify `flex-grow: 0` so the component doesn't try to take up all the horizontal space.

Still in `src/styles.ts`, create styles for `ColumnTitle`:

```
export const ColumnTitle = styled.div`
  padding: 6px 16px 12px;
  font-weight: bold;
`
```

We'll use it to wrap our column's title.

Styles For Cards

We'll need styles for the `Card` component. Open `src/styles.ts` and create a new styled component called `CardContainer`. Don't forget to export it.

```
export const CardContainer = styled.div`
  background-color: #fff;
  cursor: pointer;
  margin-bottom: 0.5rem;
  padding: 0.5rem 1rem;
  max-width: 300px;
  border-radius: 3px;
  box-shadow: #091e4240 0px 1px 0px 0px;
`
```

Here we want to let the user know that cards are interactive so we specify `cursor: pointer`. We also want our cards to look nice so we add a `box-shadow`.

Render Everything Together

Go back to `src/App.tsx` and render the styled components:

```
import {
  AppContainer,
  ColumnContainer,
  ColumnTitle,
  CardContainer
} from "./styles"

export const App = () => {
  return (
    <AppContainer>
      <ColumnContainer>
        <ColumnTitle>Todo:</ColumnTitle>
        <CardContainer>FirstItem</CardContainer>
        <CardContainer>SecondItem</CardContainer>
        <CardContainer>ThirdItem</CardContainer>
      </ColumnContainer>
    </AppContainer>
  )
}
```

Create Column Components

In this section, I won't explain how React components work. If you need to pick this knowledge up, refer to the React documentation[44]. Make sure you know what props and state are, and how lifecycle events work.

We'll start with the `Column` component. Create a new file `src/Column.tsx`:

[44] https://reactjs.org/docs/components-and-props.html

```
import { ColumnContainer, ColumnTitle, CardContainer } from "./styles"

type ColumnProps = {
  text: string
}

export const Column = ({ text }: ColumnProps) => {
  return (
    <ColumnContainer>
      <ColumnTitle>{text}</ColumnTitle>
      <CardContainer>Generate app scaffold</CardContainer>
      <CardContainer>Learn TypeScript</CardContainer>
      <CardContainer>Begin to use static typing</CardContainer>
    </ColumnContainer>
  )
}
```

This component will receive the `text` prop and render it as a column title.

Update the `src/App.tsx` to render the `Column` component:

```
import { AppContainer } from "./styles"
import { Column } from "./Column"

export const App = () => {
  return (
    <AppContainer>
      <Column text="Todo:" />
    </AppContainer>
  )
}
```

How to define props

You can use a `type` or an `interface` to define the form of your `props` object. Most of the time, types and interfaces can be used interchangeably. We'll get to some differences later in this chapter.

In our `Column` component we definened the props as a `type`:

```
type ColumnProps = {
  text: string
}
```

Or in other words we've defined a `type` with field `text` of type `string` and assigned an alias `ColumnProps` to it. Now if we say that some variable has type `ColumnProps` - it will mean that this variable is an object that has a field `text` of type `string`.

To use this type for our component props we specified it as the type of our functional component first argument:

```
const Column = ({ text }: ColumnProps) => {
        //...
}
```

Here we also immediately destructure the props object to get the `text` field from it.

By default all the fields you define on your types are `required`. It means that if the field will be missing you will get a type error. To make the field optional you can add a question mark before the colon:

```
type ExampleProps = {
  someField?: string
}
```

In this case, TypeScript will conclude that `text` can be `undefined`:

```
(property) ExampleProps.someField?: string | undefined
```

How to accept `children` prop

There are several ways to define the `children` prop on your props type.

Use the `FC` type for the component

The first option is to use the `React.FunctionalComponent` or its alias `React.FC` as your component type:

```
type ParentProps = {
  someProp: any
}

const Parent: React.FC<ParentProps> = ({children, ...props}) => {
  return <>{children}</>
}
```

The `FunctionalComponent` or `FC` is a generic type, so you can pass other props to it to combine them with the `children` prop.

Use `PropsWithChildren`

Alternatively we could use the `React.PropsWithChildren` type that can enhance your props type, and add a definition for `children`.

Here is how `React.PropsWithChildren` type is defined:

```
type React.PropsWithChildren<P> = P & {
  children?: React.ReactNode;
}
```

The letter `P` is a *type argument*. It works similar to function arguments. We can pass an actual type which will be used instead of this letter. For example:

```
type ColumnProps = React.PropsWithChildren<{
  text: string
}>
// will result in the following type
//
// type ColumnProps = {
//   text: string;
// } & {
//   children?: React.ReactNode;
// }
//
```

The ampersand combines the two types into one. In TypeScript this is called a *type intersection*.

```
type ColumnProps = {
  text: string;
} & {
  children?: React.ReactNode;
}

// is the same as:

type ColumnProps = {
  text: string;
  children?: React.ReactNode;
}
```

Define the `children` prop manually

We could also manually add the `children` field to the props type:

```
type ColumnProps = {
  text: string
  children?: React.ReactNode;
}
```

Here we've added an optional field `children` of type `ReactNode`.

Create Card Components

Moving on to the `Card` component. Create a new file `src/Card.tsx`:

```
import { CardContainer } from "./styles"

type CardProps = {
  text: string
}

export const Card = ({ text }: CardProps) => {
  return <CardContainer>{text}</CardContainer>
}
```

It will also accept only the text prop. Define the `CardProps` type for the props with the field `text` of type `string`.

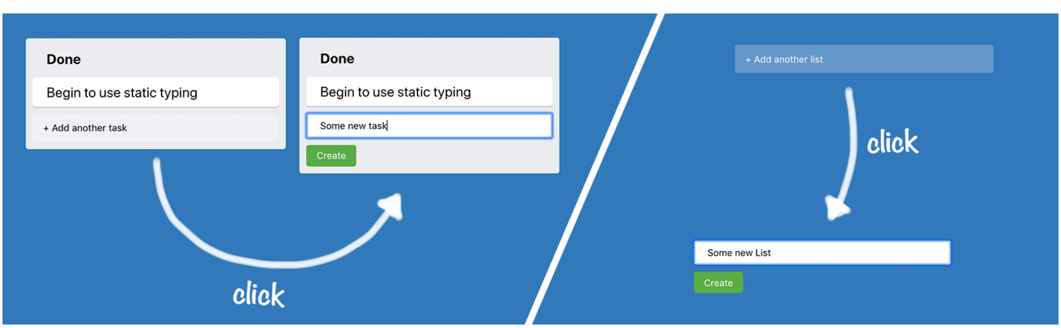
Render everything together

Now render the `Card` component inside the `Column` component. Update the `src/Column.tsx` to look like this:

```
import { ColumnContainer, ColumnTitle } from "./styles"
import { Card } from "./Card"

type ColumnProps = {
  text: string
}

export const Column = ({ text }: ColumnProps) => {
  return (
    <ColumnContainer>
      <ColumnTitle>{text}</ColumnTitle>
      <Card text="Generate app scaffold" />
      <Card text="Learn TypeScript" />
      <Card text="Begin to use static typing" />
    </ColumnContainer>
  )
}
```

Component For Adding New Items

In this lesson, we're going to create a component that will allow us to create new lists and new cards.



AddItemButton

This component will have two states. Initially, it will be a button that says "+ Add another card" or "+ Add another list". When you click this button the component renders an input field and another button saying "Create". When you click the "Create" button it will trigger the callback function that we'll pass as a prop.

Styles For The Button

Open `src/styles.ts` and define a type for `AddItemButtonProps`.

```
type AddItemButtonProps = {
  dark?: boolean
}
```

We'll use the `AddItemButton` component for both lists and tasks. When we use it for lists, it will be rendered on a dark background, so we'll need white color for text. When we use it for tasks, we will render it inside the `Column` component, which already has a light grey background, so we will want the text color to be black.

Button on light and dark background

Now define the `AddNewItemButton` styled-component:

```
export const AddItemButton = styled.button<AddItemButtonProps>`
  background-color: #ffffff3d;
  border-radius: 3px;
  border: none;
  color: ${(props) => (props.dark ? "#000" : "#fff")};
  cursor: pointer;
  max-width: 300px;
  padding: 10px 12px;
  text-align: left;
  transition: background 85ms ease-in;
  width: 100%;
  &:hover {
    background-color: #ffffff52;
  }
`
```

Make sure to define it as `styled.button<AddItemButtonProps>`. If you forget to provide the props type you will have an error on `color` parameter, where we use the value of the prop `dark`.

Create AddNewItem Component. Using State

Create `src/AddNewItem.tsx`, and import the `useState` hook and the `AddItemButton` styles:

```
import { useState } from "react"
import { AddItemButton } from "./styles"
```

This component will accept an item type and some text props for its buttons. Define a type for its props:

```
type AddNewItemProps = {
  onAdd(text: string): void
  toggleButtonText: string
  dark?: boolean
}
```

- `onAdd` is a callback function that will be called when we click the `Create` button.
- `toggleButtonText` is the text we'll render when this component is a button.
- `dark` is a flag that we'll pass to the styled component.

Define the `AddNewItem` component:

```
export const AddNewItem = (props: AddNewItemProps) => {
  const [showForm, setShowForm] = useState(false)
  const { onAdd, toggleButtonText, dark } = props

  if (showForm) {
    // We show item creation form here
  }

  return (
    <AddItemButton dark={dark} onClick={() => setShowForm(true)}>
      {toggleButtonText}
    </AddItemButton>
  )
}
```

It holds a `showForm` boolean state. When this state is `true`, we show an input with the `Create` button. When it's `false`, we render the button with `toggleButtonText` on it.

When you call the `useState` hook you can provide the default value to it. The type of this default value will be used to infer the type of the stored state.

In our case we passed the boolean value `false`, so TypeScript was able to infer that the type of the `showForm` state is `boolean`.

We could also pass the type for the state manually, because `useState` is a generic function and it has a type property `S`:

```
function useState<S>(initialState: S | (() => S)): [S, Dispatch<SetStat\
eAction<S>>]
```

Here you can see that the initial state can have two forms. You can pass the value itself or a function that will return the initial value.

In both cases the value will have the type that comes from the type variable `S`.

If we would need to be more specific about the type of our state - we could provide the type for it manually:

```
const [showForm, setShowForm] = useState<boolean>(false);
```

In this case it is just unnecessary.

Let's add our `AddNewItem` component to the application layout.

Adding New Lists

First let's add the `AddNewItem` to the `App` component. Go to `src/App.tsx` and import the component:

```
import { AddNewItem } from "./AddNewItem"
```

Now add the `AddNewItem` component to the `App` layout:

```
export const App = () => {
  return (
    <AppContainer>
      <Column text="Todo:" />
      <AddNewItem
        toggleButtonText="+ Add another list"
        onAdd={console.log}
      />
    </AppContainer>
  )
}
```

For now, we'll pass `console.log` to our `onAdd` prop.

Adding New Tasks

Open `src/Column.tsx` and import the `AddNewItem` component:

```
import { AddNewItem } from "./AddNewItem"
```

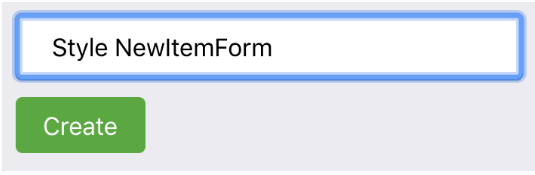
And update the `Column` layout:

```
export const Column = ({ text }: ColumnProps) => {
  return (
    <ColumnContainer>
      <ColumnTitle>{text}</ColumnTitle>
      <Card text="Generate app scaffold" />
      <Card text="Learn TypeScript" />
      <Card text="Begin to use static typing" />
      <AddNewItem
        toggleButtonText="+ Add another card"
        onAdd={console.log}
        dark
      />
    </ColumnContainer>
  )
}
```

NewItemForm component

Styles For The Form

We are aiming to have a form styled like this:



Styled NewItemForm

Define a `NewItemFormContainer` in `src/styles.ts` file.

```
export const NewItemFormContainer = styled.div`
  max-width: 300px;
  display: flex;
  flex-direction: column;
  width: 100%;
  align-items: flex-start;
`
```

Create a `NewItemButton` component with the following styles:

```
export const NewItemButton = styled.button`
  background-color: #5aac44;
  border-radius: 3px;
  border: none;
  box-shadow: none;
  color: #fff;
  padding: 6px 12px;
  text-align: center;
`
```

We want our button to be green and have nice rounded corners.

Define styles for the input as well:

```
export const NewItemInput = styled.input`
  border-radius: 3px;
  border: none;
  box-shadow: #091e4240 0px 1px 0px 0px;
  margin-bottom: 0.5rem;
  padding: 0.5rem 1rem;
  width: 100%;
`
```

Create NewItemForm component

Create a new file `src/NewItemForm.tsx`. Import the `useState` hook and the styled components:

```
import { useState } from "react"
import {
  NewItemFormContainer,
  NewItemButton,
  NewItemInput
} from "./styles"
```

Define the `NewItemFormProps` type:

```
type NewItemFormProps = {
  onAdd(text: string): void
}
```

- `onAdd` is a callback passed through `AddNewItemProps`.

Now define the `NewItemForm` component:

```
export const NewItemForm = ({ onAdd }: NewItemFormProps) => {
  const [text, setText] = useState("")

  return (
    <NewItemFormContainer>
      <NewItemInput
        value={text}
        onChange={(e) => setText(e.target.value)}
      />
      <NewItemButton onClick={() => onAdd(text)}>
        Create
      </NewItemButton>
    </NewItemFormContainer>
  )
}
```

The component uses a controlled input. We'll store the value for it in the `text` state. Whenever you type in the text inside this input, the `text` state is updated.

Here we didn't have to provide any type for the `event` argument of our `onChange` callback. TypeScript gets the type from React type definitions.

Update AddNewItem Component

Import `NewItemForm`:

```
import { NewItemForm } from "./NewItemForm"
```

Add `NewItemForm` to the `AddNewItem` component.

```
export const AddNewItem = (props: AddNewItemProps) => {
  const [showForm, setShowForm] = useState(false)
  const { onAdd, toggleButtonText, dark } = props

  if (showForm) {
    return (
      <NewItemForm
        onAdd={(text) => {
          onAdd(text)
          setShowForm(false)
        }}
      />
    )
  }

  return (
    <AddItemButton dark={dark} onClick={() => setShowForm(true)}>
      {toggleButtonText}
    </AddItemButton>
  )
}
```

Automatically focus on input

To focus on the input we'll use a React feature called refs.

Refs provide a way to reference the actual DOM nodes of rendered React elements.

There are several ways you can define refs in React, we are going to use the hook version.

Create the `useFocus` hook

Create a new file `src/utils/useFocus.ts`:

```
import { useRef, useEffect } from "react"

export const useFocus = () => {
  const ref = useRef<HTMLInputElement>(null)

  useEffect(() => {
    ref.current?.focus()
  }, [])

  return ref
}
```

Here we use the `useRef` hook to get access to the rendered `input` element. TypeScript can't automatically know what the element type will be, so we provide the actual type to it. In our case, we're working with an input so it's `HTMLInputElement`.

> When I need to know what the name is of some element type, I usually check the @types/react/global.d.ts[45] file. It contains type definitions for types that have to be exposed globally (not in `React` namespace).

We use the `useEffect` hook to trigger the focus on the input element. As we've passed an empty dependency array to the `useEffect` callback - it will be triggered only when the component using our hook will be mounted.

If you peek the type of the `ref` object you will see that it is a generic interface that looks like this:

```
interface RefObject<T> {
  readonly current: T | null;
}
```

It has a type variable `T` in our case we specified it to be `HTMLInputElement`. This type is used to describe the field `current` that can have type `T` or `null`.

Note that it is marked as readonly, so you can't reassign the `current` field manually. You will get this error if you try to do it:

[45] https://github.com/DefinitelyTyped/DefinitelyTyped/blob/master/types/react/global.d.ts

> Cannot assign to 'current' because it is a read-only property.ts(2540)

This happened because we specified the default value `null` for our ref. It seems to be an intentional design decision[46]. It is assumed that if you pass `null` as the default value - you want React to manage this ref object, and you don't want the field `current` to be overriden.

You can have a mutable ref as well. Don't pass `null` as a default value, or specify null as a possible ref type:

```
const mutableRef = useRef<HTMLInputElement | null>(null)
// Specify null as a possible value type

const mutableRef = useRef<HTMLInputElement>()
// Or don't pass null as a default value
```

In both casses the type of your ref will be `React.MutableRefObject`:

```
interface MutableRefObject<T> {
  current: T;
}
```

So you will be able to mutate the field `current` of your ref. It is useful when you want to store some data related to your component that should not cause re-renders when you update it.

In our case we want the ref to be immutable, because we pass it to the input component and have no intent of reassigning it manually.

The field `current` can still be `null`. So inside the `useEffect` callback we are using the optional chaining operator (`?.`) to access it.

> In our case the field `current` will never be `null`, because the `useEffect` callback is called after the component is rendered, so the `ref` will already contain the reference to our input element.

[46] https://github.com/DefinitelyTyped/DefinitelyTyped/issues/31065#issuecomment-446425911

Optional chaining operator allows you to access nested fields of an object without explicitly validating that the references to them are valid. So in our case if the `current` will be `null` or `undefined` it just won't call the `focus` method.

Alternatively we could check the value of the `current` field manually:

```
if(inputRef.current){
  inputRef.current.focus()
}
```

So the optional chaining operator is just a nicer way to do it.

Use the `useFocus` hook

Go back to `src/NewItemForm.tsx` and import the hook:

```
import { useFocus } from "./utils/useFocus"
```

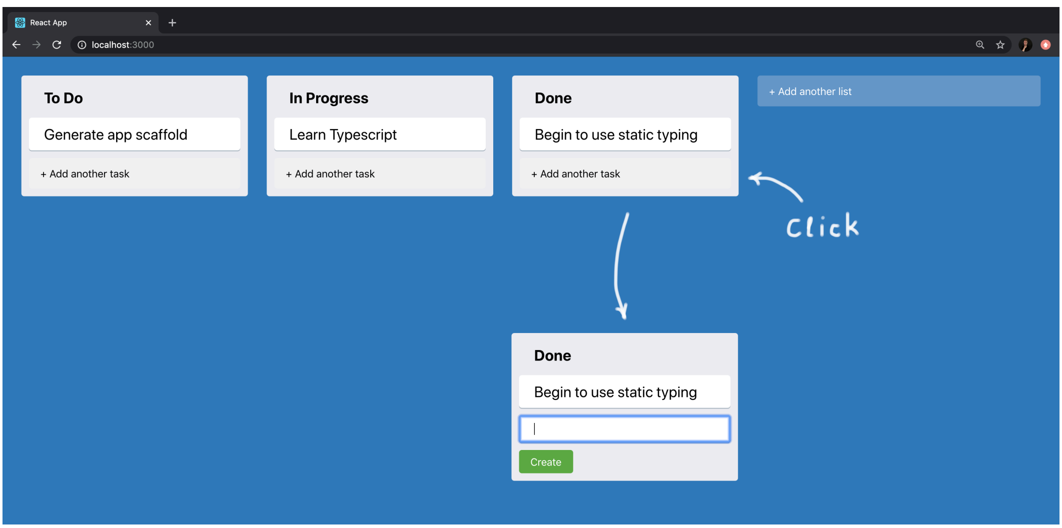
Add it to the component code:

```
type NewItemFormProps = {
  onAdd(text: string): void
}

export const NewItemForm = ({ onAdd }: NewItemFormProps) => {
  const [text, setText] = useState("")
  const inputRef = useFocus()

  return (
    <NewItemFormContainer>
      <NewItemInput
        ref={inputRef}
        value={text}
        onChange={(e) => setText(e.target.value)}
      />
```

```
        <NewItemButton onClick={() => onAdd(text)}>
          Create
        </NewItemButton>
      </NewItemFormContainer>
    )
}
```

We passed the reference that we get from the `useFocus` hook to our `input` element.

If you launch the app and click the new item button, you should see that the form input is focused automatically.



Complete application layout

Submit on enter

Let's make the `NewItemForm` component to submit the input on an `Enter` key press as well, so that the items could be created by pressing the `Enter` key instead of clicking the `Create` button.

To do this we are going to add an `onKeyPress` handler to the text input in the `NewItemForm` component.

Open `NewItemForm` component and add a new function right after the `inputRef` definition:

```
  const handleAddText = (
    event: React.KeyboardEvent<HTMLInputElement>
  ) => {
    if (event.key === "Enter") {
      onAdd(text)
    }
  }
```

Then add the `onKeyPress` event handler to the `NewItemInput` element:

```
      <NewItemInput
        ref={inputRef}
        value={text}
        onChange={(e) => setText(e.target.value)}
        onKeyPress={handleAddText}
      />
```

Here we used the `KeyboardEvent` type from React. You can find the available events in the React documentation[47] and the types for them in the React type definitions[48].

Right now in our `App.tsx` we already pass `console.log` as the `onAdd` prop to the `NewItemForm` element.

Launch the app and try pressing `Enter` after you enter some text into the list-adding input.

You can find the working example for this part in the `code/01-first-app/01.11-submit-on`

[47] https://reactjs.org/docs/events.html

[48] https://github.com/DefinitelyTyped/DefinitelyTyped/blob/14d95eb0fe90f5e0579c49df136cccdfe89b2855/types/react/index.d.ts#L1211

Add Global State And Business Logic. Using the useReducer

In this chapter we will add interactivity to our application.

We'll implement drag-and-drop using the React DnD library, and we will add state management. We won't use any external framework like Redux or Mobx. Instead, we'll throw together a poor man's version of Redux using `useReducer` hook and React context API.

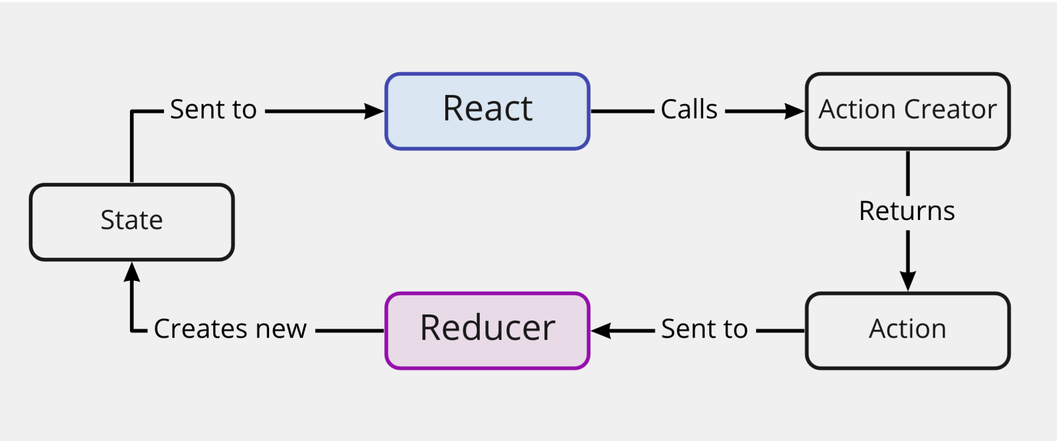
Before we jump into the action I will give a little primer on using `useReducer`.

Using the useReducer

> Disclaimer: The following code is separate from the Trello-clone app and is located in the examples inside the `code/01-first-app/use-reducer` folder.

`useReducer` is a React hook that allows us to manage complex state-like objects with multiple fields.

The main idea is that instead of mutating the original object we always create a new instance with desired values.

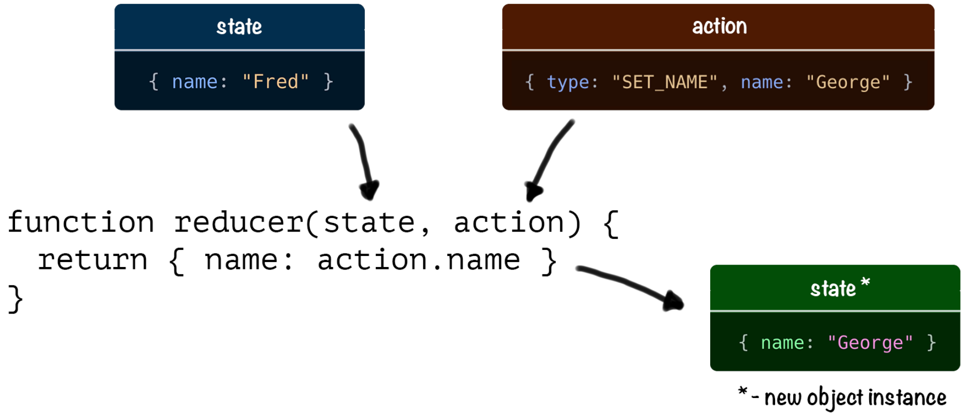


Instead of mutating the object we create a new instance

The state is updated using a special function called *reducer*.

What Is a Reducer?

A reducer is a function that calculates a new state by combining an old state with an action object.



Reducer

Reducer must be a pure function. It means it shouldn't produce any side effects (I/O operations or modifying global state) and for any given input it should return the same output.

Usually a reducer looks like this:

```
function exampleReducer(state, action) {
  switch(action.type){
    case "SOME_ACTION": {
      return { ...state, updatedField: action.payload }
    }
    default:
      return state
  }
}
```

Depending on the passed action `type` field we return a new state value. The key point here is that we always generate a new object that represents the state.

If the passed action type did not match with any of the cases we return the state unchanged.

How to Call useReducer

You can call `useReducer` inside your functional components. On every state change, your component will be re-rendered.

Here's the basic syntax:

```
const [state, dispatch] = useReducer(reducer, initialState)
```

`useReducer` accepts a reducer and initial state. It returns the current `state` paired with a `dispatch` method.

`dispatch` method is used to send actions to the reducer.

`state` contains the current state value from the reducer.

What Are Actions?

Actions are special objects that are passed to the reducer function to calculate the new state.

Actions must contain a `type` field and some field for payload. The `type` field is mandatory. Payload often has some arbitrary name.

Here is an action that could be used to update the `name` field:

```
{ type: "SET_NAME", name: "George" }
```

We pass them to the `dispatch` method provided by the `useReducer` hook:

```
const [ state, dispatch ] = useReducer(reducer, initialState)

dispatch({ type: "SET_NAME", name: "George" })
```

Usually, instead of creating the actions directly, we generate them using special functions called *action creators*:

```
const setName = (name) => ({ type: "SET_NAME", name })
```

The name of the action creator usually matches the `type` field of the action it creates.

After you have the action creator you can use it to dispatch actions like this:

```
const [ state, dispatch ] = useReducer(reducer, initialState)

dispatch(setName("George"))
```

Counter Example

The code for the counter example is in `code/01-first-app/use-reducer`.

Let's look at the reducer first. Open `src/App.tsx`:

```
const counterReducer = (state: State, action: Action) => {
  switch (action.type) {
    case "increment":
      return { count: state.count + 1 }
    case "decrement":
      return { count: state.count - 1 }
    default:
      throw new Error()
  }
}
```

This reducer can process `increment` and `decrement` actions.

This is TypeScript so we must provide types for `state` and `action` attributes.

We'll define the `State` type with a `count: number` field:

```
interface State {
  count: number
}
```

The `action` argument has a mandatory `type` field that we use to decide how should we update our state.

Let's define the `Action` type:

```
type Action =
  | {
      type: "increment"
    }
  | {
      type: "decrement"
    }
```

We've defined it as a `type` having one of the two forms: `{ type: "increment" }` or `{ type: "decrement" }`. In TypeScript this is called a *union type*.

The syntax might look strange because of the leading `"|"` and also because it's spread between multiple lines, but that is how Prettier formats it. Alternatively you could write it like this:

```
type Action = { type: "increment" } | { type: "decrement" }
```

This way it would be more clear. So the leading `"|"` just allows us to define the union type in multiple lines.

You might wonder why didn't we define it as an interface with a field `type: string` like this:

```
interface Action {
  type: string
}
```

But defining our `Action` as a `type` instead of an `interface` gives us a bunch of important advantages. Bear with me — we'll get back to this topic later in the chapter.

For now let's see how can you use this in your components. Here is a counter component that will use the reducer we've defined previously:

```
const App = () => {
  const [state, dispatch] = useReducer(counterReducer, { count: 0 })
  return (
    <>
      <p>Count: {state.count}</p>
      <button onClick={() => dispatch({ type: "decrement" })}>
        -
      </button>
      <button onClick={() => dispatch({ type: "increment" })}>
        +
      </button>
    </>
  )
}
```

Here we call the `dispatch` method inside the `onClick` handlers. With each `dispatch` call we send an `Action` object and then we calculate the new state in our counter reducer.

Define the action creators:

```
const increment = (): Action => ({ type: "increment" })
const decrement = (): Action => ({ type: "decrement" })
```

We define them outside of the component. Specify the return type of them to be our `Action` type.

> Try to create an action creator that would have the `type` field with the value that is not defined on the `Action` type.

Now let's use the action creators instead of creating the action objects manually:

```
const App = () => {
  const [state, dispatch] = useReducer(counterReducer, { count: 0 })
  return (
    <>
      <p>Count: {state.count}</p>
      <button onClick={() => dispatch(decrement())}>-</button>
      <button onClick={() => dispatch(increment())}>+</button>
    </>
  )
}
```

If you launch the app from the examples in the `code/01-first-app/use-reducer` folder you should see a counter with two buttons:

Counter app

Click the buttons to make the number on the counter go up or down.

Now let's get back to our Trello-clone project.

Implement Global State

First let's define a data structure for our application and make it available to all the components through React's Context API.

Create a new file called `src/state/AppStateContext.tsx`. Let's start with the imports:

```
import { createContext, useContext, FC } from "react"
```

We'll use the `createContext` to define the `AppStateContext`, `useContext` to define a helper hook to access the context data easier, and the `FC` type to define the `AppStateProvider` so that it accepts `children`.

Hardcode the data

Define the types for the application state:

```
type Task = {
  id: string
  text: string
}

type List = {
  id: string
  text: string
  tasks: Task[]
}

export type AppState = {
  lists: List[]
}
```

The root type is `AppState` it depends on `List` and `Task` types.

We use arrays to store the lists and the tasks. They will allow us to move the items around because arrays preserve the elements' order.

Both lists and tasks have unique IDs that will allow us to identify them. Also they need to have the `text` field that we'll render inside the components.

I decided to use the terms `Task`/`List` for the data types and `Column`/`Card` for UI components. This way there should be less ambiguity. So if there is a mention of a `Task` - we are talking about the data, and if we are mentioning a `Card` then it is definitely a component. I don't know if that's a good idea, the time will show.

Define the application data — for now let's hardcode it:

```
const appData: AppState = {
  lists: [
    {
      id: "0",
      text: "To Do",
      tasks: [{ id: "c0", text: "Generate app scaffold" }]
    },
    {
      id: "1",
      text: "In Progress",
      tasks: [{ id: "c2", text: "Learn Typescript" }]
    },
    {
      id: "2",
      text: "Done",
      tasks: [{ id: "c3", text: "Begin to use static typing" }]
    }
  ]
}
```

We set the type of this object to `AppState`.

Define the Context

Define the type for the context value and the context itself:

```
type AppStateContextProps = {
  lists: List[]
  getTasksByListId(id: string): Task[]
}

const AppStateContext = createContext<AppStateContextProps>(
  {} as AppStateContextProps
)
```

The `AppStateContextProps` contains two fields: `lists` and `getTasksByListId`. We'll use the `lists` field in the `App` component to render the columns, and the `getTasksByListId` in the `Column` component to render the cards.

React wants us to provide the default value for our context. This value will only be used if we don't wrap our application into our `AppStateProvider`, so we can omit it. To do this, pass an empty object that we'll cast to `AppStateContextProps` to `createContext` function. Here we use an `as` operator to make TypeScript think that our empty object actually has `AppStateContextProps` type:

```
const AppStateContext = createContext<AppStateContextProps>(
  {} as AppStateContextProps
)
```

Define the Context provider

Now define the `AppStateProvider`:

```
export const AppStateProvider: FC = ({ children }) => {
  const { lists } = appData

  const getTasksByListId = (id: string) => {
    return lists.find((list) => list.id === id)?.tasks || []
  }

  return (
    <AppStateContext.Provider value={{ lists, getTasksByListId }}>
      {children}
    </AppStateContext.Provider>
  )
}
```

Inside of this component we defined the `lists` const and the `getTasksByListId` function. We will pass them through the `value` prop of the `AppStateContext.Provider` to make them available to all the context consumers.

Our component will accept `children` as a prop, because we want to be able to wrap components into the `AppStateProvider`. So we specify its type as `FC`.

Go to `src/index.tsx` and wrap the `App` component into the `AppStateProvider`.

```
import React from "react"
import ReactDOM from "react-dom"
import "./index.css"
import { App } from "./App"
import { AppStateProvider } from "./state/AppStateContext"

ReactDOM.render(
  <React.StrictMode>
    <AppStateProvider>
      <App />
    </AppStateProvider>
  </React.StrictMode>,
  document.getElementById("root")
)
```

Now we'll be able to get the `lists` and `getTasksByListId` from any component.

Let's create a custom hook to make it easier to access them.

Using Data From Global Context. Implement Custom Hook

Import the `useContext` hook if you didn't do in on the previous step:

```
import { createContext, useContext, FC } from "react"
```

Then define a custom hook called `useAppState`:

```
export const useAppState = () => {
  return useContext(AppStateContext)
}
```

Inside this hook, we'll get the value from the `AppStateContext` using the `useContext` hook and return the result.

We don't need to specify the types, because TypeScript can derive them automatically based on `AppStateContext` type. Verify this by hovering the `useAppState` hook definition with your mouse and checking its return type.

Get The Data From AppStateContext

Let's update the `Card` component first. As we now need to link the components with the corresponding data we'll need to pass the `id` to them.

Open `src/Card.tsx` and define the `id` field on the `CardProps` type:

```
type CardProps = {
  text: string
  id: string
}
```

Open `src/Column.tsx` and update the `Column` props as well:

```
type ColumnProps = {
  text: string
  id: string
}
```

We'll use the `id` prop to find the corresponding tasks.

Import the `useAppState` hook:

```
import { useAppState } from "./state/AppStateContext"
```

Then change the `Column` layout. We'll call `useAppState` to get the `getTasksByListId` function. Then we use this function to get the tasks to show in this column:

```
export const Column = ({ text, id }: ColumnProps) => {
  const { getTasksByListId } = useAppState()

  const tasks = getTasksByListId(id)

  return (
    <ColumnContainer>
      <ColumnTitle>{text}</ColumnTitle>
      {tasks.map((task) => (
        <Card text={task.text} key={task.id} id={task.id} />
      ))}
      <AddNewItem
        toggleButtonText="+ Add another card"
        onAdd={console.log}
        dark
```

```
      />
    </ColumnContainer>
  )
}
```

Open the `src/App.tsx` file. Use our `useAppState` hook to retrieve the `lists`.

Import the hook:

```
import { useAppState } from "./state/AppStateContext"
```

Update the layout:

```
export const App = () => {
  const { lists } = useAppState()

  return (
    <AppContainer>
      {lists.map((list) => (
        <Column text={list.text} key={list.id} id={list.id} />
      ))}
      <AddNewItem
        toggleButtonText="+ Add another list"
        onAdd={console.log}
      />
    </AppContainer>
  )
}
```

Make sure to pass the `id` to the `Column` component. We'll need it to find the corresponding tasks in the context.

We don't have to specify the type of the loop variable `list`. TypeScript derives it automatically. If we make a typo and instead of `list.text` we write `list.test`, TypeScript will correct us and show a list of available fields.

Now all our components can get the app data from the context. In the next section we'll add some actions and reducers to be able to update the data.

You can find the working example for this part in the `code/01-first-app/01.13-implement`

Define the business logic

In this chapter, we'll define the actions and reducers necessary to create new cards and components. We will provide the reducer's `dispatch` method through the `React.Context` and will use it in our `AddNewItem` component.

We will use Immer to simplify updating the state. Immer will allow us to mutate the state instead of creating a new instance.

Install `use-immer`:

```
yarn add use-immer@0.5.1
```

This library is written in TypeScript so we don't need to install an additional `@types` package.

Create Actions

We'll begin by adding two actions: `ADD_TASK` and `ADD_LIST`. To do this we'll have to define the `Action` type alias.

Create `src/state/actions.ts` and define a new type:

```
export type Action =
  | {
      type: "ADD_LIST"
      payload: string
    }
  | {
      type: "ADD_TASK"
      payload: { text: string; listId: string }
    }
```

We've defined the type alias `Action` and then we've passed two types separated by a vertical line to it. This means that the `Action` type now can resolve to one of the forms that we've passed. So it works like logical inclusive disjunction[49], in other words it is a logical "or".

Each action has an associated `payload` field:

- `ADD_LIST` - contains the list title.
- `ADD_TASK` - `text` is the task text, and `listId` is the reference to the list it belongs to.

We could also define the types in the union using the `interface` syntax:

```
interface AddListAction {
  type: "ADD_LIST"
  payload: string
}

interface AddTaskAction {
  type: "ADD_TASK"
  payload: { text: string; listId: string }
}

type Action = AddListAction | AddTaskAction
```

It would work same way, I just prefer using types.

The technique we are using here is called discriminated union[50].

Each action has a `type` property. This property will be our *discriminant*. It means that TypeScript can look at this property and tell what the other fields of the type will be.

For example, here is an `if` statement:

[49] https://en.wikipedia.org/wiki/Logical_disjunction
[50] https://en.wikipedia.org/wiki/Tagged_union

```
if (action.type === "ADD_LIST") {
  return typeof action.payload
  // Will return "string"
}

if (action.type === "ADD_TASK") {
  return typeof action.payload
  // Will return { text: string; listId: string }
}
```

Here TypeScript already knows that if the `action.type` is `ADD_LIST` then `action.payload` is a `string`, and if the `action.type` is `ADD_TASK` then the payload is going to be an object.

This is one of the things that only types can do.

It will be useful when we'll define our reducers.

Ok, we have the `Action` type, now let's define the action creators. Still inside the `src/state/actions.ts` define and export two functions:

```
export const addTask = (text: string, listId: string): Action => ({
  type: "ADD_TASK",
  payload: {
    text,
    listId
  }
})

export const addList = (text: string): Action => ({
  type: "ADD_LIST",
  payload: text
})
```

Define the appStateReducer

Create a new file `src/state/appStateReducer.ts` it will contain our reducer function.

Import the `Action` type from the `./actions` module:

```
import { Action } from "./actions"
```

Move the `AppState` type definition from the `AppStateContext` to this new `appStateReducer` file:

```
export type Task = {
  id: string
  text: string
}

export type List = {
  id: string
  text: string
  tasks: Task[]
}

export type AppState = {
  lists: List[]
}
```

Export the `List` and the `Task` types as well.

Define and export the `appStateReducer`:

```
export const appStateReducer = (
  draft: AppState,
  action: Action
): AppState | void => {
  switch (action.type) {
  // ...
  }
}
```

Here we call the state a `draft`, because we are using `Immer` and we'll mutate this object directly. This way we remind ourselves that it is not a regular reducer state and we don't have to worry about the immutability.

Adding Lists

Each newly created list will have a unique id, that we'll use to find it in the state. We'll use nanoid[51] to generate them.

Install this library:

```
yarn add nanoid@3.1.22
```

Then import `nanoid` in `src/state/appStateReducer.ts`:

```
import { nanoid } from "nanoid"
```

Add the `ADD_LIST` block to the reducer:

[51] https://github.com/ai/nanoid

```
  switch (action.type) {
    case "ADD_LIST": {
      draft.lists.push({
        id: nanoid(),
        text: action.payload,
        tasks: []
      })
      break
    }
  // ...
}
```

Each list has the `id`, `text` and `tasks` fields. The `id` is the list identifier generated by `nanoid`, the `text` field contains the list's title from the `action.payload`, and the `tasks` is initially an empty array.

Because we are using `Immer` - we can just push the new list to the `draft.lists` array.

Adding Tasks

Adding tasks is a bit more complex because they need to be added to the specific list's `tasks` array.

We'll need a helper function to find the items by their indices.

Create a new file `src/utils/arrayUtils.ts`. We are going to define a function that will accept any object that has a field `id: string`. So we'll define it as a generic function.

Define a new type `Item`.

```
type Item = {
  id: string
}
```

We will use a type variable `TItem` that extends `Item`. This means that we constrained our generic to have the fields that are defined on the `Item` type, in this case the `id` field.

Define the function:

```
export const findItemIndexById = <TItem extends Item>(
  items: TItem[],
  id: string
) => {
  return items.findIndex((item: TItem) => item.id === id)
}
```

Now try to pass in an array of objects that don't have the `id` field:

```
const itemsWithoutId = [{text: "test"}]
findItemIndexById(itemsWithoutId, "testId")
```

You will get a type error:

```
Argument of type '{ text: string; }[]' is not assignable to parameter o\
f type 'Item[]'.
  Property 'id' is missing in type '{ text: string; }' but required in \
type 'Item'.ts(2345)
```

If you remove the constraint and just write `<TItem>` then TypeScript will allow you to pass the `itemsWithoutId` array but will complain that the `id` field is not defined on type `TItem`.

So type constraints guarantee that the items that we pass to the function have the fields defined on the extended type.

> If you followed the instructions on testing out the type constraints — don't forget to remove that code.

Return to `src/state/appStateReducer.ts` and import the `findItemByIndex` function:

```
import { findItemIndexById } from "../utils/arrayUtils"
```

Define the `ADD_TASK` handler:

```
    case "ADD_TASK": {
      const { text, listId } = action.payload
      const targetListIndex = findItemIndexById(draft.lists, listId)

      draft.lists[targetListIndex].tasks.push({
        id: nanoid(),
        text
      })
      break
    }
```

Here we get the `text` and `listId` values by destructuring the `action.payload`. Then we find the array index of the target list using the `findItemIndexById`. After we have the index — we just push the new task object to the target list.

Ok, now our reducer allows us to add lists and tasks, let's implement this in the UI.

Provide Dispatch Through The Context

Open the `src/state/AppStateContext.tsx` and add the imports.

```
import { createContext, useContext, Dispatch, FC } from "react"
import {
  appStateReducer,
  AppState,
  List,
  Task
} from "./appStateReducer"
import { Action } from "./actions"
import { useImmerReducer } from "use-immer"
```

Then add the `dispatch` method to the `AppStateContextProps` definition:

```
type AppStateContextProps = {
  lists: List[]
  getTasksByListId(id: string): Task[]
  dispatch: Dispatch<Action>
}
```

Here we've manually specified the type of the `dispatch` method. Try hovering the variable `dispatch` that we get from the `useImmerReducer`:

```
type React.Dispatch<A> = (value: A) => void
```

This type is generic, so we were able to set our `Action` type as the type for the dispatched actions.

Update the `AppStateProvider`:

```
export const AppStateProvider: FC = ({ children }) => {
  const [state, dispatch] = useImmerReducer(appStateReducer, appData)

  const { lists } = state
  const getTasksByListId = (id: string) => {
    return lists.find((list) => list.id === id)?.tasks || []
  }

  return (
    <AppStateContext.Provider
      value={{ lists, getTasksByListId, dispatch }}
    >
      {children}
    </AppStateContext.Provider>
  )
}
```

Now we get the state value from the reducer and also we provide the `dispatch` method through the context.

Dispatching Actions

Go to `src/App.tsx` and import the `addList` action creator from `src/state/actions.ts`:

```
import { addList } from "./state/actions"
```

Update the `App` component definition:

```
export const App = () => {
  const { lists, dispatch } = useAppState()

  return (
    <AppContainer>
      {lists.map((list) => (
        <Column text={list.text} key={list.id} id={list.id} />
      ))}
      <AddNewItem
        toggleButtonText="+ Add another list"
        onAdd={(text) => dispatch(addList(text))}
      />
    </AppContainer>
  )
}
```

We get the `dispatch` method from the `useAppState` hook and then call it in the `onAdd` callback.

Open `src/Column.tsx` and update it as well. Import the `addTask` action creator:

```
import { addTask } from "./state/actions"
```

Then update the component:

```
export const Column = ({ text, id }: ColumnProps) => {
  const { getTasksByListId, dispatch } = useAppState()
  const tasks = getTasksByListId(id)

  return (
    <ColumnContainer>
      <ColumnTitle>{text}</ColumnTitle>
      {tasks.map((task) => (
        <Card text={task.text} key={task.id} id={task.id} />
      ))}
      <AddNewItem
        toggleButtonText="+ Add another card"
        onAdd={(text) => dispatch(addTask(text, id))}
        dark
      />
    </ColumnContainer>
  )
}
```

Now when the user adds the new task we call the `dispatch` method. We pass the `id` with the `text` because we need to know which list will contain the new task.

Let's launch the app and check that we can create new tasks and lists.

> You can find the working example for this part in the `code/01-first-app/01.14-define-th`

Moving Items

Now that we can add new items, it's time to move them around. We'll start with the columns.

Define the `moveItem` helper function

First we'll define a utility function that will help us to move the items inside the array.

Moving the item means that we remove it from the old position and then add to the new position. Let's define the helper functions for it. Open `src/utils/arrayUtils.ts` and define the `removeItemAtIndex` function:

```
export function removeItemAtIndex<TItem>(
  array: TItem[],
  index: number
) {
  return [...array.slice(0, index), ...array.slice(index + 1)]
}
```

We want to be able to work with arrays with any kind of items in them, so we use a type variable `TItem`.

We use the spread operator to generate a new array with the portion before the `index` that we get using the `slice` method, and the portion after the `index` using the `slice` method with `index + 1`.

Define the `insertItemAtIndex`:

```
export function insertItemAtIndex<TItem>(
  array: TItem[],
  item: TItem,
  index: number
) {
  return [...array.slice(0, index), item, ...array.slice(index)]
}
```

This function is very similar to `removeItemAtIndex`, we also generate a new array from two slices of the original array. The difference is that we put the `item` between the array slices.

Now we can define the `moveItem` function:

```
export const moveItem = <TItem>(
  array: TItem[],
  from: number,
  to: number
) => {
  const item = array[from]
  return insertItemAtIndex(removeItemAtIndex(array, from), item, to)
}
```

First we store the item in the `item` constant. Then we use the `removeItemAtIndex` function to remove the item from its original position and then we insert it back to the new position using the `insertItemAtIndex` function.

Handling the `MOVE_LIST` action

Open `src/state/appStateReducer.ts` and import the `moveItem` function:

```
import { findItemIndexById, moveItem } from "../utils/arrayUtils"
```

Add a new action type to the `Action` union type:

```
| {
    type: "MOVE_LIST"
    payload: {
      draggedId: string
      hoverId: string
    }
  }
```

> Do not override the whole `Action` type. Append this code to the end of the `Action` definition.

Now define the action creator for it:

```
export const moveList = (
  draggedId: string,
  hoverId: string
): Action => ({
  type: "MOVE_LIST",
  payload: {
    draggedId,
    hoverId
  }
})
```

We've added the `MOVE_LIST` action. This action has `draggedId` and `hoverId` in its payload. When we start dragging the column, we remember its id and pass it as `draggedId`. When we hover over other columns we take their ids and use them as a `hoverId`.

Add a new `case` block to the `appStateReducer`:

```
case "MOVE_LIST": {
  const { draggedId, hoverId } = action.payload
  const dragIndex = findItemIndexById(draft.lists, draggedId)
  const hoverIndex = findItemIndexById(draft.lists, hoverId)
  draft.lists = moveItem(draft.lists, dragIndex, hoverIndex)
  break
}
```

Here we take the `draggedId` and the `hoverId` from the action payload. Then we calculate the indices of the dragged and the hovered columns. And then we override the `draft.lists` value with the result of the `moveItem` function, which takes the source array, and two indices that it swaps.

Add Drag and Drop (Install React DnD)

To implement drag and drop we will use the `react-dnd` library. This library has several adapters called backends to support different APIs. For example to use `react-dnd` with HTML5 we will use `react-dnd-html5-backend`.

Install the library:

```
yarn add react-dnd@14.0.1 react-dnd-html5-backend@14.0.0
```

`react-dnd` has built-in type definitions, so we don't have to install them separately.

Open `src/index.tsx` and add `DndProvider` to the layout.

```
import React from "react"
import ReactDOM from "react-dom"
import "./index.css"
import { App } from "./App"
import { DndProvider } from "react-dnd"
import { HTML5Backend as Backend } from "react-dnd-html5-backend"
import { AppStateProvider } from "./state/AppStateContext"

ReactDOM.render(
  <React.StrictMode>
    <DndProvider backend={Backend}>
      <AppStateProvider>
        <App />
      </AppStateProvider>
    </DndProvider>
  </React.StrictMode>,
  document.getElementById("root")
)
```

This provider will add a dragging context to our app. It will allow us to use `useDrag` and `useDrop` hooks inside our components.

Define The Type For Dragging

When we begin to drag an item we'll provide information about it to `react-dnd`. We'll pass an object that will describe the item we are currently dragging. This object will have a `type` field that for now will be `COLUMN`. We'll also pass the column's `id` and `text` that we'll get from the `Column` component.

Create a new file `src/DragItem.ts`. Define a `ColumnDragItem` and assign it to the `DragItem` type:

```
export type ColumnDragItem = {
  id: string
  text: string
  type: "COLUMN"
}

export type DragItem = ColumnDragItem
```

Later the `DragItem` will be a union type, and we will add the `CardDragItem` type to it.

Store The Dragged Item In The State

Let's store the dragged item in our app state. Go to `src/state/appStateReducer.ts` and import the `DragItem` type:

```
import { DragItem } from "../DragItem"
```

Update the `AppState` type:

```
export type AppState = {
  lists: List[]
  draggedItem: DragItem | null
}
```

Go to `src/state/AppStateContext.tsx` and update the `appData` constant, add the `draggedItem` field with value `null` to it:

```
const appData: AppState = {
  draggedItem: null,
  // ...
}
```

Add the `draggedItem` field to the `AppStateContextProps`:

```
import { DragItem } from "../DragItem"
  // ...
type AppStateContextProps = {
  draggedItem: DragItem | null
  lists: List[]
  getTasksByListId(id: string): Task[]
  dispatch: Dispatch<Action>
}
```

Don't forget to import the `DragItem` type.

Then update the `AppStateProvider` so it provides the `draggedItem` through the context:

```
export const AppStateProvider: FC = ({ children }) => {
  const [state, dispatch] = useImmerReducer(appStateReducer, appData)

  const { draggedItem, lists } = state
  const getTasksByListId = (id: string) => {
    return lists.find((list) => list.id === id)?.tasks || []
  }

  return (
    <AppStateContext.Provider
      value={{ draggedItem, lists, getTasksByListId, dispatch }}
    >
      {children}
    </AppStateContext.Provider>
  )
}
```

In the `src/state/actions.ts` add a new action type `SET_DRAGGED_ITEM` to the `Action` union type, don't forget to import the `DragItem` type here as well:

```
import { DragItem } from "../DragItem"
  // ...
  | {
      type: "SET_DRAGGED_ITEM"
      payload: DragItem | null
    }
```

It will hold the `DragItem` that we defined earlier. We want to be able to set it to `null` if we are not dragging anything. We are not using the `undefined` here because it would mean that the field could be omitted. In our case it's not true, it can just be empty sometimes.

Define the action creator:

```
export const setDraggedItem = (
  draggedItem: DragItem | null
): Action => ({
  type: "SET_DRAGGED_ITEM",
  payload: draggedItem
})
```

Add a new `case` block to `appStateReducer`:

```
case "SET_DRAGGED_ITEM": {
  draft.draggedItem = action.payload
  break
}
```

In this block, we set the `draggedItem` field of our draft state to whatever we get from the `action.payload`.

Define The useItemDrag Hook

The dragging logic will be similar for both cards and columns. I suggest we move it to a custom hook.

This hook will return a `drag` method that accepts the `ref` of a draggable element. Whenever we start dragging the item, the hook will dispatch a `SET_DRAG_ITEM` action to save the item in the app state. When we stop dragging, it will dispatch this action again with `null` as the payload.

Create a new file `src/utils/useItemDrag.ts`. Inside of it write the following:

```
import { useDrag } from "react-dnd"
import { useAppState } from "../state/AppStateContext"
import { DragItem } from "../DragItem"
import { setDraggedItem } from "../state/actions"

export const useItemDrag = (item: DragItem) => {
  const { dispatch } = useAppState()
  const [, drag] = useDrag({
    type: item.type,
    item: () => {
      dispatch(setDraggedItem(item))
      return item
    },
    end: () => dispatch(setDraggedItem(null))
  })
  return { drag }
}
```

Internally this hook uses `useDrag` from `react-dnd`. We pass an `options` object to it.

- `type` - it will be `CARD` or `COLUMN`
- `item` - returns dragged item object and dispatches the `SET_DRAGGED_ITEM` action
- `end` - is called when we release the item

As you can see inside this hook we dispatch the new `SET_DRAGGED_ITEM` action. When we start dragging, we store the `item` in our app state, and when we stop, we reset it to `null`.

The `useDrag` hook returns three values inside the array:

- `[0]` - Collected Props: An object containing collected properties from the collect function. If no collect function is defined, an empty object is returned.
- `[1]` - DragSource Ref: A connector function for the drag source. This must be attached to the draggable portion of the DOM.
- `[2]` - DragPreview Ref: A connector function for the drag preview. This may be attached to the preview portion of the DOM.

It is a common pattern with hooks, because it allows us to destructure this array and assign its values to variables that have the names we want.

> An example of this is the `useState` hook that returns two values inside the array:
>
> - `[0]` - getter, allows us to get the state value.
> - `[1]` - setter function, allows us to update the state value.
>
> It allows us to call the getter and the setter however we want. For example `const [fruit, setFruit] = useState("apple")`.

In our hook we don't need the Collected Props object, so we skip it which leaves us with this a hanging comma in the beginning. The syntax might look a bit awkward, but really we are just skipping the value that we aren't going to use.

Drag Column

Let's implement dragging for the `Column` component.

Import the `useRef` and the `useItemDrag` hook that we've just defined:

```
import { useRef } from "react"
import { useItemDrag } from "./utils/useItemDrag"
```

Define the `ref` constant that will hold the reference to the dragged `div` element.

```
const ref = useRef<HTMLDivElement>(null)
```

We need a `ref` to specify the drag target. Here we know that it will be a `div` element. We manually provide the `HTMLDivElement` type to `useRef` call.

Pass the ref to the `ColumnContainer` element:

```
<ColumnContainer ref={ref}>
```

We will use the `useItemDrag` hook to find out when did the user begin dragging the column.

```
const { drag } = useItemDrag({ type: "COLUMN", id, text })
```

We pass an object that represents the dragged item. We say that it's a `COLUMN` and then we pass the `id` and `text` properties. This hook returns the `drag` function. We'll pass the column ref to it later.

To find a place to drop the column we'll use other columns as drop targets. So when we hover over another column we dispatch a `MOVE_LIST` action to swap the dragged and target column positions.

First of all we'll need to know what are we dragging, so let's get the `draggedItem` from the state:

```
const { draggedItem, getTasksByListId, dispatch } = useAppState()
```

The hover event might be triggered too frequently, so we'll use the `throttle` function from the `throttle-debounce-ts` package.

Install the package:

```
yarn add throttle-debounce-ts
```

Add the imports, you will need `useDrop` from `react-dnd`, `throttle` from `throttle-debounce-ts`, and `moveList` from `src/state/actions.ts`:

```
import { useDrop } from "react-dnd"
import { moveList, addTask } from "./state/actions"
import { throttle } from "throttle-debounce-ts"
```

Now add the call to `useDrop` at the beginning of the `Column` component right after the `useRef` call:

```
  const ref = useRef<HTMLDivElement>(null)
  const [, drop] = useDrop({
    accept: "COLUMN",
    hover: throttle(200, () => {
      if (!draggedItem) {
        return
      }
      if (draggedItem.type === "COLUMN") {
        if (draggedItem.id === id) {
          return
        }

        dispatch(moveList(draggedItem.id, id))
      }
    })
  })
```

Here we pass the accepted item type and then define the throttled `hover` callback. Inside of it we first check that the `draggedItem` exists, then we check that we are dragging a column and that the dragged item and hovered item IDs are different. If everything is fine we dispatch a `MOVE_LIST` action.

Now we can combine the `drag` and the `drop` functions. Add this code right before the component return statement:

```
drag(drop(ref))
```

Launch the app, you should be able to drag the columns.

> You can find the working example for this part in the `code/01-first-app/01.19-drag-colu`

Hide The Dragged Item

Styles For DragPreviewContainer

If you try to drag the column around, you will see that the original dragged column is still visible.

Let's go to `src/styles.ts` and add an option to hide it.

We'll need to reuse this logic, so we'll move it out to `DragPreviewContainer`.

```
interface DragPreviewContainerProps {
  isHidden?: boolean
}

export const DragPreviewContainer = styled.div<DragPreviewContainerProp\
s>`
  opacity: ${(props) => (props.isHidden ? 0.3 : 1)};
`
```

For now, we won't hide the column completely - we'll just make it semitransparent. Set the `opacity` in the hidden state to `0.3`. This way we'll see the hidden element. Later we'll change this value to `0` to hide the element completely.

Now update the `ColumnContainer` to extend the `DragPreviewContainer`:

```
export const ColumnContainer = styled(DragPreviewContainer)`
  background-color: #ebecf0;
  width: 300px;
  min-height: 40px;
  margin-right: 20px;
  border-radius: 3px;
  padding: 8px 8px;
  flex-grow: 0;
`
```

As you can see the `styled` namespace that we used to define the styles for the `div` elements before can also be used as a function. This way we can extend the styled components that we defined earlier.

> Read more about the `styled` factory in the Styled Components documentation[52]

While we are still in the `src/styles.ts`, let's update the `CardContainer` as well, make it extend the `DragPreviewContainer`:

```
export const CardContainer = styled(DragPreviewContainer)`
  background-color: #fff;
  cursor: pointer;
  margin-bottom: 0.5rem;
  padding: 0.5rem 1rem;
  max-width: 300px;
  border-radius: 3px;
  box-shadow: #091e4240 0px 1px 0px 0px;
`
```

Calculate The isHidden Flag

Let's add a helper method to calculate if we need to hide the column.

Create a new file `src/utils/isHidden.ts` with the following code:

[52]https://styled-components.com/docs/api

```
import { DragItem } from "../DragItem"

export const isHidden = (
  draggedItem: DragItem | null,
  itemType: string,
  id: string
): boolean => {
  return Boolean(
    draggedItem &&
      draggedItem.type === itemType &&
      draggedItem.id === id
  )
}
```

This function compares the `type` and `id` of the currently dragged item with the `type` and `id` we pass to it as arguments.

Go to `src/Column.tsx` and import the `isHidden` function:

```
import { isHidden } from "./utils/isHidden"
```

Update the layout. We now pass the result of `isHidden` function to the `isHidden` prop of our `ColumnContainer`:

```
<ColumnContainer
  ref={ref}
  isHidden={isHidden(draggedItem, "COLUMN", id)}
>
```

Now you can launch the app and verify that we are actually hiding the items that we are dragging.

> You can find the working example for this part in the `code/01-first-app/01.20-hide-drag`

Implement The Custom Dragging Preview

React DnD allows you to have a custom element that will represent the dragged item preview. To implement this feature we'll have to use the `customDragLayer` from `react-dnd`

We want a container component to render the preview. It needs to have `position: fixed` and should take up the whole screen size.

It is going to be a separate layer that will be rendered on top of all the other elements. We will render our dragging preview inside of it. Having `position: fixed` will allow us to specify the dragging preview position relative to this container.

Define a new styled component in `src/styles.ts`:

```
export const CustomDragLayerContainer = styled.div`
  height: 100%;
  left: 0;
  pointer-events: none;
  position: fixed;
  top: 0;
  width: 100%;
  z-index: 100;
`
```

We want this container to be rendered on top of any other element on the page, so we provide `z-index: 100`. Also, we specify `pointer-events: none` so it will ignore all mouse events.

Now create a new file `src/CustomDragLayer.tsx` and add the imports:

```
import { useDragLayer } from "react-dnd"
import { Column } from "./Column"
import { CustomDragLayerContainer } from "./styles"
import { useAppState } from "./state/AppStateContext"
```

- `useDragLayer` - will provide us the information about the dragged item.

- `Column` - it is going to be our dragged element
- `CustomDragLayerContainer` - is our dragging layer, we'll render the dragging preview inside of it.
- `useAppState` - we will get the `draggedItem` from it

Define the `CustomDragLayer` component:

```
export const CustomDragLayer = () => {
  const { draggedItem } = useAppState()
  const { currentOffset } = useDragLayer((monitor) => ({
    currentOffset: monitor.getSourceClientOffset()
  }))

  return draggedItem && currentOffset ? (
    <CustomDragLayerContainer>
      <Column
        id={draggedItem.id}
        text={draggedItem.text}
  // ...
      />
    </CustomDragLayerContainer>
  ) : null
}
```

Here we get the `draggedItem` from the application state using the `useAppState` hook and `currentOffset` value from the `useDragLayer` hook.

The `useDragLayer` hook allows us to get the information from the React-DnD internal state. To do this we pass a collector function to it, that has access to the `monitor` object. We don't need to specify the type of the `monitor` argument, because TypeScript will infer it from the `useDragLayer` type definition:

```
declare function useDragLayer<CollectedProps>(
  collect: (monitor: DragLayerMonitor) => CollectedProps
): CollectedProps;
```

We can see that the `useDragLayer` is a generic function that has a type placeholder called `CollectedProps`. The actual type of this placeholder will be inferred from the return value of the collector function that we'll pass to the `useDragLayer`. So to get the correct types for the `useDragLayer` returned values we need to type the returned values of our collector function properly.

We want to collect the curren position of the dragged item from the `monitor`. To do this we use the `currentOffset` it is an object that contains the `x` and `y` coordinates of the dragged item.

We don't have to worry about the `currentOffset` type, because it is correctly defined as the return value of the `monitor.getSourceClientOffset` method.

We'll use the `currentOffset` value to provide the position to the dragged item. But first we need to fix another problem.

Prevent The Column Preview From Hiding

Right now if you launch the app - you will see that the column preview is also getting hidden. This happens because inside the `Column` component we compare the `type` and the `id` of the column with the `type` and the `id` field of the dragged item. If they match - the `isHidden` function returns `true` and we hide the element.

In case of the preview component those fields will always match, because we get them from the dragged item object. To fix this let's mark the preview components with a special flag.

First let's modify the `ColumnContainer`. Open `src/styles.ts` and add the `isPreview` prop to the `DragPreviewContainerProps`:

```
type DragPreviewContainerProps = {
  isHidden?: boolean
  isPreview?: boolean
}

export const DragPreviewContainer = styled.div<DragPreviewContainerProp\
s>`
  transform: ${(props) =>
    props.isPreview ? "rotate(5deg)" : undefined};
  opacity: ${(props) => (props.isHidden ? 0.3 : 1)};
`
```

Here we also use this new prop to tilt the preview container a bit, just like it happens in the real Trello application. We do it by adding the `transform` property that will be `rotate(5deg)` if the `isPreview` prop is `true`.

At this point we don't need to make the dragged columns semitransparent so we set the hidden state `opacity` to `0`.

Then let's add the `isPreview` flag to the `isHidden` function. Open `src/utils/isHidden.ts` and add a new `boolean` argument `isPreview`:

```
export const isHidden = (
  draggedItem: DragItem | null,
  itemType: string,
  id: string,
  isPreview?: boolean
): boolean => {
  return Boolean(
    !isPreview &&
      draggedItem &&
      draggedItem.type === itemType &&
      draggedItem.id === id
  )
}
```

Open the `src/Column.tsx` and add a new prop `isPreview`:

```
type ColumnProps = {
  text: string
  id: string
  isPreview?: boolean
}
```

We make this prop optional so we don't have to pass the `isPreview` to the regular columns.

Now get the `isPreview` inside the component and pass it to the `ColumnContainer` and to the `isHidden` function:

```
export const Column = ({ text, id, isPreview }: ColumnProps) => {
  // ...
  return (
    <ColumnContainer
      isPreview={isPreview}
      ref={ref}
      isHidden={isHidden(draggedItem, "COLUMN", id, isPreview)}
    >
  // ...
    </ColumnContainer>
  )
}
```

> Do not remove the omitted parts of the code. I've skipped them only because we don't change them here. To see how your file should look at this point check the `code/01-first-app/01.21-implement-the-custom-dragging-preview/src`

Now we can pass the `isPreview` flag to the column preview in the `CustomDragLayer` component:

```
<Column
  id={draggedItem.id}
  text={draggedItem.text}
  isPreview
/>
```

After it's done add the `CustomDragLayer` component to the `App` component layout. Open `src/App.tsx`, import `CustomDragLayer` and add it to the `App` layout above the columns:

```
import { CustomDragLayer } from "./CustomDragLayer"
  // ...
export const App = () => {
  const { lists, dispatch } = useAppState()

  return (
    <AppContainer>
      <CustomDragLayer />
      {lists.map((list) => (
        <Column text={list.text} key={list.id} id={list.id} />
      ))}
      <AddNewItem
        toggleButtonText="+ Add another list"
        onAdd={(text) => dispatch(addList(text))}
      />
    </AppContainer>
  )
}
```

Move The Dragged Item Preview

Right now we are only rendering the preview component. We need to write some extra code to make it follow the cursor.

We will create a styled component that will get the dragged item coordinates from `react-dnd` and generate the styles with the `transform` attribute to move the preview around.

Open `src/styles.ts` and define the props for this styled component:

```
type DragPreviewWrapperProps = {
  position: {
    x: number
    y: number
  }
}
```

It will receive a prop `position` with the `x` and `y` coordinates.

Now define the styled component:

```
export const DragPreviewWrapper = styled.div.attrs<DragPreviewWrapperPr\
ops>(
  ({ position: { x, y } }) => ({
    style: {
      transform: `translate(${x}px, ${y}px)`
    }
  })
)<DragPreviewWrapperProps>``
```

By default for every property passed to the styled component it will automatically generate a CSS class. It has a big performance overhead. To avoid this we use the attrs[53] method. This way it will assign the `styles` attribute to our component instead of generating a new class every time the position of the preview changes.

Note that we are passing the type of the props twice. The first time we do it to provide the type for the attributes that we are passing, and the second time we do it to define the props of the resulting component.

Go back to `src/CustomDragLayer.tsx` and import the `DragPreviewWrapper` from the styles:

[53] https://styled-components.com/docs/api#attrs

```
import {
  CustomDragLayerContainer,
  DragPreviewWrapper
} from "./styles"
```

Then wrap the `Column` component into the `DragPreviewWrapper`. Pass the `currentOffset` to the `DragPreviewWrapper`.

```
<DragPreviewWrapper position={currentOffset}>
  <Column
    id={draggedItem.id}
    text={draggedItem.text}
    isPreview
  />
</DragPreviewWrapper>
```

Now the preview item should actually follow the cursor.

Hide The Default Drag Preview

To hide the default drag preview we'll have to modify the `useItemDrag` hook.

Open `src/utils/useItemDrag.ts`. We'll use the `getEmptyImage` function to create the preview that won't be rendered. Import the function from `react-dnd-html5-backend`:

```
import { getEmptyImage } from "react-dnd-html5-backend"
```

Also import the `useEffect` hook from `react`:

```
import { useEffect } from "react"
```

Now add a new `useEffect` call in the end of our hook:

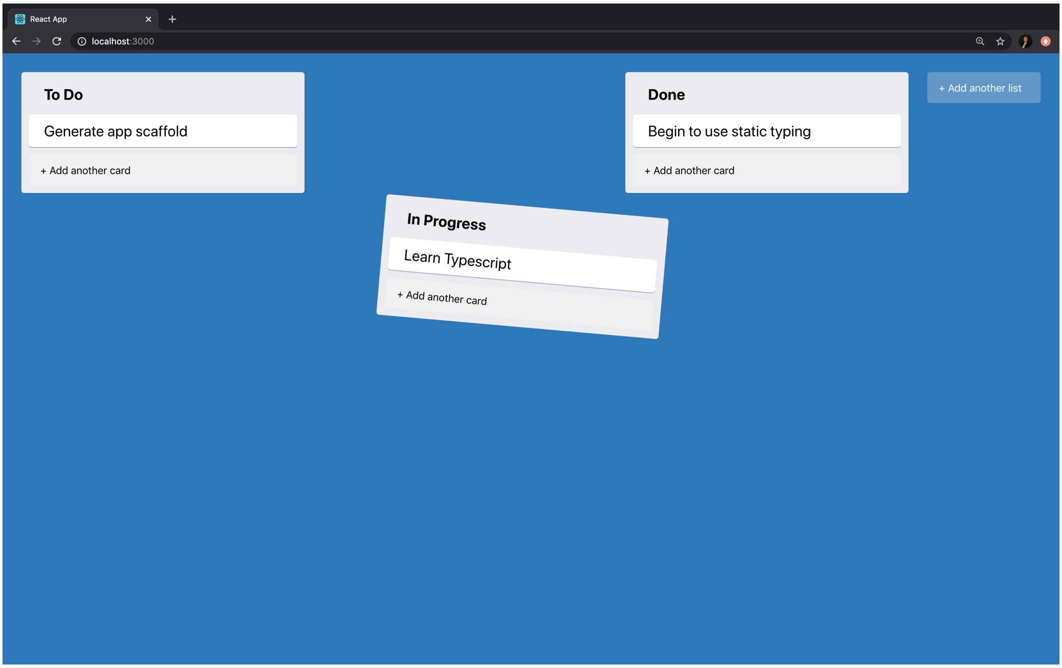
```
export const useItemDrag = (item: DragItem) => {
  const { dispatch } = useAppState()
  const [, drag, preview] = useDrag({
    type: item.type,
    item: () => {
      dispatch(setDraggedItem(item))
      return item
    },
    end: () => dispatch(setDraggedItem(null))
  })
  useEffect(() => {
    preview(getEmptyImage(), { captureDraggingState: true })
  }, [preview])
  return { drag }
}
```

Get the `preview` function from `useDrag`. The `preview` function accepts an element or node to use as a drag preview. This is where we use `getEmptyImage`.

Now we can also go to styled and change the opacity of the dragged item from 0.3 to 0.

```
export const DragPreviewContainer = styled.div<DragPreviewContainerProp\
s>`
  transform: ${(props) =>
    props.isPreview ? "rotate(5deg)" : undefined};
  opacity: ${(props) => (props.isHidden ? 0 : 1)};
`
```

Launch the app. Now you can drag columns around and they will have a nice little tilt to them!



Tilted column drag preview

You can find the working example for this part in the `code/01-first-app/01.23-hide-the-c`

Drag Cards

It's time to drag the cards around. First we need to add a new `Action` type. Open `src/state/actions.ts` and add a `MOVE_TASK` action:

```
| {
    type: "MOVE_TASK"
    payload: {
      draggedItemId: string
      hoveredItemId: string | null
      sourceColumnId: string
      targetColumnId: string
    }
  }
```

This action accepts `draggedItemId` and `hoveredItemId` just like `MOVE_LIST`, but it also needs to know between which columns we are dragging the card. So - it also contains the `sourceColumnId` and the `targetColumnId` attributes that hold source and target column ids.

Define the action creator as well:

```
export const moveTask = (
  draggedItemId: string,
  hoveredItemId: string | null,
  sourceColumnId: string,
  targetColumnId: string
): Action => ({
  type: "MOVE_TASK",
  payload: {
    draggedItemId,
    hoveredItemId,
    sourceColumnId,
    targetColumnId
  }
})
```

Open `src/DragItem.ts` and add the `CardDragItem` type.

```
export type CardDragItem = {
  id: string
  columnId: string
  text: string
  type: "CARD"
}

export type ColumnDragItem = {
  id: string
  text: string
  type: "COLUMN"
}

export type DragItem = CardDragItem | ColumnDragItem
```

Update the `DragItem` type to be either a `CardDragItem` or a `ColumnDragItem`.

Our `CardDragItem` also has the `columnId` property. We need this value to know in which column should the card be located. Let's add this property to the `Card` component.

Open `src/Card.tsx` and add `columnId` to the props:

```
type CardProps = {
  text: string
  id: string
  columnId: string
  isPreview?: boolean
}
```

Here we also get the `isPreview` prop, to avoid hiding the card that we render in the `CustomDragLayer` component.

Get these props from the destructured props object:

```
export const Card = ({
  text,
  id,
  columnId,
  isPreview
}: CardProps) => {
  // ...
}
```

Now we can pass the `columnId` to our `Card` components. Open the `src/Column.tsx` and pass the `id` as the `columnId` to the cards:

```
<Card
  columnId={id}
  text={task.text}
  key={task.id}
  id={task.id}
/>
```

After it's done switch back to the `src/Card.tsx` and add the imports:

```
import { useRef } from "react"
import { CardContainer } from "./styles"
import { useItemDrag } from "./utils/useItemDrag"
import { useDrop } from "react-dnd"
import { useAppState } from "./state/AppStateContext"
import { isHidden } from "./utils/isHidden"
import { moveTask, setDraggedItem } from "./state/actions"
import { throttle } from "throttle-debounce-ts"
```

Get the `draggedItem` and `dispatch` from the `useAppState`, get the `CardContainer` ref and update the card layout:

```
const { draggedItem, dispatch } = useAppState()
const ref = useRef<HTMLDivElement>(null)
// ...
return (
  <CardContainer
    isHidden={isHidden(draggedItem, "CARD", id, isPreview)}
    isPreview={isPreview}
    ref={ref}
  >
    {text}
  </CardContainer>
)
```

Pass the `ref`, `isHidden` and `isPreview` props to the `CardContainer`.

Call the `useItemDrag` hook to get the `drag` function. Add the following code right after the `useRef` call:

```
const { drag } = useItemDrag({
  type: "CARD",
  id,
  text,
  columnId
})
```

This code is very similar to what we had in the `Column` component. The main difference is that the `type` field is `CARD` now.

Next we need to enable our cards to be drop targets. Add this `useDrop` block right after the `useItemDrag` call:

```
  const [, drop] = useDrop({
    accept: "CARD",
    hover: throttle(200, () => {
      if (!draggedItem) {
        return
      }
      if (draggedItem.type !== "CARD") {
        return
      }
      if (draggedItem.id === id) {
        return
      }

      dispatch(
        moveTask(draggedItem.id, id, draggedItem.columnId, columnId)
      )
      dispatch(setDraggedItem({ ...draggedItem, columnId }))
    })
  })
```

Inside the `hover` callback we check that we aren't hovering the item we currently drag. If the ids are equal, we just return.

Then we take the `draggedItem.id` and `draggedItem.columnId` from the dragged item, and `id` and `columnId` from the hovered card.

We dispatch those values inside the `MOVE_TASK` action payload.

We also dispatch the `SET_DRAGGED_ITEM` action, because we might have dragged our item into another column, so the `columnId` might have changed.

After it's done, wrap the `ref` into the `drag` and the `drop` function calls, just like we did in our `Column` component:

```
drag(drop(ref))
```

Update CustomDragLayer

Open `src/CustomDragLayer.tsx` and import the `Card` component:

```
import { Card } from "./Card"
```

Then add a ternary operator to the layout to check what we are dragging:

```
{draggedItem.type === "COLUMN" ? (
  <Column
    id={draggedItem.id}
    text={draggedItem.text}
    isPreview
  />
) : (
  <Card
    columnId={draggedItem.columnId}
    isPreview
    id={draggedItem.id}
    text={draggedItem.text}
  />
)}
```

Update The Reducer

We also need to add a new `MOVE_TASK` case block to our reducer:

```
  case "MOVE_TASK": {
// ...
  }
```

Then inside this block we destructure the `action.payload` like this:

```
const {
  draggedItemId,
  hoveredItemId,
  sourceColumnId,
  targetColumnId
} = action.payload
```

Then we get the source and target list indices:

```
const sourceListIndex = findItemIndexById(
  draft.lists,
  sourceColumnId
)
const targetListIndex = findItemIndexById(
  draft.lists,
  targetColumnId
)
```

Then we find the indices of the dragged and hovered items:

```
    const dragIndex = findItemIndexById(
      draft.lists[sourceListIndex].tasks,
      draggedItemId
    )
// ...
    const hoverIndex = hoveredItemId
      ? findItemIndexById(
          draft.lists[targetListIndex].tasks,
          hoveredItemId
        )
      : 0
```

Here we return `0` if the index for the `hoverId` could not be found. It is possible because when we'll drag the card to an empty column we'll pass `null` as `hoverId` for the card.

After we have them store the moved item in a variable:

```
const item = draft.lists[sourceListIndex].tasks[dragIndex]
```

And now we can remove the item from the source list and add it to the target list:

```
      // Remove the task from the source list
      draft.lists[sourceListIndex].tasks.splice(dragIndex, 1)

      // Add the task to the target list
      draft.lists[targetListIndex].tasks.splice(hoverIndex, 0, item)
      break
```

Now - launch the app and enjoy dragging the cards around. Soon you'll see that after you've moved all the cards from a column, you can't move them back. Let's fix that.

> You can find the working example for this part in the `code/01-first-app/01.26-update-th`

Drag the Card To an Empty Column

Let's make it possible to move the cards to an empty column.

To implement this functionality, we'll use columns as a drop target for our cards as well.

This way if the column is empty, and we drag and drop a card over it, the card will be moved to this empty column.

To do this we'll edit our `Column.tsx` drop `hover` code and add `CARD` to supported item types.

```
accept: ["COLUMN", "CARD"],
```

Now inside of our `hover` callback, we'll need to check what the actual type of our dragged item is. The `draggedItem` has a `DragItem` type which is a union of `ColumnDragItem` and `CardDragItem`. Both `ColumnDragItem` and `CardDragItem` have a common field `type` that we can use to discriminate the `DragItem`.

Add an `if` block. If our `draggedItem.type` is `COLUMN`, then we do what we did before. Just leave the previous logic there.

Import the `moveTask` action creator:

```
import {
  addTask,
  moveTask,
  moveList,
  setDraggedItem
} from "./state/actions"
```

Then add the following code to the `useDrop` hook:

```
      if (draggedItem.type === "COLUMN") {
  // ...
      } else {
        if (draggedItem.columnId === id) {
          return
        }
        if (tasks.length) {
          return
        }

        dispatch(
          moveTask(draggedItem.id, null, draggedItem.columnId, id)
        )
        dispatch(setDraggedItem({ ...draggedItem, columnId: id }))
      }
```

> Don't remove the code in the `draggedItem.type === "COLUMN"` block. It should still contain the column dragging logic.

Here we have almost the same code as in the `Card` component.

There are a few differences though. We pass `null` as the hovered item id there, because we are literally hovering an empty space inside the column. And also we dispatch the `setDraggedItem` action to update the `columnId` of the dragged item.

Now launch the app and check that everything works.

> You can find the working example for this part in the `code/01-first-app/01.27-drag-the-`

Saving State On Backend. How To Make Network Requests

In this chapter, we'll learn to work with network requests.

Network requests are tricky. They are resolved only during run-time, so you have to account for that when you write your TypeScript code.

In previous sections, we wrote a kanban board application where you can create tasks, organize them into lists and drag them around.

Let's upgrade our app and let the user save the application state on the backend.

Sample Backend

I've prepared a simple backend application for this chapter.

This backend will allow us to store and retrieve the application state. We'll use a naive approach and will send the whole state every time it changes.

You will need to keep it running for this chapter's examples to work.

To launch it go to `code/01-first-app/trello-backend`, install dependencies using `yarn` and run `yarn start`:

```
yarn && yarn start
```

You should see this message:

```
Kanban backend running on http://localhost:4000!
```

You can verify that the backend works correctly by manually sending cURL requests. There are two endpoints available, one for storing data and one for retrieving.

Here is the command to store the data:

```
curl --header "Content-Type: application/json" \
  --request POST \
  --data '{"lists":[]}' \
  http://localhost:4000/save
```

And here is the one to retrieve:

```
curl http://localhost:4000/load
```

Every time you `POST` a JSON object to the `/save` endpoint, the backend stores it in memory. Next time you call the `/load` endpoint, the backend sends the saved value back.

The Final Result

Before we start working on our application, let's see what are we aiming to get in the end.

Launch the sample backend in a separate terminal tab:

```
cd code/01-first-app/trello-backend
yarn && yarn start
```

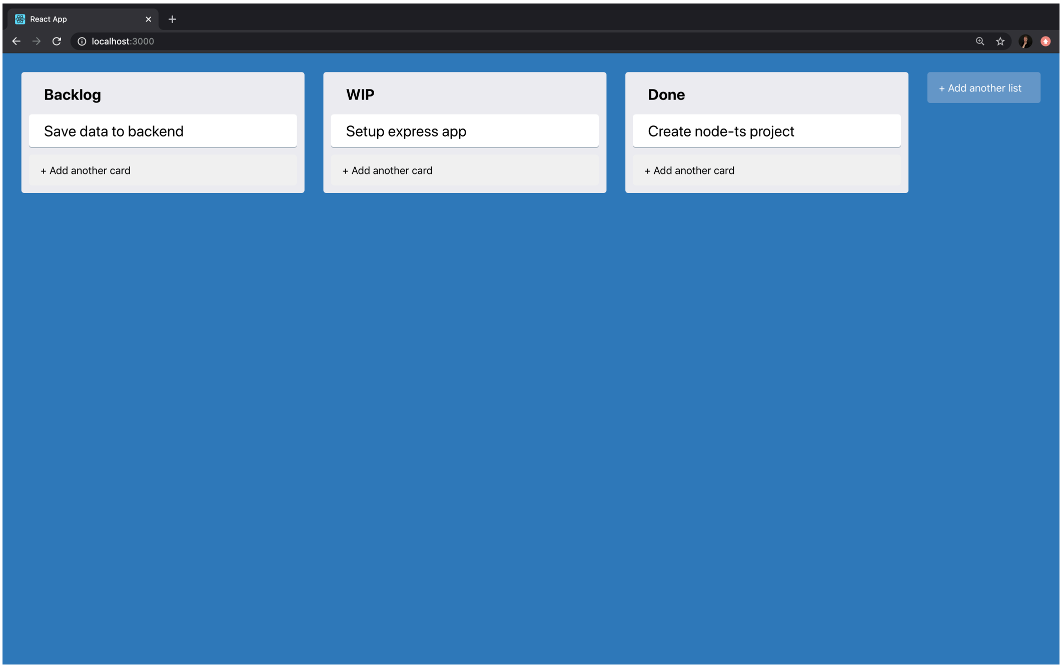
The completed example for this chapter is located in `code/01-first-app/01.28-saving-state-on` `cd` to this folder and launch the app:

```
cd code/01-first-app/01.28-saving-state-on-backend
yarn && yarn start
```

Initially, you should see an empty field with the "+ Add another list" button.

Empty field

Create a few lists and tasks and then reload the page. You should see that all the items are preserved.



Items preserved after page reload

The Starting Point

If you've completed the instructions from the first two chapters, then you can continue from where you left off.

If you didn't follow the previous chapters then you can use `code/01-first-app/01.28-saving-st` as your starting point. Copy the folder somewhere into your working projects directory.

Using Fetch With TypeScript

Browser JavaScript has a built-in `fetch` method that allows network requests to be made. Here is a TypeScript type declaration for this function:

```
function fetch(
  input: RequestInfo,
  init?: RequestInit
): Promise<Response>;
```

It says here that `fetch` accepts two arguments:

- `input` of type `RequestInfo`. `RequestInfo` is a `union` type defined like `string | Request`. It means it can be a `string` or an object having `Request` type.
- `init` - optional argument of type `RequestInit`. This argument contains options that can control a bunch of different settings. Using this parameter you can specify request method, custom headers, request body, etc.

Performing requests. Here is a typical `POST` request performed with `fetch`:

```
fetch('https://example.com/profile', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({username: 'example'}),
})
```

Working with responses. `fetch` returns a promise that resolves to `Response` type. We will usually work with `JSON` type responses, so to us the most interesting field is `.json()` method. This method returns a promise that resolves to response body text as `JSON`. Unfortunately, this method is not defined as generic so we will have to do some trickery to specify the type for the returned value.

Let's say I make a request to `https://api.github.com`. I know that this API returns an object with available endpoints, and amongst other fields there will be `current_user_url`:

```
const { current_user_url } = await fetch('https://api.github.com')
  .then((response) => {
    return response.json<{ current_user_url: string }>();
  })
}
console.log(typeof current_user_url) // string
```

> You can run this code in the TypeScript Playground[54].

Here I specified the return value of `json()` function call to be of type `{ current_user_url: string }`.

Create API Module

When I work with network requests I prefer to create a separate module with asynchronous functions that abstract the actual network calls.

Let's say we want to get some data from Github API:

```
export const githubAPI = <T>() => {
  return fetch('https://api.github.com').then((response) => {
    if (response.ok) {
      return response.json() as Promise<T>;
    } else {
      throw new Error("Something went wrong.");
    }
  })
}
```

Here I defined a *generic* function `githubAPI` that accepts a type argument `T`. I use it then to specify the type of the return value of `response.json()` function. I had to do this because by default the `response.json()` would have the type `any`. I'm also

[54] https://www.typescriptlang.org/play/?ssl=8&ssc=13&pln=1&pc=1#code/MYewdgzgLgBAZgUysAFgEQIZQzAvDDCATzGBgAoBK rwgJ+4-gBsYAXzwEA7hgCWsRMhTkA5CihQADhABcAegsZj6gHQBzTSl4AjO6AC2eyi3b+ YOygUYXJyIQhjcElqXDomVgCkoSgBMBgIqMgEOwArCHAqAggYAAV+EE91SQAeHmABIRFxSWkBWTMYaH51MAdFGgBuPyTFX0

checking the response status and throw an error if there was a problem with my request.

It allows me to use this function like this:

```
try {
  const { user_search_url } = await githubAPI<{
    user_search_url: string
  }>();
} catch (error) {
  // handle error
}
```

Now in my components, I won't have to think in terms of requests and responses. I will have an asynchronous function that returns data or throws an error.

This approach has a bunch of benefits:

- **We are not bound to a specific `fetch` implementation.** If you want to switch to axios[55], you will have only one place in your application where you'll have to make the changes.
- **Testing is easier.** I don't have to mock the request and response object. What I have to do is to mock an asynchronous function that returns some data.
- **Easy to add types.** If you have an API module where you wrap all your network requests into asynchronous functions, you can provide nice types for them.

To use our API we'll need to define our backend url somewhere. Create a `.env` file in the root of you project with the following contents:

```
REACT_APP_BACKEND_ENDPOINT=http://localhost:4000
```

You might want to restart your react dev server at this point so that it would read the values from the `.env` file.

Now create a new file `src/api.ts`, there we'll need to import the `AppState` type:

[55] https://github.com/axios/axios

```
import { AppState } from "./state/appStateReducer"
```

Then define the `save` function:

```
export const save = (payload: AppState) => {
  return fetch(`${process.env.REACT_APP_BACKEND_ENDPOINT}/save`, {
    method: "POST",
    headers: {
      Accept: "application/json",
      "Content-Type": "application/json"
    },
    body: JSON.stringify(payload)
  }).then((response) => {
    if (response.ok) {
      return response.json()
    } else {
      throw new Error("Error while saving the state.")
    }
  })
}
```

This function will accept the current state and send it to the backend as JSON. In case of an unsuccessful save we'll throw an error.

Define the `load` function:

```
export const load = () => {
  return fetch(`${process.env.REACT_APP_BACKEND_ENDPOINT}/load`).then(
    (response) => {
      if (response.ok) {
        return response.json() as Promise<AppState>
      } else {
        throw new Error("Error while loading the state.")
      }
    }
  )
}
```

This function will load the previously saved data from the backend. We cast the JSON parsing result to the `AppState` type. Just like in the `save` function we'll throw an error if the backend will return a non-ok status.

Ok, now you have an API with two functions:

- `save` function that makes a `POST` request and sends a JSON representation of our application state to the backend.
- `load` function that makes a `GET` request to retrieve the previously saved state.

Saving The State

We want to save our application state every time it changes. This means that every time we move the items around or create new ones, we want to make a request to our backend.

In our application, we have a redux-like architecture. It means that we have a centralized store that holds our application state.

We don't use Redux, but we use React's built-in hook `useReducer` which is fairly similar.

In order to save the state on the backend we'll use a `useEffect` hook.

Go to `src/state/AppStateContext.tsx` and import the `useEffect` hook from React and the `save` function from the `api` module:

```
import {
  createContext,
  useContext,
  useEffect,
  Dispatch,
  FC
} from "react"
import { save } from "../api"
```

Add the following code right before the `AppStateProvider` return statement:

```
useEffect(() => {
  save(state)
}, [state])
```

The useEffect[56] hook allows us to run side effect callbacks on some value change.

It accepts a callback function and a dependency array. Then it triggers the callback function every time the variables in the dependency array get updated.

So in our case, we call our `save` method with the value of the `state` every time the `state` is updated.

Let's verify that everything works correctly. Every time you send the data to the backend it logs it to the console.

Try to drag the items around and then check the backend console output. It should look like this:

```
$ yarn start
yarn run v1.19.1
$ tsc && node dist/index.js
Kanban backend running on http://localhost:4000!
{
  lists: [
    { id: '0', text: 'To Do', tasks: [Array] },
    { id: '1', text: 'In Progress', tasks: [Array] },
    { id: '2', text: 'Done', tasks: [Array] }
  ]
}
{
  draggedItem: {
    type: 'CARD',
    id: 'c0',
    index: 0,
    text: 'Generate app scaffold',
    columnId: '0'
  },
  lists: [
    { id: '0', text: 'To Do', tasks: [Array] },
    { id: '1', text: 'In Progress', tasks: [Array] },
    { id: '2', text: 'Done', tasks: [Array] }
  ]
}
{
  draggedItem: {
    type: 'CARD',
    id: 'c0',
    index: 0,
    text: 'Generate app scaffold',
    columnId: '1'
  },
  lists: [
    { id: '0', text: 'To Do', tasks: [] },
    { id: '1', text: 'In Progress', tasks: [Array] },
    { id: '2', text: 'Done', tasks: [Array] }
  ]
```

Backend console output

[56] https://reactjs.org/docs/hooks-effect.html

Loading The Data

In our application, the only time we want to load the data is when we first render it.

We have a provider component that is mounted once when we render our application. The problem is that we can't load the data directly inside it because then our application will first initialize with the default data. We would then get the data from the backend but our reducer would already be initialized.

The solution is to have a wrapper component that will load the data for us and then pass the data to our context provider as a prop so it initializes with correct data.

We could create another component that will render our `AppStateProvider` inside it. But I propose to create a more generic solution using the HOC pattern.

What is HOC?

HOC (Higher Order Component) is a React pattern in which you create a factory function that accepts a wrapped component as an argument, wraps it into another component that implements the desired behavior and then returns this construction.

We will talk about HOCs and other React patterns in the next chapters. For now, let's practice creating one.

Creating your first HOC

Our HOC will accept `AppStateProvider` and inject the `initialState` prop containing loaded data into it. This kind of HOCs is called an *injector HOC*

Create a new file `src/withInitialState.tsx` and make necessary imports:

```
import { useState, useEffect } from "react"
import { AppState } from "./state/appStateReducer"
import { load } from "./api"
```

Then define and export our `withInitialState` HOC:

```
type InjectedProps = {
  initialState: AppState
}

type PropsWithoutInjected<TBaseProps> = Omit<
  TBaseProps,
  keyof InjectedProps
>

export function withInitialState<TProps>(
  WrappedComponent: React.ComponentType<
    PropsWithoutInjected<TProps> & InjectedProps
  >
) {
  return (props: PropsWithoutInjected<TProps>) => {
    const [initialState, setInitialState] = useState<AppState>({
      lists: [],
      draggedItem: null
    })
  // ...
    return <WrappedComponent {...props} initialState={initialState} />
  }
}
```

Let's go line-by-line. First we define a type that represents the props that we are injecting. In this case it is the `initialState: AppState` prop:

```
type InjectedProps = {
  initialState: AppState
}
```

Then we define a helper type `PropsWithoutInjected`:

```
type PropsWithoutInjected<TBaseProps> = Omit<
  TBaseProps,
  keyof InjectedProps
>
```

This is a generic type that accepts the `TBaseProps` type variable that will represent the original props type of the wrapped component. We use `Omit` to remove the fields of the `InjectedProps` type from it.

The utility type `Omit` constructs a new type removing the keys that you provide to it:

```
type Book = {
  title: string;
  length: number;
  author: string;
  description: string;
}

type BookWithoutDescription = Omit<Book, "description">;
// type BookWithoutDescription = {
//   title: string
//   length: number
//   author: string
// }
```

For a complete list of utility types refer to TypeScript handbook[57].

The query `keyOf` returns a union type that contains the keys of the type that you pass to it, for example:

[57] https://www.typescriptlang.org/docs/handbook/utility-types.html

```
type Book = {
  title: string;
  length: number;
  author: string;
}
type BookKeys = keyof Book; // "title" | "length" | "author"
```

> Read more about the `keyof` indexed type query in the TypeScript Documentation[58].

Then, we define a `withInitialState` generic function that accepts a `WrappedComponent` argument and a `TProps` type variable.

```
export function withInitialState<TProps>(
  WrappedComponent: React.ComponentType<
    PropsWithoutInjected<TProps> & InjectedProps
  >
) {
  // ...
}
```

The `WrappedComponent` argument has a complex props type declaration, we define its props as an intersection type between the `PropsWithoutInjected<TProps>` and the `InjectedProps`:

```
WrappedComponent: React.ComponentType<
  PropsWithoutInjected<TProps> & InjectedProps
>
```

We end up with a type that is very similar to the `TProps`. We removed the injected props, and then added them back. This might look tautological, but it is necessary to let TypeScript know that the wrapped component will accept the `InjectedProps`. TypeScript is very cautious with generic types and if we wouldn't perform this trick it wouldn't let us pass the fields defined in the `InjectedProps` type to our component.

[58] https://www.typescriptlang.org/docs/handbook/release-notes/typescript-2-1.html#keyof-and-lookup-types

Let's continue. Inside of the `withInitialState` function, we return the wrapper component:

```
return (props: PropsWithoutInjected<TProps>) => {
// ...
}
```

Here we remove the props that our HOC injects from the wrapper component props.

Inside of this wrapper component we render the `WrappedComponent`(in our app it will be `AppStateProvider`) passing the `initialState` and the rest of the props to it.

```
return <WrappedComponent {...props} initialState={initialState} />
```

So as you can see we have a function that creates a wrapper component for some component that we pass to this function as an argument.

> If you don't understand how HOCs work yet, don't worry, we have a dedicated chapter about advanced React patterns, where we talk in more detail about them.

Load The Data Inside The HOC

Inside our wrapper component add two more states and a `useEffect` hook:

```
    const [isLoading, setIsLoading] = useState(true)
    const [error, setError] = useState<Error | undefined>()

    useEffect(() => {
      const fetchInitialState = async () => {
        try {
          const data = await load()
          setInitialState(data)
        } catch (e) {
          if (e instanceof Error) {
```

```
          setError(e)
        }
      }
      setIsLoading(false)
    }
    fetchInitialState()
  }, [])
```

Our `useEffect` call will be triggered once we mount our component and then we might have one of the three different states:

- **Pending**. We have this state when we've started loading data but not finished yet. `isLoading` is `true`. We render some kind of loader.
- **Success**. The data is loaded successfully and is stored inside the `initialState`, `isLoading` is `false`, `error` is `null`. We can render our app.
- **Failure**. We got an error and stored it in the `error` state, `isLoading` is `false`. We render the error message.

Inside our `useEffect` callback, we defined the `fetchInitialState` asynchronous function. We did it so that we could use the `async/await` syntax.

Inside the `fetchInitialState` function we have a `try/catch` block. When we load the data we store it in the state and if something goes wrong we save the error.

> In Javascript you can throw any value as an error. You can throw a string, a number or even an object. This means that in TypeScript the caught error will have type `any` or the `unknown` by default. If you want to catch a specific error type - then you'll have to use the `instanceof` check like we did in our example.

Update the wrapper component layout.

```
    if (isLoading) {
      return <div>Loading</div>
    }

    if (error) {
      return <div>{error.message}</div>
    }

    return <WrappedComponent {...props} initialState={initialState} />
  }
}
```

If the `isLoading` state is `true` we show the loader. If there is an error - we show an error message. And if data was loaded successfully we return the wrapped component.

Use The HOC

Now the HOC is ready, import it into the `src/state/AppStateContext.tsx`:

```
import { withInitialState } from "../withInitialState"
```

Define the `AppStateProviderProps`:

```
type AppStateProviderProps = {
  children: React.ReactNode
  initialState: AppState
}
```

Here we define the `children` prop as a required field to make it clear that the `AppStateProvider` is supposed to wrap other components.

Wrap the `AppStateProvider` into `withInitialState` HOC:

```
export const AppStateProvider =
  withInitialState<AppStateProviderProps>(
    ({ children, initialState }) => {
      const [state, dispatch] = useImmerReducer(
        appStateReducer,
        initialState
      )

      useEffect(() => {
        save(state)
      }, [state])

      const { draggedItem, lists } = state
      const getTasksByListId = (id: string) => {
        return lists.find((list) => list.id === id)?.tasks || []
      }

      return (
        <AppStateContext.Provider
          value={{ draggedItem, lists, getTasksByListId, dispatch }}
        >
          {children}
        </AppStateContext.Provider>
      )
    }
  )
```

We don’t need the `FC` type anymore, so remove the import:

```
import {
  createContext,
  useContext,
  useEffect,
  Dispatch
} from "react"
```

You can also remove the `appData` const, we don't need it anymore.

Launch The App

Now the app should preserve the state on our backend.

Launch the app and try to move the columns and cards around. Reload the page to verify that the state was preserved.

> You can find the working example for this part in the `code/01-first-app/01.29-loading-t`

How to Test Your Applications: Testing a Digital Goods Store

Introduction

In this part, we will learn how to test our React + TypeScript applications. Unlike other sections where we start from scratch and then build an application, in this one we'll begin with an existing app and will cover it with tests.

We will use the React testing library[59] because it has a simple API, is easy to set up, and is recommended by the React team. Oh, and of course it supports TypeScript.

It isn't always obvious how to test a front-end application, but the React testing library makes it easy.

Below, we're going to walk through how to test components in React with *Jest*, how to mock dependencies, test routing, and even test React hooks.

Get familiar with the application

Before we begin, let's get familiar with the example application that we'll be covering with tests.

This book has an attached `zip` archive with examples for each step. The completed example is in `code/02-testing/completed`.

Unzip the archive and `cd` to the app folder.

```
cd code/02-testing/completed
```

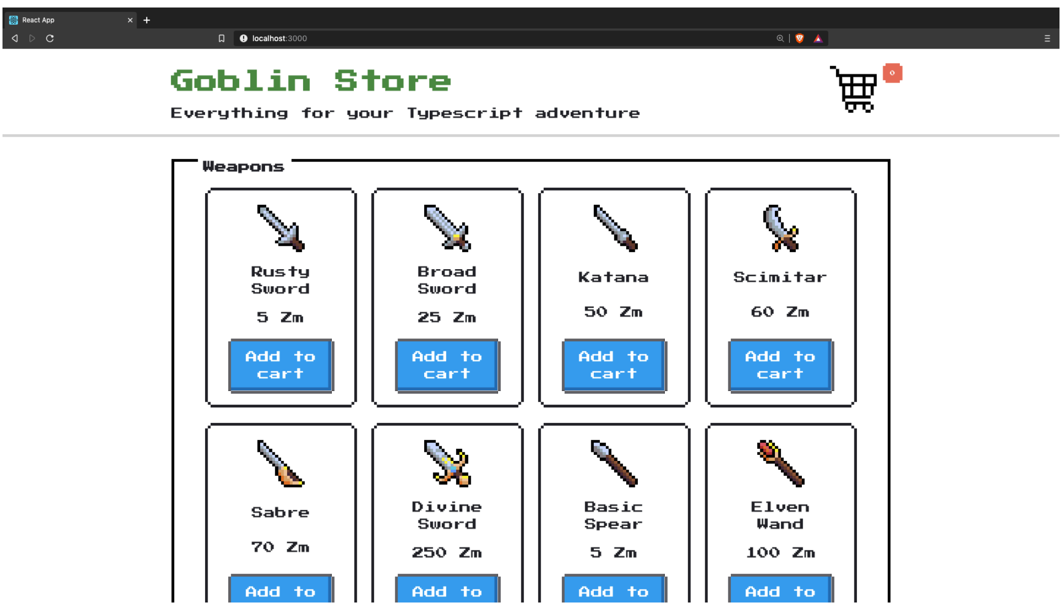
When you are there, install the dependencies and launch the app:

[59] https://testing-library.com/docs/react-testing-library/intro

```
yarn --ignore-engines && yarn dev
```

> The `yarn dev` command runs both a server and a client. We use concurrently[60] to launch two scripts at the same time. You can check `src/package.json` to see how we do it.
>
> In this app we'll have to use the `--ignore-engines` flag when we install dependenciens because of the `nes.css` package that specifies an old version of NodeJS as a dependency.

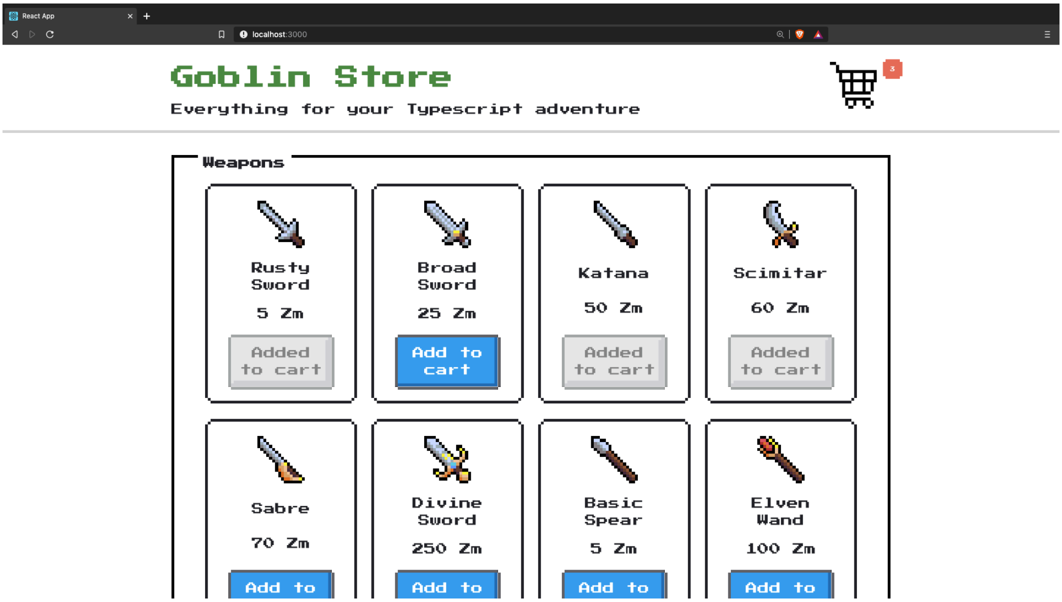
It should also open the app in the browser. If that doesn't happen, navigate to `http://localhost:3000` and open it manually.



Main screen

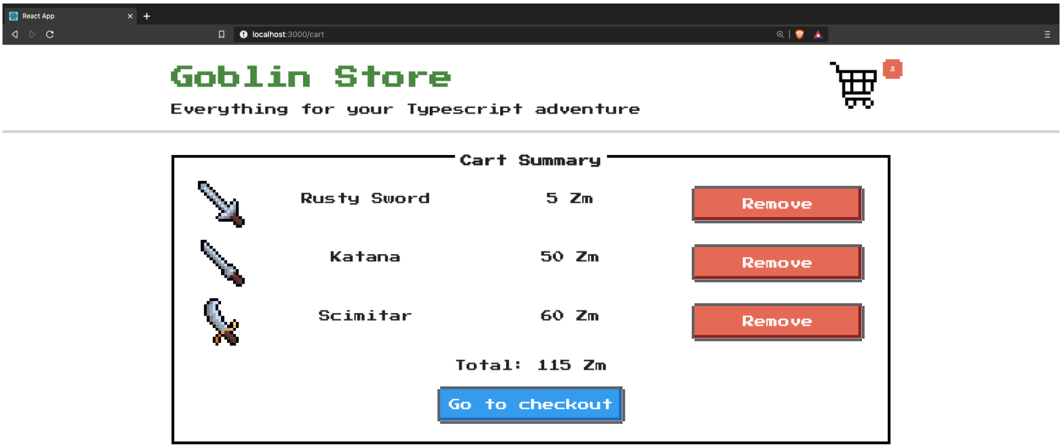
You should see a list of hero equipment: weapons, armor, potions. Click the **Add to cart** buttons to add items to the cart.

[60] https://www.npmjs.com/package/concurrently



Selected items

You should also see that the cart widget in the top right corner shows the number of items you are going to buy. Click that widget.



Cart summary

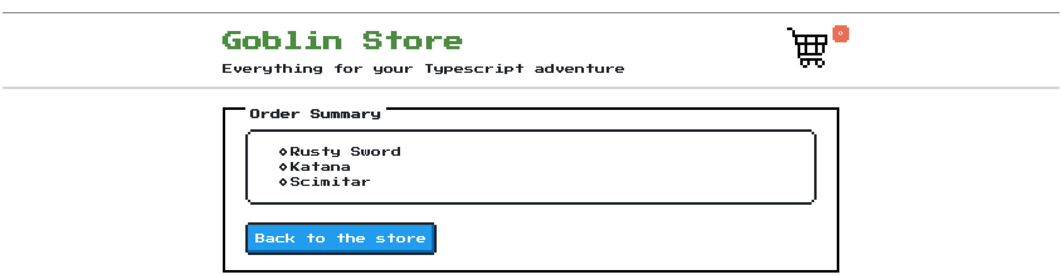
You will end up on the *Cart Summary* page. Here you can review the cart and remove any items if you don't want to buy them anymore. Click the **Go to checkout** button.

Selected items

Now you are on the *Checkout* page. Here you can see a list of products you are going to buy with the total amount of Zorkmids you have to pay.

Below the list, you will see the checkout form. Fill in the fields. If you try to skip the fields or input incorrect values, you'll see error messages. Also, note that we are normalizing the **Card number** field to have the `xxxx xxxx xxxx xxxx` format.

After you are done filling in the form, press the **Checkout** button.



Selected items

Now the cart will be purged, and you will be redirected to the *Order Summary* page.

On this page, you should see the list of products you've bought and the **Back to the store** button. Click the button to get back to the main page.

That's it - here we have a tiny fantasy store where you can put products into the cart, review the cart, maybe remove some products from it, and then fill in the checkout form and perform the purchase.

We will go through the code of each page, discuss its functionality, and then cover it with tests.

Initial Setup

To begin working on this project copy the `code/02-testing/02.02-initial-setup` to your workspace folder. It will be our starting point.

> Just like in the previous lesson it is important to use the Workspace version of TypeScript. You can see the instructions on how to specify it here[61]

In this tutorial, I assume that you will be using VSCode. Open the project in the editor.

```
.
├── .vscode
│   └── launch.json // Settings for debugging in VSCode
├── node_modules
├── public
├── src
├── .gitignore
├── .nvmrc // This file contains Node version
├── package.json
├── README.md
├── tsconfig.json
├── yarn-error.log
└── yarn.lock
```

[61]https://stackoverflow.com/questions/50432556/cannot-use-jsx-unless-the-jsx-flag-is-provided

You should see the following file structure.

Our application is written using Create React App, so Jest is already pre-configured there.

> In the first chapter of this book I go through the whole application structure generated by CRA and explain the purpose of each file.

Jest supports TypeScript out of the box. We don't need any additional setup to run the tests.

To verify that everything works, install the dependencies using `yarn` and run the tests:

```
yarn --ignore-engines && yarn test
```

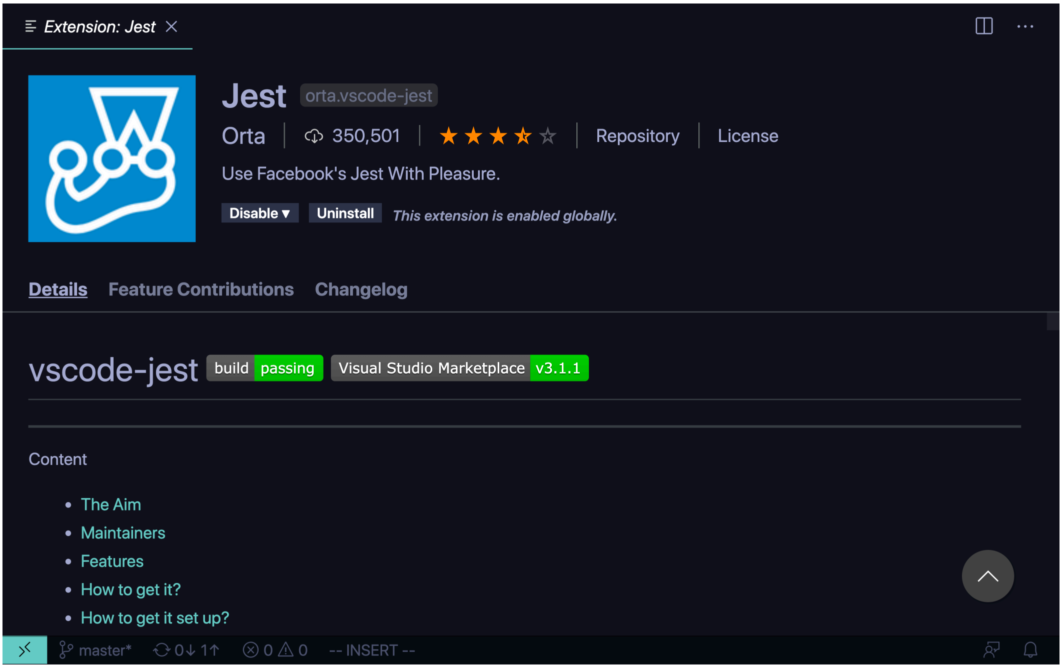
This will launch the Jest runner in watch mode. If you change the code or test files, it will re-run the tests. You can quit the runner by pressing `q`.

> We use the `--ignore-engines` option here because the `nes.css` package that is only compatible with node `10.x` and you likely have a newer version.

Install VSCode plugin

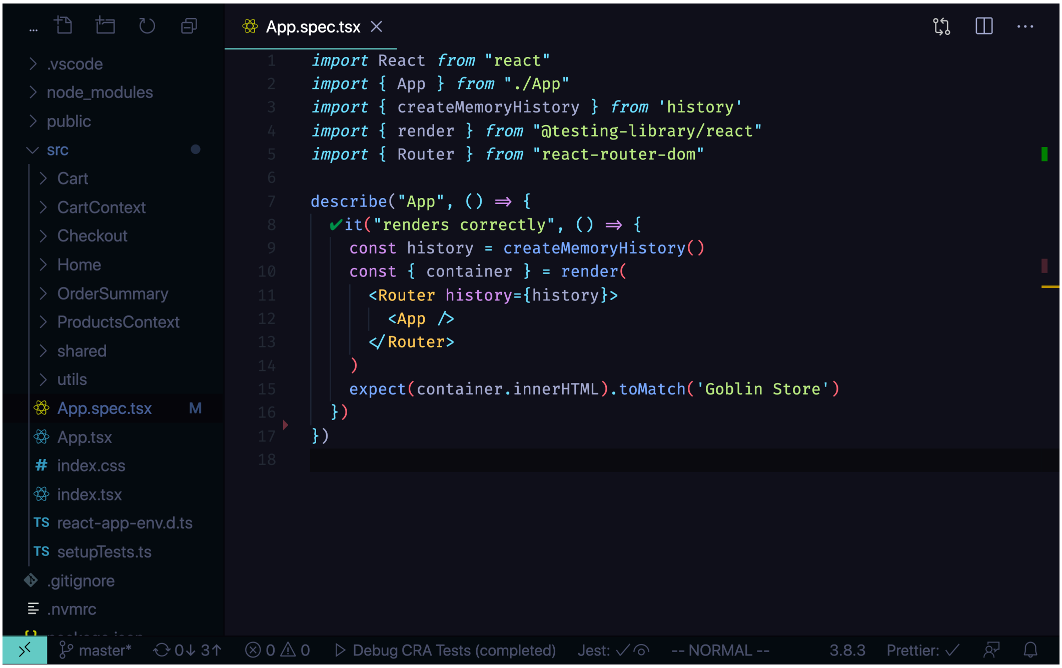
If you are using VSCode, you can install a useful Jest plugin[62] that automatically runs the tests and displays the test results right in the text editor.

[62] https://marketplace.visualstudio.com/items?itemName=Orta.vscode-jest



Jest VSCode plugin

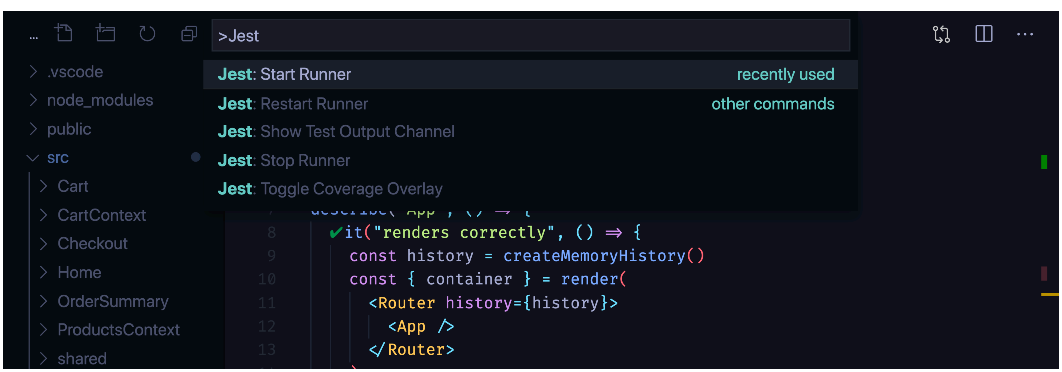
To verify that it works, open `src/App.spec.tsx`. You should see the green checkmark near the first test case:



Jest VSCode plugin

This way you can get visual feedback from running your tests way quicker.

If it doesn't show up automatically, launch the `Command Palette` and select `Jest: Start Runner`.



Jest VSCode plugin

Troubleshooting

> If your VSCode Jest plugin doesn't seem to work, check the "Output" console at the bottom of your window. It should contain some messages that will help you diagnose the issue.
>
> `vscode-jest` also contains a troubleshooting section in their documentation.[63]

Enable Debugging Tests

Before we begin there is one more thing that is good to know. How can you debug your tests? To enable debugging in VSCode you need to add a `launch.json` configuration into the `.vscode` folder in the root of your project.

In this project, I already did this for you. You can open `.vscode/launch.json` to see what it contains:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Debug CRA Tests",
      "type": "node",
      "request": "launch",
      "runtimeExecutable": "${workspaceRoot}/node_modules/.bin/react-sc\
ripts",
      "args": [
        "test",
        "--runInBand",
        "--no-cache",
        "--watchAll=false"
      ],
      "cwd": "${workspaceRoot}",
      "protocol": "inspector",
      "console": "integratedTerminal",
      "internalConsoleOptions": "neverOpen",
```

[63] https://github.com/jest-community/vscode-jest/blob/master/README.md#troubleshooting

```
      "env": { "CI": "true" },
      "disableOptimisticBPs": true
    }
  ]
}
```

Here we specify a launch configuration called `Debug CRA Tests`. It uses React scripts with parameters from the `args` field. It's the equivalent of running the following in your terminal:

```
yarn test --runInBand --no-cache --watchAll=false
```

- `--runInBand` makes the tests run serially in one process. We use it because it's hard to debug many processes that are running at the same time.
- `--no-cache` disables the cache, to avoid cache-related problems during debugging.
- `--watchAll=false` disables re-running the tests when any related files change. We want to perform a single run, so we set this flag to `false`.

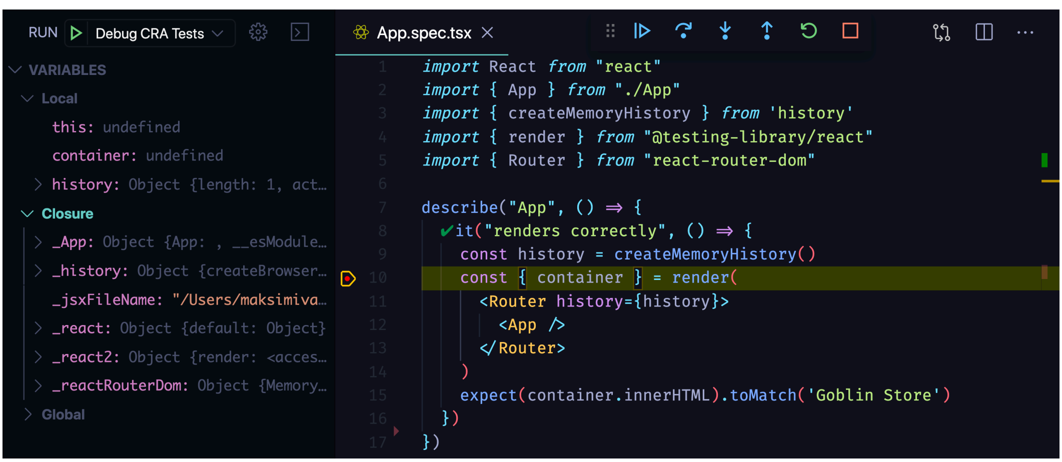
This configuration will work with any Create React App generated application.

Set a Breakpoint

Let's verify our debugging configuration. Open `src/App.spec.tsx` and place a breakpoint:

Jest VSCode plugin

Now open the `Command Palette` (`View -> Command Palette`) and select `Debug: Select and Start Debugging` and the `Debug CRA Tests`.



Jest VSCode plugin

You should see the debug pane with the runtime variables, call stack, and breakpoints sections on the left and the control buttons at the top of the screen.

You can use this interface to go through your test's execution step-by-step and observe the values of all the variables in your code. We will use this functionality later in this chapter, but for now, stop the execution by pressing the red square button (or press `Shift + F5`).

Remove the breakpoint by clicking on it.

Writing Tests

Our application entry point is `src/index.tsx`. This is where we render our component tree into the HTML.

```
import { StrictMode } from "react"
import ReactDOM from "react-dom"
import { BrowserRouter } from "react-router-dom"
import { App } from "./App"
import { CartProvider } from "./CartContext"
import "./index.css"

ReactDOM.render(
  <StrictMode>
    <CartProvider>
      <BrowserRouter>
        <App />
      </BrowserRouter>
    </CartProvider>
  </StrictMode>,
  document.getElementById("root")
)
```

Here we render our `App` component. Note that it is wrapped into two providers here:

- `<CartProvider>` manages the cart state. It persists the information in `localStorage`.
- `<BrowserRouter>` this provider allows using routing across our app.

Note that some of the components we are going to test will depend on those providers. We will have to acknowledge this when writing tests.

This file only contains the application initialization code and doesn't have any logic we can test. We will skip it and go to the `App` component.

Testing the App component

Open `src/App.tsx`. This file contains `App` component definition.

```
export const App = () => {
  return (
    <>
      <Header/>
      <div className="container">
        <Switch>
          <Route exact path="/">
            <Home />
          </Route>
          <Route path="/checkout">
            <Checkout />
          </Route>
          <Route path="/cart">
            <Cart />
          </Route>
          <Route path="/order">
            <OrderSummary />
          </Route>
          <Route>Page not found</Route>
        </Switch>
      </div>
    </>
  )
}
```

`App` is a functional component. It doesn't accept any props, nor does it contain any business logic. The only thing it does is render the layout.

Most of your components will output some layout and this is the first thing you can test.

Let's write a test that verifies that `App` component at least renders successfully. Create `src/App.spec.tsx` and add the following code:

```
import { App } from "./App"
import { createMemoryHistory } from "history"
import { render } from "@testing-library/react"
import { Router } from "react-router-dom"

describe("App", () => {
  it("renders successfully", () => {
    const history = createMemoryHistory()
    const { container } = render(
      <Router history={history}>
        <App />
      </Router>
    )
    expect(container.innerHTML).toMatch("Goblin Store")
  })})
```

Here we wrap the whole testing code into a `describe('App')` block. This way we specify that all the `it` blocks containing specific test cases are related to testing the `App` component. You can greatly improve the readability of your tests by using `describe` blocks wisely. We will talk about it more in this chapter.

Inside the `describe` we have an `it` block. `it` blocks contain individual tests. Optimally each `it` block should test one aspect of the tested entity. Here we test that our `App` component `renders successfully`.

Every `it` block has a name - in our case it's `renders successfully` - and a callback.

A good practice is to use the present simple tense for names and keep them short and unambiguous. Treat the `it` word as a part of the sentence:

- **Bad:** `it("component was rendered successfully")`
- **Good:** `it("renders successfully")`

The callback contains the actual testing code.

```
describe("App", () => {
  it("renders successfully", () => {
```

Now if you run the test it will fail with the following error:

```
Invariant failed: You should not use <Switch> outside a <Router>
```

Where is this coming from?

Our `App` component uses `Switch` - which comes from React Router - to render different pages depending on the URL we are on. But the `Switch` component has a constraint: it can only be used inside a `BrowserRouter` context (`BrowserRouter` also comes from React Router).

Look the `src/index.tsx` again. When you open `src/index.tsx`, you'll see that, when we run our application outside of our tests, we wrap our `App` component there into the `BrowserRouter`:

```
import { StrictMode } from "react"
import ReactDOM from "react-dom"
import { BrowserRouter } from "react-router-dom"
import { App } from "./App"
import { CartProvider } from "./CartContext"
import "./index.css"

ReactDOM.render(
  <StrictMode>
    <CartProvider>
      <BrowserRouter>
        <App />
      </BrowserRouter>
    </CartProvider>
  </StrictMode>,
  document.getElementById("root")
)
```

However, in our *test* we were trying to run the `App` component directly – *without* the `BrowserRouter` context.

To fix this, we wrap our `App` component into a `BrowserRouter` in our tests as well.

Tests Run in Node

It is important to note that our tests run in the *Node* environment - not an actual browser! - and we use a simulated DOM API provided by jsdom[64]. It means that some functionality can be missing or work differently compared to the browser environment.

One of the missing things is the History API[65], so to use routing we'll have to install an additional package that will provide us the History API functionality.

> Alternatively we could use the `MemoryRouter` provided by `react-router-dom`, but in our case it will be more convenient to have direct access to the `history` object. This way it will be easier to control the navigation inside our tests.

Install `history` as a dev dependency:

```
yarn add --ignore-engines --dev history@5.0.0
```

Now let's fix our test by using our synthetic History API:

[64] https://www.npmjs.com/package/jsdom
[65] https://developer.mozilla.org/en-US/docs/Web/API/History_API

```
import { App } from "./App"
import { createMemoryHistory } from "history"
import { render } from "@testing-library/react"
import { Router } from "react-router-dom"

describe("App", () => {
  it("renders successfully", () => {
    const history = createMemoryHistory()
    const { container } = render(
      <Router history={history}>
        <App />
      </Router>
    )
    expect(container.innerHTML).toMatch("Goblin Store")
  })})
```

There are three things going on here:

Initial setup. We create the `history` object and pass it to the `Router` component.

Rendering. We call the `render` method from @testing-library/react[66] and get the `container` instance. The container represents the containing DOM node of the rendered React component.

Expectation. We call the `expect` method provided by Jest[67]. We pass the HTML contents of our container to it and check if it contains the string `"Goblin Store"` in it. Our `App` layout always renders the `Header` component that contains this text, so it can be a good indication that our component rendered successfully.

Mocking Components

Our `App` component also defines the routing system and renders the `Home` page at the root route.

We can test it as well, but our `Home` page component depends on data from the `ProductsProvider` to render the products list. It might also render other components

[66] https://testing-library.com/docs/react-testing-library
[67] https://jestjs.io/docs/en/expect

with more dependencies, so in the end, the test can become quite cumbersome to set up.

A common approach in such situations is to mock the dependency, so we can test our component in isolation.

Let's write the test that will verify that `App` will render the `Home` component at the root route. We will mock the `App` component so that we won't have to work with extra dependencies.

In `src/App.spec.tsx` call `jest.mock` to mock the module containing the `Home` component:

```
jest.mock("./Home", () => ({ Home: () => <div>Home</div> }));
```

Add this line right after the imports.

`jest.mock` allows you to mock whole modules. Mocking means that we substitute the real object with a fake double that mimics its behavior. You can also spy on mocked objects and functions to track how your code is using them. But we'll get back to this later.

Here we defined our mock component that will be used instead of the real `Home` component. It will render `"Home component"` text, that we can refer to in our test to verify that the component was rendered.

Now add a new `it` block right after the first one:

```
it("renders Home component on root route", () => {
	const history = createMemoryHistory()
	history.push("/")
	const { container } = render(
		<Router history={history}>
			<App />
		</Router>
	)
	expect(container.innerHTML).toMatch("Home")
})
```

Here we `push` the root url to our `history` object before rendering the `App` component. Then we check that the content of the `container` matches with the `"Home"` string that we render in our mocked `Home` component.

If you are using the Jest VSCode plugin you should see the green checkbox near this test. If you decided not to use the plugin, run the tests in the terminal from the project root:

```
yarn test
```

The tests should pass.

It is always a good idea to check if your tests could fail. If your test is always passing - it is likely not testing anything.

Try to push some other url instead of the root one, for example `/test`, make sure that the test is failing and then revert this change.

Jest helper to test navigation

If you open `src/App.tsx` file, you'll see that our `App` component renders four different routes using `Switch`.

```
<Switch>
  <Route exact path="/">
    <Home />
  </Route>
  <Route path="/checkout">
    <Checkout />
  </Route>
  <Route path="/cart">
    <Cart />
  </Route>
  <Route path="/order">
    <OrderSummary />
  </Route>
```

```
  <Route>Page not found</Route>
</Switch>
```

Aside from the root route where it renders the `Home` component it also renders `/checkout`, `/cart`, and `/order` routes.

We can test those routes as well. But we will end up with a lot of duplicated code. All those route's tests will look like the root route test. The only things that will be different will be the `url` and the expected strings to render.

Let's create a helper method to render components with the router.

Global Helper With TypeScript

First of all create a new file `./testHelpers.tsx` that will hold our helper function.

Add the imports:

```
import { ReactNode } from "react"
import { createMemoryHistory, MemoryHistory} from "history"
import {
  render,
  RenderResult
} from "@testing-library/react"
import { Router } from "react-router-dom"
```

Then define the `renderWithRouter` function:

```
global.renderWithRouter = (renderComponent, route) => {
  const history = createMemoryHistory()
  if (route) {
    history.push(route)
  }
  return {
    ...render(
      <Router history={history}>{renderComponent()}</Router>
    ),
    history
  }
}
```

This function creates a `history` object and pushes the `route` to it if we got it through the arguments. Then we call the `render` method from the `testing-library/react` and return all the fields that we got from it plus the `history` object.

We've defined the `renderWithRouter` function on the `global` object. The `global` object is a global namespace object in node[68].

Everything that we define on this object we'll be able to address directly in our tests. For example, we'll be able to call the `renderWithRouter` function without importing it.

Now TypeScript will complain that the `Property 'renderWithRouter' does not exist on type 'Global'` and also the type of the arguments is `any`. Let's fix that.

First, define the type for our function:

```
type RenderWithRouter = (
  renderComponent: () => ReactNode,
  route?: string
) => RenderResult & { history: MemoryHistory }
```

Here we defined a function that accepts `renderComponent` and optionally a `route`. As a result, it should return a `RenderResult` from `@testing-library/react`, which is a return type of its `render` function with an additional field `history`.

[68]https://nodejs.org/api/globals.html#globals_global

By default, the `global` object has type `Global`. We can add a new field to it.

```
declare global {
  namespace NodeJS {
    interface Global {
      renderWithRouter: RenderWithRouter
    }
  }
```

The type `Global` is a part of `NodeJS` namespace which is globally available. It means that we can address `NodeJS` namespace from any module directly without the need to import it first.

We can augment global namespaces by using the `declare global {}` syntax. Read more about it in the TypeScript documentation[69].

Here we augment the `Global` type by adding a `renderWithRouter` field to it with type `RenderWithRouter`.

Great. Now we'll be able to call our function by referencing it on the `global` object like this:

```
global.renderWithRouter(() => <ExampleComponent />, "/")
```

If you call it without the `global` at the beginning, TypeScript will give you an error: `can't find name 'renderWithRouter'`.

To call it without referencing the `global` object we'll need to augment the globalThis[70] type as well. It is a variable that refers to the global scope.

[69]https://www.typescriptlang.org/docs/handbook/release-notes/typescript-1-8.html#augmenting-globalmodule-scope-from-modules
[70]https://www.typescriptlang.org/docs/handbook/release-notes/typescript-3-4.html#type-checking-for-globalthis

```
declare global {
  namespace NodeJS {
    interface Global {
      renderWithRouter: RenderWithRouter
    }
  }

  namespace globalThis {
    const renderWithRouter: RenderWithRouter
  }
}
```

Now you should be able to call `renderWithRouter` directly:

```
renderWithRouter(() => <ExampleComponent />, "/")
```

Let's make it available in our test files. Go to `src/setupTests.ts` and import the `src/testHelpers.tsx`:

```
import "./testHelpers"
```

Testing navigation

Let's write the routing tests.

Go to `src/App.spec.tsx` and mock the page's components. Add the following code right after you mock the `Home` component:

```
jest.mock("./OrderSummary", () => ({
  OrderSummary: () => <div>Order summary</div>,
}));
jest.mock("./Checkout", () => ({
  Checkout: () => <div>Checkout</div>,
}));
```

Create a new `describe` block with the name `routing` and move our root route test there. Make it use the `renderWithRouter` helper function:

```
describe("routing", () => {
  it("renders home page on '/'", () => {
    const { container } = renderWithRouter(() => <App />, "/");
    expect(container.innerHTML).toMatch("Home");
  });
});
```

Make sure that your tests pass and then add a new `it` block for the `/checkout` route:

```
it("renders checkout page on '/checkout'", () => {
        const { container } = renderWithRouter(
                () => <App />,
                "/checkout"
        )
        expect(container.innerHTML).toMatch("Checkout")
})
```

Repeat it for the `/cart` and `/order` routes.

After you are done with all the existing routes, it's time to check if the nonexistent routes also render correctly:

```
it("renders 'page not found' message on nonexistent route", () => {
        const { container } = renderWithRouter(
                () => <App />,
                "/this-route-does-not-exist"
        )
        expect(container.innerHTML).toMatch("Page not found")
})
```

Here we check that for an arbitrary route that is not defined, we'll render the `Page not found` message.

Shared Components

Before we move on and start testing our pages, let's test the shared components. All of them are defined inside the `src/shared` folder. They have less dependencies so it will be a good warm up.

Header

The `Header` component renders the title of the store and also the cart widget. The cart widget is defined in a separate component, so we'll mock it and test the `Header` in isolation.

We will test that the header renders correctly, and that if you click on the store logo it will redirect you to the main page.

Create a new file called `src/shared/Header.spec.tsx` with the following contents:

```
describe("Header", () => {
  it.todo("renders correctly");

  it.todo("navigates to / on header title click");
});
```

Here we've planned out the tests we are going to write using `it.todo` syntax. This syntax allows you to write only the test case name and omit the callback. It is useful when you want to list the aspects that you want to test, but you don't want to write the actual tests yet.

Add the imports:

```
import { Header } from "./Header";
import { fireEvent } from "@testing-library/react";
```

The `Header` component has a dependency to `CartWidget` component. It will be easier if we mock the `CartWidget` component. Add this code above the top level `describe` block:

```
jest.mock("./CartWidget", () => ({
  CartWidget: () => <div>Cart widget</div>,
}));
```

Next let's test that the `Header` component will render correctly:

```
describe("Header", () => {
  it("renders correctly", () => {
    const { container } = renderWithRouter(() => <Header />);
    expect(container.innerHTML).toMatch("Goblin Store");
    expect(container.innerHTML).toMatch("Cart widget");
  });
  // ...
});
```

The header contains a link to the main page. This link is using the `Link` component from the `react-router-dom` so we'll have to use `renderWithRouter` to be able to test it.

We've mocked the `CartWidget` component to render the `"Cart widget"` string. Now in our test, we can make sure that it was rendered by checking if the `"Cart widget"` string ends up in rendered layout.

Now let's verify that if we click the "Goblin Store" sign, we'll get redirected to the root url.

Import the `fireEvent` method from the `@testing-library/react`:

```
import { fireEvent } from "@testing-library/react";
```

Now let's implement the second test case:

```
it("navigates to / on header title click", () => {
  const { getByText, history } = renderWithRouter(() => <Header />);
  fireEvent.click(getByText("Goblin Store"));
  expect(history.location.pathname).toEqual("/");
});
```

Remember, how inside of our `renderWithRouter` helper function we we returned the `history` object along with the rendering results? Here it comes in handy, it allows us to check the current location.

We click the element that has the text "Goblin Store" on it, and then we expect that we end up on the root url.

CartWidget

Let's move on to the `CartWidget` component. This component displays the number of products in the cart. Also, the whole component acts as a link, so if you click on it, you get redirected to the cart summary page.

This component also uses an icon `cart.svg`, so it has a dedicated folder called `CartWidget`.

Let's create a test file. Create a new file `src/shared/CartWidget.spec.tsx`:

```
describe("CartWidget", () => {
  it.todo("shows the amount of products in the cart")

  it.todo("navigates to cart summary page on click")
})
```

Add the imports:

```
import { CartWidget } from "./CartWidget";
import { fireEvent } from "@testing-library/react";
import { useCartContext } from "../../CartContext";
```

Ok, we already know how to test the navigation by click. Let's write the test that will check that we get redirected to the cart summary page when we click the widget.

Remove the `todo` from the `navigates to cart summary page on click` test and add the following code there:

```
  it("navigates to cart summary page on click", () => {
    useCartContextMock.mockReturnValue({
      products: [],
    });
    const { getByRole, history } = renderWithRouter(() => <CartWidget /\
>);

    fireEvent.click(getByRole("link"));

    expect(history.location.pathname).toEqual("/cart");
  });
```

Here we use the getByRole[71] selector from `@testing-library/react`. This selector uses the `aria-role` attribute to find the element. Some elements have the default `aria-role` value, for example `<a>` elements, have the `link` role. You can find the complete list of default `aria-role` values on the WHATWG site[72].

[71] https://testing-library.com/docs/dom-testing-library/api-queries#byrole
[72] https://html.spec.whatwg.org/multipage/index.html#contents

So in our test, we click the link element and then check if we end up on the `/cart` route.

Now let's test that `CartWidget` renders the number of products in the cart correctly.

The `CartWidget` component does not have any logic to track the number of products in the cart. It just takes the value provided by the `CartContext` through the `useCartContext` hook.

Open the `CartWidget` component code. It's located in `src/shared/CartWidget/CartWidget.tsx`:

```
import { Link } from "react-router-dom"
import cart from "./cart.svg"
import { useCartContext } from "../../CartContext"

export const CartWidget = () => {
  const { products } = useCartContext()

  return (
    <Link to="/cart" className="nes-badge is-icon">
      <span className="is-error">{products?.length || 0}</span>
      <img src={cart} width="64" height="64" alt="cart" />
    </Link>
  )
}
```

Look what happens here. We get the products array from the `useCartContext` hook.

Go back to the test code. Let's test that we render the amount of products in the cart correctly:

```
jest.mock("../../CartContext", () => ({
  useCartContext: jest.fn(),
}));

const useCartContextMock = useCartContext as unknown as jest.Mock<
  Partial<ReturnType<typeof useCartContext>>
>;

describe("CartWidget", () => {
  it("shows the amount of products in the cart", () => {
    useCartContextMock.mockReturnValue({
      products: [
        {
          name: "Product foo",
          price: 0,
          image: "image.png",
        },
      ],
    });

    const { container } = renderWithRouter(() => <CartWidget />);

    expect(container.innerHTML).toMatch("1");
  });
  // ...
});
```

Here we define a mock version of the `useCartContext`. The mock version returns only the `products` field with a hardcoded product.

But there is a problem. We want to tell TypeScript that this `useCartContextMock` hook is actually a mock that returns the same value as the original `useCartContext` hook.

By defaut Jest will reset all the mocks on each spec run. So in order to mock the returned values of this hook we'll need to mock it for every test.

To do this we define a constant `useCartContextMock` and specified its type to be `jest.Mock`.

As the types of the `useCartContext` and `jest.Mock` are very different we had to cast the type of the `useCartContext` to `unknown` first and only then to `jest.Mock`.

Then we specified the actual type of the generic `jest.Mock` function. We want to be able to skip some fields of the mocked returned value, so we specify the type as a `Partial`.

```
jest.mock("../../CartContext", () => ({
  useCartContext: jest.fn(),
}));

const useCartContextMock = useCartContext as unknown as jest.Mock<
  Partial<ReturnType<typeof useCartContext>>
>;
```

We used two utility types provided by TypeScript:

- `ReturnType` - constructs type from function return type. For example if we have a function type `() => string`, we can use `ReturnType<() => string>` to get `string`.
- `Partial` - allows us to create a type that accepts a subset of fields of the original object.

Now our `CartWidget` test should be passing.

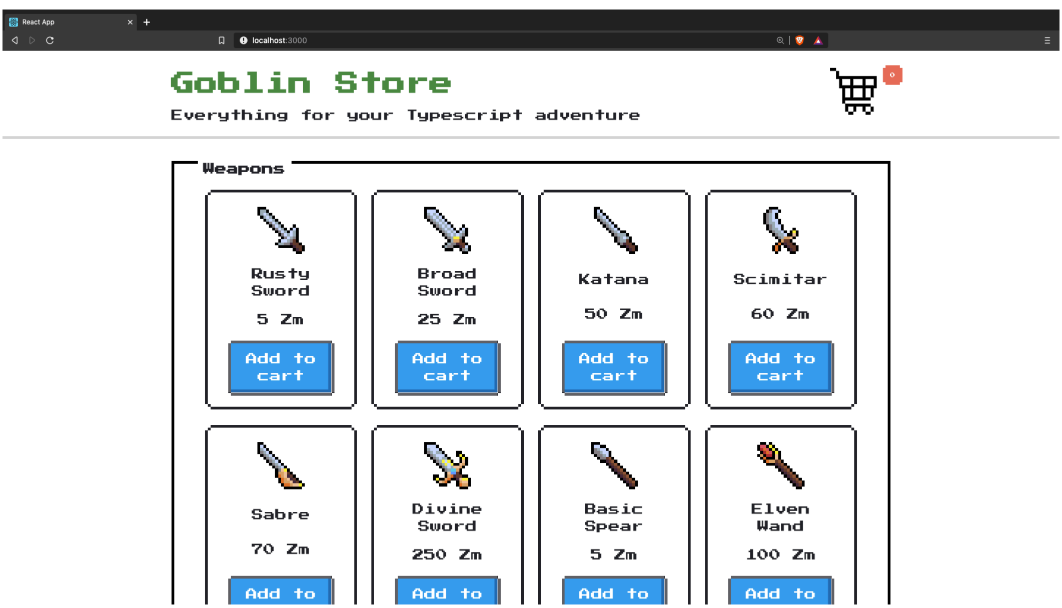
Loader Component

Our `Loader` component does not contain any logic. In our test we'll only make sure that it renders correctly:

```
import { Loader } from "./Loader";
import { render } from "@testing-library/react";

describe("Loader", () => {
  it("renders correctly", () => {
    const { container } = render(<Loader />);
    expect(container.innerHTML).toMatch("Loading");
  });
});
```

Home Page

Our home page renders the list of products that we get from the backend.



Home page

Open the `src/Home` folder. I'll walk you through the files there:

```
index.tsx
Home.tsx
Product.tsx
```

First of all, we have an `index.ts` file. It's used to control the visibility of the module contents.

```
export * from './Home'
```

As you can see, we export only the `Home` component. The `Product` component won't be visible outside this module. The benefit of it is that the `Product` component won't be accidentally used on other pages. If we decide to reuse it – we'll have to move it to the `shared` folder

Now let's move on to the tests. Create a test file called `src/Home/Home.spec.tsx`.

The `Home` component gets the data from the `useProducts` hook and then does one of three things:

- while products are being loaded
  - renders the `<Loader />`
- if it gets an error from `useProducts`
  - render the error message
- when products are loaded successfully
  - render the products list

Let's reflect it in our tests. Define a `describe` block for each case listed above:

```
describe("Home", () => {
  describe("while loading", () => {
    it.todo("renders loader")
  })

  describe("with data", () => {
    it.todo("renders categories with products")
  })

  describe("with error", () => {
    it.todo("renders error message")
  })
})
```

Add the imports to the test file:

```
import { Home } from "./Home"
import { Category } from "../shared/types"
import { render } from "@testing-library/react"
import { ProductCardProps } from "./ProductCard"
import { useProducts } from "./useProducts"
```

Now let's write the individual test cases.

First mock the `useProducts` hook:

```
jest.mock("./useProducts", () => ({
  useProducts: jest.fn()
}))

const useProductsMock = useProducts as unknown as jest.Mock<
  Partial<ReturnType<typeof useProducts>>
>
```

Now we'll be able to mock the return value of this hook for each test.

Let's test that the *loading* state is processed correctly:

```
describe("Home", () => {
  describe("while loading", () => {
    it("renders loader", () => {
      useProductsMock.mockReturnValue({
        categories: [],
        isLoading: true,
        error: false
      })

      const { container } = render(<Home />)

      expect(container.innerHTML).toMatch("Loading")
    })
  })
  // ...
})
```

Here we defined the `useProducts` return value so that it contains `isLoading: true` and then we verified that in this case, we'll find the word `"Loading"` in rendered layout.

Then let's check that our error state will also be processed correctly:

```
  describe("with error", () => {
    it("renders error message", () => {
      useProductsMock.mockReturnValue({
        categories: [],
        isLoading: false,
        error: true
      })

      const { container } = render(
        <Home />
      )

      expect(container.innerHTML).toMatch("Error")
```

```
    })
  })
```

This test is very similar to the loading state test, the only difference is that now `error` is `true` and `isLoading` is `false`.

Let's verify that when we get the products, we render them correctly.

`Home` component uses the `ProductCard` component to render products. I don't want to introduce it as a dependency to this test. Let's mock the `ProductCard` component, to do this we first need to import the `ProductCardProps` type:

```
import { ProductCardProps } from "./ProductCard"
```

Then we can define the mock:

```
jest.mock("./ProductCard", () => ({
  ProductCard: ({ datum }: ProductCardProps) => {
    const { name, price, image } = datum
    return (
      <div>
        {name} {price} {image}
      </div>
    )
  }
}))
```

Our mock renders the product data that it gets through the props. This way we'll be able to verify that we pass this data to the real component as well.

Let's verify that if we render the home page with this data, we'll see the category titled `Category foo`, and it will contain the rendered product:

```
  describe("with data", () => {
    it("renders categories with products", () => {
      const category: Category = {
        name: "Category Foo",
        items: [
          {
            name: "Product foo",
            price: 55,
            image: "/test.jpg"
          }
        ]
      }

      useProductsMock.mockReturnValue({
        categories: [category],
        isLoading: false,
        error: false
      })

      const { container } = render(
        <Home />
      )

      expect(container.innerHTML).toMatch("Category Foo")
      expect(container.innerHTML).toMatch("Product foo 55 /test.jpg")
    })
  })
```

Here we don't need to test that if we click on the product's `Add to cart` button we'll add the product to the cart. We'll do that in the `ProductCart` component tests.

ProductCard Component

Moving on to the `ProductCard` component. Let's see what do we have here.

First of all, we render the product data: the image should have the correct `alt` and `src` tags, and the component should the price and the product name.

Then we render the `Add to cart` button. This button can have one of two states. If the product was added to the cart, the button should be disabled and the text on it should say `Added to cart`. Otherwise, it should be `Add to cart` and the button should trigger the `addToCart` function from the `useCart` hook when clicked.

Let's write the test. Create the `src/Home/ProductCard.spec.tsx` file with the following contents:

```
describe("ProductCard", () => {
  it.todo("renders correctly")

  describe("when the product is in the cart", () => {
    it.todo("the 'Add to cart' button is disabled")
  })

  describe("when the product is not in the cart", () => {
    describe("on 'Add to cart' click", () => {
      it("calls the 'addToCart' function")
    })
  })
})
```

The first thing we can test is that our `ProductCard` renders correctly. There are two states in which it should be rendered:

- product is in the cart
  - render with disabled button saying `Added to cart`
- product is not in the cart
  - render with primary button saying `Add to cart`
  - on `Add to cart` click
    - add the product to the cart

Also in both cases, it renders the `name`, the `price`, and the `image` of the product.

Add the necessary imports:

```
import { render, fireEvent } from "@testing-library/react"
import { ProductCard } from "./ProductCard"
import { Product } from "../shared/types"
import { useCartContext } from "../CartContext"
```

First let's check that our product renders the data correctly. Define the `useCartContext` mock:

```
jest.mock("../CartContext", () => ({
  useCartContext: jest.fn()
}))

const useCartContextMock = useCartContext as unknown as jest.Mock<
  Partial<ReturnType<typeof useCartContext>>
>
```

We'll need the products for several tests, so let's define them as a constant outside of the top level `describe` block:

```
const product: Product = {
  name: "Product foo",
  price: 55,
  image: "/test.jpg"
}
```

Now let's write the "renders correctly" test:

```
describe("ProductCard", () => {
  it("renders correctly", () => {
    useCartContextMock.mockReturnValue({
      addToCart: () => {},
      products: [product]
    })
    const { container, getByRole } = render(
      <ProductCard datum={product} />
    )

    expect(container.innerHTML).toMatch("Product foo")
    expect(container.innerHTML).toMatch("55 Zm")
    expect(getByRole("img")).toHaveAttribute("src", "/test.jpg")
  })
  // ...
})
```

Here we make sure that we can find the product `name` and `price` and that the image has correct attributes.

Test that if the product is in the cart already, the `Add to cart` button will be disabled:

```
  describe("when the product is in the cart", () => {
    it("the 'Add to cart' button is disabled", () => {
      useCartContextMock.mockReturnValue({
        addToCart: () => {},
        products: [product]
      })

      const { getByRole } = render(<ProductCard datum={product} />)
      expect(getByRole("button")).toBeDisabled()
    })
  })
```

Now let’s test how our component works when its product is not in the cart. Add this code to the “when product is not in the cart” `describe` block:

```
describe("when the product is not in the cart", () => {
  describe("on 'Add to cart' click", () => {
    it("calls the 'addToCart' function", () => {
      const addToCart = jest.fn()
      useCartContextMock.mockReturnValue({
        addToCart,
        products: []
      })

      const { getByText } = render(<ProductCard datum={product} />)

      fireEvent.click(getByText("Add to cart"))
      expect(addToCart).toHaveBeenCalledWith(product)
    })
  })
})
```

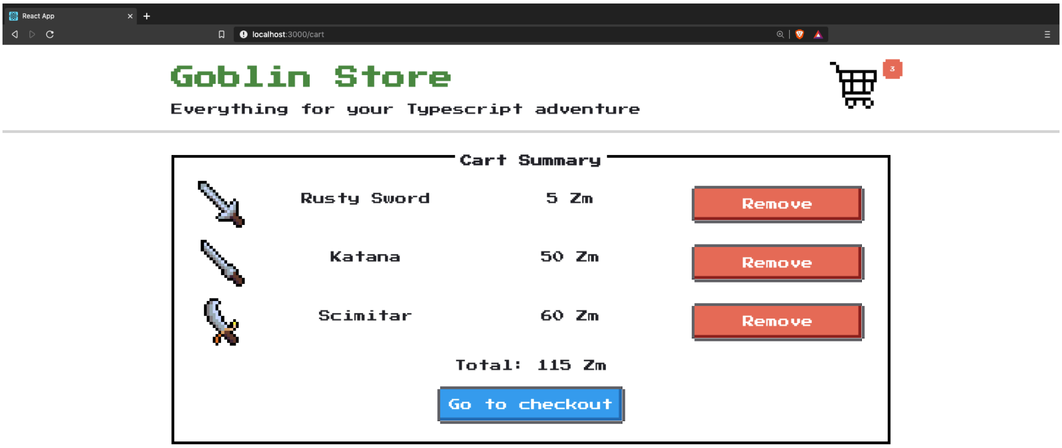
Here we set the cart products list to be an empty array. We use `jest.fn()` to mock our `addToCart` function:

We fire the `click` event on our button and then we check that the `addToCart` function was called with the product data.

We are done testing the `Home` page components. We'll test the `useProducts` hook later, but for now, let's move on to the `Cart` page.

Cart page

This page renders the list of items that you've added to the cart.



Cart summary page

Here you can review the products and remove them from the cart if you've changed your mind and don't want to buy them any more.

If there are no products, this page renders a message saying that the cart is empty, and provides a button to go back to the main page.

Open the `src/Cart` folder. Here you should see the following files:

```
index.ts
Cart.tsx
CartItem.tsx
```

The `index.ts` file controls the module visibility. It exports only the `Cart` page component.

`CartItem` represents the product that was added to the cart. It also renders the *Remove* button, that you can click to remove the item from the cart.

Cart component

Open the `src/Cart/Cart.tsx`. Here we use the `useCart` hook to get the cart data.

The `Cart` component has a condition in its layout code:

- when the products array is empty
  - renders the "empty cart" message with the link to the products page
  - on products page link redirects to `/`
- with products in the cart
  - renders the list of products
  - renders the total price
  - renders the "Go to checkout" button
  - on "Go to checkout" click
    - redirects to `/checkout`

Create the test file `src/Cart/Cart.spec.tsx` with the following contents:

```
describe("Cart", () => {
  describe("without products", () => {
    it.todo("renders empty cart message")

    describe("on 'Back to main page' click", () => {
      it.todo("redirects to '/'")
    })
  })

  describe("with products", () => {
    it.todo("renders cart products list with total price")

    describe("on 'go to checkout' click", () => {
      it.todo("redirects to '/checkout'")
    })
  })
})
```

First, let's check that our `Cart` component will render the "empty cart" message with the link if the cart is empty.

Import the `Cart` component and the `useCartContext` hook:

```
import { Cart } from "./Cart"
  // ...
import { useCartContext } from "../CartContext"
```

Mock the `useCartContext` hook, so that we can change the returned value for the tests:

```
jest.mock("../CartContext", () => ({
  useCartContext: jest.fn()
}))

const useCartContextMock = useCartContext as unknown as jest.Mock<
  Partial<ReturnType<typeof useCartContext>>
>
```

Now inside the products block, mock the `useCartContext` return value to contain an empty `products` array:

```
describe("Cart", () => {
  describe("without products", () => {
    beforeEach(() => {
      useCartContextMock.mockReturnValue({
        products: []
      })
    })
  // ...
  })
  // ...
})
```

Now we can test that with the empty products list we render the empty cart message:

```
it("renders empty cart message", () => {
  const { container } = renderWithRouter(() => <Cart />)
  expect(container.innerHTML).toMatch("Your cart is empty.")
})
```

Now it's time to check that if we click the `Back to main page` button we get redirected to the main page.

Here we'll need to simulate click, so import `fireEvent`:

```
import { fireEvent } from "@testing-library/react"
```

Add the following code inside the `on 'Back to main page' click` block:

```
    describe("on 'Back to main page' click", () => {
      it("redirects to '/'", () => {
        const { getByText, history } = renderWithRouter(() => (
          <Cart />
        ))

        fireEvent.click(getByText("Back to main page."))

        expect(history.location.pathname).toBe("/")
      })
    })
```

Here we use the `renderWithRouter` helper that we defined at the beginning of this chapter. We find an element that has the `Back to main page` text on it, click it and then verify that we ended up on the root route.

Now let's verify that the cart with products in it also renders correctly. Inside the `with products` block, define a `beforeEach` block where you'll mock the array of products:

```
describe("with products", () => {
  beforeEach(() => {
    const products = [
      {
        name: "Product foo",
        price: 100,
        image: "/image/foo_source.png"
      },
      {
        name: "Product bar",
        price: 100,
        image: "/image/bar_source.png"
      }
    ]

    useCartContextMock.mockReturnValue({
      products,
      totalPrice: () => 55
    })
  })
// ...
})
```

Now let's check if the component will render correctly. It means that the products are rendered and also that we display the total price.

Before we write the test let's mock the `CartItem` component. Import the `CartItemProps` type:

```
import { CartItemProps } from "./CartItem"
```

Then add this code at the beginning of our test file:

```
jest.mock("./CartItem", () => ({
  CartItem: ({ product }: CartItemProps) => {
    const { name, price, image } = product
    return (
      <div>
        {name} {price} {image}
      </div>
    )
  }
}))
```

Now we can implement the `renders cart products list with total price` test case:

```
    it("renders cart products list with total price", () => {
      const { container } = renderWithRouter(() => <Cart />)

      expect(container.innerHTML).toMatch(
        "Product foo 100 /image/foo_source.png"
      )
      expect(container.innerHTML).toMatch(
        "Product bar 100 /image/bar_source.png"
      )
      expect(container.innerHTML).toMatch("Total: 55 Zm")
    })
```

Here we check that we can find product names, prices, and image URLs in the rendered layout.

Let's verify that if we click the `Go to checkout` button it will redirect us to the checkout page:

```
    describe("on 'go to checkout' click", () => {
      it("redirects to '/checkout'", () => {
        const { getByText, history } = renderWithRouter(() => (
          <Cart />
        ))

        fireEvent.click(getByText("Go to checkout"))

        expect(history.location.pathname).toBe("/checkout")
      })
    })
```

This test is very similar to the one that checks that the empty state button redirects you to the main page.

CartItem component

Time to test our `CartItem` component. This component renders the product information and also renders a `Remove` button that allows removal of the product from the cart. If we summarize its functionality it will look like this:

- renders correctly
- on `Remove` button click
  - removes the item from the cart

Create a new file called `src/Cart/CartItem.spec.tsx` and plan out the tests.

```
describe("CartItem", () => {
  it.todo("renders correctly")

  describe("on 'Remove' click", () => {
    it.todo("calls passed in function")
  })
})
```

Let's test that it renders correctly first. Hardcode some product data inside the top-level `describe` block:

```
const product: Product = {
  name: "Product Foo",
  price: 100,
  image: "/image/source.png"
}
```

Import the `Product` type and the `CartItem` component:

```
import { CartItem } from "./CartItem"
import { Product } from "../shared/types"
```

Now inside the `renders correctly` block add the following code:

```
  it("renders correctly", () => {
    const {
      container,
      getByAltText
    } = renderWithRouter(() => (
      <CartItem
        product={product}
        removeFromCart={() => {}}
      />
    ))

```

```
    expect(container.innerHTML).toMatch("Product Foo")
    expect(container.innerHTML).toMatch("100 Zm")
    expect(getByAltText("Product Foo")).toHaveAttribute(
      "src",
      "/image/source.png"
    )
  })
```

Here we verify that all the data related to the product is rendered, we can find the image by its `alt` attribute and it has the correct `src`.

Let's move on and test that when a user clicks the `Remove` button, we call the function passed through the `removeFromCart` prop.

We'll need import the `fireEvent` for this test:

```
import { fireEvent } from "@testing-library/react"
```

Add this code inside the `on 'Remove' click` block:

```
    it("calls passed in function", () => {
      const removeFromCartMock = jest.fn()

      const { getByText } = renderWithRouter(() => (
        <CartItem
          product={product}
          removeFromCart={removeFromCartMock}
        />
      ))

      fireEvent.click(getByText("Remove"))

      expect(removeFromCartMock).toBeCalledWith(product)
    })
```

Here we defined a mock function using `jest.fn`. The cool thing about those is that we can check if they have been called. We can even verify that such a function was called with specific arguments. Here we check that when we click the `Remove` button, our `removeFromCartMock` gets called with the product rendered by this component.

Checkout Page

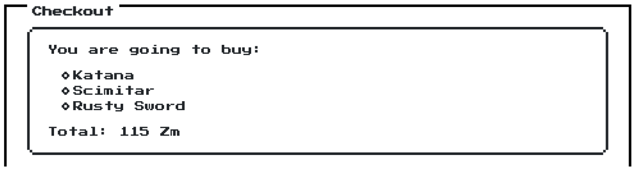
This is the page where the user can input their payment credentials and confirm the order.

Checkout page

We also render the list of products that the user is going to buy here.

CheckoutList component

The list of products is rendered by the `CheckoutList` component.



Checkout list

This component also uses `CartContext` through the `useCart` hook.

It has one task, so it better do it well! Let's test the `CheckoutList`. Create a new file `src/Checkout/CheckoutList.spec.tsx`:

```
import { CheckoutList } from "./CheckoutList"
import { Product } from "../shared/types"
import { render } from "@testing-library/react"

describe("CheckoutList", () => {
  it.todo("renders list of products")
})
```

As you can see we are only going to test that `CheckoutList` correctly renders the list of products provided to it:

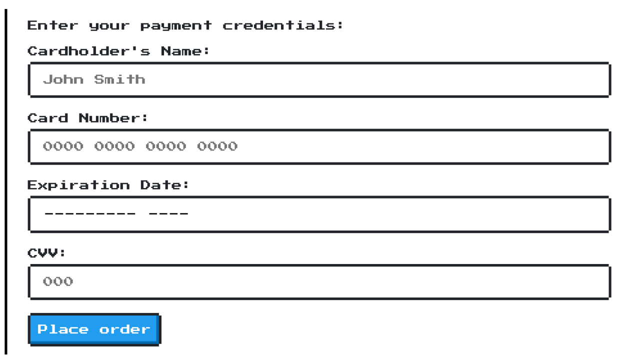
```
  it("renders list of products", () => {
    const products: Product[] = [
      {
        name: "Product foo",
        price: 10,
        image: "/image.png"
      },
      {
        name: "Product bar",
        price: 10,
        image: "/image.png"
      }
    ]

```

```
    const { container } = render(
      <CheckoutList products={products} />
    )
    expect(container.innerHTML).toMatch("Product foo")
    expect(container.innerHTML).toMatch("Product bar")
  })
```

We verify that we can find the titles of the provided products in the rendered layout.

Testing The Form

The next component that we are going to test is `CheckoutForm`.



Checkout form

Here we want to verify the following things:

- When the input values are invalid
  - The form renders an error message
- When the input values are valid
  - When you click the `Order` button
    - The submit function is called

Create the test file `src/Checkout/CheckoutForm.spec.tsx` with the following contents:

```
describe("CheckoutForm", () => {
  it.todo("renders correctly")

  describe("with invalid inputs", () => {
    it.todo("shows errors")
  })

  describe("with valid inputs", () => {
    describe("on place order button click", () => {
      it("calls submit function with form data")
    })
  })
})
```

When we render the form we expect to see the following fields:

- Card holder's name
- Card number
- Card expiration date
- CVV number

This will be our first test.

Add the imports:

```
import { render, fireEvent, waitFor } from "@testing-library/react"
import { CheckoutForm } from "./CheckoutForm"
import { act } from "react-dom/test-utils"
```

Remove the `todo` part from the `renders correctly` test and add the following code:

```
describe("CheckoutForm", () => {
  it("renders correctly", () => {
    const { container } = render(<CheckoutForm />)

    expect(container.innerHTML).toMatch("Cardholders Name")
    expect(container.innerHTML).toMatch("Card Number")
    expect(container.innerHTML).toMatch("Expiration Date")
    expect(container.innerHTML).toMatch("CVV")
  })

  // ...
})
```

Here we verify that all the form fields are present.

At this moment you might get an error regarding missing mutation observer, to fix it we'll need to install a shim and include it into the `setupTests.ts` file.

Let's install the shim first:

```
yarn add mutationobserver-shim --ignore-engines
```

Then go to the `src/setupTests.ts` and add the import there:

```
import "mutationobserver-shim"
```

Now the test should be passing.

Next we check that the form will show the errors if we click `Place Order` with invalid values. Go back to the `src/Checkout/CheckoutForm.spec.tsx` and add the following test:

```
  describe("with invalid inputs", () => {
    it("shows errors ", async () => {
      const { container, getByText } = render(<CheckoutForm />)

      await act(async () => {
        fireEvent.click(getByText("Place order"))
      })

      expect(container.innerHTML).toMatch("Error:")
    })
  })
```

Here we expect that if we click the `Place Order` button while the form is not filled in, it will render an error message.

Let's check that if we provide valid values to our form inputs and then click the `Place Order` button, the form component will call the `onSubmit` function.

Inside the `calls submit function with form data` block define the `mockSubmit` function:

```
describe("with valid inputs", () => {
  describe("on place order button click", () => {
    it("calls submit function with form data", async () => {
      const mockSubmit = jest.fn()
// ...
    })
  })
})
```

Make sure to make the `it` block `async`.

And then use it to render our form component:

```
const { getByLabelText, getByText } = render(
  <CheckoutForm submit={mockSubmit} />
)
```

Now we will fill in the form inputs. But the trick is that it will trigger state updates in our form. Our form uses React hook form[73] to manage the inputs. It means that the inputs are controlled[74] and filling them in triggers state updates.

When you have the code in your test that triggers state updates in your components, you need to wrap it into act[75].

Let's fill in the inputs:

```
fireEvent.change(getByLabelText("Cardholders Name:"), {
  target: { value: "Bibo Bobbins" }
})
fireEvent.change(getByLabelText("Card Number:"), {
  target: { value: "0000 0000 0000 0000" }
})
fireEvent.change(getByLabelText("Expiration Date:"), {
  target: { value: "3020-05" }
})
fireEvent.change(getByLabelText("CVV:"), {
  target: { value: "123" }
})
```

Then click the `Place order` button. Technically we could put it into the same `act` block, but I decided that it is clearer if first we create specific conditions and then we perform an action:

```
fireEvent.click(getByText("Place order"))
```

Finally we can check that our mock function was called:

[73] https://react-hook-form.com/
[74] https://reactjs.org/docs/forms.html#controlled-components
[75] https://reactjs.org/docs/test-utils.html#act

```
await waitFor(() => expect(mockSubmit).toHaveBeenCalled())
```

The form submission happens asynchronously so we use the `waitFor` function from the React Testing Library.

Testing The FormField

The checkout form uses the `FormField` component to render the inputs. This component renders label, input, and if we pass an error object to it, it also renders a paragraph with an error message.

It also supports normalization. For example, we can pass a `normalize` function to it that will limit the length of the input value. It is needed for the `CVV` field, which accepts only three digits. This `normalize` function could also format the input in some specific way. For example, our card number field needs to be formatted into four blocks of four digits each.

Create a new file called `src/Checkout/FormField.spec.tsx`:

```
import { render, fireEvent } from "@testing-library/react"
import { FormField } from "./FormField"

describe("FormField", () => {
  it.todo("renders correctly")

  describe("with error", () => {
    it.todo("renders error message")
  })

  describe("on change", () => {
    it.todo("normalizes the input")
  })
})
```

First let's check that our `FormField` component renders correctly:

```
it("renders correctly", () => {
  const { getByLabelText } = render(
    <FormField label="Foo label" name="foo" />
  )
  const input = getByLabelText("Foo label:")
  expect(input).toBeInTheDocument()
  expect(input).not.toHaveClass("is-error")
  expect(input).toHaveAttribute("name", "foo")
})
```

Here we verify that we render the `input` element with the correct `name` value and without the `is-error` class by default. Also, note that we find it by the label value, so we additionally verify that the `label` was rendered as well.

Let's verify that if we pass an error object to our `FormField`, it will render the error message:

```
describe("with error", () => {
  it("renders error message", () => {
    const { getByText } = render(
      <FormField
        label="Foo label"
        name="foo"
        errors={{ message: "Example error" }}
      />
    )
    expect(getByText("Error: Example error")).toBeInTheDocument()
  })
})
```

Here we try to find the error message in the rendered layout.

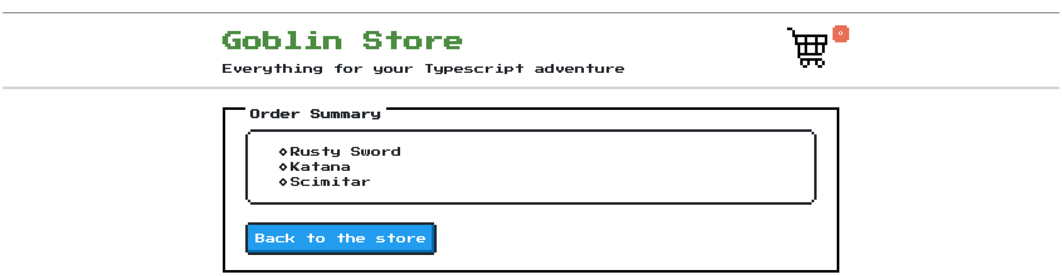
Next let's verify that the `normalize` function will work. Add this test inside the `on change` describe block:

```
  it("normalizes the input", () => {
    const { getByLabelText } = render(
      <FormField
        label="Foo label"
        name="foo"
        errors={{ message: "Example error" }}
        normalize={(value:string) => value.toUpperCase()}
      />
    )

    const input = getByLabelText(
      "Foo label:"
    ) as HTMLInputElement
    fireEvent.change(input, { target: { value: "test" } })

    expect(input.value).toEqual("TEST")
  })
```

Here we define the `normalize` function to call the `toUppercase` method on input values. Then we expect that the input value will be capitalized.

Order summary page

This page fetches the order information from the backend by `orderId` and displays the products included in the order.



Order summary

It gets the `orderId` from the current location query parameters and makes a request to the backend using the `api` module.

Create a new file `src/OrderSummary/OrderSummary.spec.tsx` with the following code:

```
describe("OrderSummary", () => {
  afterEach(jest.clearAllMocks)

  describe("while order data being loaded", () => {
    it.todo("renders loader")
  })

  describe("when order is loaded", () => {
    it.todo("renders order info")

    it.todo("navigates to main page on button click")
  })

  describe("without order", () => {
    it.todo("renders error message")
  })
})
```

First, let's test that in the loading state we'll render the `Loader` component. Add the imports:

```
import { OrderSummary } from "./OrderSummary"
import { render, fireEvent } from "@testing-library/react"
import { useOrder } from "./useOrder"
```

Mock the `useOrder` hook:

```
jest.mock("./useOrder", () => ({
  useOrder: jest.fn()
}))

const useOrderMock = useOrder as unknown as jest.Mock<
  Partial<ReturnType<typeof useOrder>>
>
```

Now define the test for the loading state:

```
describe("OrderSummary", () => {
  afterEach(jest.clearAllMocks)

  describe("while order data being loaded", () => {
    it("renders loader", () => {
      useOrderMock.mockReturnValue({
        isLoading: true,
        order: undefined
      })

      const { container }= render(<OrderSummary />)
                        expect(container.innerHTML).toMatch("Loading")
    })
  })
  // ...
})
```

Let's test that when an order is loaded successfully, we render the products list from it:

```
describe("when order is loaded", () => {
  beforeEach(() => {
    useOrderMock.mockReturnValue({
      isLoading: false,
      order: {
        products: [
          {
            name: "Product foo",
            price: 10,
            image: "image.png"
          }
        ]
      }
    })
  })

  it("renders order info", () => {
    const { container } = renderWithRouter(() => <OrderSummary />)

    expect(container.innerHTML).toMatch("Product foo")
  })
// ...
})
```

When order information is loaded successfully, we also render a link to the main page. Let's write a test for that as well:

```
  it("navigates to main page on button click", () => {
    const { getByText, history } = renderWithRouter(() => (
      <OrderSummary />
    ))

    fireEvent.click(getByText("Back to the store"))

    expect(history.location.pathname).toEqual("/")
  })
```

Let's test that when the order data cannot be loaded, we render a failure message:

```
describe("without order", () => {
  it("renders error message", () => {
    useOrderMock.mockReturnValue({
      isLoading: false,
      order: undefined
    })

    const { container } = render(<OrderSummary />)

    expect(container.innerHTML).toMatch("Couldn't load order info.")
  })
})
```

At this point, we've tested all the components that our app has. It's time to test the hooks.

Testing React Hooks

At this point we've tested all the regular components that we had. The only things left for testing are the hooks and the context provider. In this part we'll test the hooks. We can skip testing the `CartContext`, because all the logic is inside the `useCart` hook.

Let's go back to our `Home` page and test how we fetch the products list.

Our `Home` page uses the `useProducts` hook to fetch the products from the backend.

To test the hooks we'll have to install the `@testing-library/react-hooks`. From the root of the project run the following command:

```
yarn add --dev @testing-library/react-hooks@5.1.2 --ignore-engines
```

Testing useProducts

Our `useProducts` hook does a bunch of things:

- fetches products on mount
- while the data is loading
  - returns `isLoading = true`
- if loading fails
  - returns `error = true`
- when data is loaded
  - returns the loaded data

Create a new file `src/Home/useProducts.spec.ts`:

```
describe("useProducts", () => {
  it.todo("fetches products on mount")

  describe("while waiting API response", () => {
    it.todo("returns correct loading state data")
  })

  describe("with error response", () => {
    it.todo("returns error state data")
  })

  describe("with successful response", () => {
    it.todo("returns successful state data")
  })
})
```

First let's add the imports:

```
import { renderHook, act } from "@testing-library/react-hooks"
import { useProducts } from "./useProducts"
import { getProducts } from "../utils/api"
```

Mock the `getProducts` api function:

```
jest.mock("../utils/api", () => ({
  getProducts: jest.fn()
}))

const getProductsMock = getProducts as unknown as jest.Mock<
  Partial<ReturnType<typeof getProducts>>
>
```

Now let's test that the `useProducts` hook will start fetching data when it is mounted:

```
describe("useProducts", () => {
  it("fetches products on mount", async () => {
    await act(async () => {
      renderHook(() => useProducts())
    })

    expect(getProducts).toHaveBeenCalled()
  })
  // ...
})
```

We render the hook using the `renderHook` method from `@testing-libary/react-hooks` and then we check if the mocked `getProducts` function was called.

Let's test the waiting state when the data is being loaded.

```
describe("while waiting API response", () => {
  it("returns correct loading state data", () => {
    getProductsMock.mockReturnValue(new Promise(() => {}))

    const { result } = renderHook(() => useProducts())
    expect(result.current.isLoading).toEqual(true)
    expect(result.current.error).toEqual(false)
    expect(result.current.categories).toEqual([])
  })
})
```

Note how we define the `getProducts` return value:

```
getProductsMock.mockReturnValue(new Promise(() => {}))
```

We make it return a `Promise` that will never resolve (or reject).

This way we can make sure that our `useProducts` hook will return a correct set of values while we are fetching the data.

Let's test that we correctly handle loading failure:

```
describe("with error response", () => {
  it("returns error state data", async () => {
    getProductsMock.mockReturnValue(
      new Promise((resolve, reject) => {
        reject("Error")
      })
    )

    const { result, waitForNextUpdate } = renderHook(() =>
      useProducts()
    )

    await act(() => waitForNextUpdate())

    expect(result.current.isLoading).toEqual(false)
```

```
      expect(result.current.error).toEqual("Error")
      expect(result.current.categories).toEqual([])
    })
  })
```

Here we mock the API method so that it instantly rejects with an error.

```
getProductsMock.mockReturnValue(
  new Promise((resolve, reject) => {
    reject("Error")
  })
)
```

The data fetching happens inside of the `async` function in our hook, and as a result it will update its state. To handle it correctly we use `act` to wait for the next update before we can test our expectations:

```
await act(() => waitForNextUpdate())
```

Let's test the happy path, where we successfully get the data and return it from our hook. We are going to add the `returns successful state data` test. Begin by mocking the API function so that it resolves with products data:

```
describe("with successful response", () => {
  it("returns successful state data", async () => {
    getProductsMock.mockReturnValue(
      new Promise((resolve, reject) => {
        resolve({
          categories: [{ name: "Category", items: [] }]
        })
      })
    )
// ...
  })
})
```

Then render the hook and wait for next update, so that the internal state of our hook has the correct value:

```
      const { result, waitForNextUpdate } = renderHook(() =>
        useProducts()
      )

      await act(() => waitForNextUpdate())
```

Check the expectations:

```
expect(result.current.isLoading).toEqual(false)
expect(result.current.error).toEqual(false)
expect(result.current.categories).toEqual([
  {
    name: "Category",
    items: []
  }
])
```

I like to be verbose when I check the data inside my tests, it makes it easier for me to see if the returned data is wrong.

Testing `useCart`

Another hook that we have in our application is `useCart`. This hook allows us to get the list of products in the cart, add new products, or clear the cart.

This hook provides a bunch of functions and we'll check each of them in our tests. Create a new file `src/CartContext/useCart.spec.ts` with the following code:

```
describe("useCart", () => {
  describe("on mount", () => {
    it.todo("it loads data from localStorage")
  })

  describe("#addToCart", () => {
    it.todo("adds item to the cart")
  })

  describe("#removeFromCart", () => {
    it.todo("removes item from the cart")
  })

  describe("#totalPrice", () => {
    it.todo("returns total products price")
  })

  describe("#clearCart", () => {
    it.todo("removes all the products from the cart")
  })
})
```

Here I'm using a naming convention from RSpec[76] where function tests are called with a pound sign prefix: `#functionName`.

Make the necessary imports:

```
import { useCart } from "./useCart"
import { renderHook, act } from "@testing-library/react-hooks"
import { Product } from "../shared/types"
```

Let's go through the planned tests. First, let's check that when the `useCart` hook mounts, it loads the data from `localStorage`. Let's start by mocking the `localStorage`.

Define the `localStorage` constant right after the imports:

[76] https://rspec.rubystyle.guide/

```
const localStorageMock = (() => {
  let store: { [key: string]: string } = {}
  return {
    clear: () => {
      store = {}
    },
    getItem: (key: string) => {
      return store[key] || null
    },
    removeItem: (key: string) => {
      delete store[key]
    },
    setItem: (key: string, value: string) => {
      store[key] = value ? value.toString() : ""
    }
  }
})()
```

Then assign it on the `window` object using `Object.assign` method:

```
Object.defineProperty(window, "localStorage", {
  value: localStorageMock
})
```

`localStorage` is a read-only property, you cannot assign a value to it directly. You'll get an error:

```
window.localStorage = localStorageMock;
// Cannot assign to 'localStorage' because it is a read-only property.
```

One last thing before we move on to the test. Add this clean-up code inside the top-level `describe`:

```
describe("useCart", () => {
  afterEach(() => {
    localStorageMock.clear()
    jest.restoreAllMocks()
  })
  // ...
})
```

This way we won't have to manually clean up the mocked `localStorage` after each test.

Now we are ready to test that our hook will load its initial state from `localStorage`:

```
  describe("on mount", () => {
    it("loads data from the localStorage", () => {
      const products: Product[] = [
        {
          name: "Product foo",
          price: 0,
          image: "image.jpg"
        }
      ]
      localStorageMock.setItem(
        "products",
        JSON.stringify(products)
      )

      const { result } = renderHook(useCart)

      expect(result.current.products).toEqual(products)
    })
  })
```

Here we set the `products` in `localStorage` to be a string representation of our hardcoded `products` array. Then we render our hook and check if the `products` value that it returns matches the original hardcoded array.

Next make sure that we can add items to the cart:

```
describe("#addToCart", () => {
  it("adds item to the cart", () => {
    const product: Product = {
      name: "Product foo",
      price: 0,
      image: "image.jpg"
    }
    const { result } = renderHook(useCart)

    const setItemSpy = jest.spyOn(localStorageMock, "setItem")

    act(() => {
      result.current.addToCart(product)
    })

    expect(result.current.products).toEqual([product])
    expect(setItemSpy).toHaveBeenCalledWith(
      "products",
      JSON.stringify([product])
    )
    setItemSpy.mockRestore()
  })
})
```

Here we hardcode a `product`, render our hook, and call the `addToCart` method. We wrap the `addToCart` method into `act` because it updates the state inside our hook. Then we verify that the `products` array from our hook matches an array with our hardcoded product. Finally, we check that the data stored in `localStorage` is also correct.

Moving on to `#removeFromCart` - this method should remove an existing product from the cart and update the data in `localStorage`.

Let's write the callback for the `removes item from the cart` block.

First define a `product` and save it into `localStorage` as a JSON string:

```
describe("#removeFromCart", () => {
  it("removes item from the cart", () => {
    const product: Product = {
      name: "Product foo",
      price: 0,
      image: "image.jpg"
    }
    localStorageMock.setItem("products", JSON.stringify([product]))
// ...
  })
})
```

Next render our hook:

```
const { result } = renderHook(useCart)
```

Set a spy to track the `setItem` method on `localStorage` and call the `removeFromCart` method. Remember to wrap this call into `act` because it alters the state of the hook:

```
      const setItemSpy = jest.spyOn(localStorageMock, "setItem")

      act(() => {
        result.current.removeFromCart(product)
      })
```

Check the expectations and reset the spy. The `products` array should be empty and `localStorage` should be updated with an empty array:

```
expect(result.current.products).toEqual([])
expect(localStorageMock.setItem).toHaveBeenCalledWith(
  "products",
  "[]"
)
setItemSpy.mockRestore()
```

Let's test the `totalPrice` method. This method should return the sum of the prices of all the products located in the cart.

```
  describe("#totalPrice", () => {
    it("returns total products price", () => {
      const product: Product = {
        name: "Product foo",
        price: 21,
        image: "image.jpg"
      }
      localStorageMock.setItem(
        "products",
        JSON.stringify([product, product])
      )
      const { result } = renderHook(useCart)

      expect(result.current.totalPrice()).toEqual(42)
    })
  })
```

Here we hardcode a `product` that costs twenty-one zorkmid. Then we store an array of two similar products in `localStorage`.

After we render the hook we check that the returned value of the `totalPrice` function is forty-two.

The last method we'll test is `clearCart`.

```
describe("#clearCart", () => {
  it("removes all the products from the cart", () => {
    const product: Product = {
      name: "Product foo",
      price: 21,
      image: "image.jpg"
    }
    localStorageMock.setItem(
      "products",
      JSON.stringify([product, product])
    )
    const { result } = renderHook(useCart)
    const setItemSpy = jest.spyOn(localStorageMock, "setItem")

    act(() => {
      result.current.clearCart()
    })

    expect(result.current.products).toEqual([])
    expect(localStorageMock.setItem).toHaveBeenCalledWith(
      "products",
      "[]"
    )
    setItemSpy.mockRestore()
  })
})
```

Here we also save two instances of `product` in the `localStorage`. Then we render the hook, call the `clearCart` method and check that the cart is empty.

Congratulations

If you've got to this point, you've tested the whole application. Well done!

Patterns in React TypeScript Applications: Making Music with React

Introduction

In this chapter, we're going to talk about some common, useful patterns for React applications and how to use them with proper TypeScript types.

We will talk about:

- *what* these patterns are
- *why* these patterns are useful
- *which* pattern should be used in which situation
- *tradeoffs, constraints, and limitations* of some of the patterns

Particularly, we will talk about React-specific patterns such as *Render-Props* and *Higher-Order Component* and how they are connected to more general concepts.

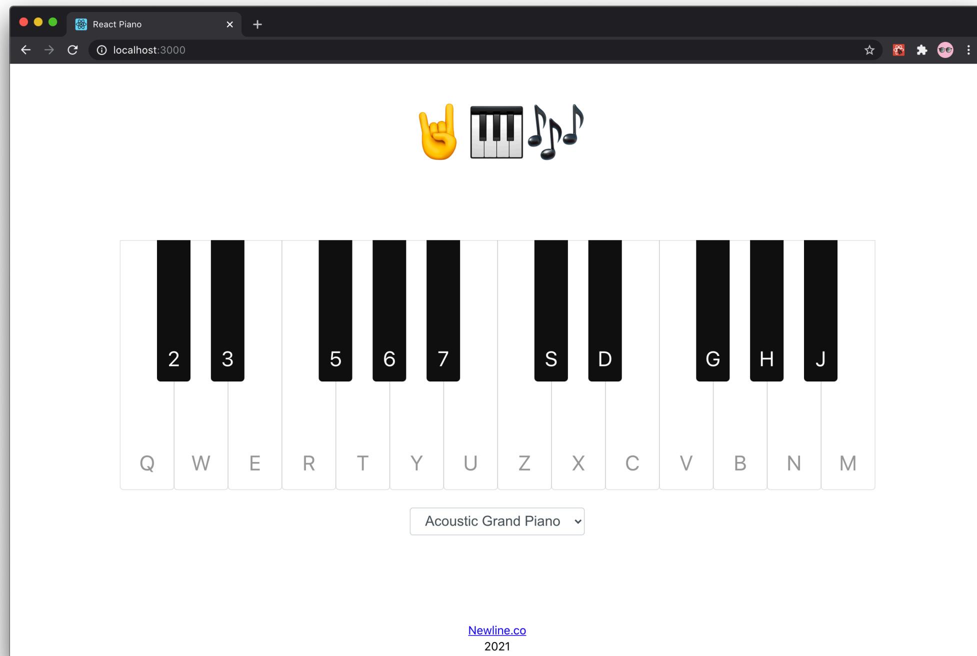
This chapter is going to help you think-in-React by seeing common patterns behind specific code.

What We're Going to Build

The application we're going to build is a virtual piano keyboard with a list of instruments playable with it.

We will use a third-party API to generate musical notes and the browser built-in `AudioContext API` to access a user's sound hardware. The real computer keyboard will be connected to a virtual one so that when users press the button on their keyboard they will hear a musical note. And, of course, we will create a list of instruments to select different sounds for our keyboard.

The completed application will look like this:



The completed react piano application

Its code is located in `code/03-react-piano/completed`.

Unzip the archive and `cd` to the app folder.

```
cd code/03-react-piano/completed
```

When you are there, install the dependencies and launch the app:

```
yarn && yarn start
```

It should open the app in the browser. If it doesn't, navigate to http://localhost:3000 and open it manually.

In the browser, at the center of the screen, you will see a keyboard with letter labels on each key and a `select` underneath with a default instrument.

Go ahead and try it out! You will hear the musical notes played on an acoustic grand piano.

What We're Going to Use

Besides React, we will use `AudioContext API` for generating notes sound. The `AudioContext API` itself is a bit verbose. To generate a sound, we would need to create an oscillator, set a note frequency and its duration, handle the instrument timbre. To make it more convenient, we're going to use a third-party library called Soundfont[77] that will provide us with a more flexible API.

Also, to see differences in the app components structure we will need a Chrome browser extension called `React Dev Tools`[78]. It will allow us to inspect not only the real DOM of our app but the component tree as well.

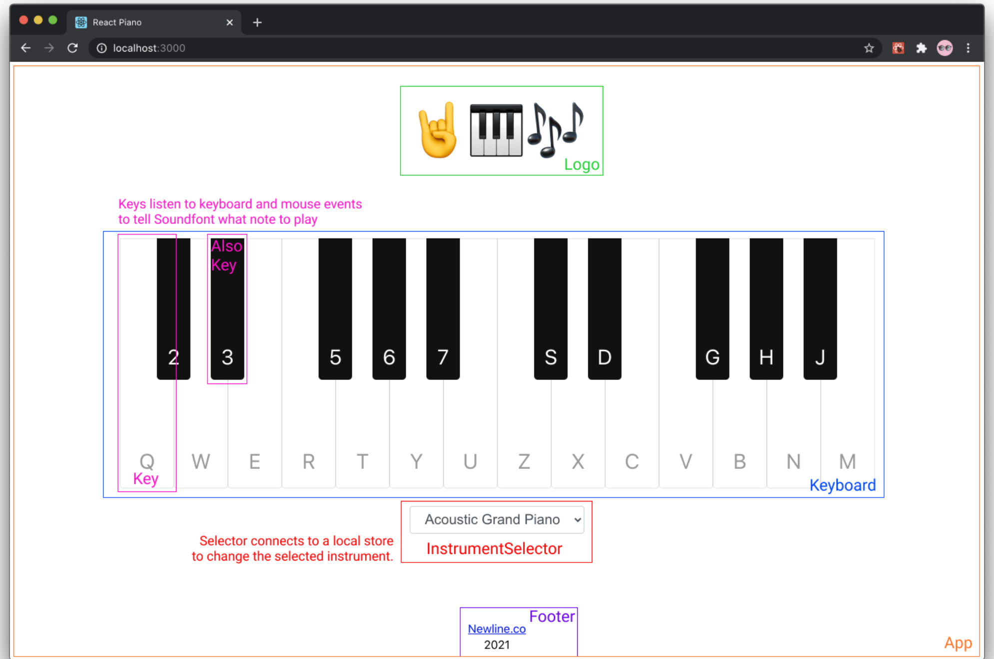
> For consistency, we use Chrome in the examples. Although, there are similar plugins for browsers other than Chrome.

So, let's try and build the keyboard!

First Steps and Basic Application Layout

First, let's inspect our future application and see what components it will be have.

[77]https://www.npmjs.com/package/soundfont-player
[78]https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=en



Application components scheme

The biggest component is the root `App` component. This is the entry point of our application.

There are 2 simple components: `Footer` and `Logo`. Those are components sometimes called "dumb". They aren't connected to anything like third-party libraries or store management. Their main goal is to render the logo and the copyright on the screen.

Also, there are more complex components like `Keyboard`, `InstrumentSelector`, and `Key`. We will wrap those components in adapters to either browser API or Soundfont. We will see why do adapters have such a name.

The structure looks good, so let's start building the app! Create another template application using `create-react-app`, like we did in previous chapters. Open your terminal and run:

```
npx create-react-app --template typescript react-piano
```

Now, `cd` to the `react-piano` folder and open the project in a text editor or IDE.

After that, we will have to clean our project directory and remove all the files and code that we will not need. Also, we will create a basic application layout and apply some global styles.

In `App.tsx`, we can safely remove the importing of `logo.svg` along with the corresponding file as we won't need it anymore. Instead, we create and import `Footer` and `Logo` components:

```
import { Footer } from "./components/Footer"
import { Logo } from "./components/Logo"
import styles from "./App.module.css"

export const App = () => {
  return (
    <div className={styles.app}>
      <Logo />
      <main className={styles.content} />
      <Footer />
    </div>
  )
}
```

We changed the default component export to the named one, so you'll need to update the `index.tsx` as well:

```
import { App } from './App';
```

Right now the code won't work because we haven't created `Footer` and `Logo`. Let's fix it!

`Footer` component

Let's start with creating the `components` directory. We will keep all the components inside it.

Each component will have a directory named with the component name. For example, the `Footer` component will be placed in the `Footer` directory. Each component will have the main `.tsx` file with its sources and the `index.ts` file for re-exports. Some components will also contain styles in the same directory.

So basic structure for a component will look like this:

```
components/
  Footer/             — component directory;
    Footer.tsx        — main source file;
    Footer.module.css — component styles;
    index.ts          — re-export file.
```

Let's try creating the `Footer` component. It will contain a signature and a current year. Create a directory inside `components` and call it `Footer`. Then, create a file `Footer.tsx` and add the following code:

```
import styles from "./Footer.module.css"

export const Footer = () => {
  const currentYear = new Date().getFullYear()

  return (
    <footer className={styles.footer}>
      <a href="https://newline.co">Newline.co</a>
      <br />
      {currentYear}
    </footer>
  )
}
```

This component imports a stylesheet, let's create a file `Footer.module.css` to hold them.

Using CSS Modules and CSS Variables

Wait for a second! Is that a CSS-file we're going to import here? Yup, this is regular old CSS. We can import stylesheets into our components, and the Create React App builder will automatically resolve them and include them in our bundle. More of that, if we use `.module.css` notation, we import those files as CSS modules.

Why use CSS modules? They give us all the perks of CSS but also isolation and close location to components that use them.

The main advantage of CSS is that it doesn't require JS-engine to render the element styles. Styled components, for example, require a browser to parse the JS code, then "translate" styles from JS into CSS, and only then apply those styles to the actual HTML element. It takes much more time than just apply styles from CSS-file.

CSS modules also generate *unique* class names for components. This makes it impossible for class names from 2 different components to collide and produce the wrong styles! Check the name for the footer element—there is no way it will collide with any other class on the page:

```
<footer class="Footer_footer__1w0lV"> == $0
  <a href="https://newline.co">Newline.co</a>
  <br>
  "2021"
</footer>
```

CSS modules create completely unique names that are assigned only to component elements and nothing else

Pretty cool! Now let's return to styling the footer.

```css
.footer {
  height: var(--footer-height);
  padding: 5px;

  text-align: center;
  line-height: 1.4;
}
```

Here we declare that `Footer` should have text alignment by a center and some 5px paddings at each side. Please, pay attention to the second line of the stylesheet: there, we declare that the component's height should be equal to a value of a *custom property*[79] (a.k.a CSS variable).

In CSS, the `var()` function searches for a custom property with a given name, in our case `--footer-height`, and if found, uses its value. So where does this value come from? We will declare it in `index.css`:

```css
:root {
  --footer-height: 60px;
  --logo-height: 8rem;
}
```

The visibility scope of our variable is `:root`. This scoping means that our variable is visible across all elements on a page. We could also define it in some selector so that it would be hidden from other elements. However, in our case, `:root` is fine.

Create `src/components/Footer/index.ts` and re-export the `Footer` component:

```ts
export * from "./Footer"
```

We will use re-exports for each component we will create. It will allow us to avoid duplications in the import paths:

[79] https://developer.mozilla.org/en-US/docs/Web/CSS/--*

```
// So we won't need to write:
import {Footer} from '../components/Footer/Footer.tsx'

// ...but instead:
import {Footer} from '../components/Footer'
```

Logo component

Now, let's create a `Logo` component. We will use emojis for our logo. A component's source code will look like this:

```
import styles from "./Logo.module.css"

export const Logo = () => {
  return (
    <h1 className={styles.logo}>
      <span role="img" aria-label="metal hand emoji">
        *Metal Hand Emoji*
      </span>
      <span role="img" aria-label="musical keyboard emoji">
        *Musical Keyboard Emoji*
      </span>
      <span role="img" aria-label="musical notes emoji">
        *Musical Notes Emoji*
      </span>
    </h1>
  )
}
```

(Unfortunately, we cannot use emojis in the example above. That's why we replaced them with text. In the sources, you will find the original code with emojis.)

We wrap every emoji in a `span` with a `role="image"` attribute. It will help screen readers to correctly parse the content of our app. Afterwards, we create a stylesheet for our `Logo` component:

```
.logo {
  font-size: 5rem;
  text-align: center;
  line-height: var(--logo-height);
  height: var(--logo-height);
  margin: 0;
  padding-top: 30px;
}
```

It will use `--logo-height`, which is declared in `index.css`.

Also, it uses `rem` for defining `font-size`[80]. This is a relative unit that refers to the value of the `font-size` property on an `html` element.

It is handy in adaptive styles to rely on that value: we won't need to update each element's `font-size` separately, but we will have to change a single `font-size` value on `html` elements instead.

Finally, re-export the component from `index.ts`:

```
export * from "./Logo"
```

Combining Components

After we have created `Footer` and `Logo` along with their styles, we're going to import and render them in `App.tsx`, so that it will look like this:

[80] https://developer.mozilla.org/en-US/docs/Web/CSS/font-size

```
import { Footer } from "./components/Footer"
import { Logo } from "./components/Logo"
import styles from "./App.module.css"

export const App = () => {
  return (
    <div className={styles.app}>
      <Logo />
      <main className={styles.content} />
      <Footer />
    </div>
  )
}
```

The last thing to do is transform `App.css` into a CSS module. To do this, rename it to `App.module.css`.

Global Styles

Now, let's finish with global styles applied to the whole project:

```
*,
*::after,
*::before {
  box-sizing: border-box;
}
```

Here we define `box-sizing: border-box` to every element on the page. It will help us calculate elements' geometry more easily.

Then, in `App.module.css` we declare that the app container should have a height of at least 100% of the screen height. Since our keyboard will be at the center of the screen, it will be convenient to do that.

```css
.app {
  min-height: 100vh;
}
```

Finally, let's ensure that the `Footer` component will be placed at the bottom of the page and the `Logo` component at the top.

```css
.content {
  --offset: calc(var(--footer-height) + var(--logo-height));
  min-height: calc(100vh - var(--offset));

  display: flex;
  justify-content: center;
  align-items: center;
}
```

Here we want all the contents of the `App` component appear in the center and the `App` itself to have a minimum height of the page excluding `Footer` and `Logo` components' heights. It ensures that the content area is at least the size of the screen.

A Bit of a Music Theory

Before we continue, let's dive into music theory.

First of all, how will we represent the musical notes in our application. Nowadays, it is considered standard to use MIDI Notes Numbers[81] for that.

A MIDI Note Number is a number that represents a note in the range from minus 1st to 9th octave. An octave is a set of 12 semitones that are different from each other by half of a tone (hence semitone).

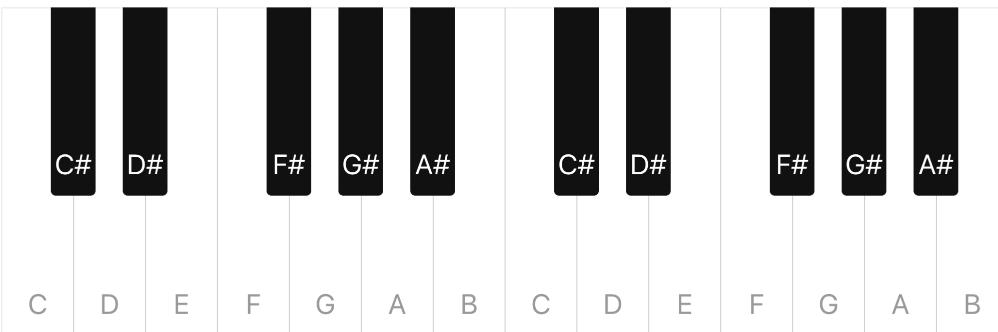
Notes in an octave start from C and go up to B like this:

[81]http://www.flutopedia.com/octave_notation.htm

```
C C# D D# E F F# G G# A A# B
```

Sharp (#) which tells us that a given note is ”sharp“. A sharp note is a half step higher than its natural note and a half step lower than the next note. So A# is a half tone higher than A and a half tone lower than B. There are also “flat” notes, but we will use only sharps for simplicity.

They would position like this on a musical keyboard: white keys are naturals, and black ones are sharps.



Notes location on a musical keyboard

Coding Music Rules

Let’s try to express all that in TypeScript. Create `src/domain/note.ts` file and add the following code:

```typescript
export type NoteType = "natural" | "flat" | "sharp"
export type NotePitch = "A" | "B" | "C" | "D" | "E" | "F" | "G"
export type OctaveIndex = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8
```

Let’s talk about the `domain` in the file path for a second.

In software, the domain[82] is a target subject of a program. This term has roots in domain-driven design[83] — the concept of how to structure applications.

[82] https://en.wikipedia.org/wiki/Domain_(software_engineering)
[83] https://en.wikipedia.org/wiki/Domain-driven_design

In our case, the domain refers to sound, note generation, note notation, and real keyboard layout.

For example, here we create a new union[84] type called `NoteType`. It will contain all the note types that we will use. Union types are useful when we want to create a set of entities to select from. In our case, `NoteType` is a set of possible notes types like natural, sharp or flat. Even though we will only use sharps, it is a good idea it clear what can be used in general.

`NotePitch` is a union type which contains all the possible note pitches from A to G. Since the order of items in a union is not important, we can order our pitches in alphabetic order to make it easier to work with later.

`OctaveIndex` is a union that contains all the octaves that can exist on a piano keyboard.

We want to create some type aliases to make the signatures of our future functions clearer.

```
export type MidiValue = number
export type PitchIndex = number
```

Here we define a `MidiValue` type which is basically a number from the Octave Notation above, and a `PitchIndex` which is also a number representing the index of a given pitch in an octave from `0` to `11`. `PitchIndex` is useful when we want to compare notes with each other to figure out which is higher, for example.

Why use these types? At first glance, it doesn't look that useful, we could just use `number` instead, and it would successfully compile. The point is in their domain meaning. When we use these types to type function arguments, they remind us what those arguments stand for.

Custom Note Type

We're going to create a custom type for our `Note` entity. This type will describe the structure of a note, what fields a note object should have, and values of what types

[84] https://www.typescriptlang.org/docs/handbook/2/everyday-types.html#union-types

those fields should have. It is a great tool for designing a software system and creating relationships between system parts or modules.

Why not use an interface here? As we discussed earlier, an interface[85] is an abstract description of some entity's behavior. It is a shared boundary across which two or more separate components of a computer system exchange information.

Although in TypeScript, an interface can fill the role of naming custom types[86], an interface still is more about defining *behavior contracts* within our code as well as contracts with code outside of our project.

So if we want to exchange information with other modules via some API, an interface will be a good way to describe that behavior. It is a powerful tool to make code components less dependent on each other and make our code reusable and less error-prone.

Types, on the other hand, are a way to describe a data structure or an entity structure. So, if we want to specify fields on an object, in reality, we describe the structure of that object. In our app, we will use both interfaces and types. There will be a point where we will use them both in the same component, where we take a closer look at the difference between them.

For now, let's go ahead and create our `Note` type:

```
export type Note = {
  midi: MidiValue
  type: NoteType

  pitch: NotePitch
  index: PitchIndex
  octave: OctaveIndex
}
```

We describe the shape of a note object which we'll use later in our code. A `Note` contains five fields, which are:

- `midi` of type `MidiValue` - a number in Octave Notation

85 https://en.wikipedia.org/wiki/Interface_(computing)
86 https://www.typescriptlang.org/docs/handbook/interfaces.html

- `type` of type `NoteType` - which note it is: natural or sharp
- `pitch` of type `NotePitch` - a literal representation of a note's pitch
- `index` of type `PitchIndex` - an index of a note in an octave
- `octave` of type `OctaveIndex` - an octave index of a given note

Some fields accept union types. For instance, the field `type` allows values of the `NoteType`. It means that we can only assign `"natural"`, `"sharp"` or `"flat"` to that field and nothing else. Otherwise, TypeScript will warn us:

> Type '"not-natural"' is not assignable to type 'NoteType'. TS2322

```
71 | export const note: Note = {
72 |   midi: 60,
73 |   type: "not-natural",
   |   ^
74 |   pitch: "C",
75 |   index: 0,
76 |   octave: 4,
```

This is very useful when we work with complex data structures and don't want to mix things up.

Application Constraints

Now, let's outline in what range we want our keyboard to contain notes. First of all, let's consider the lowest note possible to play, which is C in the first octave. It has a `MidiValue` of 24, which we will save in a `C1_MIDI_NUMBER` constant to use later.

Similarly, we create constraints for our keyboard range. The start note will be `C4_MIDI_NUMBER`, and the finish note will be `B5_MIDI_NUMBER`. Also, we're going to need to count the number of half-steps in an octave which we will save in the `SEMITONES_IN_OCTAVE` constant.

```
const C1_MIDI_NUMBER = 24
const C4_MIDI_NUMBER = 60
const B5_MIDI_NUMBER = 83

export const LOWER_NOTE = C4_MIDI_NUMBER
export const HIGHER_NOTE = B5_MIDI_NUMBER
export const SEMITONES_IN_OCTAVE = 12
```

Now, we can create some kind of map to connect literal and numerical representations of pitches of our notes.

```
export const NATURAL_PITCH_INDICES: PitchIndex[] = [
  0,
  2,
  4,
  5,
  7,
  9,
  11
]
```

`NATURAL_PITCH_INDICES` is an array which contains only indices of natural notes.

```
export const PITCHES_REGISTRY: Record<PitchIndex, NotePitch> = {
  0: "C",
  1: "C",
  2: "D",
  3: "D",
  4: "E",
  5: "F",
  6: "F",
  7: "G",
  8: "G",
  9: "A",
  10: "A",
```

```
  11: "B"
}
```

PITCHES_REGISTRY is an object with a PitchIndex as a key and NotePitch as a value.

Generics and Utility Types

Types with "arguments" like `Record<PitchIndex, NotePitch>` are called generics[87]. They allow us to create program components that can work with various types rather than a single one.

We can treat generics as "type-functions". They take type-arguments and produce a type-result. Generics allow us to describe data structures more abstractly. Let's say we want to create a type-alias for array and call it List. We can define a generic type for this:

```
// This is like a “type-function”:
// it takes an argument `TEntity`
// and returns an array of `TEntity`.
type List<TEntity> = TEntity[];

// Later we can use it like a regular type:
const numbers: List<number> = [1, 2, 3];
```

Same with other generics. Let's take a closer look at `Record`. The `Record<K, T>` type constructs[88] a type with a set of properties `K` of type `T`. In our case, it constructs a type with a set of properties `PitchIndex` of type `NotePitch`.

When to use `Record<>`? There are 2 major cases:

The first case is when you want to map the properties of a type to another type. As in our case of `Record<PitchIndex, NotePitch>`, we want to construct a type where keys can be only of type `PitchIndex` and values can be only of type `NotePitch`.

Sure, in `Record<K, T>` type `T` can be any structure. It can be another custom type as well, and it can be another `Record<K, T>`.

[87] https://www.typescriptlang.org/docs/handbook/generics.html
[88] https://www.typescriptlang.org/docs/handbook/utility-types.html#recordkeystype

The second case where you want `Record<K, T>` is when you don't know beforehand all the properties and values of a structure but know for sure their types. For example, if you want to add the values dynamically.

The `Record<K, T>` type is a so-called utility type. Typescript provides some other utility types[89] as well. Let's see what some of them do.

`Partial<T>` makes every field on the `T` type optional:

```
type MandatoryFields = {
  a: string
  b: string
}

type OptionalFields = Partial<MandatoryFields>

// It will become:
// type OptionalFields = {
//     a?: string | undefined
//     b?: string | undefined
// }
```

`Required<T>` acts opposite. It takes a type and makes every field of it mandatory:

```
type OptionalFields = {
  a?: string
  b?: string
}

type MandatoryFields = Required<OptionalFields>

// It will become:
// type MandatoryFields = {
//     a: string
//     b: string
// }
```

[89] https://www.typescriptlang.org/docs/handbook/utility-types.html

Among other utility types[90] there are direct (intrinsic) string manipulations, such as `Uppercase<>`, `Lowercase<>`, `Capitalize<>`, and `Uncapitalize<>`. They are useful when you want to perform a string-like operation on a type:

```
type Currency = 'Usd'
type NormalizedCurrency = Uppercase<Currency>
// type NormalizedCurrency = "USD"
```

Later we will create our own generic utility type called `Optional<>`!

Generating Notes

We're almost there! The only thing left to cover is a function producing a `Note` object from a given `MidiValue`. So let's create it!

```
export function fromMidi(midi: MidiValue): Note {
  const pianoRange = midi - C1_MIDI_NUMBER
  const octave = (Math.floor(pianoRange / SEMITONES_IN_OCTAVE) +
    1) as OctaveIndex

  const index = pianoRange % SEMITONES_IN_OCTAVE
  const pitch = PITCHES_REGISTRY[index]

  const isSharp = !NATURAL_PITCH_INDICES.includes(index)
  const type = isSharp ? "sharp" : "natural"

  return { octave, pitch, index, type, midi }
}
```

Here we take a `MidiValue` as an argument and determine in which `octave` this note is. After that, we figure out what `index` this note has inside its octave and what `pitch` this note is. Finally, we determine which `type` this note is and return a created note object.

[90] https://www.typescriptlang.org/docs/handbook/utility-types.html

Why explicitly define the return type? Indeed, the TS compiler can infer the type and provide us with it later itself. Why bother?

The point is that adding type annotations (and especially return types) can save the compiler much work and make the compilation process of our program much faster[91]. Another advantage is that we make it impossible to unexpectedly return another type when we define a return type on a function. (Everyone makes typos.)

```
type ExpectedReturnType = {
  fieldName: string,
};

function exampleA() {
  return { fieldNme: 'value' }
}

function exampleB(): ExpectedReturnType {
  return { fieldNme: 'value' }
  // Here, TypeScript will error because of the typo:
  // Type '{ fieldNme: string; }'
  // is not assignable to type 'ExpectedReturnType'.
}
```

Okay, return to `fromMidi` function. It will not only help us to convert numbers to notes on our keyboard but also to create an initial set of notes.

Let's make a little helper function to generate that set.

[91] https://github.com/microsoft/TypeScript/wiki/Performance#using-type-annotations

```
type NotesGeneratorSettings = {
  fromNote?: MidiValue
  toNote?: MidiValue
}

export function generateNotes({
  fromNote = LOWER_NOTE,
  toNote = HIGHER_NOTE
}: NotesGeneratorSettings = {}): Note[] {
  return Array(toNote - fromNote + 1)
    .fill(0)
    .map((_, index: number) => fromMidi(fromNote + index))
}

export const notes = generateNotes()
```

Here we create a `generateNotes()` function which takes a settings object of type `NotesGeneratorSettings`. It describes which settings we can use in our function to generate notes. A question mark (`?`) at the field's name means that this field is optional and can be omitted when creating an instance of an object.

It is better to use a settings object than optional function arguments since arguments rely on their order, and object keys don't. So, we destructure a given object with settings to access the `fromNote` and `toNote` fields of that object. If none is present we use an empty object as one with settings.

> We should be aware of possibly failing destructuring in runtime though. The TypeScript checker will throw an error if we try to pass not an object as the argument but it won't help after the compilation.

Inside, we use default values for those fields, and if they are not specified, we set them to `LOWER_NOTE` and `HIGHER_NOTE`, respectively. So when we call `generateNotes()` with no arguments, it will generate a set of notes in a range from `LOWER_NOTE` to `HIGHER_NOTE`. And that is exactly what we want for our future keyboard!

Inside `generateNotes()`, we create an array and fill it with notes from `fromNote` to `toNote`.

Third Party API and Browser API

We're going to use `Audio API` and a third-party API to create a sound. So let's talk a bit about the integration of those APIs.

Web Audio API

For starters, let's figure out what's required to create a sound in a browser in the first place. Modern web browsers support `Audio API`[92].

It uses an `AudioContext` to handle audio operations such as playing musical tracks, creating oscillators, etc. This `AudioContext`[93] has nothing to do with `React.Context` that we saw earlier. They only have similar names, but `AudioContext` is an interface that provides access to the browser's audio API.

We can access `AudioContext` via `window.AudioContext`. The problem is that not every browser has this property. The majority of modern browsers do, but we cannot rely on the assumption that a user's browser has it.

Let's ensure that the browser supports `AudioContext`. Create a helper function that will check if our browser supports `AudioContext`.

Create `src/domain/audio.ts` and add the following code:

```
import { Optional } from "./types"

export function accessContext(): Optional<AudioContextType> {
  return window.AudioContext || window.webkitAudioContext || null
}
```

Here, we create a function `accessContext()`, which takes no arguments and returns `Optional<AudioContextType>`. At this point, TypeScript will show two errors:

- It will say that the `Optional` import is impossible;
- And it will say that the `AudioContextType` type is unknown.

[92] https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API
[93] https://developer.mozilla.org/en-US/docs/Web/API/AudioContext

We will fix these one at a time. Let's start with `Optional`. The `Optional` type is a utility type. Create a file called `types.ts` beside and add the following code:

```
export type Optional<TEntity> = TEntity | null
```

> We use a slightly more verbose name `TEntity` instead of just `T` for the *type argument*, because it is more readable.

The `Optional` type is genetic, representing a union with a given type `TEntity` or a `null`. Basically, we're building an "assumption" type, and will use it when we're unsure if some entity is defined as `TEntity` type or is `null`.

This type is useful when you want to ensure that you cover all the possible cases when an entity possibly doesn't exist. In our case, `Optional` tells us that `accessContext()` returns either `AudioContextType` or `null`.

Next, let's figure out what `AudioContextType` is. For that, open `react-app-env.d.ts` and add the following code:

```
/// <reference types="react-scripts" />

type AudioContextType = typeof AudioContext

interface Window extends Window {
  webkitAudioContext: AudioContextType
}
```

Here, we see a triple-slash directive[94] with a reference to `react-scripts` package's types. We discussed these directives in the previous chapters.

Also, in this file, we create a type called `AudioContextType` which is equal to `typeof AudioContext`. This may seem a bit confusing, but technically it means that our custom type `AudioContextType` is literally a type of `window.AudioContext`. `AudioContext` is not a type *per se*, but a constructor function. To make TypeScript understand what type we want to declare, we explicitly define it as `typeof AudioContext`.

[94] https://www.typescriptlang.org/docs/handbook/triple-slash-directives.html

When is `typeof` also useful? Well, it is a tricky question. We may use it in a function to change its behavior based on a type of argument. It is considered bad practice because it leads to tightly coupled code. However, there is a case when we can use the `typeof` operator except for defining custom types. We can use it in function overloading.

Function overloading allows to define functions of the same name with different implementations:

```
function concat(a: string, b: string): string;
function concat(a: string[], b: string[]): string;

function concat(a: any, b: any): string {
  if (typeof a === 'string' && typeof b === 'string') {
    return a + b;
  }

  return a.join(',') + b.join(',')
}
```

In the `concat` function, we declare `2` possible argument sets. Based on argument types, we change the function implementation. We call this tricky because in other languages, like C#, there is a way to create multiple implementations completely separately. But since TypeScript is constrained by JavaScript runtime, we can't do that.

So, the `typeof` operator in overloading is sort of a workaround, but still, it is better to avoid using it in the code that will go to runtime. Okay, let's return to our `react-app-env.d.ts`.

Below `AudioContextType`, we can see an extension for the `Window` interface, which includes the field `webkitAudioContext` with a type of `AudioContextType`. This is required for now because TypeScript by default doesn't include[95] some vendor properties and methods on `window`.

We extend the standard window interface to gain access to this field because in some browsers `AudioContext` is available through the `webkitAudioContext` property.

[95] https://github.com/microsoft/TypeScript/issues/31686

We check if the browser supports `AudioContext` or `webkitAudioContext`. If the browser doesn't support either of them, we return `null`. It means that we cannot access `Audio API`.

Soundfont

Next, it is time to introduce the third-party API we're going to use — Soundfont[96]. It is a framework-agnostic loader and player which has a pack of pre-rendered sounds of many instruments. It also comes with typings for integration with TypeScript projects!

We prefer Soundfont over MIDI.js[97] because Soundfont satisfies all of our requirements and weighs less.

Let's start integrating Soundfont with our project. First, install it with npm:

```
yarn add soundfont-player
```

After the package is installed, create a file in the `domain` called `sound.ts` and add the following code:

```
import { InstrumentName } from "soundfont-player"

export const DEFAULT_INSTRUMENT: InstrumentName =
  "acoustic_grand_piano"
```

For now, we are good with exporting a `DEFAULT_INSTRUMENT` constant of type `InstrumentName`, which comes with the `soundfont-player` package. One of the coolest things about integrating third-party APIs which have TypeScript declarations is that we can use our IDE's autocomplete to scroll through possible options for union types. Here we can select from multiple different instruments which are listed in the `InstrumentName` union.

[96] https://www.npmjs.com/package/soundfont-player
[97] https://github.com/mudcube/MIDI.js

Patterns

So far, we have been working with our application code and third-party APIs separately. Now we'll connect them.

Sometimes connect software components can be cumbersome. The good news is that it is a typical programming problem, and typical programming problems are solved by programming *patterns*.

Adapter or Provider Pattern

An Adapter[98] pattern (sometimes called a Provider pattern) is a software design pattern that allows the interface of an existing entity (class, service, etc) to be used as another interface. It *adapts*[99] (or *provides*) a third-party API for us and makes it usable in our application code.

It is easier to understand the adapter concept with a small example. Let's say we have thermometer app that uses Celcius as a unit. We have a third-party function that returns tempreture in Fahrenheits:

```
type ThirdPartyData = {
  temperature: DegreeFahrenheit
}
```

For this function to work we want a converter from Fahrenheit to Celsius:

```
function fahrenheitToCelsius(value: DegreeFahrenheit): DegreeCelsius {
  return (value - 32) * 5 / 9
}
```

The `fahrenheitToCelsius` function is an *adapter*. It changes the external function result in such a way that it becomes compatible with our code.

[98] https://en.wikipedia.org/wiki/Adapter_pattern
[99] https://github.com/kamranahmedse/design-patterns-for-humans#-adapter

React-Specific Patterns

In our case, we want to use Provider patterns to make Soundfont's functionality accessible to our application. Also, it will be useful to connect `Audio API` to our code.

Using React, we can implement Provider patterns using multiple techniques, such as *Render Props* and *Higher-Order Components*. Those are also called patterns, so we will call them React-patterns to distinguish these from the patterns above.

Later, we will cover all those React-patterns, but before we begin, let's create a new application screen with a `Keyboard` component to be able to play notes.

Main App Screen

In this section, we will create the main app screen with a `Keyboard` component in it. Also, we will cover the case when a user's browser doesn't support `Audio API` and create a component with a message about it.

Our main app screen will be in the `Main` component.

```
import { NoAudioMessage } from "../NoAudioMessage";
import { useAudioContext } from "../AudioContextProvider";

const Keyboard = () => <>Keyboard</>;

export const Main = () => {
  const AudioContext = useAudioContext();
  return !!AudioContext ? <Keyboard /> : <NoAudioMessage />;
};
```

Then, re-export the `Main` component from `index.ts`:

```
export * from "./Main"
```

When used, it checks whether the browser supports `Audio API` or not and decides which component to render: `Keyboard` or `NoAudioMessage`. We will look at them a little later. For now, let's focus on a custom hook[100] `useAudioContext()`.

[100] https://reactjs.org/docs/hooks-intro.html

Custom Hook for Accessing Audio

Intentionally, hooks in React let us use state and other features without writing a class. Writing hooks has rules[101] and limitations. For example, all hooks' names should start with a `use*` prefix. It allows the linter to check if a hook's source code satisfies all the limitations, which are:

- We can call hooks only at the top level of our components and never conditionally.
- We can call hooks only inside functional components.

In our case, we create a hook called `useAudioContext()`, which encapsulates access to `AudioContext`.

```
import { useRef } from "react"
import { Optional } from "../../domain/types"
import { accessContext } from "../../domain/audio"

export function useAudioContext(): Optional<AudioContextType> {
  const AudioCtx = useRef(accessContext())
  return AudioCtx.current
}
```

Here, we use the `useRef()` hook[102] to "remember" the value that our `accessContext()` function is going to return. We can use the `useRef` hook with any sort of data, not necessarily with elements. Also, we may not provide the type for `useRef` because our `accessContext` has an explicitly defined return type, so it neither will affect performance nor will make a place for any mistakes.

As a result from our custom hook we return `Optional<AudioContextType>`. Again, we want to provide either an `AudioContextType` or `null` to be able to build our UI depending on that later on.

So, when a `Main` component calls `useAudioContext()`, it gets an `AudioContext` if a browser supports it and renders a `Keyboard` component, or it gets `null` and renders a `NoAudioMessage` component otherwise. Now it's time to look at both of them.

[101] https://reactjs.org/docs/hooks-rules.html
[102] https://reactjs.org/docs/hooks-reference.html#useref

Handling Missing Audio Context

Let's look at the `NoAudioMessage` component first. It is basically a `div` with some text in it. It doesn't do much, and it only renders a message for a user.

Create a directory called `NoAudioMessage` inside `components`, add the `NoAudioMessage.tsx` file, and add the following code:

```
export const NoAudioMessage = () => {
  return (
    <div>
      <p>Sorry, it's not gonna work :-(</p>
      <p>
        Seems like your browser doesn't support <code>Audio API</code>
        .
      </p>
    </div>
  )
}
```

Re-export the component from `index.ts`:

```
export * from "./NoAudioMessage"
```

Creating a Keyboard

In this section we will implement the keyboard. We'll start with the component that will render the individual keys.

Single Key on a Keyboard

In this component, we will need to compose different class names on the element. To make it easier, let's install the `clsx` package[103].

[103] https://www.npmjs.com/package/clsx

```
yarn add clsx
```

After it's done create a folder `src/components/Key`. First we define the styles.

Styles for the `Key`

Our keys will be based on the regular `button` element. To make it look silimar in all the browsers, we want to reset the default button styles. Open `src/index.css` and add the following styles:

```css
button {
  border: none;
  border-radius: 0;

  margin: 0;
  padding: 0;
  width: auto;
  background: none;
  appearance: none;

  color: inherit;
  font: inherit;
  line-height: normal;
  cursor: pointer;
}
```

Here we made the button look like a text element. We added those styles to the `src/index.css`, because we want them to affect the whole application.

Create `src/components/Key/Key.module.css` and define the `.key` class there:

```
.key {
  position: relative;
  font-size: var(--font-size);
  border-radius: 0 0 var(--radius) var(--radius);
  text-transform: uppercase;
  user-select: none;
}
```

Define the variables for the `.key` class:

```
.key {
  --radius: 2px;
  --font-size: 0.6rem;
  --white-key-width: 20px;
  --white-key-height: calc(var(--white-key-width) * 4.57);
  --white-key-padding: calc(var(--white-key-height) / 1.28);
  --black-key-width: calc(var(--white-key-width) / 1.6);
  --black-key-height: calc(var(--white-key-height) / 1.77);
  --black-key-padding: calc(var(--black-key-height) / 1.5);
}
```

Our keys will have `.natural`, `.sharp` and `.flat` modifiers:

```
.natural {
  width: var(--white-key-width);
  height: var(--white-key-height);
  padding-top: var(--white-key-padding);
  border: 1px solid rgba(0, 0, 0, 0.1);
  color: rgba(0, 0, 0, 0.4);
  margin-right: -1px;
  z-index: 1;
}

.sharp,
.flat {
```

```
  width: var(--black-key-width);
  height: var(--black-key-height);
  padding-top: var(--black-key-padding);
  background-color: #111;
  color: white;
  margin: 0 calc(-0.5 * calc(var(--black-key-width)));
  z-index: 2;
}
```

Add the styles for the pressed keys:

```
.natural:active,
.natural.is-pressed {
  background-color: rgba(0, 0, 0, 0.1);
}

.sharp:active,
.sharp.is-pressed,
.flat:active,
.flat.is-pressed {
  background-color: #555;
}
```

And for the disabled keys:

```
.key:disabled {
  background-color: none;
  cursor: wait;
}

.natural:disabled {
  color: rgba(0, 0, 0, 0.2);
  background-color: white;
}

```

```
.sharp:disabled,
.flat:disabled {
  color: rgba(255, 255, 255, 0.4);
  background-color: #111;
}
```

Define the `@media` queries for differenc screen sizes. We'll start with the smallest screen, this is how it should look on mobile phones:

```
@media (min-width: 380px) {
  .key {
    --white-key-width: 25px;
    --radius: 5px;
    --font-size: 0.8rem;
  }
}
```

Define the bigger version:

```
@media (min-width: 540px) {
  .key {
    --white-key-width: 35px;
    --font-size: 1rem;
  }
}
```

Some versions for tablets:

```css
@media (min-width: 720px) {
  .key {
    --white-key-width: 45px;
    --font-size: 1.2rem;
  }
}

@media (min-width: 960px) {
  .key {
    --white-key-width: 65px;
    --font-size: 1.5rem;
  }
}
```

And the biggest version for the desktop:

```css
@media (min-width: 1120px) {
  .key {
    --white-key-width: 75px;
    --font-size: 1.8rem;
  }
}
```

Define the `Key` component

Create `src/components/Key/Key.tsx` with the following imports:

```tsx
import { FunctionComponent } from "react"
import clsx from "clsx"
import { NoteType } from "../../domain/note"
import styles from "./Key.module.css"
```

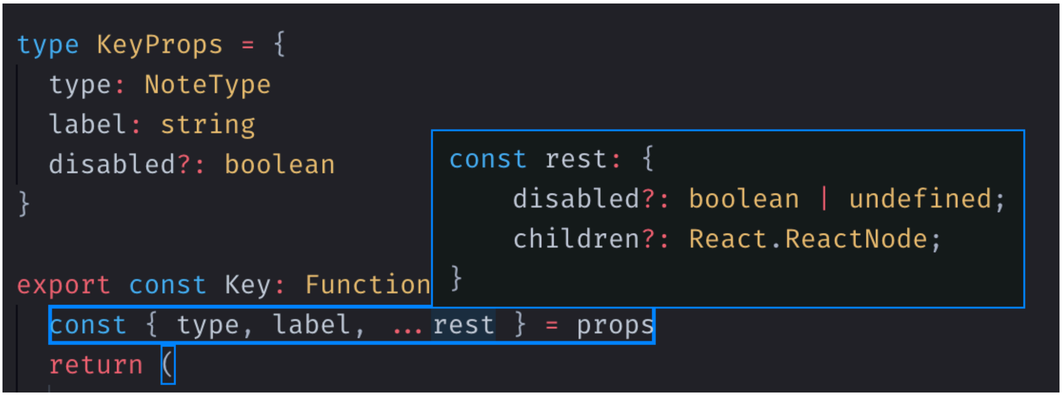
Define the `Key` component:

```
type KeyProps = {
  type: NoteType
  label: string
  disabled?: boolean
}

export const Key: FunctionComponent<KeyProps> = (props) => {
  const { type, label, ...rest } = props
  return (
    <button
      className={clsx(styles.key, styles[type])}
      type="button"
      {...rest}
    >
      {label}
    </button>
  )
}
```

Here we defined a component that accepts the props of type `KeyProps`:

- `type`, a `NoteType` — will be used to define the styles of a key
- `label`, a `string` — a letter that will be placed as a label of a key
- `disabled`, an optional `boolean` — if `true` it will disable the key from being pressed

The rest operator (`...rest`) in TypeScript keeps all the information about the types of all fields in the `rest` object. The `disabled` field is inferred from the `KeyProps` type and the `children` field is inferred from the `FunctionComponent` type.



Types of fields on the rest object

> If we wanted to explicitly and strictly specify that this component shouldn't accept `children` as a prop we wouldn't use the `FunctionComponent` type. This type implicitly adds the `children` prop to the props passed as a type argument.

Finally, re-export the component from the `index.ts` file:

```
export * from "./Key"
```

Create the Keyboard component

Create `src/components/Keyboard/Keyboard.module.css` to hold the styles for the keyboard:

```
.keyboard {
  display: flex;
}
```

Define the `Keyboard` component, create a file `src/components/Keyboard/Keyboard.tsx` with the following code:

```
import { selectKey } from "../../domain/keyboard"
import { notes } from "../../domain/note"
import { Key } from "../Key"
import styles from "./Keyboard.module.css"

export const Keyboard = () => {
  return (
    <div className={styles.keyboard}>
      {notes.map(({ midi, type, index, octave }) => {
        const label = selectKey(octave, index)
        return <Key key={midi} type={type} label={label} />
      })}
    </div>
  )
}
```

Look how we map over the `notes` array. It contains the notes from `C4` to `B5`. We destructure each note into `midi`, `type`, `index`, and `octave` fields. For each note, we render a `Key` component.

There is a function we haven't seen yet, called `selectKey()`. It is a function that selects a letter label for a given key. Let's inspect its source code.

```
import { OctaveIndex, PitchIndex } from "./note"

export type Key = string
export type Keys = Key[]

export const TOP_ROW: Keys = Array.from("q2w3er5t6y7u")
export const BOTTOM_ROW: Keys = Array.from("zsxdcvgbhnjm")
export const CHANGE_ROW_AT: OctaveIndex = 5

export function selectKey(
  octave: OctaveIndex,
  index: PitchIndex
): Key {
```

```
  const keysRow = octave < CHANGE_ROW_AT ? TOP_ROW : BOTTOM_ROW
  return keysRow[index]
}
```

In `keyboard.ts`, we create two custom types:

- `Key`, a type-alias for representing letter key labels
- `Keys`, an array of those labels

Then, we create two arrays of letters that will label our keys. If those letters are pressed on a real keyboard, we will play the sound of a key with the corresponding label. We use `Array.from()`[104] to create an array of characters from a string.

`selectKey()` is a function that takes an octave index that we choose a key by and a pitch index to select from the chosen octave. Thus, we map a letter to our key label.

Create the `src/components/Keyboard/index.ts` and re-export everything from the `./Keyboard` module:

```
export * from "./Keyboard"
```

Update the `Main` component

Go to `src/components/Main/Main.tsx` and add use the real `Keyboard` component there:

[104] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/from

```
import { Keyboard } from "../Keyboard"
import { NoAudioMessage } from "../NoAudioMessage"
import { useAudioContext } from "../AudioContextProvider"

export const Main = () => {
  const AudioContext = useAudioContext()
  return !!AudioContext ? <Keyboard /> : <NoAudioMessage />
}
```

Adapter Hook

In this section we'll add the sounds to our app.

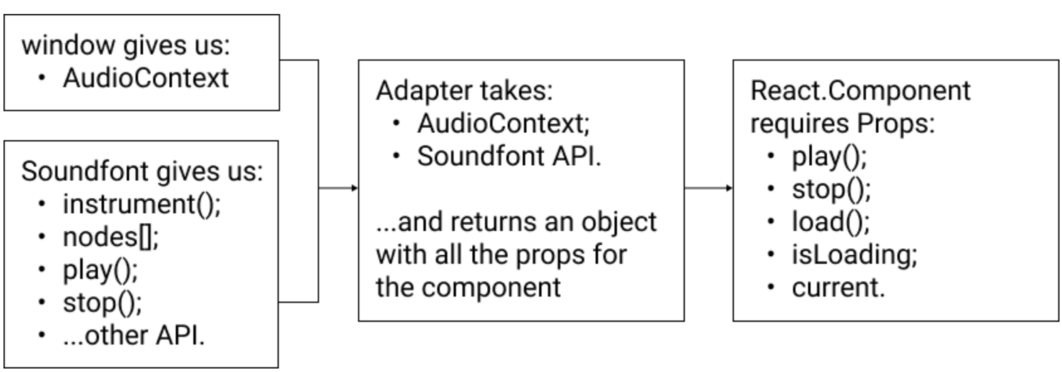
Add a custom `SoundfontType` type to `react-app-env.d.ts`:

```
type SoundfontType = typeof Soundfont
```

This type is going to be useful when we create an adapter for Soundfont.

Soundfont Adapter

The adapter should take what Soundfont provides through the public API, take what `window` gives us, and *adapt* all of that for our usage.



How Soundfont adapter should work

We'll implement the adapter multiple times. The first version of adapter will be a hook. And then we will use React-Patterns, such as *HOCs* and *Render-Props*.

Create a `src/adapters/Soundfont/useSoundfont.ts` and define the imports:

```
import { useState, useRef } from "react"
import Soundfont, { InstrumentName, Player } from "soundfont-player"
import { MidiValue } from "../../domain/note"
import { Optional } from "../../domain/types"
import {
  AudioNodesRegistry,
  DEFAULT_INSTRUMENT
} from "../../domain/sound"
```

Specify what the input and output types of the adapter:

```
type Settings = {
  AudioContext: AudioContextType
}

interface Adapted {
  loading: boolean
  current: Optional<InstrumentName>

  load(instrument?: InstrumentName): Promise<void>
  play(note: MidiValue): Promise<void>
  stop(note: MidiValue): Promise<void>
}
```

Here, the `Settings` type describes what does the `useSoundfont()` adapter require as arguments. The `Adapted` interface specifies what kind of object we're going to return from this adapter.

The `Settings` type describes a "shape" of the configuration object.

The `Adapted` is an interface that requires the `loading` flag, the `current` instrument, and the `load()`, `play()` and `stop()` methods on any object that implements it.

The `loading` field is a `boolean` flag that is set to `true` while Soundfont loads the instrument sound set. We will use it to disable `Keyboard` while loading is happening. The `current` field contains the current instrument.

Functions `load()`, `play()` and `stop()` are functions that handle loading the instrument sound set, starting playing a note, and finishing playing a note, respectively. They are all asynchronous since the `Audio API` is asynchronous by itself.

Async functions in TypeScript return the `Promise<TResult>` generic type. This way we know that this function returns a `Promise` of some value, but not the value type.

Define the adapter hook:

```
export function useSoundfont({ AudioContext }: Settings): Adapted {
  let activeNodes: AudioNodesRegistry = {}
  const [current, setCurrent] = useState<Optional<InstrumentName>>(
    null
  )
  const [loading, setLoading] = useState<boolean>(false)
  const [player, setPlayer] = useState<Optional<Player>>(null)
  const audio = useRef(new AudioContext())
  // ...
}
```

Here, `activeNodes` is an object with `AudioNode`[105] items. Those are general interfaces for handling sound operations. Soundfont uses them to store the state of played notes. The type of this state is `AudioNodesRegistry`, it is defined in `src/domain/sound.ts`.

```
import { InstrumentName, Player } from "soundfont-player";
import { MidiValue } from "./note";
import { Optional } from "./types";

export type AudioNodesRegistry = Record<MidiValue, Optional<Player>>;
```

`AudioNodesRegistry` is a `Record` with key of type `MidiValue` and value of type `Player`. The `Player` type is provided by Soundfont, it handles the musical operations for us.

[105] https://developer.mozilla.org/ru/docs/Web/API/AudioNode

Unlike other local variables, `activeNodes` is not part of the local state. That is because we don't want our component to re-render every time audio nodes change their state. We want to avoid extra repaints and avoid situations where the `.stop()` method is being called on a non-existent node or a node with an invalid audio state. So, we update this registry directly using a local variable, not using the state.

The field `current` has type `Optional<InstrumentName>` and holds the instrument playing now. Initially we set it to `null`.

The `loading` field indicates whether an instrument is an instrument playing now or not.

The `player` field holds a Soundfont `Player` instance. We use it to perform musical operations.

The `audio` is an instance of `AudioContext`. We use `useRef()` hook[106] to keep a reference to an instance of an `AudioContext` that we create when the component mounts. To access this instance, we use the `audio.current` property.

Loading Sound Set

Implement the `load()` method, it will load the instrument sound set. Add the following code in the `useSoundfont` hook:

```
async function load(
  instrument: InstrumentName = DEFAULT_INSTRUMENT
) {
  setLoading(true)
  const player = await Soundfont.instrument(
    audio.current,
    instrument
  )

  setLoading(false)
  setCurrent(instrument)
  setPlayer(player)
}
```

[106] https://reactjs.org/docs/hooks-reference.html#useref

We mark this function `async`, because of the async `instrument()` method from Soundfont.

We set the `loading` state to `true` to indicate that the sound set is loading. Then, we call the `await Soundfont.instrument()` method and keep the returned result to a `player` local state. Also, we save a given `instrument` as `current`, and when everything is done, mark `loading` as `false`.

Implement the `resume()` method:

```
async function resume() {
  return audio.current.state === "suspended"
    ? await audio.current.resume()
    : Promise.resolve()
}
```

It checks what state `audio` is in right now. If it is `suspended`[107], this means that `AudioContext` is halting audio hardware access and reducing CPU/battery usage in the process. To continue we call the `resume()` method on it.

To handle the case when the state of `audio` wasn't `suspended`, we use `Promise.resolve()`[108]. This method returns a `Promise` object that is resolved with a given value. We don't need any, so we don't pass it as an argument.

Implement the `play()` and `stop()` methods:

```
  async function play(note: MidiValue) {
    await resume()
    if (!player) return

    const node = player.play(note.toString())
    activeNodes = { ...activeNodes, [note]: node }
  }

  async function stop(note: MidiValue) {
    await resume()
```

[107] https://developer.mozilla.org/en-US/docs/Web/API/AudioContext/suspend
[108] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/resolve

```
    if (!activeNodes[note]) return

    activeNodes[note]!.stop()
    activeNodes = { ...activeNodes, [note]: null }
  }
```

The exclamation mark in the `stop()` function is a non-null assertion operator[109]. Using it we declare that we are totally sure that `activeNodes[note]` is not `null`. We can do that because we checked it on a previous line.

In the `play()` function, we take a `MidiValue` as an argument to know what note to play. Also, we check if there is no `player` yet, in which case we don't do anything. Otherwise, we create an active `audioNode` by calling `player.play()` method.

We convert the `note` to string type because the player's `play()` method only accepts strings. We can verify that by checking the Soundfont types. The `play()` method references the `start()` method, which takes a string as the first argument:

```
export declare type Player = {
  start: (
    name: string,
    when?: number,
    options?: Partial<{ /* ... */ }>
  ) => Player;

  play: Player["start"];
  // ...
};
```

We save the result node into our `activeNodes` registry. These `activeNodes` keep track of playing notes and allow to `stop()` them.

We return `loading` state, `current` instrument, and 3 methods for controlling the player `load()`, `play()`, `stop()`:

[109] https://www.typescriptlang.org/docs/handbook/release-notes/typescript-2-0.html#non-null-assertion-operator

```
return {
  loading,
  current,
  load,
  play,
  stop
}
```

Re-export the hook from `adapters/Soundfont/index.ts`:

```
export * from "./useSoundfont"
```

Congratulations, we created our first sound provider!

Connecting to a Keyboard

In the `Key` component, we will use `onDown()` and `onUp()` methods to handle `keypress` events. Let's open the `Key.tsx` and create a type alias `PressCallback` which is a function called on press event. We will use this callback in the new `onDown()` and `onUp()` methods in the `KeyProps` type:

```
type PressCallback = () => void
type KeyProps = {
  type: NoteType
  label: string
  disabled?: boolean

  onUp: PressCallback
  onDown: PressCallback
}
```

These methods exist now in `KeyProps`, and we can use them in `onMouseDown()` and `onMouseUp()` props for the `button` element.

```
export const Key: FunctionComponent<KeyProps> = ({
  type,
  label,
  onDown,
  onUp,
  ...rest
}) => {
  return (
    <button
      className={clsx(styles.key, styles[type])}
      onMouseDown={onDown}
      onMouseUp={onUp}
      type="button"
      {...rest}
    >
      {label}
    </button>
  )
}
```

Open `src/components/Keyboard/Keyboard.tsx` and update the component code:

```
import { FunctionComponent } from "react"
  // ...
import { notes, MidiValue } from "../../domain/note"
  // ...
export type KeyboardProps = {
  loading: boolean
  play: (note: MidiValue) => Promise<void>
  stop: (note: MidiValue) => Promise<void>
}

export const Keyboard: FunctionComponent<KeyboardProps> = ({
  loading,
  stop,
  play
```

```
}) => (
  <div className={styles.keyboard}>
    {notes.map(({ midi, type, index, octave }) => {
      const label = selectKey(octave, index)
      return (
        <Key
          key={midi}
          type={type}
          label={label}
          disabled={loading}
          onDown={() => play(midi)}
          onUp={() => stop(midi)}
        />
      )
    })}
  </div>
)
```

The `Keyboard` now has props that will consume `loading`, `play()`, and `stop()` provided by the adapter. We use the `loading` flag to disable the keys to forbid the user from pressing them while the keyboard is not ready.

The `play()` and `stop()` methods are typed with (note: MidiValue) => Promise<void> signature. What is Promise<void>? By using Promise<>, we can declare an `async` function. Since every async function returns a promise object, TypeScript uses this signature as well.

The `void` symbol means that this function doesn’t return any value. In some cases, functions that don’t return anything are called *procedures*. For example:

```
// Returns a number, so its return-type is a number.
function add(a: number, b: number): number {
  return a + b;
}

const sum = add(1, 2);
// It returns 3, so sum === 3.

function greet(name: string): void {
  console.log(`Hello ${name}!`);
}

const result = greet('Alex');
// It doesn't return anything, so result === undefined
```

Now we only have to actually connect our `Keyboard` to the Soundfont provider, and we're there!

Create another file inside the `Keyboard` directory called `WithInstrument.tsx` and add the following code:

```
import { useAudioContext } from "../AudioContextProvider"
import { useSoundfont } from "../../adapters/Soundfont"
import { useMount } from "../../utils/useMount"
import { Keyboard } from "../Keyboard"

export const KeyboardWithInstrument = () => {
  const AudioContext = useAudioContext()!
  const { loading, play, stop, load } = useSoundfont({ AudioContext })

  useMount(() => load())

  return <Keyboard loading={loading} play={play} stop={stop} />
}
```

Re-export it:

```
export * from "./WithInstrument"
```

In the `KeyboardWithInstrument` component, we use our custom hook to access required methods and flags. Then, when mounted, we provide those props to our `Keyboard`. We use an exclamation mark to tell the type checker that we are sure that `useAudioContext()` doesn't return `null`. We know that this component will appear only if the browser supports Audio API because we tested it earlier.

We can also see there a hook called `useMount()`. It allows us to run some code right after a component is mounted into the DOM. Let's write it as well. Create a file `src/utils/useMount/useMount.ts` and add the following code:

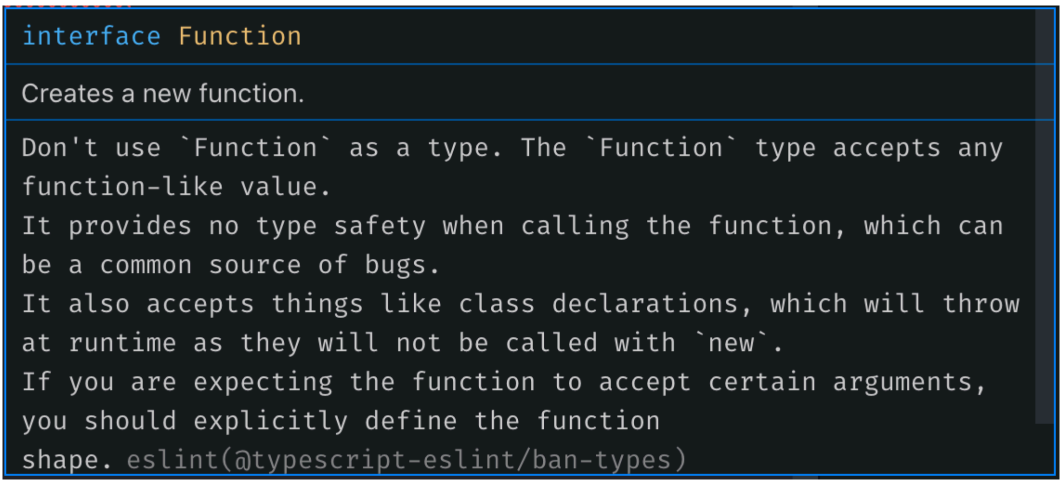
```
import { EffectCallback, useEffect } from "react"

const useEffectOnce = (effect: EffectCallback) => {
  // eslint-disable-next-line react-hooks/exhaustive-deps
  useEffect(effect, [])
}

type Effect = (...args: unknown[]) => void

export const useMount = (fn: Effect) => {
  useEffectOnce(() => {
    fn()
  })
}
```

First, we create a `useEffectOnce()` hook to encapsulate the `useEffect()` call with an empty dependency array. This array tells React what variables to observe. If either of the variables in that array changes, React will re-run the effect. In our case, we only need to run the effect once when the component appears in the DOM. That's why we set it to be empty.

Then, `useMount()` hook is a wrapper over `useEffectOnce()`. It takes an `Effect` function and runs it through the `useEffectOnce()` hook.

Why not use the global `Function` type instead of creating a custom `Effect` type? TypeScript by itself doesn't forbid us to use the global `Function` type. However, there is a catch. `Function` accepts any function-like value. So, for example, it accepts class declarations that can throw an error if called incorrectly.

We can secure ourselves by using the `ban-types` rule in the ESLint configuration. It will error if we use insecure types in declarations:



ESLint error when using global Function type

We don't pass just `fn` inside `useEffectOnce()` to avoid mistakes with return values. By default, the value returned from an effect in `useEffect` is interpreted as a clean-up function. We don't want this for `fn` so we wrap it in another function that doesn't return anything.

Re-export the hook from `src/utils/useMount/index.ts`:

```
export * from "./useMount"
```

Update the `Main` component to include the connected `KeyboardWithInstrument`. Check if `AudioContext` exists by converting it to a boolean with the double negation `!!` operator. If so, return the keyboard. Otherwise, return the fallback message.

```
import { KeyboardWithInstrument } from "../Keyboard"
import { NoAudioMessage } from "../NoAudioMessage"
import { useAudioContext } from "../AudioContextProvider"

export const Main = () => {
  const AudioContext = useAudioContext()
  return !!AudioContext ? (
    <KeyboardWithInstrument />
  ) : (
    <NoAudioMessage />
  )
}
```

Mapping Real Keys to Virtual

Right now, our `Keyboard` can play sounds when pressed by a mouse click. However, we want it to play notes when a user presses corresponding keys on their real keyboard. To do that, we want to map real keys with virtual ones so that when a user presses a key, our application would know what to do and which note to play.

We create a component that will implement another pattern called *Observer*. Its main idea is to allow us to *subscribe* to some events and handle them as we want to. In our case, we want to subscribe to `keyPress` events.

Let's start again with designing an API. Create a new file `src/components/PressObserver/usePre` and add the following code:

```
import { useEffect, useState } from "react"
import { Key as KeyLabel } from "../../domain/keyboard"

type IsPressed = boolean
type EventCode = string
type CallbackFunction = () => void

type Settings = {
  watchKey: KeyLabel
  onStartPress: CallbackFunction
  onFinishPress: CallbackFunction
}
```

`IsPressed` is a type alias for `boolean`. It helps us determine if a user has pressed a key or not. `EventCode` is a type alias for `event.code` - we will use it to figure out which key is pressed. In `Settings`, we use `KeyLabel` to define which key is to be observed. Functions `onStartPress()` and `onFinishPress()` are handlers for when a user presses a key and lifts their finger up respectively.

Define the hook:

```
export function usePressObserver({
  watchKey,
  onStartPress,
  onFinishPress
}: Settings): IsPressed {
  const [pressed, setPressed] = useState<IsPressed>(false)
  // ...
  return pressed
}
```

Here we take `Settings` as an argument and return `IsPressed` as a result. We will keep the state (`pressed` or not) in a local state of our component using `useState()` hook.

Now, let's implement the main logic using the `useEffect` hook:

```
  useEffect(() => {
    function handlePressStart({ code }: KeyboardEvent): void {
      if (pressed || !equal(watchKey, code)) return
      setPressed(true)
      onStartPress()
    }

    function handlePressFinish({ code }: KeyboardEvent): void {
      if (!pressed || !equal(watchKey, code)) return
      setPressed(false)
      onFinishPress()
    }

    document.addEventListener("keydown", handlePressStart)
    document.addEventListener("keyup", handlePressFinish)

    return () => {
      document.removeEventListener("keydown", handlePressStart)
      document.removeEventListener("keyup", handlePressFinish)
    }
  }, [watchKey, pressed, setPressed, onStartPress, onFinishPress])
```

TypeScript will show an error because the `equal()` function cannot be found. It's fine, we will create it in a minute.

Here, when a user presses a key, we call `handlePressStart()` to handle this event. We check if this key hasn't been pressed yet, and if not, we set the `pressed` variable to `true` and call `onStartPress()` callback. When a user finishes pressing the key, we call `onFinishPress()` inside `handlePressFinish()` handler.

We use `document.addEventListener()` to connect events and our named handler functions, and `document.removeEventListener()` inside a cleanup function which is returned from the `useEffect()`[110] hook. It is important to remove event listeners from a cleanup function to prevent memory leaks and unwanted event handlers calls.

Each `Key` component has its instance and thus creates a different `keyPress` event listener. When we press the real key on a keyboard each component will react

[110] https://reactjs.org/docs/hooks-effect.html

to this action. However, despite all the components reacting on an event, the real functionality gets executed only once — for the `Key` component that corresponds to a real one, because of this check:

```
if (pressed || !equal(watchKey, code)) return
```

If a given `Key` is already pressed or is not the target key, we don't do anything. This way, we prevent extra work from being done.

This effect uses 2 custom functions called `equal()` and `fromEventCode()`. Let's create them and explain what they do:

```
function fromEventCode(code: EventCode): KeyLabel {
  const prefixRegex = /Key|Digit/gi
  return code.replace(prefixRegex, "")
}

function equal(watchedKey: KeyLabel, eventCode: EventCode): boolean {
  return (
    fromEventCode(eventCode).toUpperCase() ===
    watchedKey.toUpperCase()
  )
}
```

The `fromEventCode` function takes an event code that can be presented like `KeyZ`, `KeyS`, `Digit9`, or `Digit4`. It uses regex to filter out all the `Key` and `Digit` prefixes and keep only a significant part of a code.

```
// `KeyZ` => `Z`
// `Digit9` => `9`
```

The `equal()` function compares the label of a key we observe and the pressed key. If they are the same, it means the user pressed an observed key.

Why to uppercase all of them? It is called normalization. We do it to make sure that either of `s` and `S` would work as a `watchedKey` as well as all the keys a user might press.

Okay, that's good. But why create a handler for each `Key`? We could still create a single global event handler to ensure that there is only one handler for all the key presses. However, it will violate the separation of concerns principle[111], according to which `Key` components should handle their events themselves.

Re-export the `usePressObserver` in the `src/components/PressObserver/index.ts` file:

```
export * from "./usePressObserver"
```

Let's connect the `usePressObserver()` to our `Key` component. Don't forget to import `usePressObserver` into the component.

```
import { usePressObserver } from "../PressObserver"
  // ...
  const pressed = usePressObserver({
    watchKey: label,
    onStartPress: onDown,
    onFinishPress: onUp
  })

  return (
    <button
      className={clsx(
        styles.key,
        styles[type],
        pressed && styles["is-pressed"]
      )}
      onMouseDown={onDown}
      onMouseUp={onUp}
      type="button"
      {...rest}
    >
      {label}
    </button>
  )
```

[111] https://en.wikipedia.org/wiki/Separation_of_concerns

We use `onDown()` and `onUp()` props as values for `onStartPress` and `onFinishPress` for the observer respectively, and use the returned `pressed` value to assign an active `className` to our `button`.

Instruments List

The last thing to do before we dive into *Render Props* and *Higher-Order Components* is to create an instruments list to load them dynamically. This part requires a state that will be accessible from many components, so we will use `React.Context` to share that state.

Context

Let's start with creating a new `Context`. We will call it `InstrumentContext`. Create a file `src/state/Instrument/Context.ts` and add the following code:

```
import { createContext, useContext } from "react"
import { InstrumentName } from "soundfont-player"
import { DEFAULT_INSTRUMENT } from "../../domain/sound"

export type ContextValue = {
  instrument: InstrumentName
  setInstrument: (instrument: InstrumentName) => void
}

export const InstrumentContext = createContext<ContextValue>({
  instrument: DEFAULT_INSTRUMENT,
  setInstrument() {}
})

export const InstrumentContextConsumer = InstrumentContext.Consumer
export const useInstrument = () => useContext(InstrumentContext)
```

Here we use `createContext()` function and specify that our context value is going to be of type `ContextValue`. It will keep a current `instrument` which we will be able to update via `setInstrument()`. As a default value for an instrument, we provide a `DEFAULT_INSTRUMENT` constant. From this file we want to export an `InstrumentContextConsumer` and `useInstrument()` hook to access the context.

Re-export `InstrumentContextConsumer` and `useInstrument` from `index.ts`:

```
export { InstrumentContextConsumer, useInstrument } from "./Context"
```

The next step is to create an `InstrumentContextProvider` that will provide access to the context. Create a file `src/state/Instrument/Provider.tsx` and add the following code:

```
import { FunctionComponent, useState } from "react"
import { DEFAULT_INSTRUMENT } from "../../domain/sound"
import { InstrumentContext } from "./Context"

export const InstrumentContextProvider: FunctionComponent = ({
  children
}) => {
  const [instrument, setInstrument] = useState(DEFAULT_INSTRUMENT)

  return (
    <InstrumentContext.Provider value={{ instrument, setInstrument }}>
      {children}
    </InstrumentContext.Provider>
  )
}
```

The `InstrumentContextProvider` is a component that keeps the `instrument` value in a local state and exposes the `setInstrument()` method to update it. We use `Context.Provider` to set a value and render `children` inside. That will help us wrap our entire application in this provider and gain access to the `InstrumentContext` from anywhere.

Finally, re-export the provider from `index.ts`:

```
export { InstrumentContextConsumer, useInstrument } from "./Context"
export * from "./Provider"
```

Instrument Selector

Now, let's try to update a current instrument. To select an instrument we will need a list of instruments. This list will be rendered inside a `select` element, so we also need a list of options for this `select`.

Let's start with creating those options. Create a directory called `InstrumentSelector` inside `components`, add the `options.ts` file, and add the following code:

```
import { InstrumentName } from "soundfont-player"
import instruments from "soundfont-player/names/musyngkite.json"

type Option = {
  value: InstrumentName
  label: string
}

type OptionsList = Option[]
type InstrumentList = InstrumentName[]

function normalizeList(list: InstrumentList): OptionsList {
  return list.map((instrument) => ({
    value: instrument,
    label: instrument.replace(/_/gi, " ")
  }))
}

export const options = normalizeList(instruments as InstrumentList)
```

Options are an array of `Option` objects. Each object contains a `value` of type `InstrumentName` and a `label` of type `string`. We will use a `value` as a `value` for `option`

HTML-elements in `select` - also this is our current instrument in `InstrumentContext`. `Label` is a string that we will put inside of `option` elements to render them and make them visible for users.

The function `normalizeList()` converts instrument names provided by Soundfont into readable ones. Soundfont gives us a list of instruments that are typed like `"acoustic_grand_piano"`, but we don't want our users to see this underscore between words. So we remove it and replace it with a space.

Then, create another file called `InstrumentSelector.tsx` inside `InstrumentSelector` directory. Add the imports there:

```
import { ChangeEvent } from "react"
import { InstrumentName } from "soundfont-player"
import { useInstrument } from "../../state/Instrument"
import { options } from "./options"
import styles from "./InstrumentSelector.module.css"
```

And the component code:

```
export const InstrumentSelector = () => {
  const { instrument, setInstrument } = useInstrument()
  const updateValue = ({ target }: ChangeEvent<HTMLSelectElement>) =>
    setInstrument(target.value as InstrumentName)

  return (
    <select
      className={styles.instruments}
      onChange={updateValue}
      value={instrument}
    >
      {options.map(({ label, value }) => (
        <option key={value} value={value}>
          {label}
        </option>
      ))}
    </select>
```

```
  )
}
```

Here we use our `useInstrument()` custom hook to get a current instrument value and a method for updating it. Afterwards, we create an event handler called `updateValue()` which takes a `ChangeEvent<HTMLSelectElement>` as an argument and calls `setInstrument()` with a new `InstrumentName`.

`ChangeEvent` is a generic type that tells React that this function takes a change event of an element. In our case this element is `select`, hence `ChangeEvent<HTMLSelectElement>`.

> How to inspect declarations for those types? We can right-click on the type and select "Go to definition", which will navigate us to the type declaration.

The way we set the `onChange` property to have a value of `updateValue` is how we connect our `Context` to a component in the UI. That is where all the changes affect our state.

Add component styles, create a file `InstrumentSelector.module.css` inside `InstrumentSelector` directory and add the following code:

```css
.instruments {
  display: block;
  text-transform: capitalize;
  font-size: 1.2rem;
  line-height: 1.5;

  margin: 1.5rem auto 0;
  padding: 0.4rem 1rem;

  color: #495057;
  background-color: #fff;
  background-clip: padding-box;
  border: 1px solid #ced4da;
  border-radius: 0.25rem;
}
```

Finally, re-export the component from `index.ts`:

```
export * from "./InstrumentSelector"
```

Provide access to the `InstrumentContext` using the `InstrumentContextProvider`. Create a `src/components/Playground` directory and there create a file called `Playground.tsx` with the following code:

```
import { InstrumentContextProvider } from "../../state/Instrument"
import { InstrumentSelector } from "../InstrumentSelector"
import { KeyboardWithInstrument } from "../Keyboard"

export const Playground = () => {
  return (
    <InstrumentContextProvider>
      <div className="playground">
        <KeyboardWithInstrument />
        <InstrumentSelector />
      </div>
    </InstrumentContextProvider>
  )
}
```

Here we wrap our `Keyboard` and `InstrumentSelector` in a component called `Playground`. Inside of it we use `InstrumentContextProvider`. We could wrap the entire application in it. However, that is not necessary. In our case, there are only two components that use `InstrumentContext:Keyboard` and `InstrumentSelector`, so we wrap only the two of them into the context provider.

Re-export the `Playground` component:

```
export * from "./Playground"
```

The next thing to do is update our `Main` component — we want to include and use `Playground` instead of a `Keyboard` that we used previously.

```
import { Playground } from "../Playground"
import { NoAudioMessage } from "../NoAudioMessage"
import { useAudioContext } from "../AudioContextProvider"

export const Main = () => {
  const AudioContext = useAudioContext()
  return !!AudioContext ? <Playground /> : <NoAudioMessage />
}
```

Use the `Main` component inside `App`:

```
import { Main } from "./components/Main";
  // ...
export const App = () => {
  return (
    <div className={styles.app}>
      <Logo />
      <main className={styles.content}>
        <Main />
      </main>
      <Footer />
    </div>
  );
};
```

We're almost there! The only thing to do now is to actually load a new sound set when changing a current instrument. Let's update our `KeyboardWithInstrument` component to handle this case.

Dynamically Loading Instruments

Open `src/components/Keyboard/WithInstrument.tsx` and add the imports:

```
import { useEffect } from "react"
import { useInstrument } from "../../state/Instrument"
import { useSoundfont } from "../../adapters/Soundfont"
import { useAudioContext } from "../AudioContextProvider"
import { Keyboard } from "../Keyboard"
```

Update the component:

```
export const KeyboardWithInstrument = () => {
  const AudioContext = useAudioContext()!
  const { instrument } = useInstrument()
  const { loading, current, play, stop, load } = useSoundfont({
    AudioContext
  })

  useEffect(() => {
    if (!loading && instrument !== current) load(instrument)
  }, [load, loading, current, instrument])

  return <Keyboard loading={loading} play={play} stop={stop} />
}
```

Here we use the `useInstrument()` hook to access the value of a current instrument. Later, we call `load()` function providing `instrument` as an argument for it. It will tell Soundfont to load the sound set for this particular instrument.

We replace `useMount()` hook with `useEffect()` hook because we want to change our instrument's sound set dynamically instead of loading it only on mount.

Also, we check if an instrument has changed and load the new one only if so. For that, we use the `current` value provided by `useSoundfont()` hook earlier. We compare a `current` instrument in the Soundfont provider and a wanted `instrument` from our `Context`. If they are different, we call the `load()` function.

And that's it! Now you can open the project in a browser and play with different instruments sounds.

Render Props

So far, we have used only hooks to implement a *Provider* pattern. However, we can use different techniques to achieve the same result. One of those techniques is a React pattern called *Render Props*.

In this section we'll learn what render props are and what are their pros and cons.

What is a render prop

A component with a render prop[112] receives a function that returns a React element and calls this function instead of implementing its own render logic. This technique makes it possible to share the internal logic between components.

Let's try to imagine how a component with the `render` function would look. Its usage would look like this:

```
<ExampleRenderPropsComponent
  render={(name: string) => <div>Hello, {name}!</div>}
/>
```

The `render` prop takes a function that returns another React component. However, it does not just render a component but also its inner text containing a `name`. This `name` is a value calculated inside of `ExampleRenderPropsComponent`.

So, this function for `render` in a way connects internal values of `ExampleRenderPropsComponent` with the outside world. We expose this internal value to the outer world. The coolest thing is that we can decide what to share with the outer world and what not to. We could have a hundred internal values inside of `ExampleRenderPropsComponent`, but expose only one.

Thus, we can encapsulate the logic in one place — `ExampleRenderPropsComponent` — but share some functionality with different components:

[112] https://reactjs.org/docs/render-props.html

```
<ExampleRenderPropsComponent
  render={(name: string) => <Greetings name={name} />}
/>
<ExampleRenderPropsComponent
  render={(name: string) => <Farewell name={name} />}
/>
```

Here we expose the `name` value to `Greetings` and `Farewell`. We don't recreate all the operations required to get `name` by hands, but instead, we keep them inside of `ExampleRenderPropsComponent` and use `render` to *provide* it to other components.

We don't necessarily need to call this prop `render`. We can use the `children` prop as well. In that case, the `children` prop would become a function, and we would use our provider like this:

```
<SoundfontProvider AudioContext={AudioContext} instrument={instrument}>
  {(props) => <Keyboard {...props} />}>
</SoundfontProvider>
```

Be careful when using Render Props with `React.PureComponent`[113].

Using a Render Prop can negate the advantage that comes from using `React.PureComponent` if we create the function inside a `render` method. The reason for this is that the shallow prop comparison will always return `false` for new props, and each render in this case will generate a new value for the render prop.

To get around this problem, we can sometimes define the prop as an instance method. In cases where we cannot define the prop statically, we should extend `React.Component` instead.

Pros and Cons

Each pattern has its limitations and usage cases. For *Render Props*, the pros would be that a *Render Props* Provider:

[113]https://reactjs.org/docs/render-props.html

- Explicitly shows where all the methods come from;
- Declaratively loads an instrument via prop;
- Can be written as a class and as a function component.

The cons are that a *Render Props* Provider:

- Adds one to two nesting levels to a component that uses it;
- Needs a render to be called.

Creating Render Props With Functional Components

Inside the `src/adapters/Soundfont` directory create a file called `SoundfontProvider.ts`. Add the necessary imports:

```
import {
  ReactElement,
  FunctionComponent,
  useState,
  useEffect,
  useRef,
  useCallback
} from "react"
import Soundfont, { InstrumentName, Player } from "soundfont-player"
import { MidiValue } from "../../domain/note"
import { Optional } from "../../domain/types"
import {
  AudioNodesRegistry,
  DEFAULT_INSTRUMENT
} from "../../domain/sound"
```

Declare the component props:

```
type ProvidedProps = {
  loading: boolean
  play(note: MidiValue): Promise<void>
  stop(note: MidiValue): Promise<void>
}

type ProviderProps = {
  instrument?: InstrumentName
  AudioContext: AudioContextType
  render(props: ProvidedProps): ReactElement
}
```

We would require an optional `instrument` prop to specify which instrument we want to load, and an `AudioContext` to utilize. Most importantly, we would need a `render` prop that is a function that takes `ProvidedProps` as an argument and returns a `ReactElement`. `ProvidedProps` is a type with values that we would provide to the outside world.

The same values we provided earlier with the `useSoundfont()` hook but without `load()` and `current`. We don't need them because we encapsulate the loading of sounds inside our provider. A current instrument now arrives from the outside via the `instrument` prop.

Also, we don't return them as a function result; but instead, we pass them as a `render` function argument. Thus, the usage of our new provider would look like this:

```
function renderKeyboard({
  play,
  stop,
  loading
}: ProvidedProps): ReactElement {
  return <Keyboard play={play} stop={stop} loading={loading} />
}

/** ...And we would use it like:
 * <SoundfontProvider
 *  AudioContext={AudioContext}
```

```
 *   instrument={instrument}
 *   render={renderKeyboard}
 * />
 */
```

When we are okay with the API of our new provider, we can start implementing it. A type signature of this provider would be like this:

```
export const SoundfontProvider: FunctionComponent<ProviderProps> = ({
  AudioContext,
  instrument,
  render
}) => {
  // ...
}
```

We explicitly say that this is a `FunctionComponent` that accepts `ProviderProps`.

All the work with the internal state would be the same as it was in the `useSoundfont()` hook, except that we add loading and reloading sounds when the `instrument` prop is being changed.

The local state will look like this:

```
  let activeNodes: AudioNodesRegistry = {}

  const [current, setCurrent] = useState<Optional<InstrumentName>>(
    null
  )
  const [loading, setLoading] = useState<boolean>(false)
  const [player, setPlayer] = useState<Optional<Player>>(null)
  const audio = useRef(new AudioContext())

  const loadInstrument = useCallback(() => load(instrument), [
    instrument
  ])
```

The loading instrument effect will look like this:

```
useEffect(() => {
  if (!loading && instrument !== current) loadInstrument()
}, [loadInstrument, loading, instrument, current])
```

Here, we use `useEffect()` to capture when an `instrument` prop changes and load a new sound set for that instrument. However we don't call `load()` function, instead we call a memoized version[114] of it — this is possible because of the `useCallback()` hook.

The `load()` function is as follows:

```
async function load(
  instrument: InstrumentName = DEFAULT_INSTRUMENT
) {
  setLoading(true)
  const player = await Soundfont.instrument(
    audio.current,
    instrument
  )

  setLoading(false)
  setCurrent(instrument)
  setPlayer(player)
}
```

The `play()`, `stop()`, and `resume()` functions are exactly the same as they were in the `useSoundfont` hook:

```
async function resume() {
  return audio.current.state === "suspended"
    ? await audio.current.resume()
    : Promise.resolve()
}
```

[114] https://reactjs.org/docs/hooks-reference.html#usecallback

```
  async function play(note: MidiValue) {
    await resume()
    if (!player) return

    const node = player.play(note.toString())
    activeNodes = { ...activeNodes, [note]: node }
  }

  async function stop(note: MidiValue) {
    await resume()
    if (!activeNodes[note]) return

    activeNodes[note]!.stop()
    activeNodes = { ...activeNodes, [note]: null }
  }
```

This is the logic we previously implemented in the `KeyboardWithInstrument`, but now encapsulated in the provider.

Expose the internal values and functions to the outside world. For that, we use `render()`:

```
  return render({
    loading,
    play,
    stop
  })
```

As you can see, we call `render()` and pass inside it an object with all the values and functions that we promised to pass in `ProvidedProps`.

Now, re-export the provider from `index.ts`:

```
export * from "./SoundfontProvider"
```

Tweak the code of the `KeyboardWithInstrument` component a bit.

```
import { SoundfontProvider } from "../../adapters/Soundfont"
  // ...
export const KeyboardWithInstrument = () => {
  const AudioContext = useAudioContext()!
  const { instrument } = useInstrument()

  return (
    <SoundfontProvider
      AudioContext={AudioContext}
      instrument={instrument}
      render={(props) => <Keyboard {...props} />}
    />
  )
}
```

Here we pass the `AudioContext` and an `instrument` as props to `SoundfontProvider` and then pass to `render` a callback. It takes `loading`, `play()` and `stop()`, transfers them to a `Keyboard` and returns it. We use object destructuring not to enumerate each prop for `Keyboard` manually but to pass them right away instead.

Creating Render Props With Classes

We can use classes to create *Render Props* components as well. Let's rebuild our provider using the same technique but based on a `class`.

Classes are like a blueprint for creating similar entities. In TypeScript, classes can implement interfaces and extend more general classes. For example, we have an interface `Printable` that describes a behavior contract. It guarantees that the entity implementing this interface has a method `print()`.

```
interface Printable {
  print(): void
}
```

A class can declare that it implements this interface. TypeScript will check if this class has all the methods specified in the interface:

```
class Article implements Printable {
  print(): void {
    console.log('Printed!');
  }
}
```

If some of the methods are missing, TypeScript will produce an error:

> Class 'Article' incorrectly implements interface 'Printable'. Property 'print' is missing in type 'Article' but required in type 'Printable'.

We can extend a class and modify its behavior a bit. It is useful when we want to enrich the classes' basic functionality. For example, we can specify an additional property:

```
class LongRead extends Article {
  wordsCount = 1000;

  print(): void {
    console.log('Printed!');
  }
}
```

To create a new entity of an `Article` class, we call it with `new`. Every entity is a separate object and can be manipulated separately:

```
const aboutNature = new LongRead();
aboutNature.print();
aboutNature.wordsCount === 1000
```

So, a class is a blueprint, and every entity is a separate entity… Isn't it similar to components? It is, indeed. As we will see later, React provides us with a `Component` class that we can extend and create our components based on its general functionality.

Basically, `Component` deals with the inner details of a component lifecycle: it determines when to update and re-render, how to create a local state, and stuff. Our extensions (components) only define modified functionality, like the component markup. With all that in mind, let's try and create a class component. Imports will be the same, but we're going to need to import `Component` from `React` as well.

Create a file called `SoundfontProviderClass.ts` inside `src/adapters/Soundfont` directory and add the imports:

```
import { Component, ReactElement } from "react"
import Soundfont, { InstrumentName, Player } from "soundfont-player"
import { MidiValue } from "../../domain/note"
import { Optional } from "../../domain/types"
import {
  AudioNodesRegistry,
  DEFAULT_INSTRUMENT
} from "../../domain/sound"
```

`ProvidedProps` would still be the same, because we don't change the public API. `ProviderProps`, on the other hand, will change. This time the `instrument` field will not be optional.

```
type ProvidedProps = {
  loading: boolean
  play(note: MidiValue): Promise<void>
  stop(note: MidiValue): Promise<void>
}

type ProviderProps = {
  instrument: InstrumentName
  AudioContext: AudioContextType
  render(props: ProvidedProps): ReactElement
}
```

That's because we will use `defaultProps`[115] when nothing is passed to a component. We will see how to define them in a minute.

[115] https://www.typescriptlang.org/docs/handbook/release-notes/typescript-3-0.html#support-for-defaultprops-in-jsx

Then, since we will use a `class`, we specify a state type because the `useState()` hook is not available in the class components. We can use Hooks only inside functional components. So, let's introduce the `ProviderState` type.

```
type ProviderState = {
  loading: boolean
  current: Optional<InstrumentName>
}
```

Here we declare that our local state should contain a `loading` field, a `boolean` and `current`, an `Optional<InstrumentName>`. Those are the parts that should cause re-render when changed.

```
export class SoundfontProvider extends Component<
  ProviderProps,
  ProviderState
> {
  public static defaultProps = {
    instrument: DEFAULT_INSTRUMENT
  }

  private audio: AudioContext
  private player: Optional<Player> = null
  private activeNodes: AudioNodesRegistry = {}

  public state: ProviderState = {
    loading: false,
    current: null
  }
  // ...
}
```

As you may notice, we now pass two types into the `Component<>` type. The first one describes props, and the second one describes a state. Also, we created three private fields for our class. Those are `audio`, `player`, and `activeNodes`. We make them `private` because we don't want outside entities to mess around with those fields. It is considered good practice to mark everything that is not `public` as `private` or `protected`.

> The difference[116] between private and protected is that `private` members are accessible only from inside the class, and `protected` members are accessible from inside the class and extending classes as well.

Notice, `defaultProps` there. We declare them as a `static` field on a class.

```
public static defaultProps = {
  instrument: DEFAULT_INSTRUMENT
}
```

Then, we create a `constructor()` method. This is the method[117] called right after a class creation.

```
constructor(props: ProviderProps) {
  super(props)

  const { AudioContext } = this.props
  this.audio = new AudioContext()
}
```

Here we call[118] the `super(props)` method. The `super()` method calls parent constructor. To avoid situations when `this.props` are not assigned to a component until the constructor is finished, we set them via `super(props)`. Otherwise we would not be able to access `AudioContext` from `this.props` in a constructor later. Then, we get `AudioContext` and assign `this.audio` to its instance.

So far, this seems pretty good. Now, let's imagine our component's lifecycle - what should occur and when. When a component is created, we assign `private` fields. When it's mounted, we load an initial instrument. When the latter changes due to

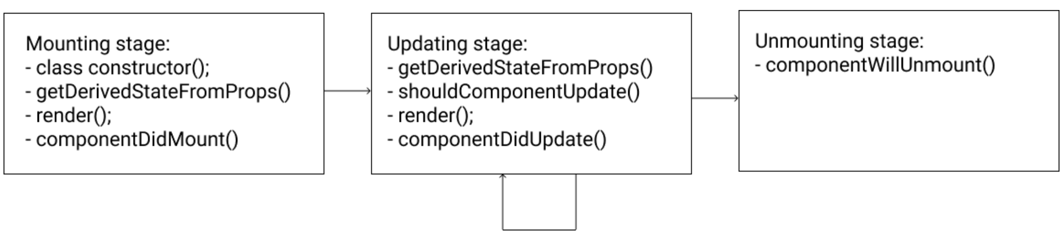
[116] https://www.typescriptlang.org/docs/handbook/2/classes.html#member-visibility
[117] https://www.typescriptlang.org/docs/handbook/2/classes.html#methods
[118] https://overreacted.io/why-do-we-write-super-props/

the component's update, we check if the new instrument differs from the current one and reload it if so.

The whole lifecycle consists of 3 stages:

- mounting, when a component is being created and inserted into the DOM;
- updating, when changes to props or state happen, a component is being re-rendered;
- unmounting, when a component leaves the DOM.

At every stage, there are available methods provided by the `Component` class. On a diagram, component lifecycle and corresponding methods would appear like this:



Component lifecycle diagram

We used four lifecycle[119] methods in our code:

- `constructor()` — which we discussed before
- `componentDidMount()` — which is called when a component is mounted into the DOM
- `shouldComponentUpdate()` — which is called right before updating and determines if a component needs to be updated and re-rendered
- `componentDidUpdate()` — which is called when a component has been updated

[119] https://reactjs.org/docs/state-and-lifecycle.html

```
  public componentDidMount() {
    const { instrument } = this.props
    this.load(instrument)
  }

  public shouldComponentUpdate({ instrument }: ProviderProps) {
    return this.state.current !== instrument
  }

  public componentDidUpdate({
    instrument: prevInstrument
  }: ProviderProps) {
    const { instrument } = this.props
    if (instrument && instrument !== prevInstrument)
      this.load(instrument)
  }
```

That is exactly what we do in those methods. When a component is mounted, we access the `instrument` prop and load it using `this.load()`. Before the update, we check if a current instrument (`this.state.current`) is different from the new one from props, and if so we load it.

The `shouldComponentUpdate()` is not an optimization here. We use it to prevent infinite reloading of instruments that could happen because of asynchronous loading.

There is no need to check if an `instrument` is defined or not in `componentDidMount()`, thanks to `defaultProps`.

Now, let's implement the `this.load()` method for loading sounds. We mark it `private` to restrict it from used by any other class or object.

```
  private load = async (instrument: InstrumentName) => {
    this.setState({ loading: true })
    this.player = await Soundfont.instrument(this.audio, instrument)

    this.setState({ loading: false, current: instrument })
  }
```

We use `this.setState()` to update `loading` flag which will be provided later to a component in `render()`. This method is `public`, since we want to expose it to the outer world. However, make sure to mark the `load()` method as private, since we don't want its exposure to the outer world in any way.

There are two other methods now that to implement and expose:

```
  public play = async (note: MidiValue) => {
    await this.resume()
    if (!this.player) return

    const node = this.player.play(note.toString())
    this.activeNodes = { ...this.activeNodes, [note]: node }
  }

  public stop = async (note: MidiValue) => {
    await this.resume()
    if (!this.activeNodes[note]) return

    this.activeNodes[note]!.stop()
    this.activeNodes = { ...this.activeNodes, [note]: null }
  }
```

It repeats the logic from our functional component provider. However, here we don't change local variables but private class fields instead. All the signatures, API, and implementation are the same.

> This is what makes abstractions, custom types, and interfaces so powerful. We can describe an interface (sort of creating a contract), and as long as

we implement this interface, we can tweak and change the internals of the implementation as we want.

Now we create `resume()` method, which is almost identical to our `resume()` function from the previous adapter.

```
private resume = async () => {
  return this.audio.state === "suspended"
    ? await this.audio.resume()
    : Promise.resolve()
}
```

We then expose the methods and values to the `render()` function. We access that function from `this.props` and take it and pass to it as an argument the object with all the values and methods we promised to provide in `ProvidedProps`.

```
public render() {
  const { render } = this.props
  const { loading } = this.state

  return render({
    loading,
    play: this.play,
    stop: this.stop
  })
}
```

And that's it! This is the *Render Props* component based on a class. We can use it the same way we used our previous provider based on a functional component.

Higher-Order Components

The next React-Pattern we're going to explore is called *Higher-Order Components* or HOC. Let's first break down this name to understand what it means.

Higher-Order Functions

To grasp what "order" means, let's have a look at the functions first.

```
function increment(a: number): number {
  return a + 1
}
```

Function `increment()` is a regular function that takes a number and returns the sum of this number and 1. It is a first-order function.

```
function twice(fn: Function): Function {
  return function (...args: unknown[]) {
    return fn(fn(...args))
  }
}
```

The `twice()` function is a function that takes another *function* as an argument and returns a *function* as a result. This characteristic makes it a function with an order *higher than the first*.

Basically, any given function that either takes a function as an argument or returns a function as a result or does both, is a function with order *higher than the first*, hence the name — *higher-order function*[120].

This kind of function is useful for *composition*. This term[121] comes from functional programming, and essentially it is a mechanism that makes it possible to take simple functions and build more complicated ones based on them.

Let's continue with our example here. We can create a function that will increment a number twice. A naive way to do that would be:

[120] https://en.wikipedia.org/wiki/Higher-order_function
[121] https://en.wikipedia.org/wiki/Function_composition_(computer_science)

```
function incrementTwice(a: number): number {
  return increment(increment(a))
}
```

This is not very good because we cannot be sure that there won't be a requirement to increase this number in the future. Also, hardcoded logic is not good in general.

The `twice()` function shares some similarities with our `incrementTwice()` function. They both call a function two times in a row, but `incrementTwice()` calls a specific function (`increment()`), and `twice()` calls an *abstract* function that comes from its argument (`fn()`).

We can use the `twice()` function to achieve the same result as we did with `incrementTwice()`.

```
const anotherIncrementTwice = twice(increment)
```

Yup, that's it! Let's see how it works step by step.

When we call `twice()` and pass the `increment` as an argument, the variable `fn` starts carrying the value of the `increment` function. So, after the first step, `fn` is `increment`.

Then, we create an anonymous function that takes an array of arguments `function(...args: unknown[])`. To prevent this function from calling `fn` right away since we only want to "prepare" and "remember" which function we plan to call two times in the future.

We return this anonymous function. Thus, when we assign `const anotherIncrementTwice` to a result of `twice(increment)`, we actually assign the anonymous function that already "remembers" which function we wanted to call twice to the `anotherIncrementTwice` constant. This functions knows that it should call `increment()` twice when called, and it takes some arguments that will be passed to `increment()`.

If we try to write it down, it will look almost exactly like it did earlier:

```
const anotherIncrementTwice = function (...args: unknown[]) {
  return increment(increment(...args))
}
```

Surely, it returns the same result as the previous one:

```
const result1 = incrementTwice(5) // returns 7
const result2 = anotherIncrementTwice(5) // returns 7

result1 === result2 // true
```

The only difference here is that this function previously took only one argument, and now it takes an array of arguments. It is a side effect of the fact that we can now use the function `twice()` with any other function to repeat it!

```
function sayHello(): void {
  console.log(`Hello world!`);
}

const sayHelloTwice = twice(sayHello);
sayHelloTwice()

// Hello world!
// Hello world!
```

Instead of implementing this logic from scratch we used a *higher-order function* `twice()` to build a compound function `sayHelloTwice()` from a simple `sayHello()`.

Higher-Order Components carry the same idea but in the realm of React components.

Define a HOC

A basic implementation of a HOC would look like this:

```
function withLogging<T>(Component: React.ComponentType<T>): React.Compo\
nentType<T> {
  return class extends React.Component<T> {
    render() {
      console.log(`Rendering ${Component.name}`);
      return <Component {...this.props} />;
    }
  };
}
```

Here we'll log a message to the console before rendering the wrapped component.

The key parts here are:

- the factory function (`withLogging`) that takes a component as an argument and returns a new component
- the wrapper component (`class`) that wraps the original component
- the wrapped component (`Component`) that is being enchanced

When to Use

We can use HOCs when we need to share functionality between many components. Injectors can extend the functionality of a given component by passing new props to it.

Sometimes HOCs are used to access network requests, provide local storage access, subscribe to event streams, or connect components to an application store. The latter was used in the Redux library to connect a component to the Redux store. These HOCs are often called *providers* but they work basically the same way.

Pros and Cons

HOCs have limitations and caveats too. We can consider as pros these aspects:

- Static composition possibility - we can "remember" arguments for the future. However, we can also do it in other patterns via Factory pattern or currying, so this is debatable.
- HOCs are a literal implementation of a Decorator pattern.

And as cons:

- Extra encapsulation and "implicitness". Sometimes HOCs hide too much logic inside them, and it is not clear what will happen when we wrap some component in a HOC.
- Unobvious typings strategy and presence of generics, type-casting "on the fly", and overall difficulty level. It is much harder to understand what is going on in the code, compared to functional components.
- HOCs may become too verbose.

Caveats

We cannot[122] wrap a component in HOC inside of `render()` (in runtime). React's diffing algorithm uses component identity to determine whether it should update the existing subtree or throw it away and mount a new one. The problem here isn't just about performance. Remounting a component causes the state of that component and all of its children to be lost. We must always apply HOCs outside the component definition so that the resulting component is created only once.

All the static methods if defined must be copied[123] over.

There may be a situation when some props provided by a HOC have the same names as props from other HOCs or wrappers. The name collision can lead us to accidentally overridden props.

[122]https://reactjs.org/docs/higher-order-components.html#dont-use-hocs-inside-the-render-method
[123]https://reactjs.org/docs/higher-order-components.html#static-methods-must-be-copied-over

Instrument adapter as a Higher-Order Component

Higher-Order Components are like *higher-order functions* but in the realm of React components.

How is it described in official docs[124]? Conceptually, components are like JavaScript functions. They accept arbitrary inputs (called "props") and return React elements describing what should appear on the screen.

So, we can say that a component is a *function* of some data passed via props. Therefore, we can continue this analogy with functions and extend it. What would a Higher-Order Component be?

Since a higher-order function either takes a function or returns a function or both, we can assume that a higher-order component takes a component and returns another one as a result. See what the official docs tell us[125].

While a component transforms props into UI, a higher-order component transforms a component into another one, enhanced somehow. In our case, the enhancement would be in connecting a component to a Soundfont functionality. With that said, let's try and build a Soundfont provider based on HOC.

First, imports. Create a file called `withInstrument.tsx` inside `src/adapters/Soundfont` and add the following code:

```
import { Component, ComponentType } from "react"
import Soundfont, { InstrumentName, Player } from "soundfont-player"
import { MidiValue } from "../../domain/note"
import { Optional } from "../../domain/types"
import {
  AudioNodesRegistry,
  DEFAULT_INSTRUMENT
} from "../../domain/sound"
```

[124] https://reactjs.org/docs/components-and-props.html
[125] https://reactjs.org/docs/higher-order-components.html

The public API would stay the same as it was before. However, `ProvidedProps` would be called `InjectedProps` now since we would inject them into a component that we will enhance. `ProviderProps` and `ProviderState` are the same as before.

```
type InjectedProps = {
  loading: boolean
  play(note: MidiValue): Promise<void>
  stop(note: MidiValue): Promise<void>
}

type ProviderProps = {
  AudioContext: AudioContextType
  instrument: InstrumentName
}

type ProviderState = {
  loading: boolean
  current: Optional<InstrumentName>
}
```

Then, we create a function `withInstrument()` that takes a component needing enhancement. We make this function generic to tell the type checker which props we're going to inject. We will cover the injection itself a bit later.

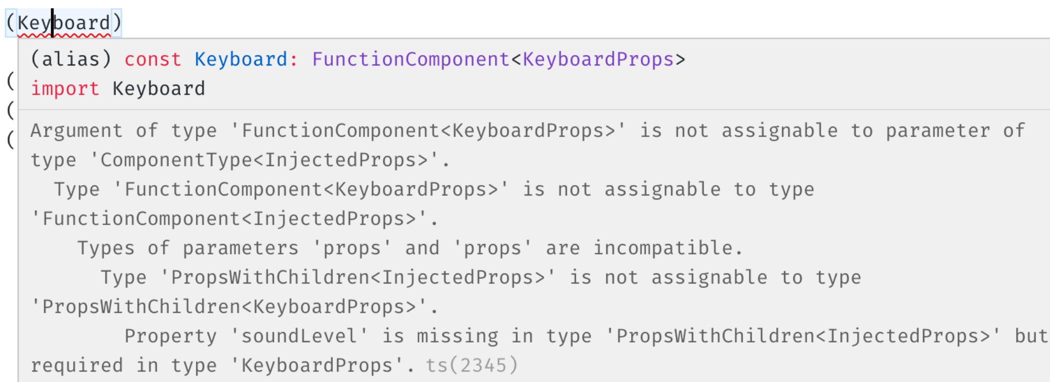
```
export function withInstrument<
  TProps extends InjectedProps = InjectedProps
>(WrappedComponent: ComponentType<TProps>) {
  // ...
}
```

Pay attention to the `extends` keyword in the type arguments declaration. This is a generic constraint[126]. We use it to define that `TProps` must include properties described in the `InjectedProps` type. Otherwise, TypeScript should give us an error.

[126] https://www.typescriptlang.org/docs/handbook/2/generics.html#generic-constraints

Why use constraints and not just `InjectedProps` right away? We don't always know what props will accept the component that we should enhance. So if we use `InjectedProps`, but the component accepts another prop, `soundLevel`, it won't be possible to enhance it.

For example, if we tried to pass the `Keyboard` component without extending props we would get an error:



Component cannot be used because of inextensible props

When we use `extends`, we tell TypeScript that it is okay to use any component that accepts `InjectedProps` even if there are more props than that.

By default, we define `TProps` to be the `InjectedProps` type using the `=` sign. This is the default type for this generic. It works exactly like default values for arguments in functions.

Inside, we create a const called `displayName` which is useful[127] for debugging. A container component that we're going to create will show up in developer tools like any other component. So, we'd better give it a name to make it recognizable in an inspector.

[127] https://reactjs.org/docs/higher-order-components.html#convention-wrap-the-display-name-for-easy-debugging

```
const displayName =
  WrappedComponent.displayName ||
  WrappedComponent.name ||
  "Component"
```

Then, we create a class `WithInstrument` that we're going to return. That is the container component that will enhance our `WrappedComponent`.

```
return class WithInstrument extends Component<
  ProviderProps,
  ProviderState
> {
```

Define the properties, they are same as in the `SoundfontProviderClass` from the render props example:

```
  public static defaultProps = {
    instrument: DEFAULT_INSTRUMENT
  }

  private audio: AudioContext
  private player: Optional<Player> = null
  private activeNodes: AudioNodesRegistry = {}

  public static displayName = `withInstrument(${displayName})`
  public state: ProviderState = {
    loading: false,
    current: null
  }
```

The only new field here is `displayName`. We make this field of a `static`[128] class to be able to access it like `WithInstrument.displayName` without creating an instance.

Define the constructor:

[128] https://www.typescriptlang.org/docs/handbook/2/classes.html#static-members

```
    constructor(props: ProviderProps) {
      super(props)

      const { AudioContext } = this.props
      this.audio = new AudioContext()
    }
```

Define the life cycle methods:

```
public componentDidMount() {
  const { instrument } = this.props
  this.load(instrument)
}
```

Add the `resume()` method:

```
private resume = async () => {
  return this.audio.state === "suspended"
    ? await this.audio.resume()
    : Promise.resolve()
}
```

It should be `private` as we don't want to expose it.

Add the `load`, `play` and `stop` methods:

```
    public load = async (instrument: InstrumentName) => {
      this.setState({ loading: true })

      this.player = await Soundfont.instrument(this.audio, instrument)
      this.setState({ loading: false, current: instrument })
    }

    public play = async (note: MidiValue) => {
      await this.resume()
      if (!this.player) return
```

```

      const node = this.player.play(note.toString())
      this.activeNodes = { ...this.activeNodes, [note]: node }
    }

    public stop = async (note: MidiValue) => {
      await this.resume()
      if (!this.activeNodes[note]) return

      this.activeNodes[note]!.stop()
      this.activeNodes = { ...this.activeNodes, [note]: null }
    }
```

Define the `render()` method:

```
    public render() {
      const injected = {
        loading: this.state.loading,
        play: this.play,
        stop: this.stop
      } as InjectedProps

      return <WrappedComponent {...(injected as TProps)} />
    }
```

Here, instead of calling `this.props.render()` and passing an object with values and methods like we did with render props, we render the `WrappedComponent` and pass these values as props it.

Why cast `as TProps` when rendering `WrappedComponent`? Well, there is an issue[129] in TypeScript that erases type of props when using the spread operator (...). This point forces us to explicitly cast `injected` props to the `TProps` type.

HOCs that inject new props to a given component are called *injectors*. They are useful when we have cross-cutting concerns in our app, and we don't want to implement the same functionality repeatedly.

[129] https://github.com/Microsoft/TypeScript/issues/28938#issuecomment-450636046

For example, we now can use our `withInstrument()` HOC with not only a `Keyboard` but with any component that expects `play()` and `stop()` props to play notes. We can create a `Trombone` component or `Guitar` component. As long as they are connected to `withInstrument()`, they know how to play sounds, and we don't need to add this functionality to them directly.

Finally, re-export the component from `index.ts`:

```
export * from "./withInstrument"
```

Using HOC with Keyboard

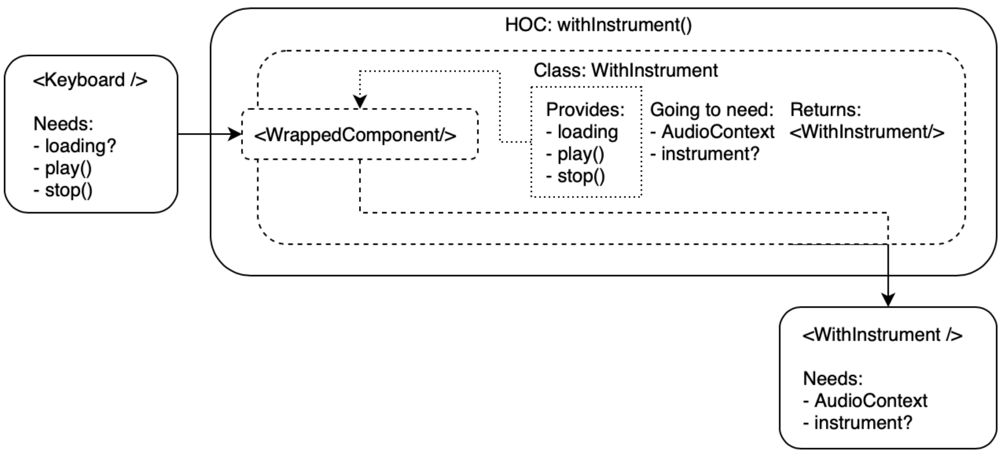
When created, we can use our HOC to enhance our `Keyboard` component to connect it to Soundfont. Let's import `withInstrument` and use it to create an enhanced `Keyboard`:

```
import { withInstrument } from "../../adapters/Soundfont"
  // ...
const WrappedKeyboard = withInstrument(Keyboard)

export const KeyboardWithInstrument = () => {
  const AudioContext = useAudioContext()!
  const { instrument } = useInstrument()

  return (
    <WrappedKeyboard
      AudioContext={AudioContext}
      instrument={instrument}
    />
  )
}
```

Here we can see how `withInstrument()` is being used; it takes a `Keyboard` component that requires `loading`, `play()` and `stop()` as props and returns a `WrappedKeyboard` that requires `AudioContext` and optional `instrument` props.

This is possible because a `Keyboard` becomes `WrappedComponent` when we call `withInstrument()`. Basically, `WrappedKeyboard` is a `WithInstrument` class that renders out a `Keyboard` with "remembered" injected props.

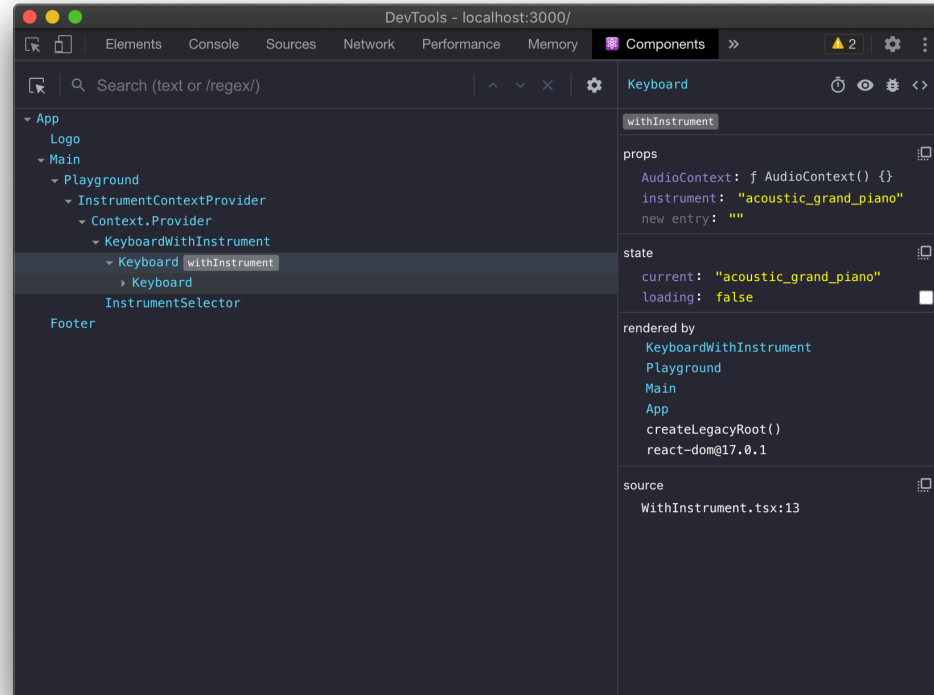
When we render `WrappedComponent`, it already has `loading`, `play()` and `stop()`, since they have been injected as `InjectedProps` earlier. It requires `ProviderProps` that were specified in `Component<ProviderProps, ProviderState>`.



Props flow in HOC

This is like when `fn` became `increment` and an anonymous function was "remembering" it.

To see what effect the `displayName` has, open the inspector now, find the components tab and click it. There we should see a component tree. It is different from the DOM tree because it shows not the HTML elements but the React components. Among others there should be a component `Keyboard withInstrument`:



Component with a display name in the components tree

Try to remove the `displayName` property from the HOC and see what will change in the components tree.

Passing Refs Through

Refs[130] provide a way to access DOM nodes or React elements created in the `render` method.

By default, refs aren't passed through[131], and for "true" reusability we can also

[130]https://reactjs.org/docs/refs-and-the-dom.html
[131]https://reactjs.org/docs/higher-order-components.html#refs-arent-passed-through

consider exposing[132] a ref for our HOC. For that we can use[133] `forwardRef()` function.

The base of our HOC will still be the same with a few changes. Create a file called `withInstrumentForwardedRef.tsx` inside `src/adapters/Soundfont` directory and add the imports:

```
import { Component, ComponentClass, Ref, forwardRef } from "react"
import Soundfont, { InstrumentName, Player } from "soundfont-player"
import { MidiValue } from "../../domain/note"
import { Optional } from "../../domain/types"
import {
  AudioNodesRegistry,
  DEFAULT_INSTRUMENT
} from "../../domain/sound"
```

The public API is the same:

```
type InjectedProps = {
  loading: boolean
  play(note: MidiValue): Promise<void>
  stop(note: MidiValue): Promise<void>
}

type ProviderProps = {
  AudioContext: AudioContextType
  instrument: InstrumentName
}

type ProviderState = {
  loading: boolean
  current: Optional<InstrumentName>
}
```

Declare some "runtime" types inside of `withInstrument()`.

[132] https://reactjs.org/docs/forwarding-refs.html
[133] https://react-typescript-cheatsheet.netlify.app/docs/basic/getting-started/forward_and_create_ref/

```
export function withInstrument<
  TProps extends InjectedProps = InjectedProps
>(WrappedComponent: ComponentClass<TProps>) {
  type ComponentInstance = InstanceType<typeof WrappedComponent>
  type WithForwardedRef = ProviderProps & {
    forwardedRef: Ref<ComponentInstance>
  }
  // ...
}
```

First, we create a `ComponentInstance` type. It is a type[134] consisting of the instance type of a component. We need it to pass it into `Ref<>` type to specify a ref of which component it would be. Then, we put this into a `WithForwardRef` type which extends `ProviderProps` type. At the same time, `forwardedRef` is a ref that we want to forward further into an enhanced component.

Basically, the root cause of the problem is that we create a container component that is just an intermediate element and has no real DOM elements. So, to provide access to a DOM node, we pass a received `ref` to the enhanced component, which will result in a DOM node when rendered.

Later, we declare a class `WithInstrument` as a `Component` of `WithForwardRef` props and `ProviderState`.

```
  const displayName =
    WrappedComponent.displayName ||
    WrappedComponent.name ||
    "Component"

  class WithInstrument extends Component<
    WithForwardedRef,
    ProviderState
  > {
  // ...
  }
```

[134] https://www.typescriptlang.org/docs/handbook/utility-types.html#instancetypetype

The private and public fields will be the same as before:

```
  private audio: AudioContext
  private player: Optional<Player> = null
  private activeNodes: AudioNodesRegistry = {}

  public static displayName = `withInstrument(${displayName})`
  public static defaultProps = {
    instrument: DEFAULT_INSTRUMENT
  }

  public state: ProviderState = {
    loading: false,
    current: null
  }
```

The constructor:

```
  constructor(props: WithForwardedRef) {
    super(props)

    const { AudioContext } = this.props
    this.audio = new AudioContext()
  }
```

...The life cycle methods:

```
  public componentDidMount() {
    const { instrument } = this.props
    this.load(instrument)
  }

  public shouldComponentUpdate({ instrument }: ProviderProps) {
    return this.state.current !== instrument
  }

  public componentDidUpdate({
    instrument: prevInstrument
  }: ProviderProps) {
    const { instrument } = this.props
    if (instrument && instrument !== prevInstrument)
      this.load(instrument)
  }
```

...The `resume()` method:

```
  private resume = async () => {
    return this.audio.state === "suspended"
      ? await this.audio.resume()
      : Promise.resolve()
  }
```

...And the public methods will also be the same as before:

```
    public load = async (instrument: InstrumentName) => {
      this.setState({ loading: true })

      this.player = await Soundfont.instrument(this.audio, instrument)
      this.setState({ loading: false, current: instrument })
    }

    public play = async (note: MidiValue) => {
      await this.resume()
      if (!this.player) return

      const node = this.player.play(note.toString())
      this.activeNodes = { ...this.activeNodes, [note]: node }
    }

    public stop = async (note: MidiValue) => {
      await this.resume()
      if (!this.activeNodes[note]) return

      this.activeNodes[note]!.stop()
      this.activeNodes = { ...this.activeNodes, [note]: null }
    }
```

In the `render()` method, we access `forwardedRef` from props and pass it as `ref` props onto a `WrappedComponent`.

```
    public render() {
      const { forwardedRef } = this.props
      const injected = {
        loading: this.state.loading,
        play: this.play,
        stop: this.stop
      } as InjectedProps

      return (
        <WrappedComponent
```

```
          ref={forwardedRef}
          {...(injected as TProps)}
        />
      )
    }
```

The rest of the class internals are the same, but we don't return this class from a `withInstrument()` function. Instead, we return a result of a `forwardRef()` function.

```
  return forwardRef<ComponentInstance, ProviderProps>(
    (props, ref) => <WithInstrument forwardedRef={ref} {...props} />
  )
```

Refs are not provided with the props. To get access to the `ref` object we call a special `forwardRef()` function.

We provide another anonymous function that returns our `WithInstrument` component as an argument for it. This function receives two arguments: `props`, the original props of a component, and a `ref`, the ref that should be forwarded.

And that's how we keep refs working in HOCs.

Static Composition

HOCs have another interesting use case. Imagine a situation where we don't need to change an instrument in runtime, and we want to specify it once. In this case, we don't really need the `instrument` property on a `WrappedKeyboard` component. Is there a way to define an instrument to load before we actually start rendering a component? Yes, there is! It is called static composition.

So far, we worked with, as they call it, dynamic composition, where arguments of functions (or props for components) were passed dynamically in runtime. However, we can create a HOC that "remembers" an argument and then uses it in runtime when rendering a component. Let's build one of those!

Again let's determine what the signature of such a HOC would look like. Create a file called `withInstrumentStatic.tsx` inside `src/adapters/Soundfont` and add the imports:

```
import React, { Component, ComponentType } from "react"
import Soundfont, { InstrumentName, Player } from "soundfont-player"
import { MidiValue } from "../../domain/note"
import { Optional } from "../../domain/types"
import {
  AudioNodesRegistry,
  DEFAULT_INSTRUMENT
} from "../../domain/sound"
```

Then, define the props:

```
type InjectedProps = {
  loading: boolean
  play(note: MidiValue): Promise<void>
  stop(note: MidiValue): Promise<void>
}

type ProviderProps = {
  AudioContext: AudioContextType
}

type ProviderState = {
  loading: boolean
  current: Optional<InstrumentName>
}
```

Then, create a function `withInstrumentStatic()` which takes an `instrument` as an argument. Our provider will load this instrument, and it won't change throughout the whole component life.

```
export function withInstrumentStatic<
  TProps extends InjectedProps = InjectedProps
>(initialInstrument: InstrumentName = DEFAULT_INSTRUMENT) {
```

Then, instead of returning a class, we return another function! This function is our original HOC which takes a `WrappedComponent` and returns a class `WithInstrument`.

```
  return function enhanceComponent(
    WrappedComponent: ComponentType<TProps>
  ) {
    const displayName =
      WrappedComponent.displayName ||
      WrappedComponent.name ||
      "Component"

    return class WithInstrument extends Component<
      ProviderProps,
      ProviderState
    > {
```

Then add the missing private and public fields:

```
      private audio: AudioContext
      private player: Optional<Player> = null
      private activeNodes: AudioNodesRegistry = {}

      public static displayName = `withInstrumentStatic(${displayName})`
      public state: ProviderState = {
        loading: false,
        current: null
      }
```

Add the constructor and the `componentDidMount()`:

```
    constructor(props: ProviderProps) {
      super(props)

      const { AudioContext } = this.props
      this.audio = new AudioContext()
    }

    public componentDidMount() {
      this.load(initialInstrument)
    }
```

Define the `resume()` method:

```
    private resume = async () => {
      return this.audio.state === "suspended"
        ? await this.audio.resume()
        : Promise.resolve()
    }
```

Define the public methods:

```
    public load = async (instrument: InstrumentName) => {
      this.setState({ loading: true })

      this.player = await Soundfont.instrument(
        this.audio,
        instrument
      )
      this.setState({ loading: false, current: instrument })
    }

    public play = async (note: MidiValue) => {
      await this.resume()
      if (!this.player) return

```

```
        const node = this.player.play(note.toString())
        this.activeNodes = { ...this.activeNodes, [note]: node }
      }

      public stop = async (note: MidiValue) => {
        await this.resume()
        if (!this.activeNodes[note]) return

        this.activeNodes[note]!.stop()
        this.activeNodes = { ...this.activeNodes, [note]: null }
      }
```

Define the `render()` method:

```
      public render() {
        const injected = {
          loading: this.state.loading,
          play: this.play,
          stop: this.stop
        } as InjectedProps

        return <WrappedComponent {...(injected as TProps)} />
      }
    }
  }
}
```

Re-export the `withInstrumentStatic` function from the `index.ts` file.

Okay, why would we create a function that returns a function that returns a class?.. Well, to answer this question, let's take look at the example usecase.

Create a file called `WithStaticInstrument.tsx` inside `src/components/Keyboard` and add the following code:

```
import { withInstrumentStatic } from "../../adapters/Soundfont/withInst\
rumentStatic"
import { useAudioContext } from "../AudioContextProvider"
import { Keyboard } from "../Keyboard"

// eslint-disable-next-line @typescript-eslint/no-unused-vars
const withGuitar = withInstrumentStatic("acoustic_guitar_steel")
const withPiano = withInstrumentStatic("acoustic_grand_piano")
const WrappedKeyboard = withPiano(Keyboard)

export const KeyboardWithInstrument = () => {
  const AudioContext = useAudioContext()!
  return <WrappedKeyboard AudioContext={AudioContext} />
}
```

Now, when we call the `withInstrumentStatic()` function, we don't get a component in return. We get another function that remembers an instrument that we want to connect to. So, we can create as many functions as we want beforehand and use them to connect components to Soundfont after!

Using Hooks with HOCs

Since HOCs are just functions that return components, they can be based on hooks. Create a file called `withInstrumentBasedOnHook.tsx` inside `src/adapters/Soundfont` and add the following code:

```
import { ComponentType, useEffect } from "react"
import { InstrumentName } from "soundfont-player"
import { MidiValue } from "../../domain/note"
import { useSoundfont } from "./useSoundfont"

type InjectedProps = {
  loading: boolean
  play(note: MidiValue): Promise<void>
  stop(note: MidiValue): Promise<void>
}

type ProviderProps = {
  AudioContext: AudioContextType
  instrument?: InstrumentName
}
```

And now, let's turn the hook component into HOC:

```
export const withInstrument = (
  WrappedComponent: ComponentType<InjectedProps>
) => {
  return function WithInstrumentComponent(props: ProviderProps) {
    const { AudioContext, instrument } = props
    const fromHook = useSoundfont({ AudioContext })
    const { loading, current, play, stop, load } = fromHook

    useEffect(() => {
      if (!loading && instrument !== current) load(instrument)
    }, [load, loading, current, instrument])

    return (
      <WrappedComponent loading={loading} play={play} stop={stop} />
    )
  }
}
```

We encapsulate sound sets' loading inside of `WithInstrumentComponent` and expose only `ProviderProps` to the outside. However, the logic of these components is based upon the functionality that `useSoundfont()` gives us.

Conclusion

Congratulations!

We have completed our piano keyboard, which can play the sounds of many instruments!

Most importantly, we now can solve problems with sharing logic and reducing duplications using different techniques such as *Render Props* and *Higher-Order Components*.

Using Redux and TypeScript

Introduction

When you work with React you usually end up with a state that is used globally across the whole application.

One of the approaches to sharing the state across the whole component tree is using the Context API[135]. You saw an example of this approach in the first chapter. There we used it in combination with the `useReducer` hook to manage the global application state.

This approach works, but it can only get you so far. In the end, you have to invent your own ways to manage the side-effects, debug your code, and split it into modules so it doesn't grow into a horrible incomprehensible mess.

A better idea is to use specialized tools. One such tool for managing the global application state is Redux.

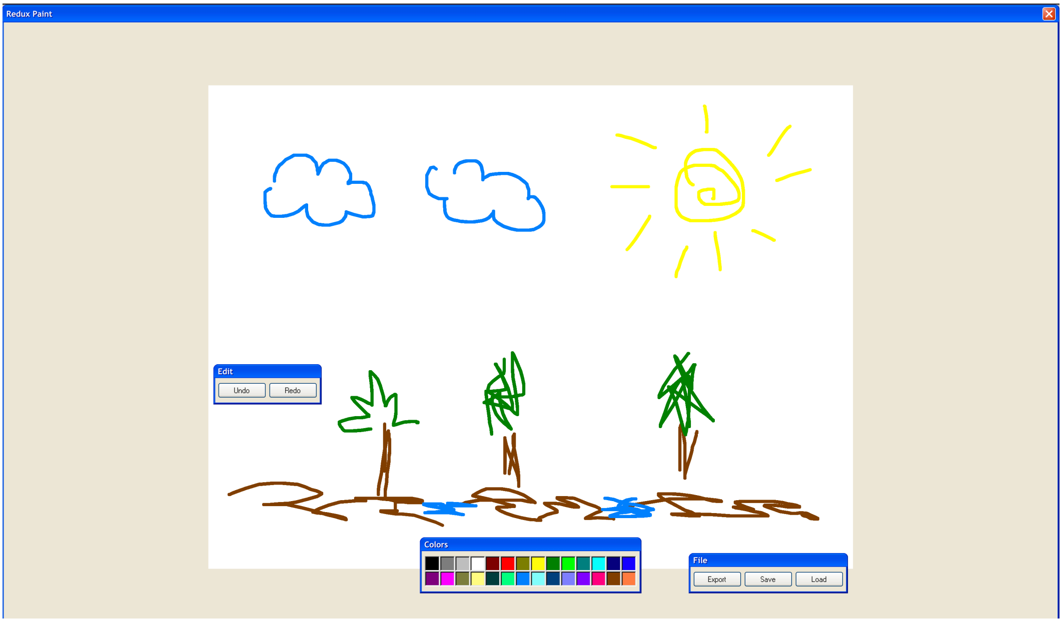
In this chapter, we build a drawing application using Redux with TypeScript and then we upgrade it to Redux Toolkit.

This way you will learn how to work with the raw Redux as well as the most modern techniques for using it.

Preview The Final Result

The application for this chapter is a drawing board.

[135]https://reactjs.org/docs/context.html



Completed application

You can pick different colors and draw lines. If you don't like the results you can "undo" some of the past actions. When you are satisfied with the results you can export the image as a `.png` file.

A complete code example is located in `code/04-redux/completed`.

Unzip the archive that comes with this book and `cd` to the app folder.

```
cd code/04-redux/completed
```

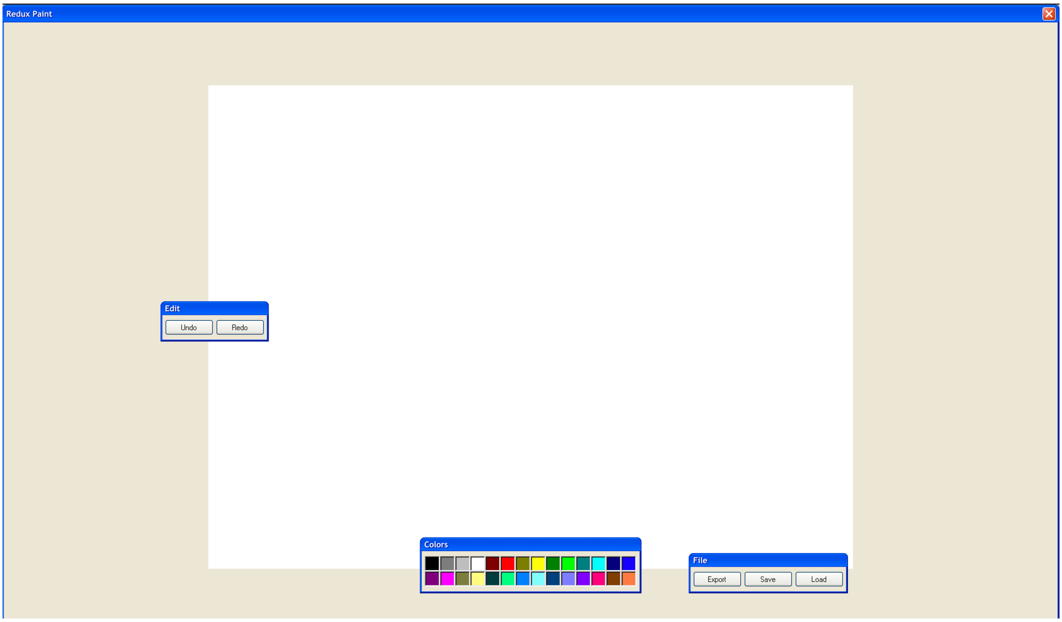
When you are there, install the dependencies and launch the app:

```
yarn && yarn dev
```

The `yarn dev` command will launch the app along with the backend script.

It should also open the app in the browser. If it doesn't, navigate to http://localhost:3000 and open it manually.

You should see an empty canvas and a color palette.

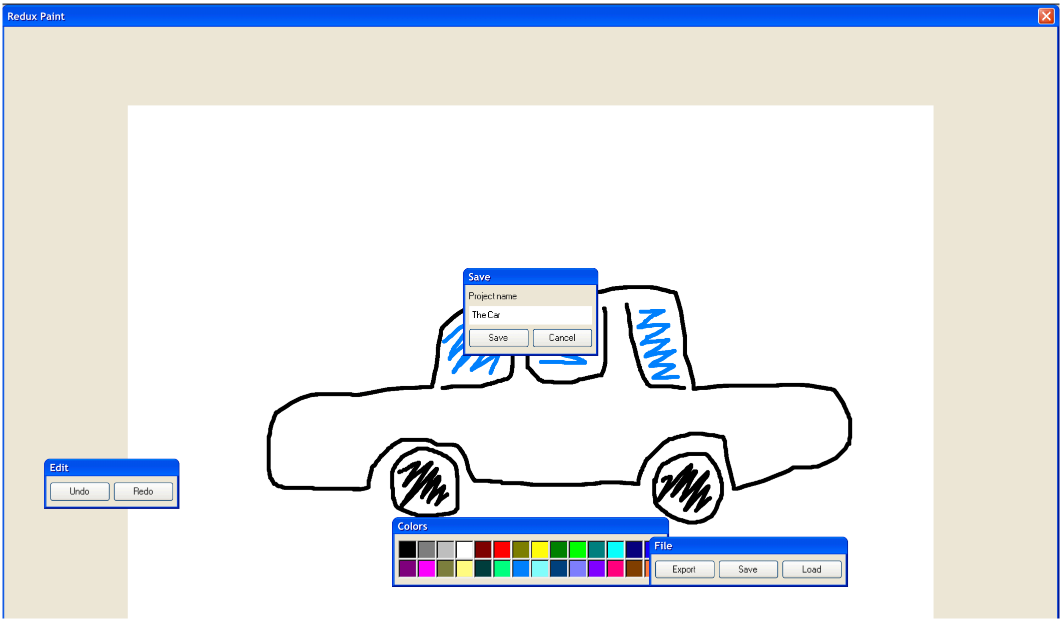


Empty canvas

Try drawing a few lines. You can pick different colors using the palette at the bottom.

If you don't like how some of the strokes turn out, click the *Undo* button. Click the *Redo* button to bring them back.

To save the project, press the *Save* button on the *File* panel. You should see the project-saving dialog.

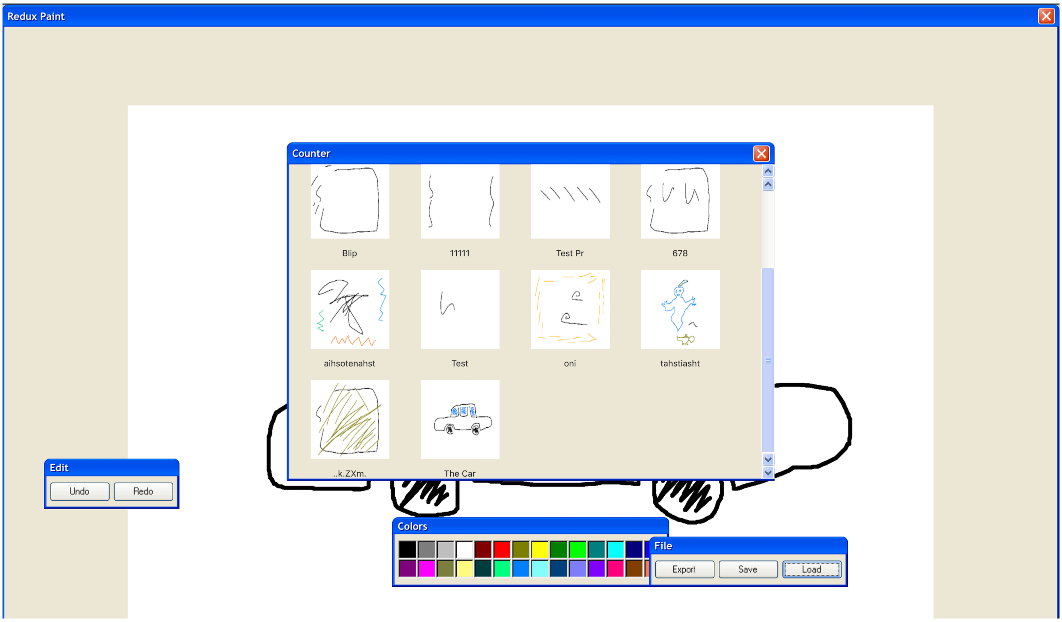


Saving the project

Pick a name for your project and press the *Save* button.

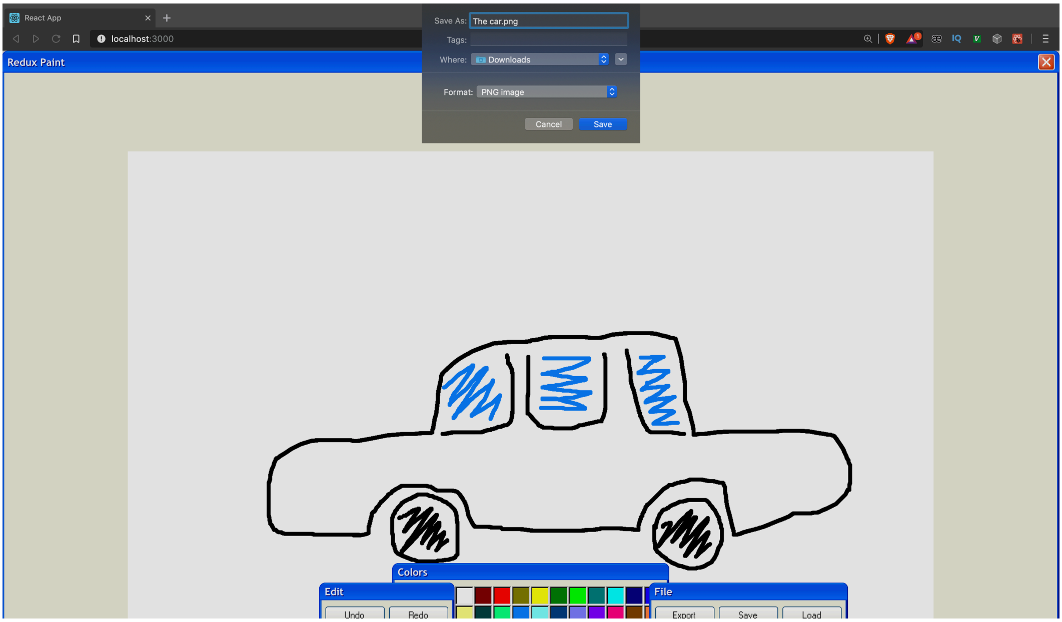
Now you can load this project and continue drawing. The changes in history will be preserved.

To do this press the *Load* button on the *File* panel.



Loading the project

You can also export your image to a file. To do this press the *Export* button.



Export to file

You should be presented with the file-saving dialog.

What is Redux?

Redux is a state management framework that is based on the idea of representing the global state of the application as a reducer function.

So to manage the state you would define a function that would accept two arguments: `state` - for the old state, and `action` - the object describing the state update.

```
function reducer(state = "", action: Action) {
  switch (action.type) {
    case "SET_VALUE":
      return action.payload
    default:
      return state
  }
}
```

This reducer represents one value of type `string`. It handles only one type of action: `SET_VALUE`.

If the received action field `type` is not `SET_VALUE`, the reducer will return the unchanged state.

After we have the reducer, we can create the store using the redux `createStore` method.

```
const store = createStore(reducer, "Initial Value")
```

The store provides a `subscribe` method that allows us to subscribe to the store updates.

```
store.subscribe(() => {
  const state = store.getState()
  console.log(state)
})
```

Here we've passed a callback to it that will log the state value to the console.

In order to update the state we'll need to `dispatch` an action:

```
store.dispatch({
  type: "SET_VALUE",
  payload: "New value"
})
```

Here we pass an object that represents the action. Every action is required to have the `type` field, and optionally a payload.

> Redux uses the Flux action format. Read more about it here[136]

Usually, instead of creating actions in place, people define action creator functions:

```
const setValue = (value) => ({
  type: "SET_VALUE",
  payload: value
})
```

And this is the essence of Redux.

You can find the example with everything set up in the `/code/04-redux/redux-example` folder.

Install the dependencies and run the script using `yarn run`:

```
yarn && yarn start
```

You should see the following output:

```
New value
```

Try dispatching more actions.

[136] https://github.com/redux-utilities/flux-standard-action

Why Can't We Use useReducer Instead of Redux?

Since version 16.8, React supports Hooks. One of them, `useReducer`, works in a very similar way to Redux.

> In the first chapter of this book we created an application managing the application state using a combination of `useReducer` and React Context API.
>
> If you need a refresher, you can find a `useReducer` example in the `/code/01-first-app/use-reducer` folder.

So why do we need Redux if we have a native tool that allows us to represent the state as a reducer as well? If we make it available across the application using the Context API, won't that be enough?

Redux provides a bunch of important advantages:

Browser Tools. You can use Redux DevTools[137] to debug your Redux code. It allows us to see the list of dispatched actions, inspect the state, and even time-travel. You can switch back and forth in the action history and see how the state looked after each of them.

Handling Side Effects. With `useReducer` you have to invent your own ways to organize the code that performs network requests. Redux provides the middleware API[138] to handle that. Also, there are tools like Redux Thunk[139] that make this task even easier.

Testing. As Redux is based on pure functions it is easy to test. All the tests boil down to checking the output with the given inputs.

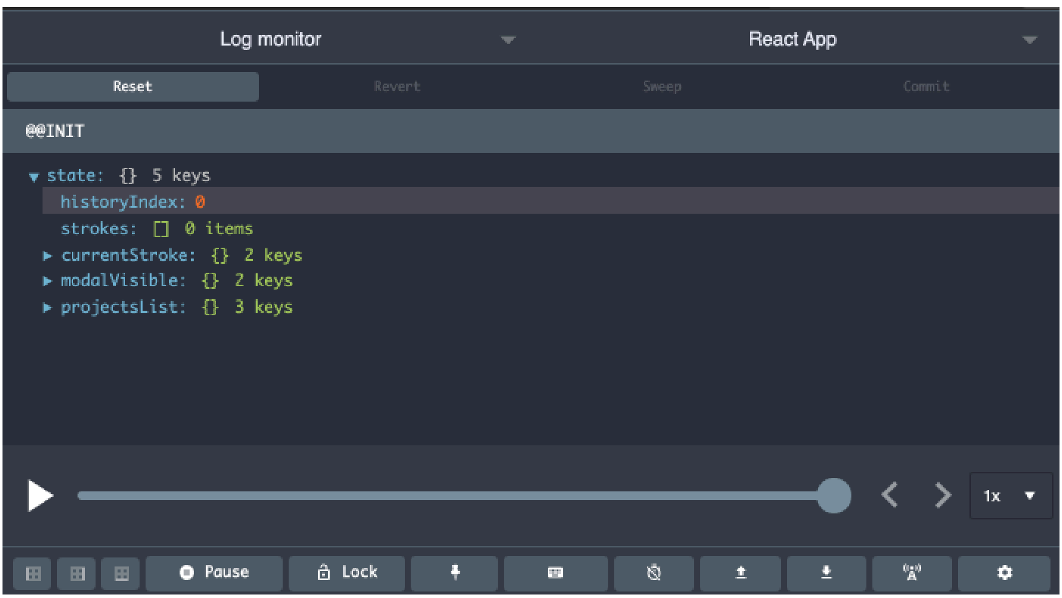
Patterns and Code Organization. Redux is well-studied and there are recipes for most of the problems. There is a methodology called Ducks[140] that you can use to organize the Redux code.

[137] https://github.com/reduxjs/redux-devtools
[138] https://redux.js.org/advanced/middleware
[139] https://github.com/reduxjs/redux-thunk
[140] https://github.com/erikras/ducks-modular-redux

Initial Setup

First, let's prepare the browser. Download Redux DevTools for your browser. There are extensions for Chrome[141] and Firefox[142].

After you install the extension you should see the *Redux DevTools* button on your browser tools panel. Try clicking this button on the page with the completed project running. You should see this:



Redux DevTools

Create The Project

After that is done let's create the project. Run `create-react-app` with the `--template typescript`:

```
npx create-react-app --template typescript redux-paint
```

After the generation is complete, go to the project folder and install the dependencies:

[141] https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd
[142] https://addons.mozilla.org/en-US/firefox/addon/reduxdevtools/

```
yarn add redux react-redux @types/react-redux
```

For Redux to work with React we need to install the `react-redux` adapter package.

Redux is written in Typescript so you don't have to install the additional types for it, but we do need to install the types for `react-redux`.

Now let's set up Redux in our application.

Create a new file `src/rootReducer.ts` and define our initial reducer there:

```
type RootState = {}

type Action = {
  type: string
}

export const rootReducer = (
  state: RootState = {},
  action: Action
) => {
  return state
}
```

We temporarily define the `RootState` to be an empty object and the `Action` to have the type field that can be any `string`. We'll use those types only to make sure that our setup works, and then we'll define the real `RootState` and `Action` types.

The reducer is not doing much just yet. For now, it returns the initial state on any dispatched action.

Install the `redux-devtools-extension`:

```
yarn add redux-devtools-extension
```

Create a new file `src/store.ts` and initialize the Redux store there.

```
import { rootReducer } from "./rootReducer"
import { devToolsEnhancer } from "redux-devtools-extension"
import { createStore } from "redux"

export const store = createStore(rootReducer, devToolsEnhancer({}))
```

Here we create and export a new store instance. We pass two arguments to it: our reducer, from the previous step, and the Redux DevTools middleware.

> *Middlewares* are functions that get triggered on each action dispatch. They are used to perform side-effects: making network requests, logging, writing data to storage. Each middleware function has access to the current action and the store and can dispatch new actions. Read more about the middlewares in the Redux documentation.[143]

Then go to `src/index.tsx` and import `Provider` from `react-redux` and `store` from `src/store.ts`:

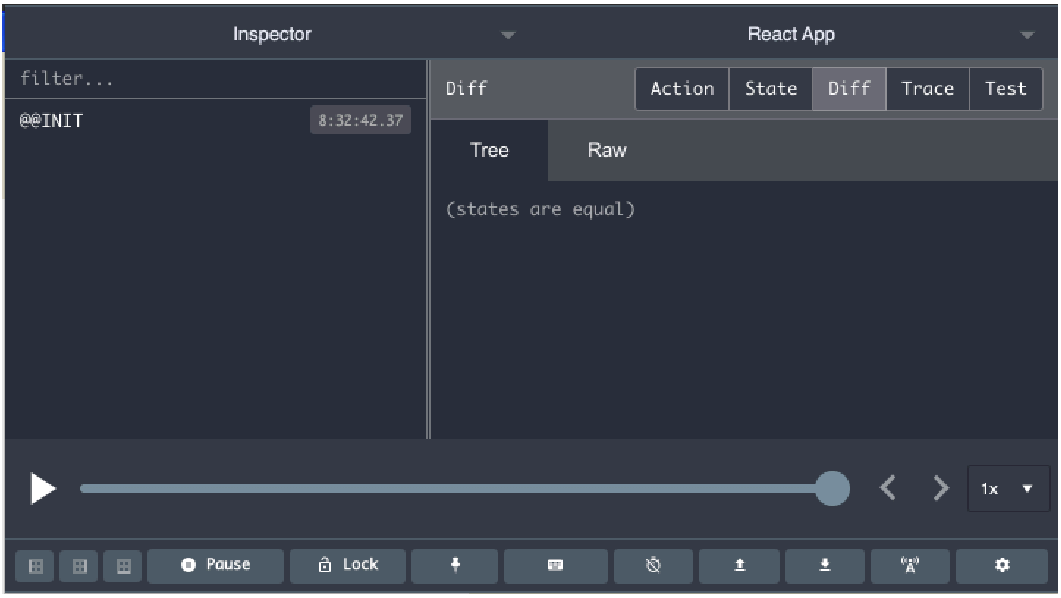
```
import { Provider } from "react-redux"
import { store } from "./store"
```

Wrap your `App` component into the `Provider`, and pass the `store` instance to it:

```
ReactDOM.render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>,
  document.getElementById("root")
)
```

Now launch the app and open it in the browser. If you click on the Redux DevTools button in the toolbar, you should see this:

[143] https://redux.js.org/advanced/middleware



Redux DevTools

Redux Logger

Redux DevTools are cool, but some people, including me, prefer to have a quicker way to observe what is happening inside their Redux application.

Install `redux-logger`:

```
yarn add redux-logger @types/redux-logger
```

Add `redux-logger` to the middlewares list in the store. Open `src/store.ts` and make it look like this:

```
import { rootReducer } from "./rootReducer"
import { createStore, applyMiddleware } from "redux"
import { composeWithDevTools } from "redux-devtools-extension"
import { logger } from "redux-logger"

export const store = createStore(
  rootReducer,
  composeWithDevTools(applyMiddleware(logger))
)
```

Here we use the `composeWithDevTools` method from the `redux-devtools-extension` to add it to the middlewares list.

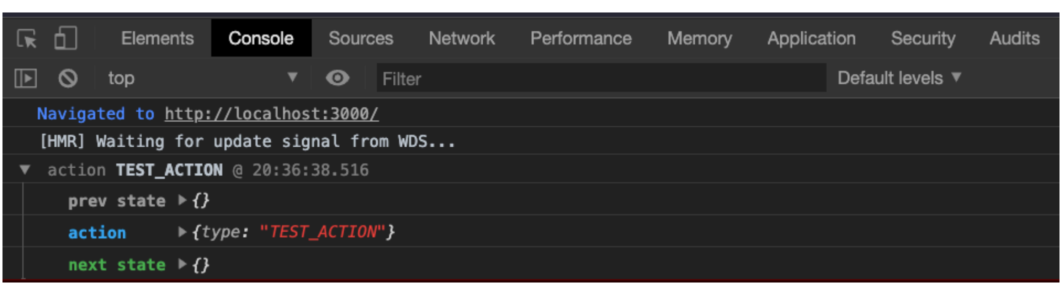
> Read more about applying middlewares to your Redux store in the Redux Documentation[144]

Temporarily add the following code to dispatch an action:

```
store.dispatch({type: "TEST_ACTION"})
```

> Don't forget to remove this line after you verify that Redux-Logger works.

Now open the browser and open the console. If everything is set up correctly you should see this:



Redux Logger output

The Redux Logger output consists of three parts:

[144] https://redux.js.org/advanced/middleware#the-final-approach

- `prev state` - the state before the dispatched action
- `action` - dispatched action
- `next state` - the state after the dispatched action

You can expand each of the parts to see the details.

I find it more convenient when I can see all the actions that are happening in the application along with the other logs.

Prepare The Styles

We are going to use XP.css[145] by Adam Hammad[146] for our styles.

Install it:

```
yarn add xp.css
```

And import it in `src/index.css`:

```
@import "~xp.css/dist/XP.css";
```

Let's also add icons. Copy them from the completed project folder `code/04-redux/completed/src` You need to create a similar folder in your project.

Update the App layout

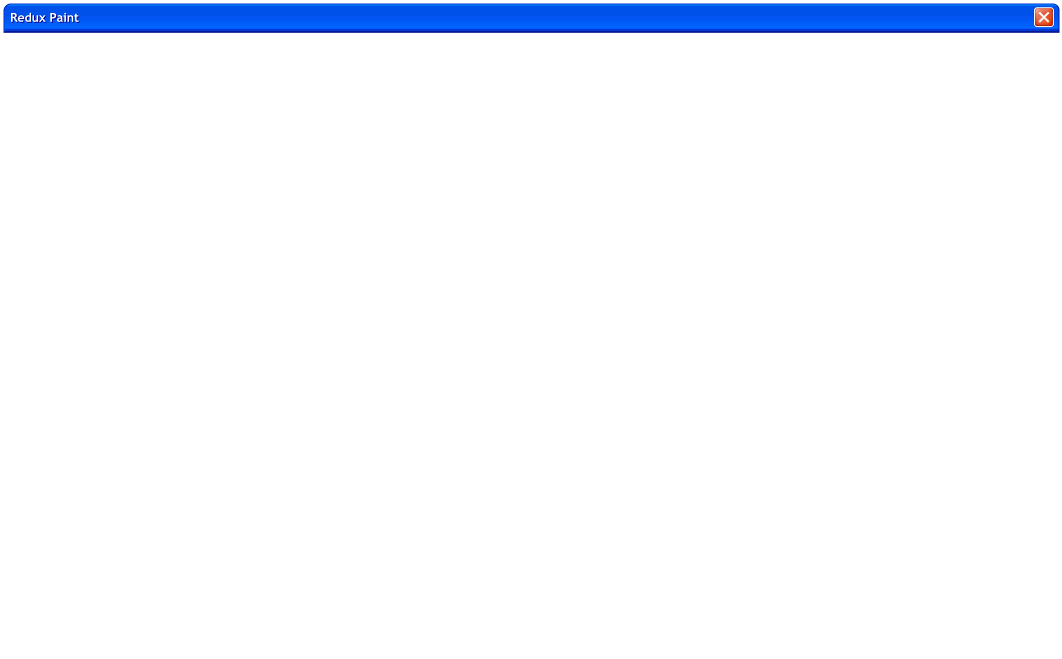
Open the `src/App.tsx` file and change the layout:

[145] https://botoxparty.github.io/XP.css/
[146] https://github.com/botoxparty

```
import React from "react"

function App() {
  return (
    <div className="window">
      <div className="title-bar">
        <div className="title-bar-text">Redux Paint</div>
        <div className="title-bar-controls">
          <button aria-label="Close" />
        </div>
      </div>
    </div>
  )
}

export default App
```

Here we've added a bunch of wrapper elements to make our app look like a Window XP window.

If you launch your app - it should look like this:



App with Windows XP styles

Working With Canvas

We will use the Canvas API[147] to handle drawing.

First of all add the following rules to the `src/index.css` file:

```
canvas {
  transform: translate3d(-50%, 0, 0);
  cursor: url(./icons/pencil.png) 0 34, auto;
  margin: 100px 50%;
}
```

Here we defined the styles that will position the `canvas` element in the center of the screen and make the cursor look like a pencil when the user hovers the canvas.

[147] https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API

You'll need to copy the pencil icon from the `completed/src/icons` folder. Create same folder in `src` and copy the icon there.

Now let's define some utility functions. We'll need a function to set the initial styles for the canvas, and a function to clear the canvas.

Create a new folder `src/utils` and inside of it a new file `src/utils/canvasUtils.ts` with the following contents:

```
export const clearCanvas = (canvas: HTMLCanvasElement) => {
  const context = canvas.getContext("2d")
  if (!context) {
    return
  }
  context.fillStyle = "white"
  context.fillRect(0, 0, canvas.width, canvas.height)
}
```

Here we defined a function that will fill the canvas with white color. We'll use it to clear the canvas when we, for example, undo the strokes.

In the same file define the `setCanvasSize` function:

```
export const setCanvasSize = (
  canvas: HTMLCanvasElement,
  width: number,
  height: number
) => {
  canvas.width = width * 2
  canvas.height = height * 2
  canvas.style.width = `${width}px`
  canvas.style.height = `${height}px`
  canvas.getContext("2d")?.scale(2, 2)
}
```

Here we adjust the canvas for retina screen bu setting the double pixel density.

Go to `src/App.tsx` and import the utility functions:

```
import { clearCanvas, setCanvasSize } from "./utils/canvasUtils"
```

We also need to import he `useEffect` and `useRef` hooks from React:

```
import React, { useRef, useEffect } from "react"
```

We'll use the `useRef` to hold the reference to our `canvas` element and the `useEffect` to prepare the canvas for drawing when we open the app.

Outside of the `App` component define the canvas size constants:

```
const WIDTH = 1024
const HEIGHT = 768
```

Now we'll need to get and store the reference to the canvas:

```
function App() {
  const canvasRef = useRef<HTMLCanvasElement>(null)
  // ...
  return (
    <div className="window">
      <div className="title-bar">
        <div className="title-bar-text">Redux Paint</div>
        <div className="title-bar-controls">
          <button aria-label="Close" />
        </div>
      </div>
      <canvas ref={canvasRef} />
    </div>
  )
}
```

Here we create a `ref` object that will hold the reference to our canvas using the `useRef` hook.

We need to specify the type of value we'll store in the `ref` object. We know that it is a `canvas` - so we pass the `HTMLCanvasElement` as a *type variable*.

We also need to pass `null` as the default value to the `useRef` hook. Otherwise, you'll get a type error stating that the `ref` prop of the `canvas` element does not accept `undefined`.

> You can remove the `src/App.css` and the `src/logo.svg` files, we are not going to use them.

We have the reference to the `canvas` element, now we need to get the canvas context. Let's define a helper function that will do it for us. Right after the call to `useRef` hook define a new function:

```
const getCanvasWithContext = (canvas = canvasRef.current) => {
  return { canvas, context: canvas?.getContext("2d") }
}
```

This function will allow us to get both the canvas and it's 2d context in one function call.

Now let's add the side effect that will be executed when we mound the `App` component:

```
  useEffect(() => {
    const { canvas, context } = getCanvasWithContext()
    if (!canvas || !context) {
      return
    }

    setCanvasSize(canvas, WIDTH, HEIGHT)

    context.lineJoin = "round"
    context.lineCap = "round"
    context.lineWidth = 5
    context.strokeStyle = "black"

    clearCanvas(canvas)
  }, [])
```

Here we set the canvas side to the predefined values, we set the strokes style and then we clear the canvas, preparing it for the first strokes.

Handling Canvas Events

We want to handle the following situations:

- The user pressed the mouse button
- The user moved the mouse
- The user released the mouse button
- The cursor left the canvas area

Define event handlers inside of the `App` component body:

```
const startDrawing = () => {}
const endDrawing = () => {}
const draw = () => {}
```

After you have the functiones defined pass them to the `canvas` element:

```
<canvas
  onMouseDown={startDrawing}
  onMouseUp={endDrawing}
  onMouseOut={endDrawing}
  onMouseMove={draw}
  ref={canvasRef}
/>
```

Now we'll handle every press, move, or release of the mouse that happens above the `canvas` element.

Define The Store Types

Create a new file `src/utils/types.ts`.

Inside this file let's define the type for our state:

```
export type RootState = {
  currentStroke: Stroke
  strokes: Stroke[]
}
```

It contains three fields:

- `currentStroke` - an array of points corresponding to the stroke that is currently being drawn.
- `strokes` - an array of already drawn strokes
- `historyIndex` - a number indicating how many of the strokes we want to undo.

Let's define the `Stroke` type:

```
export type Stroke = {
  points: Point[]
  color: string
}
```

Each stroke has a `color` represented as a hex string and a list of points, where each point is an object that holds the `x` and `y` coordinates.

Define the `Point` type:

```
export type Point = {
  x: number
  y: number
}
```

Points contain the vertical and horizontal coordinates.

Add Actions

Create a new file `src/actions.ts` and define the following types constants for actions:

```
export const BEGIN_STROKE = "BEGIN_STROKE"
export const UPDATE_STROKE = "UPDATE_STROKE"
export const END_STROKE = "END_STROKE"
```

- `BEGIN_STROKE` - we'll dispatch this action when the user presses the mouse button. It will contain the coordinates in the payload.
- `UPDATE_STROKE` - this action will be dispatched when the user moves the pressed mouse. It also contains the coordinates.
- `END_STROKE` - we'll dispatch this action when the user releases the mouse.

Import the `Point` type from the `src/utils/types.ts`:

```
import { Point } from "./utils/types"
```

Define the `Action` type:

```
export type Action =
  | {
      type: typeof BEGIN_STROKE
      payload: Point
    }
  | {
      type: typeof UPDATE_STROKE
      payload: Point
    }
  | {
      type: typeof END_STROKE
    }
```

Here we pass a `Point` as a payload for the `BEGIN_STROKE` and the `UPDATE_STROKE` actions. We need to know the coordinates of the mouse when the user started the stroke, and then we need to update the coordinates on a mouse move.

We don't pass the coordinates with the `END_STROKE` action because the mouse was moved there first.

Define the action creators for each action:

```
export const beginStroke = (x: number, y: number) => {
  return { type: BEGIN_STROKE, payload: { x, y } }
}

export const updateStroke = (x: number, y: number) => {
  return { type: UPDATE_STROKE, payload: { x, y } }
}

export const endStroke = () => {
  return { type: END_STROKE }
}
```

Add The Reducer Logic

Go to `src/rootReducer.ts`. Import the `RootState` from `src/types.d.ts` and `Action` types from the `src/actions.ts`.

```
import {
  Action,
  UPDATE_STROKE,
  BEGIN_STROKE,
  END_STROKE
} from "./actions"
import { RootState } from "./utils/types"
```

Then we define the initial state:

```
const initialState: RootState = {
  currentStroke: { points: [], color: "#000" },
  strokes: []
}
```

Remake the `rootReducer` to this:

```
export const rootReducer = (
  state: RootState = initialState,
  action: Action
) => {
  switch (action.type) {
  // ...
    default:
      return state
  }
}
```

Now let's add the logic to process the existing actions.

We'll start with the `BEGIN_STROKE` action. Add the following code inside the `switch`:

```
case BEGIN_STROKE: {
  return {
    ...state,
    currentStroke: {
      ...state.currentStroke,
      points: [action.payload]
    }
  }
}
```

On every `BEGIN_STROKE` action, we set the `points` to be a new array with the point from the `action.payload`.

Then process the `UPDATE_STROKE` action:

```
case UPDATE_STROKE: {
  return {
    ...state,
    currentStroke: {
      ...state.currentStroke,
      points: [...state.currentStroke.points, action.payload]
    }
  }
}
```

> If you feel a bit shaky on the three dots ... everywhere, it may be helpful to refresh yourself on the Immutable Patterns in Redux[148]. The basic idea is that we're trying to deeply update an object, without overwriting the existing values.

Here we update the `currentStroke` field of our state by appending a new point from the `action.payload` to it.

The last action for now is `END_STROKE`:

```
case END_STROKE: {
  if (!state.currentStroke.points.length) {
    return state
  }
  return {
    ...state,
    currentStroke: { ...state.currentStroke, points: [] },
    strokes: [...state.strokes, state.currentStroke]
  }
}
```

The `END_STROKE` action can be dispatched when the mouse leaves the canvas. It may result in calling the `END_STROKE` part of the reducer to trigger before the `currentStroke` has any points.

[148] https://redux.js.org/recipes/structuring-reducers/immutable-update-patterns

To prevent unnecessary calculations we return the unchanged state if the `currentStroke.points` array is empty.

If there are any points, we append the current stroke to the list of strokes and reset the `currentStroke.points` to the empty array.

Dispatch Actions

In `src/App.tsx`, import the `useDispatch` and `useSelector` from `react-redux`:

```
import { useSelector, useDispatch } from "react-redux"
```

Import React, we are going to use the events types from it:

```
import React, { useRef, useEffect } from "react"
```

Import the action types that we are going to dispatch:

```
import { beginStroke, endStroke, updateStroke } from "./actions"
```

We are going to need a flag that will tell us that we are currently drawing a stroke. We know that we've started drawing if there is at least one point in the current stroke points array. So we can calculate it by converting the current stroke points array length to a boolean.

Define this flag below the `getCanvasWithContext` function:

```
const isDrawing = useSelector<RootState>(
  (state) => !!state.currentStroke.points.length
)
```

Here we used the `useSelector` hook. This hook is generic and you can provide the state and the return value types. In our case we specified the type of the state as the `RootState`. You need to import thas type:

```
import { RootState } from "./utils/types"
```

Get the `dispatch` function from the `useDispatch` - add this line after the `useSelector` call:

```
const dispatch = useDispatch()
```

Now let's edit the mouse press event handler. Make it dispatch the `BEGIN_STROKE` action.

```
const startDrawing = ({
  nativeEvent
}: React.MouseEvent<HTMLCanvasElement>) => {
  const { offsetX, offsetY } = nativeEvent
  dispatch(beginStroke(offsetX, offsetY))
}
```

Here we get the `nativeEvent` field from the `event` object.

> React normalizes the events using the SyntheticEvent[149] wrapper. It is done to improve cross-browser compatibility.

We get the mouse coordinates from the `offsetX` and `offsetY` fields of the `nativeEvent` and pass them with the action.

In our app we handle the mouse move event in the `draw` handler. Define it like this:

[149] https://reactjs.org/docs/events.html

```
  const draw = ({
    nativeEvent
  }: React.MouseEvent<HTMLCanvasElement>) => {
    if (!isDrawing) {
      return
    }
    const { offsetX, offsetY } = nativeEvent

    dispatch(updateStroke(offsetX, offsetY))
  }
```

To verify that the mouse is pressed we check the `isDrawing` flag. If the mouse is moved while pressed, we dispatch the `UPDATE_STROKE` action with the updated coordinates.

Now, we want to stop drawing when we release the button. Update the mouse up and mouse out event handler:

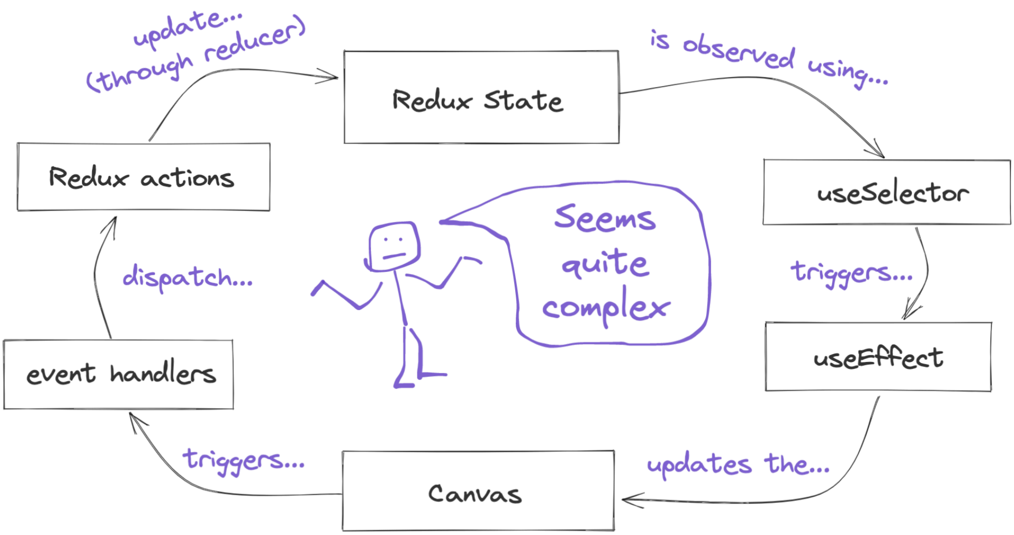
```
const endDrawing = () => {
  if (isDrawing) {
    dispatch(endStroke())
  }
}
```

In this function we dispatch the `END_STROKE` action.

The `endDrawing` function will also trigger when the mouse leaves the canvas area. This is why here we also check the `isDrawing` flag and dispatch the `endStroke` action only if we were drawing a stroke.

Draw The Current Stroke

Our app has a certain level of indirectness. Instead of updating the canvas directly in reaction of mouse events we dispatch Redux actions. The actions trigger state updates. We observe the state changes and when they happen - we draw the storokes on the canvas.



Update cycle

That seems quite complex, but in return we get an ability to undo the strokes.

First of all let's define the `drawStroke` method in a separate module. Create a new file `src/utils/canvasUtils.ts` and import `Point` from the `types` module:

```
import { Point } from "./types"
```

Now define and export the `drawStroke` method:

```
export const drawStroke = (
  context: CanvasRenderingContext2D,
  points: Point[],
  color: string
) => {
  if (!points.length) {
    return
  }
  context.strokeStyle = color
  context.beginPath()
```

```
  context.moveTo(points[0].x, points[0].y)
  points.forEach((point) => {
    context.lineTo(point.x, point.y)
    context.stroke()
  })
  context.closePath()
}
```

This function receives the context that it will use for drawing, the list of points for the current stroke and the stroke color. We check that the points array is not empty and we have something to draw. Then we set the `context.strokeStyle` to the color value passed through the arguments.

After that is done, we call the `beginPath` method. We create a separate path for each stroke so that they can all have different colors.

Next, we move to the first point in the array using the `moveTo` method. We don't draw anything yet.

Then we go through the list of points and connect them with the lines using the `lineTo` method. This method updates the current path but doesn't render anything. The actual drawing happens when we call the `stroke` method. It renders the outline along the drawn line.

After we finish drawing the stroke we need to call the `closePath` method.

Define the `currentStrokeSelector`

It is a good idea to define the selectors outside of the component. This way the component won't be tightly coupled with the state structure and the selector will be easy to reuse.

Open `src/rootReducer.ts` and define and export the `currentStrokeSelector`:

```
export const currentStrokeSelector = (state: RootState) =>
  state.currentStroke
```

Update the `App` component

Now let's observe the state and render the strokes on the canvas. Open `src/App.tsx`. Import the `currentStrokeSelector` that you've just defined:

```
import { currentStrokeSelector } from "./rootReducer"
```

Define the `currentStroke` constant in the beginnging of the `App` component body:

```
const currentStroke = useSelector(currentStrokeSelector)
```

Now we can also update the `isDrawing`, we'll calculate it using the `currentStroke` constant tha we've just defined:

```
const isDrawing = !!currentStroke.points.length
```

Import the `useEffect` from `react`, and the `drawStroke` function from `./canvasUtils`:

```
import React, { useRef, useEffect } from "react"
  // ...
import { drawStroke, clearCanvas, setCanvasSize } from "./utils/canvasU\
tils"
```

Define the side-effect to handle the `currentStroke` updates.

```
useEffect(() => {
  const { context } = getCanvasWithContext()
  if (!context) {
    return
  }
  requestAnimationFrame(() =>
    drawStroke(context, currentStroke.points, currentStroke.color)
  )
}, [currentStroke])
```

Here we get the drawing context using the `getCanvasWithContext` function. Then we call the `drawStroke` method and pass the drawing context there. We also pass the `currentStroke` points and the color. At this point, you should be able to draw the strokes. Launch your application and try to draw something.

Redux Paint Application

Implement Selecting Colors

Right now we can only draw black strokes. Let's add a color panel and make it possible to select the stroke colors.

First let's define the styles, open the `src/index.css` and add the following CSS classes:

```css
.colors {
  display: flex;
  flex-direction: row;
  flex-wrap: wrap;
  width: 336px;
}

.color {
  width: 24px;
  height: 24px;
  cursor: pointer;
  box-shadow: inset -1px -1px #fff, inset 1px 1px grey,
    inset -2px -2px #dfdfdf, inset 2px 2px #0a0a0a;
}

.colors-panel {
  position: fixed;
  bottom: 40px;
  left: 50%;
  transform: translate3d(-50%, 0, 0);
  z-index: 10;
}
```

To be able to select the color, we need to add a new action and reducer block for it.

Open `src/actions.ts` and add a new action type:

```
export const SET_STROKE_COLOR = "SET_STROKE_COLOR"
```

Expand the `Action` type definition with this block:

```
| {
    type: typeof SET_STROKE_COLOR
    payload: string
  }
| {
```

And then add a new action creator:

```
export const setStrokeColor = (color: string) => {
  return { type: SET_STROKE_COLOR, payload: color }
}
```

After we are done with the actions go to `src/rootReducer.ts` and import the `SET_STROKE_COLOR` action:

```
import {
  Action,
  UPDATE_STROKE,
  BEGIN_STROKE,
  END_STROKE,
  SET_STROKE_COLOR,
} from "./actions"
```

Add a new reducer block:

```
case SET_STROKE_COLOR: {
  return {
    ...state,
    currentStroke: {
      ...state.currentStroke,
      ...{ color: action.payload }
    }
  }
}
```

Here we get the `color` value from the `action.payload` and update the `currentStroke` with this value.

Now let's add a color picker component.

Create a new file `src/shared/ColorPanel.tsx`. First we need to import `React`, `useDispatch`, and `setStrokeColor` action:

```
import { useDispatch } from "react-redux"
import { setStrokeColor } from "../actions"
```

Define the list of colors:

```
const COLORS = [
  "#000000",
  "#808080",
  "#c0c0c0",
  "#ffffff",
  // ...
]
```

Here we show only a few colors from the list. Copy the full list from the file `code/04-redux/completed/src/shared/ColorPanel.tsx`.

Now define the component:

```
export const ColorPanel = () => {
  // ...
  return (
    <div className="window colors-panel">
      <div className="title-bar">
        <div className="title-bar-text">Colors</div>
      </div>
      <div className="window-body colors">
        {COLORS.map((color: string) => (
          <div
            key={color}
            onClick={() => {
              onColorChange(color)
            }}
            className="color"
            style={{ backgroundColor: color }}
          ></div>
        ))}
      </div>
    </div>
  )
}
```

Here, when we click on the color block we call the `onColorChange` function. This function will dispatch the `SET_STROKE_COLOR` action.

Inside the component, get the `dispatch` method using `useDispatch` and define the `onColorChange` method:

```
const dispatch = useDispatch()
// ...
const onColorChange = (color: string) => {
  dispatch(setStrokeColor(color))
}
```

Then go to `src/App.tsx` and import the `ColorPanel` component.

```
import { ColorPanel } from "./shared/ColorPanel"
```

Update the `App` component layout to look like this:

```
<div className="window">
  <div className="title-bar">
    <div className="title-bar-text">Redux Paint</div>
    <div className="title-bar-controls">
      <button aria-label="Close" />
    </div>
  </div>
  <ColorPanel />
  <canvas
    onMouseDown={startDrawing}
    onMouseUp={endDrawing}
    onMouseOut={endDrawing}
    onMouseMove={draw}
    ref={canvasRef}
  />
</div>
```

Launch the app.

Picking the colors

You should now be able to select colors.

Implement Undo and Redo

Now let's implement the *undo/redo* functionality. To do this will add a `historyIndex` field to our state. This field will keep track of the current *undo* level. We'll use it's value in the `App` component to only render the strokes that were not undone.

Update the RootState type

Open the `src/utils/types.ts` and update the `RootState` definition:

```
export type RootState = {
  currentStroke: Stroke
  strokes: Stroke[]
  historyIndex: number
}
```

Create actions

Now let's define the actions, open `src/actions.ts` and add the `UNDO` and `REDO` actions. First define the constants for their types:

```
export const UNDO = "UNDO"
export const REDO = "REDO"
```

Update the `Action` type:

```
| {
    type: typeof UNDO
  }
| {
    type: typeof REDO
  }
```

Define the action creators:

```
export const undo = () => {
  return { type: UNDO }
}
  // ...
export const redo = () => {
  return { type: REDO }
}
```

Update the reducer

Open the `src/rootReducer.ts` file and import the `UNDO` and `REDO` action types:

```
import {
  Action,
  UPDATE_STROKE,
  BEGIN_STROKE,
  END_STROKE,
  SET_STROKE_COLOR,
  UNDO,
  REDO
} from "./actions"
```

Now let's update the initial state:

```
const initialState: RootState = {
  currentStroke: { points: [], color: "#000" },
  strokes: [],
  historyIndex: 0
}
```

Add the `UNDO` and `REDO` action handlers:

```
case UNDO: {
  const historyIndex = Math.min(
    state.historyIndex + 1,
    state.strokes.length
  )
  return { ...state, historyIndex }
}
case REDO: {
  const historyIndex = Math.max(state.historyIndex - 1, 0)
  return { ...state, historyIndex }
}
```

Here we update the `historyIndex` field making sure that it's value is always bigger than zero and smaller than the amount of drawn strokes. This way we ensure that we don't undo strokes that weren't drawn yet, and also we don't redo beyond what was previously undone.

We'll also need to update the `END_STROKE` action handler, now it will have to reset the history index:

```
case END_STROKE: {
  if (!state.currentStroke.points.length) {
    return state
  }
  const historyIndex = state.strokes.length - state.historyIndex
  return {
    ...state,
    historyIndex: 0,
    currentStroke: { ...state.currentStroke, points: [] },
    strokes: [
      ...state.strokes.slice(0, historyIndex),
      state.currentStroke
    ]
  }
}
```

This way, we avoid time-travel paradoxes. When we undo the strokes, we travel to the past. If, while being in the past, you draw a new stroke - the past get's altered, which makes it impossible to return to our original version of the present. Instead of creating a multiverse of paintings we just cut off the branch of history that was undone.

To make it easier to access the data from our state let's define the selectors.

```
export const historyIndexSelector = (state: RootState) =>
  state.historyIndex
  // ...
export const strokesSelector = (state: RootState) => state.strokes
```

> Some people prefer to define selectors in a separate file. I find it more useful to hold them closer to the reducer, because the reducer and selectors are likely to change together.

Create the EditPanel component

Let's add a panel with the **Undo** and **Redo** buttons. Open `src/index.css` and define a new CSS class `.edit`:

```css
.edit {
  position: fixed;
  bottom: 40px;
  left: 30%;
  z-index: 10;
}
```

Create a new file `src/shared/EditPanel.tsx`. Import `React`, `useDispatch` and the undo/redo actions:

```
import React from "react"
import { useDispatch } from "react-redux"
import { undo, redo } from "../actions"
```

Then define the `EditPanel` component:

```
export const EditPanel = () => {
  // ...
}
```

Get the `dispatch` function using the `useDispatch` hook from `react-redux`.

```
export const EditPanel = () => {
  const dispatch = useDispatch()
  // ...
}
```

Define the component layout:

```
<div className="window edit">
  <div className="title-bar">
    <div className="title-bar-text">Edit</div>
  </div>
  <div className="window-body">
    <div className="field-row">
      <button
        className="button redo"
        onClick={() => dispatch(undo())}
      >
        Undo
      </button>
      <button
        className="button undo"
        onClick={() => dispatch(redo())}
      >
        Redo
      </button>
    </div>
  </div>
</div>
```

The buttons should dispatch the `UNDO` and `REDO` actions:

```
          <button
            className="button redo"
            onClick={() => dispatch(undo())}
          >
            Undo
          </button>
          <button
            className="button undo"
            onClick={() => dispatch(redo())}
          >
            Redo
          </button>
```

Open `src/App.tsx` and import the `EditPanel` component:

```
import { EditPanel } from "./shared/EditPanel"
```

Add the `EditPanel` to the `App` layout:

```
<div className="window">
  <div className="title-bar">
    <div className="title-bar-text">Redux Paint</div>
    <div className="title-bar-controls">
      <button aria-label="Close" />
    </div>
  </div>
  <EditPanel />
  <ColorPanel />
  <canvas
    onMouseDown={startDrawing}
    onMouseUp={endDrawing}
    onMouseOut={endDrawing}
    onMouseMove={draw}
    ref={canvasRef}
  />
</div>
```

The new element should be right above the `ColorPanel`.

We also need to redraw the screen when we undo or redo the strokes. Each time we dispatch the `UNDO` and `REDO` we update the `historyIndex` in the application state.

We've already defined the `historyIndexSelector` and the `strokesSelector` in the `src/rootReducer.ts` file, import them:

```
import {
  currentStrokeSelector,
  historyIndexSelector,
  strokesSelector
} from "./rootReducer"
```

Get the `historyIndex` and the `strokes` values, add this code in the beginning of the `App` component:

```
const historyIndex = useSelector(historyIndexSelector)
const strokes = useSelector(strokesSelector)
```

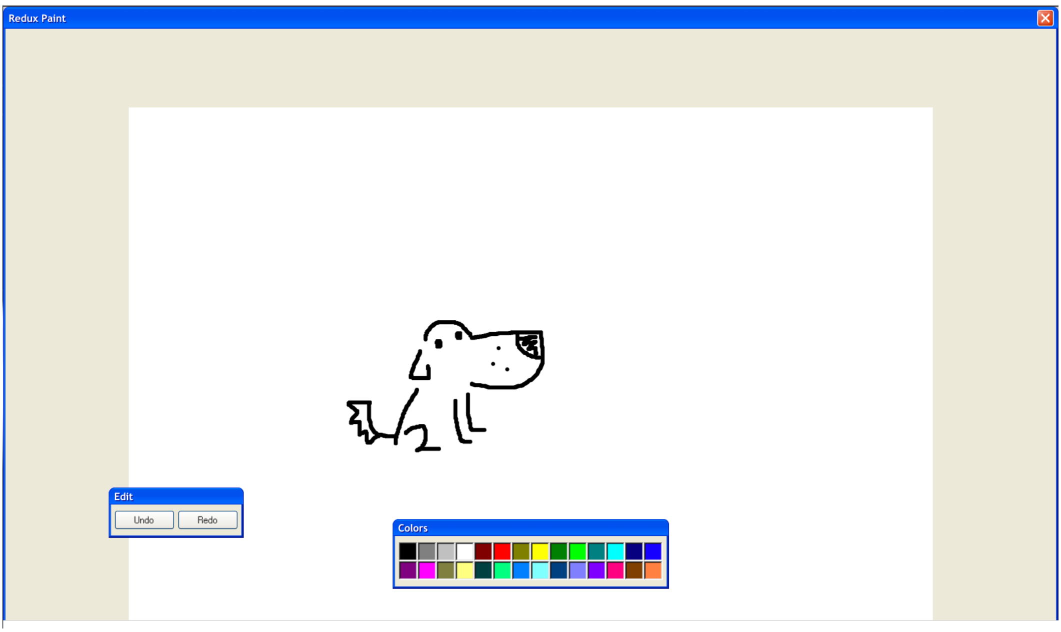
Let's add a `useEffect` block that will observe the `historyIndex` value:

```
  useEffect(() => {
    const { canvas, context } = getCanvasWithContext()
    if (!context || !canvas) {
      return
    }
    requestAnimationFrame(() => {
      clearCanvas(canvas)

      strokes
        .slice(0, strokes.length - historyIndex)
        .forEach((stroke) => {
          drawStroke(context, stroke.points, stroke.color)
        })
    })
  }, [historyIndex])
```

Every time the `historyIndex` gets updated we clear the screen and then draw only the strokes that weren't undone.

To clear the canvas we set the fill color to white and draw the rectangle the size of the canvas. We already used the `clearCanvas` to prepare the canvas on `App` component mount.

Launch your app. You should now be able to undo and redo the strokes.



Redux Paint with undo and redo

Splitting Root Reducer And Using combineReducers

If you look at our state type you'll see that it has three root-level fields:

- `currentStroke` - the stroke we are currently drawing
- `strokes` - the list of drawn lines
- `historyIndex` - the number of strokes that were undone

We can organize our code better if we split them into three separate reducers.

Separate The History Index

First, let's move out the `historyIndex` field.

Create a new folder `src/modules`. Create another folder inside it, called `historyIndex`.

Create a new file `src/modules/historyIndex/actions.ts` and move the `UNDO` and `REDO` action types and action creators from the `src/actions.ts` file.

```
import { Stroke } from "../../utils/types"

export const UNDO = "UNDO"
export const REDO = "REDO"
export const END_STROKE = "END_STROKE"

export type HistoryIndexAction =
  | {
      type: typeof UNDO
      payload: number
    }
  | {
      type: typeof REDO
    }
  | {
      type: typeof END_STROKE
      payload: { stroke: Stroke; historyIndex: number }
    }

export const undo = (undoLimit: number) => {
  return { type: UNDO, payload: undoLimit }
}

export const redo = () => {
  return { type: REDO }
}
```

The `UNDO` action now has the `payload` field. We'll pass the current amount of strokes throught that field when we undo them. We need to do it, because now our `historyIndex` reducer is separated from the other fields and it is not aware about the amount of strokes in the drawing.

Create a new file `src/modules/historyIndex/reducer.ts`. Import the actions and the `RootState` type:

```
import { RootState } from "../../utils/types"
import { HistoryIndexAction, UNDO, REDO, END_STROKE } from "./actions"
```

Now define the reducer with the following contents:

```
export const reducer = (
  state: RootState["historyIndex"] = 0,
  action: HistoryIndexAction
) => {
  switch (action.type) {
    case END_STROKE: {
      return 0
    }
    case UNDO: {
      return Math.min(state + 1, action.payload)
    }
    case REDO: {
      return Math.max(state - 1, 0)
    }
    default:
      return state
  }
}
```

Remove the `UNDO` and `REDO` action handlers from our root reducer.

Move the `historyIndex` selector to the `src/modules/historyIndex/reducer.ts`:

```
export const historyIndexSelector = (state: RootState) =>
  state.historyIndex
```

These actions were dispatched from the `EditPanel`, lets go to `src/shared/EditPanel` and update the way we import and use them:

```
import { useDispatch, useSelector } from "react-redux"
import { undo, redo } from "../modules/historyIndex/actions"
import { strokesLengthSelector } from "../modules/strokes/reducer"
```

Now we are going to need the `undoLimit` value. Add this code in the beginning of the `EditPanel` body:

```
const undoLimit = useSelector(strokesLengthSelector)
```

Update the undo button's `onClick` handler, pass the `undoLimit` as the `undo` action payload:

```
onClick={() => dispatch(undo(undoLimit))}
```

Separate The Current Stroke

Create a new folder `src/modules/currentStroke`. Inside of it create a new file `src/modules/currentStroke/actions.ts`. Import the `Point` and `Stroke` types:

```
import { Point, Stroke } from "../../utils/types"
```

Move the `BEGIN_STROKE`, `UPDATE_STROKE`, and `SET_STROKE_COLOR` types there.

```
export const BEGIN_STROKE = "BEGIN_STROKE"
export const UPDATE_STROKE = "UPDATE_STROKE"
export const SET_STROKE_COLOR = "SET_STROKE_COLOR"
export const END_STROKE = "END_STROKE"
```

Then move the `Action` type definition:

```
export type Action =
  | {
      type: typeof BEGIN_STROKE
      payload: Point
    }
  | {
      type: typeof UPDATE_STROKE
      payload: Point
    }
  | {
      type: typeof SET_STROKE_COLOR
      payload: string
    }
  | {
      type: typeof END_STROKE
      payload: { stroke: Stroke; historyIndex: number }
    }
```

Move the action creators from the `src/actions.ts`:

```
export const beginStroke = (x: number, y: number) => {
  return { type: BEGIN_STROKE, payload: { x, y } }
}

export const updateStroke = (x: number, y: number) => {
  return { type: UPDATE_STROKE, payload: { x, y } }
}

export const setStrokeColor = (color: string) => {
  return { type: SET_STROKE_COLOR, payload: color }
}

export const endStroke = (historyIndex: number, stroke: Stroke) => {
  return { type: END_STROKE, payload: { historyIndex, stroke } }
}
```

Update the action import in the `src/shared/ColorPanel.tsx`:

```
import { setStrokeColor } from "../modules/currentStroke/actions"
```

Let's separate the `currentStroke` reducer. Create a new file `src/modules/currentStroke/reducer` and import the actions and the root state type:

```
import {
  Action,
  UPDATE_STROKE,
  BEGIN_STROKE,
  END_STROKE,
  SET_STROKE_COLOR
} from "./actions"
import { RootState } from "../../utils/types"
```

Define the initial state:

```
const initialState: RootState["currentStroke"] = {
  points: [],
  color: "#000"
}
```

Move the `BEGIN_STROKE`, `UPDATE_STROKE`, `SET_STROKE_COLOR`, and `END_STROKE` action handlers from our root reducer to this file.

```
export const reducer = (
  state: RootState["currentStroke"] = initialState,
  action: Action
) => {
  switch (action.type) {
    case BEGIN_STROKE: {
      return { ...state, points: [action.payload] }
    }
    case UPDATE_STROKE: {
      return {
        ...state,
        points: [...state.points, action.payload]
      }
    }
    case SET_STROKE_COLOR: {
      return {
        ...state,
        color: action.payload
      }
    }
    case END_STROKE: {
      return {
        ...state,
        points: []
      }
    }
    default:
      return state
```

```
  }
}
```

Move the `currentStroke` selector from `src/rootReducer.ts` to `src/modules/currentStroke/red`

```
export const currentStrokeSelector = (state: RootState) =>
  state.currentStroke
```

Separate The Strokes List

Create a new folder `src/modules/strokes` and create `src/modules/strokes/actions.ts` file. Define the `END_STROKE` action type and action creator there:

```
import { Stroke } from "../../utils/types"

export const END_STROKE = "END_STROKE"

export type Action = {
  type: typeof END_STROKE
  payload: { stroke: Stroke; historyIndex: number }
}

export const endStroke = (historyIndex: number, stroke: Stroke) => {
  return { type: END_STROKE, payload: { historyIndex, stroke } }
}
```

We are going to process the `END_STROKE` action both in the `historyIndex` and the `stroke` reducers.

The `END_STROKE` action payload contains the `historyIndex` and the current `stroke` references. Just like with the `historyIndex` reducer, when we split it from the root reducer we provide the values from other reducers through the actions payloads.

Create a new file `src/modules/strokes/reducer.ts` and make the necessary imports:

```
import { RootState } from "../../utils/types"
import { Action, END_STROKE } from "./actions"
```

Add the `END_STROKE` action handler from our root reducer to this file.

```
export const reducer = (
  state: RootState["strokes"] = [],
  action: Action
) => {
  switch (action.type) {
    case END_STROKE: {
      const { historyIndex, stroke } = action.payload
      if (!stroke.points.length) {
        return state
      }
      return [...state.slice(0, state.length - historyIndex), stroke]
    }
    default:
      return state
  }
}
```

We use the `stroke` field from the action payload to add it to the strokes array. We slice the previous strokes value so that when we draw a new stroke after we've undone a bunch of previous strokes - we remove them from the history.

Also we don't modify the `historyIndex` state anymore. We have a separate `END_STROKE` action handler in the `historyIndex` reducer that sets the `historyIndex` value to zero there.

Move the `strokes` and the `strokesLength` selectors from `src/rootReducer.ts`:

```
export const strokesLengthSelector = (state: RootState) =>
  state.strokes.length

export const strokesSelector = (state: RootState) => state.strokes
```

Update the `App` component

Go to the `src/App.tsx` and update the imports:

```
import {
  beginStroke,
  endStroke,
  updateStroke
} from "./modules/currentStroke/actions"
import { strokesSelector } from "./modules/strokes/reducer"
import { currentStrokeSelector } from "./modules/currentStroke/reducer"
import { historyIndexSelector } from "./modules/historyIndex/reducer"
```

Update the `endDrawing` function to pass the `historyIndex` and the `currentStroke` to the `endStroke` action creator:

```
const endDrawing = () => {
  if (isDrawing) {
    dispatch(endStroke(historyIndex, currentStroke))
  }
}
```

Join The Reducers Using combineReducers

Now we can remove the `src/rootReducer.ts` and instead use a combination of isolated reducers.

Go to `src/store.ts`, import `combineReducers` from `redux`, and remove the `rootReducer` import.

Now instead of `rootReducer` we'll pass a combined reducer to the `createStore` method:

```
import { createStore, applyMiddleware, combineReducers } from "redux"
import { composeWithDevTools } from "redux-devtools-extension"
import { reducer as historyIndex } from "./modules/historyIndex/reducer"
import { reducer as currentStroke } from "./modules/currentStroke/reduc\
er"
import { reducer as strokes } from "./modules/strokes/reducer"
import { logger } from "redux-logger"

export const store = createStore(
  combineReducers({
    historyIndex,
    currentStroke,
    strokes
  }),
        composeWithDevTools(applyMiddleware(logger))
)
```

We import our reducers separately. Then we pass an object with our reducers as fields to the `combineReducers` method.

At this point we can also remove the `src/actions.ts` file as well.

Launch the application to check that it works.

Exporting An Image

Let's allow exporting the picture to a file. To do this we'll need to retrieve the bitmap information from our canvas, transform it into a Blob[150] and then save it as a file locally.

[150] https://developer.mozilla.org/en-US/docs/Web/API/Blob

The file saving logic will be defined in a separate component. We will use the React Context API to make the canvas reference available there.

Let's define the `CanvasProvider`. Create a new file `src/CanvasContext.tsx` with the following contents:

```
import React, {
  createContext,
  PropsWithChildren,
  useRef,
  RefObject,
  useContext
} from "react"

export const CanvasContext = createContext<
  RefObject<HTMLCanvasElement>
>({} as RefObject<HTMLCanvasElement>)

export const CanvasProvider = ({
  children
}: PropsWithChildren<{}>) => {
  const canvasRef = useRef<HTMLCanvasElement>(null)

  return (
    <CanvasContext.Provider value={canvasRef}>
      {children}
    </CanvasContext.Provider>
  )
}

export const useCanvas = () => useContext(CanvasContext)
```

This provider will store the reference to the context. Go to `src/index.tsx` and wrap the component tree into the `CanvasContext`:

```
import { CanvasProvider } from "./CanvasContext"
  // ...
ReactDOM.render(
  <React.StrictMode>
    <Provider store={store}>
      <CanvasProvider>
        <App />
      </CanvasProvider>
    </Provider>
  </React.StrictMode>,
  document.getElementById("root")
)
```

Go to `src/App.tsx`, remove the `useRef` import and import the `useCanvas` hook:

```
import React, { useEffect } from "react"
  // ...
import { useCanvas } from "./CanvasContext"
```

Change the `useRef` call to `useCanvas`:

```
const canvasRef = useCanvas()
```

Define the `getCanvasImage`

Go to `src/utils/canvasUtils.ts` add the `getCanvasImage` function:

```
export const getCanvasImage = (
  canvas: HTMLCanvasElement | null
): Promise<null | Blob> => {
  return new Promise((resolve, reject) => {
    if (!canvas) {
      return reject(null)
    }
    canvas.toBlob(resolve)
  })
}
```

This function resolves with the canvas contents transformed into a Blob. Later we'll be able to save this Blob as a file.

Create the `FilePanel`

The `FilePanel` component will contain the code that will get the binary data from the canvas and save it into a file. To save the data into a file we'll use the `file-saver` package.

Install the `file-saver` and types for it:

```
yarn add file-saver @types/file-saver
```

After it's done open `src/index.css` and add a new CSS class:

```
.file {
  position: fixed;
  bottom: 40px;
  right: 20%;
  z-index: 10;
}
```

Create a new file `src/shared/FilePanel.tsx`. This panel will contain the *Export* button.

Make the necessary imports:

```
import { useCanvas } from "../CanvasContext"
import { saveAs } from "file-saver"
import { getCanvasImage } from "../utils/canvasUtils"
```

Define the `FilePanel` component:

```
export const FilePanel = () => {
  const canvasRef = useCanvas()

  const exportToFile = async () => {
    const file = await getCanvasImage(canvasRef.current)
    if (!file) {
      return
    }
    saveAs(file, "drawing.png")
  }

  return (
    <div className="window file">
      <div className="title-bar">
        <div className="title-bar-text">File</div>
      </div>
      <div className="window-body">
        <div className="field-row">
          <button className="save-button" onClick={exportToFile}>
            Export
          </button>
        </div>
      </div>
    </div>
  )
}
```

Here we get the reference to the canvas using the `useCanvas` hook.

When the user clicks the button we call the `exportToFile` function. There we generate the `Blob` from the canvas using the `getCanvasImage` function and then we save it to a file using the `file-saver` package.

Add the `FilePanel` to the `App` layout

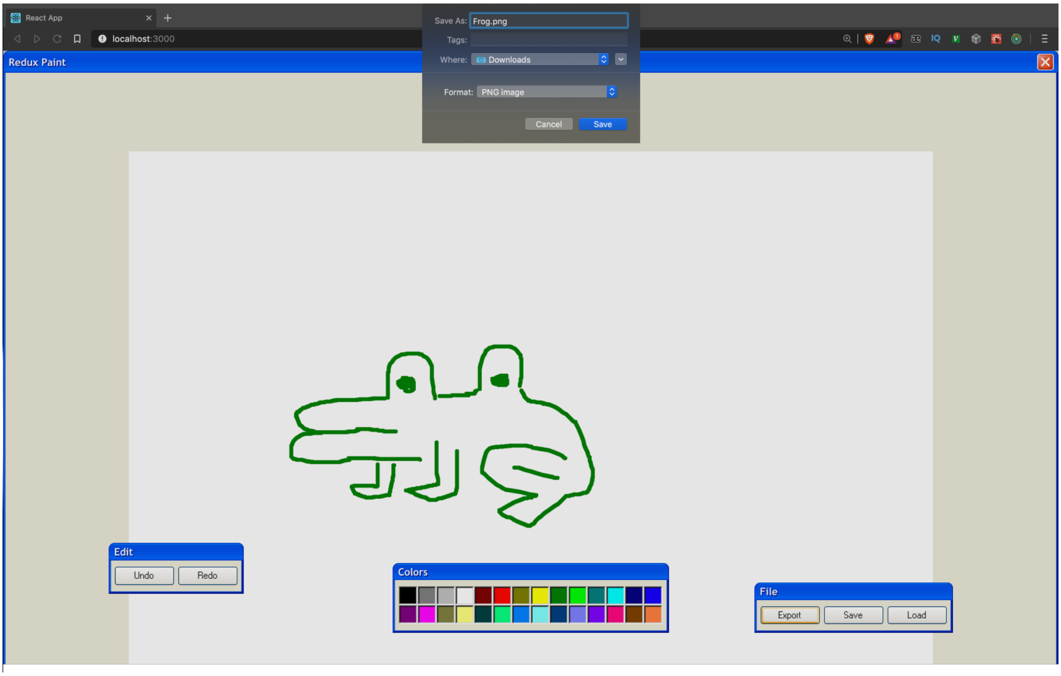
Open the `src/App.tsx` and import the `FilePanel`:

```
import { FilePanel } from "./shared/FilePanel"
```

Add the `FilePanel` to the `App` layout:

```
<div className="window">
  <div className="title-bar">
    <div className="title-bar-text">Redux Paint</div>
    <div className="title-bar-controls">
      <button aria-label="Close" />
    </div>
  </div>
  <EditPanel />
  <ColorPanel />
  <FilePanel />
  <canvas
    onMouseDown={startDrawing}
    onMouseUp={endDrawing}
    onMouseOut={endDrawing}
    onMouseMove={draw}
    ref={canvasRef}
  />
</div>
```

Launch your application, draw something, and try to export it as a file.



Exporting an image

Using Redux Toolkit

Redux Toolkit[151] is an official toolset for Redux development provided by the Redux team. It simplifies the setup and adds a bunch of neat tools that simplify developing Redux-based applications.

Let's upgrade our application to use it.

Install Redux Toolkit:

```
yarn add @reduxjs/toolkit
```

Now you can remove the `redux` package.

[151] https://redux-toolkit.js.org/

```
yarn remove redux
```

Configuring The Store

The first change is how you initialize your store. Now it's done using the configureStore[152] method.

Open `src/store.ts` and remake it like this:

```
import { configureStore } from "@reduxjs/toolkit"
import { reducer as currentStroke } from "./modules/currentStroke/reduc\
er"
import { reducer as historyIndex } from "./modules/historyIndex/reducer"
import { reducer as strokes } from "./modules/strokes/reducer"
import logger from "redux-logger"

export const store = configureStore({
  reducer: {
    historyIndex,
    strokes,
    currentStroke
  },
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware().concat(logger)
})
```

> It is expected that you'll get TypeScript errors here, because the reducers type signatures don't match what redux toolkit is expecting. We'll fix that in the next step.

Now we don't have to combine middleware, we can provide them as a list.

We use `getDefaultMiddleware` to use the default middlewares provided by `redux-toolkit`.

Currently, the list of returned middlewares contains the following:

[152] https://redux-toolkit.js.org/api/configureStore

- Immutability Check Middleware[153] - this middleware checks that you don't mutate the state in your reducers. It will throw an error if you do.
- Serializability check middleware[154] - it checks that your state does not contain non-serializable data. For example, functions, symbols, Promises, and other non-data values.

If you look at the `configureStore` arguments you'll see that instead of positional arguments where you need to remember which order they go in, it now accepts an options object. So you specify the values by name, which decreases the chance of error.

Fix Type Errors

TypeScript is complaining about the type signatures of the reducers, because it wants them to accept actions with optional payloads.

The only reducer that does not cause an error here is the `historyIndex` reducer. And that's because of the `REDO` action, that does not have the `payload` property.

To fix the other reducers let's add the `AnyAction` type to their action union types.

Open `src/modules/currentStroke/actions.ts` and add the following:

```
import { AnyAction } from "@reduxjs/toolkit"
  // ...
export type Action =
  | AnyAction
  | {
      type: typeof BEGIN_STROKE
      payload: Point
    }
  | {
      type: typeof UPDATE_STROKE
      payload: Point
```

[153] https://github.com/reduxjs/redux-toolkit/blob/master/docs/api/immutabilityMiddleware.md
[154] https://github.com/reduxjs/redux-toolkit/blob/master/docs/api/serializabilityMiddleware.md

```
    }
  | {
      type: typeof SET_STROKE_COLOR
      payload: string
    }
  | {
      type: typeof END_STROKE
      payload: { stroke: Stroke; historyIndex: number }
    }
```

Do the same with the `src/modules/strokes/actions.ts`:

```
import { AnyAction } from "@reduxjs/toolkit"
  // ...
export type Action =
  | AnyAction
  | {
      type: typeof END_STROKE
      payload: { stroke: Stroke; historyIndex: number }
    }
```

Using createAction

Right now we define a type constant and an action creator for each action in our project.

Redux Toolkit provides the createAction[155] method that simplifies it.

When you use `createAction` you only need to provide the action type string to it. The resulting action creator will set whatever arguments you pass to it as the action payload.

In Typescript we need to specify the form of payload in advance - this is why we set the payload type as a generic argument value.

[155] https://redux-toolkit.js.org/api/createAction

We are going to start with the `endStroke` action, it is going to be used by multiple reducers so we'll define it in a shared module. Create a new `src/modules/sharedActions.ts` with the following contents:

```
import { AnyAction, createAction } from "@reduxjs/toolkit"
import { Stroke } from "../utils/types"

export type SharedAction = AnyAction | ReturnType<typeof endStroke>

export const endStroke = createAction<{
  stroke: Stroke
  historyIndex: number
}>("endStroke")
```

We define the `SharedAction` type as a union with the `AnyAction` from `redux-toolkit` so that later they are compatible with reducers created using `redux-toolkit`.

Go to `src/modules/historyIndex/actions.ts` and make it look like this:

```
import { AnyAction, createAction } from "@reduxjs/toolkit"

export type Action =
  | AnyAction
  | ReturnType<typeof undo>
  | ReturnType<typeof redo>

export const undo = createAction<number>("UNDO")

export const redo = createAction("REDO")
```

Don't forget to update the `historyIndex` reducer. Update the imports:

```
import { Action, undo, redo } from "./actions"
import { endStroke } from "../sharedActions"
```

Update the reducer `action` argument type:

```
export const reducer = (
  state: RootState["historyIndex"] = 0,
  action: Action
) => {
  // ...
  }
```

Remake the reducer to use the generated actions as action types:

```
switch (action.type) {
  case endStroke.toString(): {
    return 0
  }
  case undo.toString(): {
    return Math.min(state + 1, action.payload)
  }
  case redo.toString(): {
    return Math.max(state - 1, 0)
  }
  default:
    return state
}
```

Then go to `src/modules/currentStroke/actions.ts`, update it to use the `createAction` method:

```
import { AnyAction, createAction } from "@reduxjs/toolkit"
import { Point } from "../../utils/types"

export type Action =
  | AnyAction
  | ReturnType<typeof beginStroke>
  | ReturnType<typeof updateStroke>
  | ReturnType<typeof setStrokeColor>

```

```
export const beginStroke = createAction<Point>("BEGIN_STROKE")

export const updateStroke = createAction<Point>("UPDATE_STROKE")

export const setStrokeColor = createAction<string>("SET_STROKE_COLOR")
```

Update the `currentStroke` reducer file, start with the imports:

```
import {
  Action,
  updateStroke,
  beginStroke,
  setStrokeColor
} from "./actions"
import { endStroke } from "../sharedActions"
import { RootState } from "../../utils/types"
```

Remake the `switch/case` to use the generated actions:

```
case beginStroke.toString(): {
  return { ...state, points: [action.payload] }
}
case updateStroke.toString(): {
  return {
    ...state,
    points: [...state.points, action.payload]
  }
}
case setStrokeColor.toString(): {
  return {
    ...state,
    color: action.payload
  }
}
case endStroke.toString(): {
```

```
  return {
    ...state,
    points: []
  }
}
```

After it's done open the `src/modules/strokes/reducer.ts` and import the `endStroke` and `SharedAction` from the `sharedActions` module:

```
import { endStroke, SharedAction } from "../sharedActions"
```

Set the `action` argument type in the reducer to `SharedAction`:

```
export const reducer = (
  state: RootState["strokes"] = [],
  action: SharedAction
) => {
  // ...
}
```

Update the body of the reducer to use the imported action creator:

```
export const reducer = (
  state: RootState["strokes"] = [],
  action: SharedAction
) => {
  switch (action.type) {
    case endStroke.toString(): {
  // ...
    }
  // ...
  }
}
```

Update the `App` component

Open the `src/App.tsx` and import the `endStroke` action creator from the `sharedActions` module:

```
import { endStroke } from "./modules/sharedActions"
```

Update the canvas event handlers. The action creators signarutes have changed so now we need to pass the arguments a bit differently.

In the `startDrawing` we now pass an object with `x` and `y` fields to the `beginStroke` action creator:

```
const startDrawing = ({
  nativeEvent
}: React.MouseEvent<HTMLCanvasElement>) => {
  const { offsetX, offsetY } = nativeEvent
  dispatch(beginStroke({ x: offsetX, y: offsetY }))
}
```

Same happens with the `updateStroke` action creator in the `draw` function:

```
  const draw = ({
    nativeEvent
  }: React.MouseEvent<HTMLCanvasElement>) => {
    if (!isDrawing) {
      return
    }
    const { offsetX, offsetY } = nativeEvent

                dispatch(updateStroke({ x: offsetX, y: offsetY }))
  }
```

The `endStroke` action creator now accepts an object with `historyIndex` and `stroke` fields:

```
const endDrawing = () => {
  if (isDrawing) {
    dispatch(endStroke({ historyIndex, stroke: currentStroke }))
  }
}
```

Using createReducer

Now let's update our reducers. For this, the Redux Toolkit provides the `createReducer` method.

The main difference you get when using it is that now you can mutate the state, instead of always returning the new value.

This is achieved by using the Immer[156] library internally.

CurrentStroke Reducer

Let's remake the `currentStroke` reducer first. Go to the `src/modules/currentStroke/reducer.ts` and import `createReducer` from `@reduxjs/toolkit`:

```
import { createReducer } from "@reduxjs/toolkit"
```

We won't need the `Action` type, so you can remove it from the imports:

```
import { beginStroke, setStrokeColor, updateStroke } from "./actions"
```

Now update the reducer to look like this:

[156] https://immerjs.github.io/immer/docs/introduction

```
export const reducer = createReducer(initialState, (builder) => {
  builder.addCase(beginStroke, (state, action) => {
    state.points = [action.payload]
  })
  builder.addCase(updateStroke, (state, action) => {
    state.points.push(action.payload)
  })
  builder.addCase(setStrokeColor, (state, action) => {
    state.color = action.payload
  })
  builder.addCase(endStroke, (state) => {
    state.points = []
  })
})
```

`createReducer` accepts two arguments, the initial state and the callback.

The passed callback receives an instance of `ActionReducerMapBuilder` object. It has a method `addCase` that we use do add action handlers.

This is the recommended way to add reducer cases in Typescript.

Now instead of returning a new state with an updated points array when we begin or update the stroke, we mutate the `points` array.

Strokes Reducer

Open `src/modules/strokes/reducer.ts` and rewrite the code to use `createReducer`:

```
import { endStroke } from "../sharedActions"
import { RootState } from "../../utils/types"
import { createReducer } from "@reduxjs/toolkit"

const initialStrokes: RootState["strokes"] = []

export const reducer = createReducer(initialStrokes, (builder) => {
  builder.addCase(endStroke, (state, action) => {
    const { historyIndex, stroke } = action.payload
    if (historyIndex === 0) {
      state.push(stroke)
    } else {
      state.splice(-historyIndex, historyIndex, stroke)
    }
  })
})

export const strokesLengthSelector = (state: RootState) =>
  state.strokes.length

export const strokesSelector = (state: RootState) => state.strokes
```

Here we need to add only one case that will handle the `END_STROKE` action.

If `historyIndex` is `0` we add the stroke that we just finished to the array of strokes. Otherwise, we override the number of strokes equal to the `historyIndex` value and add the new stroke to the end.

Note that we'll also have to react to this action in the `historyAction` reducer. We'll need to set it to `0` when the stroke is ended.

HistoryIndex Reducer

Go to `src/modules/historyIndex/reducer.ts` and rewrite it to `createReducer`:

```
import { endStroke } from "../sharedActions"
import { redo, undo } from "./actions"
import { createReducer } from "@reduxjs/toolkit"
import { RootState } from "../../utils/types"

const initialState: RootState["historyIndex"] = 0

export const reducer = createReducer(initialState, (builder) => {
  builder.addCase(undo, (state, action) => {
    return Math.min(state + 1, action.payload)
  })
  builder.addCase(redo, (state) => {
    return Math.max(state - 1, 0)
  })
  builder.addCase(endStroke, () => {
    return 0
  })
})

export const historyIndexSelector = (state: RootState) =>
  state.historyIndex
```

Note that here we return a new value instead of updating it like in other reducers. That's because of Immer. You can't re-define the whole state. If you need to do this, you have to return a new value instead.

In other reducers, we were updating the individual fields of the state. In this case, you can just mutate the state and Immer will internally generate the new state, based on the mutations you've made.

But when a state is a number, like in `historyIndex` reducer, and to update it you would override it with a new value, then we return a new value instead.

> Read more about the pitfalls of using Immer in the Immer Documentation.[157]

Launch the application and make sure it works.

[157] https://immerjs.github.io/immer/docs/pitfalls

Using Slices

Currently, we create actions and reducer handles for them separately.

We migrated to `createAction` and `createReducer` functions that made our code more compact. But we can move even further.

Redux provides a `createSlice` function that automatically generates action creators based on the reducer handles you have.

Let's rewrite our reducers to slices.

HistoryIndex Slice

Go to `src/modules/historyIndex/reducer.ts`, rename it as `slice.ts` and import `createSlice` and `PayloadAction` from `redux-toolkit`:

```
import { createSlice, PayloadAction } from "@reduxjs/toolkit"
```

We'll also need the `RootState` type and the `endStroke` shared action:

```
import { RootState } from "../../utils/types"
import { endStroke } from "../sharedActions"
```

Now remake the reducer into slice:

```
export const historyIndex = createSlice({
  name: "historyIndex",
  initialState: 0,
  reducers: {
    undo: (state, action: PayloadAction<number>) => {
      return Math.min(state + 1, action.payload)
    },
    redo: (state) => {
      return Math.max(state - 1, 0)
    }
```

```
  },
  extraReducers: (builder) => {
    builder.addCase(endStroke, () => {
      return 0
    })
  }
})
```

Here we pass an options object to `createSlice`. It needs to have the following fields:

- `name` - the name of the slice. It will be used as a prefix for all the generated actions of this slice
- `initialState` - the initial state value
- `reducers` - reducers that will be used to generate actions
- `extraReducers` - reducers that need to react on shared actions

Our slice has `historyIndex` as its name. It also has two action handlers - `undo` and `redo`. This means that it will generate two actions:

- `historyIndex/undo` - this action will have a number payload. We need it to limit the number of undos to the length of the strokes array.
- `historyIndex/redo` - this action won't have any payload.

We also need to handle the `END_STROKE` action to reset the `historyIndex` to `0`.

As the `END_STROKE` action is shared, we defined it in the `extraReducers`:

```
extraReducers: (builder) => {
  builder.addCase(endStroke, () => {
    return 0
  })
}
```

Export the reducer, actions and the selector from the slice:

```
export default historyIndex.reducer

export const { undo, redo } = historyIndex.actions

export const historyIndexSelector = (state: RootState) =>
  state.historyIndex
```

Remove the `src/modules/historyIndex/actions.ts` file.

Now update the `src/store.ts` file to use the `historyIndex` slice:

```
import historyIndex from "./modules/historyIndex/slice"
```

Launch the app, draw a few strokes, and press the undo and redo buttons.

Look at the `redux-logger` output. You should see the generated actions there.

Note how the actions now are composed of the slice name combined with the reducer case name.

Strokes Slice

Go to `src/modules/strokes/reducer.ts` and rename it `slice.ts`.

Make the necessary imports:

```
import { createSlice } from "@reduxjs/toolkit"
import { RootState } from "../../utils/types"
import { endStroke } from "../sharedActions"
```

Define the initial state:

```
const initialState: RootState["strokes"] = []
```

Our initial state is just an empty array. We must provide the correct type manually. This type will be used by Redux Toolkit to infer the type of your slice state.

Define the slice:

```
const strokes = createSlice({
  name: "strokes",
  initialState,
  reducers: {},
  extraReducers: (builder) => {
    builder.addCase(endStroke, (state, action) => {
      const { historyIndex, stroke } = action.payload
      if (historyIndex === 0) {
        state.push(stroke)
      } else {
        state.splice(-historyIndex, historyIndex, stroke)
      }
    })
  }
})
```

This slice doesn't have any linked actions. The only action it handles is the shared `END_STROKE`.

Export the reducer and selectors:

```
export default strokes.reducer

export const strokesLengthSelector = (state: RootState) =>
  state.strokes.length

export const strokesSelector = (state: RootState) => state.strokes
```

CurrentStroke Slice

Open `src/modules/currentStroke/reducer.ts`. Let's remake it to slice as well. Rename the file to `slice.ts` and remake the imports:

```
import { endStroke } from "../sharedActions"
import { createSlice, PayloadAction } from "@reduxjs/toolkit"
import { RootState, Point } from "../../utils/types"
```

Then define the initial state:

```
const initialState: RootState["currentStroke"] = {
  points: [],
  color: "#000"
}
```

Now let's remake the reducer into a slice:

```
const slice = createSlice({
  name: "currentStroke",
  initialState,
  reducers: {
    beginStroke: (state, action: PayloadAction<Point>) => {
      state.points = [action.payload]
    },
    updateStroke: (state, action: PayloadAction<Point>) => {
      state.points.push(action.payload)
    },
    setStrokeColor: (state, action: PayloadAction<string>) => {
      state.color = action.payload
    }
  },
  extraReducers: (builder) => {
    builder.addCase(endStroke, (state) => {
      state.points = []
    })
  }
})
```

This slice has three handlers that will generate actions:

- `currentStroke/beginStroke` - this action will have the payload of type `Point`
- `currentStroke/updateStroke` - will also hold a `Point` as a payload
- `currentStroke/updateColor` - there we'll pass a `string` representing the stroke color in its payload.

We also handle the `END_STROKE` shared action:

```
extraReducers: (builder) => {
  builder.addCase(endStroke, (state) => {
    state.points = []
  })
}
```

In this extra reducer, we reset the `currentStroke` points array.

Export the reducer, actions and selector:

```
export const currentStroke = slice.reducer

export const { beginStroke, updateStroke, setStrokeColor } =
  slice.actions

export const currentStrokeSelector = (state: RootState) =>
  state.currentStroke
```

Now the actions are generated by the slice, so you can remove the `src/modules/currentStroke/a`

Remake The Imports

Go to `src/store.ts`. Remake the remaining imports to slices:

```
import { currentStroke } from "./modules/currentStroke/slice"
  // ...
import strokes from "./modules/strokes/slice"
```

Update the `sharedActions` module, open the `src/modules/sharedActions.ts` and make it look like this:

```
import { createAction } from "@reduxjs/toolkit"
import { Stroke } from "../utils/types"

export const endStroke = createAction<{
  stroke: Stroke
  historyIndex: number
}>("endStroke")
```

We don't need to import the `AnyAction` from `redux-toolkit` and we don't need to export our own `SharedAction` type.

Go to `src/App.tsx` and update the action imports there:

```
import {
  beginStroke,
  updateStroke
} from "./modules/currentStroke/slice"
  // ...
import { strokesSelector } from "./modules/strokes/slice"
import { currentStrokeSelector } from "./modules/currentStroke/slice"
import { historyIndexSelector } from "./modules/historyIndex/slice"
```

Update the action and selector imports in the `src/EditPanel.tsx`:

```
import { undo, redo } from "../modules/historyIndex/slice"
import { strokesLengthSelector } from "../modules/strokes/slice"
```

Same in the `src/ColorPanel.tsx`:

```
import { setStrokeColor } from "../modules/currentStroke/slice"
```

Now our application uses slices - congratulations! Launch the app and verify that everything works.

Add Modal Windows

Now let's add a modal window that will allow us to save the projects.

To keep the state of this window we'll create a new slice.

Create a new file `src/modules/modals/slice.ts`.

Make the imports:

```
import { createSlice, PayloadAction } from "@reduxjs/toolkit"
import { RootState } from "../../utils/types"
```

Define the `ModalState` type:

```
export type ModalState = {
  isShown: boolean
  modalName: string | null
}
```

Then define the initial state with this type:

```
const initialState: ModalState = {
  isShown: true,
  modalName: null
}
```

Now we can define the slice:

```
const slice = createSlice({
  name: "modal",
  initialState,
  reducers: {
    show: (state, action: PayloadAction<string>) => {
      state.isShown = true
      state.modalName = action.payload
    },
    hide: (state) => {
      state.isShown = true
      state.modalName = null
    }
  }
})
```

This slice handles two actions:

- `show` - this slice has a `string` payload that holds the name of the window we want to show.
- `hide` - this action signals that we want to hide all the windows

Export the reducer, actions and selectors:

```
export const modalVisible = slice.reducer

export const { show, hide } = slice.actions

export const modalVisibleSelector = (state: RootState) =>
  state.modalVisible

export const modalNameSelector = (state: RootState) =>
  state.modalVisible.modalName
```

Go to `src/store.ts` and import the new reducer:

```
import { modalVisible } from "./modules/modals/slice"
```

Add the reducer to the combined store:

```
export const store = configureStore({
  reducer: {
    historyIndex,
    strokes,
    currentStroke,
    modalVisible
  },
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware().concat(logger)
})
```

Update the types

Open the `src/utils/types.ts` and add the `ModalState` to the `RootState`:

```
import { ModalState } from "../modules/modals/slice"
  // ...
export type RootState = {
  currentStroke: Stroke
  strokes: Stroke[]
  historyIndex: number
  modalVisible: ModalState
}
```

Add The Modal Manager Component

In this section we'll define the visual part of the modal management system.

Define the modal windows

Open `src/index.css` and add a new CSS class `.modal-panel`:

```
.modal-panel {
  position: fixed;
  top: 50%;
  left: 50%;
  transform: translate3d(-50%, -50%, 0);
  z-index: 10;
}
```

Will use it for both modal windows.

Create a new file `src/ProjectSaveModal.tsx`, later we'll use it to save our projects, for now it will only contain the basic layout:

```
import { useDispatch } from "react-redux"
import { hide } from "./modules/modals/slice"

export const ProjectSaveModal = () => {
  const dispatch = useDispatch()

  return (
    <div className="window modal-panel">
      <div className="title-bar">
        <div className="title-bar-text">Save</div>
      </div>
      <div className="window-body">
        <div className="field-row">
          <button onClick={() => dispatch(hide())}>Cancel</button>
        </div>
      </div>
    </div>
  )
}
```

As you can see it is mostly just layout code. It also contains a *Cancel* button that allows to close the modal.

Let's create the `ProjectsModal` component. Later we'll use it to load the project from the server. For now it will only contain some basic layout, just like the `ProjectSaveModal`.

Create a new file `src/ProjectsModal.tsx` with the following code:

```
import { useDispatch } from "react-redux"
import { hide } from "./modules/modals/slice"

export const ProjectsModal = () => {
  const dispatch = useDispatch()

  return (
    <div className="window modal-panel">
      <div className="title-bar">
        <div className="title-bar-text">Load Project</div>
        <div className="title-bar-controls">
          <button
            aria-label="Close"
            onClick={() => dispatch(hide())}
          />
        </div>
      </div>
      <div className="projects-container">Projects List</div>
    </div>
  )
}
```

Define the `ModalLayer` component

Let's define a component to render modal windows. Create a new file `src/ModalLayer.tsx` with the following content:

```
import React from "react"
import { useSelector } from "react-redux"
import { ProjectsModal } from "./ProjectsModal"
import { ProjectSaveModal } from "./ProjectSaveModal"
import { modalNameSelector } from "./modules/modals/slice"

export const ModalLayer = () => {
  const modalName = useSelector(modalNameSelector)

  switch (modalName) {
    case "PROJECTS_MODAL": {
      return <ProjectsModal />
    }
    case "PROJECTS_SAVE_MODAL": {
      return <ProjectSaveModal />
    }
    default:
      return null
  }
}
```

Here we use the `modalNameSelector` to get the current modal name from our slice. Then we show different window components depending on `modalName` value.

You can see that we render `ProjectsModal` and `ProjectsSaveModal` windows. We'll define them in a moment.

Render the `ModalLayer`

Go to `src/App.tsx` import and render the `ModalLayer`

```
import { ModalLayer } from "./ModalLayer"
  // ...
    <div className="window">
      <div className="title-bar">
        <div className="title-bar-text">Redux Paint</div>
        <div className="title-bar-controls">
          <button aria-label="Close" />
        </div>
      </div>
      <EditPanel />
      <ColorPanel />
      <FilePanel />
      <ModalLayer />
      <canvas
        onMouseDown={startDrawing}
        onMouseUp={endDrawing}
        onMouseOut={endDrawing}
        onMouseMove={draw}
        ref={canvasRef}
      />
    </div>
```

Add `Save` and `Load` buttons

Now let's add *Save* and *Load* buttons to the `FilePanel` and we should be good to go. Both buttons will dispatch the `show` action with the name of the modal that we want to open. Let's import the `useDispatch` hook and the `show` action creator. Open `src/shared/FilePanel` and add the following imports:

```
import { useDispatch } from "react-redux"
import { show } from "../modules/modals/slice"
```

Now let's get the `dispatch` method, add this code in the beginning of the `FilePanel` component body:

```
const dispatch = useDispatch()
```

Add the buttons to the layout:

```
<div className="window file">
  <div className="title-bar">
    <div className="title-bar-text">File</div>
  </div>
  <div className="window-body">
    <div className="field-row">
      <button className="save-button" onClick={exportToFile}>
        Export
      </button>
      <button
        className="save-button"
        onClick={() => {
          dispatch(show("PROJECTS_SAVE_MODAL"))
        }}
      >
        Save
      </button>
      <button
        className="save-button"
        onClick={() => {
          dispatch(show("PROJECTS_MODAL"))
        }}
      >
        Load
      </button>
    </div>
  </div>
</div>
```

Both buttons have inline `onClick` handlers that `dispatch` the `show` actions with corresponding action payload.

Launch your app, and make sure you can open the modal windows.

Prepare The Server

Copy the server from `code/04-redux/completed/server` to your application root folder.

You'll also need to install a few dependencies for it to work:

```
yarn add --dev concurrently@5.1.0 \
  cors@2.8.5 \
  express@4.17.1 \
  lowdb@1.0.0 nanoid@3.1.9 \
  ts-node@8.9.0
```

We install all of them as dev dependencies so they don't end up in the application bundle.

Install the types for them as well:

```
yarn add --dev @types/cors@2.8.6\
  @types/express@4.17.6\
        @types/lowdb@1.0.9
```

Now open `package.json` and add two new launch scripts:

```
"start:server": "ts-node -O '{\"module\": \"commonjs\"}' ./server/index\
.ts",
"dev": "concurrently --kill-others \"npm run start:server\" \"npm run s\
tart\""
```

- `start:server` will launch the server only
- `dev` will launch the app and the server together

If your application is already running, you can run the server in a separate console tab:

```
yarn start:server
```

I recommend stopping your app if it's running and relaunching it using the `dev` script:

```
yarn dev
```

Save The Project Using Thunks

At this point we can save our drawings as `.png` files. In this section we'll make it possible to save the project to the server. It will be possible to load them later and continue drawing them.

We will learn how to perform side effect in Redux based applications using thunks.

Define the API module

We are going to perform a server request, let's define an `API` module. Create a new file `src/modules/strokes/api.ts` and define the `newProject` function there:

```
import { Stroke } from "../../utils/types"

export const newProject = (
  name: string,
  strokes: Stroke[],
  image: string
) =>
  fetch("http://localhost:4000/projects/new", {
    method: "POST",
    headers: {
      Accept: "application/json",
      "Content-Type": "application/json"
    },
    body: JSON.stringify({
```

```
      name,
      strokes,
      image
    })
  }).then((res) => res.json())
```

This function will perform a `POST` request to our server and send the strokes list representing our project, project name and project thumbnail.

The fact that we send the whole strokes array to the backend will allow us to use `undo/redo` functionality immediately after we load the project.

Handle saving the project

Saving the project is considered a side effect, and the official way to handle side-effects in Redux Toolkit are Thunks[158].

Think of them as special kind of action creators. Instead of returning an object with `type` and `payload`, they return an async function that will perform the side-effect.

Open `src/store.ts` and define the type for our thunk, to do this you'll need to import `ThunkAction`, `Action` and `RootState` types:

```
import { configureStore, ThunkAction, Action } from "@reduxjs/toolkit"
  // ...
import { RootState } from "./utils/types"
  // ...
export type AppThunk = ThunkAction<
  void,
  RootState,
  unknown,
  Action<string>
>
```

Open the `src/modules/strokes/slice.ts` and import the `createAsyncThunk` method:

[158] https://github.com/reduxjs/redux-thunk

```
import { createSlice, createAsyncThunk } from "@reduxjs/toolkit"
```

We'll also need to import the `newProject` method frow the `api` module:

```
import { newProject } from "./api"
```

Now define the `saveProject` thunk:

```
type SaveProjectArg = {
  projectName: string
  thumbnail: string
}
  // ...
export const saveProject = createAsyncThunk(
  "SAVE_PROJECT",
  async (
    { projectName, thumbnail }: SaveProjectArg,
    { getState }
  ) => {
    try {
      const response = await newProject(
        projectName,
        (getState() as RootState)?.strokes,
        thumbnail
      )
      console.log(response)
    } catch (err) {
      console.log(err)
    }
  }
)
```

Here we defined a thunk as a function that receives the project name and thunk through the arguments and then gets the strokes array from the state.

> Thunks can have access to the whole state and also can dispatch other actions. In this section we are not going to cover the advanced functionality of Redux Toolkit thunks.

This thunk will make a `POST` request to our backend and send the project name, the list of strokes, and a generated thumbnail for this project.

Define the `getBase64Thumbnail` function

We are going to save each project with a small thumbnail image, this image will be stored on the backend as a Base64 string. Let's greate a helper function to do this.

Create a new file `src/utils/scaler.ts` with the following contents:

```
type ScalerArgs = {
  file: Blob
  scale: number
}

export function getBase64Thumbnail({
  file,
  scale = 0.1
}: ScalerArgs): Promise<string> {
  return new Promise((res, rej) => {
    const reader = new FileReader()
    reader.readAsDataURL(file)
    reader.onload = (e) => {
      const img = new Image()
      img.onload = () => {
        const el = document.createElement("canvas")
        let w = (el.width = img.width * scale)
        let h = (el.height = img.height * scale)
        const ctx = el.getContext("2d")
        if (!ctx) {
          return
```

```
        }
        ctx.drawImage(img, 0, 0, w, h)
        return res(el.toDataURL())
      }
      reader.onerror = (e) => {
        rej(e.toString())
      }
      img.src = e?.target?.result as string
    }
  })
}
```

This function accepts a file of type `Blob` and a number by which it will scale the image, by default it will make it ten times smaller.

Inside of this function we draw the image on another `canvas` element that it invisible to the user. This is where we change the size of the image.

After the image is scaled we transform it into a Base64 string using the `toDataURL` method and then resolve the promise with this value.

Update the `ProjectSaveModal`

Open the `src/ProjectSaveModal.tsx` and add new imports:

```
import { useState, ChangeEvent } from "react"
  // ...
import { getCanvasImage } from "./utils/canvasUtils"
import { useCanvas } from "./CanvasContext"
import { getBase64Thumbnail } from "./utils/scaler"
import { saveProject } from "./modules/strokes/slice"
```

Get the `canvasRef` using the `useCanvas` hook:

```
const canvasRef = useCanvas()
```

Update the component layout, well add an input and a *Save* button:

```
<div className="window modal-panel">
  <div className="title-bar">
    <div className="title-bar-text">Save</div>
  </div>
  <div className="window-body">
    <div className="field-row-stacked">
      <label htmlFor="projectName">Project name</label>
      <input
        id="projectName"
        onChange={onProjectNameChange}
        type="text"
      />
    </div>
    <div className="field-row">
      <button onClick={onProjectSave}>Save</button>
      <button onClick={() => dispatch(hide())}>Cancel</button>
    </div>
  </div>
</div>
```

Here the `input` element triggers an `onChange` event - we handle it using the `onProjectNameChange` function. Inside of this function we update the `projectName` state variable. Define the state and this function in the component body:

```
const [projectName, setProjectName] = useState("")
// ...
const onProjectNameChange = (e: ChangeEvent<HTMLInputElement>) => {
  setProjectName(e.target.value)
}
```

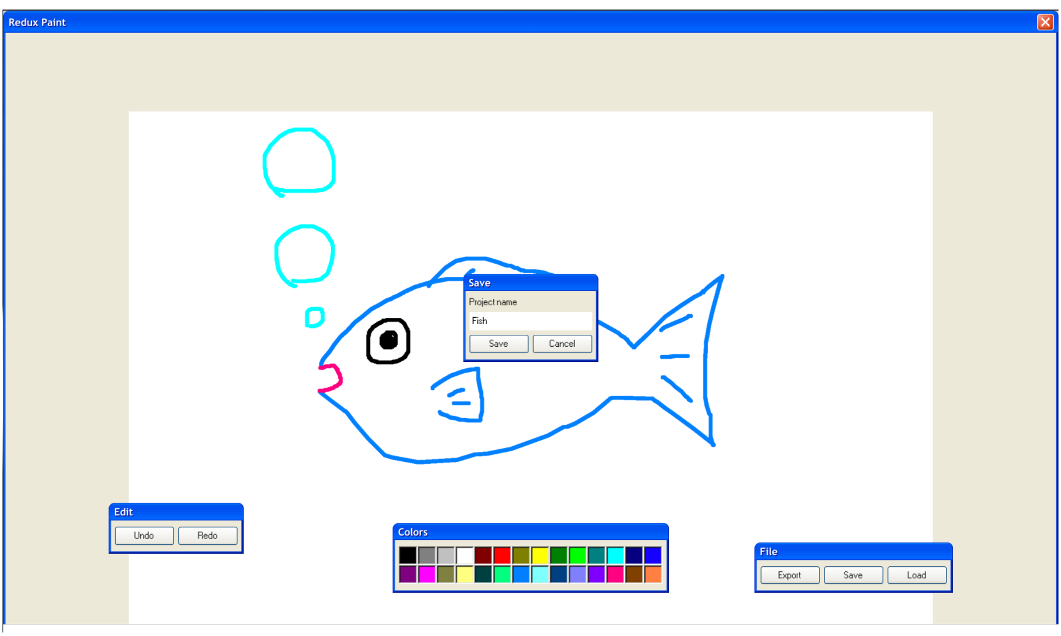
When the user clicks the *Save* button we call the `onProjectSave` handler. Let's define it:

```
const onProjectSave = async () => {
  const file = await getCanvasImage(canvasRef.current)
  if (!file) {
    return
  }
  const thumbnail = await getBase64Thumbnail({ file, scale: 0.1 })
  dispatch(saveProject({ projectName, thumbnail }))
  setProjectName("")
  dispatch(hide())
}
```

Here we get the current bitmap data from the canvas, generate a thumbnail from it and dispatch a `saveProject` action with the project name and the generated thumbnail.

After it's done we reset the `projectName` value and close the modal window.

Launch your app and try to save your drawing to the backend.



Saving the project

Use this `cURL` to check that the project was saved:

```
curl http://localhost:4000/pictures
```

You can also just copy and paste this url into the browser window. It will return the list of projects. You should see your project data there.

Load The Project

In this section will make it possible to load the projects from the server.

Update the types

Open `src/utils/types.ts` and define a `Project` type there:

```
export type Project = {
  image: string
  name: string
  id: string
}
```

Update the `RootState` as well, now it will have to contain a `projectsList` field:

```
export type RootState = {
  currentStroke: Stroke
  strokes: Stroke[]
  historyIndex: number
  modalVisible: ModalState
  projectsList: {
    error?: string
    pending: boolean
    projects: Project[]
  }
}
```

This field will contain the projects list that we'll get from the server and fields for the *loading* and *error* states.

Define the API module

All our project-loading logic will reside in a separate module. Create a new folder `src/modules/projectsList`.

Now let's define the API module. Create the `src/modules/projectsList/api.ts` file and define the `fetchProjectsList` function:

```
export const fetchProjectsList = () => {
  return fetch("http://localhost:4000/projects").then((res) =>
    res.json()
  )
}
```

This function will fetch the data from the backend and return it as a JSON object.

Create a `projectsList` slice

Create a new file `src/modules/projectList/slice.ts` and add these imports there:

```
import { createSlice, createAsyncThunk } from "@reduxjs/toolkit"
import { RootState } from "../../utils/types"
import { fetchProjectsList } from "./api"
```

Define the initial state:

```
const initialState: RootState["projectsList"] = {
  error: undefined,
  pending: false,
  projects: []
}
```

Define the slice:

```
const slice = createSlice({
  name: "projectsList",
  initialState,
  reducers: {},
  extraReducers: (builder) => {
    builder.addCase(getProjectsList.pending, (state) => {
      state.pending = true
    })
    builder.addCase(getProjectsList.fulfilled, (state, action) => {
      state.pending = false
      state.projects = action.payload
      state.error = undefined
    })
    builder.addCase(getProjectsList.rejected, (state) => {
      state.pending = false
      state.error = "Something went wrong"
    })
  }
})
```

Here we define two reducers, one to handle successful data fetching, and another to handle errors.

Export the reducer and the selectors:

```
export const projectsList = slice.reducer

export const projectsListSelector = (state: RootState) =>
  state.projectsList.projects
export const projectsListPendingSelector = (state: RootState) =>
  state.projectsList.pending
export const projectsListErrorSelector = (state: RootState) =>
  state.projectsList.error
```

Add the reducer to the store:

```
import { configureStore, ThunkAction, Action } from "@reduxjs/toolkit"
import { currentStroke } from "./modules/currentStroke/slice"
import { modalVisible } from "./modules/modals/slice"
import { projectsList } from "./modules/projectsList/slice"
import historyIndex from "./modules/historyIndex/slice"
import strokes from "./modules/strokes/slice"
import logger from "redux-logger"
import { RootState } from "./utils/types"

export const store = configureStore({
  reducer: {
    historyIndex,
    strokes,
    currentStroke,
    modalVisible,
    projectsList
  },
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware().concat(logger)
})

export type AppThunk = ThunkAction<
  void,
  RootState,
  unknown,
```

```
  Action<string>
>
```

Now we can define the thunk that will fetch the projects list. Open `src/modules/projectsList/sl` and add the following there:

```
export const getProjectsList = createAsyncThunk(
  "GET_PROJECTS_LIST",
  async () => {
    return fetchProjectsList()
  }
)
```

Here we call the `api` and then if we get the data, `dispatch` it through the `getProjectListSuccess` action.

Now let's define the selector. In the same file define this function:

```
export const projectsList = slice.reducer
```

Load the selected project

To load the selected project let's first define an API function. Open `src/modules/strokes/api.ts` and add a new function there:

```
export const getProject = (projectId: string) => {
  return fetch(`http://localhost:4000/projects/${projectId}`).then(
    (res) => res.json()
  )
}
```

Now we need to define the `loadProject` thunk, we'll do it in the `src/modules/strokes/slice.ts`:

```
import { getProject, newProject } from "./api"
  // ...
export const loadProject = createAsyncThunk(
  "LOAD_PROJECT",
  async (projectId: string) => {
    try {
      const { project } = await getProject(projectId)
      return project.strokes
    } catch (err) {
      console.log(err)
    }
  }
)
```

Here we use the `getProject` API method to load the project data.

Note that our `loadProject` returns the value that it gets from the server. This is a neat feature of Redux Toolkit thunks. When you return the data - the thunk authomatically dispatches it using a generated action. Actually it dispatches an automatic action in three cases:

- `loadProject.pending` - You've started loading data
- `loadProject.fulfilled` - You got the data
- `loadProject.rejected` - There was an error

Add a new case to the slice to handle the `loadProject.fulfilled` action:

```
builder.addCase(loadProject.fulfilled, (state, action) => {
  return action.payload
})
```

Show the list of projects

Now let's present the user with the list of loaded projects.

Define the styles for the project cards. Open `src/index.css` and add the following CSS classes:

```
.projects-container {
  overflow: auto;
  max-width: 600px;
  height: 400px;
  display: flex;
  flex-direction: row;
  flex-wrap: wrap;
  justify-content: flex-start;
  padding: 0 10px;
  width: 600px;
}

.project-card {
  width: 100px;
  height: 100px;
  margin: 20px;
  cursor: pointer;
  text-align: center;
}

.project-card img {
  width: 100px;
  height: 100px;
  margin-bottom: 10px;
}
```

Now let's update the `ProjectsModal` component. Open `src/ProjectsModal.tsx` and make these imports:

```
import React, { useEffect } from "react"
import { useDispatch, useSelector } from "react-redux"
import { hide } from "./modules/modals/slice"
import {
  getProjectsList,
  projectsListSelector
} from "./modules/projectsList/slice"
import { loadProject } from "./modules/strokes/slice"
```

Add the `projectsListSelector` to the `ProjectsModal` component body:

```
const projectsList = useSelector(projectsListSelector)
```

Now define the `useEffect` with the following contents before the layout:

```
useEffect(() => {
  dispatch(getProjectsList())
}, [])
```

Here we dispatch the `fetchProjectsList` thunk. It will get the list of projects from the backend and then save the value to the store.

Define the `onLoadProject` event handler:

```
const onLoadProject = (projectId: string) => {
  dispatch(loadProject(projectId))
  dispatch(hide())
}
```

We'll call this method when the user clicks on the project. Inside of this method we dispatch an action that loads the selected project and then we close the modal.

Now let's update the layout, add this code below the div with `.title-bar` class:

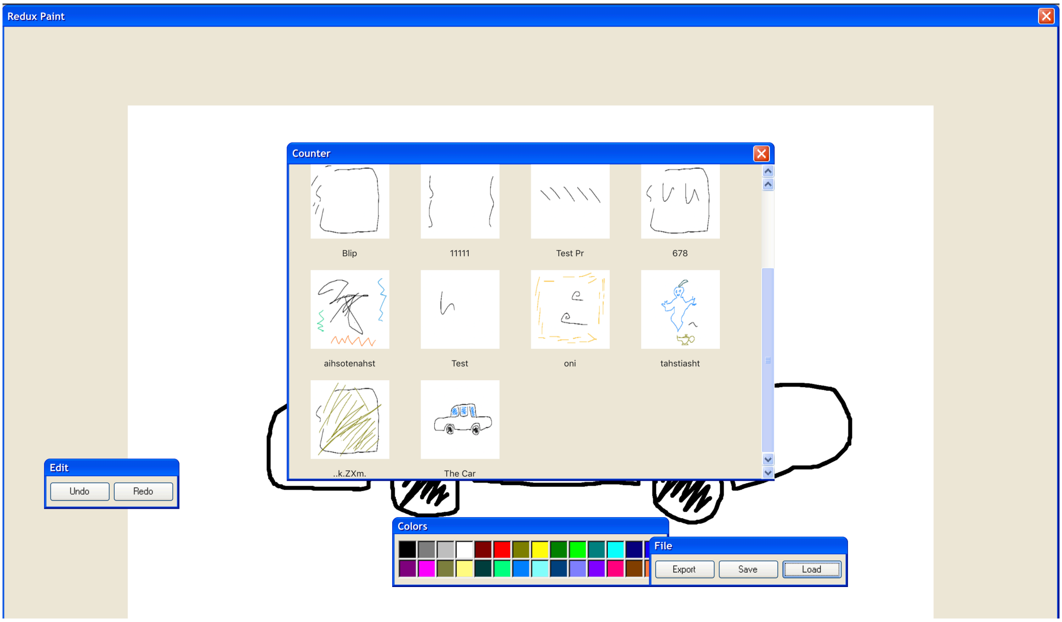
```
<div className="projects-container">
  {(projectsList || []).map((project) => {
    return (
      <div
        key={project.id}
        onClick={() => onLoadProject(project.id)}
        className="project-card"
      >
        <img src={project.image} alt="thumbnail" />
        <div>{project.name}</div>
      </div>
    )
  })}
</div>
```

Update the `App` component

We need to add a small change to on of the `App` component's `useEffect` blocks. Find the `useEffect` that redraws the strokes when the `historyIndex` changes and add the `strokes` to the dependencies array:

```
}, [historyIndex, strokes])
```

Launch the app and verify that you can save and load the projects.



Loading the project

Congratulations! You have a fully functional Redux+Typescript app!

Static Site Generation and Server-Side Rendering Using Next.js

Introduction

So far, we have been creating single-page applications[159] (SPA). A single-page application does not reload the whole page. Instead, it fetches new data and re-renders only parts of the page that need to be updated. All this happens on the same page, hence the name "single-page application".

There is a caveat in this flow, though. If all the data fetching and re-rendering only happens in a user's browser, we can't make all pages in our application detectable by search engines. The vast majority of search robots won't wait until the real content of an application appears. Instead, they will read the content of the HTML we serve them at the start, which is almost empty.

This is not acceptable for an application that hugely relies on its content, such as a blog platform or a news site. This is where pre-rendering[160] comes in.

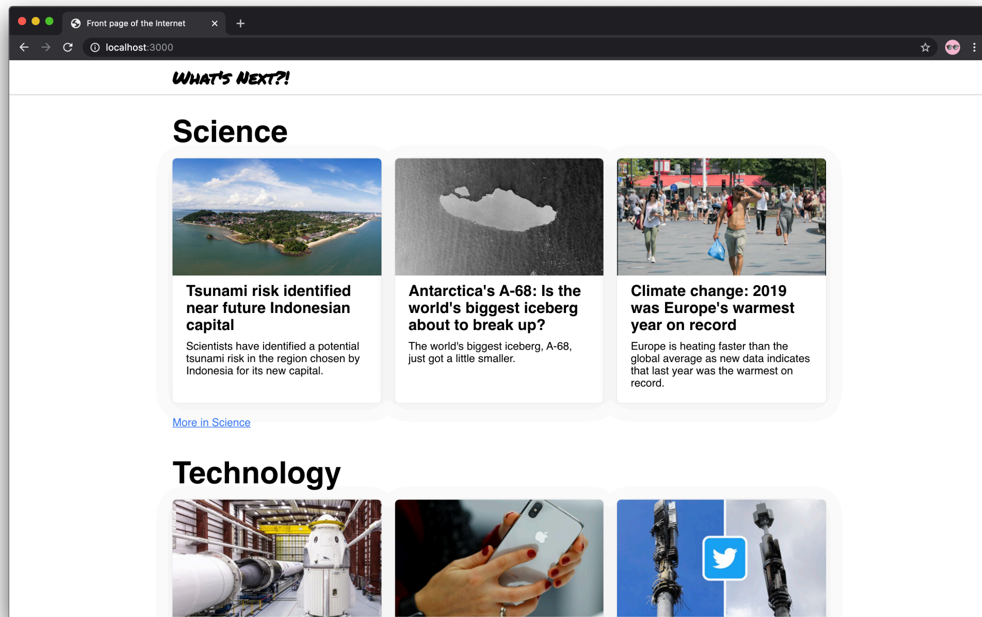
What We're Going to Build

To understand the advantages of pre-rendering, we will create a news site application. We will grab the news and images from the BBC website[161] and create an application that will have pre-rendered pages with content on them.

[159] https://en.wikipedia.org/wiki/Single-page_application
[160] https://nextjs.org/docs/basic-features/pages#pre-rendering
[161] https://www.bbc.com

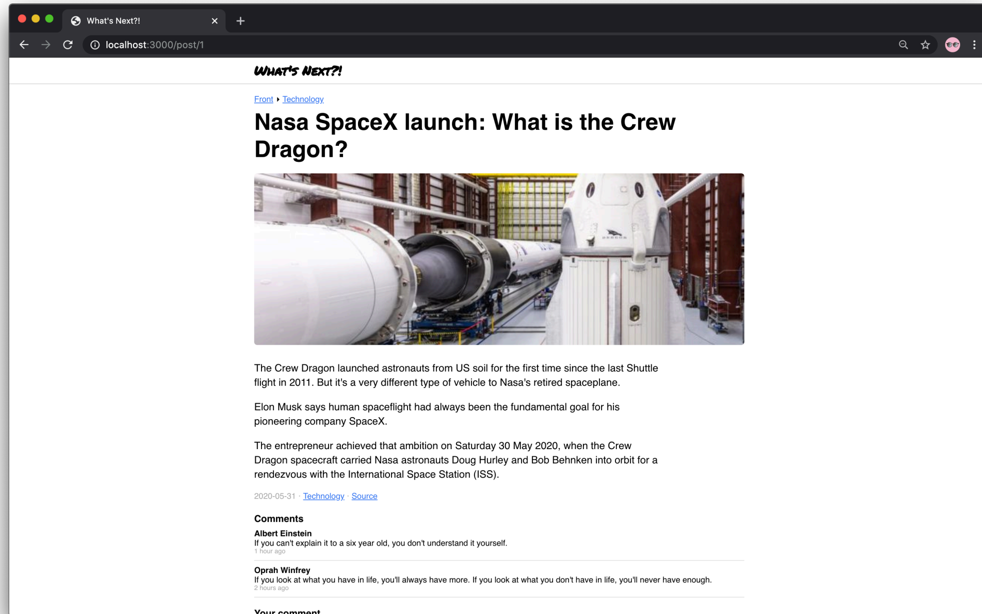
We will statically generate the front page and the post categories pages and render the individual post pages on the server. We will also use Redux to create a comment form that will be hydrated the when used on client.

The main page will look like this:



A completed news site

A post page will look like this:



A post page of the application

The completed application code is located in `code/05-next-ssg/completed`.

Unzip the archive and `cd` to the application folder:

```
cd code/05-next-ssg/completed
```

Once there, install dependencies and launch the application:

```
yarn && yarn dev
```

The application should now open in the browser. If it didn't, navigate to http://localhost:3000 to open it manually.

Pre-Rendering

As we said earlier, serving empty pages is not acceptable for an application that hugely relies on its content. What we want to do is pre-render pages of our application so that they are served with content.

There are 2 major ways to pre-render pages: server-side rendering and static site generation.

Server-side rendering

When server-side rendering[162] (SSR) is used, the server renders real HTML for every page request it gets. In our application, the server would render HTML for each post page, section page, etc.

SSR doesn't require us to store each page as an HTML file on the server. Instead, we can use middleware that fetches real data from a backend API, renders a page that we want to send as a response, fills it with data fetched earlier, and sends the whole HTML to the client.

Each page comes with the minimal necessary JavaScript code. When a page is loaded by the browser, its JavaScript code runs and makes the page interactive. This process, known as hydration[163], resurrects a previously "frozen" application.

Static site generation

Static site generation[164] (SSG) involves generating HTML once, at build time. Technically this means we will have all the real HTML files for each page.

SSG makes responses faster since it doesn't need to render every page every time. However, it is hard to use SSG in some cases. Basically, we should ask ourselves: "Can we pre-render this page ahead of a user's request?" If the answer is yes, then we should choose SSG.

We will use both SSG and SSR. We will explore the differences between them a bit later.

[162] https://nextjs.org/docs/basic-features/pages#server-side-rendering
[163] https://nextjs.org/docs/basic-features/pages#pre-rendering
[164] https://nextjs.org/docs/basic-features/pages#static-generation-recommended

Next.js

We're going to use Next.js[165] (a.k.a. Next), a framework for creating React applications.

We chose Next because it has a clean API and provides all the features we're going to need for our purposes, including SSG. In addition, it comes with great documentation and tutorials

Setting Up a Project

Next has a set of instructions[166] for getting started, we will walk through the process of setting up the project manually.

Create `news-site` directory that will contain our project:

```
mkdir news-site
```

Inside of it create 2 directories:

- `pages` where Next will search for pages[167] of our application (we will talk about pages in detail later).
- `public` for static resources[168] like images, stylesheets, etc.

```
cd news-site
mkdir pages
mkdir public
```

Initialize the project and install the dependencies:

[165] https://github.com/zeit/next.js/
[166] https://nextjs.org/docs/getting-started
[167] https://nextjs.org/docs/basic-features/pages
[168] https://nextjs.org/docs/basic-features/static-file-serving

```
yarn init -y
yarn add next react react-dom
```

Once the project is initialized, add the following scripts to the `scripts` section in the `package.json` file:

```
  "scripts": {
    "dev": "next",
    "build": "next build",
    "start": "next start"
  },
```

Here’s what these scripts will do:

- `dev` will run a development environment, we’ll use it most frequently.
- `build` will build our application and generate rendered pages.
- `start` this script starts applications on production servers. We won’t use it in this chapter

Adding TypeScript

By default, Next uses JavaScript. To integrate TypeScript, we need to performa additional steps.

Add a few more development dependencies:

```
yarn add --dev typescript @types/react @types/node
```

Create an empty `tsconfig.json` file in the project root:

```
touch tsconfig.json
```

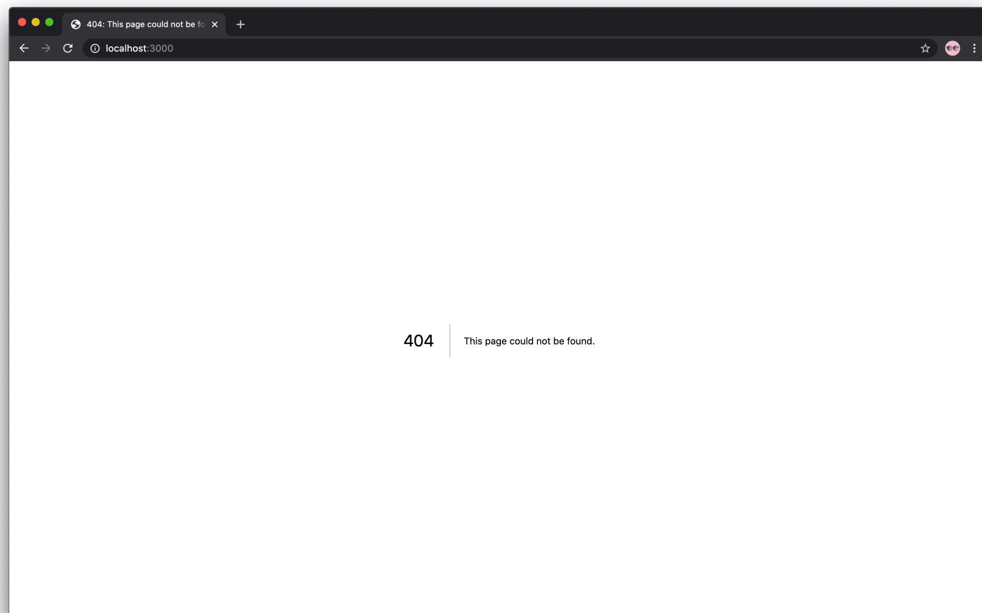
Next will add the configuration to it automatically when we run:

```
yarn dev
```

This command should open the application in the browser. If it didn't, navigate to http://localhost:3000 and open it manually.

Creating A First Page

When opened in the browser, the application will show error 404:



"Not found" error shown by default

This is fine. Next renders error 404 because we haven't created any pages yet. Time to fix this!

A page[169] in Next is a React Component exported from a `.js`, `.jsx`, `.ts`, or `.tsx` file in the `pages` directory. This is why we have created that directory — to populate it

[169] https://nextjs.org/docs/basic-features/pages

with page components. We can think of them as of containers that are associated with specific URLs.

In Next, routing is based on the file structure inside of the `pages` directory. For example, `pages/index.tsx` will be rendered when the user requests the main page of the site, and `pages/contacts.tsx` will be associated with `/contacts`.

To create our first page, let's create a new file, `pages/index.tsx`, and export a React component from it:

```
import React from "react"
import Head from "next/head"

export default function Front() {
  return (
    <>
      <Head>
        <title>Front page of the Internet</title>
      </Head>
      <main>Hello world from Next!</main>
    </>
  )
}
```

We use a default export here. That's Next requirement for the page components.

Another interesting thing is the `Head` component from `next/head`. This component injects everything we pass it as children to the `head` element of an HTML page. In our case, we only pass the `title` element with the page title. We can also put there `meta`, `link`, and `script` tags if necessary.

As soon as the file has been created, Next should automatically refresh the browser and show a new page that says "Hello world from Next!".

Basic Application Layout

At this point, we want to create a basic application layout with header, footer, and main content blocks. Let's start with the `Center` component. It is a styled component that does only one thing: aligns itself at the center of a page.

For styles, we want to use `styled-components`, install this package:

```
yarn add styled-components @types/styled-components
```

After installing this package, we can start using it in our code.

Create a new folder `components`, inside of it, create a folder called `Center` and there create a file called `style.ts`:

```
import styled from "styled-components"

export const Center = styled.div`
  max-width: 1000px;
  padding: 0 20px;
  margin: auto;

  @media (max-width: 800px) {
    max-width: 520px;
    padding: 0 15px;
  }
`
```

We will use this component to center content in many other places.

Add the index file:

```
export * from "./style"
```

`Header` component

Inside the `components` folder, create a folder called `Header` and there create a file called `Header.tsx`:

```
import Link from "next/link"
import { Center } from "../Center"
import { Container, Logo } from "./style"

export const Header = () => {
  return (
    <Container>
      <Center>
        <Logo>
          <Link href="/">
            <a>What's Next?!</a>
          </Link>
        </Logo>
      </Center>
    </Container>
  )
}
```

We declare the `Header` component with a few dependencies, such as the `Head` component and `style.ts`.

We want to create a `Container` for our `Header` component that will stick to the top of the page and contain all the component's content:

```
import styled from "styled-components"

export const Container = styled.header`
  position: fixed;
  top: 0;
  left: 0;
  right: 0;

  height: 50px;
  padding: 7px 0;

  background-color: white;
```

```
  box-shadow: 0 1px 1px rgba(0, 0, 0, 0.2);
`
```

Then, we create a `Logo`, which is a styled `h1` element. It uses props to get access to a theme, which we will cover a bit later in this section:

```
export const Logo = styled.h1`
  font-size: 1.6rem;
  font-family: ${(p) => p.theme.fonts.accent};

  a {
    text-decoration: none;
    color: black;
  }

  a:hover {
    color: ${(p) => p.theme.colors.pink};
  }
`
```

Add the index file:

```
export * from "./Header"
```

Next's `Link` component

The next dependency we use in `Header` is the `Link` component[170] imported from `next/link`. This component enables client-side transition between routes of our application — basically, between pages[171].

Note the structure of the `Link` we have created. At the top level, we use the `Link` component and provide a `href` attribute to it, and inside we use an `a` element to wrap link contents.

[170] https://nextjs.org/docs/api-reference/next/link
[171] https://nextjs.org/docs/routing/introduction

`Link` requires exactly one element to be passed as a child. When we are unable to pass an `a` element for some reason, we can use other elements or components and force[172] `Link` to pass the `href` prop further. This will be useful later when we use styled links.

Footer Component

Finally, we create a `Footer` component to be placed at the bottom of our application's pages:

```
import { Center } from "../Center"
import { Container } from "./style"

export const Footer = () => {
  const currentYear = new Date().getFullYear()

  return (
    <Container>
      <Center>
        <a href="https://newline.co">Newline.co</a> {currentYear}
      </Center>
    </Container>
  )
}
```

And the styles for it:

[172] https://nextjs.org/docs/api-reference/next/link#if-the-child-is-a-custom-component-that-wraps-an-a-tag

```
import styled from "styled-components"

export const Container = styled.footer`
  text-align: center;
  border-top: 1px solid rgba(0, 0, 0, 0.1);
  padding: 15px;
  height: 50px;
`
```

The footer will display the current year and a link to Newline.co.

We use an `a` element instead of the `Link` component, because `Link` is only used for navigation between application routes. If you try to use it for links to external resources, Next will throw an error.

Add the index file:

```
export * from "./Footer"
```

Custom Document Component

We have created global styles and a theme, but if we look closely at the theme, we can find that the accent font uses the `"Permanent Marker"` font family. This is not the kind of font that every device has, so we need to include it.

We can use Google Fonts to get this font, but first, we need to decide where to put a `link` element that would reference a stylesheet with this font. We could include it in `MyApp`, but with Next, you could use a custom `Document` component[173] instead.

Next's `Document` component not only encapsulates `html` and `body` declarations but can also include initial props[174] for expressing asynchronous server-rendering data requirements. In our case, initial props would be the styles used across the application.

[173] https://nextjs.org/docs/advanced-features/custom-document
[174] https://nextjs.org/docs/api-reference/data-fetching/getInitialProps#context-object

Why not simply render styled components as we usually do? That's a tricky question: since we want to create an application that is rendered on the server and then gets "hydrated" on the client, we make sure that the page markup on the server and on the client is equivalent. Otherwise, we would get an error notifying us that some properties are not the same. That includes the styles and class names, and that is exactly where a custom `Document` component can help us.

It is sometimes difficult to decide what component to use. To see the difference between `App` and `Document`, let's compare them:

	App	**Document**
Shared logic and layout	Yes	Not recommended[175]
Global styles	Yes	Not recommended
Renders on...	Client and server	Server
Event handlers like `onClick`	Will work	Won't work
Dev server needs to restart after change	Yes	Yes
Styled components sheet collection	No	Yes[176]
Global middleware	Page level only	Application level, request level

In addition, a custom `getInitialProps()` function in `App` will disable Automatic Static Optimization in pages that don't use static generation. Meanwhile, a custom `getInitialProps()` in `Document` is not called during client-side transitions and when a page is statically optimized.

Let's now create a blueprint for a custom `Document` component. We need to import `ServerStyleSheet` from `styled-components` to help us collect all styles to be sent to the client. We also import a bunch of entities from `next/document` that we will cover in detail later. For now, we'll focus on `Document`.

[175] https://nextjs.org/docs/advanced-features/custom-document#caveats
[176] https://github.com/vercel/next.js/tree/master/with-styled-components

```
import React from "react"
import { ServerStyleSheet } from "styled-components"
import Document, {
  Html,
  Head,
  Main,
  NextScript,
  DocumentContext
} from "next/document"

export default class MyDocument extends Document {
  // ...
}
```

We create a component called `MyDocument` that extends Next's `Document` component, and then define the `render()` method inside:

```
  render() {
    return (
      <Html>
        <Head>
          <meta
            name="description"
            content="The Next generation of a news feed"
          />
          <link
            href="https://fonts.googleapis.com/css2?family=Permanent+Ma\
rker&display=swap"
            rel="stylesheet"
          />

          {this.props.styles}
        </Head>

        <body>
          <Main />
```

```
          <NextScript />
        </body>
      </Html>
    )
  }
```

We don't use the `html` element - instead, we use the `Html` component imported from `next/document`. This is because `Html`, `Head`, `Main` and `NextScript` are required for the page to be properly rendered. `Html` is the root element, `Main` is a component that will render pages, and `NextScript` is a service component required for Next to work correctly.

Inside `Head` we create a `meta` element with description and a `link` element with a link to fonts from Google Fonts. (If we needed links to other external resources, we would add them here as well.) Then, we render `this.props.styles` — these are the styles collected using `ServerStyleSheet`. We collect them in the `getInitialProps()` method:

```
  static async getInitialProps(ctx: DocumentContext) {
    const sheet = new ServerStyleSheet()
    const originalRenderPage = ctx.renderPage

    try {
      ctx.renderPage = () =>
        originalRenderPage({
          enhanceApp: (App) => (props) =>
            sheet.collectStyles(<App {...props} />)
        })

      const initialProps = await Document.getInitialProps(ctx)

      return {
        ...initialProps,
        styles: (
          <>
            {initialProps.styles}
```

```
            {sheet.getStyleElement()}
          </>
        )
      }
    } finally {
      sheet.seal()
    }
  }
```

Because this method is static, it can be called on the class instead of a class instance: `Document.getInitialProps()`. Note that it takes Next's `DocumentContext` as an argument. This object contains a lot of useful information[177], such as `pathname` for page URL, `req` for request, `res` for response, as well as error object `err` to represent any errors that occurred during rendering.

We extend the initial props with our `styles` prop to make it available in the `render()` method. We create `sheet`, an instance of the `ServerStyleSheet` class that helps collect styles from the whole application. Next, we "remember" the `ctx.renderPage()` method in a constant called `originalRenderPage` to override the original `ctx.renderPage()` method inside the `try-finally` clause.

When overriding it, we use the `sheet.collectStyles()`[178] method and pass the whole rendered application as an argument. This collects all styles that we will later be able to extract by calling `sheet.getStyleElement()`.

We then save the original `initialProps` by calling `Document.getInitialProps()`. We call it as a static method, which explains why we had to make our own component's `getInitialProps()` method static as well.

As a result, this method returns an object that contains all original `initialProps` plus a `styles` prop. The `styles` prop holds a component with `style` elements representing all styles that need to be sent along with page markup.

In the browser, this should look like a single `style` element filled with application styles:

[177] https://nextjs.org/docs/api-reference/data-fetching/getInitialProps#context-object
[178] https://styled-components.com/docs/advanced#example

```
▼<style data-styled="active" data-styled-version="5.1.0"> == $0
  .kQelty{max-width:1000px;padding:0 20px;margin:auto;}
  @media (max-width:800px){.kQelty{max-width:520px;padding:0 15px;}}
  .kQelty{max-width:1000px;padding:0 20px;margin:auto;}
  @media (max-width:800px){.kQelty{max-width:520px;padding:0 15px;}}
  .fVgKQj{position:fixed;top:0;left:0;right:0;height:50px;padding:7px 0;background-color:white;box-
  shadow:0 1px 1px rgba(0,0,0,0.2);}
  .fVgKQj{position:fixed;top:0;left:0;right:0;height:50px;padding:7px 0;background-color:white;box-
  shadow:0 1px 1px rgba(0,0,0,0.2);}
  .lnwELG{font-size:1.6rem;font-family:"Permanent Marker",cursive;}
  .lnwELG a{-webkit-text-decoration:none;text-decoration:none;color:black;}
  .lnwELG a:hover{color:#eb57a3;}
  .lnwELG{font-size:1.6rem;font-family:"Permanent Marker",cursive;}
  .lnwELG a{-webkit-text-decoration:none;text-decoration:none;color:black;}
  .lnwELG a:hover{color:#eb57a3;}
  .juEvvx{text-align:center;border-top:1px solid rgba(0,0,0,0.1);padding:15px;height:50px;}
  .juEvvx{text-align:center;border-top:1px solid rgba(0,0,0,0.1);padding:15px;height:50px;}
  body{margin:0;font-family:Helvetica,sans-serif;-webkit-font-smoothing:antialiased;-moz-osx-font-
  smoothing:grayscale;}
  *,*::after,*::before{box-sizing:border-box;}
  h1,h2,h3,h4,h5,h6{margin:0;}
  a{color:#387af5;}
  a:hover{color:#eb57a3;}
  .main{padding:70px 0 20px;min-height:calc(100vh - 50px);}
</style>
```

Final collected styles

In the `finally` clause we call the `sheet.seal()` method to make sure that the `sheet` object is available for garbage collection[179].

Application Theme

Now it is time to create a theme for our application! Create a new file `/shared/theme.ts` with the following imports:

```
import { createGlobalStyle, ThemeProps } from "styled-components"
```

First of all, we declare an object called `theme` with fonts and colors that we're going to use:

[179] https://styled-components.com/docs/advanced#example

```
export const theme = {
  fonts: {
    basic: "Helvetica, sans-serif",
    accent: '"Permanent Marker", cursive'
  },
  colors: {
    orange: "#f4ae40",
    blue: "#387af5",
    pink: "#eb57a3"
    // Credits: https://colors.lol/fou.
  }
}
```

Then we create global styles for all pages. To do this, we declare a new type `MainThemeProps` that will be used in the generic function `createGlobalStyle()` in the next line:

```
export type MainThemeProps = ThemeProps<typeof theme>
export const GlobalStyle = createGlobalStyle<MainThemeProps>`
```

Then we create basic global styles for `body`, such as headings, links, and the `.main` block:

```
export type MainThemeProps = ThemeProps<typeof theme>
export const GlobalStyle = createGlobalStyle<MainThemeProps>`
  body {
    margin: 0;
    font-family: ${({ theme }) => theme.fonts.basic};
    -webkit-font-smoothing: antialiased;
    -moz-osx-font-smoothing: grayscale;
  }

  *,
  *::after,
  *::before { box-sizing: border-box; }
```

```

  h1, h2, h3, h4, h5, h6 { margin: 0; }
  a { color: ${({ theme }) => theme.colors.blue} }
  a:hover { color: ${({ theme }) => theme.colors.pink} }

  .main {
    padding: 70px 0 20px;
    min-height: calc(100vh - 50px);
  }
`
```

We use this `GlobalStyle` component in `MyApp` to inject styles into pages.

From now on, we will focus on components and integration with Next rather than styles. You can find all styles in the source code alongside the corresponding components.

Custom App Component

Now that we have created all the components we need, let's use them in our application's layout.

What if we just include these components in `pages/index.tsx`? This would work, but then we would have to include them into every new page we're going to create. In addition to inconvenience, this would violate the DRY principle (Don't Repeat Yourself).

For this problem, Next has a solution. We can create a wrapper component for every page that Next is going to render. This component is called `App`[180].

Next uses the `App` component to initialize pages. We can override it and control page initialization, which lets us:

- Persist layout between page changes
- Keep the state when navigating pages
- Inject additional data into pages

[180] https://nextjs.org/docs/advanced-features/custom-app

- Add global CSS

Let's create this component and see how we can use it in our application.

```
import React from "react"
import Head from "next/head"
import { ThemeProvider } from "styled-components"

import { Header } from "../components/Header"
import { Footer } from "../components/Footer"
import { Center } from "../components/Center"
import { GlobalStyle, theme } from "../shared/theme"
```

- `Head` from `next/head` to override page title.
- `ThemeProvider` from `styled-components` to use a theme (we will create a theme under `shared/theme` in a minute).
- All the components we created earlier.

Following the imports, we create a component called `MyApp` and export it. Next will inject two props for us:

- The `Component` prop is the active page. When we navigate between routes, `Component` will change to the new page.
- `pageProps` is an object with the initial props that were preloaded for a page.

We render `Component` inside and pass `pageProps` to it using the spread syntax. In other words, we render the current page and pass all the props that it requires. For example, when we create a category page, `pageProps` will contain posts to render on that page.

We use `Head` and `title` to set a default page title, and `Header` and `Footer` to create a layout. Finally, we wrap all of this in `ThemeProvider` to make sure that every styled component has access to the theme.

```
export default function MyApp({ Component, pageProps }) {
  return (
    <ThemeProvider theme={theme}>
      <GlobalStyle theme={theme} />
      <Head>
        <title>What's Next?!</title>
      </Head>

      <Header />
      <main className="main">
        <Center>
          <Component {...pageProps} />
        </Center>
      </main>
      <Footer />
    </ThemeProvider>
  )
}
```

Front Page

We have now prepared everything to create our first page.

Single post

We'll begin by creating a component to render the news posts. This component will display a full post preview consisting of an image, a title, and a short text description. Let's define styles for this component, create a new file `components/Post/PostCardStyle.ts` with the following content:

```
import styled from "styled-components"

export const Card = styled.a`
  border-radius: 6px;
  overflow: hidden;
  background-color: #fff;
  box-shadow: 0 0 0 1px rgba(0, 0, 0, 0.035),
    0 4px 25px rgba(0, 0, 0, 0.07);
  color: black;
  text-decoration: none;
  transition: all 0.2s;

  &:hover {
    color: black;
    box-shadow: 0 0 0 1px rgba(0, 0, 0, 0.035),
      0 6px 35px rgba(0, 0, 0, 0.2);
    transform: translateY(-2px);
  }
`

export const Figure = styled.figure`
  padding: 56% 0 0;
  margin: 0;
  max-width: 100%;
  position: relative;
  overflow: hidden;
  border-radius: 6px 6px 0 0;

  img {
    max-width: 100%;
    position: absolute;
    top: 0;
    left: 0;
  }
`

```

```
export const Title = styled.h3`
  margin: 10px 20px;
  font-size: 1.4rem;
`

export const Excerpt = styled.div`
  margin: 0 20px 20px;

  & > * {
    margin: 0 0 10px;
  }
`
```

Create a new file `components/Post/PostCard.tsx`:

```
import Link from "next/link"
import { Card, Figure, Title, Excerpt } from "./PostCardStyle"

export const PostCard = () => {
  return (
    <Link href="/post/example" passHref>
      <Card>
        <Figure>
          <img alt="Post photo" src="/image1.jpg" />
        </Figure>
        <Title>Post title!</Title>
        <Excerpt>
          <p>
            Lorem ipsum dolor sit amet, consectetur adipiscing elit,
            sed do eiusmod tempor incididunt ut labore et dolore magna
            aliqua.
          </p>
        </Excerpt>
      </Card>
    </Link>
```

```
  )
}
```

A couple of interesting things here.

First, the `passHref` prop passed to `Link` tells Next to push the `href` prop further to the child of `Link`. This is because we pass a `Card` to the `Link` instead of an `a` element. `Card` is a styled `a` element, so it is treated by `Link` not as an `a`, but something else. Without this prop, an `a` element doesn't get the `href` attribute.

Then, we define `href` prop on `Link` to tell Next what page to redirect to.

In earlier versions of Next (before 10), we needed to define `as` prop as well as `href`. Previously, when working with dynamic routes[181] in Next, we would use "[]" to specify the dynamic part of a route. In our case, it would be `[id]`. The `href` was the name of the page in the pages directory. And the `as` was the URL that will be shown in the browser.

Also, the `as` prop was required for Next to determine which pages were to pre-render at build time. Therefore it was possible to miss pre-rendering of some pages when using dynamic segments in `href`. For example, in Next 9 this was okay:

```
<Link href="/posts/[id]" as={`/posts/${post.id}`} />
```

...and this wasn't:

```
// this would break pre-rendering of that page
<Link href={`/posts/${post.id}`} />
```

Since Next 10 there is no need[182] to specify the `as` prop anymore. So we can safely use just `href` in our `Card` component.

The `src="/image1.jpg"` on the `img` element is a path for an image from our `public` directory. By default, Next serves everything from `public` and makes it accessible right from the `/` path. If we want to render an image, we use the `src` prop with a path to an image relative to the `public` folder's root.

[181] https://nextjs.org/docs/routing/dynamic-routes
[182] https://nextjs.org/blog/next-10#automatic-resolving-of-href

Later in this chapter we will optimize images with the `next/image` component that was introduced in the Next 10.

Define an `index` module that will export everything from the `Post`:

```
export * from "./PostCard"
```

News section

Now let's create a component that will group the news posts into a section. First let's define the styles in the new file `components/Section/style.ts`:

```
import styled from "styled-components"

export const Grid = styled.div`
  display: flex;
  flex-wrap: wrap;
  justify-content: space-between;

  &:after {
    content: "";
    flex: auto;
  }

  &:after,
  & > * {
    width: calc(33% - 10px);
    margin-bottom: 20px;
  }

  @media (max-width: 800px) {
    &:after,
    & > * {
      width: 100%;
```

```
    }
  }
`

export const Title = styled.h2`
  font-size: 2.8rem;
  line-height: 1.1;
  margin: 10px 0 15px;

  @media (max-width: 800px) {
    font-size: 2rem;
  }
`

export const MoreLink = styled.a`
  margin: -20px 0 30px;
  display: inline-block;
  vertical-align: top;
`
```

For now, the `Section` component's props only require a `title`. We will change this later.

```
import { PostCard } from "../Post"
import { Grid, Title } from "./style"

type SectionProps = {
  title: string
}
```

A `Section` itself will contain a `Title` and a `Grid` with a bunch of `Post` cards inside. The cards are hardcoded for now:

```
export const Section = ({ title }: SectionProps) => {
  return (
    <section>
      <Title>{title}</Title>
      <Grid>
        <PostCard />
        <PostCard />
        <PostCard />
      </Grid>
    </section>
  )
}
```

In this project, we're not using `FunctionComponent<>` type since none of our components, except pages, accept children as a prop, and the `FunctionComponent<>` type internally allows to pass children. To make sure that we don't accidentally pass any we will use another notation: the colon after function argument `({ title }: SectionProps)`.

The `Grid` component is a styled component that uses `display: flex` to line up the content inside. The `:after` pseudo-element is required to prevent elements in the last row from wrong positioning[183]:

```
import styled from "styled-components"

export const Grid = styled.div`
  display: flex;
  flex-wrap: wrap;
  justify-content: space-between;

  &:after {
    content: "";
    flex: auto;
  }

```

[183] https://stackoverflow.com/questions/18744164/flex-box-align-last-row-to-grid

```
  &:after,
  & > * {
    width: calc(33% - 10px);
    margin-bottom: 20px;
  }
```

We also use `@media` to define adaptive styles for our grid:

```
  @media (max-width: 800px) {
    &:after,
    & > * {
      width: 100%;
    }
  }
```

Define an `index` module that will export everything from the `Section`:

```
export * from "./Section"
```

News feed

Now create a `Feed` component. Our `Feed` will contain 3 sections with post cards inside. These sections will represent news categories: science, technology, and arts.

```
import { Section } from "../Section"

export const Feed = () => {
  return (
    <>
      <Section title="Science" />
      <Section title="Technology" />
      <Section title="Arts" />
    </>
  )
}
```

Define the `index` module:

```
export * from "./Feed"
```

Update the `Front` component

Let's update our `Front` component. Import the `Feed` and add it to the `main` element:

```
import { Feed } from "../components/Feed"
  // ...
      <main>
        <Feed />
      </main>
```

On the main page, you should now see 3 `Section` components that each contain 3 `Post` cards. However, if we click on any of the `Post` cards, we will see the default 404 page. Before we create a post page, let's create a custom 404 page.

Page 404

To create a custom 404 page[184], we need to create a file called `404.tsx`.

We define styles for our 404 page:

```
import styled from "styled-components"

const Container = styled.div`
  display: flex;
  flex-wrap: wrap;
  justify-content: center;
  align-items: center;
  text-align: center;
`

```

[184] https://nextjs.org/docs/advanced-features/custom-error-page

```
const Main = styled.h2`
  font-size: 10rem;
  line-height: 11rem;
  font-family: ${(p) => p.theme.fonts.accent};
  width: 100%;
`

const Description = styled.div`
  width: 100%;
`
```

We keep them in the same file because Next requires all pages to contain a default export for a component that is a page. This means we cannot create a directory `404` with a file `404/style.ts` and extract styles in that file. Doing so would result in an error when we build our project:

> Build error occurred Error: Build optimization failed: found pages without a React Component as default export in pages/404/style
>
> See https://err.sh/zeit/next.js/page-without-valid-component for more info.

We could extract styles into some form of shared code, but since they're fairly compact, we will keep them around to have everything about this page in one place.

Create a component `NotFound` and make it a default export:

```
const NotFound = () => {
  return (
    <Container>
      <Main>404</Main>
      <Description>Oops! The page not found!</Description>
    </Container>
  )
}

export default NotFound
```

Post Page Template

In our first take on this page, we won't render any content. Instead, we will ensure that we can get an ID of a post to load it from the server later.

To create a page that is responsible for a path with dynamic route segment[185], we should add brackets to the page's file name.

In our case, a new file will be called `[id].tsx` and located in the `pages/post` directory:

```
import { useRouter } from "next/router"

const Post = () => {
  const { pathname, query } = useRouter()

  return (
    <div>
      Pathname: {pathname};<br />
      Post Id: {query.id}.
    </div>
  )
}

export default Post
```

There's nothing special in this file so far. It uses the `useRouter()` hook[186] though, so let's see what it does.

`useRouter()` is a hook that provides access to the `router` object[187] that contains 2 useful values:

- `pathname` is the current route. This is the path of the page in the `pages` directory.
- `query` is a query string parsed to the object. It contains the `id` of the current post that we will later use for loading data.

[185] https://nextjs.org/docs/routing/dynamic-routes
[186] https://nextjs.org/docs/api-reference/next/router#userouter
[187] https://nextjs.org/docs/api-reference/next/router#router-object

Backend API Server

Before we continue, let's recall how our static site should work.

We have a bunch of pages that we want to pre-render. Pre-rendering should occur once at build time, and then generated pages should be sent as responses to requests.

In order to be able to generate these pages, we need data to inject in them. We can get this data in various ways:

- From the file system (for example, from `.md` files)
- Directly from a remote database
- From a backend server's API

Next has a great example[188] that shows how to work with the file system. However, we will create a backend server and fetch data from its API.

Let's install the required dependencies:

```
yarn add body-parser concurrently cors express node-fetch ts-node
```

Now let's update our `scripts` section:

```
"scripts": {
  "build": "next build",
  "start": "next start",
  "serve": "ts-node -O '{\"module\": \"commonjs\"}' ./server/index.ts",
  "dev": "concurrently --kill-others \"yarn serve\" \"next\""
},
```

Server setup

We've added a new script, `serve`, that sets up a server and updates the `dev` script to run `serve` and `next` at the same time. The `serve` script will run a Node.js server using a file called `server/index.ts`. Let's create it:

[188] https://nextjs.org/docs/basic-features/data-fetching#simple-example

```
import express from "express"
import cors from "cors"
import bodyParser from "body-parser"

const categories = require("./categories.json")
const posts = require("./posts.json")
const app = express()

app.use(cors())
app.use(bodyParser.json())
```

In this file, we import both data and all the packages we're going to use. We could use a database such as MongoDB, but for the sake of simplicity we will read data right from JSON files. You can find these files in the `05-next-ssg/05.13-backend-api-server/server` directory.

We use the `cors` package to enable sending requests to the server from a different localhost port. We also use `body-parser` to simplify parsing data from request bodies later on.

Post data and type

We are going to use predefined JSON files for our data. We'll use the `posts.json` file to store the data for our posts and the `categories.json` to store the list of category names. You can find these files in the `05-next-ssg/completed/server` directory. Copy them to your project `server` directory.

The `categories.json` should look like this:

```
["Science", "Technology", "Arts"]
```

Let's take a quick look at `posts.json` to see what kind of structure a single post will have. A post is an object with an ID, metadata, text content, and an image:

```
{
  "id": 1,
  "title": "Post title",
  "date": "2020-04-23",
  "category": "Technology",
  "source": "Link to original post or source",
  "image": "Link to image",
  "lead": "Lead paragraph",
  "content": "Text content of this post"
}
...
```

With that in mind, let's design a TypeScript type representing a post, which we will later use in client and server code. We create a file called `types.ts` in the `shared` directory:

```
export type UriString = string
export type UniqueString = string
export type EntityId = number | UniqueString

export type Category = "Technology" | "Science" | "Arts"
export type DateIsoString = string
```

Inside this file, we create a union type `Category`. We also create a few common type aliases (`UriString`, `UniqueString`, `EntityId` and `DateIsoString`) to make types more expressive in describing the intent of our code. We then use all these types to describe the `Post` type:

```
export type Post = {
  id: EntityId
  date: DateIsoString
  category: Category
  title: string
  lead: string
  content: string
  image: UriString
  source: UriString
}
```

API endpoints

We now want to create API endpoints to make data accessible via `GET` requests:

```
const port = 4000

app.get("/posts", (_, res) => {
  return res.json(posts)
})

app.get("/categories", (_, res) => {
  return res.json(categories)
})

app.listen(port, () =>
  console.log(`DB is running on http://localhost:${port}!`)
)
```

We set up port 4000 for this server and create 2 endpoints: `/posts` and `/categories`. When a client sends a request to `http://localhost:4000/posts`, it will get the list of posts as a response. Same goes for `/categories`: a request sent to `http://localhost:4000/catego` results in a response with the list of categories.

Frontend API Client

Now that we have created a server API, we can create a frontend client for that API. Let's create a new directory, `api`, with 2 files in it: `config.ts` and `summary.ts`.

`config.ts` will contain configuration settings for our requests. The `baseUrl` setting will help us reduce duplication across our request functions:

```
export const config = {
  baseUrl: "http://localhost:4000"
}
```

`summary.ts` will contain functions for fetching data for the main page from our server:

```
import fetch from "node-fetch"
import { Post, Category } from "../shared/types"
import { config } from "./config"

export async function fetchPosts(): Promise<Post[]> {
  const res = await fetch(`${config.baseUrl}/posts`)
  return await res.json() as Promise<Post[]>
}

export async function fetchCategories(): Promise<Category[]> {
  const res = await fetch(`${config.baseUrl}/categories`)
  return await res.json() as Promise<Category[]>
}
```

When Next builds a project, it runs outside of the browser environment, where it does not have access to the `fetch()` function. The `node-fetch` package provides the `fetch()` function in the Node environment.

Then there are two async functions that both return `Promise` objects:

- `fetchPosts()` requests `/posts` and returns a promise of `Post[]`.

- `fetchCategories()` requests `/categories` and returns a promise of `Category[]`.

We will use these functions to fetch and pre-fetch data for the main page.

Updating The Main Page

Now that functions for data fetching are ready, we can use them to fetch data on the main page.

Make the imports:

```
import { Post, Category } from "../shared/types"
import { Feed } from "../components/Feed"
import { fetchPosts, fetchCategories } from "../api/summary"
```

Add the `posts` and `categories` as props:

```
type FrontProps = {
  posts: Post[]
  categories: Category[]
}
```

Use this new type for the `Front` component props:

```
export default function Front({ posts, categories }: FrontProps) {
  return (
    <>
      <Head>
        <title>Front page of the Internet</title>
      </Head>

      <main>
        <Feed posts={posts} categories={categories} />
      </main>
    </>
  )
}
```

We also change the `Feed` component's API to make it accept posts and categories as props. Before we update it, let's take a look at how we can pre-render this page.

Fetching data

Next has a concept of static props[189] that are injected to a page component at build time. In our case, categories and posts for the main page will be represented as static props.

In order to tell Next that we want to fetch some data and pre-render a page, we export an `async` function called `getStaticProps()`:

```
export async function getStaticProps() {
  const categories = await fetchCategories()
  const posts = await fetchPosts()
  return { props: { posts, categories } }
}
```

This function makes 2 requests to our backend API: `fetchCategories()` fetches categories for the main page, and `fetchPosts()` fetches posts. Then, we return an object with `props` that contain `categories` and `posts`.

This object is going to be injected as `Front` component's props, making them available inside the component. We should be aware that `getStaticProps()` only runs on the server side. It will never run on the client and won't even be included in a bundle for the browser.

Updating `Feed`

It is now time to update the `Feed` component, since we want to pass the props from the `Front` page.

[189] https://nextjs.org/docs/basic-features/data-fetching#getstaticprops-static-generation

```
import { Post, Category } from "../../shared/types"
  // ...
type FeedProps = {
  posts: Post[]
  categories: Category[]
}
```

We start by declaring an type called `FeedProps` and accessing the props inside the component:

```
import { Section } from "../Section"
  // ...
export const Feed = ({ posts, categories }: FeedProps) => {
  return (
    <>
      {categories.map((currentCategory) => {
        const inSection = posts.filter(
          (post) => post.category === currentCategory
        )

        return (
          <Section
            key={currentCategory}
            title={currentCategory}
            posts={inSection}
          />
        )
      })}
    </>
  )
}
```

Then we iterate over each category and filter posts for it. Finally, we render a `Section` for each category and pass `title` and `posts` for this category as props.

Updating `Section`

The `Section` component needs to be updated as well.

We start by declaring a `SectionProps` type and accessing the props inside the component:

```
import { Post } from "../../shared/types"
  // ...

type SectionProps = {
  title: string
  posts: Post[]
}
```

Then we render `Title` and `Grid` with `Post` cards inside:

```
import { PostCard } from "../Post"
import { Grid, Title } from "./style"
  // ...
export const Section = ({ title, posts }: SectionProps) => {
  return (
    <section>
      <Title>{title}</Title>
      <Grid>
        {posts.map((post) => (
          <PostCard key={post.id} post={post} />
        ))}
      </Grid>
    </section>
  )
}
```

Updating `Post` card

Update the `Post` card component. Open the `components/Post/PostCard.tsx`, update the imports and the props type:

```
import Link from "next/link"
import { Post } from "../../shared/types"
import { Card, Figure, Title, Excerpt } from "./PostCardStyle"

type PostCardProps = {
  post: Post
}
```

We declare a new type `PostProps` with one field, `post`.

Update the component layout:

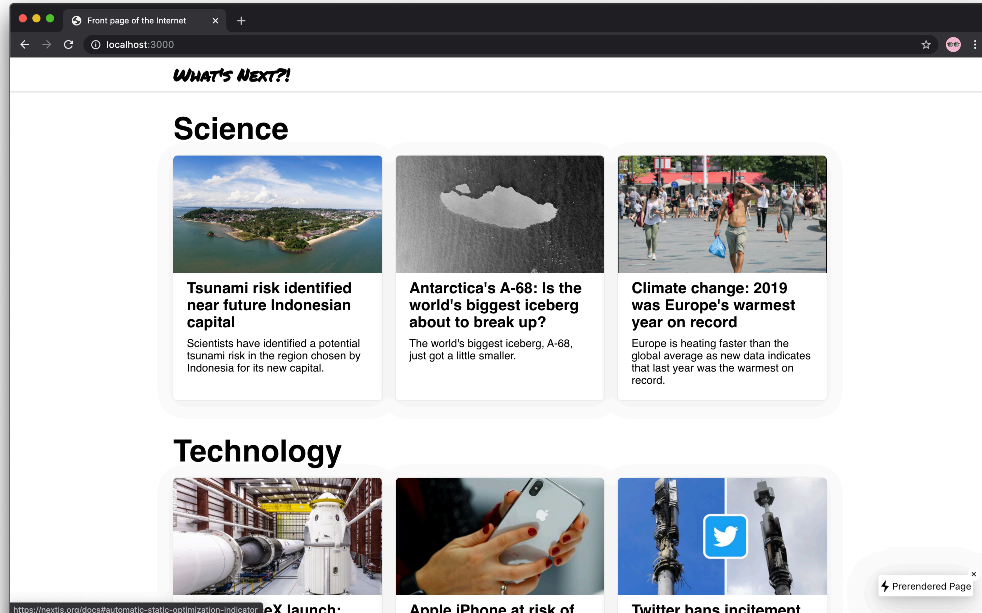
```
export const PostCard = ({ post }: PostCardProps) => {
  return (
    <Link href={`/post/${post.id}`} passHref>
      <Card>
        <Figure>
          <img alt={post.title} src={post.image} />
        </Figure>
        <Title>{post.title}</Title>
        <Excerpt>{post.lead}</Excerpt>
      </Card>
    </Link>
  )
}
```

Here we render a `Link` with 2 props:

- `href` specifies the path to our `post/[id].tsx` page.
- `passHref` forces Next to pass `href` further to a child component.

We also render an image, a title and lead text from the `post`.

Run `yarn dev` and see the result!



Statically generated front page

As we can see, the front page displays categories fetched from the server. Each category contains a list of posts for that category that was also fetched from our backend API.

Pre-Render Post Page

Post API

First, we need to create an API endpoint to get data for a single post. Open `server/index` and import the `Post` type:

```
import { Post } from "../shared/types"
```

Then define a new server endpoint to get a single post:

```
app.get("/posts/:id", (req, res) => {
  const wantedId = String(req.params.id)
  const post = posts.find(({ id }: Post) => String(id) === wantedId)
  return res.json(post)
})
```

A new endpoint, `/posts/:id`, extracts the `id` of a requested post, searches for the post with this `id` in the list of all posts, and returns what it found.

Define the function to fetch that data. Create `api/post.ts` with the following code:

```
import fetch from "node-fetch"
import { Post, EntityId } from "../shared/types"
import { config } from "./config"

export async function fetchPost(id: EntityId): Promise<Post> {
  const res = await fetch(`${config.baseUrl}/posts/${id}`)
  return await res.json()
}
```

This `fetchPost()` function takes an `EntityId` of a post and returns a `Promise` of a `Post`. That's it!

Static props and static paths on the post page

Since the `Post` component will accept data via props, we want to declare a props type:

```
import { GetStaticProps } from "next"
import { useRouter } from "next/router"
import { fetchPost } from "../../api/post"
import { Post as PostType } from "../../shared/types"
import { Loader } from "../../components/Loader"
import { postPaths as paths } from "../../shared/staticPaths"
import { PostBody } from "../../components/Post/PostBody"

type PostProps = {
  post: PostType
}
```

As this page is also going to be pre-rendered, we create the `getStaticProps()` function:

```
export const getStaticProps: GetStaticProps<PostProps> = async ({
  params
}) => {
  if (typeof params.id !== "string") throw new Error("Unexpected id")
  const post = await fetchPost(params.id)
  return { props: { post } }
}
```

We check if `params.id` is a `string` because this field can also be an array of strings.

We import `GetStaticProps` from `next` to declare this function's arguments types and the returned result.

We pass a `context object`[190] as an argument to this function. It contains the `params` object with route parameters for pages that use dynamic routes. Since our page has a dynamic segment (`[id]`), this object has an `id` property with a value equal to the `id` of the current post, which we will use to fetch data.

[190] https://nextjs.org/docs/basic-features/data-fetching#getstaticprops-static-generation

Static paths

There is another exported function, `getStaticPaths()`. This function determines[191] which paths should be rendered to HTML at build time:

```
export async function getStaticPaths() {
  return { paths, fallback: true }
}
```

This function returns an object with 2 fields: `fallback` and `paths`.

`fallback` is set to `true`. When it's `false`, any paths not returned by `getStaticPaths()` will result in a 404 page. When `true`, Next returns the "fallback" version of these paths.

In our case, the `router.isFallback` property is used to render the `Loader` component (which we'll discuss later). When a user requests a page that is not yet rendered but has a "fallback", they see a `Loader`. Meanwhile in the background, Next statically generates HTML and JSON for the requested path. As soon as the browser receives HTML and JSON, the "fallback" page is replaced with a real rendered page.

The second property is `paths`. This is the list of paths that should be rendered at build time. In our case, we take them from the `shared/staticPaths.ts` file:

```
import { EntityId } from "./types"

type PostStaticParams = {
  id: EntityId
}

type PostStaticPath = {
  params: PostStaticParams
}

const staticPostsIdList: EntityId[] = [1, 2, 3, 4, 5, 6, 7, 8, 9]

```

[191] https://nextjs.org/docs/basic-features/data-fetching#getstaticpaths-static-generation

```
export const postPaths: PostStaticPath[] = staticPostsIdList.map(
  (id) => ({
    params: { id: String(id) }
  })
)
```

In this file, we generate a list of objects that follow the structure `{params: { id: post.id }}` for each post. This is our way of telling Next the IDs of posts that it should pre-render.

Let's now complete our `Post` page component:

```
const Post = ({ post }: PostProps) => {
  const router = useRouter()

  if (router.isFallback) return <Loader />
  return <PostBody post={post} />
}

export default Post
```

We use the `useRouter()` hook to access the `router` object. We then check if `router.isFallback` is `true`. If so, it means that this post hasn't been pre-rendered, and we render the `Loader` component. Otherwise, we render the `PostBody` component.

`Loader` component

The `Loader` component simply displays a message that says `Loading...`:

```
import { Container } from "./style"

export const Loader = () => {
  return <Container>Loading...</Container>
}
```

And the styles for it:

```
import styled from "styled-components"

export const Container = styled.div`
  font-family: ${(p) => p.theme.fonts.accent};
`
```

Don't forget to define the `index.ts` file:

```
export * from "./Loader"
```

`PostBody` component

To render the whole post, we'll create a `PostBody` component. Let's begin with styles for it. Create a new file `components/Post/PostBodyStyle.ts`:

```
import styled from "styled-components"

export const Title = styled.h2`
  font-size: 2.8rem;
  line-height: 1.2;
  margin: 10px 0 20px;

  @media (max-width: 800px) {
    font-size: 1.8rem;
    margin: 15px 0;
  }
```

```
`

export const Figure = styled.figure`
  padding: 35% 0 0;
  margin: 0 0 30px;
  max-width: 100%;
  position: relative;
  overflow: hidden;
  border-radius: 6px;

  img {
    width: 100%;
    height: 100%;
    position: absolute;
    top: 0;
    object-fit: cover;
    object-position: center;
  }

  @media (max-width: 800px) {
    margin-bottom: 20px;
  }
`

export const Content = styled.div`
  font-size: 1.25rem;
  line-height: 1.4;
  max-width: 800px;
`

export const Meta = styled.footer`
  color: ${(p) => p.theme.colors.gray};

  & > * {
    margin-right: 0.3em;
  }
```

```
`
```

that takes `post` as a prop:

```
import Link from "next/link"
import { Post } from "../../shared/types"
import { Title, Figure, Content, Meta } from "./PostBodyStyle"

type PostBodyProps = {
  post: Post
}
```

The component returns a block that starts with the main post info:

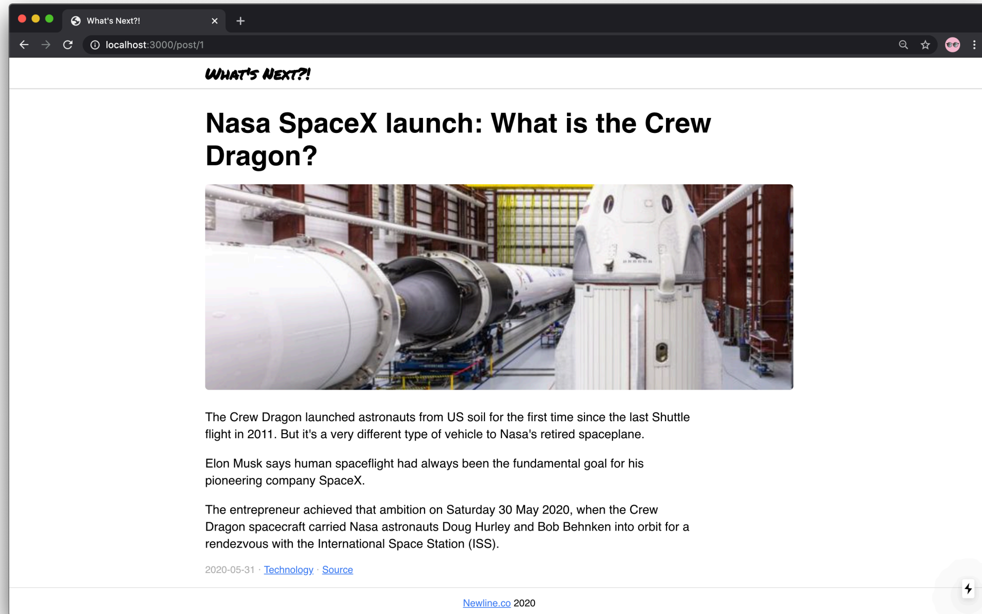
```
export const PostBody = ({ post }: PostBodyProps) => {
  return (
    <div>
      <Title>{post.title}</Title>
      <Figure>
        <img src={post.image} alt={post.title} />
      </Figure>

      <Content dangerouslySetInnerHTML={{ __html: post.content }} />
```

...and proceeds with post metadata:

```
      <Meta>
        <span>{post.date}</span>
        <span>&middot;</span>
        <Link href={`/category/${post.category}`}>
          <a>{post.category}</a>
        </Link>
        <span>&middot;</span>
        <a href={post.source}>Source</a>
      </Meta>
    </div>
  )
}
```

For simplicity, we use `dangerouslySetInnerHTML` in the `Content` component. Since our posts have HTML markup in their `content` fields, we render them right away. In a real-world application, we should consider text preprocessing to avoid XSS and other security vulnerabilities.

Among other things, `Meta` contains a link to the category page. This is the page we're going to create next. For now, let's run `yarn dev` to see what a post page looks like:



Statically generated post page

It's working!

Category Page

The last thing to do before our application is ready is create a category page. It will contain a list of posts from a given category. Again, we will start with API.

Category API

Let's create a new endpoint at `/categories/:id`. We use `id` as category identifier and search for posts that have the `category` field with the same value:

```
app.get("/categories/:id", (req, res) => {
  const { id } = req.params
  const found = posts.filter(({ category }: Post) => category === id)
  const categoryPosts = [...found, ...found, ...found]
  return res.json(categoryPosts)
})
```

Then we repeat the list of found posts 3 times, just to make it look longer than it really is. In a real-world API, we would instead make a request to a database and pull a list of category posts from there.

Now we need to create a function for fetching category data. Create a new file `api/category.ts` with the following contents:

```
import fetch from "node-fetch"
import { Post, EntityId } from "../shared/types"
import { config } from "./config"

export async function fetchPosts(
  categoryId: EntityId
): Promise<Post[]> {
  const url = `${config.baseUrl}/categories/${categoryId}`
  const res = await fetch(url)
  return await res.json()
}
```

The function `fetchPosts()` takes a category identifier `EntityId` and returns a `Promise` of `Post[]`.

Define the `staticPaths` for the category page. Open `shared/staticPaths.ts` and add the following imports:

```
import { EntityId, Category } from "./types"
```

Define the `CategoryStaticParams` and `CategoryStaticPath` types:

```
type CategoryStaticParams = {
  id: Category
}

type CategoryStaticPath = {
  params: CategoryStaticParams
}
```

`Category` page component

Next we want to create the `Category` page component. First of all, let's design props for it. The `Category` component should take a list of `Post` items as the `posts` prop:

```
import { GetStaticProps } from "next"
import { useRouter } from "next/router"
import { Post } from "../../shared/types"
import { fetchPosts } from "../../api/category"
import { Section } from "../../components/Section"
import { Loader } from "../../components/Loader"
import { categoryPaths as paths } from "../../shared/staticPaths"

type CategoryProps = {
  posts: Post[]
}
```

Since we want this page to be pre-rendered as well, we create a `getStaticProps()` function. Inside we call `fetchPosts()` and return a props object with the `posts` property:

```
export const getStaticProps: GetStaticProps<CategoryProps> = async ({
  params
}) => {
  if (typeof params.id !== "string") throw new Error("Unexpected id")
  const posts = await fetchPosts(params.id)
  return { props: { posts } }
}
```

We also want to create a `getStaticPaths()` function to go along with `getStaticProps()`. Again, we set `fallback` to `true` to make sure that pages don't return `404` when they are not pre-rendered:

```
export async function getStaticPaths() {
  return { paths, fallback: true }
}
```

Static paths for this page will be a list of objects with `{params: { id: category }}`. By default, we choose to pre-render 3 categories that are specified in `categoriesToPreRender`:

```
const categoriesToPreRender: Category[] = [
  "Science",
  "Technology",
  "Arts"
]

export const categoryPaths: CategoryStaticPath[] =
  categoriesToPreRender.map((category) => ({
    params: { id: category }
  }))
```

Finally, if the page is not pre-rendered, we display the `Loader` component. Otherwise, a `Section` gets rendered:

```
const Category = ({ posts }: CategoryProps) => {
  const router = useRouter()

  if (router.isFallback) return <Loader />
  return <Section posts={posts} title={String(router.query.id)} />
}

export default Category
```

Updating `Section`

We currently use our `Section` component both on the main page and on the category page. The main page only contains 3 post cards per section. Let's create a link that says "More in this section" on the main page to refer the user to the section page.

First, let's update `SectionProps` and append an optional `isCompact` field that will define whether the "More" link should be rendered:

```
import Link from "next/link"
import { Post } from "../../shared/types"
import { PostCard } from "../Post"
import { Grid, Title, MoreLink } from "./style"

type SectionProps = {
  title: string
  posts: Post[]
  isCompact?: boolean
}
```

Here's how we access this prop:

```
export const Section = ({
  title,
  posts,
  isCompact = false
}: SectionProps) => {
```

Then we conditionally render a `Link` component that leads to a given category:

```
  return (
    <section>
      <Title>{title}</Title>
      <Grid>
        {posts.map((post) => (
          <PostCard key={post.id} post={post} />
        ))}
      </Grid>

      {isCompact && (
        <Link href={`/category/${title}`} passHref>
          <MoreLink>More in {title}</MoreLink>
        </Link>
      )}
    </section>
  )
```

Once again we use `passHref` to force the `Link` component to pass `href` further down to `MoreLink`, which is a styled link:

```
export const MoreLink = styled.a`
  margin: -20px 0 30px;
  display: inline-block;
  vertical-align: top;
`
```

When `isCompact` is not `true`, we shouldn't see this link. We'll handle it later though - for now, we need to update `Feed` to enable rendering this link on the main page:

```
  return (
    <>
      {categories.map((currentCategory) => {
        const inSection = posts.filter(
          (post) => post.category === currentCategory
        )

        return (
          <Section
            key={currentCategory}
            title={currentCategory}
            posts={inSection}
          />
        )
      })}
    </>
  )
}
```

We append the `isCompact` prop to `Section` components inside `map()`. As a result, all sections in `Feed` will now render `MoreLink` and provide access to category pages.

Adding Breadcrumbs

The last thing we would like to show to our users is breadcrumbs on post pages. Breadcrumbs is a component that contains a "link path" from the main page to the current page. In our case, it will show links to the main page and to the category that the current post belongs to.

Let's create the `Breacrumbs` component. Create a new folder `components/Breadcrumbs`. Inside of this folder create the `styles.ts` file. We'll need only one styled component for breadcrumbs, the `Container`:

```
import styled from "styled-components"

export const Container = styled.nav`
  & > * {
    margin-right: 0.3em;
  }
`
```

It is going to be a styled `nav` element.

Now create a new file `component/Breadcrumbs/Breadcrumbs.tsx`. Start with the `BreadcrumbsProps` type, and getting access to the `post` prop:

```
import Link from "next/link"
import { Post } from "../../shared/types"
import { Container } from "./style"

type BreadcrumbsProps = {
  post: Post
}
```

Then we render a `Container` (a styled `nav` element) that contains a couple of links:

```
export const Breadcrumbs = ({ post }: BreadcrumbsProps) => {
  return (
    <Container>
      <Link href="/">
        <a>Front</a>
      </Link>
      <span>□</span>
      <Link href={`/category/${post.category}`}>
        <a>{post.category}</a>
      </Link>
    </Container>
  )
}
```

Open the `components/Post/PostBody.tsx` file and import the breadcrumbs component:

```
import { Breadcrumbs } from "../../components/Breadcrumbs"
```

Finally, we render `Breadcrumbs` in the `PostBody` component right above the post title:

```
    <div>
      <Breadcrumbs post={post} />
      <Title>{post.title}</Title>
```

Now let's make the component accessible outsiede of the module. Create an `index.ts` file inside of the `components/Breadcrumbs` directory:

```
export * from "./Breadcrumbs"
```

Comments and Server-Side Rendering

So far we have been working with content that can be pre-fetched and rendered in advance at build time. What if we wanted to use some dynamic content on our pages, such as comments?

First of all, we wouldn't be able to use static site generation anymore, because users can write comments after we build our site, and we would not be able to display them. This is where server-side rendering (SSR) comes in.

Updates on each request

As we recall, with SSR pages get updated on each request[192]. This is exactly what we need for our comments to be rendered and updated.

We will still get rendered HTML from our server, but this HTML won't include comments at build time. Instead, comments will be rendered "live" at request time on the server.

[192]https://nextjs.org/docs/basic-features/pages#server-side-rendering

Comments backend API

Let's create a mock API for our comments. The comment data structure will look like this:

```
{
  "id": 13,
  "author": "Theodore Roosevelt",
  "content": "Believe you can and you're halfway there.",
  "time": "1 hour ago",
  "post": 7
}
```

This object contains:

- `id`, the comment ID.
- `author`, the name of the author of the comment.
- `content`, comment text.
- `time`, a string with relative time. In a real API it would be a timestamp or ISO string, but for our example, a simple string works fine.
- `post`, the ID of a post that this comment is written for.

You can copy the `server/comments.json` file from the `completed` folder. Put this file in the `server` folder of your project.

We'll need a helper function that will return a list of comments for a given post ID.

```
const found = posts.filter(
  ({ category: id }: Post) => id === req.params.id
)
```

Back in `server/index.ts`, we create another endpoint that returns comments for a given post:

```
app.get("/comments/:post", (req, res) => {
  const postId = Number(req.params.post)
  const found = comments.filter(({ post }) => post === postId)
  return res.json(found)
})
```

We get a post ID from a URL and filter through the `comments` array that we import above:

```
const comments = require("./comments.json")
```

Comment type

Now that the server API is ready, let's create client code. First of all, we want to describe comments in TypeScript terms. To do this, we create a new type in `types.ts` called `Comment`:

```
export type Person = string
export type RelativeTime = string
export type Comment = {
  id: EntityId
  author: Person
  content: string
  time: RelativeTime
  post: EntityId
}
```

It defines the comment data structure in terms of types and uses 2 new types:

- `Person` is just a `string` in our example, but it could be a more complex data structure.
- `RelativeTime`, again, is just a `string` in our example.

Go to `api/comments/fetch.ts` and add the imports:

```
import fetch from "node-fetch"
import { Comment, EntityId } from "../../shared/types"
import { config } from "../config"
```

Now we can create the `fetchComments()` function that takes `postId` as an argument and returns a `Promise<Comment[]>`:

```
export async function fetchComments(
  postId: EntityId
): Promise<Comment[]> {
  const res = await fetch(`${config.baseUrl}/comments/${postId}`)
  return await res.json()
}
```

Components to render comments

Let's create components to render our comments on a page. We will need 3 of them:

- `Comment` for a single comment.
- `CommentForm` to enable users to post new comments.
- `Comments` as a container that will wrap the two components above.

Component for a single comment

The `Comment` component will take a comment as a prop. Its markup will include the author's name, comment text, and creation date.

Add imports:

```
import React from "react"
import { Comment as CommentType } from "../../shared/types"
import { Container, Author, Body, Meta } from "./style"

type CommentProps = {
  comment: CommentType
}
```

...then define the component:

```
export const Comment: React.FC<CommentProps> = ({ comment }) => {
  return (
    <Container>
      <Author>{comment.author}</Author>
      <Body>{comment.content}</Body>
      <Meta>{comment.time}</Meta>
    </Container>
  )
}
```

Here’s the code that we will use to style our comments:

```
import styled from "styled-components"

export const Container = styled.article`
  padding: 10px 0;
`

export const Author = styled.h4`
  display: block;
  font-size: 1rem;
`
export const Body = styled.p`
  margin: 0;
`
```

```

export const Meta = styled.footer`
  color: ${(p) => p.theme.colors.gray};
  font-size: 0.8em;
`
```

Don't forget to create an `index.ts` file in the `components/Comment` directory and export the `Comment` component from there:

```
export * from "./Comment"
```

Comment form

To provide a form for our users to send comments, we'll create a `CommentForm` component. Let's start with the styles for this component. Create a new folder `components/CommentForm` and inside of it create a file `style.ts` with the following code:

```
import styled from "styled-components"

export const Form = styled.form`
  input,
  textarea {
    display: block;
    width: 100%;
    border: 1px solid rgba(0, 0, 0, 0.1);
    box-shadow: none;
    resize: none;
    font-size: 1em;
    padding: 5px;
    border-radius: 2px;
    margin: 10px 0;
  }

```

```
  button {
    border: 0;
    font-size: 1rem;
    padding: 8px 20px;
    border-radius: 6px;
    background-color: #fff;
    box-shadow: 0 0 0 1px rgba(0, 0, 0, 0.035),
      0 4px 25px rgba(0, 0, 0, 0.07);
    transition: all 0.2s;
    cursor: pointer;

    &:hover {
      box-shadow: 0 0 0 1px rgba(0, 0, 0, 0.035),
        0 6px 35px rgba(0, 0, 0, 0.2);
    }
  }
`
```

Now let's create the `CommentForm` component. Create a new file `components/CommentForm/Commen` add the imports and define the props type:

```
import React, { useState, FormEvent } from "react"
import { EntityId } from "../../shared/types"
import { Form } from "./style"
import { submitComment } from "../../api/comments/submit"

type CommentFormProps = {
  post: EntityId
}
```

We pass it a prop with post ID, which will later help us figure out which post this form should be attached to.

Inside `CommentForm` we create 3 fields for the local state: `loading`, `name` and `value`:

- `name` is the author's name.

- `value` is comment text.
- `loading` is a flag that is `true` if a comment is currently being submitted.

```
export const CommentForm: React.FC<CommentFormProps> = ({ post }) => {
  const [loading, setLoading] = useState<boolean>(false)
  const [value, setValue] = useState<string>("")
  const [name, setName] = useState<string>("")
  // ...
}
```

This component returns a `form` element with `input` and `textarea` elements inside:

```
  return (
    <Form onSubmit={submit}>
      <h3>Your comment</h3>
      <input
        type="text"
        name="name"
        value={name}
        placeholder="Your name"
        onChange={(e) => setName(e.target.value)}
        required
      />
      <textarea
        name="comment"
        value={value}
        placeholder="What do you think?"
        onChange={(e) => setValue(e.target.value)}
        required
      />
      {loading ? <span>Submitting...</span> : <button>Submit</button>}
    </Form>
  )
```

We also create an `async` function that should be called when the form is submitted. Here's what this function does:

- It disables the default HTTP form submission behavior with `e.preventDefault()`.
- It sets `loading` to true, which replaces the *Submit* button with a label that says "Submitting…"
- It calls `submitComment()`.
- After receiving a response from the server, it checks if the response status is `201` (which means that something was created), and if so, it refreshes the page to get fresh comments.

```
async function submit(e: FormEvent<HTMLFormElement>) {
  e.preventDefault()
  setLoading(true)

  const { status } = await submitComment(post, name, value)
  setLoading(false)

  if (status === 201) {
    location.hash = "comments"
    location.reload()
  }
}
```

Later we'll make reloading the page unnecessary.

Create a new file `components/CommentForm/index.ts` and export the `CommentForm` component from there:

```
export * from "./CommentForm"
```

API for Adding Comments

Our function `submitComment()` looks like this:

```
import { EntityId, Person } from "../../shared/types"
import { config } from "../config"

export async function submitComment(
  postId: EntityId,
  name: Person,
  comment: string
): Promise<Response> {
  return await fetch(`${config.baseUrl}/posts/${postId}/comments`, {
    method: "POST",
    headers: { "Content-Type": "application/json;charset=utf-8" },
    body: JSON.stringify({ name, comment })
  })
}
```

It takes `postId`, `name` and `comment`, creates an object, converts it to a string using `JSON.stringify()`, and sends it to the server. We include `postId` in the URL of an endpoint that we send a request to.

On the backend, we create a new comment object and respond with status `201`. Right now the code for creating a comment looks more like a mock than real code. In a real-world API, we would save the comment to a database. However, in our example we keep the `comments` array in memory and `push()` a new value to it when we submit a comment:

```
app.post("/posts/:id/comments", (req, res) => {
  const postId = Number(req.params.id)
  comments.push({
    id: comments.length + 1,
    author: req.body.name,
    content: req.body.comment,
    post: postId,
    time: "Less than a minute ago"
  })
  return res.sendStatus(201)
})
```

Adding comments to a page

To inject comments to a page, we want to create a wrapper for the comments section. Let's create a new component, `Comments`.

For starters, we create the `CommentsProps` type where the `comments` field defines an array of comments to render, and the `post` field contains the current post ID:

```
import { Comment as CommentType, EntityId } from "../../shared/types"
import { Comment } from "../Comment/Comment"
import { Container, List, Item } from "./style"
import { CommentForm } from "../CommentForm"

type CommentsProps = {
  post: EntityId
  comments: CommentType[]
}
```

Then we create the `Comments` component itself. It renders each comment as an item of a list and adds a form under this list:

```
export const Comments = ({ post, comments }: CommentsProps) => {
  return (
    <Container id="comments">
      <h3>Comments</h3>
      <List>
        {comments.map((comment) => (
          <Item key={comment.id}>
            <Comment comment={comment} />
          </Item>
        ))}
      </List>
      <CommentForm post={post} />
    </Container>
  )
}
```

This is how we style this component:

```
import styled from "styled-components"

export const Container = styled.section`
  margin: 1.5rem 0;
`

export const List = styled.ul`
  margin: 0;
  padding: 0;
  list-style: none;
  margin-bottom: 20px;
`

export const Item = styled.li`
  list-style: none;
  border-bottom: 1px solid rgba(0, 0, 0, 0.1);
`
```

Just like with the other components we also create the `index.ts` file:

```
export * from "./Comments"
```

We are now ready to add the comments section to a page. We change the `PostProps` type for the post page to include the `comments` field:

```
type PostProps = {
  post: PostType
  comments: Comment[]
}
```

Then we change the component itself to render the `Comments` component. We access the `comments` prop and pass it over to `Comments`:

```
const Post = ({ post, comments }: PostProps) => {
  const router = useRouter()

  if (router.isFallback) return <Loader />
  return (
    <>
      <PostBody post={post} />
      <Comments comments={comments} post={post.id} />
    </>
  )
}

export default Post
```

We add the `id` prop to make sure that when a user submits a comment, their browser scrolls right to the comments section after reload.

The last thing to do is update this statically generated page to use server-side rendering.

Updating a statically generated page to use server-side rendering

In order to enable server-side rendering for a page, we need to export[193] a `getServerSideProps()` function.

Create the `getServerSideProps()` function that returns the `GetServerSideProps` type. Inside, we fetch the current post and invoke `fetchComments()`:

[193] https://nextjs.org/docs/basic-features/data-fetching#getserversideprops-server-side-rendering

```
export const getServerSideProps: GetServerSideProps<PostProps> =
  async ({ params }) => {
    if (typeof params.id !== "string")
      throw new Error("Unexpected id")
    const post = await fetchPost(params.id)
    const comments = await fetchComments(params.id)
    return { props: { post, comments } }
  }
```

We cannot[194] use it along with the `getStaticPaths()` function, so remove the `getStaticPaths()`.

Comments will now be fetched on every page request and no data will be missing.

Update the imports in the `pages/post/[id].tsx` file. First of all, we import the `GetServerSideProps` type from `next`:

```
import { GetServerSideProps } from "next"
```

Then we import the `fetchComments` function from the `api/comments` module:

```
import { fetchComments } from "../../api/comments/fetch"
```

Import the `Comment` type from `shared/types` and the `Comment` component:

```
import { Post as PostType, Comment } from "../../shared/types"
  // ...
import { Comments } from "../../components/Comments"
```

Connecting Redux

The post page currently reloads when a user submits a comment. Let's try to make it work without reloading. In order to do this, we need some kind of store on the client. For this purpose, we will use Redux.

[194] https://nextjs.org/docs/basic-features/data-fetching#use-together-with-getstaticprops

There is a package[195] called `next-redux-wrapper` that will help us connect Redux with Next easier.

First, let's add all the required packages:

```
yarn add next-redux-wrapper react-redux @types/react-redux
```

We don't add `redux` itself because it is included in dependencies for `next-redux-wrapper`. However, it requires[196] `react-redux` as peer dependency, so we'll install it separately.

Configuring a store

Let's take a look at the `store/index.ts` file:

```
import { Store, createStore, combineReducers } from "redux"
import { MakeStore, createWrapper } from "next-redux-wrapper"
import { comments, CommentsState } from "./comments"
import { post, PostState } from "./post"

export type State = {
  post: PostState
  comments: CommentsState
}

const combinedReducer = combineReducers({ post, comments })
const makeStore: MakeStore<Store<State>> = () =>
  createStore(combinedReducer)

export const store = createWrapper<Store<State>>(makeStore, {
  debug: true
})
```

First of all, there is a `State` type that defines the structure of our future state. In our case, since only the post page is dynamic, we will only need a store for `comments` and the current `post`.

[195] https://github.com/kirill-konshin/next-redux-wrapper
[196] https://github.com/kirill-konshin/next-redux-wrapper#installation

`PostState` is typed as `Optional<Post>` and defined in `store/post.ts`. It is optional because when we later use it in a reducer and the default state doesn't correspond to any post yet, we will define it as `null`:

```
export type PostState = Optional<Post>
```

We need an `Optional` type, so let's create it:

```
export type Optional<TEntity> = TEntity | null
```

`CommentsState` is an array of `Comment` items from `store/comments.ts`:

```
export type CommentsState = Comment[]
```

Then there is a `combinedReducer` that contains definitions for `post` and `comments` reducers. We will cover them in a minute.

The `makeStore()` is a function that creates a Redux store. The `MakeStore` type will help the `createWrapper()` function create a wrapper that we will be able to use with our components.

Actions for comments

Let's define types for our reducer and actions for the `comments` state:

```
import { AnyAction } from "redux"
import { HYDRATE } from "next-redux-wrapper"
import { Comment } from "../shared/types"
import { HydrateAction } from "./hydrate"

export const UPDATE_COMMENTS_ACTION = "UPDATE_COMMENTS"

export interface UpdateCommentsAction extends AnyAction {
  type: typeof UPDATE_COMMENTS_ACTION
  comments: Comment[]
```

```
}

export type CommentsState = Comment[]

type CommentsAction = HydrateAction | UpdateCommentsAction
```

We create the `UpdateCommentsAction` interface that extends `AnyAction` from `redux`. We set the `type` field to be of type of the `UPDATE_COMMENTS_ACTION` constant. The second field in this action is `comments`, which is an array of `Comment`.

> We use an interface and not a type even though an action is not a "public API". This is because we need to extend the `AnyAction` and interfaces are better at extension than types. They are better at merging fields than types and extending an interface is faster than using a union. In this project, when extending `AnyAction` we will always use interfaces.

A union type for actions, `CommentsAction`, contains either `UpdateCommentsAction` or `HydrateAction`, which is defined in `store/hydrate.ts`:

```
import { AnyAction } from "redux"
import { HYDRATE } from "next-redux-wrapper"

export interface HydrateAction extends AnyAction {
  type: typeof HYDRATE
}
```

This action has a type of `HYDRATE` that is imported from the `next-redux-wrapper` package. This is a special action that must be used[197] in order to properly reconcile the hydrated state on top of the existing state.

Each reducer must have a handler for this action. This is because every time a user opens a page that has a `getServerSideProps()` function, the `HYDRATE` action is dispatched.

Reducer for comments

With that in mind, let's create our `comments()` reducer:

[197]https://github.com/kirill-konshin/next-redux-wrapper#usage

```
export const comments = (
  state: CommentsState = [],
  action: CommentsAction
) => {
  switch (action.type) {
    case HYDRATE:
      return action.payload?.comments ?? []
    case UPDATE_COMMENTS_ACTION:
      return action.comments
    default:
      return state
  }
}
```

Inside the `HYDRATE` case we see the familiar optional chaining[198] operator `?`, but after it there is another operator: `??`. This is the nullish coalescing[199] operator.

When the whole expression `action.payload?.comments` is `null` or `undefined`, nullish coalescing will tell TypeScript to use the fallback value, which is an empty array.

In our case, because we need to load new comments for a new post, it's fine to simply replace the entire old state with a fresh state when hydration occurs. However, sometimes you can't get away with full refresh. Instead, you should consider comparing states and merging them[200].

The second case handles `UpdateCommentsAction` calls. It replaces existing comments with those in the payload.

As the default value for the state, we provide an empty array.

Reducer for posts

Next, let's create the `post()` reducer:

[198] https://www.typescriptlang.org/docs/handbook/release-notes/typescript-3-7.html#optional-chaining
[199] https://www.typescriptlang.org/docs/handbook/release-notes/typescript-3-7.html#nullish-coalescing
[200] https://github.com/kirill-konshin/next-redux-wrapper#state-reconciliation-during-hydration

```
import { AnyAction } from "redux"
import { HYDRATE } from "next-redux-wrapper"
import { Post, Optional } from "../shared/types"
import { HydrateAction } from "./hydrate"

export const UPDATE_POST_ACTION = "UPDATE_POST"

export interface UpdatePostAction extends AnyAction {
  type: typeof UPDATE_POST_ACTION
  post: Post
}

export type PostState = Optional<Post>

type PostAction = HydrateAction | UpdatePostAction
```

The `UpdatePostAction` interface extends `AnyAction`, defines the `type` field to be of type `UPDATE_POST_ACTION` and `post` to be of type `Post`. The `PostAction` union is either `HydrateAction` or `UpdatePostAction`.

This reducer provides cases for two actions: `HYDRATE` and `UPDATE_POST_ACTION`. When hydration occurs, we either take the `post` from `action.payload` or set the state to `null`. We also provide `null` as the default value for the state — this is what we needed the `Optional<>` type for.

```
export const post = (state: PostState = null, action: PostAction) => {
  switch (action.type) {
    case HYDRATE:
      return action.payload?.post ?? null
    case UPDATE_POST_ACTION:
      return action.post
    default:
      return state
  }
}
```

If the action type is `UpdatePostAction`, we replace the current value with the new one to render a freshly loaded post.

Changing the custom application component

Now that our store is ready, we can connect it to Next's `_app`. First of all, we don't default export the `MyApp()` function anymore. Instead, we default export a wrapped version of it:

```
export default store.withRedux(MyApp)
```

This `store` is the wrapper that we have created earlier:

```
import { store } from "../store"
```

In `next-redux-wrapper` 6.x we needed to define the `MyApp.getInitialProps()` static method. Starting from 7.x we must not extend `MyApp`[201] as we'll be opted out of Automatic Static Optimization[202]. Now we use a regular function component instead.

```
function MyApp({ Component, pageProps }) {
  return (
    <ThemeProvider theme={theme}>
      <GlobalStyle theme={theme} />
      <Head>
        <title>What's Next?!</title>
      </Head>

      <Header />
      <main className="main">
        <Center>
          <Component {...pageProps} />
        </Center>
      </main>
```

[201] https://github.com/kirill-konshin/next-redux-wrapper#usage
[202] https://nextjs.org/docs/messages/opt-out-auto-static-optimization

```
      <Footer />
    </ThemeProvider>
  )
}

export default store.withRedux(MyApp)
```

Updating the post page

Now we need to update the post page. Since we want to store comments and post data in the Redux store, we need to connect this page to the store.

We're going to use the `useSelector()` hook from `react-redux` package to access the store.

```
import React from "react"
import { NextPage } from "next"
import { useSelector } from "react-redux"
import { Loader } from "../../components/Loader"
import { PostBody } from "../../components/Post/PostBody"
import { Comments } from "../../components/Comments"

import { fetchPost } from "../../api/post"
import { fetchComments } from "../../api/comments/fetch"
import { State, store } from "../../store"
import { PostState, UPDATE_POST_ACTION } from "../../store/post"
import {
  CommentsState,
  UPDATE_COMMENTS_ACTION
} from "../../store/comments"
```

The whole page component will look like this:

```
const Post: NextPage = () => {
  const post = useSelector<State, PostState>(({ post }) => post)
  const comments = useSelector<State, CommentsState>(
    ({ comments }) => comments
  )

  if (!post) return <Loader />
  return (
    <>
      <PostBody post={post} />
      <Comments comments={comments} post={post.id} />
    </>
  )
}

export default Post
```

We access the state, destructure it into `post` and `comments` objects, and pass them further as props. Since post data can be `null`, we render the `Loader` component when there's no post to display yet.

If we start our project right now, it won't work because Next doesn't know what data to inject into the store and how to do it on request. We need to use our `store` wrapper to modify the `getServerSideProps()` function:

```
export const getServerSideProps = store.getServerSideProps(
  (store) =>
    async ({ params }) => {
      if (typeof params.id !== "string") {
        throw new Error("Unexpected id")
      }

      const comments = await fetchComments(params.id)
      const post = await fetchPost(params.id)

      store.dispatch({ type: UPDATE_POST_ACTION, post })
```

```
      store.dispatch({ type: UPDATE_COMMENTS_ACTION, comments })

      return null
    }
)
```

Here we use the `store.getServerSideProps()` function that takes a callback which is a higher order function. Inside the callback, we fetch the required data and pass it into the store. The basic idea is the same: we define what data needs to be pre-fetched and rendered on response, but instead of passing it in `Post` component's props, we `dispatch()` actions that update our store with this data.

The `Post` component doesn't take any props at all. It gets all the data from the store using the `useSelector()` hook. Since it doesn't accept any props anymore we can safely return `null` from the `getServerSideProps` callback.

Making the comment form work without reloads

To make the comment form work without page reloads, we need to `dispatch()` some action that will update the store instead of reloading a page.

When we submit a comment to the server, we want to get the data to refresh the comments section on the page. Let's modify our server response: instead of status `201`, it should return the list of comments for the current post.

> In canonical REST APIs `POST` requests should return `201` and an ID of the created entity. By returning the whole list of comments instead, we make our response less canonical but more convenient for us to work with.

We need to update the `return` statement in the `post()` method in `server/index.ts`. We will return all comments for the post with a given `postId`:

```
app.post("/posts/:id/comments", (req, res) => {
  const postId = Number(req.params.id)
  comments.push({
    id: comments.length + 1,
    author: req.body.name,
    content: req.body.comment,
    post: postId,
    time: "Less than a minute ago"
  })
  return res.json(comments.filter(({ post }) => post === postId))
})
```

Go to the `CommentForm` component and add the imports:

```
import { useDispatch } from "react-redux"
  // ...
import { UPDATE_COMMENTS_ACTION } from "../../store/comments"
import { Form } from "./style"
```

In the `CommentForm` component we use the `useDispatch()` hook to get access to the `dispatch()` function. This `dispatch()` is going to be used to dispatch actions as soon as a request is completed:

```
  const dispatch = useDispatch()

  async function submit(e: FormEvent<HTMLFormElement>) {
    e.preventDefault()
    setLoading(true)

    const response = await submitComment(post, name, value)
    const comments = await response.json()
    setLoading(false)
    setValue("")
    setName("")

```

```
    if (response.status === 200) {
      dispatch({ type: UPDATE_COMMENTS_ACTION, comments })
    }
  }

  return (
```

We access all `comments` from the server's `response`. We then use `setValue()` and `setName()` to clear the form, and if the request has succeeded, we dispatch `UPDATE_COMMENTS_ACTION` with the list of `comments` as a payload. This updates the comments store and re-renders the comments section on the page.

The form itself stays the same.

Optimizing Images

Okay, our app is already in a good shape! However, we can even make it better by using optimized images. Next 10 introduced a `next/image` component[203] that can make it so much easier to create adaptive images and convert them into more lightweight formats on the fly! Let's try using it.

In our app, we have 2 components that render images: `PostCard` and `PostBody`. The first one renders a preview image in a posts list, the second one renders the main post image on the post page. We will use different strategies for optimizing both and explain them along the way.

Let's start with `PostBody` component. The first thing to do is to import Next image component:

```
import Link from "next/link"
import Image from "next/image"
import { Post } from "../../shared/types"
import { Breadcrumbs } from "../../components/Breadcrumbs"
import { Title, Figure, Content, Meta } from "./PostBodyStyle"
```

Then, we can replace the old `img` tag with the new `Image` component:

[203] https://nextjs.org/docs/api-reference/next/image

```
      <Figure>
        <Image
          alt={post.title}
          src={post.image}
          loading="lazy"
          layout="responsive"
          objectFit="cover"
          objectPosition="center"
          width={960}
          height={340}
        />
      </Figure>
```

For this component to work, we need to provide a couple of required props:

- `alt`, an alternative text to show when the browser cannot find an image;
- `src`, the default source URL for an image;
- `width` and `height`, the default size for an image.

Don't worry about `width` and `height`, our image will be responsive. We need them for 2 reasons. First of all, they will help Next automatically figure out the aspect ratio of an image. We won't need to use the `padding-top` trick anymore!

Second, the `width` and `height` props reduce cumulative layout shift, because they allocate the place for an image on a page. When the image is loaded it doesn't push the content underneath down.

There are some other props we're passing for the `Image` component as well. Let's review them:

- `loading`, tells the browser how to load an image. When it is set to `lazy` the browser will wait until the image is in the viewport and load only then.
- `layout`, tells Next how to scale an image when the viewport size changes. We set it to `responsive` to make the image adapt to the size of its container when it changes.
- `objectFit` and `objectPosition`, basically, aliases for CSS properties we used earlier.

> We can also use the `fixed` layout to fix image sizes or `intrinsic` to make an image only scale down.

The image is ready, now let's clean up styles a bit. We don't need the image styles anymore because Next will handle them for us, so we can safely remove `img` styles from the `PostBodyStyle.ts`:

```
export const Figure = styled.figure`
  margin: 0 0 30px;
  max-width: 100%;
  position: relative;
  overflow: hidden;
  border-radius: 6px;

  @media (max-width: 800px) {
    margin-bottom: 20px;
  }
`
```

Before we run our dev server and see what Next will output, we need to set up a configuration file[204]. Create a file called `next.config.js` in the root of the project directory and add this configuration:

```
module.exports = {
  images: {
    domains: ["ichef.bbci.co.uk"],
    deviceSizes: [320, 640, 860, 1000]
  }
}
```

This config contains the `images` field that sets up how Next will handle our images. The `domains` array specifies what external domains are allowed to load images from. By default, Next won't let us load an image from external domains.

[204]https://nextjs.org/docs/api-reference/next.config.js/introduction

The `deviceSizes` property tells Next what breakpoints we're going to consider in the app layout. These breakpoints define how to scale images and what images for the browser to load.

By default, Next uses `[640, 750, 828, 1080, 1200, 1920, 2048, 3840]`—that's a lot of breakpoints! For each of them Next creates an image with the corresponding size. So when the `deviseSizes` is not set Next generates `8` different variants for each image. In some cases, `8` variants for each image is too many. In our app, we use `4` different breakpoints because we don't need extra-large images since the app container's `max-width` is 1000px.

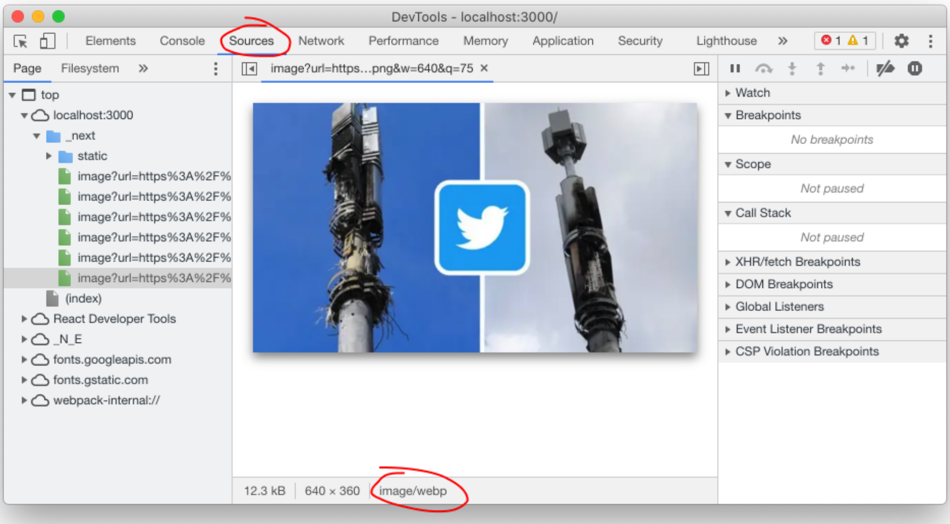
> For `intrinsic` and `fixed` image layouts we should use `imageSizes` instead of `deviceSizes`.

After it's done, we can finally start our server and see what Next produces as a result. If we now inspect the image's HTML we will see that Next wrapped it with a `div` that uses `padding` to imitate the `aspect-ratio` of the image inside. The image itself now has an `srcset` attribute with a bunch of URLs:

```
srcset="
  /_next/image?url=image-name&w=320&q=75 320w,
  /_next/image?url=image-name&w=640&q=75 640w,
  /_next/image?url=image-name&w=860&q=75 860w
  /_next/image?url=image-name&w=1000&q=75 1000w
"
```

These URLs specify all the possible images that the browser can download. The cool thing is the browser knows what image is best to load in a given situation. It will make a decision based on the network quality, device viewport size, screen pixel ratio, and other factors to choose the best option.

Another cool thing is that Next will automatically serve modern image formats like `webp` if the browser supports them. If we inspect an image from the Sources tab we can see that loaded image has `image/webp` format. And all of this with no extra work!



Loaded image is in webp format

Wait a minute? If the browser makes a decision based on `srcset` how can we change it? What if we want to load a smaller image when the viewport is bigger? We can do it as well! Let's update our card preview images and see how we can control them.

Telling Browser What Images to Load

Let's again start with imports and use the `Image` component:

```
import Link from "next/link"
import Image from "next/image"
import { Post } from "../../shared/types"
import { Card, Figure, Title, Excerpt } from "./PostCardStyle"
```

Then replace the old `img` with the new component:

```
        <Figure>
          <Image
            alt={post.title}
            src={post.image}
            loading="lazy"
            layout="responsive"
            objectFit="cover"
            objectPosition="center"
            width={320}
            height={180}
            sizes="(min-width: 1000px) 320px, 100vw"
          />
        </Figure>
```

The basics are the same. We all the properties we used with images in `PostBody` but this time we add another prop called `sizes`.

The `sizes` prop is a way for us to talk to the browser and tell it that we already know what image is the best option for a given viewport. Let's review its value to understand how it works:

```
sizes="(min-width: 1000px) 320px, 100vw"
```

The string contains 2 records divided by a comma. The first one contains a media-query and a number, the last one contains only a number. The media-query specifies the viewport constraint as it does in CSS. The following number is the width of an image that best fits.

Here we mean that whenever the viewport is bigger than 1000px we want the browser to load an image with a width of 320px. Why? Because our preview card is about 300px wide itself at this point and we don't need a 1600px wide image.

Otherwise, load whatever suits the whole viewport width. Why? Because when the viewport is less than 1000px our layout becomes a column where a card takes 100% of the container's width.

> The order of `sizes` records matters. The browser will take only *the first* matching media-query and use it. That's why the default value should be last.

Now we only need to clean up our styles and remove old `img` styles from the `PostCardStyle.ts`:

```
export const Figure = styled.figure`
  margin: 0;
  max-width: 100%;
  position: relative;
  overflow: hidden;
  border-radius: 6px 6px 0 0;
`
```

Building Project

Now it is finally time to build our project. If we run it right now, we won't see any build artifacts in the project directory. That's because Next puts artifacts in the `.next` directory by default.

Next offers an option to export generated code[205] to the `out` directory via `next export`. But we want to make `build` the build destination directory.

> The `next/image` works only with a next application live-running on a server[206] via `next start`. If we want to export our app as a static site we need to either specify a loader[207] that will process images or to replace `next/image` with another component. For brevity, in this step, we will use standard `img` tags for images as we did in step 8.

To do that, we create `next.config.js`, a configuration file[208] for the Next framework. A configuration option that defines a custom build directory[209] is called `distDir`. Let's set `build` as the value for that option:

[205]https://nextjs.org/docs/advanced-features/static-html-export
[206]https://github.com/vercel/next.js/issues/18356
[207]https://nextjs.org/docs/basic-features/image-optimization#loader
[208]https://nextjs.org/docs/api-reference/next.config.js/introduction
[209](https://nextjs.org/docs/api-reference/next.config.js/setting-a-custom-build-directory)

```
module.exports = {
  images: {
    domains: ["ichef.bbci.co.uk"],
    deviceSizes: [320, 640, 860, 1000]
  },
        distDir: 'build'
}
```

We can now run `yarn serve` to set up a backend server in one terminal window, and `yarn build` in another. As soon as the project is built, you will see a bunch of files in the `build` directory.

The `BUILD_ID` file contains a hash of the current build. This hash is the name of the directory inside `build/server/pages` that contains current build artifacts such as page HTML and JSON.

Pages that can be statically generated (such as `Section` and `Front`) all have `.html` files associated with them. In contrast, pages that can only be rendered on the server (`Post`) only have `.js` files.

Deploying Project

In the previous section, we set up a custom build directory. It is usually used for deploying statically generated pages to a server.

With Next though, the easiest way to deploy the application is to deploy on Vercel[210]. It is a platform for SSR and SSG sites made with modern frontend frameworks.

The coolest thing with Vercel is that they provide a way to host API and the frontend in the same project so we won't need a separate server. Also, Vercel is optimized for use with Next.

Let's now update the project a bit and deploy it on Vercel to see how convenient this is.

[210] https://vercel.com/docs

Remaking API

Now our project has a separate server in the `server` folder. With Vercel, we will no longer need it because we can use serverless functions[211]. To create a serverless function we need to use Next's API routes[212].

Serverless functions here are a wrapper on the file system or a 3-rd party server. Conceptually these API routes are very similar to what we did in the `server` directory. The difference is that with API routes we need to change the routing model.

Previously we used Express to define routes and handlers for them. With Next's API routes, we need to create the `api` directory inside `pages` and use directories and files to define routes—just like with ordinary Next pages.

Create an `api` folder inside `pages` and define our "Hello world" controller:

```
import type { NextApiRequest, NextApiResponse } from "next"

export default function hello(
  req: NextApiRequest,
  res: NextApiResponse
) {
  res.status(200).json({ hello: "World" })
}
```

Just like with regular Next pages, we export the function from the module. But this time instead of a page we export an API controller that takes 2 arguments: `request` and `response`.

`request` is an instance of `http.IncomingMessage`[213] with pre-built middlewares[214]. It is conveniently typed with the `NextApiRequest` type.

`response` is a `http.ServerResponse`[215] with Next response helpers[216]. It is typed with the `NextApiResponse` type.

[211] https://vercel.com/docs/serverless-functions/introduction#deploying-serverless-functions
[212] https://nextjs.org/docs/api-routes/introduction
[213] https://nodejs.org/api/http.html#http_class_http_incomingmessage
[214] https://nextjs.org/docs/api-routes/api-middlewares
[215] https://nodejs.org/api/http.html#http_class_http_serverresponse
[216] https://nextjs.org/docs/api-routes/response-helpers

By default, API routes handle GET requests. So, if we now go to http://localhost:3000/api[217] in the browser, we should see the API response:

```
{"hello":"World"}
```

We made our first API controller! Now remove the `pages/api/index.ts` and let's rebuild all our existing APIs using API routes.

Remaking Posts

Let's start with remaking posts APIs. We will have 2 endpoints:

- `/api/posts` for getting the list of all the posts;
- `/api/posts/[id]` for getting a particular post by its ID.

We put the first handler in `pages/api/posts/index.ts` so that Next can understand what route this module is responsible for.

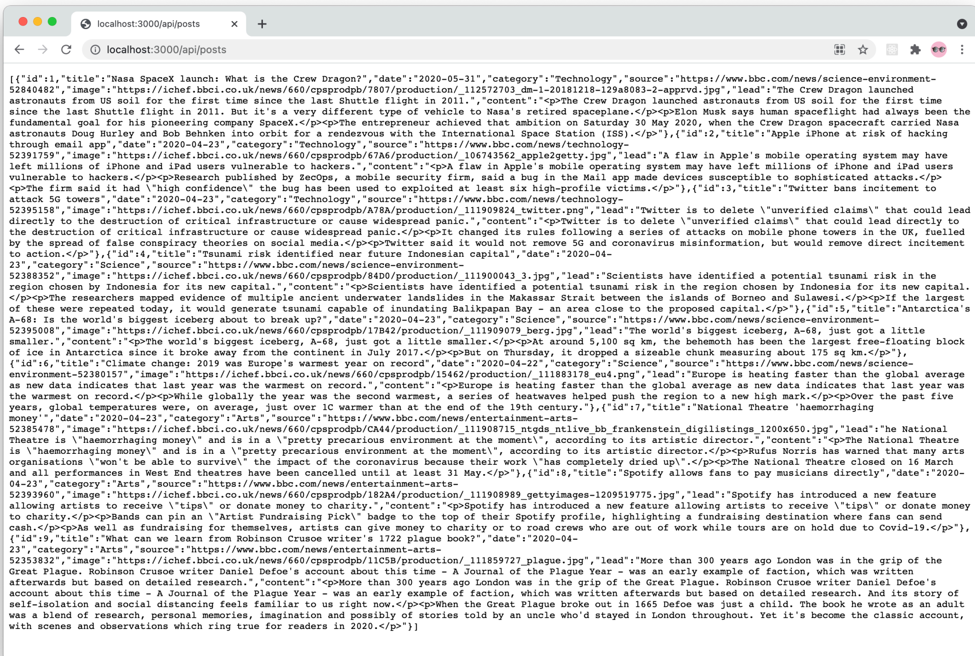
```
import type { NextApiRequest, NextApiResponse } from "next"
import type { Post } from "../../../shared/types"
import postsSource from "../../../server/posts.json"

export default function postsHandler(
  req: NextApiRequest,
  res: NextApiResponse<Post[]>
) {
  const posts = postsSource as Post[]
  return res.status(200).json(posts)
}
```

Inside, we import the posts list from `posts.json` just as we did before. The `postsHandler` controller will respond with this list.

[217] http://localhost:3000/api/

The `NextApiResponse` type is generic, so we can explicitly define what type the response will have. In our case the response is a list of posts, so we use `NextApiResponse<Post[]>` as the response type.

If we now go to http://localhost:3000/api/posts, we get the response with all of the comments we have in the file:




```
[{"id":1,"title":"Nasa SpaceX launch: What is the Crew Dragon?","date":"2020-05-31","category":"Technology","source":"https://www.bbc.com/news/science-environment-
52840482","image":"https://ichef.bbci.co.uk/news/660/cpsprodpb/7807/production/_112572703_dm-1-20181218-129a8083-2-apprvd.jpg","lead":"The Crew Dragon launched
astronauts from US soil for the first time since the last Shuttle flight in 2011.","content":"<p>The Crew Dragon launched astronauts from US soil for the first time
since the last Shuttle flight in 2011. But it's a very different type of vehicle to Nasa's retired spaceplane.</p><p>Elon Musk says human spaceflight had always been the
fundamental goal for his pioneering company SpaceX.</p><p>The entrepreneur achieved that ambition on Saturday 30 May 2020, when the Crew Dragon spacecraft carried Nasa
astronauts Doug Hurley and Bob Behnken into orbit for a rendezvous with the International Space Station (ISS).</p>"},{"id":2,"title":"Apple iPhone at risk of hacking
through email app","date":"2020-04-23","category":"Technology","source":"https://www.bbc.com/news/technology-
52391759","image":"https://ichef.bbci.co.uk/news/660/cpsprodpb/67A6/production/_106743562_apple2getty.jpg","lead":"A flaw in Apple's mobile operating system may have
left millions of iPhone and iPad users vulnerable to hackers.","content":"<p>A flaw in Apple's mobile operating system may have left millions of iPhone and iPad users
vulnerable to hackers.</p><p>Research published by ZecOps, a mobile security firm, said a bug in the Mail app made devices susceptible to sophisticated attacks.</p>
<p>The firm said it had \"high confidence\" the bug has been used to exploited at least six high-profile victims.</p>"},{"id":3,"title":"Twitter bans incitement to
attack 5G towers","date":"2020-04-23","category":"Technology","source":"https://www.bbc.com/news/technology-
52395158","image":"https://ichef.bbci.co.uk/news/660/cpsprodpb/A78A/production/_111909824_twitter.png","lead":"Twitter is to delete \"unverified claims\" that could lead
directly to the destruction of critical infrastructure or cause widespread panic.","content":"<p>Twitter is to delete \"unverified claims\" that could lead directly to
the destruction of critical infrastructure or cause widespread panic.</p><p>It changed its rules following a series of attacks on mobile phone towers in the UK, fuelled
by the spread of false conspiracy theories on social media.</p><p>Twitter said it would not remove 5G and coronavirus misinformation, but would remove direct incitement
to action.</p>"},{"id":4,"title":"Tsunami risk identified near future Indonesian capital","date":"2020-04-
23","category":"Science","source":"https://www.bbc.com/news/science-environment-
52388352","image":"https://ichef.bbci.co.uk/news/660/cpsprodpb/84D0/production/_111900043_3.jpg","lead":"Scientists have identified a potential tsunami risk in the
region chosen by Indonesia for its new capital.","content":"<p>Scientists have identified a potential tsunami risk in the region chosen by Indonesia for its new capital.
</p><p>The researchers mapped evidence of multiple ancient underwater landslides in the Makassar Strait between the islands of Borneo and Sulawesi.</p><p>If the largest
of these were repeated today, it would generate tsunami capable of inundating Balikpapan Bay - an area close to the proposed capital.</p>"},{"id":5,"title":"Antarctica's
A-68: Is the world's biggest iceberg about to break up?","date":"2020-04-23","category":"Science","source":"https://www.bbc.com/news/science-environment-
52395008","image":"https://ichef.bbci.co.uk/news/660/cpsprodpb/17B42/production/_111909079_berg.jpg","lead":"The world's biggest iceberg, A-68, just got a little
smaller.","content":"<p>The world's biggest iceberg, A-68, just got a little smaller.</p><p>At around 5,100 sq km, the behemoth has been the largest free-floating block
of ice in Antarctica since it broke away from the continent in July 2017.</p><p>But on Thursday, it dropped a sizeable chunk measuring about 175 sq km.</p>"},
{"id":6,"title":"Climate change: 2019 was Europe's warmest year on record","date":"2020-04-22","category":"Science","source":"https://www.bbc.com/news/science-
environment-52380157","image":"https://ichef.bbci.co.uk/news/660/cpsprodpb/15462/production/_111883178_eu4.png","lead":"Europe is heating faster than the global average
as new data indicates that last year was the warmest on record.","content":"<p>Europe is heating faster than the global average as new data indicates that last year was
the warmest on record.</p><p>While globally the year was the second warmest, a series of heatwaves helped push the region to a new high mark.</p><p>Over the past five
years, global temperatures were, on average, just over 1C warmer than at the end of the 19th century."},{"id":7,"title":"National Theatre 'haemorrhaging
money'","date":"2020-04-23","category":"Arts","source":"https://www.bbc.com/news/entertainment-arts-
52385478","image":"https://ichef.bbci.co.uk/news/660/cpsprodpb/CA44/production/_111908715_ntgds_ntlive_bb_frankenstein_digilistings_1200x650.jpg","lead":"he National
Theatre is \"haemorrhaging money\" and is in a \"pretty precarious environment at the moment\", according to its artistic director.","content":"<p>The National Theatre
is \"haemorrhaging money\" and is in a \"pretty precarious environment at the moment\", according to its artistic director.</p><p>Rufus Norris has warned that many arts
organisations \"won't be able to survive\" the impact of the coronavirus because their work \"has completely dried up\".</p><p>The National Theatre closed on 16 March
and all performances in West End theatres have been cancelled until at least 31 May.</p>"},{"id":8,"title":"Spotify allows fans to pay musicians directly","date":"2020-
04-23","category":"Arts","source":"https://www.bbc.com/news/entertainment-arts-
52393960","image":"https://ichef.bbci.co.uk/news/660/cpsprodpb/182A4/production/_111908989_gettyimages-1209519775.jpg","lead":"Spotify has introduced a new feature
allowing artists to receive \"tips\" or donate money to charity.","content":"<p>Spotify has introduced a new feature allowing artists to receive \"tips\" or donate money
to charity.</p><p>Bands can pin an \"Artist Fundraising Pick\" badge to the top of their Spotify profile, highlighting a fundraising destination where fans can send
cash.</p><p>As well as fundraising for themselves, artists can give money to charity or to road crews who are out of work while tours are on hold due to Covid-19.</p>"},
{"id":9,"title":"What can we learn from Robinson Crusoe writer's 1722 plague book?","date":"2020-04-
23","category":"Arts","source":"https://www.bbc.com/news/entertainment-arts-
52353832","image":"https://ichef.bbci.co.uk/news/660/cpsprodpb/11C5B/production/_111859727_plague.jpg","lead":"More than 300 years ago London was in the grip of the
Great Plague. Robinson Crusoe writer Daniel Defoe's account about this time - A Journal of the Plague Year - was an early example of faction, which was written
afterwards but based on detailed research.","content":"<p>More than 300 years ago London was in the grip of the Great Plague. Robinson Crusoe writer Daniel Defoe's
account about this time - A Journal of the Plague Year - was an early example of faction, which was written afterwards but based on detailed research. And its story of
self-isolation and social distancing feels familiar to us right now.</p><p>When the Great Plague broke out in 1665 Defoe was just a child. The book he wrote as an adult
was a blend of research, personal memories, imagination and possibly of stories told by an uncle who'd stayed in London throughout. Yet it's become the classic account,
with scenes and observations which ring true for readers in 2020.</p>"}]
```


API response with a list of all the posts

Okay, getting the list of posts was pretty straightforward. But what if we need to define a dynamic route? The second controller that takes a particular post is exactly this case.

To respond with a particular post, we need to know what ID is requested. For this, Next offers the same model as with pages—we can use dynamic API routes[218].

The pattern for a dynamic route is the same as for dynamic pages. We use square brackets to define a dynamic route and the argument name for it.

[218] https://nextjs.org/docs/api-routes/dynamic-api-routes

For our post controller, we can create file `pages/api/posts/[id].ts`. It will mean that whenever we request the `/api/post/42` endpoint the controller will handle the request and the request will have `id` parameter with value a equal to `42`.

```
import type { NextApiRequest, NextApiResponse } from "next"
import type { Post } from "../../../shared/types"
import postsSource from "../../../server/posts.json"

export default function postHandler(
  req: NextApiRequest,
  res: NextApiResponse<Post>
) {
  const posts = postsSource as Post[]
  const wantedId = String(req.query.id)
  const post = posts.find(({ id }: Post) => String(id) === wantedId)
  return res.status(200).json(post)
}
```

The functionality of the controller is the same as we had earlier with Express controllers. We take the `id` from the request, find the post with the same ID in the data, and return it.

Now if we request http://localhost:3000/api/posts/4 we will get the data for this post.

Remaking Categories

For categories we will also have `2` endpoints:

- `/api/categories/` for getting the list of categories;
- `/api/categories/[id]` for getting posts for the category.

The first one will import the data from a file and return it as the response:

```
import type { NextApiRequest, NextApiResponse } from "next"
import type { Category } from "../../../shared/types"
import categoriesSource from "../../../server/categories.json"

export default function categoriesHandler(
  req: NextApiRequest,
  res: NextApiResponse<Category[]>
) {
  const categories = categoriesSource as Category[]
  return res.status(200).json(categories)
}
```

We also will type the response with `NextApiResponse<Category[]>` so that the returned data would be typed with the `Category[]` type.

The second one will take the category ID, filter posts with this category, and return the list of these posts:

```
import type { NextApiRequest, NextApiResponse } from "next"
import type { Post } from "../../../shared/types"
import postsSource from "../../../server/posts.json"

export default function categoryHandler(
  req: NextApiRequest,
  res: NextApiResponse<Post[]>
) {
  const posts = postsSource as Post[]
  const found = posts.filter(
    ({ category: id }: Post) => id === req.query.id
  )
  const categoryPosts = [...found, ...found, ...found]
  return res.status(200).json(categoryPosts)
}
```

The response for this controller we will type as `NextApiResponse<Post[]>`.

Remaking Comments

The only APIs left are the comments. We will need only 1 endpoint for working with comments but it will handle 2 different methods: GET and POST.

- GET `/api/comments/[postID]` for getting the comments for a given post;
- POST `/api/comments/[postID]` for submitting a comment for a given post.

Let's start with getting comments for a post. We will write a `commentsForPost` function. This function will filter the comments for a given post and return them as the result:

```
import path from "path"
import { writeFile } from "fs/promises"

import type { NextApiRequest, NextApiResponse } from "next"
import type { Comment, EntityId } from "../../../shared/types"
import commentsSource from "../../../server/comments.json"

const comments = commentsSource as Comment[]

function commentsForPost(postId: EntityId) {
  return comments.filter(({ post }) => post === postId)
}
```

To handle the POST request we will have to check the request method. Let's try doing this using `switch`. We will check the `req.method` and handle differently each case:

```
export default function commentsHandler(
  req: NextApiRequest,
  res: NextApiResponse
) {
  const postId = Number(req.query.id)

  switch (req.method) {
    case "GET": {
      return res.status(200).json(commentsForPost(postId))
    }
    case "POST": {
      comments.push({
        id: comments.length + 1,
        author: req.body.name,
        content: req.body.comment,
        post: postId,
        time: "Less than a minute ago"
      })

      writeFile(
        path.resolve(process.cwd(), "server/comments.json"),
        JSON.stringify(comments)
      )

      return res.json(commentsForPost(postId))
    }
    default:
      return res.status(404)
  }
}
```

We move the previous functionality into the GET case. In the POST case, we create a new comment from the request data and push it to the list of comments. Optionally, we can update the JSON file. As a result, return the updated list of comments for the given post.

In the `default` case we return the 404 status if the request method doesn't match expected. We could return any other status, for example, 403 to say that this method is forbidden.

Creating Client Requests

Now, when the server APIs are ready, we can create the module for client requests. We need a couple of functions to call our API routes from the client. These functions we will need in pages' code to get data for pre-render pages.

Let's start with a `request` function which will send GET requests to the API and fetch data. Create a `request/index.ts` file and add the following code:

```
import { UriString, EntityId, Person, Post } from "../shared/types"
import { config } from "./config"

const { baseUrl } = config

async function request<TResponse>(url: UriString) {
  const response = await fetch(`${baseUrl}/${url}`)
  const data = (await response.json()) as TResponse
  return data
}
```

The `request` function will take an endpoint URL and send a request to that URL. It will also parse the result as JSON and return the response data typed with a `TResponse` generic argument.

The `baseUrl` is the root URL for our API routes which we will create in a moment. Create a new file `request/config.ts` and add the code:

```
const IS_PRODUCTION = process.env.NODE_ENV === "production"

const protocol = IS_PRODUCTION ? "https" : "http"
const host = process.env.NEXT_PUBLIC_VERCEL_URL || "localhost:3000"

export const config = {
  baseUrl: `${protocol}://${host}/api`
}
```

It is considered a good practice to separate the configs from the code. At the end of this chapter, we will also see why it is useful.

The second function we're going to need is a `post` function. It will take the data and send POST requests to the API. Let's add this function in the index file:

```
async function post<TPayload>(url: UriString, data: TPayload) {
  return fetch(`${baseUrl}/${url}`, {
    method: "POST",
    headers: { "Content-Type": "application/json;charset=utf-8" },
    body: JSON.stringify(data)
  })
}
```

Create the wrappers for the requests that our app will use:

- list of posts
- a particular post;
- categories list
- list of posts for a category;
- comments for a post
- submitting a comment.

```
export const fetchPosts = () => request("posts")
export const fetchPost = (id: EntityId) => request(`posts/${id}`)

export const fetchCategories = () => request("categories")
export const fetchCategory = (categoryId: EntityId) =>
  request<Post[]>(`categories/${categoryId}`)

export const fetchComments = (postId: EntityId) =>
  request(`comments/${postId}`)

export const submitComment = (
  postId: EntityId,
  name: Person,
  comment: string
) => post(`comments/${postId}`, { name, comment })
```

Now we can use these functions in pages to fetch the data for pre-rendering.

Updating Pages

When the client requests are ready we can replace the old request functions with them. This will allow us to remove the old API, use API routes, and deploy the project to Vercel using serverless functions.

Let's review what pages use any APIs and what we need to replace:

- main page fetches categories and posts;
- post page fetches the post and its comments;
- post page also can submit a new comment for the post;
- category page loads the posts for the category.

We will have to replace all the imports to the request functions on these pages. Also, it is important that serverless functions don't support static export[219] so we will also need to replace `getStaticProps` with `getServerSideProps`.

[219] https://nextjs.org/docs/api-routes/introduction#caveats

Updating Main Page

On the main page we need to replace the old imports from `api/summary` with the new one:

```
import { fetchPosts, fetchCategories } from "../request"
```

And use `getServerSideProps` instead of old `getStaticProps`:

```
export async function getServerSideProps() {
  const categories = await fetchCategories()
  const posts = await fetchPosts()
  return { props: { posts, categories } }
}
```

Updating Category Page

On the category page, we also replace the old import from `api/category` with new request import and define `getServerSideProps`:

```
import type { GetServerSideProps } from "next"
import { useRouter } from "next/router"
import { Post } from "../../shared/types"
import { fetchCategory } from "../../request"
import { Section } from "../../components/Section"
import { Loader } from "../../components/Loader"

type CategoryProps = {
  posts: Post[]
}

export const getServerSideProps: GetServerSideProps<CategoryProps> =
  async ({ params }) => {
    if (typeof params.id !== "string")
```

```
      throw new Error("Unexpected id")
    const posts = await fetchCategory(params.id)
    return { props: { posts } }
  }

const Category = ({ posts }: CategoryProps) => {
  const router = useRouter()

  if (router.isFallback) return <Loader />
  return <Section posts={posts} title={String(router.query.id)} />
}

export default Category
```

The rest of the code stays the same because we kept the request functions' signature and updated them only under the hood.

Updating Post Page

On the post page, we replace the imports from `api/post` and `api/comments` with new request imports:

```
import React from "react"
import { NextPage } from "next"
import { useSelector } from "react-redux"
import { Loader } from "../../components/Loader"
import { PostBody } from "../../components/Post/PostBody"
import { Comments } from "../../components/Comments"

import { fetchPost, fetchComments } from "../../request"
import { State, store } from "../../store"
import { PostState, UPDATE_POST_ACTION } from "../../store/post"
import {
  CommentsState,
  UPDATE_COMMENTS_ACTION
```

```
} from "../../store/comments"

export const getServerSideProps = store.getServerSideProps(
  (store) =>
    async ({ params }) => {
      if (typeof params.id !== "string")
        throw new Error("Unexpected id")

      const comments = await fetchComments(params.id)
      const post = await fetchPost(params.id)

      store.dispatch({ type: UPDATE_POST_ACTION, post })
      store.dispatch({ type: UPDATE_COMMENTS_ACTION, comments })

      return null
    }
)

const Post: NextPage = () => {
  const post = useSelector<State, PostState>(({ post }) => post)
  const comments = useSelector<State, CommentsState>(
    ({ comments }) => comments
  )

  if (!post) return <Loader />
  return (
    <>
      <PostBody post={post} />
      <Comments comments={comments} post={post.id} />
    </>
  )
}

export default Post
```

And that's it! Sine this page already uses `getServerSideProps` because of the Redux store the rest of the code in this file stays the same.

However, we also need to update the comment form to submit comments to the correct endpoint.

In the `CommentForm` component, we need to replace imports from `api/comments` with the new one:

```
import { submitComment } from "../../request"
```

The rest of the code also stays the same. Now if we run the project and open it in the browser it should work as before.

Cleaning Up

When we migrated to the API routes, we can safely delete obsolete code. For example, we don't need `api` directory, `shared/staticPaths.ts`, `server/index.ts` anymore.

Also, we don't need server packages, so we can safely remove:

- `body-parser`
- `concurrently`
- `cors`
- `express`
- `node-fetch`
- `ts-node`

Clean up the `scripts` section in the `package.json`:

```
"scripts": {
  "build": "next build",
  "start": "next start",
  "dev": "next"
},
```

Now, let's deploy the project!

Deployment with Serverless Functions

Now the project is almost ready to be deployed on Vercel. We're going to have to push it to a repository on GitHub or GitLab, define some environment variables for API configuration, and we're ready to go!

Let's start with checking the `next.config.js`. This config should contain the `images` field that sets up how Next will handle our images. And the `deviceSizes` property tells Next what breakpoints we're going to consider in the app layout.

```
module.exports = {
  images: {
    domains: ["ichef.bbci.co.uk"],
    deviceSizes: [320, 640, 860, 1000]
  },
        distDir: 'build'
}
```

Then, we need to update our `request/config.ts` file. Right now it contains only a declaration for `localhost`. When deployed, the app won't be able to call API by that URL. We need to inject the real API URL in the config. For this, we're going to use environment variables[220].

By default, Next and Vercel don't expose env variables to the client code for security reasons. But we can explicitly tell them to inject a variable into the client code using the `NEXT_PUBLIC_` prefix.

There is also a whole bunch of environment variables that Vercel exposes automatically[221] for us. Among those variables is `VERCEL_URL` (or `NEXT_PUBLIC_VERCEL_URL` for the client) that contains the real deployment URL.

We will use this variable to set up configs for our client requests module. Let's update the configs:

[220] https://vercel.com/docs/projects/environment-variables
[221] https://vercel.com/docs/projects/environment-variables#system-environment-variables

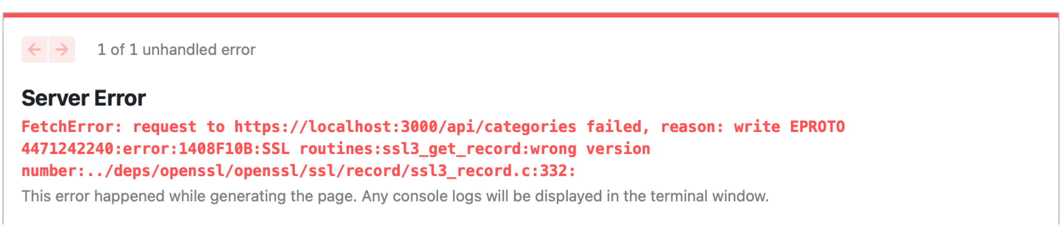
```
const IS_PRODUCTION = process.env.NODE_ENV === "production"

const protocol = IS_PRODUCTION ? "https" : "http"
const host = process.env.NEXT_PUBLIC_VERCEL_URL || "localhost:3000"

export const config = {
  baseUrl: `${protocol}://${host}/api`
}
```

Okay, now when deployed, the app will send requests to the real app URL and reach the API endpoints. But this code won't run locally because there is no `NEXT_PUBLIC_VERCEL_URL`.

To solve this we can declare this variable by using `.env.local` file. Let's create this file in the project root and add the variable:

```
NEXT_PUBLIC_VERCEL_URL = "localhost:3000"
```

We're going to need to restart the Next dev server to see any changes. If we run the project again we'll see an error:



Certificate error on localhost

That's because we didn't set up HTTPS on localhost. Let's replace `https` with `http` on the local development server in the configs:

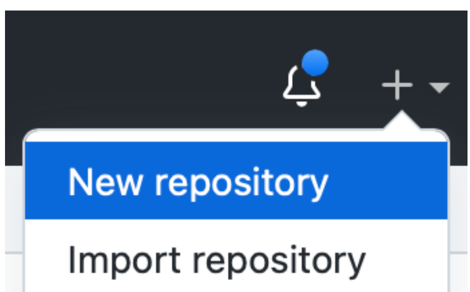
```
const IS_PRODUCTION = process.env.NODE_ENV === "production"

const protocol = IS_PRODUCTION ? "https" : "http"
const host = process.env.NEXT_PUBLIC_VERCEL_URL || "localhost:3000"

export const config = {
  baseUrl: `${protocol}://${host}/api`
}
```

Here, we check if we're in production. If so, we use `https` and env variables exposed from the deployment platform. If not, we use `http` and variables from the `.env.local` file loaded by Next for development.

Now we can push this project to a repo on GitHub or GitLab and deploy it on Vercel.

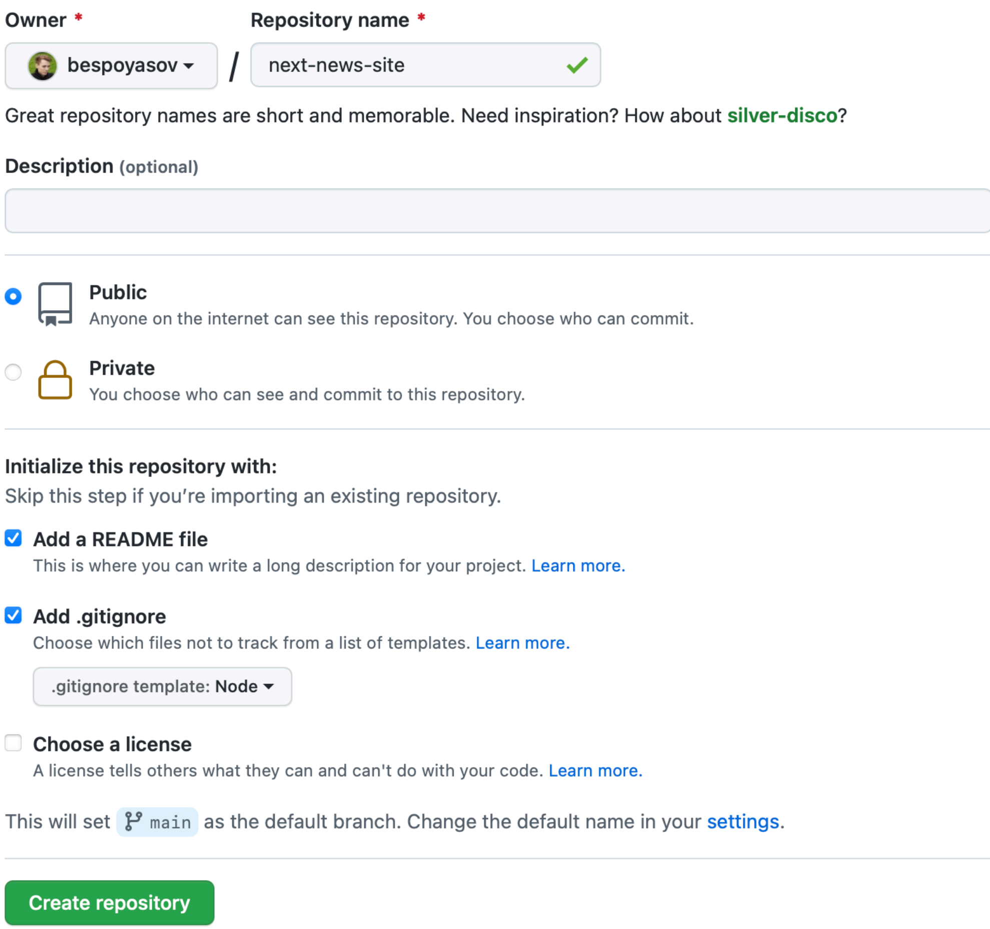
Pushing to GitHub

As an example of a remote repository, we will use GitHub. Sign up or log in to your account on GitHub and create a new repository for this project.



Create a new repo on GitHub

You will be navigated to the "New Repo" page. Add a name and a description for this project (e.g. "next-new-site"). You can keep the repo public or make it private, it won't affect the deployment. Optionally, you can add a Readme file or a `.gitignore` file.



Repo details

When everything is set up hit the "Create Repo" button.

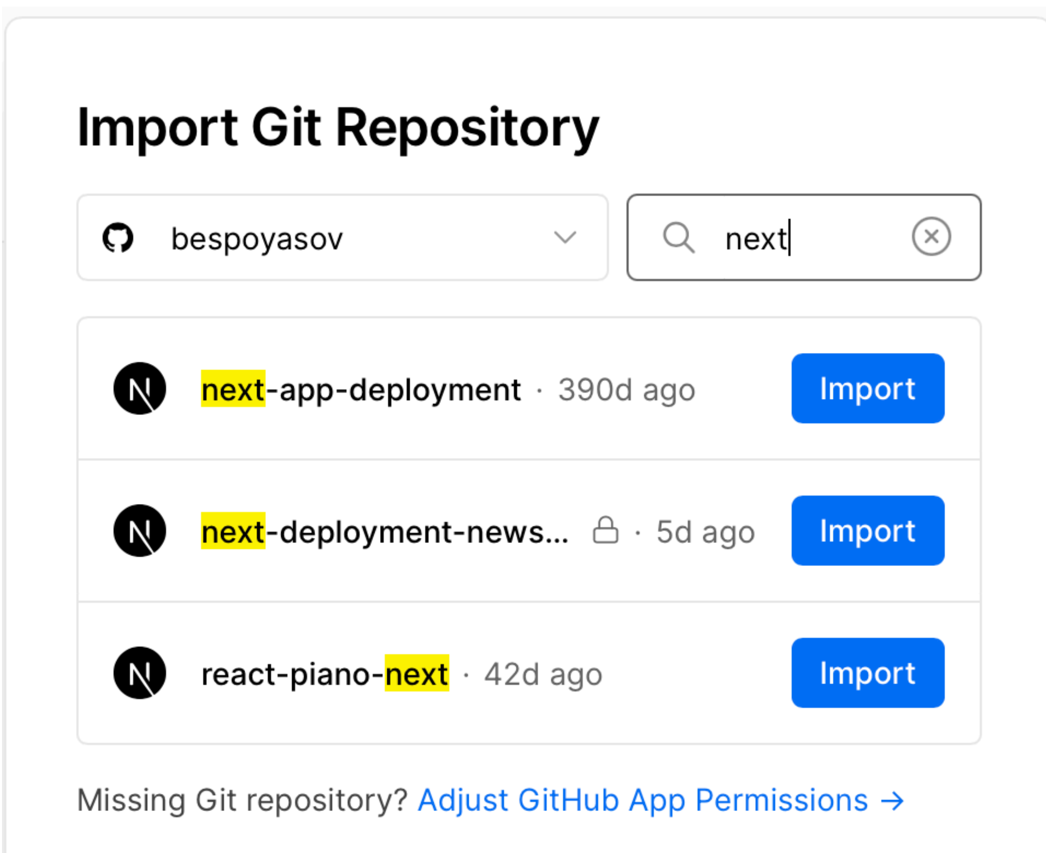
After the repository is created, push the source code to it. Be careful and don't forget to add `.env.local` to the `.gitignore` to avoid leaking variables to the repo!

Deploying to Vercel

When the source code is pushed to the repo, go to vercel.com[222] and create an account there. If you already have an account login.

Once you've logged in, you will be navigated to the Dashboard. Here you will see a "New Project" button. Click on it.

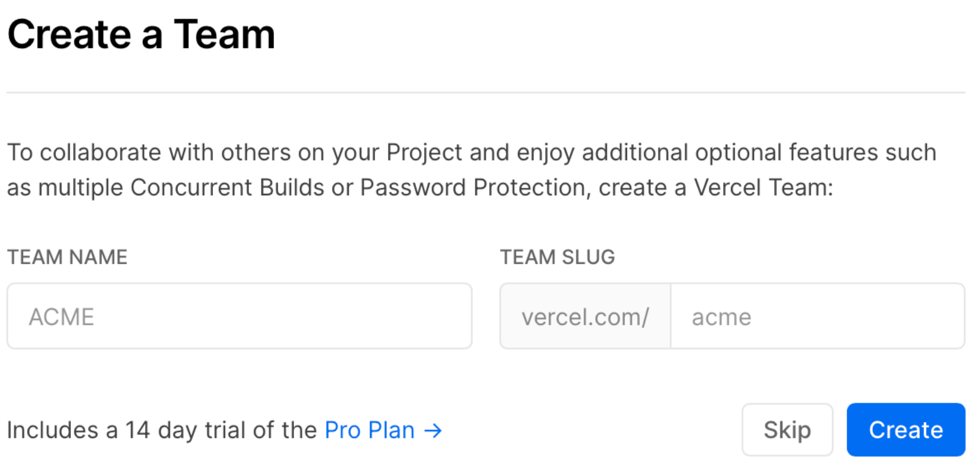
Below you will see an "Import Git Repository" block. There you can select the remote repo from which to build and deploy the project. Find your created repo with this project and click "Import".



Import project from GitHub

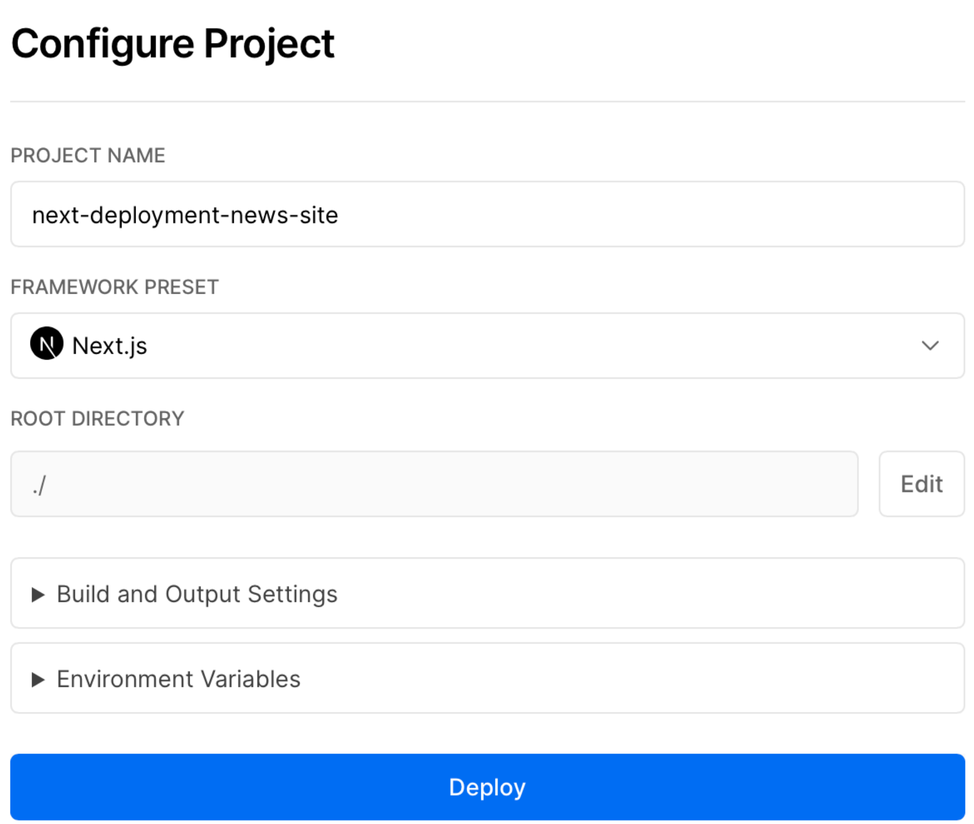
[222] https://vercel.com/

On the "New Project" page you might be asked about creating a team. You can skip this part.



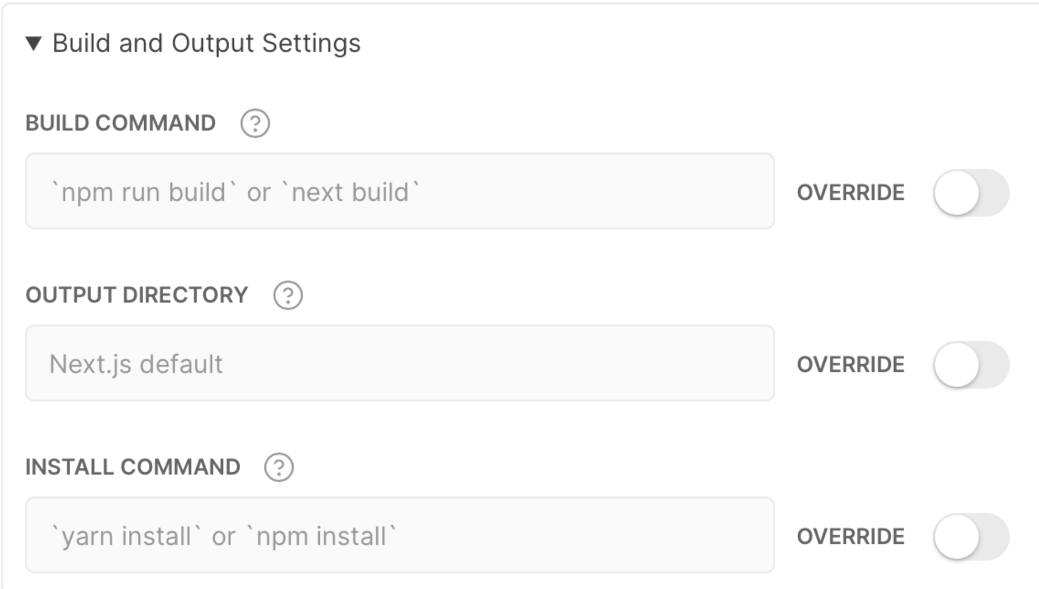
Skip creating a team

In the "Configure Project" section, make sure that the selected "Framework Preset" is "Next.js".



Configure project

You can also set up different build command and output settings if you need. This is done in the “Build and Output Settings” section:



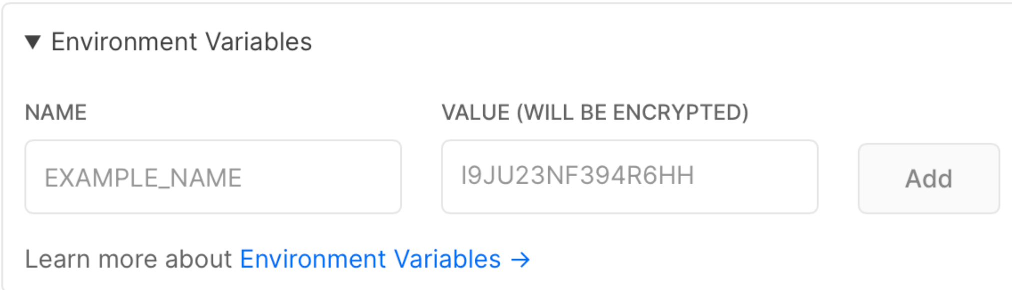
Build settings

By default, they use Next presets and are configured to run the application as it suggested by Next. In the majority of cases, we won't need to change them.

They are useful when, for example, the app should be built using multiple steps or different commands than `npm run build`. Also, we can set up the output directory which is useful for complex deployment systems.

In our case, we can safely keep them default.

You also can set up environment variables in the section below:



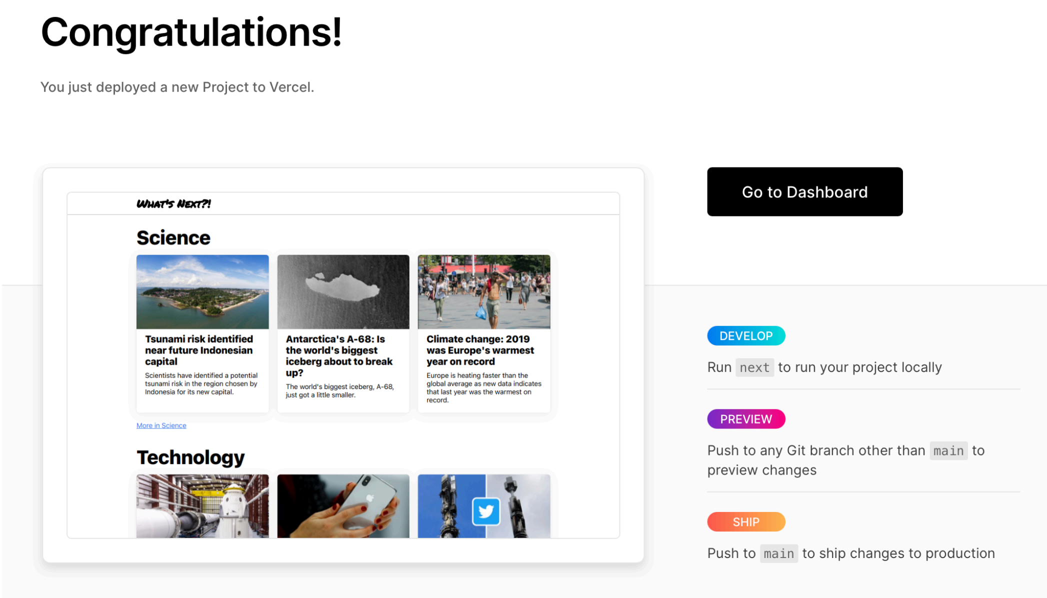
Environment variables

In our case, we use only the system environment variables[223] which are exposed automatically. So we don't need to specify anything else.

When the project is configured, hit the "Deploy" button.

After a few minutes, Vercel will tell you that the project is successfully deployed and will give you a link to the deployment.

[223] https://vercel.com/docs/projects/environment-variables#system-environment-variables



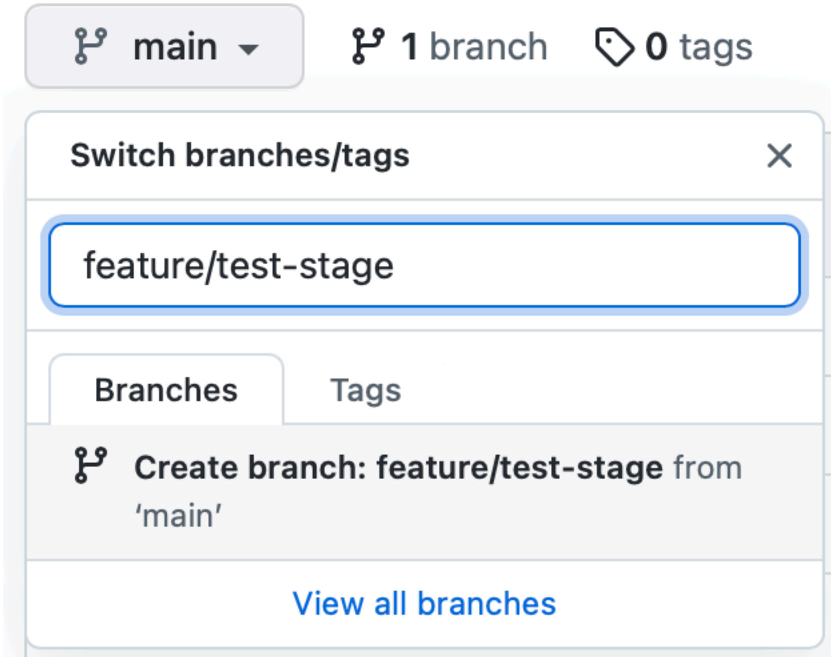
Deployed project

Now, on the dashboard, you can see the freshly deployed project. Click on it and hit the "View Deployment" button. You will be navigated to the production deployment of your project.

Deploying Stages

Sometimes you need to create a non-production deployment, for testing, presenting changes. With Vercel, it is also possible to deploy tests, stages, and pre-production environments.

To deploy a non-production environment we need to create a new branch in the repo from which the project is deployed:



Create a new branch in the project's repo

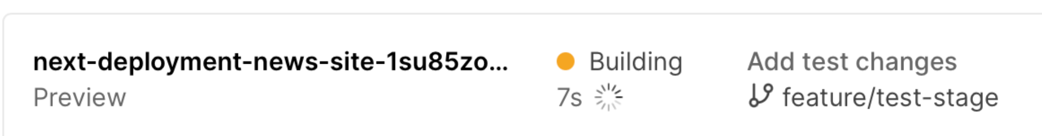
Check out to this branch and add some changes. For example, add the following text on the main page:

```
import React from "react"
import Head from "next/head"
import { Post, Category } from "../shared/types"
import { Feed } from "../components/Feed"
import { fetchPosts, fetchCategories } from "../request"

type FrontProps = {
  posts: Post[]
  categories: Category[]
}

export async function getServerSideProps() {
  const categories = await fetchCategories()
```

```
  const posts = await fetchPosts()
  return { props: { posts, categories } }
}

export default function Front({ posts, categories }: FrontProps) {
  return (
    <>
      <Head>
        <title>Front page of the Internet</title>
      </Head>

      <main>
        <Feed posts={posts} categories={categories} />
      </main>
    </>
  )
}
```

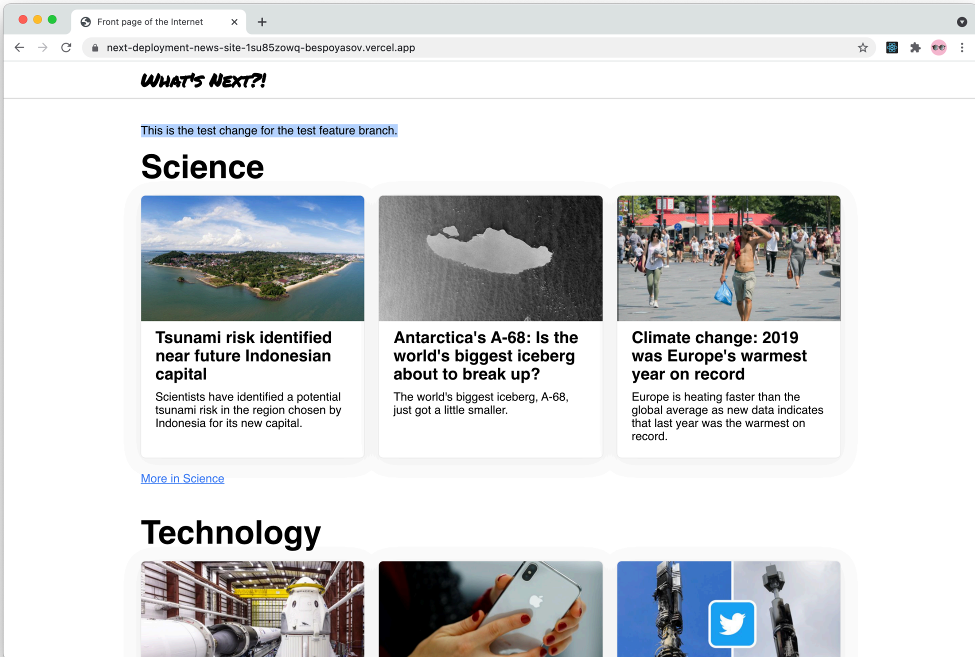
Commit and push the changes to the remote repo.

In the “Deployments” section of the Vercel dashboard, you will see a new deployment for this branch.



Stage deployment

When it’s built and deployed you will see a link to this stage. This link is unique for the commit you just made. Click on it and you will be navigated to the stage with the changes:



Working stage

Summary

In this chapter, we have learned how to create applications using the Next.js framework and use static site generation to pre-render pages.

We connected the app to the Redux store and learned how to optimize images using built in Next components.

Finally we deployed the application on Vercel using API routes and serverless functions.

GraphQL, React, and TypeScript

Introduction

In this chapter, we'll learn how to use GraphQL with TypeScript.

GraphQL is a query language that allows you to exactly specify which fields of data you want to get from the backend.

Let's say you work with a Pokemon API and you want to fetch information about a pokemon.

You would send a query containing the fields you are interested in:

```
query {
  pokemon(name: "Pikachu") {
    id
    number
    name
  }
}
```

The response would contain an object with data for the fields you have requested:

```
{
  "data": {
    "pokemon": {
      "id": "UG9rZW1vbjowMjU=",
      "number": "025",
      "name": "Pikachu"
    }
  }
}
```

To use GraphQL, you need to support it both in the backend and the frontend of your application.

For the frontend, there's a bunch of libraries available, and all of them have React bindings:

- Relay[224] is a library by Facebook released alongside GraphQL. It has a steep learning curve, and you may need some time to learn it.
- Apollo[225] is a platform that provides client libraries for all popular web frameworks and mobile platforms. It is popular and has an easy-to-learn API. We will use it in this chapter.
- URQL[226] is a GraphQL library by Formidable Labs that also has a nice and easy-to-learn API.

All these libraries provide wrappers to make GraphQL requests. You can also perform GraphQL requests manually: after all, GraphQL is based on HTTP.

For example, try to run this `cURL` script in the terminal:

[224] https://relay.dev/
[225] https://www.apollographql.com/
[226] https://formidable.com/open-source/urql/

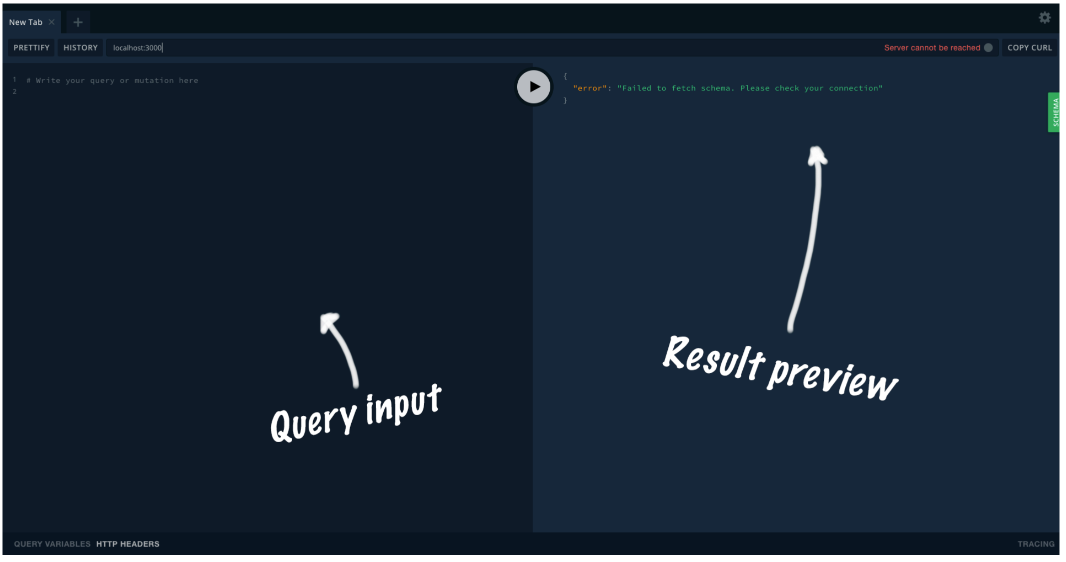
```
curl 'https://graphql-pokemon2.vercel.app/?' \
  -H 'content-type: application/json' \
  --request POST \
  --data '{"query":"query { pokemon(name: \"Pikachu\") { id number name\
 } }","variables":null}' \
```

The server will respond with a JSON formatted object:

```
{"data":{"pokemon":{"id":"UG9rZW1vbjowMjU=","number":"025","name":"Pika\
chu"}}}
```

Most GraphQL server implementations also provide a schema explorer.

For example, when you launch the Apollo GraphQL server, you'll have the `__graphql` endpoint with the following interface:



Apollo GraphQL schema explorer

Here you can enter a query on the left, press the *Execute* button, and see the result in the right pane.

This feature allows you to easily explore any provided GraphQL schema.

If you want to play with the Pokemon API, you can do it here[227].

Is GraphQL better than REST?

REST (REpresentational State Transfer) is an architectural style that defines a set of conventions and constraints that allow you to write an organized and manageable API.

REST was described by Roy Fielding, a computer scientist who presented the principles of REST in his Ph.D. dissertation[228] in 2000.

Here are the key characteristics of a REST API:

- Client-server architecture[229]. User interface concerns should be separated from data storage concerns to improve user interface portability across multiple platforms.
- Statelessness[230]. A stateless server does not persist any information about API users.
- Cacheability[231]. REST API responses must define themselves as cacheable or non-cacheable to prevent clients from providing inappropriate data in future requests.
- Layered system[232]. If a proxy or load balancer sits between the client and the server, connections between them shouldn't be affected and the client shouldn't know whether or not it's connected to the end server.
- Uniform interface[233]. There should be a way of interacting with a given server that is uniform across application types (such as a website or a mobile app). The main guideline is that each individual resource has to be identified on requests.

When you create a REST API, you define HTTP endpoints for each of your resources. For example, if you want to allow to create, read, update and delete users in your application, you would create the following endpoints:

[227]https://graphql-pokemon2.vercel.app/?
[228]https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
[229]https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_1_2
[230]https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_1_3
[231]https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_1_4
[232]https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_1_6
[233]https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_1_5

```
GET http://api.example/users  // Get all users
POST http://api.example/users  // Create a new user
GET http://api.example/users/:id  // Get user by ID
PUT http://api.example/users/:id  // Update user by ID
DELETE http://api.example/users/:id  // Delete user by ID
```

If users in your application have repositories, then you would have to create a set of endpoints to work with them as well:

```
GET http://api.example/users/:id/repositories
// Get repositories for a given user ID
```

It also means that when you need to fetch both users and their repositories, you have two options:

- Create another endpoint that would return users and their repositories.
- Make two successive calls to the API to first fetch the users and then their repositories.

As you can see, this approach creates overhead, and you have to write more code to extend your API.

This is why in 2015 Facebook started developing GraphQL[234].

GraphQL allows the client to specify what data it needs to get from the server.

When you use GraphQL, you need to:

- Define the complete schema on the backend.
- Implement special functions called *resolvers* that will fill the schema with data.

This approach allows you to make fewer assumptions about the client's needs. You don't have to define additional endpoints when your client needs more data.

It also fixes the problem of over-fetching. Your client can specify if it needs additional data right in a query.

[234]https://engineering.fb.com/core-data/graphql-a-data-query-language/

In general, GraphQL requires less work to define a decent API, and it is easier to maintain.

Many services currently provide GraphQL versions of their APIs, including:

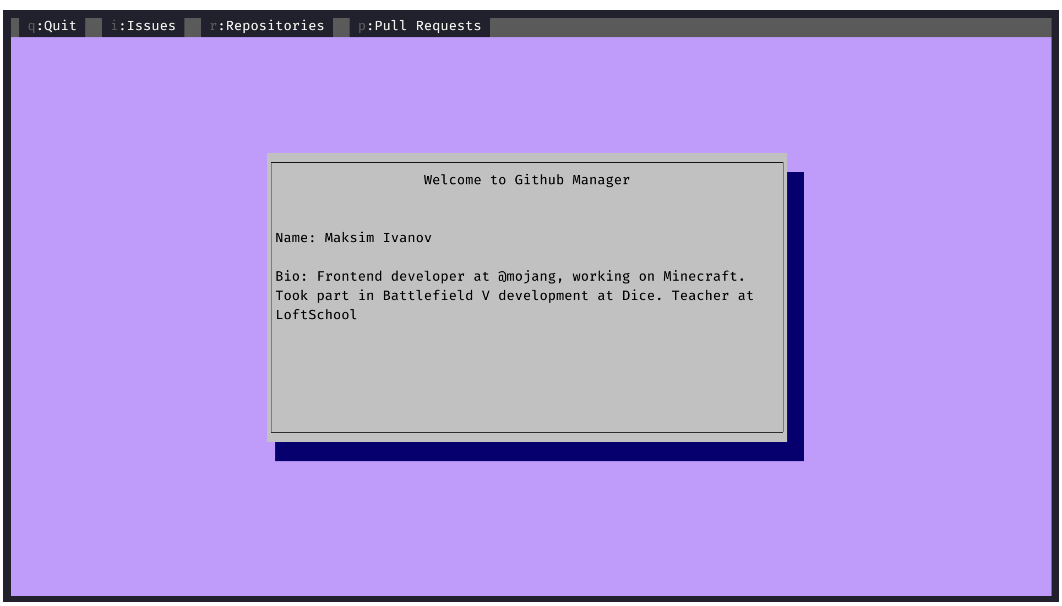
- Facebook
- Instagram
- GitHub

What are we building?

In this chapter, we'll create a GitHub GraphQL client that will run in the terminal. It will allow the user to see the list of their repositories, issues, and pull requests.

The application will have a graphical UI made using the *curses* library.

On the main screen, we'll display information about the currently logged-in user:

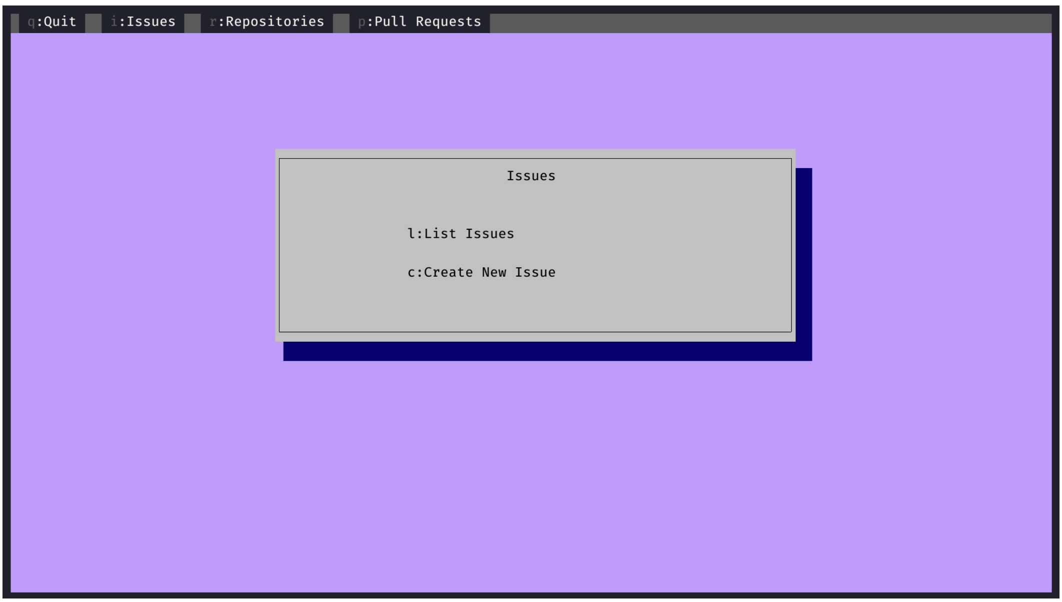


Main screen

We'll have a navigation bar at the top with the list of resources that you can perform operations with:

- Repositories
- Issues
- Pull Requests

You'll be able to switch between screens by pressing keys on the keyboard. For example, you'll be able to open the *Issues* tab by pressing `i`:

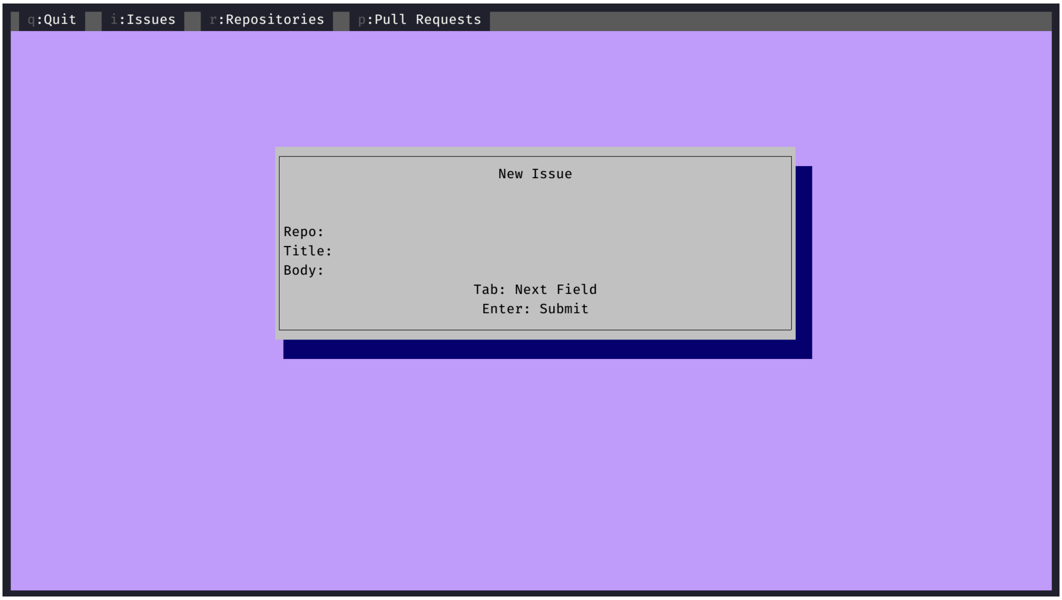


Issues screen

You will be presented with a window giving you two options:

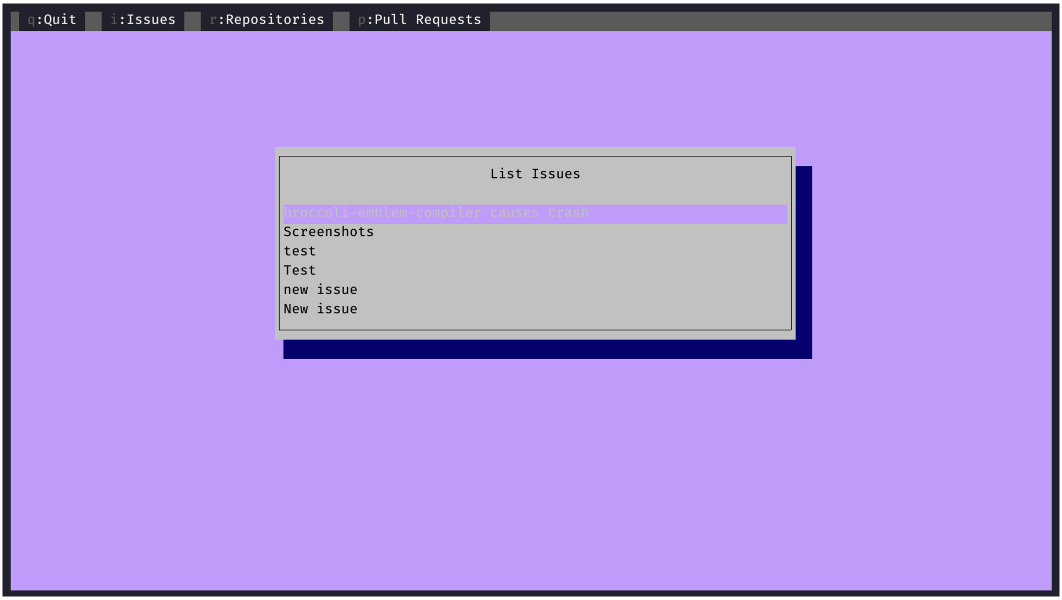
- Press `c` to create a new issue.
- Press `l` to see the list of existing issues.

If you press `c`, the application will open a form to enter a title and description for a new issue. Every issue belongs to a specific repository, so you'll also have to specify a repository name:



Create Issue screen

If you press `l`, you will see the list of available issues. You'll be able to select an issue using the mouse, arrow keys, or Vim-style using `j` and `k` keys. After selecting an issue, you'll be able to press `Enter` or click on it to open the browser and navigate to the selected issue:

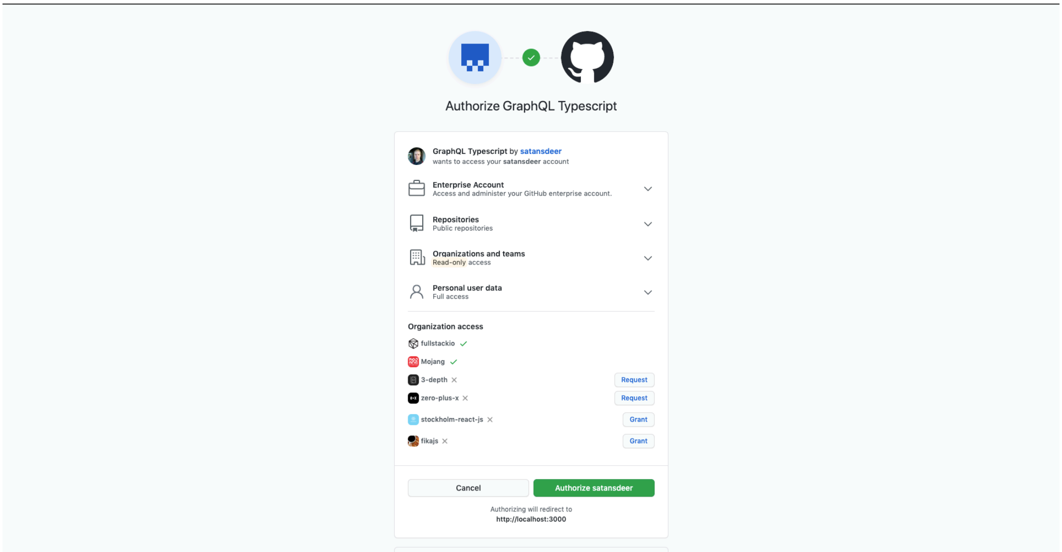


List Issues screen

You'll be able to manage pull requests and repositories in a similar manner.

GitHub requires authentication to make API calls. In our application, we'll be using the OAuth 2 authentication flow.

When you launch the application for the first time, it will open the browser and display the GitHub authentication page:



GitHub authentication screen

After you authenticate, the application will store an authentication token and won't require you to repeat this process unless you remove the token from key storage.

The key storage is specific to the operating system you use:

- Keychain on Mac.
- Credential Vault on Windows.
- Secret Service API/libsecret on Linux.

Authenticate in GitHub and Preview The Final Result

Authenticating in GitHub

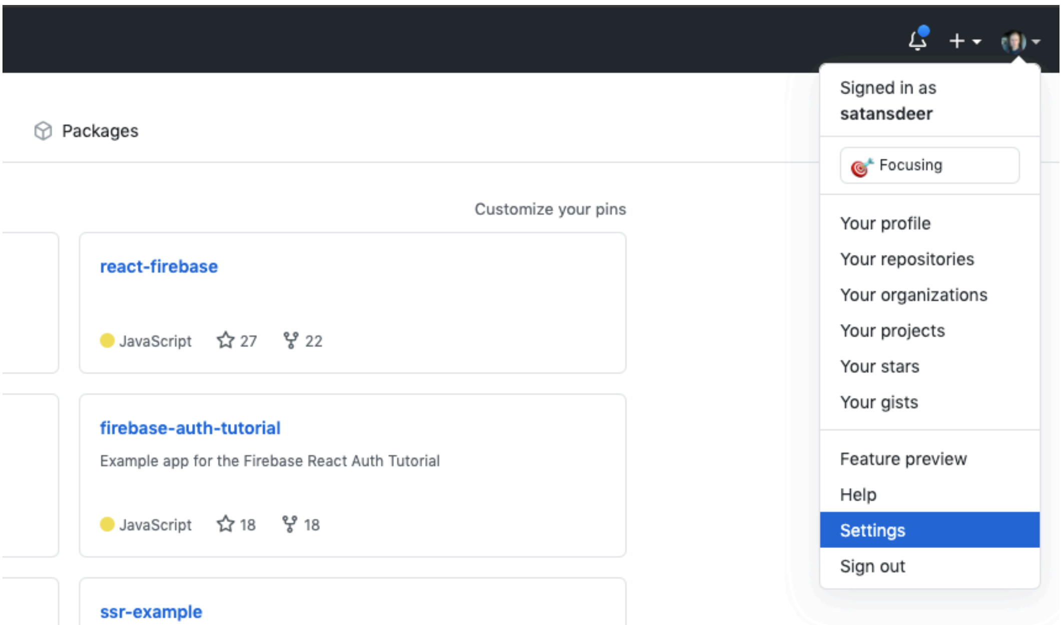
The first thing we need to do to be able to use the GitHub API is authenticate.

To communicate with the GraphQL server we'll need an OAuth token with the right scopes. We will follow the web application flow[235].

[235] https://docs.github.com/en/developers/apps/authorizing-oauth-apps#web-application-flow

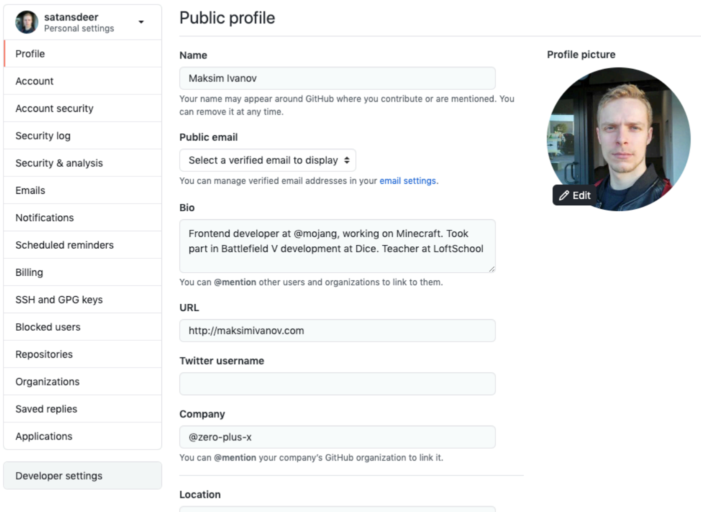
To enable the web authentication flow in our application, we need to get the `client_-id` and `client_secret`. To do this go to your GitHub profile and generate a new key.

Click on your avatar in the top right corner, and then click the *Settings* link:



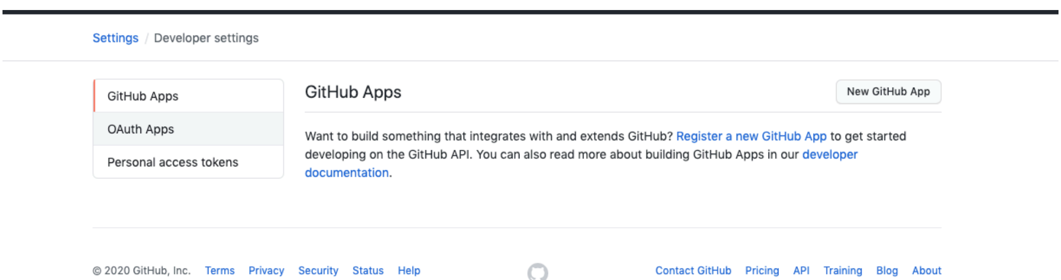
Profile dropdown

From the *Settings* page, go to *Developer Settings*:



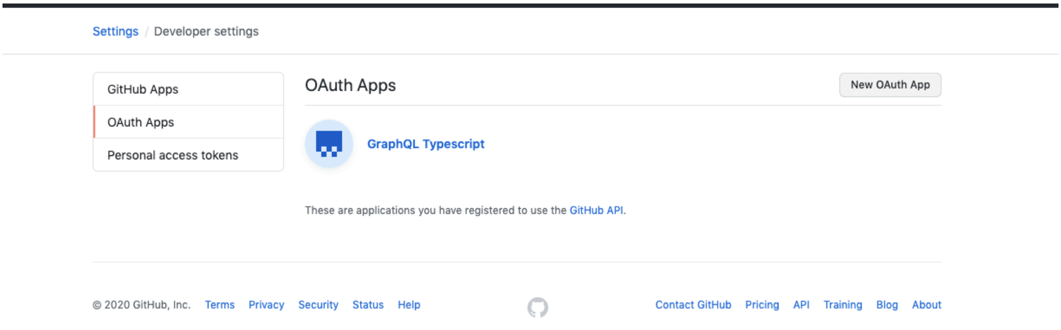
Developer settings

From *Developer settings*, select *OAuth Apps*:



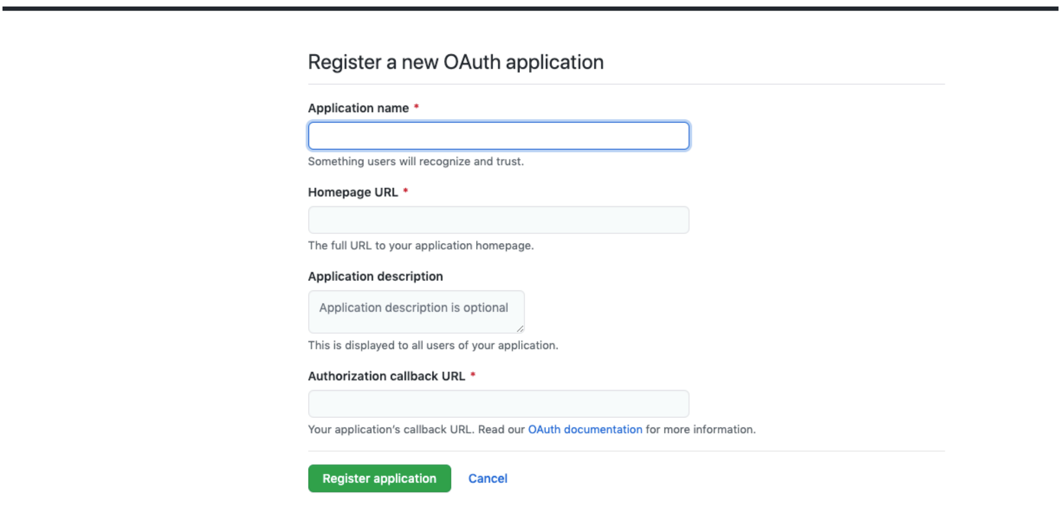
OAuth applications

Once there, click *New OAuth App*:



New OAuth App

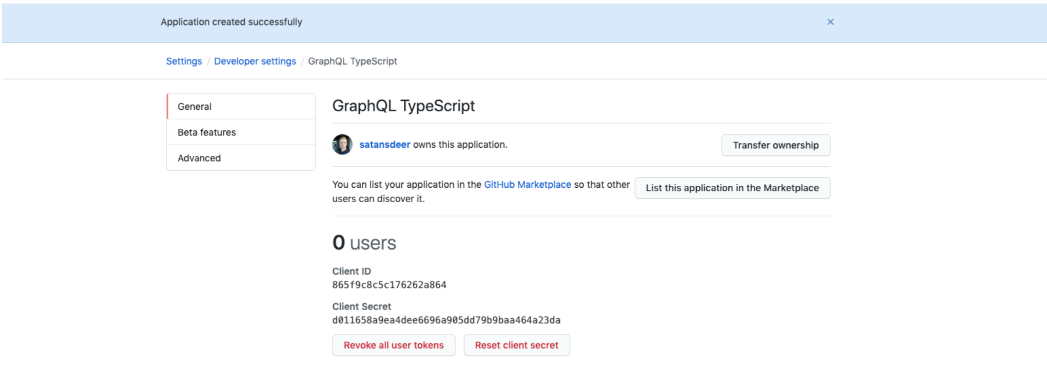
Now enter the info about your application:



New application form

Pick a name for your application and specify any homepage URL.

We specify the return URL to be `http://localhost:3000`. After the user agrees to give us access to the API, GitHub will redirect us to this URL with the authorization token, and we'll need to store the token in the keychain.



Application keys

Now we can construct the URL and start writing authentication code.

Save the `CLIENT_ID` and `CLIENT_SECRET` somewhere safe:

Previewing the final result

The complete code example can be found in `code/06-graphql/completed`.

Unzip the archive that comes with this book and `cd` to the application folder:

```
cd code/06-graphql/completed
```

Open the `.env` file in the root and add the keys from the previous step:

```
CLIENT_ID=af3e6d7f80518e8d23e6
CLIENT_SECRET=dcc2fd666649a9169fce3d8e3b9088ba995cfd0b
```

Install dependencies and launch the application:

```
yarn && yarn start
```

It will open a browser window where you'll need to log in to GitHub and authorize the application to get access to your GitHub resources.

As soon as you've done this, you can try to create issues, pull requests, or repositories.

Setting up the project

Unlike the projects in previous chapters, this project runs in the terminal instead of the browser.

It will be a Node.js application that we'll write in TypeScript. We'll use a custom React renderer called `react-blessed` to be able to render text-based GUI in the terminal.

To start, let's create a new folder for the project. We'll call it `github-client`:

```
mkdir github-client
cd github-client
```

Open the new folder and run `npm init` to generate `package.json`:

```
npm init -y
```

Running TypeScript in the console

There are two major ways to run TypeScript in the console:

- Precompile TypeScript using `tsc` or `babel`.
- Use a TypeScript runtime like `Deno`, `ts-node` or `babel-node`.

We will use `babel-node` for development because it is easier to set up.

First let's install the dependencies:

```
yarn add @apollo/client@3.4.8 \
react@17.0.2 react-blessed@0.7.2 react-devtools@4.8.0 \
react-router@5.2.0 open@7.0.4 keytar@6.0.1 blessed@0.1.81 \
cross-fetch@3.0.5 dotenv@8.2.0 form-data@4.0.0
```

Wow, that's a lot of packages. Here's what they do:

- `@apollo/client` will allow us to perform the GraphQL queries
- `react`, `react-blessed` and `blessed` will render the UI in the Terminal
- `react-router` to navigate between screens
- `open` allows to open webpages in the browser, we'll need it to implement authentication
- `keytar` will let us store the keys in the system keychain
- `form-data` is needed to perform an auth request to GitHub
- `dotenv` will allow us to load the configuration from the `.env` file
- `cross-fetch` is a polyfill that will add the `fetch` method to the node environment

Then we install the dev dependencies:

```
yarn add --dev @babel/core@7.10.4 @babel/node@7.10.4 \
@babel/preset-env@7.10.4 @babel/preset-react@7.10.4 \
@babel/preset-typescript@7.10.4 @babel/register@7.10.4 \
babel-plugin-transform-class-properties@6.24.1 \
msw@0.35.0 react-devtools@4.8.0
```

`react-blessed` and `react-router` don't include type definitions, so we'll install them separately:

```
yarn add @types/react-blessed@0.3.2 @types/react-router@5.1.8
```

Alright, now add the `start` script that will launch `babel-node` with inspector enabled. We'll need it to be able to use the debugger and see logs in the console:

```
  "scripts": {
    "start": "babel-node --inspect src/index.tsx --extensions \".js,.ts\
,.jsx,.tsx,\""
  },
```

Here we pass the `--inspector` parameter to enable the debugger.

Add the .env file

In the root of your project create a `.env` file with the keys that you got in the previous step.

```
CLIENT_ID=af3e6d7f80518e8d23e6
CLIENT_SECRET=dcc2fd666649a9169fce3d8e3b9088ba995cfd0b
```

Running the application

Create an `src` folder. Here we'll define our first component. Create a new file, `src/App.tsx`, with the following code:

```
import React from "react"

export const App = () => {
  return (
    <blessed-box
      style={{
        bg: "#0000ff"
      }}
    >
      Hello React-Blessed
    </blessed-box>
  )
}
```

Now we need to mount our `App` component. Create a new file `src/index.tsx` and add the necessary imports:

```
import React from "react"
import blessed from "blessed"
import { render } from "react-blessed"
import * as dotenv from "dotenv"
import { App } from "./App"
import { MemoryRouter } from "react-router"
```

Next we'll switch the default `console` namespace to the `console` provided by the `inspector` module. We do this to avoid logging directly to the standard output, as we use it to render the text based UI:

```
global.console = require("inspector").console
```

Then load environment variables from the `.env` file:

```
dotenv.config()
```

Initialize the screen by calling `blessed.screen()`:

```
const screen = blessed.screen({
  autoPadding: true,
  smartCSR: true,
  sendFocus: true,
  title: "Github Manager",
  cursor: {
    color: "black",
    shape: "underline",
    artificial: true,
    blink: true
  }
})
```

Add key press event listeners to be able to exit the application:

```
screen.key(["q", "C-c"], () => process.exit(0))
```

We want to close the application when the user presses `q` or a combination of `Ctrl` with letter `c`.

Now render the component tree:

```
render(
  <MemoryRouter>
    <App />
  </MemoryRouter>,
  screen
)
```

As you remember we defined the `start` script in our `package.json`. This script runs our app using the `babel-node`. To be able to use `babel-node` we need to set up Babel properly.

Create a new file `.babelrc` with the following contents:

```
{
  "presets": ["@babel/preset-env", "@babel/preset-typescript", "@babel/\
preset-react"],
  "plugins": ["transform-class-properties"]
}
```

Make sure that you can launch the application, run `yarn start`.

Get the auth code

Define the HTML page

Create a file `src/auth/auth.html` with the following contents:

```html
<!DOCTYPE html>
<html lang="en">
<head>
        <meta charset="UTF-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>You are authenticated</title>
</head>
<body>
  <h1>You are logged in</h1>
        <p>Now you can go back to the command line.</p>
</body>
</html>
```

This page will let the user know if they got authenticated.

Define the `getCode`

Create a new file, `src/auth/getCode.ts`, and add an import block inside:

```ts
import http from "http"
import fs from "fs"
import "cross-fetch/polyfill"
import fetch from "cross-fetch"
import open from "open"
import * as url from "url"
import * as keytar from "keytar"
const FormData = require("form-data")
```

Define the `PORT` constant. We'll need it to run a server that will handle our return URL after GitHub authentication:

```ts
const PORT = 3000
```

Define the `getCode()` function:

```
export const getCode = (): Promise<string> => {
  return new Promise((resolve) => {
  // ...
  })
}
```

Here we create an async function that returns a `Promise`. We pass a callback to the promise constructor, where we can use the `resolve` function to return the code we'll get from GitHub.

Inside of the promise callback we load an html file that we'll show on the return page, and after it's loaded we launch an HTTP server that will serve the return URL for GitHub:

```
  fs.readFile("./src/auth/auth.html", (err, html) => {
    console.log(err)
    http
      .createServer(async (req, res) => {
        if (!req.url) {
          return
        }
// ...
      })
      .listen(PORT)
  })
```

Inside the `createServer` callback we get the `code` from the return url and render the HTML page that we've loaded:

```
          const { code } = url.parse(req.url, true).query

          res.writeHead(200, { "Content-Type": "text/html" })
          res.write(html)
          res.end()
```

Now we need to get the `access_token`. To do it we'll send a `POST` request with `FormData`. We send the `code` along with `CLIENT_ID` and `CLIENT_SECRET` to GitHub's `login` endpoint:

```
        const data = new FormData()
        data.append("client_id", process.env.CLIENT_ID!)
        data.append("client_secret", process.env.CLIENT_SECRET!)
        data.append("code", code)
        data.append("state", "abc")
        data.append("redirect_uri", "http://localhost:3000")

        const { access_token } = await fetch(
          "https://github.com/login/oauth/access_token",
          {
            method: "POST",
            body: data,
            headers: {
              Accept: "application/json"
            }
          }
        ).then((res) => res.json())
```

Here we create a `FormData` and append the following values to it:

- `client_id`: the client ID we received from GitHub for our GitHub App.
- `client_secret`: the client secret we received from GitHub for our GitHub App.
- `code`: the code we received as a response on our return URL.
- `state`: a random string we provided when starting authentication.
- `redirect_url`: a URL to direct the user to after authentication.

Then we call the `fetch()` method with the form data and set the `Accept` header to `application/json`.

Now we have the `access_token`, let's save it in the system storage using `keytar`:

```
await keytar.setPassword(
  "github",
  process.env.CLIENT_ID!,
  access_token
)
```

`keytar` automatically detects what key storages are available in the system. On macOS it will use the Keychain Access app.

After `access_token` is stored we resolve the promise with it.

```
resolve(access_token)
```

Now we need to initiate the authentication. This process happens on the frontend so we need to open the url in the browser. After the code that launches the server, add the following:

```
open(
  `https://github.com/login/oauth/authorize?client_id=${process.env.CLI\
ENT_ID}&scope=user%20read:org%20public_repo%20admin:enterprise&state=ab\
c`
)
```

Here we've constructed a url that opens the `authorize` page and requests a bunch of permisions for our app. We want to be able to fetch user data and create new repositories, issues and pull requests.

Auth Flow Link

In this section we will define an authorization flow link.

Create a new file `src/auth/authFlowLink.ts` with the following imports:

```
import { setContext } from "@apollo/client/link/context"
import { onError } from "@apollo/client/link/error"
import { getCode } from "./getCode"
import { ServerError } from "@apollo/client"
import * as keytar from "keytar"
import { RetryLink } from "@apollo/client/link/retry"
import { HttpLink, from } from "@apollo/client"
```

Then we define the `GITHUB_BASE_URL`, this is where we are going to make all our requests:

```
const GITHUB_BASE_URL = "https://api.github.com/graphql"
```

Now we can define the base HTTP link, it will be responsible for actually making the requests:

```
const httpLink = new HttpLink({ uri: GITHUB_BASE_URL })
```

We are going to preserve the token in memory, lets define variables to do this:

```
// cached storage for the user token
let token: string | null
let tokenInvalid = false
```

Define the token middleware. It will return the cached token, otherwise if the cached token does not exist it will return the token stored in the system keychain or get and return a new one.

```
const withToken = setContext(async (_, { headers = {} }) => {
  if (token) return { token }

  if (tokenInvalid) {
    token = await getCode()
    tokenInvalid = false
  } else {
    token =
      (await keytar.getPassword("github", process.env.CLIENT_ID!)) ||
      (await getCode())
  }

  return { token }
})
```

Define a `withAuthBearer` middleware, it will add an `authorization` header with the `Bearer` token to all the requests:

```
const withAuthBearer = setContext(
  async (_, { headers = {}, token }) => {
    return {
      headers: {
        ...headers,
        authorization: `Bearer ${token}`
      }
    }
  }
)
```

Define a middleware that will reset the token when we receive a server error:

```
const resetToken = onError(({ networkError }) => {
  if ((networkError as ServerError)?.statusCode === 401 && !!token) {
    token = null
    tokenInvalid = true
  }
})
```

We'll also need a retry link that will retry the whole flow if the previous attempt fails:

```
const retry = new RetryLink({
  delay: {
    initial: 300,
    max: Infinity,
    jitter: true
  }
})
```

Now let's combine all of the links into the `authFlowLink` and export it:

```
export const authFlowLink = from([
  retry,
  resetToken,
  withToken,
  withAuthBearer,
  httpLink
])
```

Authentication context

Inside the `src/auth` folder, create a new file called `ClientProvider.tsx` and add the imports:

```
import React, { FC, PropsWithChildren } from "react"
import {
  ApolloProvider,
  ApolloClient,
  InMemoryCache
} from "@apollo/client"
import { authFlowLink } from "./authFlowLink"
```

Define the `ClientProvider` component:

```
export const ClientProvider: FC<PropsWithChildren<{}>> = ({
  children
}) => {
  const client = new ApolloClient({
    cache: new InMemoryCache(),
    link: authFlowLink
  })

  return <ApolloProvider client={client}>{children}</ApolloProvider>
}
```

Here we initialize an Apollo client with the `authFlowLink` that we defined in the previous chapter. Then we pass the client to the `ApolloProvider`.

Now wrap our applcation into the `ClientProvider`. Open `src/index.tsx` and import `ClientProvider`:

```
import { ClientProvider } from "./auth/ClientProvider"
```

Wrap the application into `ClientProvider`:

```
render(
  <MemoryRouter>
    <ClientProvider>
      <App />
    </ClientProvider>
  </MemoryRouter>,
  screen
)
```

Now run the application:

```
yarn start
```

The app should launch successfully.

GraphQL queries. Getting user data

Let's make our first query.

Create a new file, `src/WelcomeWindow.tsx`, that we'll use to define the `WelcomeWindow` component

In this component, we want to load the currently authenticated user's data and display it in a window.

First, add the necessary imports:

```
import React from "react"
import { useQuery, gql } from "@apollo/client"
```

Then define a constant for the user info query:

```
const GET_USER_INFO = gql`
  query getUserInfo {
    viewer {
      name
      bio
    }
  }
`
```

If you go to GitHub API documentation[236], you'll see that this query returns an object with the `viewer` field that contains subfields with user data. We'll use two of these subfields: `name` and `bio`. Let's define a type for this query:

```
type UserInfo = {
  viewer: {
    name: string
    bio: string
  }
}
```

Now let's define the actual component:

```
export const WelcomeWindow = () => {
  const { loading, data } = useQuery<UserInfo>(GET_USER_INFO)

  return <>{loading ? "Loading..." : JSON.stringify(data)}</>
}
```

Here we use the `useQuery` hook to perform the query. This hook will make a request immediately after the component mounts:

When we call `useQuery()`, three variables are returned:

- `isLoading` is a boolean flag that shows if we are still waiting for a response from the server.

[236]https://docs.github.com/en/graphql

- `data` is our data. You can provide a type argument to the `useQuery()` hook to specify the type of the data.
- `error`: if something goes wrong, this object will contain information about the error.

We show a loader while `isLoading` is true, and when loading completes, we show values from the `data` object.

For now, we just render parsed JSON of the data that we got from the GitHub API.

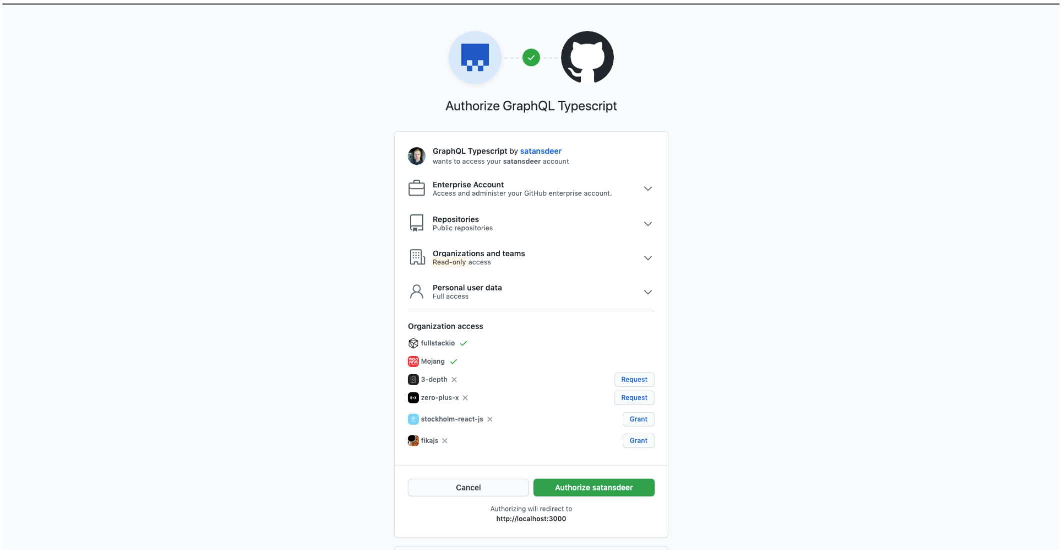
Open `src/App.tsx` and render the `WelcomeWindow` component:

```
import React from "react"
import { WelcomeWindow } from "./WelcomeWindow"

export const App = () => {
  return (
    <blessed-box
      style={{
        bg: "#0000ff"
      }}
    >
      <WelcomeWindow />
    </blessed-box>
  )
}
```

Now launch the application and make sure that data is displayed:

```
yarn start
```

As we are not authenticated yet the `FlowLink` will try to get the authentication token. You should see the following page in your browser:



Authentication page

Click the *Authorize* button, and then return to the command line.

You should see something like this:

Getting user data from GitHub

If everything is OK, we can start adding a proper layout.

Adding helper components

Before we implement the main functionality we'll add the following helper components:

- Button.tsx
- List.tsx
- Field.tsx
- Form.tsx
- Panel.tsx
- Text.tsx
- TextBox.tsx

All of them will go into the `src/shared` folder.

Define the `Button` component

Here we first import React:

```
import React from "react"
```

Then we define the props type for the button:

```
type ButtonProps = {
  children: string
} & any
```

Define and export the `Button` component itself:

```
export const Button = ({ children, ...rest }: ButtonProps) => {
  return (
    <blessed-button
      content={`< ${children} >`}
      mouse
      focused
      height={1}
      width={children.length + 4}
      align="center"
      left="center"
```

```
      bottom={1}
      bg="blue"
      fg="white"
      {...rest}
    />
  )
}
```

Define the `List` component

```
import React, { FC, forwardRef } from "react"

type ListItem = {
  content: string
}

type ListProps = {
  top?: string | number
  left?: string | number
  right?: string | number
  bottom?: string | number
  height?: string | number
  width?: string | number
  onAction?(item: ListItem): void
  items: string[]
}

export const List = forwardRef<any, ListProps>(
  ({ onAction, items, ...rest }, ref) => {
    return (
      <blessed-list
        ref={ref}
        onAction={onAction}
        focused
```

```
          mouse
          keys
          vi
          items={items}
          style={{
            bg: "white",
            fg: "black",
            selected: {
              bg: "blue",
              fg: "white"
            },
            border: {
              type: "line"
            }
          }}
          {...rest}
        />
      )
    }
)
```

Define the `Text` component

```
import React from "react"

type TextProps = {
  children: string
} & any

export const Text = ({ children, ...rest }: TextProps) => {
  return (
    <blessed-text
      width={children.length}
      content={children}
```

```
      style={{
        bg: "white",
        fg: "black"
      }}
      {...rest}
    />
  )
}
```

Define the `TextBox` component

Make the imports:

```
import React, { FC } from "react"
```

Define the props type:

```
type TextBoxProps = {
  top?: string | number
  bottom?: string | number
  left?: string | number
  onSubmit(): void
}
```

Define and export the `TextBox` component:

```
export const TextBox: FC<TextBoxProps> = ({ onSubmit, ...rest }) => {
  return (
    <blessed-textbox
      height={1}
      style={{
        bg: "white",
        fg: "black"
      }}
      keys
      inputOnFocus
      mouse
      onSubmit={onSubmit}
      {...rest}
    />
  )
}
```

Define the `Panel` component

If you've launched the application from the example folder, you saw that it rendered a window, a.k.a. panel, on each screen. Let's define a component for it.

Create a new file, `src/shared/Panel.tsx`, and add the necessary imports:

```
import React, { PropsWithChildren, FC } from "react"
import { forwardRef } from "react"
```

Then define a type for the `Panel` component's props:

```
type PanelProps = {
  top?: number | string
  left?: number | string
  right?: number | string
  bottom?: number | string
  width?: number | string
  height?: number | string
}
```

Define the `Panel` component:

```
export const Panel = forwardRef<any, PropsWithChildren<PanelProps>>(
  ({ children, ...rest }, ref) => {
    return (
      <blessed-box
        ref={ref}
        focused
        mouse
        shadow
        border={{
          type: "line"
        }}
        keys
        align="center"
        style={{
          bg: "white",
          shadow: true,
          border: {
            bg: "white",
            fg: "black"
          },
          label: {
            bg: "white",
            fg: "black"
          }
        }}
```

```
        {...rest}
      >
        {children}
      </blessed-box>
    )
  }
)
```

Form helper components

Create a new file, `src/shared/Form.tsx`, and add these imports:

```
import React, {
  PropsWithChildren,
  FC,
  ReactNode,
  useRef
} from "react"
```

Define the types for our form:

```
export type FormValues = {
  textbox: string[]
}
  // ...
type FormProps = {
  onSubmit(values: FormValues): void
  children(triggerSubmit: () => void): ReactNode
}
```

Here we define `children` to be a function. We need to do this to be able to send the `triggerSubmit()` function to form children. Unfortunately, `react-blessed` does not trigger a form's `onSubmit()` automatically when its inputs are submitted, which forces us to use this little hack.

Define the `Form` component:

```
export const Form: FC<FormProps> = ({ children, onSubmit }) => {
  const form = useRef<any>()

  const triggerSubmit = () => {
    form.current.submit()
  }

  React.useEffect(() => {
    setTimeout(() => {
      form.current.focus()
    }, 0)
  }, [])

  return (
    <blessed-form
      top={3}
      keys
      focused
      ref={form}
      style={{
        bg: "white"
      }}
      onSubmit={onSubmit}
    >
      {children(triggerSubmit)}
    </blessed-form>
  )
}
```

- We define the `triggerSubmit()` function that will call the `submit()` method on our form when triggered.
- We define `useEffect()` to automatically focus the form when the component is mounted.
- In the `Form` layout, we render the `children()` function and pass `triggerSubmit()` as an argument.

Now define the `Field` component. Create a new file called `src/shared/Field.tsx` and add imports:

```
import React from "react"
import { FC } from "react"
import { TextBox } from "./TextBox"
import { Text } from "./Text"
```

Then define a type for the component's props:

```
type FieldProps = {
  label: string
  top?: number | string
  onSubmit(): void
}
```

- `label` will be displayed in front of input.
- `top` represents an offset from the top.
- `onSubmit()` is an input submit handler that triggers on pressing the `Enter` key.

Define the `Field` component:

```
export const Field: FC<FieldProps> = ({ label, top, onSubmit }) => {
  return (
    <>
      <Text top={top}>{label}</Text>
      <TextBox top={top} left={label.length} onSubmit={onSubmit} />
    </>
  )
}
```

In this component, we render a label and a text box. We'll have a lot of these in our forms, so it's better to have them as a reusable component.

Informationbal message components

In our resource related components, we'll need to show error and success messages to the users.

Entity error component

For example when the user tries to create a new pull request, repository or issue and the server returns an error, we'll need to show an error message. Let's define a component for this. Create a new file `src/shared/NewEntityError.tsx` and add the following code:

```
import React, { useRef, useEffect } from "react"
import { Panel } from "./Panel"
import { Text } from "./Text"
import { Button } from "./Button"

type NewEntityErrorProps = {
  onClose(): void
  error: Error
}

export const NewEntityError = ({
  onClose,
  error
}: NewEntityErrorProps) => {
  // ...
}
```

This component will accept an error message and render it. It will also accept an `onClose()` function that will be called when the user clicks on the close button or press enter.

Define the component body:

```
const ref = useRef<any>()

useEffect(() => {
  ref.current.key("enter", onClose)
  return () => {
    ref.current.unkey("enter", onClose)
  }
}, [])

return (
  <Panel ref={ref} top="25%" left="center" height={10}>
    <Text left="center">An error occured</Text>
    <Text left="center" top={3}>
      {error.message}
    </Text>

    <Button left="center" bottom={1} onPress={onClose}>
      Enter:OK
    </Button>
  </Panel>
)
```

Here we render the layout and listen to keyboard events in the `useEffect()` hook.

Entity success component

When the user successfully creates a new entity, we'll need to show a success message. Let's define a component for this. Create a new file `src/shared/NewEntitySuccess.tsx` and add the following code:

```
import open from "open"
import React, { useRef, useEffect, useCallback } from "react"
import { Panel } from "./Panel"
import { Text } from "./Text"
import { Button } from "./Button"
import { debounce } from "../utils/debounce"

type NewEntitySuccessProps = {
  url: string
  title: string
  onClose(): void
}

export const NewEntitySuccess = ({
  url,
  title,
  onClose
}: NewEntitySuccessProps) => {
  // ...
}
```

This component will allow the user to open the entity in the browser. This is why we import the `open` function. Just like with the error component we'll accept the `onClose` callback. We'll also accept the entity's URL.

Add this to the component body:

```
  const ref = useRef<any>()

  const openUrl = useCallback(
    debounce(() => open(url), 100),
    [url]
  )

  useEffect(() => {
  // ...
```

```
  }, [])

  return (
    <Panel ref={ref} top="25%" left="center" height={10}>
      <Text left="center">{title}</Text>

      <Button left="center" bottom={3} onPress={openUrl}>
        o: Open in the browser
      </Button>
      <Button left="center" bottom={1} onPress={onClose}>
        Enter: Ok
      </Button>
    </Panel>
  )
```

What is left is to define the `useEffect` where we'll subscribe to keyboard events and call the `onClose` and `openUrl` functions:

```
useEffect(() => {
  ref.current.key("enter", onClose)
  ref.current.key("o", openUrl)

  return () => {
    ref.current.unkey("enter", onClose)
    ref.current.unkey("o", openUrl)
  }
}, [])
```

That's it, we are ready to continue with the next step and define the `WelcomeWindow` component.

Defining the WelcomeWindow layout

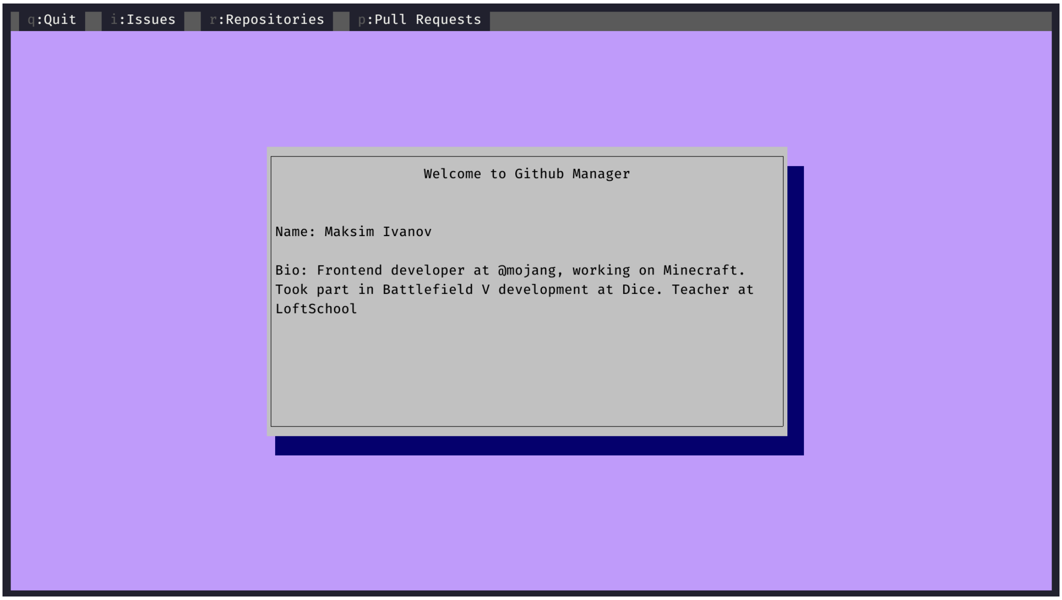
Go to `src/WelcomeWindow.tsx` and add the `Text` and `Panel` components:

```
import { Text } from "./shared/Text"
import { Panel } from "./shared/Panel"
```

Edit the `WelcomeWindow` layout:

```
return (
  <Panel height={12} left="center" top="25%">
    <Text left="center">Welcome to Github Manager</Text>
    {loading ? (
      <Text top={3}>Loading...</Text>
    ) : (
      <>
        <Text top={3}>{`Name: ${data?.viewer.name}`}</Text>
        <Text top={5} width={50}>{`Bio: ${data?.viewer.bio}`}</Text>
      </>
    )}
  </Panel>
)
```

Now if you launch the application again, you should see this:



Main screen

Getting GitHub GraphQL schema

We have just written our first query, but we had to provide types for it manually.

In fact, type information is already available in the GraphQL schema, and we just need to extract it to use with TypeScript.

To extract type information, we first need to obtain the full GraphQL schema definition.

The Apollo CLI allows to do this. Let's install it as a dev dependency:

```
yarn add --dev apollo
```

> If you get an error about `Ineffective mark-compacts near heap limit...` try to run this command instead: `yarn add --dev --max_old_space_size=8196 apollo`. Read more about this here[237].

[237] https://stackoverflow.com/questions/53230823/fatal-error-ineffective-mark-compacts-near-heap-limit-allocation-failed-javas

The request that we are going to make requires authentication so we'll need to get the Bearer token. Lucky for us we already have it, because we made an authenticated request to retrieve user data.

Run the following command to get the authentication token:

```
node -e "require('dotenv').config(); \
  require('keytar')\
        .getPassword('github', process.env.CLIENT_ID)\
    .then(console.log)"
```

> This command will only work if you've already run the application and made an authenticated request to GitHub API.

Now run the following command in the terminal:

```
yarn run apollo schema:download \
--header="Authorization: Bearer <token>" \
--endpoint=https://api.github.com/graphql \
graphql-schema.json
```

> Change the `<token>` to your token that you got from the previous command.

This script will download the schema and save it to a JSON file.

Generating types

We can now generate TypeScript types from the downloaded schema.

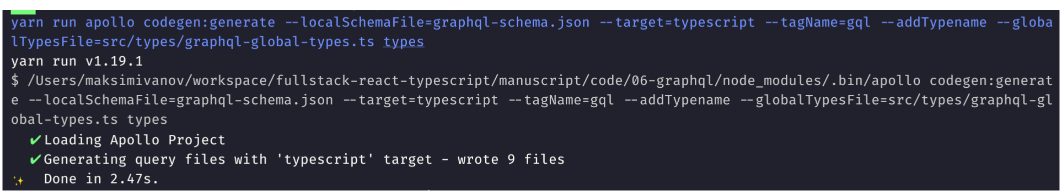
Apollo provides a special CLI utility to get TypeScript types from a GraphQL schema. Run it like this:

```
yarn run apollo codegen:generate \
--localSchemaFile=graphql-schema.json \
--target=typescript \
--tagName=gql \
--addTypename \
--globalTypesFile=src/types/graphql-global-types.ts \
types
```

We pass the following options to the `codegen` script:

- `localSchemaFile` - the json file that we created on the previous step
- `target` - the target language for the types
- `tagName` - the template literal that will contain the queries
- `addTypename` - will add the `__typename` to your queries
- `globalTypesFile` - will override the default types file path. The default one is `globalTypes.d.ts`

If everything goes fine, you should see output similar to this:



Types generated successfully

The script that we just ran created a new folder `src/types`. If you open the folder, you'll see type definitions for the `getUserInfo()` query. This file has the linters disabled and explicitly states that it is generated automatically and should not be edited:

```
/* tslint:disable */
/* eslint-disable */
// @generated
// This file was automatically generated and should not be edited.
```

If you look at the contents you'll see that it exports the types for our query:

```
// ====================================================
// GraphQL query operation: getUserInfo
// ====================================================

export interface getUserInfo_viewer {
  __typename: "User"
  /**
   * The user's public profile name.
   */
  name: string | null
  /**
   * The user's public profile bio.
   */
  bio: string | null
}

export interface getUserInfo {
  /**
   * The currently authenticated user.
   */
  viewer: getUserInfo_viewer
}
```

From now on, every time we write new GraphQL queries or mutations, we'll run this code generator to get types for those queries.

Let's now update our code to use the automatically generated types instead of our custom types.

Remove the `UserInfo` type, open `src/WelcomeWindow.tsx` and import the generated types:

```
import { getUserInfo } from "./types/getUserInfo"
```

Change the call to `useQuery()` to this:

```
const { loading, data } = useQuery<getUserInfo>(GET_USER_INFO)
```

Adding routing

Right now we only have one window that greets the user and shows profile information.

We want to let the user navigate between different pages. To do this, we'll use the `react-router` library.

Define the resource screens

Create three new folders, one for each resource type. Create an `index.ts` file and the component file inside each folder. The resulting file structure should look like this:

- src
  - /Issues
    - index.ts * Issues.tsx
  - /Repositories
    - index.ts * Repositories.tsx
  - /PullRequests
    - index.ts * PullRequests.tsx

Inside each `index.ts` file, export everything from the corresponding component file. For example, `src/Issues/index.ts` should look like this:

```
export * from "./Issues"
```

Define and export the component from the component file. For issues it is going to be `src/Issues/Issues.tsx` and it's will look like this:

```
import React from "react"
import { Panel } from "../shared/Panel"
import { Text } from "../shared/Text"

export const Issues = () => {
  return (
    <Panel height={10} top="25%" left="center">
      <Text>Issues</Text>
    </Panel>
  )
}
```

Repeat for each of the remaining resources.

Define the routing scheme

Go to `src/App.tsx` and add the following imports:

```
import { Issues } from "./Issues"
import { Repositories } from "./Repositories"
import { PullRequests } from "./PullRequests"
import { Switch, Route } from "react-router"
```

We'll use `Switch` and `Route` to define routing, and the `useHistory()` hook to navigate between pages.

Define a `Switch` with routes inside the `blessed-box` element:

```
<Switch>
  <Route exact path="/" component={WelcomeWindow} />
  <Route path="/issues" component={Issues} />
  <Route path="/repositories" component={Repositories} />
  <Route path="/pull-requests" component={PullRequests} />
</Switch>
```

Here we define routes for three more pages: repositories, issues, and pull requests.

Implement navigation

Now we can define the navigation panel using a component called `blessed-listbar`. It allows you to render a list of options with associated keys. When the user presses a key, it triggers the associated callback.

Define the `debounce` function

Before we implement the navigation component we'll have to define `debounce` function. There is a bug in `react-blessed` that causes the keyboard and mouse event callbacks to be executed multiple times. This can cause problems with navigation.

To prevent this bug from happening we'll wrap our callbacks into the `debounce` function. This function will limit the amount of calls per time unit. For example we'll be able to say that the navitgation should happen only once per 100 milliseconds.

Create a new file `src/utils/debounce.ts` with the following contents:

```
export function debounce<T extends unknown[], U>(
  callback: (...args: T) => PromiseLike<U> | U,
  wait: number
) {
  let timer: ReturnType<typeof setTimeout>

  return (...args: T): Promise<U> => {
    clearTimeout(timer)
    return new Promise((resolve) => {
      timer = setTimeout(() => resolve(callback(...args)), wait)
    })
  }
}
```

In this function we set a new timer every time the wrapped function is called.

Define the `Header`

Create a new folder `src/Header` and define an `index.ts` file there:

```
export * from "./Header"
```

Create the `Header.tsx` in the same folder and make the following imports:

```
import React, { useCallback } from "react"
import { useHistory, useLocation } from "react-router"
import { debounce } from "../utils/debounce"
```

Define and export the `Header` component:

```
export const Header = () => {
  // ...
}
```

Inside of the component get the `history` and `location` objects using the hooks from `react-router`:

```
const history = useHistory()
const location = useLocation()
```

Define the navigation callbacks:

```
  const goToIssues = useCallback(
    debounce(() => history.push("/issues"), 100),
    []
  )

  const goToRepositories = useCallback(
    debounce(() => history.push("/repositories"), 100),
    []
  )

  const goToPRs = useCallback(
    debounce(() => history.push("/pull-requests"), 100),
    []
  )

  const goToRoot = useCallback(
    debounce(() => history.push("/"), 100),
    []
  )
```

Here we define four callbacks, one for each page, including the home page. Since we are using the `react-router` library, we can take advantage of the `history` object to perform navigation programmatically.

Render the layout, we are going to use the `blessed-listbar` component:

```
return (
  <blessed-listbar
    height={1}
    items={{
      Quit: {
        keys: "q"
      },
      Issues: {
        keys: "i",
        callback: goToIssues
      },
      Repositories: {
        keys: "r",
        callback: goToRepositories
      },
      "Pull Requests": {
        keys: "p",
        callback: goToPRs
      },
      ...(location.pathname !== "/" && {
        "Back to main screen": {
          keys: "b",
          callback: goToRoot
        }
      })
    }}
    style={{
      bg: "grey",
      height: 1
    }}
  />
)
```

This component accepts a config object with the navigation items. We use the `location` to render the `Back to main screen` button conditionally.

Render the `Header`

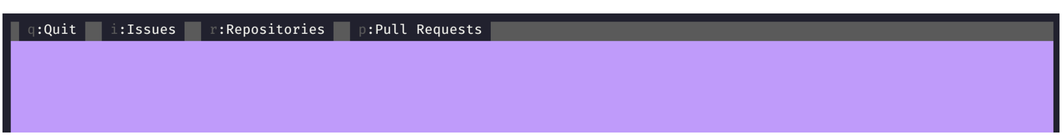
Go to `src/App.tsx` and import the `Header` component:

```
import { Header } from "./Header"
```

Render the `Header` above the `Switch` element:

```
return (
  <blessed-box
    style={{
      bg: "#0000ff"
    }}
  >
    <Header />
    <Switch>
      <Route exact path="/" component={WelcomeWindow} />
      <Route path="/issues" component={Issues} />
      <Route path="/repositories" component={Repositories} />
      <Route path="/pull-requests" component={PullRequests} />
    </Switch>
  </blessed-box>
)
```

Launch the application and make sure you can navigate between pages:



Navigation bar

Try pressing assigned keys to see if navigation works.

Repositories main component

In our application, the user should be able to list their existing repositories and create new repositories. We'll achieve this by defining these three components:

- `RepositoriesMain` will show links to two other routes.
- `NewRepository` will contain a form to create new repositories.
- `RepositoriesList` will show a scrollable list of existing repositories.

Let's start with the main component for repositories.

Create a new file, `src/Repositories/RepositoriesMain.tsx`, and add import statements:

```
import React from "react"
import { useHistory, useRouteMatch } from "react-router"
import { useRef } from "react"
import { Panel } from "../shared/Panel"
import { Button } from "../shared/Button"
import { Text } from "../shared/Text"
```

Then define the actual component with the following layout:

```
export const RepositoriesMain = () => {
  // ...
  const ref = useRef<any>()
  // ...
  return (
    <Panel ref={ref} height={11} top="25%" left="center">
      <Text left="center">Repositories</Text>
      <Text top={2} left="center">
        Click on the button or press the corresponding key.
      </Text>

      <Button left="center" bottom={3}>
```

```
        l:List Repositories
      </Button>

      <Button left="center" bottom={1}>
        c:Create New Repository
      </Button>
    </Panel>
  )
}
```

Here we display instructions on navigating to other pages. We also get the reference to the panel. We'll use the `ref` to attach the scheen specific event listeners. Add this code before the layout:

```
const history = useHistory()
const match = useRouteMatch()
// ...
React.useEffect(() => {
  ref.current.key("c", () => history.push(`${match.url}/new`))
  ref.current.key("l", () => history.push(`${match.url}/list`))
}, [])
```

Here we listen to keyboard events. If the user presses `c` we navigate to the repo creation screen, and if the user presses `l` we go to the repos list.

Import the main component into `src/Repositories/Repositories.tsx`:

```
import React from "react"
import { Route, Switch, useRouteMatch } from "react-router"
import { RepositoriesMain } from "./RepositoriesMain"
```

Define the `NewRepository` and `RepositoriesList` components stubs:

```
const NewRepository = () => <>New Repository</>
const RepositoriesList = () => <>List Repositories</>
```

Update the `Repositories` component layout:

```
export const Repositories = () => {
  const match = useRouteMatch()

  return (
    <Switch>
      <Route exact path={match.path} component={RepositoriesMain} />
      <Route path={`${match.path}/new`} component={NewRepository} />
      <Route
        path={`${match.path}/list`}
        component={RepositoriesList}
      />
    </Switch>
  )
}
```

Getting the list of repositories

In this section we'll define a component that will render the list of repositories. Create a new file, `src/Repositories/RepositoriesList.tsx`, and add the following imports:

```
import React, { useRef } from "react"
import { Panel } from "../shared/Panel"
import { useEffect } from "react"
import open from "open"
import { useQuery, gql } from "@apollo/client"
import { List } from "../shared/List"
import { Text } from "../shared/Text"
```

Let's define a query that will fetch the list of available repositories:

```
const LIST_REPOSITORIES = gql`
  query listRepositories {
    viewer {
      repositories(first: 100) {
        nodes {
          name
          url
        }
      }
    }
  }
`
```

Define the `RepositoriesList` component:

```
export const RepositoriesList = () => {
  // ...
}
```

This component will perform a GraphQL query to fetch the repositories, so we need to generate the type for it. Run this script in the project root:

```
yarn run apollo codegen:generate \
--localSchemaFile=graphql-schema.json \
--target=typescript \
--tagName=gql \
--addTypename \
--globalTypesFile=src/types/graphql-global-types.ts \
types
```

After running the generator you should have a new folder `src/Repositories/types`.

Go back to the `src/Repositories/RepositoriesList.tsx` and import the generated types:

```
import { listRepositories } from "./types/listRepositories"
```

Inside the component call the `useQuery()` hook with the query that we've defined:

```
const { loading, error, data } =
  useQuery<listRepositories>(LIST_REPOSITORIES)
```

Here we provide the types generated from the query and get the `data`, `error` and the `loading` flag.

Get the repos list from the data:

```
const repos = data?.viewer.repositories.nodes
```

Use the `useQuery` return values to render the component layout. First we need to process the loading state:

```
if (loading) {
  return (
    <Panel height={10} top="25%" left="center">
      <Text left="center">Loading...</Text>
    </Panel>
  )
}
```

If we get an error - we need to render the error message:

```
if (error) {
  return <>Error: {JSON.stringify(error)}</>
}
```

If we got the data successfully we render the `blessed-list` component:

```
  return (
    <Panel height={10} top="25%" left="center">
      <Text left="center">List Repositories</Text>

      <List
  // ...
        top={2}
        onAction={(el) =>
          open(
            repos?.find((repo) => repo?.name === el.content)?.url ||
              ""
          )
        }
        items={repos?.map((repo) => repo?.name || "") || []}
      />
    </Panel>
  )
```

When we open this screen we want to focus on the list automatically, so that the user won't have to make an extra click before selecting the repo in the list. Define the `listRef` and pass it to the list element:

```
const listRef = useRef<any>()
// ...
    <List
      ref={listRef}
// ...
    />
```

Add a `useEffect()` call before the layout:

```
useEffect(() => {
  listRef?.current?.focus()
}, [data])
```

Here we call the list element `focus` method after we mount the component.

Open `src/Repositories/Repositories.tsx` and change it to use the real `RepositoriesList` component:

```
import React from "react"
import { Route, Switch, useRouteMatch } from "react-router"
import { RepositoriesMain } from "./RepositoriesMain"
import { RepositoriesList } from "./RepositoriesList"

const NewRepository = () => <>New Repository</>

export const Repositories = () => {
  const match = useRouteMatch()

  return (
    <Switch>
      <Route exact path={match.path} component={RepositoriesMain} />
      <Route path={`${match.path}/new`} component={NewRepository} />
      <Route
        path={`${match.path}/list`}
        component={RepositoriesList}
      />
```

```
    </Switch>
  )
}
```

Run the application to make sure it works:

```
yarn start
```

Here you might get a data merge error. To fix it we will set the merge strategy for the `User` model.

Open `src/auth/ClientProvider.tsx` and add the following code:

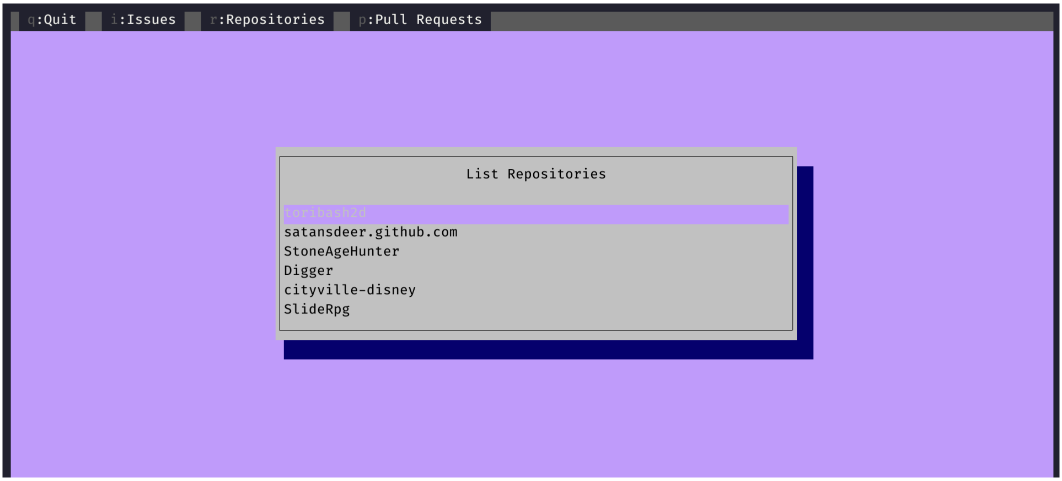
```
const cache = new InMemoryCache({
  typePolicies: {
    User: {
      merge: true
    }
  }
})
```

Here we define the merge policy for the `User` model. This way when we fetch the new data from GitHub - Apollo will try to merge the user data instead of overriding it.

Add the `cache` argument to the `ApolloClient`:

```
  children
}) => {
  const client = new ApolloClient({
    cache,
```

Launch the app again and try to get the list of repositories. What you see should look like this:



List of repositories

GraphQL mutations. Creating repositories

So far we've only been fetching data. It's time to write our first mutation to create new repositories.

Create a new file, `src/Repositories/NewRepository.tsx`, and add these imports:

```
import React, { useState } from "react"
import { useMutation, gql } from "@apollo/client"
import { Field } from "../shared/Field"
import { Text } from "../shared/Text"
import { Button } from "../shared/Button"
import { Form, FormValues } from "../shared/Form"
import { Panel } from "../shared/Panel"
import { NewEntitySuccess } from "../shared/NewEntitySuccess"
import { NewEntityError } from "../shared/NewEntityError"
```

Then let's define the actual mutation:

```
const CREATE_REPOSITORY = gql`
  mutation createNewRepository(
    $name: String!
    $description: String!
    $visibility: RepositoryVisibility!
  ) {
    createRepository(
      input: {
        name: $name
        description: $description
        visibility: $visibility
      }
    ) {
      repository {
        name
        url
        id
      }
    }
  }
`
```

Now we can run the code generator to get types:

```
yarn run apollo codegen:generate \
--localSchemaFile=graphql-schema.json \
--target=typescript \
--tagName=gql \
--addTypename \
--globalTypesFile=src/types/graphql-global-types.ts \
types
```

Import the generated types:

```
import {
  createNewRepository_createRepository_repository,
  createNewRepository,
  createNewRepositoryVariables
} from "./types/createNewRepository"
import { RepositoryVisibility } from "../types/graphql-global-types"
```

Next, define the `NewRepository` component:

```
export const NewRepository = () => {
  // ...
  const onSubmit = async (values: FormValues) => {
    const [name, description] = values.textbox
  // ...
  }
  // ...
  return (
    <Panel top="25%" left="center" height={12}>
      <Text left="center">New repository</Text>
      <Form onSubmit={onSubmit}>
        {(triggerSubmit) => {
          return (
            <>
              <Field
                top={0}
                label="Name: "
                onSubmit={triggerSubmit}
              />
              <Field
                top={1}
                label="Description: "
                onSubmit={triggerSubmit}
              />
            </>
          )
        }}
```

```
      </Form>
  // ...
    </Panel>
  )
}
```

Add hint regarding the use of the `Tab` button and the submit button belown the form:

```
<Text left="center" bottom={3}>
  Tab: Next Field
</Text>
<Button left="center" bottom={1} onPress={onSubmit}>
  Enter: Submit
</Button>
```

Here we have a form and an `onSubmit()` handler that for now just extracts the `name` and `description` values from the form inputs.

To use the mutation, add this code to the beginning of the component:

```
const [createrepository] = useMutation<
  createNewRepository,
  createNewRepositoryVariables
>(CREATE_REPOSITORY)
```

Here we're using the `useMutation()` hook from `react-apollo` to get the `createRepository` function. Let's call the `createrepository()` mutation inside the `onSubmit()` callback:

```
const onSubmit = async (values: FormValues) => {
  const [name, description] = values.textbox

  try {
    const result = await createrepository({
      variables: {
        name,
        description,
        visibility: RepositoryVisibility.PUBLIC
      }
    })
// ...
  } catch (error) {
// ...
  }
}
```

Make sure that `onSubmit()` is an `async` function.

Since we provide automatically generated types to `createrepository()`, we'll be getting correct data in return. We get type suggestions when we pass variables to it:

Type suggestions

Now that we have received `result` from the mutation, we want to store it in a state. Define the `repository` state:

```
const [repository, setRepository] =
  useState<createNewRepository_createRepository_repository | null>()
```

Save `result` from the mutation call using this state:

```
  const onSubmit = async (values: FormValues) => {
    const [name, description] = values.textbox

    try {
      const result = await createrepository({
        variables: {
          name,
          description,
          visibility: RepositoryVisibility.PUBLIC
        }
      })

      setRepository(result.data?.createRepository?.repository)
    } catch (error) {
  // ...
    }
  }
```

We also need to handle errors. Define the `error` state:

```
const [error, setError] = useState<Error | null>()
```

Update the `onSubmit` callback to handle the `error` state:

```
  const onSubmit = async (values: FormValues) => {
    const [name, description] = values.textbox

    try {
      const result = await createrepository({
        variables: {
          name,
          description,
          visibility: RepositoryVisibility.PUBLIC
        }
      })
```

```

      setRepository(result.data?.createRepository?.repository)
    } catch (error) {
      setError(error)
    }
  }
```

Now let's handle the success and the error states in the component layout.

Add an early return and render the success screen in case a `repository` is already in the state:

```
if (repository) {
  return (
    <NewEntitySuccess
      title="New repository created"
      url={repository.url}
      onClose={() => setRepository(null)}
    />
  )
}
```

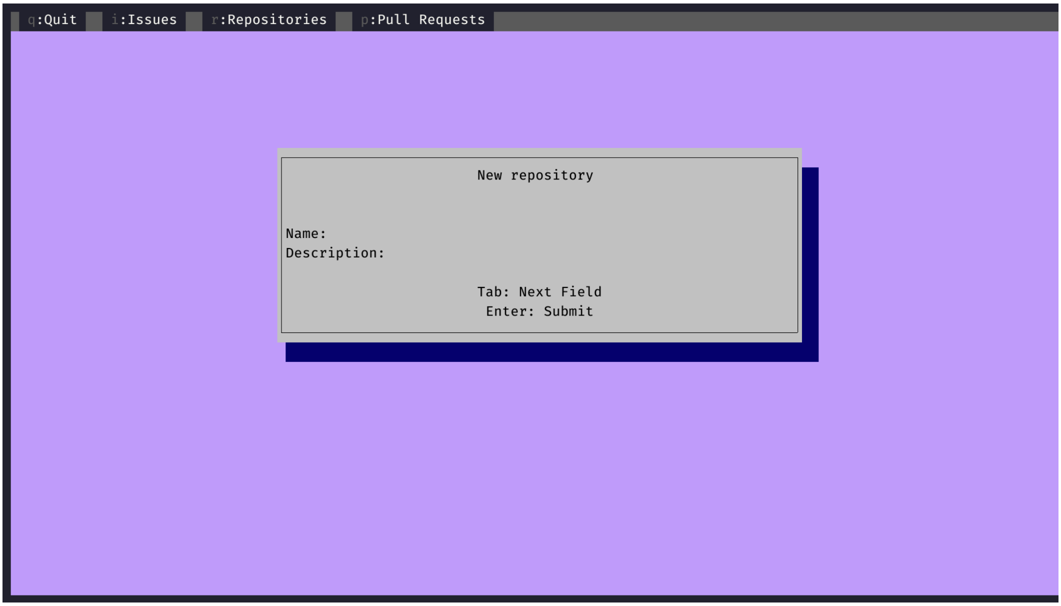
To handle the error add another early return and render the error screen:

```
if (error) {
  return (
    <NewEntityError error={error} onClose={() => setError(null)} />
  )
}
```

Our component is ready, now go to `src/Repositories/Repositories.tsx` and import the real `NewRepository` component:

```
import { NewRepository } from "./NewRepository"
```

Remove the stubbed out `NewRepository` component and launch the application to see if everything works:



Create repository form

Try to create a new repository and navigate to it.

Getting the repository ID

Before we move on to other resources, we will create a shared query that will get the ID of the repository by its name. We'll use this id to get or create the issues and pull requests for the given repository.

Create a new file, `src/queries/getRepository.ts`, with the following code:

```
import { gql } from "@apollo/client"

export const GET_REPOSITORY = gql`
  query getRepository($owner: String!, $name: String!) {
    repository(owner: $owner, name: $name) {
      id
    }
  }
`
```

Here we want to find the repository by the owner and name. In the query we specify that we want only the id field.

Run the code generator to get types for the query:

```
yarn run apollo codegen:generate\
  --localSchemaFile=graphql-schema.json\
  --target=typescript\
        --tagName=gql\
        --addTypename\
        --globalTypesFile=src/types/graphql-global-types.ts\
  types
```

Make sure that you have the `src/queries/types` folder with types for this query.

Working with GitHub issues

We can now start working on GitHub issues. Issues are basically discussions bound to specific repositories. Let's define the navigation component first. Create a new file, `src/Issues/IssuesMain.tsx`, and start with adding imports:

```
import React from "react"
import { useHistory, useRouteMatch } from "react-router"
import { useRef } from "react"
import { Panel } from "../shared/Panel"
import { Button } from "../shared/Button"
import { Text } from "../shared/Text"
```

Then define the `IssuesMain` component with the following layout:

```
export const IssuesMain = () => {
  // ...
  const ref = useRef<any>()
  // ...
  return (
    <Panel ref={ref} height={11} top="25%" left="center">
      <Text left="center">Issues</Text>
      <Text top={2} left="center">
        Click on the button or press the corresponding key.
      </Text>

      <Button left="center" bottom={3}>
        l:List Issues
      </Button>

      <Button left="center" bottom={1}>
        c:Create New Issue
      </Button>
    </Panel>
  )
}
```

This component displays instructions on navigating to other pages. It also has a reference to the panel, which enables us to have screen-specific event listeners. To add the event listeners add the following code before the layout:

```
  const history = useHistory()
  const match = useRouteMatch()
  // ...
  React.useEffect(() => {
    const goToNew = () => history.push(`${match.url}/new`)
    const goToList = () => history.push(`${match.url}/list`)

    ref.current.key("c", goToNew)
    ref.current.key("l", goToList)
    return () => {
      ref.current.unkey("c", goToNew)
      ref.current.unkey("l", goToList)
    }
  }, [])
```

Go back to `src/Issues/Issues.tsx` and remake it to look like this:

```
import React from "react"
import { Route, Switch, useRouteMatch } from "react-router"
import { IssuesMain } from "./IssuesMain"

const NewIssue = () => <>New Issue</>
const IssuesList = () => <>Issues List</>

export const Issues = () => {
  const match = useRouteMatch()

  return (
    <Switch>
      <Route exact path={match.path} component={IssuesMain} />
      <Route path={`${match.path}/new`} component={NewIssue} />
      <Route path={`${match.path}/list`} component={IssuesList} />
    </Switch>
  )
}
```

Getting the list of issues

Create a new file called `src/Issues/IssuesList.tsx` and start by adding imports:

```
import React, { useRef } from "react"
import { Panel } from "../shared/Panel"
import { useEffect } from "react"
import open from "open"
import { useQuery, gql } from "@apollo/client"
import { List } from "../shared/List"
import { Text } from "../shared/Text"
```

Now let's define a query:

```
const LIST_ISSUES = gql`
  query listIssues {
    viewer {
      issues(first: 100) {
        nodes {
          title
          url
        }
      }
    }
  }
`
```

Run the code generator to get types for the new query:

```
yarn run apollo codegen:generate\
  --localSchemaFile=graphql-schema.json\
  --target=typescript\
  --tagName=gql\
  --addTypename\
  --globalTypesFile=src/types/graphql-global-types.ts\
  types
```

After you have the types, you can import them in the `src/Issues/IssuesList.tsx` file:

```
import { listIssues } from "./types/listIssues"
```

Now define the actual component:

```
export const IssuesList = () => {
  const listRef = useRef<any>()
  const { loading, error, data } = useQuery<listIssues>(LIST_ISSUES)
  const issues = data?.viewer.issues.nodes
  // ...
  return (
    <Panel height={10} top="25%" left="center">
      <blessed-text
        left="center"
        bg="white"
        fg="black"
        content="List Issues"
      />
      <List
        ref={listRef}
        top={2}
        onAction={(el) =>
          open(
            issues?.find((issue) => issue?.title === el.content)
              ?.url || ""
```

```
          )
        }
        items={issues?.map((issue) => issue?.title || "") || []}
      />
    </Panel>
  )
}
```

Here we call `useQuery()` to get data, just like we did to get the list of repositories. Then we pass the `issues` array to the `List` component.

Define a `useEffect` that will trigger the `focus` method on the list element when the component is mounted:

```
useEffect(() => {
  listRef?.current?.focus()
}, [data])
```

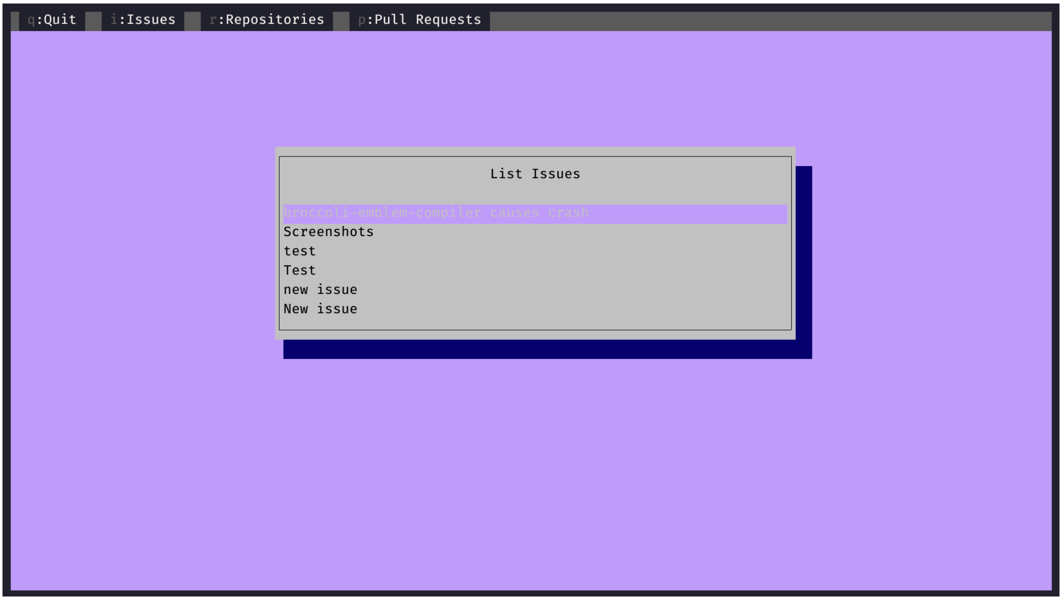
Define the early returns for the `loading` and `error` states:

```
  if (loading) {
    return (
      <Panel height={10} top="25%" left="center">
        <Text left="center">Loading...</Text>
      </Panel>
    )
  }

  if (error) {
    return <>Error: {JSON.stringify(error)}</>
  }
```

Open the `src/Issues/Issues.ts` and remake it to use the real `IssuesList` component:

```
import React from "react"
import { Route, Switch, useRouteMatch } from "react-router"
import { IssuesMain } from "./IssuesMain"
import { IssuesList } from "./IssuesList"

const NewIssue = () => <>New Issue</>

export const Issues = () => {
  const match = useRouteMatch()

  return (
    <Switch>
      <Route exact path={match.path} component={IssuesMain} />
      <Route path={`${match.path}/new`} component={NewIssue} />
      <Route path={`${match.path}/list`} component={IssuesList} />
    </Switch>
  )
}
```

Launch the application and make sure you can get the list of issues:



List issues screen

You should also be able to open a selected issue in the browser.

Creating an issue

Create a new file, `src/Issues/NewIssue.tsx`, and add imports:

```
import React, { useState } from "react"
import { useApolloClient, useMutation, gql } from "@apollo/client"
import { Field } from "../shared/Field"
import { Form, FormValues } from "../shared/Form"
import { NewEntitySuccess } from "../shared/NewEntitySuccess"
import { NewEntityError } from "../shared/NewEntityError"
import { Panel } from "../shared/Panel"
import { Button } from "../shared/Button"
import { Text } from "../shared/Text"
```

Now let's define the mutation:

```
const CREATE_ISSUE = gql`
  mutation createNewIssue(
    $title: String!
    $body: String
    $repository: ID!
  ) {
    createIssue(
      input: { title: $title, body: $body, repositoryId: $repository }
    ) {
      issue {
        title
        url
      }
    }
  }
`
```

This mutation accepts the repository id that we defined in one of the previous sections.

Generate the types for the mutation

Run the code generator to get types for this query:

```
yarn run apollo codegen:generate\
  --localSchemaFile=graphql-schema.json\
  --target=typescript\
        --tagName=gql\
        --addTypename\
        --globalTypesFile=src/types/graphql-global-types.ts\
        types
```

Import the generated types, along the `GET_REPOSITORY` query and types:

```
import {
  createNewIssue,
  createNewIssueVariables,
  createNewIssue_createIssue_issue
} from "./types/createNewIssue"
import { GET_REPOSITORY } from "../queries/getRepository"
import {
  getRepository,
  getRepositoryVariables
} from "../queries/types/getRepository"
```

Define the component

Now we can define the `NewIssue` component itself:

```
export const NewIssue = () => {
  // ...
  const [createIssue] = useMutation<
    createNewIssue,
    createNewIssueVariables
  >(CREATE_ISSUE)
  // ...
  const onSubmit = async (values: FormValues) => {
    const [repo, title, body] = values.textbox
    const [owner, name] = repo.split("/")
  // ...
  }
  // ...
}
```

The `NewIssue` component will have the `onSubmit()` handler that will get input values from the form. In this component it will be significantly more complex than a similar function in the `NewRepository` component. We'll have to tackle it step by step. Before we can work on this function let's define the component layout.

Just like in the `NewRepository` component we'll need to handle the success and error states. Define them using the `useState` hook:

```
const [error, setError] = useState<Error | null>()
const [issue, setIssue] =
  useState<createNewIssue_createIssue_issue | null>()
```

Define the layout

Define the layout of the `NewIssue` component:

```
return (
  <Panel top="25%" left="center" height={12}>
    <Text left="center">New Issue</Text>
    <Form onSubmit={onSubmit}>
      {(triggerSubmit) => {
        return (
          <>
            <Field
              top={0}
              label="Repo: "
              onSubmit={triggerSubmit}
            />
            <Field
              top={1}
              label="Title: "
              onSubmit={triggerSubmit}
            />
            <Field
              top={2}
              label="Body: "
              onSubmit={triggerSubmit}
            />
          </>
```

```
      )
    }}
  </Form>
  <Text left="center" bottom={3}>
    Tab: Next Field
  </Text>
  <Button left="center" bottom={1} onPress={onSubmit}>
    Enter: Submit
  </Button>
</Panel>
)
```

It will contain a form with 3 input fields:

- *Repository name*: we use this value to get the repository ID. When creating a new issue, repository ID is a mandatory field.
- *Issue title*: this is also a mandatory field.
- *Issue description*: an optional field that you can use to provide additional information about the new issue.

Create a new issue on form submit

Let's get back to the `onSubmit` function. We've implemented a similar function in the `NewRepository` component. This time we will have to run a query to get the repository ID before we can run the `createIssue`.

Previously we've only used queries through the hooks, but now we'll need to run the query inside of a callback function. It is doable using the Apollo `client` reference.

Get a reference to the Apollo client using the `useApolloClient()` hook. Add this code somewhere in beginning of the component body:

```
const client = useApolloClient()
```

Using the `client` we can perform the query directly. Add the following code to the `onSubmit()` handler:

```
    if (!owner || !name) {
      setError(
        new Error(
          "Repository name should have <owner>/<name> format."
        )
      )
      return
    }

    const { data } = await client.query<
      getRepository,
      getRepositoryVariables
    >({
      query: GET_REPOSITORY,
      variables: {
        owner,
        name
      }
    })
```

Here we make sure that the `owner` and the `name` fields are not empty. If they are empty we'll display an error message. Then we manually perform a query to get the ID of the repository by its name.

Now we might have the repository ID, but we need to verify that. Add the following check:

```
if (!data || !data.repository) {
  return
}
```

If we don't get the `repository` field in the response, we just return from the callback. Otherwise we can continue.

Now we want to perform the mutation:

```
    try {
      const result = await createIssue({
        variables: {
          title,
          body,
          repository: data.repository.id
        }
      })

      setIssue(result.data?.createIssue?.issue)
    } catch (e) {
      setError(e)
    }
```

We wrap the call in a try/catch block to handle errors. In any case we store the result or the error in a designated state.

Render the success and error results

Now we want to check if we have `issue` in the state, and if so, render the success screen. Add the following code right before the layout:

```
if (issue) {
  return (
    <NewEntitySuccess
      title="New issue created"
      url={issue.url}
      onClose={() => setIssue(null)}
    />
  )
}
```

If we get an error, we'll render the error message. Add an early return block for this case:

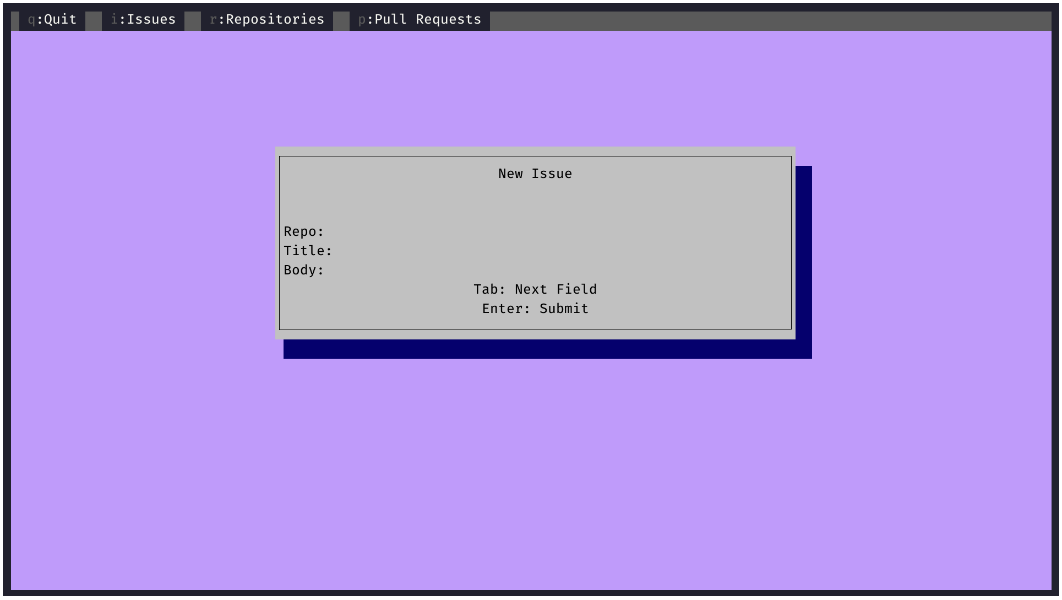
```
if (error) {
  return (
    <NewEntityError error={error} onClose={() => setError(null)} />
  )
}
```

Render the `NewIssue` component

Then go to `src/Issues/Issues.tsx` and and remake it to use the real `NewIssue` component:

```
import React from "react"
import { Route, Switch, useRouteMatch } from "react-router"
import { IssuesMain } from "./IssuesMain"
import { IssuesList } from "./IssuesList"
import { NewIssue } from "./NewIssue"

export const Issues = () => {
  const match = useRouteMatch()

  return (
    <Switch>
      <Route exact path={match.path} component={IssuesMain} />
      <Route path={`${match.path}/new`} component={NewIssue} />
      <Route path={`${match.path}/list`} component={IssuesList} />
    </Switch>
  )
}
```

Now launch the application and make sure everything works:



New Issue screen

Working with GitHub pull requests

Pull requests are very similar to issues as they are also bound to specific repositories. In this section we'll define the routing.

Define the main component for pull requests. Create a new file, `src/PullRequests/PullRequests` and add imports:

```
import React, { useEffect, useCallback } from "react"
import { useHistory, useRouteMatch } from "react-router"
import { useRef } from "react"
import { Panel } from "../shared/Panel"
import { debounce } from "../utils/debounce"
import { Button } from "../shared/Button"
import { Text } from "../shared/Text"
```

Then define the actual component with the following layout:

```
export const PullRequestsMain = () => {
  const history = useHistory()
  const match = useRouteMatch()
  const ref = useRef<any>()
  // ...
  return (
    <Panel ref={ref} height={11} top="25%" left="center">
      <Text left="center">Pull Requests</Text>
      <Text top={2} left="center">
        Click on the button or press the corresponding key.
      </Text>

      <Button left="center" bottom={3} onPress={goToList}>
        l:List Pull Requests
      </Button>

      <Button left="center" bottom={1} onPress={goToNew}>
        c:Create new Pull Request
      </Button>
    </Panel>
  )
}
```

Here we display instructions on navigating to other pages. We also get a reference to the panel, enabling us to have screen-specific event listeners. Define the `goToList` and `goToNew` handlers:

```
  const goToNew = useCallback(
    debounce(() => history.push(`${match.url}/new`), 100),
    []
  )

  const goToList = useCallback(
    debounce(() => history.push(`${match.url}/list`), 100),
    []
  )
```

Define the `useEffect` where we'll subscribe to keyboard events:

```
useEffect(() => {
  ref.current.key("c", goToNew)
  ref.current.key("l", goToList)
  return () => {
    ref.current.unkey("c", goToNew)
    ref.current.unkey("l", goToList)
  }
}, [])
```

Don't forget to unsubscribe in the cleanup function.

Define the routing

Open the `src/PullRequests/PullRequests.tsx`, and add the routing:

```
import React from "react"
import { Route, Switch, useRouteMatch } from "react-router"
import { PullRequestsMain } from "./PullRequestsMain"

const NewPullRequest = () => <>New PullRequest</>
const ListPullRequests = () => <>List</>

export const PullRequests = () => {
  const match = useRouteMatch()

  return (
    <Switch>
      <Route exact path={match.path} component={PullRequestsMain} />
      <Route path={`${match.path}/new`} component={NewPullRequest} />
      <Route
        path={`${match.path}/list`}
        component={ListPullRequests}
      />
    </Switch>
  )
}
```

Getting the list of pull requests

Create a new file, `src/PullRequests/ListPullRequests.tsx`, with the following imports:

```
import React, { useRef } from "react"
import { Panel } from "../shared/Panel"
import { useEffect } from "react"
import open from "open"
import { useQuery, gql } from "@apollo/client"
import { List } from "../shared/List"
import { Text } from "../shared/Text"
```

Next, define a query:

```
const LIST_PULL_REQUESTS = gql`
  query listPullRequests {
    viewer {
      pullRequests(first: 100) {
        nodes {
          title
          url
        }
      }
    }
  }
`
```

Run the code generator to get types:

```
yarn run apollo codegen:generate\
  --localSchemaFile=graphql-schema.json\
        --target=typescript --tagName=gql\
        --addTypename\
        --globalTypesFile=src/types/graphql-global-types.ts\
        types
```

Import the generated type and define the `ListPullRequests` component:

```
import { listPullRequests } from "./types/listPullRequests"
  // ...
export const ListPullRequests = () => {
  const { loading, error, data } = useQuery<listPullRequests>(
    LIST_PULL_REQUESTS
  )
  // ...
}
```

Render the `ListPullRequests` component layout. First let's handle the loading and error states, add early returns for them:

```
  if (loading) {
    return (
      <Panel height={10} top="25%" left="center">
        <Text left="center">Loading...</Text>
      </Panel>
    )
  }

  if (error) {
    return <>Error: {JSON.stringify(error)}</>
  }
```

Then add the normal layout:

```
  const listRef = useRef<any>()
  // ...
  const pullRequests = data?.viewer.pullRequests.nodes
  // ...
  return (
    <Panel height={10} top="25%" left="center">
      <Text left="center">List Pull Requests</Text>

      <List
```

```
          ref={listRef}
          top={2}
          onAction={(el) =>
            open(
              pullRequests?.find(
                (pullRequest) => pullRequest?.title === el.content
              )?.url || ""
            )
          }
          items={
            pullRequests?.map(
              (pullRequest) => pullRequest?.title || ""
            ) || []
          }
        />
      </Panel>
  )
```

We create `listRef` and pass it to the `List` element. Define a `useEffect` that will trigger the `focus()` method on this ref when we get the data.

```
useEffect(() => {
  listRef.current?.focus()
}, [data])
```

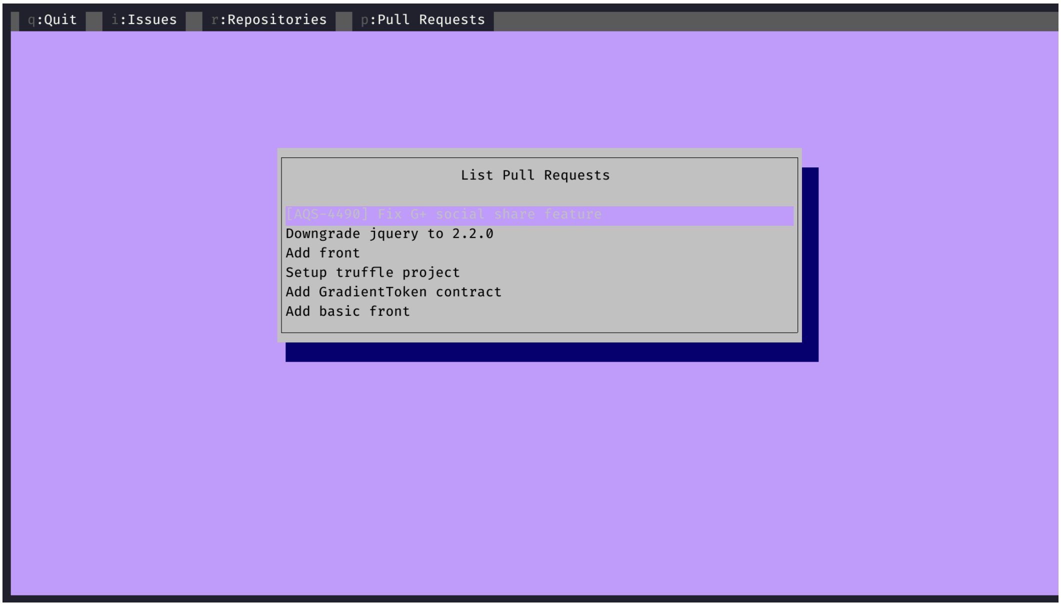
We don't do it on component mount because at that point component might be in the loading state.

Update the root pull requests component

Open the `src/PullRequests/PullRequests.tsx` and import the `ListPullRequests` component:

```
import React from "react"
import { Route, Switch, useRouteMatch } from "react-router"
import { PullRequestsMain } from "./PullRequestsMain"
import { ListPullRequests } from "./ListPullRequests"

const NewPullRequest = () => <>New PullRequest</>

export const PullRequests = () => {
  const match = useRouteMatch()

  return (
    <Switch>
      <Route exact path={match.path} component={PullRequestsMain} />
      <Route path={`${match.path}/new`} component={NewPullRequest} />
      <Route
        path={`${match.path}/list`}
        component={ListPullRequests}
      />
    </Switch>
  )
}
```

Run the application again and verify that you can see the list of pull requests and you can open the pull request in the browser.



List of pull requests

Creating a new pull request

Create a new file, `src/PullRequests/NewPullRequest.tsx`, and add the following imports:

```
import React, { useState } from "react"
import { useApolloClient, useMutation, gql } from "@apollo/client"
import { Field } from "../shared/Field"
import { Form, FormValues } from "../shared/Form"
import { NewEntitySuccess } from "../shared/NewEntitySuccess"
import { NewEntityError } from "../shared/NewEntityError"
import { Panel } from "../shared/Panel"
import { Text } from "../shared/Text"
import { Button } from "../shared/Button"
import {
  getRepository,
  getRepositoryVariables
```

```
} from "../queries/types/getRepository"
import { GET_REPOSITORY } from "../queries/getRepository"
```

Now define a GraphQL query to create a pull request:

```
const CREATE_PULL_REQUEST = gql`
  mutation createNewPullRequest(
    $baseRefName: String!
    $headRefName: String!
    $body: String
    $title: String!
    $repositoryId: ID!
  ) {
    createPullRequest(
      input: {
        title: $title
        body: $body
        repositoryId: $repositoryId
        baseRefName: $baseRefName
        headRefName: $headRefName
      }
    ) {
      pullRequest {
        title
        url
      }
    }
  }
`
```

Run the code generator to generate types:

```
yarn run apollo codegen:generate\
  --localSchemaFile=graphql-schema.json\
        --target=typescript\
        --tagName=gql\
        --addTypename\
        --globalTypesFile=src/types/graphql-global-types.ts\
        types
```

Import the generated types:

```
import {
  createNewPullRequest,
  createNewPullRequestVariables,
  createNewPullRequest_createPullRequest_pullRequest
} from "./types/createNewPullRequest"
```

Now define the `NewPullRequest` component:

```
export const NewPullRequest = () => {
  // ...
  return (
    <Panel top="25%" left="center" height={14}>
      <Text left="center">New Pull Request</Text>
  // ...
    </Panel>
  )
}
```

For now we'll just render a `Panel` with a `Title`. Later we'll add a form that will collect the data for the pull request.

Let's start with the component state where we'll store the error and the created pull request:

```
const [error, setError] = useState<Error | null>()
const [pullRequest, setPullRequest] =
  useState<createNewPullRequest_createPullRequest_pullRequest | null>()
```

After that define the `creatPullRequest` mutation:

```
const [createPullRequest] = useMutation<
  createNewPullRequest,
  createNewPullRequestVariables
>(CREATE_PULL_REQUEST)
```

In this component we'll need to query the repository id before we can create a new pull request. We'll perform the query inside of the form submit callback, so we'll need the apollo client instance. Define it:

```
const client = useApolloClient()
```

Now define the `onSubmit` function that we'll pass to the form:

```
const onSubmit = async (values: FormValues) => {
// ...
}
```

Inside of this function define a `try/catch` block. If at any point during the process of creating a new pull request we'll get an error - we'll store the it in the component state.

```
  try {
// ...
  } catch (e) {
    setError(e)
  }
```

Now inside the try block we get the values from the form and make sure that we can get the `owner` and the `name` of the repository:

```
      const [repo, title, body, baseRefName, headRefName] =
        values.textbox
      const [owner, name] = repo.split("/")

      if (!owner || !name) {
        setError(
          new Error(
            "Repository name should have <owner>/<name> format."
          )
        )
        return
      }
```

Here we get following values to create a pull request:

- repository name in `owner/repo-name` format. We'll use it to get the repository id.
- the title and the body of the pull request
- base reference name (usually the `main` branch)
- head reference name (usually a feature branch)

After we have those values we can query the repository id:

```
const { data } = await client.query<
  getRepository,
  getRepositoryVariables
>({
  query: GET_REPOSITORY,
  variables: {
    owner,
    name
  }
})
```

We're almost there, now create the pull request. Call the `createPullRequest` mutation:

```
const result = await createPullRequest({
  variables: {
    title,
    body,
    repositoryId: data.repository.id,
    baseRefName,
    headRefName
  }
})
```

Store the `pullRequest` from in the component state:

```
setPullRequest(result.data?.createPullRequest?.pullRequest)
```

Let's show the success and error screens. Add this code right after `onSubmit` handler:

```
  if (error) {
    return (
      <NewEntityError error={error} onClose={() => setError(null)} />
    )
  }

  if (pullRequest) {
    return (
      <NewEntitySuccess
        title="New pull request created"
        url={pullRequest.url}
        onClose={() => setPullRequest(null)}
      />
    )
  }
```

Define the form

Now add the form to the component:

```
<Form onSubmit={onSubmit}>
  {(triggerSubmit) => {
    return (
      <>
        <Field
          top={0}
          label="Repo: "
          onSubmit={triggerSubmit}
        />
        <Field
          top={1}
          label="Title: "
          onSubmit={triggerSubmit}
        />
        <Field
          top={2}
          label="Body: "
          onSubmit={triggerSubmit}
        />
        <Field
          top={3}
          label="Base: "
          onSubmit={triggerSubmit}
        />
        <Field
          top={4}
          label="Head: "
          onSubmit={triggerSubmit}
        />
      </>
    )
  }}
</Form>
```

We've already discussed what values do we need to create a pull request. Here we defined the form fields to get them.

Add the navigation instructions

Add the instructions on how to select the field and how to submit the form. Add this block right after the `Form` element in the layout:

```
<Text left="center" bottom={3}>
  Tab: Next Field
</Text>
<Button left="center" bottom={1} onPress={onSubmit}>
  Enter: Submit
</Button>
```

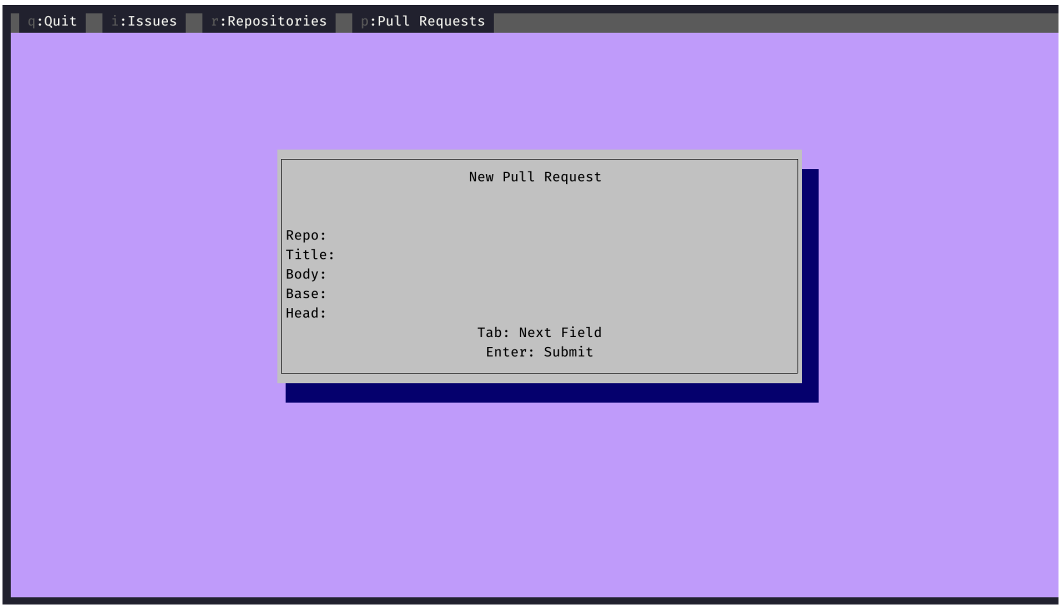
Use the component

Then open `src/PullRequests/PullRequests.tsx` and use the real `NewPullRequest` component instead of a stub:

```
import React from "react"
import { Route, Switch, useRouteMatch } from "react-router"
import { PullRequestsMain } from "./PullRequestsMain"
import { ListPullRequests } from "./ListPullRequests"
import { NewPullRequest } from "./NewPullRequest"

export const PullRequests = () => {
  const match = useRouteMatch()

  return (
    <Switch>
      <Route exact path={match.path} component={PullRequestsMain} />
      <Route path={`${match.path}/new`} component={NewPullRequest} />
      <Route
        path={`${match.path}/list`}
        component={ListPullRequests}
```

```
      />
    </Switch>
  )
}
```

Run the application and make sure you can create pull requests like this:



Creating a pull request

Summary

In this chapter, we've learned to combine GraphQL with TypeScript. It is a great duo because GraphQL allows us to preserve type information while communicating with the backend.

A great advantage of using GraphQL on your backend is that you can provide the full schema definition to your clients, just like GitHub does.

Another great benefit of using GraphQL is that you can generate types from a GraphQL schema. It makes using queries and mutations super easy, as the editor can provide code completion suggestions based on the actual schema.

I hope you liked working on this fun project, and good luck in your next endeavors!

Appendix

Changelog

Revision r12 (31-12-2021)

- Added a code example for each chapter
- Fixed numerous typos
- Fixed the code examples

Revision r11 (26-03-2021)

- Updated the react-dnd package in the first chapter
- Introduced Immer for state management in the first chapter
- Fixed typos and missing links
- Replaced interfaces with types
- Added a section about optimizing images in the fifth chapter

Revision r10 (03-03-2021)

- Improved HOC explanation in the first chapter
- Expanded Class and Function components explanations

Revision r9 (26-02-2021)

- Fixed missing code issues in the first chapter
- Fixed some confusing wording

Revision r8 (17-02-2021)

- Fixed grammatical errors and typos

Revision r7 (01-12-2020)

- Fixed typos in the first chapter and the book intro
- Added a link to `react-scripts/package.json` on GitHub

Revision r6 (01-12-2020)

- Fixed the order of steps in the Testing chapter

Revision r5 (10-11-2020)

- Updated the first chapter to the last version of create-react-app
- Added a requested feature in trello-clone to submit new items by pressing "Enter"
- Made all the data updates in the trello-clone immutable
- Fixed typos and code errors

Revision r4 (26-08-2020)

- Added GraphQL chapter
- Fixed typos and code errors
- Updated react-dnd packages

Revision 3p (07-30-2020)

- Added Redux with Typescript chapter
- Fixed various typos and grammar

Revision 2p (06-08-2020)

- Added information on SSR with Next.js
- Fixed various typos and grammar

Revision 1p (05-20-2020)

First “Early Draft” Release