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

© 2020 Fullstack.io

All rights reserved. No portion of the book manuscript may be reproduced, stored in a retrieval system, or transmitted in any form or by any means beyond the number of purchased copies, except for a single backup or archival copy. The code may be used freely in your projects, commercial or otherwise.

The authors and publisher have taken care in preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damagers in connection with or arising out of the use of the information or programs container herein.

Published by \newline

Contents

Book Revision

Revision r11 - 2021-26-03

If you'd like to report any bugs or typos, join our Discord or email us below.

Join Our Discord Channel

If you'd like to get help, help others, and hang out with other readers of this book, come join our Discord channel:

https://newline.co/discord/[1]

Bug Reports

If you'd like to report any bugs, typos, or suggestions just email us at: **us@fullstack.io**.

Be notified of updates via Twitter

If you'd like to be notified of updates to the book on Twitter, follow us at @fullstackio[2].

We'd love to hear from you!

Did you like the book? Did you find it helpful? We'd love to add your face to our list of testimonials on the website! Email us at: us@fullstack.io[3].

[1] https://newline.co/discord/
[2] https://twitter.com/fullstackio
[3] mailto:us@fullstack.io

Introduction

Welcome to *Fullstack React with TypeScript*! React and TypeScript are a powerful combination that can prevent bugs and help you (and your team) ship products faster. But understanding idiomatic React patterns and getting the typings set up isn't always straightforward.

This practical, hands-on book is a guide that will have you (and your team) writing React apps with TypeScript (and hooks) in no time.

This book consists of several sections. Each section covers one practical case of using TypeScript with React.

Your First React and TypeScript Application: ***Building Trello with Drag and Drop***: Here you will learn how to bootstrap a React TypeScript application and all the basics of using React with TypeScript. We will build a kanban board application like Trello that will store its state on backend.

Testing React With TypeScript: ***Testing a Digital Goods Store***: In this section you will set up your testing environment and learn how to test your application. We will take an online store application and cover it with tests.

Patterns in React TypeScript Applications: ***Making Music with React***: Here we cover Higher Order Components (HOCs) and render props React patterns. We show when are they useful and how to use them with TypeScript. In this section we will build a virtual piano that supports different sound sets.

Next.js and Static Site Generation: ***Building a Medium-like Blog***: React can be rendered server-side. It allows us to create multi-page interactive websites. In this section we cover the basics of server-side generation with React and then we build an advanced application using Next.js framework. The example application will be a blogging platform (like Medium).

State Management With Redux and TypeScript Some React applications are so complex that they require use of some external state management library. Redux is a solid choice in this case. It is worth learning how to use it with TypeScript. In this section we will build a drawing application with undo/redo support. It will also let you save your drawings on backend.

VI GraphQL With React And TypeScript. GraphQL is a query language that allows us to create flexible APIs. Facebook, Github, Twitter and many other companies provide GraphQL APIs. TypeScript works pretty well with GraphQL. In this section we will build a Github issue viewer.

We recommend you read this book in linear order, from start to finish. The sections are arranged from basic topics to more complex ones. Most sections assume that you are familiar with topics explained in previous sections.

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 skill (you know how to run a command in 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[4].

[4] mailto:us@fullstack.io

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:

```
$ 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[5] as a terminal application.

We recommend installing node using nvm-windows[6]. 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[7]:

[5]https://cmder.net/
[6]https://github.com/coreybutler/nvm-windows
[7]https://github.com/nvm-sh/nvm

```
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[8].

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[9]:

```
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[10].

Code Blocks And Context

Code Block Numbering

In this book, we build example applications in steps. Every time we achieve a runnable state we put it in a separate `step` folder.

---

[8] https://github.com/nvm-sh/nvm#troubleshooting-on-macos
[9] https://github.com/nvm-sh/nvm
[10] https://github.com/nvm-sh/nvm#troubleshooting-on-linux

```
01-first-app/
├── step1
├── step2
├── step3
... // other steps
```

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

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

```
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[11]! 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[12].

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

- What revision of the book are you referring to?

[11] mailto:fullstack-react-typescript@newline.co
[12] mailto:fullstack-react-typescript@newline.co

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

What is TypeScript

> TypeScript is a typed superset of JavaScript that compiles to plain JavaScript - typescriptlang.org[13].

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:

[13] https://typescriptlang.org

```
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:

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

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

Now if we try to call our function with the wrong type, the TypeScript typechecker will give us an error.

TypeScript typechecker can tell if we have an error in our code just by analyzing the syntax. That means that you won't have to run your program. Most code editors support TypeScript so the error will be immediately highlighted when you try to call the function with the wrong value type.

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 type `User` and specify it as a type of our function `user` argument:

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

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

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

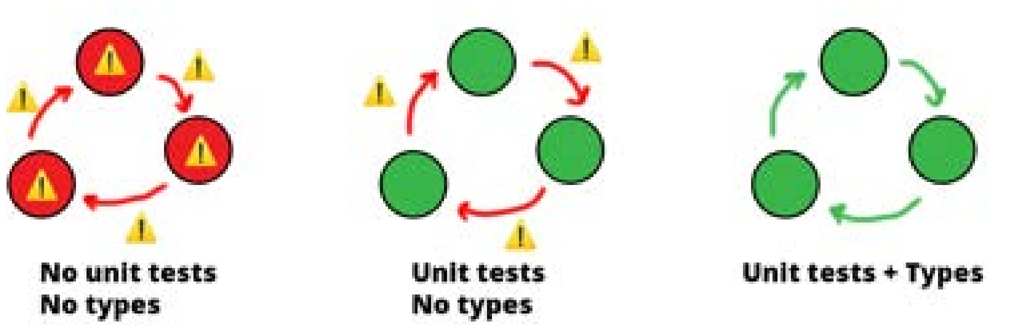
```
greet({firstName: "Maksim", lastName: "Ivanov"}) // Returns "Hello Maks\
im Ivanov!"
```

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[14] 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[15] 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.

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

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.

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 gets only correct inputs.

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 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 the 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[16]. You will get `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 to 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 it is convenient because an array that is defined as having `number` or `string` elements can actually have only strings.

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

```
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.

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 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
- styling components
- using external libraries
- making network requests

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[17] or Cmder[18]. If you are more experienced you can use Windows Subsystem for Linux[19].

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[20] on Mac or Linux to switch Node versions. For Windows there is nvm-windows[21].

You also need to know how to use node package managers. In this chapter's examples, I will use Yarn[22]. You can use npm[23] if you want.

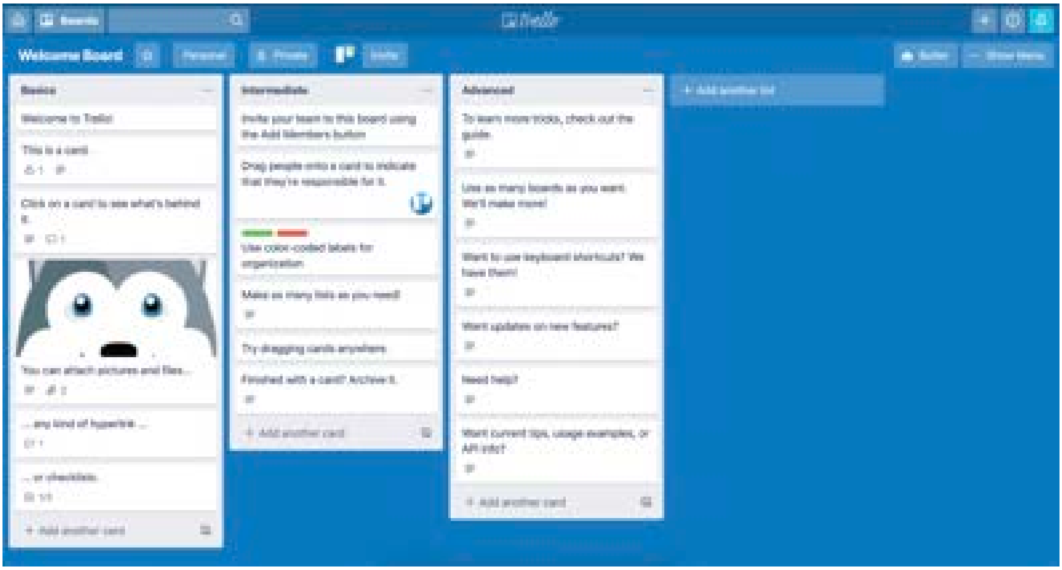
All the examples for this chapter contain `yarn.lock` files. Remove them if you want to use npm to install dependencies.

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[24] to refresh your knowledge.

What Are We Building?

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

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



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.

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`.

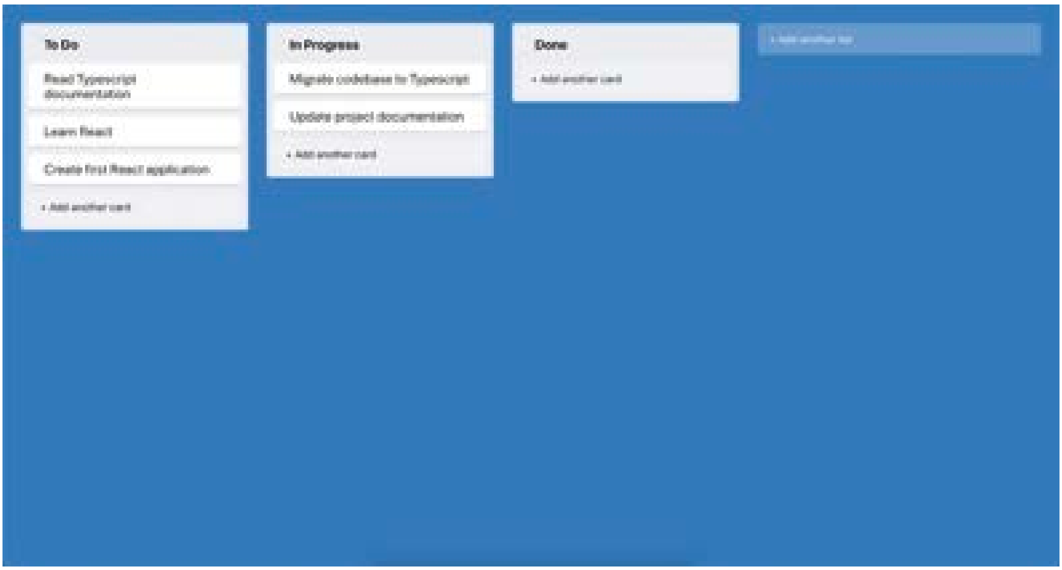
Unzip the archive and `cd` to the app folder.

```
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

Now let's go through the steps needed to create your version.

In this chapter, we will use an automatic CLI tool to generate 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[25]. 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[26] or Parcel[27].

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`.

[25] https://babeljs.io/
[26] https://webpack.js.org/
[27] 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[28].

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.

Now, `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
```

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

```
├── node_modules
│   └── ...
├── 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.[29]

Now let's open the `package.json` file and check what packages are installed with `create-react-app`:

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

01-first-app/step1/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": "^17.0.3",
    "@types/react-dom": "^17.0.2",
    "react": "^17.0.1",
    "react-dom": "^17.0.1",
    "react-scripts": "4.0.3",
    "typescript": "^4.2.3",
    "web-vitals": "^0.2.4"
  },
```

Now, 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 we know that it's not meant to return any value.

Now we could make this file into a package and publish it through the npm registry. And this is what all those `@types/*` packages are.

It is a convention that all the types-packages are published under the `@types` namespace. Those packages are provided by the DefinitelyTyped[30] 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.

[30] http://definitelytyped.org/

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

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[31].

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[32]. 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.[33]

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.

[31] https://create-react-app.dev/docs/using-the-public-folder/
[32] https://web.dev/progressive-web-apps/
[33] https://developer.mozilla.org/en-US/docs/Web/Manifest

Read more about `robots.txt` on the robotstxt website.[34]

src Folder

Now let's take a look at the `src` folder. Files in this folder will be 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.[35]

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:

[34] https://www.robotstxt.org/robotstxt.html
[35] https://reactjs.org/docs/introducing-jsx.html

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

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

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

You can read about it in the TypeScript Documentation[36].

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:

01-first-app/step1/src/index.tsx

```
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\
```

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

```
als
reportWebVitals();
```

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

01-first-app/step1/src/index.tsx

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

Babel will transpile `<App />` to `React.createElement(App, null)`. It means that we are implicitly referencing React in this file, so we need to have it imported.

Then we import `ReactDOM`. We'll 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.

Next, 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 need 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[37].

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[38] for the updated list of the `StrictMode` functionality.

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

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.

> Currently, in JavaScript apps generated with `create-react-app`, you don't need to import React at all. Read more here[39].

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[40]. React exports an object full of different classes and functions:

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

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

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.

```
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[41].

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:

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

01-first-app/step1/src/react-app-env.d.ts

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

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

Read more about "triple slash directives" in the TypeScript documentation[42].

By default, this would reference the file `./node_modules/react-scripts/index.d.ts`, but `reacts-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[43].

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:

[42] https://www.typescriptlang.org/docs/handbook/triple-slash-directives.html#-reference-types-
[43] 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[44].

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

[44]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:

01-first-app/step1/src/App.tsx

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

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

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

01-first-app/step2/src/index.tsx

```
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')
);
```

Note that we also changed the default `App` export to named, so now inside the `index.tsx` file we need to use the curly brackets.

K> 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

We need some styles to apply to the whole application.

Let's edit `src/index.css` and add some global CSS rules.

01-first-app/step2/src/index.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 an `index.css` file. It contains styles that we need to be applied globally.

To start using CSS rules from such a file you need to import it. We will import `index.css` in `index.tsx` file.

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

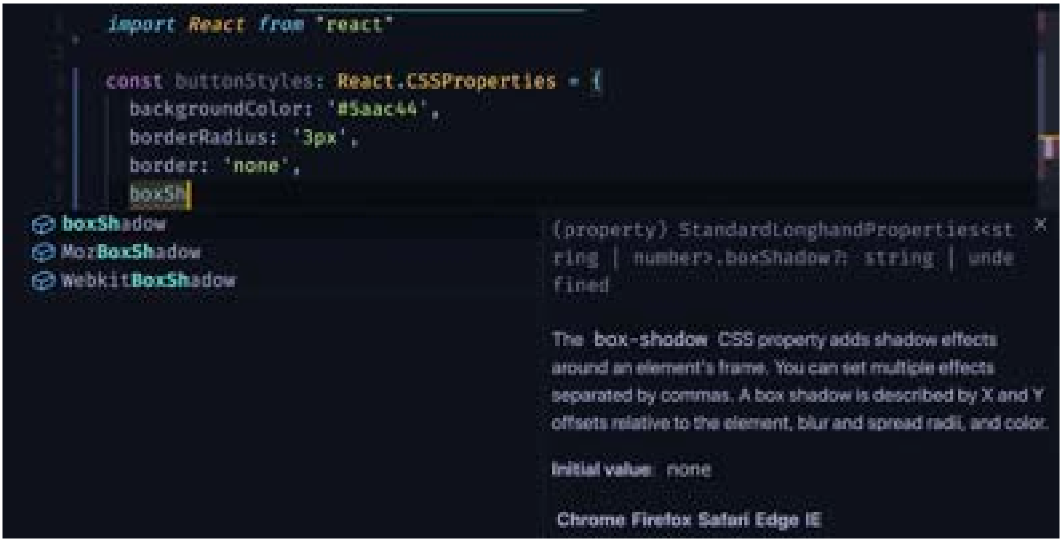
Another option is to pass an object with styling rules through the `style` property. You can declare the object inline, then you won't need to specify a type for it:

```
<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

Keep in mind that we aren't using real CSS attribute names. Because of how React works with the `styles` prop we have to provide them in camel case form. For example `background-color` becomes `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[45].

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

[45]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.

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.

First we need to install the `styled-components` library:

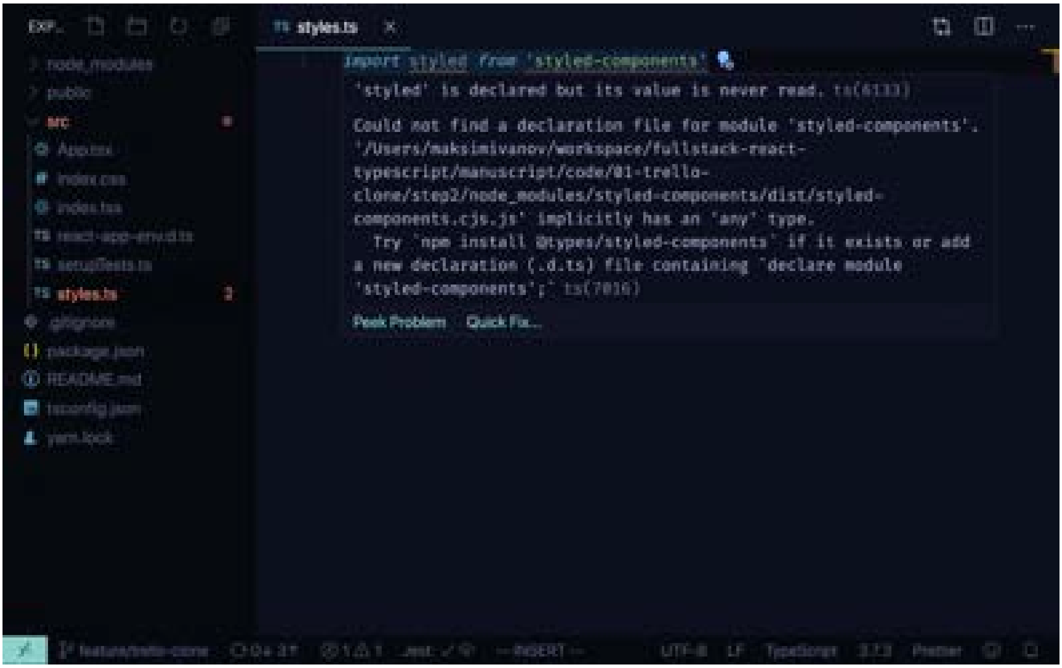
```
yarn add styled-components@^5.2.1
```

After the library is installed we can try to define our first styled component.

Create the `src/styles.ts` file and try to 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`.

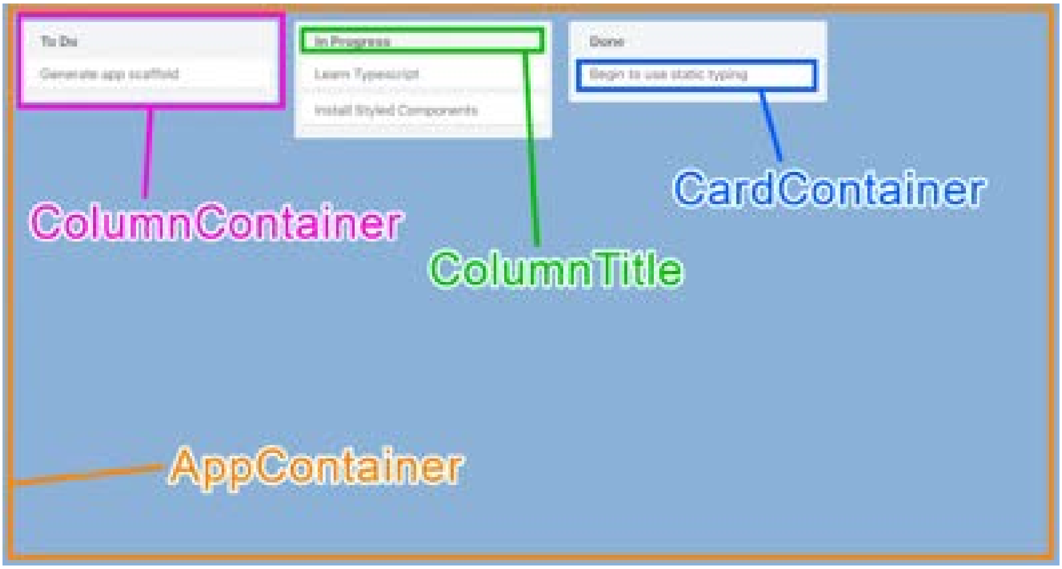
Install the missing types:

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

Now we are ready to define our styled-components.

Prepare Styled Components

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



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 need 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.

01-first-app/step2/src/styles.ts

```
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.

Go to `src/App.tsx` and import `AppContainer`:

01-first-app/step2/src/App.tsx

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

Now use it in `App` layout:

01-first-app/step2/src/App.tsx

```
export const App = () => {
  return (
    <AppContainer>
      Columns will go here
    </AppContainer>
  )
}
```

Styles For Columns

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

01-first-app/step2/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`:

01-first-app/step2/src/styles.ts

```
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.

01-first-app/step2/src/styles.ts

```
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`.

Create Columns and Cards. How to Define React Components

Now that we have our styles ready we can begin working on actual components for our cards and columns.

In this section, I'm not going to explain how React components work. If you need to pick this knowledge up, refer to the React documentation[46]. Make sure you know what props and state are, and how lifecycle events work.

> In the following section I'm going to show examples from a separate mini project. You can find it inside `code/01-trello/class-components` folder.

Now let's see what is different when you define React components in TypeScript.

How to Define Class Components. When you define a class component, you need to provide types for its props and state. You do this by using special triangle brackets syntax:

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

01-first-app/class-components/src/Counter.tsx

```
type CounterState = {
  count: number
}

export class Counter extends React.Component<{}, CounterState> {
  state: CounterState = {
    count: 0
  }

  private increment = () => {
  // ...
  }

  private decrement = () => {
  // ...
  }

  render() {
    return (
      <>
        <p>
          Count: {this.state.count}
        </p>
        <button onClick={this.increment}>Increment</button>
        <button onClick={this.decrement}>Decrement</button>
      </>
    )
  }
}
```

`React.Component` is a generic type that accepts *type variables* for props and state.

Let's inspect the type of `React.Component`:

```
class React.Component<P = {}, S = {}, SS = any>
```

In VSCode you can get the type information by hovering the item. You can also trigger it using the `Show Hover` command from the command palette or using the `Ctrl+K Ctrl+I` (`⌘K ⌘I` for Mac users). If you use VSCodeVim you can type `gh` in normal mode.

Here `P` stands for **P**rops, `S` stands for **S**tate, and `SS` stands for **S**nap**S**hot. You can peek the type definition to see how the `SS` type is being used.

To check how the type was defined you can click the item with pressed `Ctrl` or `⌘` key or call the `Peek Type Definition` command from the command palette.

Try to peek the `React.Component` type definition and track down the `SS` type to find where is it going to be used. For example I found that it is used as a return value of the `getSnapshotBeforeUpdate` method.

To be honest I don't like to use single-letter names in my code. I would use `Props` instead of `P` and `State` instead of `S`. You can also use some convention to show that some types are in fact type variables. For example prefix them with `T`, so `Props` would be `TProps` and `State` would be `TState`.

All three type variables have default values, so we don't need to always specify them. If we won't have props and state we can define our component like this:

```
class SimpleComponent extends React.Component {
  render(){
    return null
  }
}
```

In this case TypeScript will know that both state and props types are `{}`.

In our `Counter` component we specified the type of the props to be an empty object, because we are not passing any props to our component.

> Try to pass a property to the `Counter` component. Open `code/01-trello/class-components/src/index.tsx` and pass a prop `foo="bar"`. You should see a TypeScript error.

Our `Counter` component needs to store the `count` value in its state. To be able to do this we need to define the shape of the `Counter` state.

There are two ways we can define the shape of an object. We can do it using type aliases and we can do it using interfaces.

For example here we defined the form of the state of our component as a type alias:

01-first-app/class-components/src/Counter.tsx

```
type CounterState = {
  count: number
}
```

By saying type alias I mean that we could just pass the shape of the state of our component directly, without giving it a name:

```
class SimpleComponent extends React.Component<{}, { count: number }> {
  // ...
}
```

This way the code would be harder to read so we've assigned the type `{ count: number }` an alias `CounterState`.

This is very similar to defining constants. But instead of assigning a value to a const, we assign a literal type to a type alias.

It is important to understand that types and values live it two different worlds. The syntax to define them can look similar, but they can not be used interchangeably:

```
type CounterState = {
  count: number
}
// Here we assign a type literal to a type alias

const counterState = {
  count: 0
}
// Here we assign an object literal to a constant

const foo = counterState // You can do this
const bar = CounterState // 'CounterState' only refers to a type, but i\
s being used as a value here.
```

We could also define the `CounterState` as an `interface`:

```
interface CounterState {
  count: number
}
```

With both interfaces and type aliases we limit the shape of the `Counter` state to an object with the field `count` of type `number`. Then what is the difference?

To be fair most of the time you can use types and interfaces interchangeably. In my opinion semantically interfaces are better suited to describe the API of a class, and type aliases fit better to describe the shape of the data.

It is important to note type checking works faster for interfaces[47]. TypeScript can detect property conflicts for them and also type relations between interfaces are cached. So if you need to optimise the type checking speed - use interfaces.

This being said I see the props that we pass to our components and the state that they hold as data, so throughout this book we will define components `props` and `state` as type aliases. It is my personal preference and if you don't agree - just use interfaces.

Defining Functional Components. In TypeScript, when you create a functional component, you don't have to provide types for it manually.

```
export const Example = () => {
  return <div>Functional component text</div>
}
```

Here we return a string wrapped into a `div` element, so TypeScript will automatically conclude that the return type of our function is `JSX.Element`.

If you want to be verbose, you can use `React.FC` or `React.FunctionalComponent` types.

```
export const Example: React.FC = () => {
  return <div>Functional component text</div>
}
```

The `React.FC` type is an alias to `React.FunctionalComponent` type, so it does not matter which one you use. You can verify this by checking the type definition of `React.FC`.

Previously you could also see `React.SFC` or `React.StatelessFunctionalComponent` but after the release of hooks, it's deprecated.

It is important to note that the `React.FC` type also defines the prop `children` for your component. Let's verify this. Open the `src/App.tsx` in your application, import the

[47] https://github.com/microsoft/TypeScript/wiki/Performance#preferring-interfaces-over-intersections

type `FC` from react, set it as the type of your component and try to get the `children` from the props:

01-first-app/step2/src/App.tsx

```
import { FC } from "react"
import { Column } from "./Column"
import { Card } from "./Card"
import { AppContainer } from "./styles"

export const App: FC = ({children}) => {
  return (
    <AppContainer>
      Columns will go here
    </AppContainer>
  )
}
```

Now if you check the type of the `children` prop you will see that its type is known:

```
children: React.ReactNode
```

You can also try to pass some element as a child to the `App` component inside the `src/index.tsx`:

01-first-app/step2/src/index.tsx

```
  <React.StrictMode>
    <App>
      <p>Hello I'm React Element</p>
    </App>
  </React.StrictMode>,
```

Now back in the `src/App.tsx` remove the `FC` type and the `children` prop from the `App` component. You will get a TypeScript error inside the `src/index.tsx` file.

We can use this to make it clear if the components accept `children`. So in the examples in this book we will set the component type to `React.FC` if the component renders `children` and specify the type of the props directly if it doesn't.

Now remove the paragraph element that you were passing to the `App` component and let's continue with the code.

Create Column Component

It's time to create our first functional component.

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

```
export const Column = () => {
  return <div>Column Title</div>
}
```

Update Column Layout

Now let's use this wrapper component in our `Column` layout:

01-first-app/step2/src/Column.tsx

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

export const Column = () => {
  return (
    <ColumnContainer>
      <ColumnTitle>Column Title</ColumnTitle>
    </ColumnContainer>
  )
}
```

We want to be able to provide the column title using `props`.

Let's see how to use `props` with functional components.

As I said before you can use a `type` or an `interface` to define the form of your `props` object. In a lot of cases, types and interfaces can be used interchangeably. We'll get to some differences later in this chapter.

Here let's define props as a `type`:

01-first-app/step2/src/Column.tsx

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

type ColumnProps = {
  text: string
}

export const Column = ({ text }: ColumnProps) => {
  return (
    <ColumnContainer>
      <ColumnTitle>{text}</ColumnTitle>
    </ColumnContainer>
  )
}
```

Here we define a `type` alias called `ColumnProps` and then specify it as the type of the first argument of our functional component.

Inside the `ColumnProps` type, we define a field `text` of type `string`. By default this field will be `required`, so you'll get a type error if you don't provide this prop to your component.

To make the prop optional you can add a question mark before the colon.

01-first-app/step2/src/Column.tsx

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

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

```
(property) ColumnProps.text?: string | undefined
```

We want the `text` prop to be `required`, so don't add the question mark.

Create The Card Component

After that's done we can start working on our `Card` component. Create a new file `src/Card.tsx`.

01-first-app/step2/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 Children Inside The Columns

Now we have a `Card` component and a `Column` component and we can render everything at once.

To do this we'll pass the `Card` components `children` to our `Column` components.

Go to `src/Column.tsx` and import the type `FC` from `react`:

01-first-app/step2/src/Column.tsx

```
import { FC } from "react"
```

Then modify the component:

01-first-app/step2/src/Column.tsx

```
export const Column: FC<ColumnProps> = ({ text, children }) => {
  return (
    <ColumnContainer>
      <ColumnTitle>{text}</ColumnTitle>
      {children}
    </ColumnContainer>
  )
}
```

Here we used the `React.FC` type to define the `children` prop on our component.

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

Or we could manually add `children?: React.ReactNode` to our `ColumnProps` type.

Here is the `React.PropsWithChildren` type definition:

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

Here the letter `P` is a *type argument*. When we used `React.PropsWithChildren` we passed our `ColumnProps` type to it. Then it was combined with another type using an ampersand.

As a result, we've got a new type that combines the fields of both source types. In TypeScript this is called a *type intersection*.

For example:

```
type ColumnProps = React.PropsWithChildren<{
  text: string
}>
// type ColumnProps = {
//   text: string;
// } & {
//   children?: React.ReactNode;
// }
//
// Which is the same as the following:
type ColumnProps = {
  text: string
  children?: React.ReactNode;
}>
```

Component For Adding New Items. State, Hooks, and Events

Before we move on to the next chapter where we'll add the business logic, let's create a component that will allow us to create new items.

AddItemComponent

This component will have two states. Initially, it will be a button that says "+ Add another task" 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.

Prepare Styled Components

Styles For The Button

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

01-first-app/step2/src/styles.ts

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

We'll use the `AddNewItemButton` 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:

01-first-app/step2/src/styles.ts

```
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`.

Styles For The Form

We are aiming to have a form styled like this:

Styled NewItemForm

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

01-first-app/step2/src/styles.ts

```
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:

01-first-app/step2/src/styles.ts

```
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:

01-first-app/step2/src/styles.ts

```
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 AddNewItem Component. Using State

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

01-first-app/step2/src/AddNewItem.tsx

```
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:

01-first-app/step2/src/AddNewItem.tsx

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

- `onAdd` is a callback function that will be called when we click the `Create item` 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:

01-first-app/step2/src/AddNewItem.tsx

```
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.

Now let's define the form that we'll show inside the condition block.

Create Input Form. Using Events

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

01-first-app/step2/src/NewItemForm.tsx

```
import { useState } from "react"
import { NewItemFormContainer, NewItemButton, NewItemInput } from "./st\
yles"
```

Define the `NewItemFormProps` type:

01-first-app/step2/src/NewItemForm.tsx

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

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

Now define the `NewItemForm` component:

01-first-app/step2/src/NewItemForm.tsx

```
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`:

01-first-app/step2/src/AddNewItem.tsx

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

Now let's add `NewItemForm` to `AddNewItem` component.

01-first-app/step2/src/AddNewItem.tsx

```
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>
  )
}
```

Use AddNewItem Component

Our `AddNewItem` component is now fully functional and we can add it to the application layout. For now, we won't create the new items, instead, we'll log messages to console.

Adding New Lists

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

01-first-app/step2/src/App.tsx

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

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

01-first-app/step2/src/App.tsx

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

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

Adding New Tasks

Now go to `src/Column.tsx`, import the component:

01-first-app/step2/src/Column.tsx

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

And update the `Column` layout:

01-first-app/step2/src/Column.tsx

```
export const Column: FC<ColumnProps> = ({ text, children }) => {
  return (
    <ColumnContainer>
      <ColumnTitle>{text}</ColumnTitle>
      {children}
      <AddNewItem
        toggleButtonText="+ Add another task"
        onAdd={console.log}
        dark
      />
    </ColumnContainer>
  )
}
```

Render Everything Together

Let's combine all the parts and render what we have so far. Go to `src/App.tsx` and make sure you have all the necessary imports:

01-first-app/step2/src/App.tsx

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

Now change the layout code to this:

01-first-app/step2/src/App.tsx

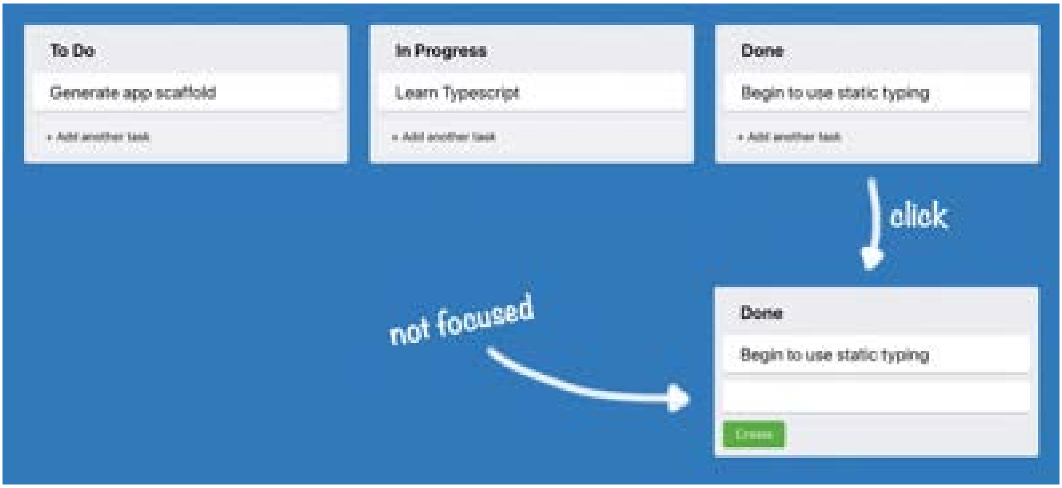
```
  return (
    <AppContainer>
      <Column text="To Do">
        <Card text="Generate app scaffold" />
      </Column>
      <Column text="In Progress">
        <Card text="Learn Typescript" />
      </Column>
      <Column text="Done">
        <Card text="Begin to use static typing" />
      </Column>
      <AddNewItem toggleButtonText="+ Add another list" onAdd={console.\
log} />
    </AppContainer>
  )
```

Let's launch the app and make sure it works.

Run `yarn start` and open the browser.

When you click the buttons you should see the new item forms.

There is one problem though; when you open the form, you have to make one more click to focus the input.



Input is not focused by default

Let's see how can we focus the input automatically.

Automatically Focus on Input Using Refs

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

01-first-app/step2/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[48] 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:

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[49]. It is assumed that if you pass `null` as the default

[48] https://github.com/DefinitelyTyped/DefinitelyTyped/blob/master/types/react/global.d.ts
[49] https://github.com/DefinitelyTyped/DefinitelyTyped/issues/31065#issuecomment-446425911

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.

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.

Now let's use our `useFocus` hook in the `NewItemForm` component. Go back to `src/NewItemForm.tsx` and import the hook:

01-first-app/step2/src/NewItemForm.tsx

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

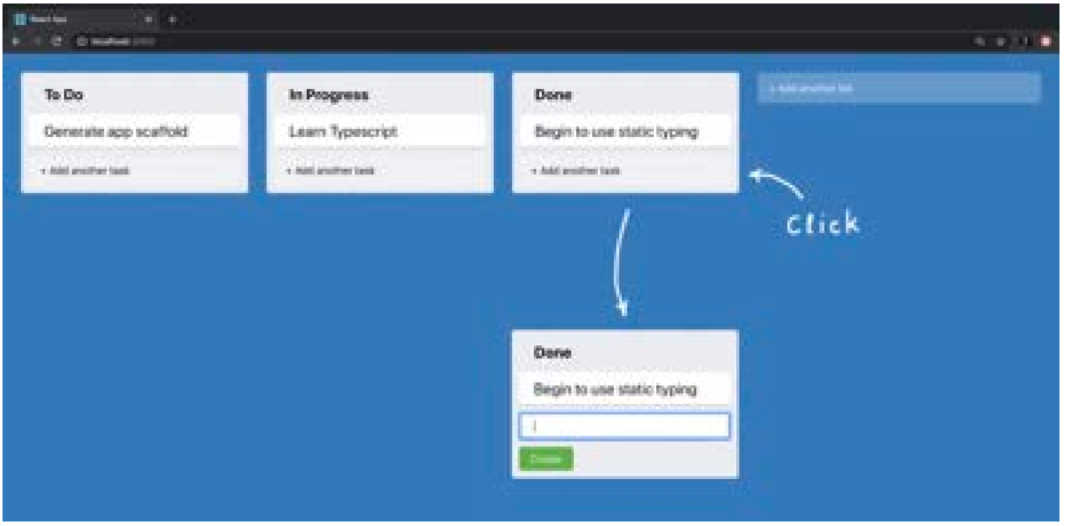
And then use it in the component code.

01-first-app/step2/src/NewItemForm.tsx

```
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>
  )
}
```

Here we pass 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

Requested Feature - Submit on Enter Press

Some readers requested 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. Let's implement it.

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:

01-first-app/step2/src/NewItemForm.tsx

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

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

01-first-app/step2/src/NewItemForm.tsx

```
<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[50] and the types for them in the React type definitions[51].

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/step2`.

[50] https://reactjs.org/docs/events.html
[51] https://github.com/DefinitelyTyped/DefinitelyTyped/blob/14d95eb0fe90f5e0579c49df136cccdfe89b2855/types/react/index.d.ts#L1211

Add Global State And Business Logic

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.

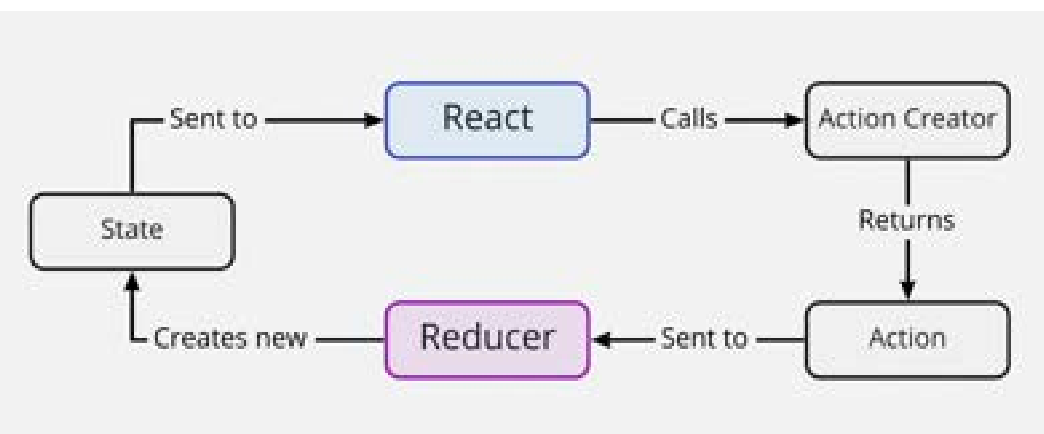
Before we jump into the action I will give a little primer on using `useReducer`.

> 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.

Using the useReducer

`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.

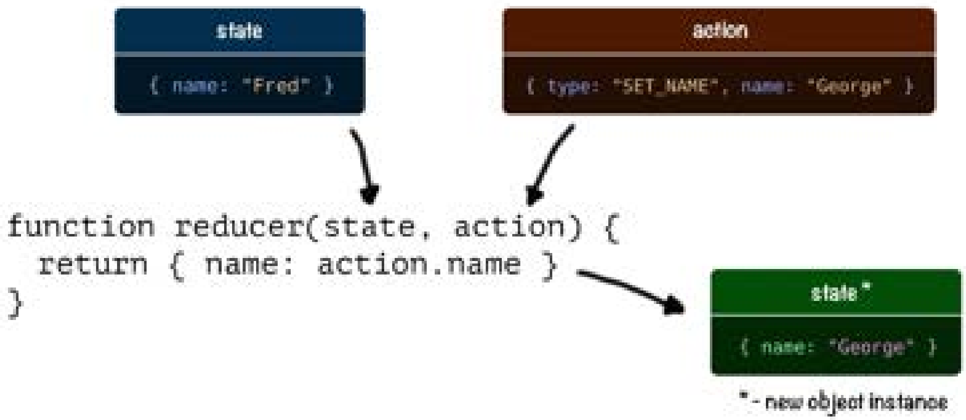


Instead of mutating the object we create a new instance

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 they are generated 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`:

01-first-app/use-reducer/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:

01-first-app/use-reducer/src/App.tsx

```
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:

01-first-app/use-reducer/src/App.tsx

```
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:

01-first-app/use-reducer/src/App.tsx

```
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:

01-first-app/use-reducer/src/App.tsx

```
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.

Now let's define the action creators:

01-first-app/use-reducer/src/App.tsx

```
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:

01-first-app/use-reducer/src/App.tsx

```
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 State Management

Define App State Context. Using ReactContext With TypeScript

Here we'll 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`. Define the application data - for now let's hardcode it:

01-first-app/step3/src/state/AppStateContext.tsx

```
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" }]
    }
  ]
}
```

Here we use arrays to store the lists and the tasks. It will allow us to move the items around, because arrays preserve the order of the elements in them.

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.

As you can see our data object has the `AppState` type. Let's define it along with the types it depends on:

01-first-app/step3/src/state/AppStateContext.tsx

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

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

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

We create the types for our data so that each type has one level of properties.

I decided to use the terms `Task`/`List` for the data types and `Column`/`Card` for UI components.

Now we'll define a context to propagate the data across the whole application. So you won't have to pass the props through multiple components.

Import `createContext` from `react`:

01-first-app/step3/src/state/AppStateContext.tsx

```
import { createContext } from "react"
```

Use `createContext` to define the `AppStateContext`.

01-first-app/step3/src/state/AppStateContext.tsx

```
const AppStateContext = createContext()
```

We'll need to provide the type for our context. Let's define it first:

01-first-app/step3/src/state/AppStateContext.tsx

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

For now, we only want to make our `appState` available through the context so it's the only field in our type as well.

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:

01-first-app/step3/src/state/AppStateContext.tsx

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

Import the `FC` type from react, and also the `useContext` hook, we'll need it in a moment:

01-first-app/step3/src/state/AppStateContext.tsx

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

And now let's define the `AppStateProvider`:

01-first-app/step3/src/state/AppStateContext.tsx

```
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`.

01-first-app/step3/src/index.tsx

```
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:

01-first-app/step3/src/state/AppStateContext.tsx

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

Then define a custom hook called `useAppState`:

01-first-app/step3/src/state/AppStateContext.tsx

```
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 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:

01-first-app/step3/src/Card.tsx

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

Then update the `Column` component. Remove the `React.FC` type from the component definition. Now we'll specify the type of the props as the argument type:

01-first-app/step3/src/Column.tsx

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

Define the prop `id`. We'll need this value to find the corresponding tasks.

Import the `useAppState` hook:

01-first-app/step3/src/Column.tsx

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

And the `Card` component:

01-first-app/step3/src/Column.tsx

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

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:

01-first-app/step3/src/Column.tsx

```
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 task"
        onAdd={console.log}
        dark
      />
    </ColumnContainer>
  )
}
```

Now go to `src/App.tsx`. Let's use our `useAppState` hook to retrieve the `lists`.

Import the hook:

01-first-app/step3/src/App.tsx

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

Then update the layout:

01-first-app/step3/src/App.tsx

```
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>
  )
}
```

K> Don't forget to remove the `Card` component import.

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

We didn't have to specify the type of the loop variable `list`. TypeScript derived 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. It's time to make it possible to update the data. Let's add some actions and reducers.

You can find the working example for this part in the `code/01-first-app/step3`.

Adding Items

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.

Before we do it let's reorganise our code a bit. Create a new folder `src/state`. It will contain the code related to global state management.

Move the `src/state/AppStateContext.tsx` to this folder. And create a new file called `src/state/actions.ts`.

You might have to update the imports, because the path to the `AppStateContext` has changed.

Usually VSCode updates the imports paths automatically. The only thing that will be left to do in this case will be to run `Save all` command from the command palette

Define 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:

01-first-app/step4/src/state/actions.ts

```
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[52].

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 define the types in the union using the `interface` syntax:

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

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

type Action = AddListAction | AddTaskAction
```

[52] https://en.wikipedia.org/wiki/Logical_disjunction

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

The technique we are using here is called *discriminated union*.

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:

```
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` define and export two functions:

01-first-app/step4/src/state/actions.ts

```
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 appStateReducer

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

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

01-first-app/step4/src/state/appStateReducer.ts

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

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

01-first-app/step4/src/state/appStateReducer.ts

```
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`:

01-first-app/step4/src/state/appStateReducer.ts

```
export const appStateReducer = (state: AppState, action: Action): AppSt\
ate => {
  switch (action.type) {
  // ...
    default: {
      return state
    }
  }
}
```

Now go to `src/state/AppStateContext.tsx` and import the `appStateReducer`, `AppState`, `List` and `Task` types:

01-first-app/step4/src/state/AppStateContext.tsx

```
import {
  appStateReducer,
  AppState,
  List,
  Task
} from "./appStateReducer"
```

Provide Dispatch Through The Context

Open the `src/state/AppStateContext.tsx`, import the `Action` type from `./actions`, `useReducer` hook and the `Dispatch` type from `react`.

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

01-first-app/step4/src/state/AppStateContext.tsx

```
import { createContext, useReducer, useContext, Dispatch, FC } from "re\
act"
import { Action } from './actions'
  // ...
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 `useReducer`:

```
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`:

01-first-app/step4/src/state/AppStateContext.tsx

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

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

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

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

Adding Lists

The reducer needs to return a new instance of an object. Se we'll use the spread operator to get all the fields from the previous state. Then we'll set `lists` field to be a new array of the old lists plus the new item.

Open the `src/state/appStateReducer.ts` and add a new `case` block to the reducer:

01-first-app/step4/src/state/appStateReducer.ts

```
    case "ADD_LIST": {
      return {
        ...state,
        lists: [
          ...state.lists,
          { id: nanoid(), text: action.payload, tasks: [] }
        ]
      }
    }
```

New columns have `text`, `id` and `tasks` fields. The `text` field contains the list's title (we get its value from `action.payload`), `lists` will be an empty array and the `id` for each list has to be unique. We'll use nanoid[53] to generate new identifiers.

We need to install this library:

```
yarn add nanoid@3.1.22
```

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

01-first-app/step4/src/state/appStateReducer.ts

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

Adding Tasks

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

So first we'll need to find the target list index. Then we override this list with a new one, where we add the new task. And then we return a new state object, where we override the target list with the updated one.

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

This is a lot of code. If only we could mutate the state and just push the new task to the target list.

Thanks to ImmerJS it is possible. This is a library that allows you to mutate an object and it will create a new object instance based on your mutations. That's exactly what we need.

This library also has hook version that allows you to use it instead of `useReducer`. Let's install the lib:

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

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

After it is installed go to `src/state/AppStateContext` and import `useImmerReducer` from `use-immer`:

01-first-app/step4/src/state/AppStateContext.tsx

```
import { useImmerReducer } from "use-immer"
```

Remove the `useReducer` import and update the `AppStateProvider` so that it uses `useImmerReducer`:

01-first-app/step4/src/state/AppStateContext.tsx

```
  const [state, dispatch] = useImmerReducer(appStateReducer, appData)
```

After it's done go back to the `src/state/appStateReducer` and update the reducer:

01-first-app/step4/src/state/appStateReducer.ts

```
export const appStateReducer = (draft: AppState, action: Action): AppSt\
ate | void => {
  switch (action.type) {
    case "ADD_LIST": {
      draft.lists.push({
        id: nanoid(),
        text: action.payload,
        tasks: []
      })
      break
    }
  // ...
    default: {
      break
    }
  }
}
```

Here we renamed the `state` into `draft`, so we know that we can mutate it. Also we've changed the `ADD_LIST` case so that it just pushes the new list object to the `lists` array.

We don't need to return the new state value anymore, ImmerJS will handle it automatically.

We also updated the return type of our reducer. The type is now `AppState | void`. Sometimes we still might need to return a new instance of the state, for example to reset the state to the initial value, but as we usually won't return anything - we added the `void` type to the union.

Now we can add the `ADD_TASK` case:

01-first-app/step4/src/state/appStateReducer.ts

```
    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` which we'll define in a moment. After we have the index - we just push the new task object to the target list.

Now let's define the `findItemIndexById` function.

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`.

01-first-app/step4/src/utils/arrayUtils.ts

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

We will use a type variable `TItem` that extends `Item`. That 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:

01-first-app/step4/src/utils/arrayUtils.ts

```
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 not 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.

Now go back to `src/state/appStateReducer` and import the `findItemByIndex` function:

01-first-app/step4/src/state/appStateReducer.ts

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

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

Dispatching Actions

Go to `src/App.tsx` and update the code.

Import the `addList` action creator from `src/state/actions`:

01-first-app/step4/src/App.tsx

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

Then update the `App` component layout:

01-first-app/step4/src/App.tsx

```
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>
  )
}
```

Now 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:

01-first-app/step4/src/Column.tsx

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

Then update the component:

01-first-app/step4/src/Column.tsx

```
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>
  )
}
```

Here we also 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/step4`.

Moving Items

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

Moving Columns

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

Open `src/utils/arrayUtils.ts` which will hold this function:

01-first-app/step5/src/utils/arrayUtils.ts

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

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

First we store the item in the `item` constant.

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.

Let's define `removeItemAtIndex` first:

01-first-app/step5/src/utils/arrayUtils.ts

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

Here 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`.

Then define the `insertItemAtIndex`:

01-first-app/step5/src/utils/arrayUtils.ts

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

01-first-app/step5/src/state/appStateReducer.ts

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

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

01-first-app/step5/src/state/actions.ts

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

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

Now define the action creator for it:

01-first-app/step5/src/state/actions.ts

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

We've added a `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`:

01-first-app/step5/src/state/appStateReducer.ts

```
    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@4.0.0
```

`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.

01-first-app/step5/src/index.tsx

```
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 have to 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:

01-first-app/step5/src/DragItem.ts

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

export type DragItem = ColumnDragItem
```

Later it 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` and import the `DragItem` type:

01-first-app/step5/src/state/appStateReducer.ts

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

Update the `AppState` type:

01-first-app/step5/src/state/appStateReducer.ts

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

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

01-first-app/step5/src/state/AppStateContext.tsx

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

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

01-first-app/step5/src/state/AppStateContext.tsx

```
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:

01-first-app/step5/src/state/AppStateContext.tsx

```
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, getTasksByLi\
stId, dispatch }}>
      {children}
    </AppStateContext.Provider>
```

```
  )
}
```

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

01-first-app/step5/src/state/actions.ts

```
  | {
      type: "SET_DRAGGED_ITEM"
      payload: DragItem | null
    }
```

It will hold the `DragItem` that we defined earlier. We need 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:

01-first-app/step5/src/state/actions.ts

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

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

01-first-app/step5/src/state/appStateReducer.ts

```
    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:

01-first-app/step5/src/utils/useItemDrag.ts

```
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] = setState("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 the dragging for the `Column` component.

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

01-first-app/step5/src/Column.tsx

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

Define the `ref` that will hold the reference to the dragged `div` element. Get the `drag` connector function from the `useItemDrag`. Pass the `ref` to the `drag` function and also pass it as a prop to the `ColumnContainer`:

01-first-app/step5/src/Column.tsx

```
export const Column = ({ text, id }: ColumnProps) => {
  const { draggedItem, getTasksByListId, dispatch } = useAppState()
  const tasks = getTasksByListId(id)
  const ref = useRef<HTMLDivElement>(null)

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

  drag(ref)

  return (
    <ColumnContainer ref={ref}>
      //... Column layout
    </ColumnContainer>
  )
}
```

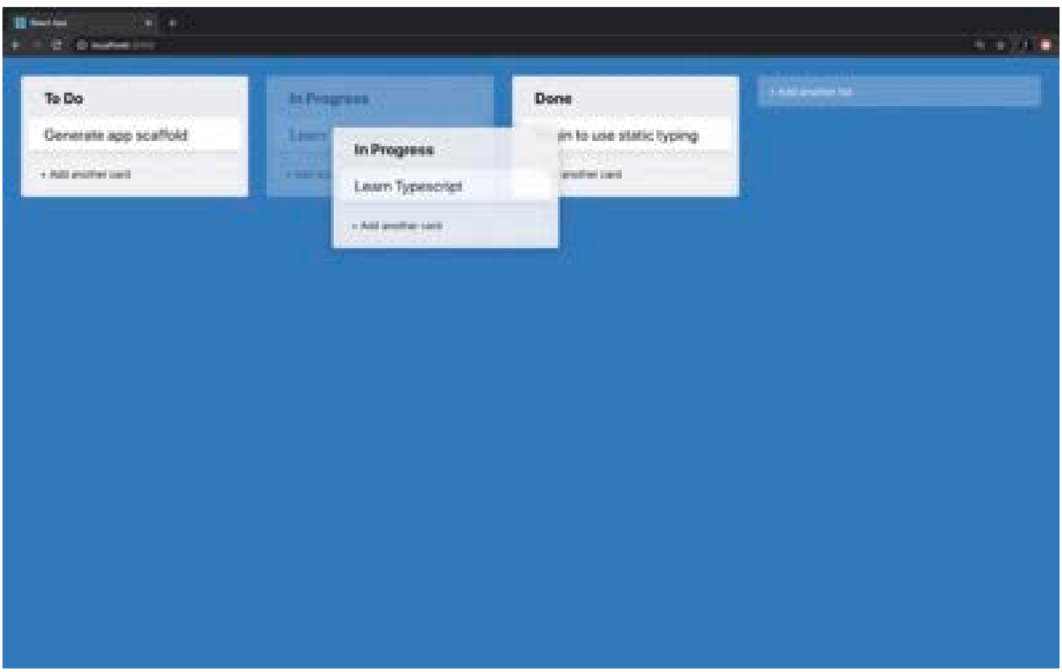
You don't need to remove the `ColumnContainer` contents. I've just omitted them here for brevity. The only thing that changes in the layout is that we add the `ref` to the `ColumnContainer` element.

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. You can see that we provided it as a `ref` prop to `ColumnContainer`.

Then we call our `useItemDrag` hook. We pass an object that will represent the dragged item. We can tell that it's a `COLUMN` and we pass the `id`, `index` and `text`. This hook returns the `drag` function.

Next, we pass our `ref` to the `drag` function.

Now you can launch the app and verify that you can drag the column.



Column is leaving a "ghost" image

Move The Column

We can now drag the column, but it just creates a "ghost" image of the dragged column and leaves the original column in place. Also, we can't drop the column anywhere.

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

Open `src/Column.tsx` file and add the imports, you will need `useDrop` from `react-dnd`, `moveList` from `src/state/actions` and the `DragItem` type from `src/DragItem`:

01-first-app/step5/src/Column1.tsx

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

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

01-first-app/step5/src/Column1.tsx

```
  const [, drop] = useDrop({
    accept: "COLUMN",
    hover() {
      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 `hover` callback. The `hover` callback is triggered whenever you move the dragged item above the drop target.

Inside our `hover` callback we check that `dragIndex` and `hoverIndex` are not the same (which means we aren't hovering above the dragged item).

If the `dragIndex` and `hoverIndex` are different, we dispatch a `MOVE_LIST` action.

Finally, we update the index of the `react-dnd` item reference.

Now combine the `drag` and `drop` calls:

01-first-app/step5/src/Column1.tsx

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

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`.

01-first-app/step5/src/styles.ts

```
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`. It has to extend `DragPreviewContainer` component:

01-first-app/step5/src/styles.ts

```
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[54]

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

01-first-app/step5/src/styles.ts

```
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;
`
```

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

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` with the following code:

01-first-app/step5/src/utils/isHidden.ts

```
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:

01-first-app/step5/src/Column.tsx

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

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

01-first-app/step5/src/Column.tsx

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

At this point, we have an app in which we can drag the columns around.

You can find the working example for this part in the `code/01-first-app/step5`.

Implement The Custom Dragging Preview

If you open an actual *Trello* board, you'll notice that when you drag the items around, their preview is a little bit slanted.

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

We need 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`:

01-first-app/step6/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:

01-first-app/step6/src/CustomDragLayer.tsx

```
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:

01-first-app/step6/src/CustomDragLayer.tsx

```
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: DragLa\
yerMonitor) => 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 need 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 a bit later in this chapter 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 semitransparent. 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 `Column` componen that we use as a preview here those fields will always match, because we get them from the dragged item object.

To fix this let's pass an additional prop `isPreview` to our `Column` component:

01-first-app/step6/src/CustomDragLayer.tsx

```
export const CustomDragLayer = () => {
  const { draggedItem } = useAppState()
  const { currentOffset } = useDragLayer((monitor) => ({
    currentOffset: monitor.getSourceClientOffset()
  }))

  return draggedItem && currentOffset ? (
    <CustomDragLayerContainer>
      <Column
        id={draggedItem.id}
        text={draggedItem.text}
        isPreview
      />
    </CustomDragLayerContainer>
  ) : null
}
```

You will notice that immediately after you pass the `isPreview` prop to the `Column` you will get a TypeScript error:

Property 'isPreview' does not exist on type 'IntrinsicAttributes & ColumnProps'

Open the `src/Column.tsx` and add a new prop `isPreview`:

01-first-app/step6/src/Column.tsx

```
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:

01-first-app/step6/src/Column.tsx

```
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/step6/src/Column.tsx`.

Now TypeScript will complain that neither the `ColumnContainer` component nor the `isHidden` function accept this new property.

Let's fix the `ColumnContainer` first. Open `src/styles.ts` and add a new prop to the `DragPreviewContainerProps`:

01-first-app/step6/src/styles.ts

```
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 : 1)};
`
```

Here we immediately 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`.

Then we'll fix the `isHidden` function. Open `src/utils/isHidden` and add a new `boolean` argument `isPreview`:

01-first-app/step6/src/utils/isHidden.ts

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

```
  )
}
```

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:

01-first-app/step6/src/styles.ts

```
type DragPreviewWrapperProps = {
  position: {
    x: number
    y: number
  }
}
```

It will receive a prop `position` with the `x` and `y` coordinates.

Now define the styled component:

01-first-app/step6/src/styles.ts

```
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[55] 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. 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` and import thet `DragPreviewWrapper` from the styles:

01-first-app/step6/src/CustomDragLayer.tsx

```
import {
  CustomDragLayerContainer,
  DragPreviewWrapper
} from "./styles"
```

Then wrap the `Column` component into the `DragPreviewWrapper`. Pass the `currentOffset` to the `DragPreviewWrapper`.

[55] https://styled-components.com/docs/api#attrs

01-first-app/step6/src/CustomDragLayer.tsx

```
      <DragPreviewWrapper position={currentOffset}>
        <Column
          id={draggedItem.id}
          text={draggedItem.text}
          isPreview
        />
      </DragPreviewWrapper>
```

Now we need to mount the `CustomDragLayer` component inside the `App` layout, and then we'll need to hide the default drag preview.

Open `src/App.tsx`, import `CustomDragLayer` and add it to the `App` layout above the columns:

01-first-app/step6/src/App.tsx

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

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

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

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`:

<<01-first-app/step6/src/utils/useItemDrag.ts[56]

Also import the `useEffect` hook from `react`:

<<01-first-app/step6/src/utils/useItemDrag.ts[57]

Now add a new `useEffect` call in the end of our hook:

01-first-app/step6/src/utils/useItemDrag.ts

```
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`.

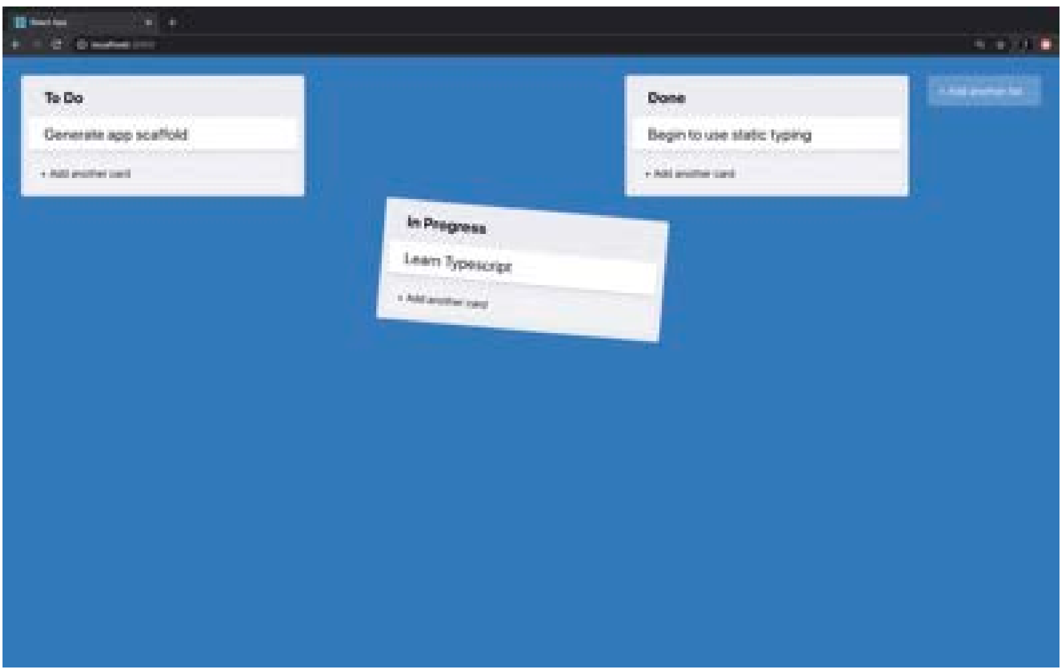
At this point we don't need to make the dragged columns semi transparent. Open `src/styles.ts` and set the hidden state `opacity` to `0`.

56 ./code/01-first-app/step6/src/utils/useItemDrag.ts
57 ./code/01-first-app/step6/src/utils/useItemDrag.ts

01-first-app/step6/src/styles.ts

```
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/step6`.

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:

01-first-app/step7/src/state/actions.ts

```
  | {
      type: "MOVE_TASK"
      payload: {
        draggedItemId: string
        hoveredItemId: string | null
        sourceColumnId: string
        targetColumnId: string
      }
    }
```

This action accepts `draggedId` and `hoverId` 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:

01-first-app/step7/src/state/actions.ts

```
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.

01-first-app/step7/src/DragItem.ts

```
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:

01-first-app/step7/src/Card.tsx

```
type CardProps = {
  text: string
  id: string
  columnId: string
  isPreview?: boolean
}
```

Get this new prop from the destructured props object:

01-first-app/step7/src/Card.tsx

```
export const Card = ({
  text,
  id,
  columnId,
  isPreview
}: CardProps) => {
  // ...
```

Now we can pass the `columnId` to our `Card` components. Open the `src/Column` and pass the `id` as the `columnId` to the cards:

01-first-app/step7/src/Column.tsx

```
        <Card
          id={task.id}
          columnId={id}
          text={task.text}
          key={task.id}
        />
```

After it's done switch back to the `src/Card.tsx` and add the imports:

01-first-app/step7/src/Card.tsx

```
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"
```

Get the `state` and `dispatch` from the `useAppState`, get the `CardContainer` ref and update the card layout:

01-first-app/step7/src/Card.tsx

```
export const Card = ({
  text,
  id,
  columnId,
  isPreview
}: CardProps) => {
  const { draggedItem, dispatch } = useAppState()
  const ref = useRef<HTMLDivElement>(null)
  // ...
    <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:

01-first-app/step7/src/Card.tsx

```
  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:

01-first-app/step7/src/Card.tsx

```
  const [, drop] = useDrop({
    accept: "CARD",
    hover() {
      if (!draggedItem) {
        return
      }
      if (draggedItem.type !== "CARD") {
        return
      }
      if (draggedItem.id === id) {
        return
      }

      dispatch(
        moveTask(draggedItem.id, id, draggedItem.columnId, 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.

After it's done, wrap the `ref` into the `drag` and the `drop` function calls, just like we did in our `Column` component:

01-first-app/step7/src/Card.tsx

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

Update CustomDragLayer

Open `src/CustomDragLayer` and import the `Card` component:

01-first-app/step7/src/CustomDragLayer.tsx

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

Then add a ternary operator to the layout to check what we are dragging:

01-first-app/step7/src/CustomDragLayer.tsx

```
        {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:

01-first-app/step7/src/state/appStateReducer.ts

```
    case "MOVE_TASK": {
  // ...
    }
```

Then inside this block we need to destructure the `action.payload` like this:

01-first-app/step7/src/state/appStateReducer.ts

```
      const {
        draggedItemId,
        hoveredItemId,
        sourceColumnId,
        targetColumnId
      } = action.payload
```

Then we need to get the source and target list indices:

01-first-app/step7/src/state/appStateReducer.ts

```
      const sourceListIndex = findItemIndexById(
        draft.lists,
        sourceColumnId
      )
      const targetListIndex = findItemIndexById(
        draft.lists,
        targetColumnId
      )
```

Then we need to find the indices of the dragged and hovered items:

01-first-app/step7/src/state/appStateReducer.ts

```
      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 we need to store the moved item in a variable:

01-first-app/step7/src/state/appStateReducer.ts

```
      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:

01-first-app/step7/src/state/appStateReducer.ts

```
      // 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. Pretty soon you'll notice 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/step7.

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 a card over it, the card will be moved to this empty column.

To do this we'll edit our `Column` drop `hover` code and add `CARD` to supported item types.

01-first-app/step8/src/Column.tsx

```
    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:

01-first-app/step8/src/Column.tsx

```
import { addTask, moveTask, moveList, setDraggedItem } from "./state/ac\
tions"
```

Then add the following code to the `useDrop` hook:

01-first-app/step8/src/Column.tsx

```
    hover(item: DragItem) {
      if (item.type === "COLUMN") {
        // ... dragging 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 `item.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/step8`.

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.

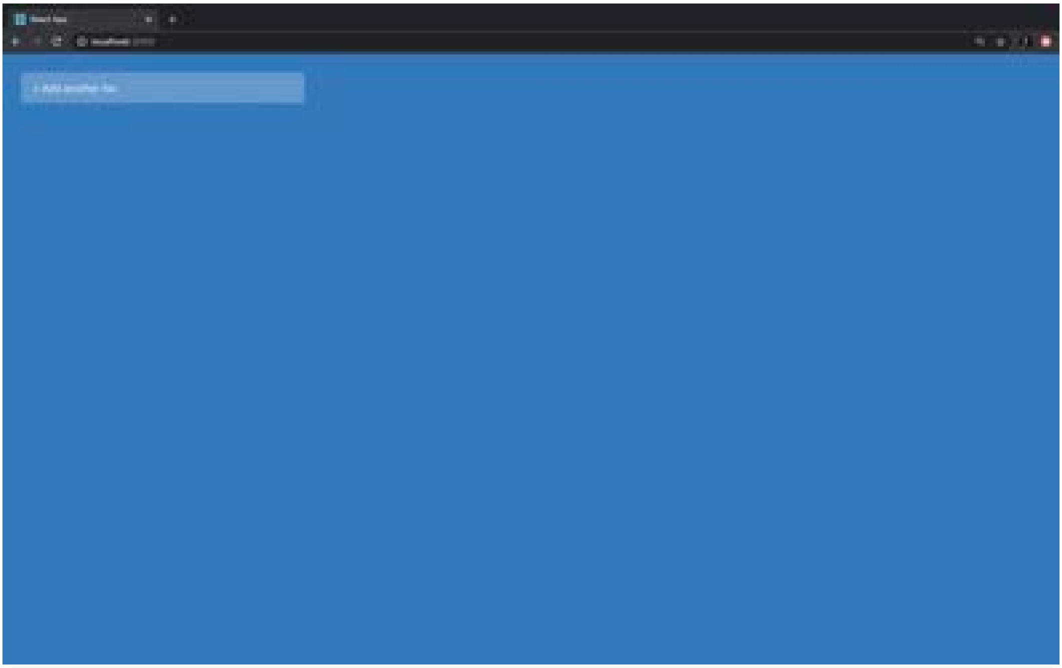
Launch the sample backend in a separate terminal tab:

```
cd code/01-first-app/trello-backend
yarn && yarn start
```

The completed example for this chapter is located in `code/01-first-app/step9`. `cd` to this folder and launch the app:

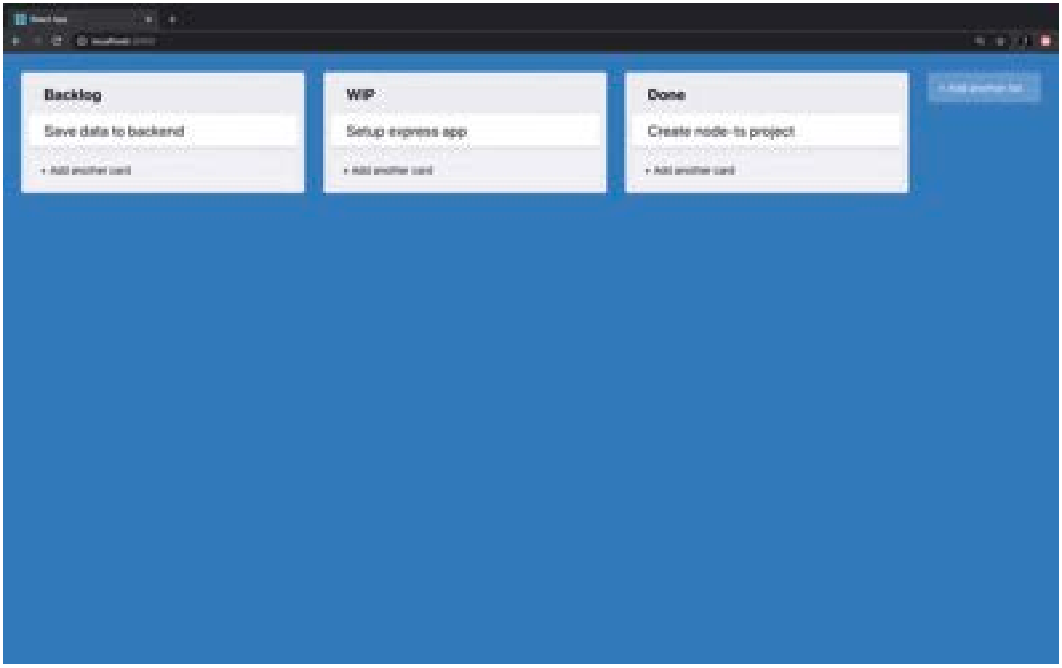
```
cd code/01-first-app/step9
yarn && yarn start
```

Initially, you should see an empty field with the "+ Create new 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/step9` 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<Respons\
e>;
```

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[58].

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

[58] https://www.typescriptlang.org/play/?ssl=8&ssc=13&pln=1&pc=1#code/MYewdgzgLgBAZgUysAFgEQIZQzAvDDCATzGBgAoBK
rwgJ+4-gBsYAXzwEA7hgCWsRMhTkA5CihQADhABcAegsZj6gHQBzTSl4AjO6AC2eyi3b+
YOygUYXJyIQhjcElqXDomVgCkoSgBMBgIqMgEOwArCHAqAggYAAV+EE91SQAeHmABIRFxSWkBWTMYaH51MAdFGgBuPyTFX0

do this because by default the `response.json()` would have the type `any`. I'm also 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[59], 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 with the following contents:

<<01-first-app/step9/.env[60]

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 `api.ts` and define the `save` function:

[59]https://github.com/axios/axios
[60]./code/01-first-app/step9/.env

01-first-app/step9/src/api.ts

```
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:

01-first-app/step9/src/api.ts

```
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:

01-first-app/step9/src/state/AppStateContext.tsx

```
import { createContext, useContext, useEffect, Dispatch } from "react"
  // ...
import { save } from "../api"
```

Add the following code right before the `AppStateProvider` return statement:

01-first-app/step9/src/state/AppStateContext.tsx

```
  useEffect(() => {
    save(state)
  }, [state])
```

The useEffect[61] 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:

61

Backend console output

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:

01-first-app/step9/src/withInitialState.tsx

```
import { useState, useEffect, ComponentType } from "react"
import { AppState } from "./state/appStateReducer"
```

Then define and export our `withInitialState` HOC:

01-first-app/step9/src/withInitialState.tsx

```
type InjectedProps = {
  initialState: AppState
}

export function withInitialState<TProps>(
  WrappedComponent: ComponentType<
    TProps & InjectedProps
  >
) {
  return (props: Omit<TProps, keyof InjectedProps>) => {
    const [initialState, setInitialState] = useState<AppState>({
      lists: [],
      draggedItem: null
```

```
    })
  // ...
    return (
      <WrappedComponent
        {...props as TProps}
        initialState={initialState}
      />
    )
  }
}
```

Let's go line-by-line. First we define a type that will represent the props that we are injecting. In this case it is the `initialState: AppState` prop:

01-first-app/step9/src/withInitialState.tsx

```
type InjectedProps = {
  initialState: AppState
}
```

Then, we define a `withInitialState` function that accepts a `WrappedComponent` argument. This `WrappedComponent` has a complex type declaration:

```
WrappedComponent: React.ComponentType<
  TProps & InjectedProps
>
```

Here we say that `WrappedComponent` accepts an intersection type that contains the props from the type variable `TProps` and the props defined in the `InjectedProps`.

The `TProps` is defined as a type argument of our generic function `withInitialState`. This way if the component that we'll wrap into `withInitialState` will receive some other props, TypeScript will use them as `TProps`.

Then inside our function, we return a nameless function component:

01-first-app/step9/src/withInitialState.tsx

```
  return (props: Omit<TProps, keyof InjectedProps>) => {
    const [initialState, setInitialState] = useState<AppState>({
      lists: [],
      draggedItem: null
    })
  // ...
    return (
      <WrappedComponent
        {...props as TProps}
        initialState={initialState}
      />
    )
  }
```

This component should not accept the prop that we inject using this HOC. We don't want to let the user provide this prop, because our HOC already does it. This is why we use a utility type `Omit`. It allows us to create a new type that won't have the keys of the `InjectedProps` type.

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[62].

The query `keyOf` returns a union type that contains the keys of the type that you pass to it, for example:

```
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[63].

Then we return the `WrappedComponent`(in our app it will be `AppStateProvider`) passing the `initialState` and the rest of the props to it.

01-first-app/step9/src/withInitialState.tsx

```
    return (
      <WrappedComponent
        {...props as TProps}
        initialState={initialState}
      />
    )
```

We have to add the type assertion for the `props` here, because otherwise we'll get a typescript error:

'TProps' could be instantiated with an arbitrary type which could be unrelated to 'Pick<TProps, Exclude<keyof TProps, "initialState">> & { initialState: AppState; }'.ts(2322)

[62] https://www.typescriptlang.org/docs/handbook/utility-types.html
[63] typescriptlang.org/docs/handbook/release-notes/typescript-2-1.html#keyof-and-lookup-types

Here is what happens. TypeScript treats *type variables* like they can be anything, we don't know what will be the actual type, so it is extra cautious with them.

In our code we set the type of the props of the `WrappedComponent` to be `TProps`. For TypeScript it means that it can potentially be any type.

Then on the wrapper component we define the props to be `Omit<TProps, keyof InjectedProps>`. For us it looks like this type should be a subset of the `TProps`, because we just remove one of its fields. But for TypeScript it is a completely different type, it does not "see" it as a subset of `TProps`.

So when we spread the `props` and pass the `initialState` prop to our `WrappedComponent` TypeScript does not understand that together they matche te `TProps` type.

Or in other words:

```
TProps !== Omit<TProps, keyof InjectedProps> & InjectedProps
```

Here we fixed it by using the type assertion and forcing TypeScript to beleive that the `props` that we pass to the `WrappedComponent` have the `TProps` type.

Using type assertions can be harmful sometimes and can increase the chance of human error, so I try to avoid using them when possible.

In our case we can actually help TypeScript to figure out the correct types:

01-first-app/step9/src/withInitialState.tsx

```
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} />
  }
}
```

First of all we define an additional type `PropsWithoutInjected`:

01-first-app/step9/src/withInitialState.tsx

```
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.

Then we define the `WrappedComponent` props as an intersection type between the `PropsWithoutInjected<TProps>` and the `InjectedProps`:

01-first-app/step9/src/withInitialState.tsx

```
export function withInitialState<TProps>(
  WrappedComponent: React.ComponentType<
    PropsWithoutInjected<TProps> & InjectedProps
  >
) {
```

So we kind of reconstruct the original `TProps` type by first removing the injected prop from it and then creating a new type as an intersection with the `InjectedProps`.

Then we specify the type of the wrapper component `props` to be `PropsWithoutInjected<TProps>`:

01-first-app/step9/src/withInitialState.tsx

```
  return (props: PropsWithoutInjected<TProps>) => {
```

Here we don't add the `InjectedProps` to prevent the user from passing them.

Now we can pass both `props` and the `initialState` value to the `WrappedComponent`:

01-first-app/step9/src/withInitialState.tsx

```
    return <WrappedComponent {...props} initialState={initialState} />
```

Now TypeScript won't complain and we didn't have to use the type assertion! Woohoo!

Now we can add the data loading logic to our HOC.

> 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

Import `useState` and `useEffect` from React and the `load` function from the `api` module:

01-first-app/step9/src/withInitialState.tsx

```
import { useState, useEffect } from "react"
  // ...
import { load } from "./api"
```

Inside our wrapper component add two more states and a `useEffect` hook:

01-first-app/step9/src/withInitialState.tsx

```
  return (props: PropsWithoutInjected<TProps>) => {
    const [initialState, setInitialState] = useState<AppState>({
      lists: [],
      draggedItem: null
    })
    const [isLoading, setIsLoading] = useState(true)
    const [error, setError] = useState<Error | undefined>()

    useEffect(() => {
      const fetchInitialState = async () => {
        try {
          const data = await load()
          setInitialState(data)
        } catch (e) {
          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 need to 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 need to 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 where we load the data and store it in our state and if something goes wrong we save the error.

Now let's update the wrapper component layout.

01-first-app/step9/src/withInitialState.tsx

```
  return (props: PropsWithoutInjected<TProps>) => {
  // ...

    if (isLoading) {
      return <div>Loading</div>
    }

    if (error) {
      return <div>{error.message}</div>
    }

    return <WrappedComponent {...props} initialState={initialState} />
  }
}
```

Here I've omitted the data loading logic, but it is still there, don't remove it.

Here we show the loader if `isLoading` state is `true`. We show an error message if something went wrong. And we return the wrapped component if the data was loaded successfully.

Use The HOC

Now the HOC is ready, import it into `src/state/AppStateContext.tsx`:

01-first-app/step9/src/state/AppStateContext.tsx

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

Define the `AppStateProviderProps`:

01-first-app/step9/src/state/AppStateContext.tsx

```
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:

01-first-app/step9/src/state/AppStateContext.tsx

```
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>
    )
  }
)
```

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/step9`.

How to Test Your Applications: Testing a Digital Goods Store

Introduction

In this part, we will learn 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[64] 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
```

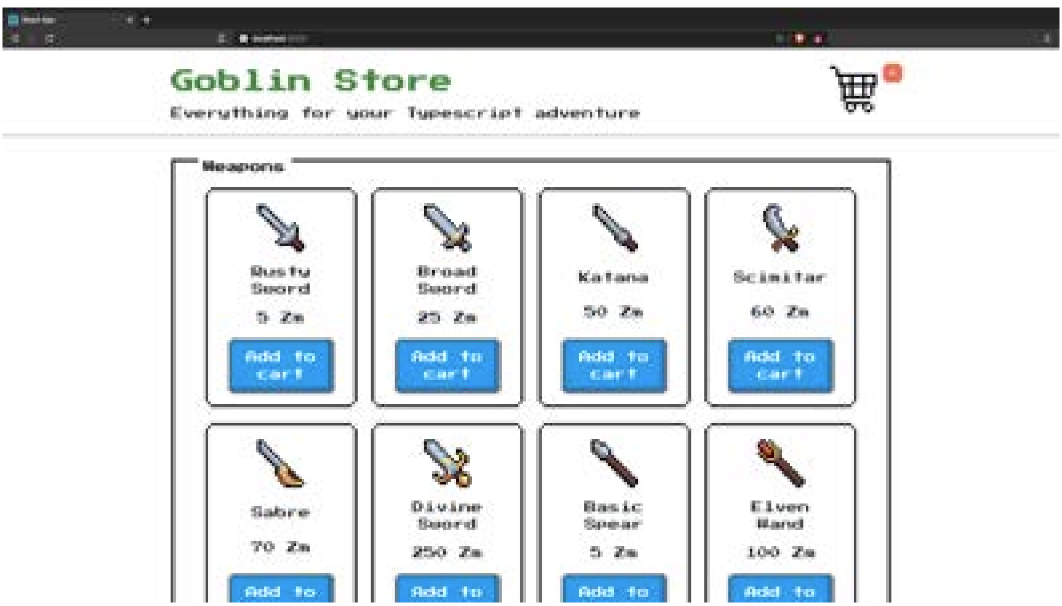
When you are there, install the dependencies and launch the app:

[64] https://testing-library.com/docs/react-testing-library/intro

```
yarn && yarn dev
```

> The `yarn dev` command runs both a server and a client. We use concurrently[65] to launch two scripts at the same time. You can check `src/package.json` to see how we do it.

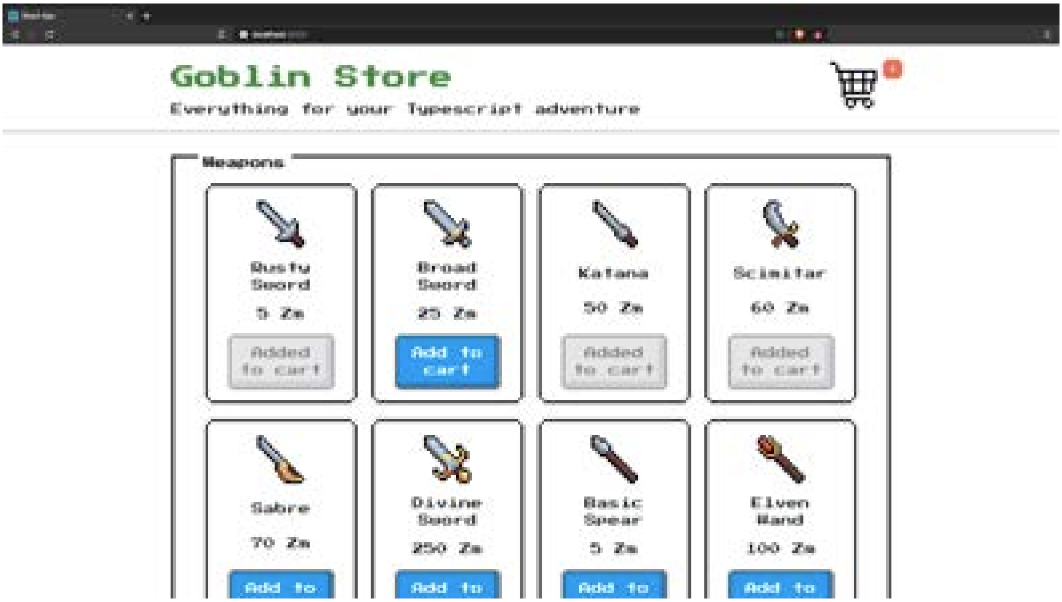
It should also open the app in the browser. If that doesn't happen, navigate to `http://localhost:3000` and open it manually.



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.

[65]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 any more. 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/step1` to your workspace folder. It will be our starting point.

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
```

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 && 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`.

Install VSCode plugin

If you are using VSCode, you can install a useful Jest plugin[66] that automatically runs the tests and displays the test results right in the text editor.

[66] 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 the visual feedback from running your tests way quicker.

If it doesn't show up automatically, launch `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.[67]

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 it 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",
```

[67] https://github.com/jest-community/vscode-jest/blob/master/README.md#troubleshooting

```
      "console": "integratedTerminal",
      "internalConsoleOptions": "neverOpen",
      "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 tests run serially in one process. It's hard to debug many processes at the same time.
- `--no-cache` disables cache, to avoid cache-related problems during debugging.
- `--watchAll=false` disables re-running tests when any related files change. We want to perform a single run, so we set this flag to `false`.

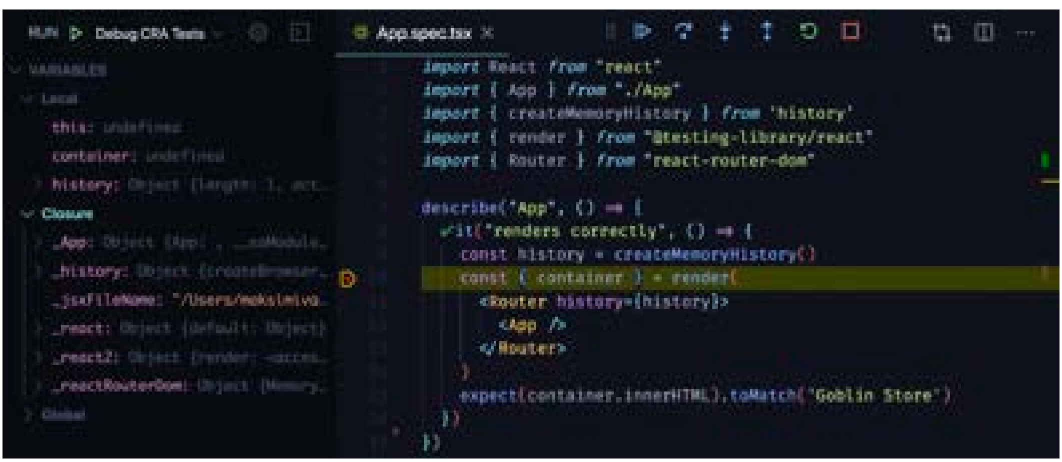
This configuration will work with any Create React App generated application.

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 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.

02-testing/completed/src/index.tsx

```
import React 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(
  <React.StrictMode>
    <CartProvider>
      <BrowserRouter>
        <App />
      </BrowserRouter>
    </CartProvider>
  </React.StrictMode>,
  document.getElementById("root")
)
```

Here we render our `App` component. Note that it is wrapped into three providers here:

- `<ProductsProvider>` holds information about products. It automatically loads the data from the backend and makes it available across the application.
- `<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.

App Component and Testing Context

Open `src/App.tsx`. This file contains `App` component definition.

02-testing/completed/src/App.tsx

```
import React from "react"
import { Switch, Route } from "react-router-dom"
import { Checkout } from "./Checkout"
import { Home } from "./Home"
import { Cart } from "./Cart"
import { Header } from "./shared/Header"
import { OrderSummary } from "./OrderSummary"

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. Open `src/App.spec.tsx` and add the following code:

02-testing/completed/src/App.spec.tsx

```
import React from "react"
import { App } from "./App"
import { render } from "@testing-library/react"

describe("App", () => {
  it("renders successfully", () => {
    const { container } = render(<App />)
    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.

02-testing/completed/src/App.spec.tsx

```
    const { container } = render(<App />)
    expect(container.innerHTML).toMatch("Goblin Store")
```

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 `<Router>` context (`Router` also comes from React Router).

Look again back at our `src/index.tsx`. 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 a `BrowserRouter`:

02-testing/completed/src/index.tsx

```
import React 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(
  <React.StrictMode>
    <CartProvider>
      <BrowserRouter>
        <App />
      </BrowserRouter>
    </CartProvider>
  </React.StrictMode>,
  document.getElementById("root")
)
```

However, in our *test* we were trying to run the `App` component directly – *without* the `Router` context (that is, the `<Router>` tag wrapping - or being a parent of - our `App`).

To fix this, **we need to wrap our `App` component into a `Router`** 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[68]. It means that some functionality can be missing or work differently compared to the browser environment.

One of the missing things is a History API[69], so to use routing we'll have to install an additional package that will provide us the History API functionality.

Install `history` as a dev dependency:

[68] https://www.npmjs.com/package/jsdom
[69] https://developer.mozilla.org/en-US/docs/Web/API/History_API

```
yarn add --dev history
```

Now let's fix our test by using our synthetic History API:

02-testing/completed/src/App.spec.tsx

```
import React from "react"
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")
  })

  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")
  })
})
```

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[70] 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[71]. 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 Dependencies

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 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` import the `Home` component and then call `jest.mock` to mock this module:

02-testing/completed/src/App.spec.tsx

```
jest.mock("./Home", () => ({ Home: () => <div>Home</div> }))
```

`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

[70] https://testing-library.com/docs/react-testing-library
[71] https://jestjs.io/docs/en/expect

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 right after the first `it` block define a new `it` block:

02-testing/completed/src/App.spec.tsx

```
  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.

Routing Testing

If you open `src/App.tsx` file, you'll see that our `App` component renders four different routes using `Switch`.

02-testing/completed/src/App.tsx

```
        <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 `src/testHelpers.tsx` that will hold our helper function:

02-testing/completed/src/testHelpers.tsx

```
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[72].

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.

One problem though. TypeScript complains that `Property 'renderWithRouter' does not exist on type 'Global'`. Let's fix that.

First, define the type for our function:

[72] https://nodejs.org/api/globals.html#globals_global

02-testing/completed/src/testHelpers.tsx

```
type RenderWithRouter = (
  renderComponent: () => React.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`.

By default, the `global` object has type `Global`. We can add a new field to it.

02-testing/completed/src/testHelpers.tsx

```
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[73].

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:

[73]https://www.typescriptlang.org/docs/handbook/release-notes/typescript-1-8.html#augmenting-globalmodule-scope-from-modules

```
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[74] type as well. It is a variable that refers to the global scope.

02-testing/completed/src/testHelpers.tsx

```
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`:

02-testing/completed/src/setupTests.ts

```
import "./testHelpers"
```

Writing The Tests

Now let's finally write our routing tests. First, mock the page's components. Add the following code right after you mock the `Home` component:

[74] https://www.typescriptlang.org/docs/handbook/release-notes/typescript-3-4.html#type-checking-for-globalthis

02-testing/completed/src/App.spec.tsx

```
jest.mock("./Cart", () => ({ Cart: () => <div>Cart</div> }))
jest.mock("./Checkout", () => ({
  Checkout: () => <div>Checkout</div>
}))
jest.mock("./OrderSummary", () => ({
  OrderSummary: () => <div>Order summary</div>
}))
```

Now create a new `describe` block with the name `routing` and move our root route test there. Remake it so that it uses `renderWithRouter`:

02-testing/completed/src/App.spec.tsx

```
  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 `/checkout` route:

02-testing/completed/src/App.spec.tsx

```
    it("renders checkout page on '/cart'", () => {
      const { container } = renderWithRouter(
        () => <App />,
        "/cart"
      )
      expect(container.innerHTML).toMatch("Cart")
    })
```

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:

02-testing/completed/src/App.spec.tsx

```
  it("renders checkout page on '/cart'", () => {
    const { container } = renderWithRouter(
      () => <App />,
      "/cart"
    )
    expect(container.innerHTML).toMatch("Cart")
  })
```

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.

Header Component

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 `Header` in isolation.

Create a new file called `src/shared/Header.spec.tsx` with the following contents:

02-testing/completed/src/shared/App.spec.tsx

```
import React from "react"
import { Header } from "./Header"

jest.mock("./CartWidget", () => ({
  CartWidget: () => <div>Cart widget</div>
}))

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 so we'll have to use `renderWithRouter` to be able to test it.

Here 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.

02-testing/completed/src/shared/Header.spec.tsx

```
  it("navigates to / on header title click", () => {
    const { getByText, history } = renderWithRouter(() => <Header />)
    fireEvent.click(getByText("Goblin Store"))
    expect(history.location.pathname).toEqual("/")
  })
```

We click the element that has the text "Goblin Store" on it, and then we expect that we end up on root url.

Here it comes in handy that we return the history object from our `renderWithRouter` helper function. This allows us to check that the current location matches 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`:

02-testing/completed/src/shared/CartWidget/CartWidget.spec.tsx

```
import React from "react"
import { CartWidget } from "./CartWidget"
import { fireEvent } from "@testing-library/react"

describe("CartWidget", () => {
  it.todo("shows the amount of products in the cart")

  it.todo("navigates to cart summary page on 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.

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:

02-testing/completed/src/shared/CartWidget/CartWidget.spec.tsx

```
  it("navigates to cart summary page on click", () => {
    const { getByRole, history } = renderWithRouter(() => (
      <CartWidget />
    ))

    fireEvent.click(getByRole("link"))

    expect(history.location.pathname).toEqual("/cart")
  })
})
```

Here we use the getByRole[75] 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[76].

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`:

[75] https://testing-library.com/docs/dom-testing-library/api-queries#byrole
[76] https://html.spec.whatwg.org/multipage/index.html#contents

02-testing/completed/src/shared/CartWidget/CartWidget.tsx

```
import React from "react"
import { Link } from "react-router-dom"
import cart from "./cart.svg"
import { useCartContext } from "../../CartContext"

interface CartWidgetProps {
  useCartHook?: typeof useCartContext;
}

export const CartWidget = ({useCartHook = useCartContext}: CartWidgetPr\
ops) => {
  const { products } = useCartHook()

  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. But we don't call it directly. Instead, we define a prop called `useCartHook` and assign the `useCartContext` hook as the default value to it.

To specify the type of this prop we use a built-in `typeof` util from TypeScript. This way we can get the type of some value, in this case the type of `useCartContext` hook, and reuse it.

This way, in our test we can easily provide the mocked version of this hook to our component.

Go back to the test code. Let's test that we render the amount of products in the cart correctly:

02-testing/completed/src/shared/CartWidget/CartWidget.spec.tsx

```
  it("shows the amount of products in the cart", () => {
    const stubCartHook = () => ({
      products: [
        {
          name: "Product foo",
          price: 0,
          image: "image.png"
        }
      ],
    })

    const { container } = renderWithRouter(() => (
      <CartWidget useCartHook={stubCartHook} />
    ))

    expect(container.innerHTML).toMatch("1")
  })
```

Here we define a mock version of the `useCartHook`. The mock version returns only the `products` field with a hardcoded product.

But here is the problem. If we define only the `products` field in our returned object, the types of our mocked hook and the `useCartHook` prop of the `CartWidget` won't match.

When we wrote that `useCartHook` has the type of the `useCartContext` hook it meant that we need to have the same type signature. If the `useCartContext` hook has some method or field in returned values then our mocked version should have them as well.

How can we skip the fields that we don't need for our test?

Well, the easiest way to do it is to use the type `any`. Like we did in our test when we passed the mocked hook through the `useCartHook` prop.

02-testing/completed/src/shared/CartWidget/CartWidget.spec.tsx

```
      <CartWidget useCartHook={stubCartHook} />
```

This way you lose the real type information, so I don't recommend this approach. Instead, we could be more specific when defining this `useCartHook` type on our component.

Let's go back to the `src/shared/CartWidget/CartWidget.tsx` and modify the `useCartHook` type.

02-testing/completed/src/shared/CartWidget/CartWidget.tsx

```
interface CartWidgetProps {
  useCartHook?: () => Pick<ReturnType<typeof useCartContext>, "products\
">;
}
```

Now we define the `useCartHook` as a function that returns an object with one field, `products`, from the `useCartContext` return type.

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`. * `Pick` - allows us to create a type with a subset of fields. For example: {lang=ts,line-numbers=off}
interface ExampleType { foo: string; bar: number; }

```
Pick<ExampleType, 'bar'> // { bar: number }
```

Now in our test we don't need to typecast our mocked `useCartHook`:

02-testing/completed/src/shared/CartWidget/CartWidget.spec.tsx

```
  it("shows the amount of products in the cart", () => {
    const stubCartHook = () => ({
      products: [
        {
          name: "Product foo",
          price: 0,
          image: "image.png"
        }
      ],
    })

    const { container } = renderWithRouter(() => (
      <CartWidget useCartHook={stubCartHook} />
    ))

    expect(container.innerHTML).toMatch("1")
  })
```

Loader Component

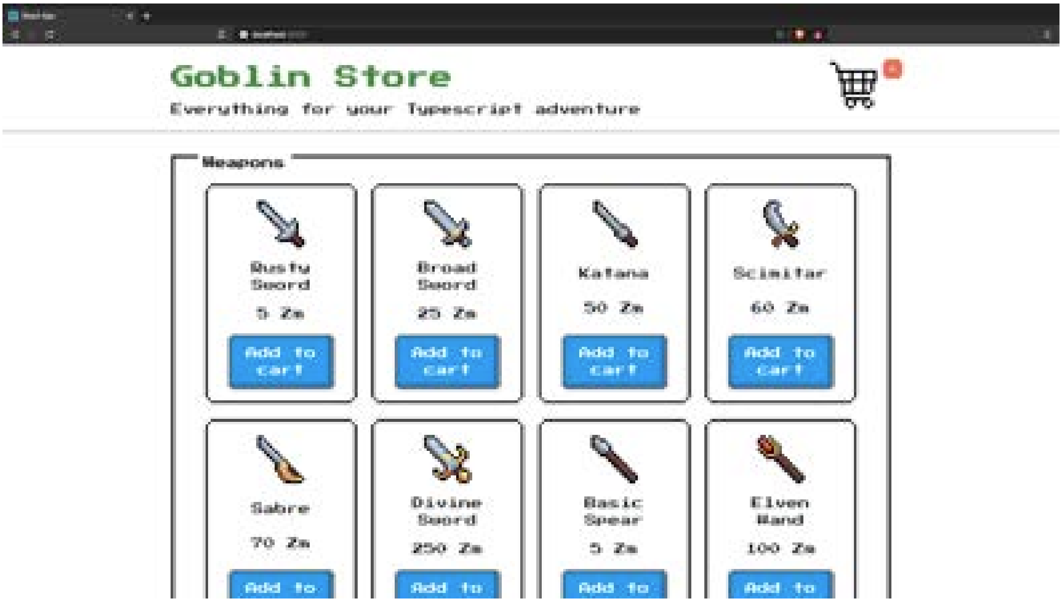
Our `Loader` component does not contain any logic. In our test we'll only make sure that it renders correctly:

02-testing/completed/src/shared/Loader.spec.tsx

```
import React from "react"
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.

02-testing/completed/src/Home/index.ts

```
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

Let's look at the `Home` component props:

02-testing/completed/src/Home/Home.tsx

```
interface HomeProps {
  useProductsHook?: () => {
    categories: Category[]
    isLoading: boolean
    error: boolean
  }
}
```

This component gets the products to render from the `useProducts` hook. To simplify testing of this component I made `useProducts` an explicit dependency by adding it to the component props and setting the default value to be the imported hook.

This way we won't have to mock the `useProducts` module using Jest. We'll be able to pass the stub through the props. It will make our tests a bit simpler and easier to set up.

Also, this approach makes all the component dependencies obvious, which greatly decreases the chance of creating a component that depends on too many things and thus is hard to test.

But as you can see we are manually specifying the return value of the `useProductsHook` function. As we now know a more efficient way, let's rewrite it:

02-testing/completed/src/Home/Home.tsx

```
interface HomeProps {
  useProductsHook?: () => Pick<
    ReturnType<typeof useProducts>,
    "categories" | "isLoading" | "error"
  >
}
```

Now let's move on to the tests. Create a test file called `src/Home.spec.tsx`.

This 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 state our component can end up in:

02-testing/completed/src/Home/Home.spec.tsx

```
describe("Home", () => {
  describe("while loading", () => {
    it.todo("renders categories with products")
  })

  describe("with data", () => {
    it.todo("renders categories with products")
  })

  describe("with error", () => {
    it.todo("renders categories with products")
  })
})
```

Now let's write the individual test cases. First, let's verify that when `isLoading` is `true`, we'll render the `Loader` component.

02-testing/completed/src/Home/Home.spec.tsx

```
describe("while loading", () => {
  it("renders loader", () => {
    const mockUseProducts = () => ({
      categories: [],
      isLoading: true,
      error: false
    })

    const { container } = render(
      <Home useProductsHook={mockUseProducts} />
    )

    expect(container.innerHTML).toMatch("Loading")
  })
})
```

Here we defined our `mockUseProducts` function so that it returns `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:

02-testing/completed/src/Home/Home.spec.tsx

```
describe("with error", () => {
  it("renders error message", () => {
    const mockUseProducts = () => ({
      categories: [],
      isLoading: false,
      error: true
    })

    const { container } = render(
```

```
        <Home useProductsHook={mockUseProducts} />
      )

      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`.

And finally, 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:

02-testing/completed/src/Home/Home.spec.tsx

```
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.

Inside the `describe("with data")` block define a `category` constant:

02-testing/completed/src/Home/Home.spec.tsx

```
const category: Category = {
  name: "Category Foo",
  items: [
    {
      name: "Product foo",
      price: 55,
      image: "/test.jpg"
    }
  ]
}
```

Now 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:

02-testing/completed/src/Home/Home.spec.tsx

```
it("renders categories with products", () => {
  const mockUseProducts = () => ({
    categories: [category],
    isLoading: false,
    error: false
  })

  const { container } = render(
    <Home useProductsHook={mockUseProducts} />
  )

  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.

ProductCart Component

Moving on to the `ProductCart` component. Let's see what we have here.

First of all, we need to render the product data: the image should have the correct `alt` and `src` tags, we need to render the price and 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:

02-testing/completed/src/Home/ProductCard.spec.tsx

```
import React from "react"
import { render, fireEvent } from "@testing-library/react"
import { ProductCard } from "./ProductCard"
import { Product } from "../shared/types"

describe("ProductCard", () => {
  it.todo("renders correctly")

  describe("when product is in the cart", () => {
    it.todo("the 'Add to cart' button is disabled")
  })

  describe("when product is not in the cart", () => {
    describe("on 'Add to cart' click", () => {
      it("calls '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.

First let's check that our product renders the data correctly. Define the `product` const in the top `describe` block:

02-testing/completed/src/Home/ProductCard.spec.tsx

```
const product: Product = {
  name: "Product foo",
  price: 55,
  image: "/test.jpg"
}
```

Now let's write the test:

02-testing/completed/src/Home/ProductCard.spec.tsx

```
it("renders correctly", () => {
  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.

Now let's test that if the product is in the cart already, the `Add to cart` button will be disabled:

02-testing/completed/src/Home/ProductCard.spec.tsx

```
describe("when product is in the cart", () => {
  it("the 'Add to cart' button is disabled", () => {
    const mockUseCartHook = () => ({
      addToCart: () => {},
      products: [product]
    })

    const { getByRole } = render(
      <ProductCard
        datum={product}
        useCartHook={mockUseCartHook as any}
      />
    )
    expect(getByRole("button")).toBeDisabled()
  })
})
```

If you look at our `mockUseCartHook` here you'll see that we also had to provide the `addToCart` function. That's because in `ProductCard` props we defined that `useCartHook` returns `products` list and the `addToCart` function:

02-testing/completed/src/Home/ProductCard.tsx

```
export interface ProductCardProps {
  datum: Product
  useCartHook?: () => Pick<
    ReturnType<typeof useCartContext>,
    "products" | "addToCart"
  >
}
```

Note that we've exported the `ProductCartProps` interface. We used it in the `Home` component tests.

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:

02-testing/completed/src/Home/ProductCard.spec.tsx

```
    describe("on 'Add to cart' click", () => {
      it("calls 'addToCart' function", () => {
        const addToCart = jest.fn()
        const mockUseCartHook = () => ({
          addToCart,
          products: []
        })

        const { getByText } = render(
          <ProductCard
            datum={product}
            useCartHook={mockUseCartHook}
          />
        )

        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.

Just like with the home page I decided to add this hook to the props and specify the default value.

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:

02-testing/completed/src/Cart/Cart.spec.tsx

```
import React from "react"

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.

02-testing/completed/src/Cart/Cart.spec.tsx

```
import React from "react"
import { Cart } from "./Cart"
import { fireEvent } from "@testing-library/react"
import { CartItemProps } from "./CartItem"

jest.mock("./CartItem", () => ({
  CartItem: ({ product }: CartItemProps) => {
    const { name, price, image } = product
    return (
      <div>
```

```
        {name} {price} {image}
      </div>
    )
  }
}))

describe("Cart", () => {
  describe("without products", () => {
    const stubCartHook = () => ({
      products: [],
      removeFromCart: () => {},
      totalPrice: () => 0
    })

    it("renders empty cart message", () => {
      const { container } = renderWithRouter(() => (
        <Cart useCartHook={stubCartHook} />
      ))
      expect(container.innerHTML).toMatch(
        "Your cart is empty."
      )
    })

    describe("on 'Back to main page' click", () => {
      it("redirects to '/'", () => {
        const {
          getByText,
          history
        } = renderWithRouter(() => (
          <Cart useCartHook={stubCartHook} />
        ))

        fireEvent.click(getByText("Back to main page."))

        expect(history.location.pathname).toBe("/")
      })
```

```
    })
  })

  describe("with products", () => {
    const products = [
      {
        name: "Product foo",
        price: 100,
        image: "/image/foo_source.png"
      },
      {
        name: "Product bar",
        price: 100,
        image: "/image/bar_source.png"
      }
    ]

    const stubCartHook = () => ({
      products,
      removeFromCart: () => {},
      totalPrice: () => 55
    })

    it("renders cart products list with total price", () => {
      const { container } = renderWithRouter(() => (
        <Cart useCartHook={stubCartHook} />
      ))

      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")
    })
```

```
    describe("on 'go to checkout' click", () => {
      it("redirects to '/checkout'", () => {
        const {
          getByText,
          history
        } = renderWithRouter(() => (
          <Cart useCartHook={stubCartHook} />
        ))

        fireEvent.click(getByText("Go to checkout"))

        expect(history.location.pathname).toBe("/checkout")
      })
    })
  })
})
```

Now let's check that if we click the link, we get redirected to the main page. Now we hardcode the cart value with the empty products array inside the `without products` block:

02-testing/completed/src/Cart/Cart.spec.tsx

```
    const stubCartHook = () => ({
      products: [],
      removeFromCart: () => {},
      totalPrice: () => 0
    })
```

Still inside the products block, write the test that will check that our component will render the `Your cart is empty` message:

02-testing/completed/src/Cart/Cart.spec.tsx

```
    it("renders empty cart message", () => {
      const { container } = renderWithRouter(() => (
        <Cart useCartHook={stubCartHook} />
      ))
      expect(container.innerHTML).toMatch(
        "Your cart is empty."
      )
    })
```

It's time to check that if we click the `Back to main page` button we get redirected to the main page. Right after the `renders empty cart message` test add a new describe block `on 'Back to main page' click` with the following code:

02-testing/completed/src/Cart/Cart.spec.tsx

```
    describe("on 'Back to main page' click", () => {
      it("redirects to '/'", () => {
        const {
          getByText,
          history
        } = renderWithRouter(() => (
          <Cart useCartHook={stubCartHook} />
        ))

        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, hardcode an array of products:

02-testing/completed/src/Cart/Cart.spec.tsx

```
    const products = [
      {
        name: "Product foo",
        price: 100,
        image: "/image/foo_source.png"
      },
      {
        name: "Product bar",
        price: 100,
        image: "/image/bar_source.png"
      }
    ]
```

Define the `cartHook` with these products:

02-testing/completed/src/Cart/Cart.spec.tsx

```
    const stubCartHook = () => ({
      products,
      removeFromCart: () => {},
      totalPrice: () => 55
    })
```

Now let's check if the component will render correctly. We need to make sure that the products are rendered and also that we display the total price.

Before we write the test let's mock the `CartItem` component. Add this code at the beginning of our test file:

02-testing/completed/src/Cart/Cart.spec.tsx

```
jest.mock("./CartItem", () => ({
  CartItem: ({ product }: CartItemProps) => {
    const { name, price, image } = product
    return (
      <div>
        {name} {price} {image}
      </div>
    )
  }
}))
```

Now add this code inside the `renders cart products list with total price` block:

02-testing/completed/src/Cart/Cart.spec.tsx

```
    it("renders cart products list with total price", () => {
      const { container } = renderWithRouter(() => (
        <Cart useCartHook={stubCartHook} />
      ))

      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:

02-testing/completed/src/Cart/Cart.spec.tsx

```
  describe("on 'go to checkout' click", () => {
    it("redirects to '/checkout'", () => {
      const {
        getByText,
        history
      } = renderWithRouter(() => (
        <Cart useCartHook={stubCartHook} />
      ))

      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.

02-testing/completed/src/Cart/CartItem.spec.tsx

```
import React from "react"

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:

02-testing/completed/src/Cart/CartItem.spec.tsx

```
  const product: Product = {
    name: "Product Foo",
    price: 100,
    image: "/image/source.png"
  }
```

Now inside the `renders correctly` block add the following code:

02-testing/completed/src/Cart/CartItem.spec.tsx

```
  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. Add this code inside the `on 'Remove' click` block:

02-testing/completed/src/Cart/CartItem.spec.tsx

```
    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

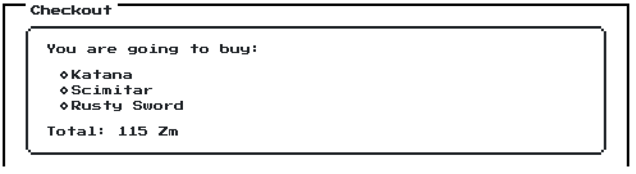
This is the page where the user can input their payment credentials and confirm the order.

Checkout page

We also render the list of products that the user is going to buy here.

Testing CheckoutList

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`:

02-testing/completed/src/Checkout/CheckoutList.spec.tsx

```
import React from "react"
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:

02-testing/completed/src/Checkout/CheckoutList.spec.tsx

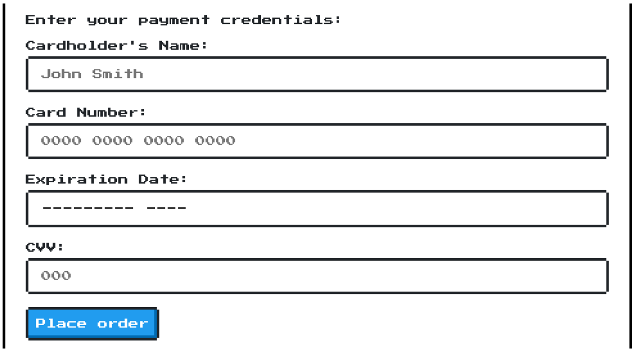
```
  it("renders list of products", () => {
    const products: Product[] = [
      {
        name: "Product foo",
        price: 10,
        image: "/image.png"
      },
      {
        name: "Product bar",
        price: 10,
        image: "/image.png"
      }
    ]

    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 with the following contents:

02-testing/completed/src/Checkout/CheckoutForm.spec.tsx

```
import React from "react"
import { render, fireEvent } from "@testing-library/react"
import { CheckoutForm } from "./CheckoutForm"
import { act } from "react-dom/test-utils"

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. Remove the `todo` part from the `renders correctly` test and add the following code:

02-testing/completed/src/Checkout/CheckoutForm.spec.tsx

```
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 fields we need in this form are present.

Next we need to check that the form will show the errors if we click `Place Order` with invalid values. Add the following test:

02-testing/completed/src/Checkout/CheckoutForm.spec.tsx

```
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.

Now 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:

02-testing/completed/src/Checkout/CheckoutForm.spec.tsx

```
      const { getByLabelText, getByText } = render(
        <CheckoutForm submit={mockSubmit} />
      )
```

And then use it to render our form component:

02-testing/completed/src/Checkout/CheckoutForm.spec.tsx

```
      const mockSubmit = jest.fn()
```

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[77] to manage the inputs. It means that the inputs are controlled[78] 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[79].

Let's fill in the inputs:

02-testing/completed/src/Checkout/CheckoutForm.spec.tsx

```
      await act(async () => {
        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:"), {
```

[77]https://react-hook-form.com/
[78]https://reactjs.org/docs/forms.html#controlled-components
[79]https://reactjs.org/docs/test-utils.html#act

```
          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:

02-testing/completed/src/Checkout/CheckoutForm.spec.tsx

```
      await act(async () => {
        fireEvent.click(getByText("Place order"))
      })
```

Finally we can check that our mock function was called:

02-testing/completed/src/Checkout/CheckoutForm.spec.tsx

```
      expect(mockSubmit).toHaveBeenCalled()
```

Testing FormField

The checkout form uses `FormField` 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`:

02-testing/completed/src/Checkout/FormField.spec.tsx

```
import React from "react"
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:

02-testing/completed/src/Checkout/FormField.spec.tsx

```
  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.

Now let's verify that if we pass an error object to our `FormField`, it will render the error message:

02-testing/completed/src/Checkout/FormField.spec.tsx

```
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:

02-testing/completed/src/Checkout/FormField.spec.tsx

```
    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.

02-testing/completed/src/OrderSummary/OrderSummary.spec.tsx

```
import React from "react"
import { OrderSummary } from "./OrderSummary"

describe("OrderSummary", () => {
  afterEach(jest.clearAllMocks)

  describe("while order data being loaded", () => {
    it("renders loader")
  })

  describe("when order is loaded", () => {
```

```
    it("renders order info")

    it("navigates to main page on button click")
  })

  describe("without order", () => {
    it("renders error message")
  })
})
```

First, let's test that in the loading state we'll render `Loader`. First, let's mock the `Loader` component.

02-testing/completed/src/OrderSummary/OrderSummary.spec.tsx

```
jest.mock("../shared/Loader", () => ({
  Loader: jest.fn(() => null)
}))
```

Here we defined `Loader` using `mock.fn` function. It will allow us to check if it was called, instead of checking the rendered results.

Add this code to `renders loader` block:

02-testing/completed/src/OrderSummary/OrderSummary.spec.tsx

```
  describe("while order data being loaded", () => {
    it("renders loader", () => {
      const stubUseOrder = () => ({
        isLoading: true,
        order: undefined
      })

      render(<OrderSummary useOrderHook={stubUseOrder} />)
      expect(Loader).toHaveBeenCalled()
    })
  })
```

Now let's test that when an order is loaded successfully, we render the products list from it. Hardcode the `useOrder` hook inside the `when order is loaded` block:

02-testing/completed/src/OrderSummary/OrderSummary.spec.tsx

```
    const stubUseOrder = () => ({
      isLoading: false,
      order: {
        products: [
          {
            name: "Product foo",
            price: 10,
            image: "image.png"
          }
        ]
      }
    })
```

Now let's check that it renders correctly. Add the following code:

02-testing/completed/src/OrderSummary/OrderSummary.spec.tsx

```
    it("renders order info", () => {
      const { container } = renderWithRouter(() => (
        <OrderSummary useOrderHook={stubUseOrder} />
      ))

      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:

02-testing/completed/src/OrderSummary/OrderSummary.spec.tsx

```
    it("navigates to main page on button click", () => {
      const {
        getByText,
        history
      } = renderWithRouter(() => (
        <OrderSummary useOrderHook={stubUseOrder} />
      ))

      fireEvent.click(getByText("Back to the store"))

      expect(history.location.pathname).toEqual("/")
    })
```

And finally let's test that if the order data cannot be loaded, we render a failure message:

02-testing/completed/src/OrderSummary/OrderSummary.spec.tsx

```
  describe("without order", () => {
    it("renders error message", () => {
      const stubUseOrder = () => ({
        isLoading: false,
        order: undefined
      })

      const { container } = render(
        <OrderSummary useOrderHook={stubUseOrder} />
      )

      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

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
```

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`:

02-testing/completed/src/Home/useProducts.spec.ts

```
import { renderHook } from "@testing-library/react-hooks"
import { useProducts } from "./useProducts"

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 test that the `useProducts` hook will start fetching data when it is mounted:

02-testing/completed/src/Home/useProducts.spec.ts

```
  it("fetches products on mount", async () => {
    const mockApiGetProducts = jest.fn()

    await act(async () => {
      renderHook(() => useProducts(mockApiGetProducts))
    })

    expect(mockApiGetProducts).toHaveBeenCalled()
  })
```

Here, it comes in very handy that we can just pass the mocked version of the API as an argument.

We render the hook using the `renderHook` method from `@testing-libary/react-hooks` and then we check if the `mockApiGetProducts` function was called.

Let's test the waiting state when the data is being loaded.

02-testing/completed/src/Home/useProducts.spec.ts

```
    it("returns correct loading state data", () => {
      const mockApiGetProducts = jest.fn(
        () => new Promise(() => {})
      )

      const { result } = renderHook(() =>
        useProducts(mockApiGetProducts)
      )
      expect(result.current.isLoading).toEqual(true)
      expect(result.current.error).toEqual(false)
      expect(result.current.categories).toEqual([])
    })
```

Note how we define our `mockApiGetProducts` now:

02-testing/completed/src/Home/useProducts.spec.ts

```
  describe("while waiting API response", () => {
    it("returns correct loading state data", () => {
```

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:

02-testing/completed/src/Home/useProducts.spec.ts

```
  it("returns error state data", async () => {
    const mockApiGetProducts = jest.fn(
      () =>
        new Promise((resolve, reject) => {
          reject("Error")
        })
    )

    const { result, waitForNextUpdate } = renderHook(() =>
      useProducts(mockApiGetProducts)
    )

    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.

02-testing/completed/src/Home/useProducts.spec.ts

```
    const mockApiGetProducts = jest.fn(
      () =>
        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 need to use `act` to wait for the next update before we can test our expectations:

02-testing/completed/src/Home/useProducts.spec.ts

```
    await act(() => waitForNextUpdate())
```

And finally, we can test the happy path, when we successfully get the data and return it from our hook. We are going to add the `returns successful state data` test.

We begin by mocking an API function so that it resolves with products data:

02-testing/completed/src/Home/useProducts.spec.ts

```
    const mockApiGetProducts = jest.fn(
      () =>
        new Promise((resolve, reject) => {
          resolve({
            categories: [{ name: "Category", items: [] }]
          })
        })
    )
```

Then we render our hook and wait for next update, so that the internal state of our hook has the correct value:

02-testing/completed/src/Home/useProducts.spec.ts

```
    const { result, waitForNextUpdate } = renderHook(() =>
      useProducts(mockApiGetProducts)
    )

    await act(() => waitForNextUpdate())
```

And finally we check our expectations:

02-testing/completed/src/Home/useProducts.spec.ts

```
      expect(result.current.isLoading).toEqual(false)
      expect(result.current.error).toEqual(false)
      expect(result.current.categories).toEqual([
        {
          name: "Category",
          items: []
        }
      ])
```

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:

02-testing/completed/src/CartContext/useCart.spec.ts

```
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[80] where function tests are called with a pound sign prefix: `#functionName`.

Let's go through one-by-one. First we need to make sure that when this hook is mounted, it loads the data from `localStorage`. Let's start by mocking the `localStorage`.

Define the `localStorage` constant:

02-testing/completed/src/CartContext/useCart.spec.ts

```
const localStorageMock = (() => {
  let store: { [key: string]: string } = {}
  return {
    clear: () => {
      store = {}
    },
    getItem: (key: string) => {
      return store[key] || null
    },
    removeItem: (key: string) => {
      delete store[key]
    },
    setItem: jest.fn((key: string, value: string) => {
      store[key] = value ? value.toString() : ""
    })
  }
```

Then assign it on the `window` object using `Object.assign` method:

[80] https://rspec.rubystyle.guide/

02-testing/completed/src/CartContext/useCart.spec.ts

```
Object.defineProperty(window, "localStorage", {
  value: localStorageMock
```

One last thing before we move on to the test. Add this clean-up code inside the top-level `describe`:

02-testing/completed/src/CartContext/useCart.spec.ts

```
describe("useCart", () => {
  afterEach(() => {
    localStorageMock.clear()
```

Now we are ready to test that our hook will load its initial state from `localStorage`:

02-testing/completed/src/CartContext/useCart.spec.ts

```
  describe("on mount", () => {
    it("it loads data from 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 we need to make sure that we can add items to the cart:

02-testing/completed/src/CartContext/useCart.spec.ts

```
describe("#addToCart", () => {
  it("adds item to the cart", () => {
    const product: Product = {
      name: "Product foo",
      price: 0,
      image: "image.jpg"
    }
    const { result } = renderHook(useCart)

    act(() => {
      result.current.addToCart(product)
    })

    expect(result.current.products).toEqual([product])
    expect(localStorageMock.setItem).toHaveBeenCalledWith(
      "products",
      JSON.stringify([product])
    )
  })
```

Here we hardcode a `product`, render our hook, then we call the `addToCart` method. Note that as this method will update the state inside our hook, we need to wrap it into `act`. 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:

02-testing/completed/src/CartContext/useCart.spec.ts

```
  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:

02-testing/completed/src/CartContext/useCart.spec.ts

```

```

Now call the `removeFromCart` method. Remember to wrap this call into `act` because it alters the state of the hook:

02-testing/completed/src/CartContext/useCart.spec.ts

```
    act(() => {
      result.current.removeFromCart(product)
```

And finally check the expectations. The `products` array should be empty and `localStorage` should be updated:

02-testing/completed/src/CartContext/useCart.spec.ts

```
      expect(result.current.products).toEqual([])
      expect(localStorageMock.setItem).toHaveBeenCalledWith(
        "products",
        "[]"
```

Let's test the `totalPrice` method. This method should return the sum of prices of all the products located in the cart.

02-testing/completed/src/CartContext/useCart.spec.ts

```
  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`.

02-testing/completed/src/CartContext/useCart.spec.ts

```
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)

    act(() => {
      result.current.clearCart()
    })

    expect(result.current.products).toEqual([])
    expect(localStorageMock.setItem).toHaveBeenCalledWith(
      "products",
      "[]"
```

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.

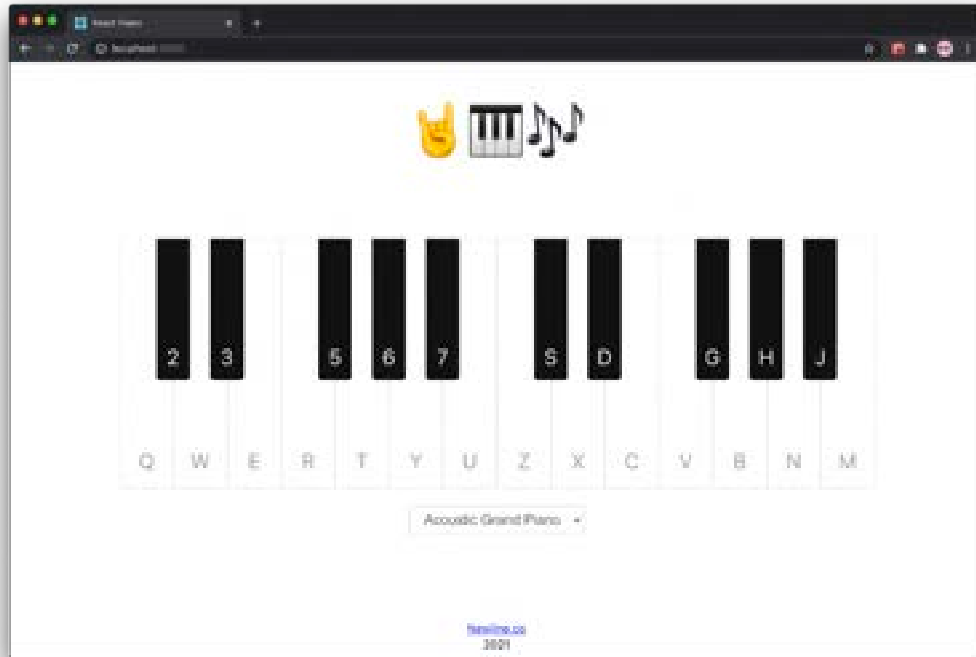
This chapter is going to help you think-in-React by seeing common patterns with specific code.

What We're Going to Build

The application we're going to build is a virtual piano keyboard with a list of instruments that can be played with this keyboard.

We will use a third-party API to generate musical notes and the browser built-in `AudioContext API` to get access to a user's sound hardware. The real computer keyboard will be connected to a virtual one, so that when a user presses 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:



A completed react piano application

A complete code example 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[81] 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, and 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[82] that will provide us with a more flexible API.

Also, to see differences in the app components structure we're going to need a Chrome browser extension called `React Dev Tools`[83]. It will allow us to inspect not only the real DOM of our app but the component tree as well.

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 built of.

---

[81]http://localhost:3000
[82]https://www.npmjs.com/package/soundfont-player
[83]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`. Those components will be wrapped in adapters to either browser API or Soundfont. We will create those wrappers and see why they are called "adapters".

The structure is looking 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're not going to 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 a `Footer` component. It will contain a signature and a current year:

03-react-piano/step-1/src/components/Footer/Footer.tsx

```
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>
  )
}
```

Notice that our component imports a stylesheet, so let's create a file called `Footer.module.css` beside our `Footer.tsx` and fill it up with these styles.

Using CSS Modules and CSS Variables

Wait 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 wrong styles! Check the name for the footer element—there is no way it will collide with any other class on the page:

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.

03-react-piano/step-1/src/components/Footer/Footer.module.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 center and some 5px paddings at each side. 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*[84] (a.k.a CSS variable).

[84] https://developer.mozilla.org/en-US/docs/Web/CSS/--*

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`:

03-react-piano/step-1/src/index.css

```
:root {
  --footer-height: 60px;
  --logo-height: 8rem;
```

The visibility scope of our variable is `:root`. This 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.

Now, let's create a `Logo` component. We will use emojis for our logo. A component's source code will look like this:

03-react-piano/step-1/src/components/Logo/Logo.tsx

```
import styles from "./Logo.module.css"

export const Logo = () => {
  return (
    <h1 className={styles.logo}>
      <span role="img" aria-label="metal hand emoji">
        ▯
      </span>
      <span role="img" aria-label="musical keyboard emoji">
        ▯
      </span>
      <span role="img" aria-label="musical notes emoji">
        ▯
      </span>
    </h1>
  )
}
```

(Unfortunately, we cannot use emojis in the example above, that's why we replaced them with a single symbol of a musical note. 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:

03-react-piano/step-1/src/components/Logo/Logo.module.css

```
.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`[85]. 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.

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:

[85] https://developer.mozilla.org/en-US/docs/Web/CSS/font-size

03-react-piano/step-1/src/App.tsx

```
import React from "react"
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>
  )
}
```

Notice that we write `components/Footer` as an import path for the `Footer` component instead of `components/Footer/Footer.tsx`. This is because we use `index.ts` files in directories or each component to re-export them.

Global Styles

Now, let's finish with global styles which will be applied to the whole project:

03-react-piano/step-1/src/index.css

```
*,
*::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. Also, we declare that the page should have a height of at least 100% of the screen height. Since our keyboard will be placed at the center of the screen, it will be convenient to do that.

Finally, let's style our `App` component to ensure that the `Footer` component will be placed at the bottom of the page, and the `Logo` component at the top.

03-react-piano/step-1/src/App.module.css

```css
.app {
  min-height: 100vh;
}

.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 an `App` component to be placed in the center and the `App` itself to have a minimal height of the page but without `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

In order to understand what we're building, we have to make sure that we understand how music works and what rules apply to a musical keyboard. So before we continue developing our application, let's dive into music theory a little.

First of all, we have to determine how we want to represent musical notes in our application. Nowadays, it is considered standard to use MIDI Notes Numbers[86] for that.

[86] http://www.flutopedia.com/octave_notation.htm

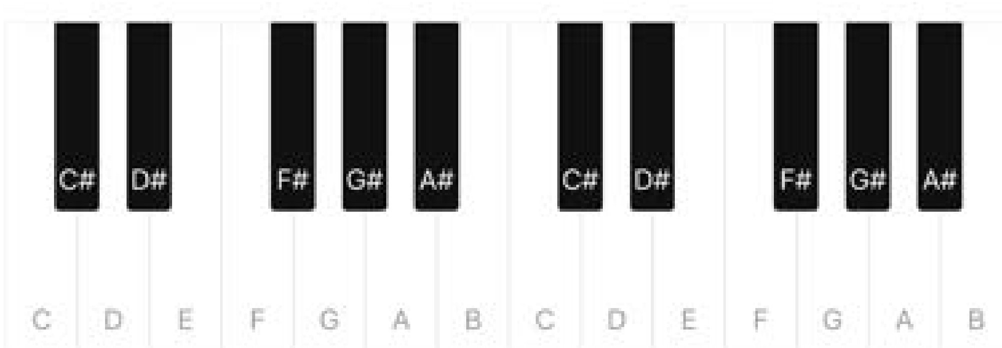
Long story short: a MIDI Note Number is a number that represents a given note in the range from the minus 1st to the 9th octave. An octave is a set of 12 semitones that are different from each other by half of a tone (hence semitone).

Notes in an octave start from C and go up to B like this:

```
C C# D D# E F F# G G# A A# B
```

Sharp (#) is a sign which tells us that a given note is "sharp". There are also "flat" notes, but for simplicity we will focus on and use sharps. A sharp note is a note that is half a step higher than its natural note and half a step lower than the next note. So A# is half a tone higher than A and half a tone lower than B.

On a musical keyboard they would be positioned like this; white keys are naturals and black ones are sharps.



Notes location on a musical keyboard

Coding Music Rules

With all that said, let's try to formalize these rules and express them in TypeScript:

03-react-piano/step-2/src/domain/note.ts

```
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
```

First of all, you may notice `domain` in the file path. Let's talk about this for a second.

In software, the domain[87] is a target subject of a program. This term has roots in domain driven design[88] - the concept of how to structure applications.

In our case, domain refers to sound, note generation, note notation, and real keyboard layout.

Inside the domain directory we'll create a file called `note.ts` - here we describe everything about notes that we want to express in TypeScript.

For example, inside we create a new custom union[89] type called `NoteType`. It will contain all the possible note types that we will use across our app. 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. Despite the fact that we're going to use only sharps it is good practice to keep union types as full as possible to make it clear what can be used in general.

Next, `NotePitch` is a union type which contains all the possible note pitches from A to G. Since the order of items in union is not important we can order our pitches in alphabetic order to make it easier to work with later.

And finally, `OctaveIndex` is a union which contains all the octaves that can be placed on a piano keyboard.

Now, we want to create some type aliases just to make the signatures of our future functions more clear.

[87]https://en.wikipedia.org/wiki/Domain_(software_engineering)
[88]https://en.wikipedia.org/wiki/Domain-driven_design
[89]https://www.typescriptlang.org/docs/handbook/advanced-types.html#union-types

03-react-piano/step-2/src/domain/note.ts

```
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 and represents 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 the first glance, it doesn't look so 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 should those fields have. It is a great tool to use when designing a software system and creating relationships between system parts or modules.

Why not use an interface here? As we discussed earlier, an interface[90] 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[91], 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

[90] https://en.wikipedia.org/wiki/Interface_(computing)
[91] https://www.typescriptlang.org/docs/handbook/interfaces.html

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:

03-react-piano/step-2/src/domain/note.ts

```
export type Note = {
  midi: MidiValue
  type: NoteType

  pitch: NotePitch
  index: PitchIndex
  octave: OctaveIndex
}
```

We describe the shape of a note object which is going to be used later in our code. A `Note` contains five fields, which are:

- `midi` of type `MidiValue` - a number in Octave Notation
- `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 notes in an octave
- `octave` of type `OctaveIndex` - an octave index of a given note

Notice that some fields accept union types. For instance, the field `type` accepts values with the type of `NoteType`. That means that the value for the field `type` can only be one of those described earlier in `NoteType`. So we can only assign `"natural"`, `"sharp"` or `"flat"` to the field `type` and nothing more.

If we try to do that, TypeScript type checker will warn us as follows:

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.

03-react-piano/step-2/src/domain/note.ts

```
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.

03-react-piano/step-2/src/domain/note.ts

```
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.

03-react-piano/step-2/src/domain/note.ts

```
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

You may notice that its type is `Record<PitchIndex, NotePitch>`. Types with "arguments" like this one are called generics[92]. Those are types that allow us to create

[92] https://www.typescriptlang.org/docs/handbook/generics.html

a program component that can work over a variety of 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[93] 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 when we need it.

The first case is when we need to map 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<>`.

The second case when we need `Record<K, T>` is when we don’t know beforehand all the properties and values of a structure but know for sure their types. For example, if we want to add values dynamically.

The `Record<K, T>` type is a so-called utility type. Typescript provides some other utility types[94] as well. Let’s see what some of them do.

`Partial<T>` makes every field on `T` optional:

[93] https://www.typescriptlang.org/docs/handbook/utility-types.html#recordkt
[94] https://www.typescriptlang.org/docs/handbook/utility-types.html

```
type MandatoryFields = {
  a: string
  b: string
}

type OptionalFields = Partial<MandatoryFields>

// It will become:
// type OptionalFields = {
//     a?: string | undefined;
//     b?: string | undefined;
// }
```

`Required<T>` on the other hand, acts opposite. It takes a type and makes every field on it mandatory:

```
type OptionalFields = {
  a?: string
  b?: string
}

type MandatoryFields = Required<OptionalFields>

// It will become:
// type MandatoryFields = {
//     a: string;
//     b: string;
// }
```

Among other utility types[95] there are direct (intrinsic) string manipulations, such as `Uppercase<>`, `Lowercase<>`, `Capitalize<>`, and `Uncapitalize<>`. They are useful when we need to perform a string-like operation on a type:

[95] https://www.typescriptlang.org/docs/handbook/utility-types.html

```
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 which can create a `Note` object from a given `MidiValue`. So let's create it!

03-react-piano/step-2/src/domain/note.ts

```
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 of its octave, and what `pitch` this note is. Finally, we determine which `type` this note is, and return a created note object.

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 a lot of work and make the compilation process of our program much faster[96]. Another advantage is that when we define a return type on a function we make it impossible to unexpectedly return another type. (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.

03-react-piano/step-2/src/domain/note.ts

```
type NotesGeneratorSettings = {
  fromNote?: MidiValue
  toNote?: MidiValue
}

export function generateNotes({
  fromNote = LOWER_NOTE,
  toNote = HIGHER_NOTE
}: NotesGeneratorSettings = {}): Note[] {
  return Array(toNote - fromNote + 1)
    .fill(0)
```

[96] https://github.com/microsoft/TypeScript/wiki/Performance#using-type-annotations

```
    .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 settings object to get access to the `fromNote` and `toNote` fields of that object. If none is given we use an empty object as settings.

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 need for our future keyboard!

Inside of `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`[97].

It uses an `AudioContext` which allows us to handle audio operations such as playing musical tracks, creating oscillators etc. This `AudioContext`[98] has nothing to do with

[97] https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API
[98] https://developer.mozilla.org/en-US/docs/Web/API/AudioContext

`React.Context` that we saw earlier. Those 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.

So we have to ensure that the user's browser supports `AudioContext` and only after that can we continue using it. Let's create a helper function which will check if our browser supports `AudioContext`:

03-react-piano/step-2/src/domain/audio.ts

```
import { Optional } from "./types"

export function accessContext(): Optional<AudioContextType> {
  return window.AudioContext || window.webkitAudioContext || null
}
```

We create a function `accessContext()`, which takes no arguments and returns `Optional<AudioContextType>`. `Optional` is a utility type, which we want to create in `types.ts`:

03-react-piano/step-2/src/domain/types.ts

```
export type Optional<TEntity> = TEntity | null
```

Our `Optional` type is a generic type, which represents a union of a given type `TEntity` or a `null`. Basically we're building an "assumption" type, and will use it when we're not sure if some entity is defined as `TEntity` type or is `null`.

You may notice that we use different notation for defining _type arguments - in this case a slightly more verbose one - we use `TEntity` instead of `T`. This is not mandatory. We will use this only for readability's sake, because later on, when we are building complex interfaces and generic functions, we will need a way to describe what our type arguments are, and what they are for.

This type is useful when we need to make sure that we 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, let's open `react-app-env.d.ts`:

03-react-piano/step-2/src/react-app-env.d.ts

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

type AudioContextType = typeof AudioContext

interface Window extends Window {
  webkitAudioContext: AudioContextType
}
```

Here, we see a triple-slash directive[99] 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`. We need it because `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`.

When `typeof` is 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 a bad practice because it leads to tightly coupled code. However, there is a case when the `typeof` operator can be used except for defining custom types. We can use it in function overloading.

Basically, function overloading is a way to create multiple functions of the same name with different implementations. Like so:

[99] https://www.typescriptlang.org/docs/handbook/triple-slash-directives.html

```
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. However, 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[100] some vendor properties and methods on `window`.

So we have to extend the standard window interface to gain access to this field because in some browsers `AudioContext` is accessible via `AudioContext` property and in some via `webkitAudioContext`.

That is exactly what we cover in our `accessContext()` function! We tell a browser to check if it supports `AudioContext` and use it, or to check if it supports `webkitAudioContext`. If a browser doesn't support either of them, then we want to return `null`, just to be able to determine later that we cannot access `Audio API`.

[100] https://github.com/microsoft/TypeScript/issues/31686

Soundfont

Next, it is time to introduce the third-party API which we're going to use - Soundfont[101]. 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[102] because Soundfont satisfies all of our requirements and weighs less.

Let's start integrating Soundfont with our project.

03-react-piano/step-2/src/domain/sound.ts

```
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 `InstrumentName` union.

Patterns

So far we have been working with our application code and third-party APIs separately. However, in order to combine and use them together we have to connect them.

In programming it is not always easy to connect different software components with each other. The good news is that many of those problems have been solved for us a long time ago. The solutions for typical software development problems are called *patterns*.

[101] https://www.npmjs.com/package/soundfont-player
[102] https://github.com/mudcube/MIDI.js

Adapter or Provider Pattern

An Adapter[103] 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. Basically, it *adapts*[104] (or *provides*) a third-party API for us and makes it usable in our application code.

It is easier to understand an adapter concept with a small example. Let's imagine we have a third-party function, that returns an object of type:

```
type ThirdPartyData = {
  temperature: DegreeFahrenheit;
}
```

Let's say that our app works with Celsius. For this function to work we need 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 own code.

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 to distinguish these from the patterns above, we will call them React-patterns.

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.

[103] https://en.wikipedia.org/wiki/Adapter_pattern
[104] (https://github.com/kamranahmedse/design-patterns-for-humans#-adapter)

Creating a Keyboard

In this section, we're going to create a 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.

Main App Screen

Our main app screen will be in the `Main` component.

03-react-piano/step-3/src/components/Main/Main.tsx

```
import { Keyboard } from "../Keyboard"
import { NoAudioMessage } from "../NoAudioMessage"
import { useAudioContext } from "../AudioContextProvider"

export const Main = () => {
  const AudioContext = useAudioContext()
  return !!AudioContext ? <Keyboard /> : <NoAudioMessage />
}
```

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[105] `useAudioContext()`.

Custom Hook for Accessing Audio

Intentionally, hooks in React let us use state and other features without writing a class. Writing hooks has rules[106] 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.

[105] https://reactjs.org/docs/hooks-intro.html
[106] https://reactjs.org/docs/hooks-rules.html

- We can call hooks only inside functional components.

In our case, we create a hook called `useAudioContext()` which encapsulates an access to `AudioContext`.

03-react-piano/step-3/src/components/AudioContextProvider/useAudioContext.ts

```
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[107] to "remember" the value that our `accessContext()` function is going to return. We can use `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.

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, only renders a message for a user.

[107] https://reactjs.org/docs/hooks-reference.html#useref

03-react-piano/step-3/src/components/NoAudioMessage/NoAudioMessage.tsx

```
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>
  )
}
```

Keyboard Layout

The `Keyboard` component however is a bit more interesting.

03-react-piano/step-3/src/components/Keyboard/Keyboard.tsx

```
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>
  )
}
```

And the styles for it:

03-react-piano/step-3/src/components/Keyboard/Keyboard.module.css

```
.keyboard {
  display: flex;
}
```

Let's start analyzing it with a `notes` array which we `map()` over.

As we remember, it is an array of generated notes from `C4` to `B5`. When mapping each note we destructure it into `midi`, `type`, `index`, and `octave`. For each note we render a `Key` component which we will look at a bit later.

There is a function, however, which 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.

03-react-piano/step-3/src/domain/keyboard.ts

```
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 function selectKey(
  octave: OctaveIndex,
  index: PitchIndex
): Key {
  const keysRow = octave < 5 ? 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()`[108] to create an array of characters from a string. This static method creates a new array from an iterable object.

And finally, `selectKey()` is a function which takes an octave index that we are choosing a key by, and a pitch index to select from the chosen octave. Thus, we select a letter for our key label.

Single Key on a Keyboard

Next, we want to inspect a `Key` component. Let's start with all of the required imports:

03-react-piano/step-3/src/components/Key/Key.tsx

```
import { FunctionComponent } from "react"
import clsx from "clsx"
import { NoteType } from "../../domain/note"
import styles from "./Key.module.css"
```

And the component code:

03-react-piano/step-3/src/components/Key/Key.tsx

```
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"
```

[108]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/from

```
      {...rest}
    >
      {label}
    </button>
  )
}
```

First of all, let's pay attention to the type definition of the component — it is a `FunctionComponent<KeyProps>`.

We could write this without `FunctionComponent` and it would be fine:

```
export const Key = ({ type, label, ...rest }: KeyProps) => /*...*/
```

However, let's try to use `FunctionComponent` as well. First of all, `FunctionComponent` is a generic type[109] from the React package which takes props type as an argument. When using it we can be sure that a compiler understands that this particular component wants a specified props to be provided. It is also useful for autocompletion in an IDE, because when the IDE knows what props a component can have, it can help with suggestions of what we can or must provide when using it.

In our case these argument-props are described with a type `KeyProps`. Inside we define:

- `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

Keep in mind that spread operator (`...rest`) in TypeScript keeps all the information about the types of all the fields in the `rest` object. It knows that this object will contain the `disabled` and `children` fields:

[109] https://www.typescriptlang.org/docs/handbook/jsx.html#function-component

Types of fields on the rest object

We want to use `clsx` package[110] to compose a component's `className` with others in the future.

As a base for our, component we use the `button` element. To ensure that all browsers render our keys more or less equally, we want to reset the default button styles. These styles are placed in the `src/index.css` because they are global.

03-react-piano/step-3/src/index.css

```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;
```

[110] https://www.npmjs.com/package/clsx

```
  -webkit-font-smoothing: inherit;
  -moz-osx-font-smoothing: inherit;
}
```

Here we drop the default styles and make a button look like a text item.

Then, in styles for the `Key` component we describe how the keys should look. The whole stylesheet can be found in `src/components/Key/Key.module.css`. Here we focus only on the difference between black and white keys.

03-react-piano/step-3/src/components/Key/Key.module.css

```
.key {
  position: relative;
  font-size: var(--font-size);
  border-radius: 0 0 var(--radius) var(--radius);
  text-transform: uppercase;
  user-select: none;
}
```

We use `sharp` and `natural` from the `NodeType` union as class modifiers for our styles. Thus, when changing the `type` prop of our `Key` component we automatically change its `className`, and therefore its style.

03-react-piano/step-3/src/components/Key/Key.module.css

```
.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;
}
```

And finally, we add styles for keys when they are pressed:

03-react-piano/step-3/src/components/Key/Key.module.css

```
.natural:active,
.natural.is-active {
  background-color: rgba(0, 0, 0, 0.1);
}

.sharp:active,
.sharp.is-active,
.flat:active,
.flat.is-active {
  background-color: #555;
}
```

And when keys are disabled:

03-react-piano/step-3/src/components/Key/Key.module.css

```
.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;
}
```

Playing a Sound

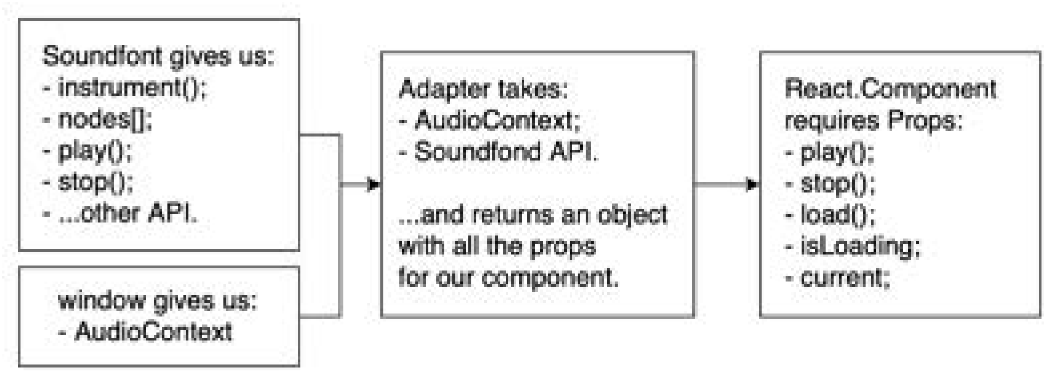
Alright, it seems like everything is ready, so we can actually play some sounds in our app. Before we begin, let's add a new custom type called `SoundfontType` in our `.d.ts`. This is going to be useful when we create an adapter for Soundfont. Add this to `react-app-env.d.ts`:

03-react-piano/step-4/src/react-app-env.d.ts

```
type SoundfontType = typeof Soundfont
```

Soundfont Adapter

Let's examine what we want the adapter to do. It should take what Soundfont provides as a public API, take what `window` gives us, and *adapt* all of that for our usage.



How Soundfont adapter should work

For starters, we create an adapter based on a custom hook, and later on we will use React-Patterns, such as *HOCs* and *Render Props*. For now, just to get to know the Soundfont's API, we use a custom hook. Okay, let's again start with imports:

03-react-piano/step-4/src/adapters/Soundfont/useSoundfont.ts

```
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"
```

Now, let's specify what we need as dependencies and as a result.

03-react-piano/step-4/src/adapters/Soundfont/useSoundfont.ts

```
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>
}

export function useSoundfont({ AudioContext }: Settings): Adapted {
```

Here, a `Settings` type describes what our `useSoundfont()` adapter hook requires as arguments. In our case, we want an `AudioContext` constructor. Then the `Adapted` interface specifies what kind of object we're going to return from our hook.

Why do we use a type for `Settings` and an interface for `Adapted`? Previously we discussed the difference between an interface as a behavior contract and a type as a structure description. Here, we can see that the `Settings` type describes a "shape" of the configuration object. It can't be used as an independent entity, it only represents a data structure for configs.

`Adapted` on the other hand is an entity. It has a state (`loading` flag and a `current` instrument), and most importantly it provides a behavior contract. It guarantees that it provides `load()`, `play()` and `stop()` methods for any entity that tries to communicate with any object that implements the `Adapted` interface.

Let's review this interface in detail. A `loading` field is a `boolean` that is `true` when Soundfont loads the instrument sounds 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 which handle loading the instrument sounds 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 are typed with `Promise<TResult>` generic type. It allows us to comprehend that this function returns a `Promise` of some value, but not the value right away.

Now, let's prepare a local state for our adapter.

03-react-piano/step-4/src/adapters/Soundfont/useSoundfont.ts

```
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 something called `AudioNode`[111] items. Those are general interfaces to handling sound operations. Soundfont uses them to store a state of played notes. Notice that the type of this state part is `AudioNodesRegistry`. This is the type that we create especially for this case in our domain.

03-react-piano/step-4/src/domain/sound.ts

```
import { MidiValue } from "./note"
import { Optional } from "./types"

export type AudioNodesRegistry = Record<MidiValue, Optional<Player>>
```

`AudioNodesRegistry` is a `Record` of `MidiValue` as a key and a `Player` as a value. `Player` type is a type provided by Soundfont, and it is basically an entity that handles for us every musical operation that we want to perform.

Notice that in contrast to other local variables, `activeNodes` is not a part of a local state. That is because we don't want our component to re-render every time audio nodes change their state to avoid extra repaints and also to avoid situations where `.stop()` method is being called on a non-existent node or on a node with an invalid

[111] https://developer.mozilla.org/ru/docs/Web/API/AudioNode

audio state. So, we update this registry directly using a local variable, not using the state.

Next, `current` is a current instrument that is being played. By default we set it to `null` and make it of type `Optional<InstrumentName>`, just because we have to download its sound before we can start playing. A `loading` field indicates whether an instrument is being loaded or not. A `player` is a Soundfont `Player` instance, which helps us perform musical operations.

And finally, `audio` is an `AudioContext` instance. Again, we use `useRef()` hook[112] to keep a reference to an instance of an `AudioContext` that we create when the component mounts. To access this instance we will have to use the `audio.current` property.

Loading Sounds Set

To load an instrument sounds set, we have to implement a `load()` function for our adapter.

03-react-piano/step-4/src/adapters/Soundfont/useSoundfont.ts

```
async function load(
  instrument: InstrumentName = DEFAULT_INSTRUMENT
) {
  setLoading(true)
  const player = await Soundfont.instrument(
    audio.current,
    instrument
  )

  setLoading(false)
  setCurrent(instrument)
  setPlayer(player)
}
```

[112] https://reactjs.org/docs/hooks-reference.html#useref

Notice that we mark this function as `async`. That's because Soundfont's `instrument()` method is async as well. In our `load()` function we take an instrument as an argument and make its default value equal to `DEFAULT_INSTRUMENT`.

First of all, we set the `loading` state to `true` to indicate that the sounds set is being loaded. Then, we call the `await Soundfont.instrument()` method and keep returned result to a `player` local state. Also, we save a given `instrument` as `current` and when everything is done, mark `loading` as `false`.

Now, we have to implement two more functions: `play()` and `stop()`. Let's build them.

03-react-piano/step-4/src/adapters/Soundfont/useSoundfont.ts

```
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 exclamation mark in the `stop()` function is a non-null assertion operator[113]. 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.

Here, we can see a `resume()` function that is being called as a first step of both functions.

[113]https://www.typescriptlang.org/docs/handbook/release-notes/typescript-2-0.html#non-null-assertion-operator

03-react-piano/step-4/src/adapters/Soundfont/useSoundfont.ts

```
  async function resume() {
    return audio.current.state === "suspended"
      ? await audio.current.resume()
      : Promise.resolve()
  }
```

This function checks what state `audio` is in right now. If it is `suspended`[114] that means that `AudioContext` is halting audio hardware access and reducing CPU/battery usage in the process. To continue we have to `resume()` it. And since it also has an `async` interface we have to implement our `resume()` wrapper as async too.

To handle the case when the state of `audio` wasn't `suspended`, we use `Promise.resolve()`[115]. This method returns a `Promise` object that is resolved with a given value. We don't need any value, so we don't pass any as an argument.

Next, in our `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.

There, we have to convert the `note` to string type because player's `play()` method accepts only strings. We can double check that by seeing 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"];
  // ...
};
```

[114] https://developer.mozilla.org/en-US/docs/Web/API/AudioContext/suspend
[115] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/resolve

Then, we save the result node into our `activeNodes` registry. These `activeNodes` are needed to keep track of what notes are being played and be able to `stop()` them. Again, we `resume()` an `AudioContext`, then make sure that a needed node exists and call a `stop()` method on it.

Finally, we need to return all the Soundfont functionality we adapted. We return `loading` state, `current` instrument, and `3` methods for controlling the player `load()`, `play()`, `stop()`. All of this functionality will be used later:

03-react-piano/step-4/src/adapters/Soundfont/useSoundfont.ts

```
  return {
    loading,
    current,

    load,
    play,
    stop
  }
}
```

And that is how we created our first sound provider!

Connecting to a Keyboard

To use our adapter, we have to tweak the props of our `Keyboard` and `Key` components a bit. First, let's look at the keyboard. Again, start with imports:

03-react-piano/step-4/src/components/Keyboard/Keyboard.tsx

```
import { FunctionComponent } from "react"
import { selectKey } from "../../domain/keyboard"
import { notes, MidiValue } from "../../domain/note"
import { Key } from "../Key"
import styles from "./Keyboard.module.css"
```

And the component code:

03-react-piano/step-4/src/components/Keyboard/Keyboard.tsx

```
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>
)
```

Notice that `Keyboard` now has props that will consume `loading`, `play()` and `stop()` that are 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, function 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
```

Also, we use `onDown()` and `onUp()` methods to handle keypress events. Here we create a type alias `PressCallback` which is a function that is called on press event:

03-react-piano/step-4/src/components/Key/Key.tsx

```
type PressCallback = () => void
type KeyProps = {
  type: NoteType
  label: string
  disabled?: boolean

  onUp: PressCallback
  onDown: PressCallback
}
```

Those methods are described now in `KeyProps` and we use them in `onMouseDown()` and `onMouseUp()` props for the `button` element.

03-react-piano/step-4/src/components/Key/Key.tsx

```
    <button
      className={clsx(styles.key, styles[type])}
      onMouseDown={onDown}
      onMouseUp={onUp}
      type="button"
      {...rest}
    >
```

Now we only have to actually connect our `Keyboard` to the Soundfont provider, and we're there!

03-react-piano/step-4/src/components/Keyboard/WithInstrument.tsx

```
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} />
}
```

Here 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`. That is because we know that this component will be rendered 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:

03-react-piano/step-4/src/utils/useMount/useMount.ts

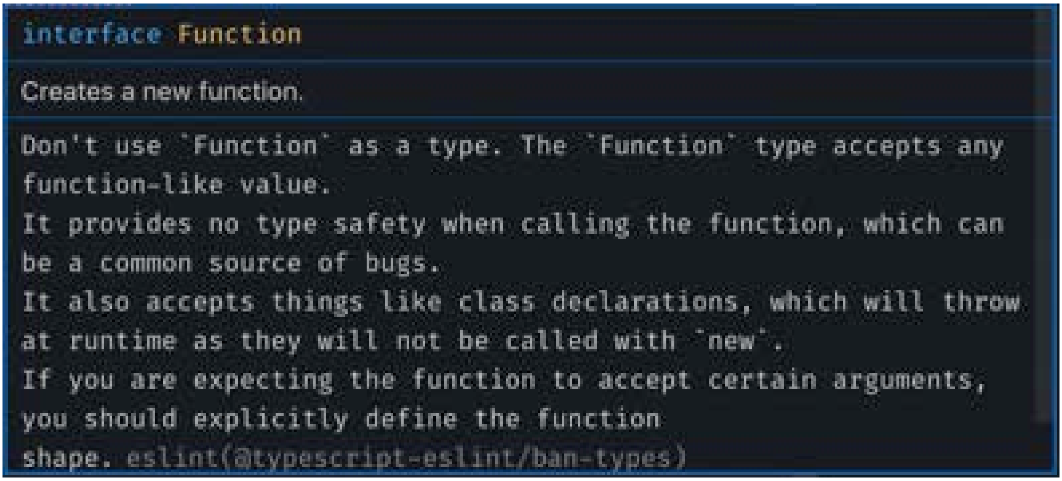
```
import { EffectCallback, useEffect } from "react"

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 ESLint configuration. It will error if we use insecure types in declarations:



ESLint error when using global Function type

Finally, the only thing we have to do is to update our `Main` component to include our connected `KeyboardWithInstrument`. Here we check if `AudioContext` exists by converting it to a boolean with the double negation `!!`. If so, return the keyboard, otherwise, return the fallback message.

03-react-piano/step-4/src/components/Main/Main.tsx

```
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. In order to do that, we need to map real keys with virtual ones, so that when a user presses a key our application would know what to do and what 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.

03-react-piano/step-5/src/components/PressObserver/usePressObserver.ts

```
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.

The hook type signature will look like this:

03-react-piano/step-5/src/components/PressObserver/usePressObserver.ts

```
export function usePressObserver({
  watchKey,
  onStartPress,
  onFinishPress
}: Settings): IsPressed {
  const [pressed, setPressed] = useState<IsPressed>(false)
```

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 `useEffect()`.

03-react-piano/step-5/src/components/PressObserver/usePressObserver.ts

```
}: Settings): IsPressed {
  const [pressed, setPressed] = useState<IsPressed>(false)

  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])

  return pressed
}
```

In 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 `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()`[116] 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 own instance, and thus creates a different `keyPress` event listener. This means that when we press the real key on a keyboard each component will react 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 if it is not the target key, we don't do anything. This way we prevent extra work being done.

This effect uses 2 custom functions called `equal()` and `fromEventCode()`. Let's create them and explain what they do:

[116] https://reactjs.org/docs/hooks-effect.html

03-react-piano/step-5/src/components/PressObserver/usePressObserver.ts

```
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 need 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, just to make sure that there is only one handler for all the key presses. However, it will violate the separation of concerns principle[117], according to which `Key` components should handle their events themselves.

When `usePressObserver()` is ready, we connect it to our `Key` component. Don't forget to import `usePressObserver` into the component.

[117] https://en.wikipedia.org/wiki/Separation_of_concerns

03-react-piano/step-5/src/components/Key/Key.tsx

```
  const pressed = usePressObserver({
    watchKey: label,
    onStartPress: onDown,
    onFinishPress: onUp
  })

  return (
    <button
      className={clsx(
        styles.key,
        styles[type],
        pressed && "is-pressed"
      )}
      onMouseDown={onDown}
      onMouseUp={onUp}
      type="button"
      {...rest}
    >
      {label}
    </button>
  )
```

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 in to *Render Props* and *Higher Order Components* is to create an instruments list to be able to load them dynamically. This part requires a state that will be accessible from many components, so we're going to use `React.Context` to share that state.

Context

Let's start with creating a new `Context`. We will call it `InstrumentContext`.

03-react-piano/step-6/src/state/Instrument/Context.ts

```
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.

The next step is to create an `InstrumentContextProvider` that will provide access to the context.

03-react-piano/step-6/src/state/Instrument/Provider.tsx

```
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 to wrap our entire application in this provider and gain access to the `InstrumentContext` from anywhere.

Instrument Selector

Now, let's try to update a current instrument. To do that, we create a new component called `InstrumentSelector`. Starting with imports:

03-react-piano/step-6/src/components/InstrumentSelector/InstrumentSelector.tsx

```
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:

03-react-piano/step-6/src/components/InstrumentSelector/InstrumentSelector.tsx

```
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", it will navigate us to the type declaration.

Notice how we set the `onChange()` property to have a value of `updateValue()`. That is how we connect our `Context` to a component in the UI. That is where all the changes affect our state.

Later, we render the `select` element filled with the `options` list. We import the `options` list from another file.

03-react-piano/step-6/src/components/InstrumentSelector/options.ts

```
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 is 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` 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.

Now, in order to provide access to our `InstrumentContext`, we have to expose it via `InstrumentContextProvider`.

03-react-piano/step-6/src/components/Playground/Playground.tsx

```
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 actually use `InstrumentContext`: `Keyboard` and `InstrumentSelector`, so we wrap only the two of them into the context provider.

The next thing to do is to update our `Main` component - we want to include and use `Playground` instead of a `Keyboard` that we used previously.

03-react-piano/step-6/src/components/Main/Main.tsx

```
import { Playground } from "../Playground"
import { NoAudioMessage } from "../NoAudioMessage"
import { useAudioContext } from "../AudioContextProvider"

export const Main = () => {
  const AudioContext = useAudioContext()
  return !!AudioContext ? <Playground /> : <NoAudioMessage />
}
```

We're almost there! The only thing to do now is to actually load a new sounds set when changing a current instrument. Let's update our `KeyboardWithInstrument` component to handle this case.

Dynamically Loading Instruments

Again, let's first import all we need:

03-react-piano/step-6/src/components/Keyboard/WithInstrument.tsx

```
import { useEffect } from "react"
import { useInstrument } from "../../state/Instrument"
import { useSoundfont } from "../../adapters/Soundfont"
import { useAudioContext } from "../AudioContextProvider"
import { Keyboard } from "../Keyboard"
```

And create the component itself:

03-react-piano/step-6/src/components/Keyboard/WithInstrument.tsx

```
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 `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 sounds set for this particular instrument.

Notice that we replace `useMount()` hook with `useEffect()` hook. We have to do that since we want to dynamically change our instrument's sounds set, instead of loading it once when mounted.

Also, we check if an instrument has actually changed, and load the new one only if so. For that, we use the `current` value which is 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 `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 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*.

The key idea of this technique is reflected in the title. A component with a render prop[118] takes a function that returns a React element and calls it instead of implementing its own render logic. This technique makes it possible and convenient to share the internal logic of one component with another.

Let's try to imagine how a component with `render` function would look. Its usage would look something like this:

```
<ExampleRenderPropsComponent
  render={(name: string) => <div>Hello, {name}!</div>}
/>
```

If we look closely at `render`, we would notice that it takes a function that returns another React component. However, it does not just render a component, but it renders a component with a text that contains 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:

```
<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.

Now, let's try and build a *Provider* for Soundfont using *Render Props*.

[118] https://reactjs.org/docs/render-props.html

Creating Render Props With Functional Components

There are two ways to create a *Render Props* component: using a functional component and a class. Let's start with functional components first.

First of all, we need to determine what props this component would need to be passed to. Let's add them:

03-react-piano/step-7/src/adapters/Soundfont/SoundfontProvider.ts

```
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 work with. Most importantly, we would require `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. We would describe it like this:

03-react-piano/step-7/src/adapters/Soundfont/SoundfontProvider.ts

```
type ProvidedProps = {
  loading: boolean
  play(note: MidiValue): Promise<void>
  stop(note: MidiValue): Promise<void>
}
```

Basically, those are the same values that we provided earlier with `useSoundfont()` hook, but without `load()` and `current`. We don't need them because we encapsulate the loading of sounds inside of our provider, and a current instrument now is being set from the outside via `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): ReactE\
lement {
  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:

03-react-piano/step-7/src/adapters/Soundfont/SoundfontProvider.ts

```
export const SoundfontProvider: FunctionComponent<ProviderProps> = ({
  AudioContext,
  instrument,
  render
}) => {
```

We explicitly say that this is a `FunctionComponent` that takes `ProviderProps`.

All the work with the internal state would be the same as it was in `useSoundfont()` hook, except that we add loading and reloading sounds when the `instrument` prop is being changed. It will look like this:

03-react-piano/step-7/src/adapters/Soundfont/SoundfontProvider.ts

```
  useEffect(() => {
    if (!loading && instrument !== current) loadInstrument()
  }, [loadInstrument, loading, instrument, current])
```

Here, we use `useEffect()` to capture the moment when an `instrument` prop changes and load a new sounds set for that instrument. However we don't call `load()`

function, instead we call a memoized version[119] of it - this is possible because of the `useCallback()` hook.

You may notice that this is the logic that we implemented in the `KeyboardWithInstrument` component previously, and you would be totally right! This is exactly the same functionality, but now it is encapsulated inside of a provider as well.

Finally, we have to expose our internal values and functions to the outside world. For that we use `render()`:

03-react-piano/step-7/src/adapters/Soundfont/SoundfontProvider.ts

```
  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 the only thing that we have to do for the application to work is tweak the code of the `KeyboardWithInstrument` component a bit.

03-react-piano/step-7/src/components/Keyboard/WithInstrument.tsx

```
export const KeyboardWithInstrument = () => {
  const AudioContext = useAudioContext()!
  const { instrument } = useInstrument()

  return (
    <SoundfontProvider
      AudioContext={AudioContext}
      instrument={instrument}
      render={(props) => <Keyboard {...props} />}
    />
  )
}
```

[119] https://reactjs.org/docs/hooks-reference.html#usecallback

Here we pass the `AudioContext` and an `instrument` as props to `SoundfontProvider` and then pass to `render()` a function that takes `loading`, `play()` and `stop()`, then passes them to a `Keyboard` and returns it. We use object destructuring not to manually enumerate each prop for `Keyboard` 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 declared in the interface it implements:

```
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 need to extend the basic functionality of a class. For example, we can specify a property on an extended class:

```
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, 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 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.

`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.

03-react-piano/step-7/src/adapters/Soundfont/SoundfontProviderClass.ts

```
type ProviderProps = {
  instrument: InstrumentName
  AudioContext: AudioContextType
  render(props: ProvidedProps): ReactElement
}
```

That's because we will use `defaultProps`[120] to use them when nothing will be passed to a component. We will see how they are defined in a minute.

Then, since we are going to use a `class` we need to specify a state type, because the `useState()` hook is not available in class components. Hooks can be used only inside functional components. So, let's introduce the `ProviderState` type.

03-react-piano/step-7/src/adapters/Soundfont/SoundfontProviderClass.ts

```
type ProviderState = {
  loading: boolean
  current: Optional<InstrumentName>
}
```

Here we declare that our local state should contain a `loading` field which is a `boolean` and `current` which is an `Optional<InstrumentName>`. Those are the parts that should cause re-render when changed.

[120] https://www.typescriptlang.org/docs/handbook/release-notes/typescript-3-0.html#support-for-defaultprops-in-jsx

03-react-piano/step-7/src/adapters/Soundfont/SoundfontProviderClass.ts

```
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 `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[121] 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.

[121] https://www.typescriptlang.org/docs/handbook/classes.html#public-private-and-protected-modifiers

03-react-piano/step-7/src/adapters/Soundfont/SoundfontProviderClass.ts

```
public static defaultProps = {
  instrument: DEFAULT_INSTRUMENT
}
```

Then, we create a `constructor()` method. This is the method[122] that is being called right after a class is created.

03-react-piano/step-7/src/adapters/Soundfont/SoundfontProviderClass.ts

```
constructor(props: ProviderProps) {
  super(props)

  const { AudioContext } = this.props
  this.audio = new AudioContext()
}
```

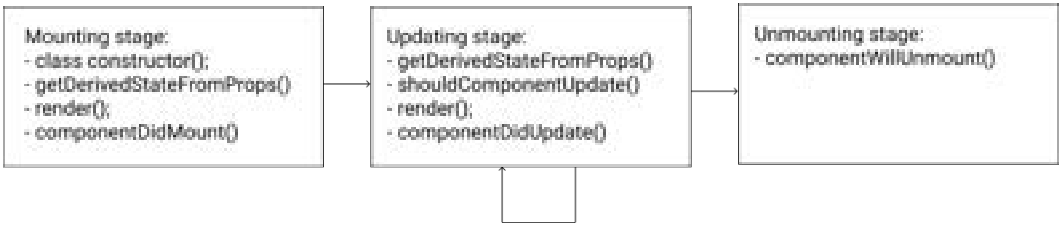
The first thing we have to do is to call[123] `super(props)` method. A `super()` method calls a parent constructor. In order to avoid situations when `this.props` are not assigned to a component until the constructor is finished, we have to assign them via `super(props)`. If we didn't do that we would not be able to access `AudioContext` from `this.props` in a constructor later. Then, we get `AudioContext` and assign `this.audio` to an instance of it.

Sor far, this seems pretty good. Now, let's imagine our component's lifecycle - what should be done when. When a component is created we assign `private` fields. When it's mounted we have to load an initial instrument. When an instrument is changed (a component has been updated) we have to check if the new instrument is different 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 are made and a component is being re-rendered; - unmounting, when a component is being removed from the DOM.

[122] https://www.typescriptlang.org/docs/handbook/classes.html#classes
[123] https://overreacted.io/why-do-we-write-super-props/

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[124] 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

03-react-piano/step-7/src/adapters/Soundfont/SoundfontProviderClass.ts

```
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
```

[124] https://reactjs.org/docs/state-and-lifecycle.html

```
    if (instrument && instrument !== prevInstrument)
      this.load(instrument)
  }
```

That is exactly what we do in those methods. When a component is mounted, we access `instrument` prop and load it using `this.load()`. Before it is going to be updated we check if a current instrument (`this.state.current`) is different from the new one from props, and if so we load it.

Notice that `shouldComponentUpdate()` is not an optimization in this case, but a part of a provider's logic. We use it to prevent infinite reloading of instruments, that could happen because of asynchronous loadings.

Also there is no need to check if an `instrument` is defined or not in `componentDidMount()`, thanks to `defaultProps`.

Now, we have to implement `this.load()` method for loading sounds. We mark is `private` to make it impossible to be used by any other class or object.

03-react-piano/step-7/src/adapters/Soundfont/SoundfontProviderClass.ts

```
  private load = async (instrument: InstrumentName) => {
    this.setState({ loading: true })
    this.player = await Soundfont.instrument(this.audio, instrument)

    this.setState({ loading: false, current: instrument })
  }
```

We are using `this.setState()` to update `loading` flag which will be provided later to a component in `render()`. Also, notice that 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 it to be exposed to the outer world in any way.

There are two other methods now that we need to implement and expose.

03-react-piano/step-7/src/adapters/Soundfont/SoundfontProviderClass.ts

```
  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.

> That is what makes abstractions, custom types, and interfaces so powerful. We can describe an interface (sort of create 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 have to create a `this.resume()` method, which is almost identical to our `resume()` function from the previous adapter.

03-react-piano/step-7/src/adapters/Soundfont/SoundfontProviderClass.ts

```
  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 it as an argument to the object with all the values and methods we promised to provide in `ProvidedProps`.

03-react-piano/step-7/src/adapters/Soundfont/SoundfontProviderClass.ts

```
  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.

Tips and Tricks

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>
```

Caveats

Be careful when using Render Props with `React.PureComponent`[125].

Using a Render Prop can negate the advantage that comes from using `React.PureComponent` if we create the function inside a `render` method. This is because 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 own limitations and usage cases. For *Render Props*, the pros would be that a *Render Props* Provider:

- 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 which uses it
- Needs a render to be called

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.

[125]https://reactjs.org/docs/render-props.html

Higher Order Functions

To grasp on what "order" means, we need to have a look at 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))
  }
}
```

Function `twice()` is a function that takes another *function* as an argument and returns a *function* as a result - that makes it a function with an order *higher than first*.

Basically, any given function that either takes a function as an argument, or returns a function as a result, or both, is a function with order *higher than first*, hence the name -*higher order function*[126].

This kind of function is useful for *composition*. This term[127] 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:

[126]https://en.wikipedia.org/wiki/Higher-order_function
[127]https://en.wikipedia.org/wiki/Function_composition_(computer_science)

```
function incrementTwice(a: number): number {
  return increment(increment(a))
}
```

However, this is not very good. First, we cannot be sure that in the future there won't be a requirement to increment the number three or five times. Also, hardcoded logic is not good in general. Finally, if we zoom into the `twice()` function we can notice similarities with our `incrementTwice()` function.

They both call a function two times in a row, but `incrementTwice()` calls a concrete function (`increment()`), and `twice()` calls an *abstract* function that comes from its argument (`fn()`).

We can try to 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 `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[])`. We need to create this function to prevent calling `fn` right away, since we only want to "prepare" and "remember" which function we want 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 `const anotherIncrementTwice` to that anonymous function that already "remembers" which function we wanted to call twice. It 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 would 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 previously, this function 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 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!
```

Notice that we didn't implement this logic again from scratch. We used a *higher order function* `twice()` to build a more complex function `sayHelloTwice()` from a simple one `sayHello()`.

Higher Order Components carry the same idea but in the realm of React components.

Component as a Higher Order Function

As we said previously, *Higher Order Components* are like *higher order functions* but in the realm of React components. Let's first define a component.

How is it described in official docs[128]? 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 is one that takes a component and returns another one as a result. This is what the official docs tell us[129].

While a component transforms props into UI, a higher-order component transforms a component into another component, enhanced in some way. 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:

03-react-piano/step-8/src/adapters/Soundfont/withInstrument.tsx

```
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"
```

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 is going to be enhanced. `ProviderProps` and `ProviderState` are the exact same as before.

[128] https://reactjs.org/docs/components-and-props.html
[129] https://reactjs.org/docs/higher-order-components.html

03-react-piano/step-8/src/adapters/Soundfont/withInstrument.tsx

```
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 needed to be enhanced. 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.

03-react-piano/step-8/src/adapters/Soundfont/withInstrument.tsx

```
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[130]. We use it to define that `TProps` must include properties that those described in `InjectedProps` type, otherwise, TypeScript should give us an error.

Why use constraints and not just `InjectedProps` right away? We don't always know what props will accept the component that should be enhanced. So if we use `InjectedProps` but the component accepts another prop `soundLevel` it won't be possible to enhance it.

[130] https://www.typescriptlang.org/docs/handbook/generics.html#generic-constraints

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.

Also, notice that by default we define `TProps` to be `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[131] 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.

03-react-piano/step-8/src/adapters/Soundfont/withInstrument.tsx

```
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`.

[131] https://reactjs.org/docs/higher-order-components.html#convention-wrap-the-display-name-for-easy-debugging

03-react-piano/step-8/src/adapters/Soundfont/withInstrument.tsx

```
  return class WithInstrument extends Component<
    ProviderProps,
    ProviderState
  > {
```

Assign a `displayName` to it. We make this field of a `static`[132] class to be able to access it like `WithInstrument.displayName` without creating an instance.

03-react-piano/step-8/src/adapters/Soundfont/withInstrument.tsx

```
    public static displayName = `withInstrument(${displayName})`
```

The rest of the class is the same as it was in `SoundfontProviderClass` from step 7, except the `render()` method.

03-react-piano/step-8/src/adapters/Soundfont/withInstrument.tsx

```
    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 to it, we render our `WrappedComponent` and inject these values and method on it.

[132] https://www.typescriptlang.org/docs/handbook/classes.html#static-properties

Notice that we first spread `this.props` of a component and then `injected` functionality. This is because we don't want any of our injected props to be overridden by someone else afterwards.

Why cast `as TProps` when rendering `WrappedComponent`? Well, there is an issue[133] in TypeScript that erases type of props when using the spread operator (. . .). This forces us to explicitly cast `injected` props to `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 over and over again.

For example, our `withInstrument()` HOC now can be used 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.

Using HOC with Keyboard

When created, our HOC can be used to enhance our `Keyboard` component to connect it to Soundfont. Let's import `withInstrument` and use it to create an enhanced `Keyboard`:

03-react-piano/step-8/src/components/Keyboard/WithInstrument.tsx

```
const WrappedKeyboard = withInstrument(Keyboard)

export const KeyboardWithInstrument = () => {
  const AudioContext = useAudioContext()!
  const { instrument } = useInstrument()

  return (
    <WrappedKeyboard
      AudioContext={AudioContext}
      instrument={instrument}
    />
```

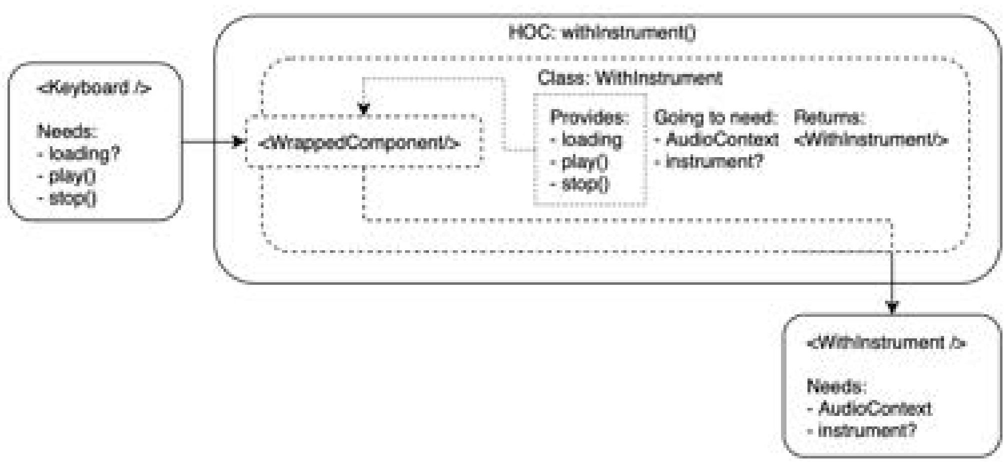
133 https://github.com/Microsoft/TypeScript/issues/28938#issuecomment-450636046

```
  )
}
```

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.

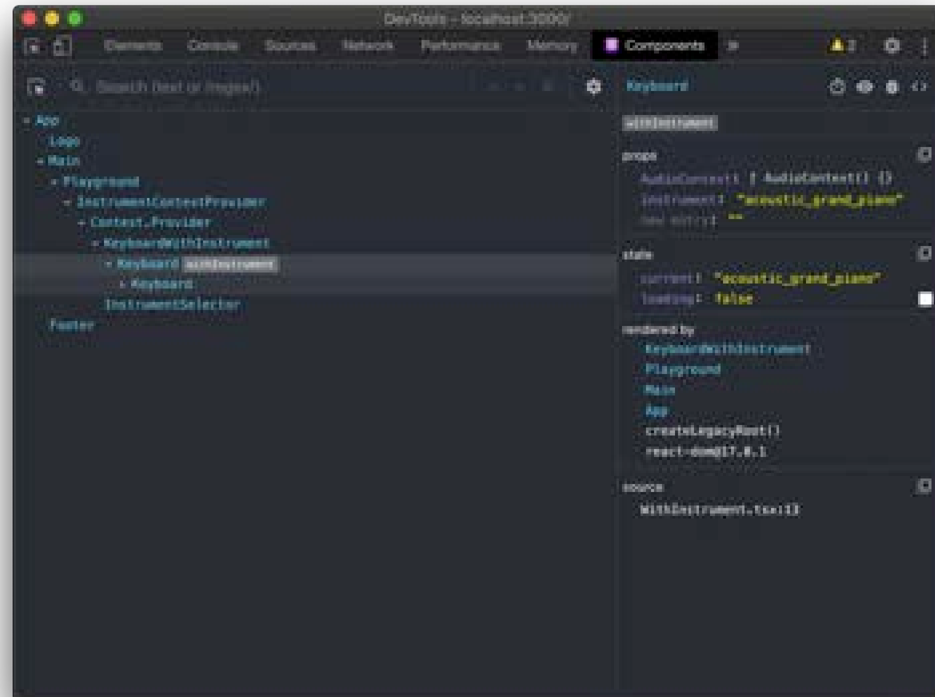
At the moment, when we render `WrappedComponent` it already has `loading`, `play()` and `stop()`, since they have been injected as `InjectedProps` earlier. What it requires is `ProviderProps` that were specified in `Component<ProviderProps, ProviderState>`.



Props flow in HOC

You may notice that this is almost exactly like the example with functions, when `fn` became `increment` and an anonymous function was "remembering" it.

To see what effect the `displayName` has, open the inspector now, find 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 React components. Among others there should be a component `Keyboard withInstrument`:



Component with a display name in the component tree

Try to remove the `displayName` property from the HOC and see what will change in the component tree.

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 for accessing network requests, providing local storage access, subscribing to event streams, or connecting 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.

Caveats

We cannot[134] 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[135] 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.

Passing Refs Through

Refs[136] provide a way to access DOM nodes or React elements created in the render method.

By default, refs aren't passed through[137], and for "true" reusability we have to also consider exposing[138] a ref for our HOC. For that we can use[139] `forwardRef()` function.

The base of our HOC will still be the same. Let's start with imports again:

[134] https://reactjs.org/docs/higher-order-components.html#dont-use-hocs-inside-the-render-method
[135] https://reactjs.org/docs/higher-order-components.html#static-methods-must-be-copied-over
[136] https://reactjs.org/docs/refs-and-the-dom.html
[137] https://reactjs.org/docs/higher-order-components.html#refs-arent-passed-through
[138] https://reactjs.org/docs/forwarding-refs.html
[139] https://github.com/typescript-cheatsheets/react-typescript-cheatsheet/blob/master/HOC.md

03-react-piano/step-8/src/adapters/Soundfont/withInstrumentForwardedRef.tsx

```
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:

03-react-piano/step-8/src/adapters/Soundfont/withInstrumentForwardedRef.tsx

```
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>
}
```

However, we have to declare some “runtime” types inside of `withInstrument()`.

03-react-piano/step-8/src/adapters/Soundfont/withInstrumentForwardedRef.tsx

```
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[140] 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. While `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 which is just an intermediate element and has no real DOM elements. So, in order to be able to provide access to a DOM node, we have to pass a received `ref` further onto an enhanced component which when rendered will result in a DOM node.

Later, we declare a class `WithInstrument` as a `Component` of `WithForwardRef` props and `ProviderState`.

03-react-piano/step-8/src/adapters/Soundfont/withInstrumentForwardedRef.tsx

```
  const displayName =
    WrappedComponent.displayName ||
    WrappedComponent.name ||
    "Component"

  class WithInstrument extends Component<
    WithForwardedRef,
    ProviderState
  > {
```

In `render()` method, we access `forwardedRef` from props and pass it as `ref` props onto a `WrappedComponent`.

[140] https://www.typescriptlang.org/docs/handbook/utility-types.html#instancetypet

03-react-piano/step-8/src/adapters/Soundfont/withInstrumentForwardedRef.tsx

```
    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.

03-react-piano/step-8/src/adapters/Soundfont/withInstrumentForwardedRef.tsx

```
  return forwardRef<ComponentInstance, ProviderProps>(
    (props, ref) => <WithInstrument forwardedRef={ref} {...props} />
  )
```

This is because by default refs are not provided as all other props. In order to get access to a `ref`, we have to call a special `forwardRef()` function.

As an argument for it, we provide another anonymous function which returns our `WithInstrument` component. Notice that 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.

03-react-piano/step-8/src/adapters/Soundfont/withInstrumentStatic.tsx

```
export function withInstrumentStatic<
  TProps extends InjectedProps = InjectedProps
>(initialInstrument: InstrumentName = DEFAULT_INSTRUMENT) {
```

Here we create a function `withInstrumentStatic()` which takes an `instrument` as an argument. This is the instrument that our provider will load - it won't change through the whole component life.

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`.

03-react-piano/step-8/src/adapters/Soundfont/withInstrumentStatic.tsx

```
  return function enhanceComponent(
    WrappedComponent: ComponentType<TProps>
  ) {
    const displayName =
      WrappedComponent.displayName ||
      WrappedComponent.name ||
      "Component"

    return class WithInstrument extends Component<
```

```
      ProviderProps,
      ProviderState
    > {
```

Why would we create a function that returns a function that returns a class?.. Well, to answer this question we have to look at a use case for this HOC.

03-react-piano/step-8/src/components/Keyboard/WithStaticInstrument.tsx

```
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 `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!

From Hooks to HOCs

Since HOCs are just functions that return components, we reckon that they can be based on hooks as well. Let's import required modules and define the types:

03-react-piano/step-8/src/adapters/Soundfont/withInstrumentBasedOnHook.tsx

```
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:

03-react-piano/step-8/src/adapters/Soundfont/withInstrumentBasedOnHook.tsx

```
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} />
    )
```

```
  }
}
```

Again, we encapsulate the loading of sound sets 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.

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, it can be done 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 of them and it is not clear what is going to 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.

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[141]. 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.

What Are We Building?

The application for this chapter is a drawing board.

[141] 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.

Preview The Final Result

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
```

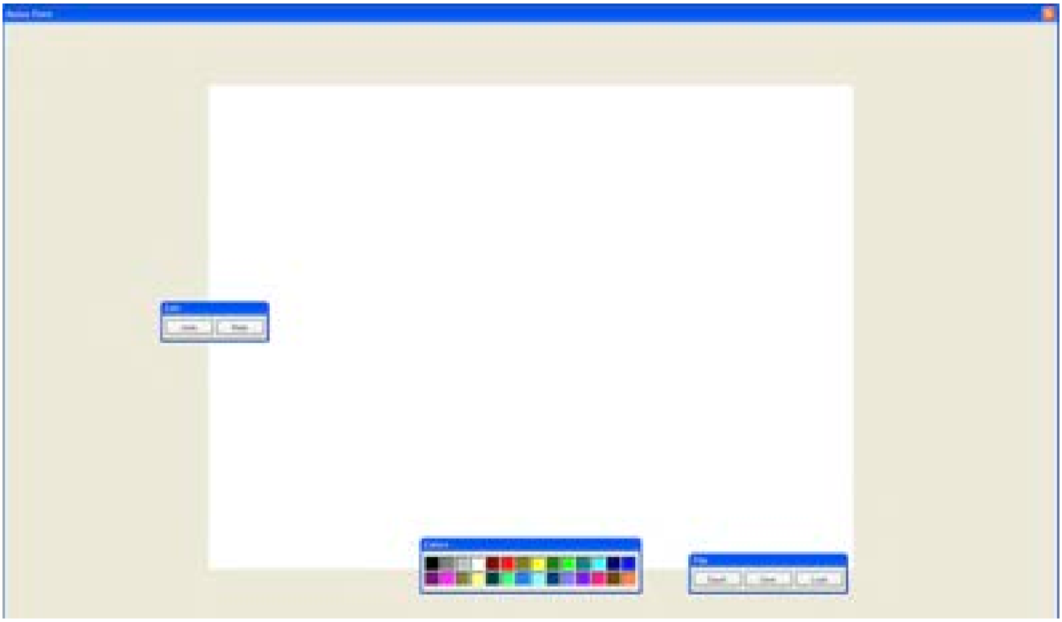
The `yarn dev` command will launch the app along with the backend script.

It should also open the app in the browser. If it doesn't, navigate to http://localhost:3000[142]

[142]http://localhost:3000

and open it manually.

You should see an empty canvas and a color palette.

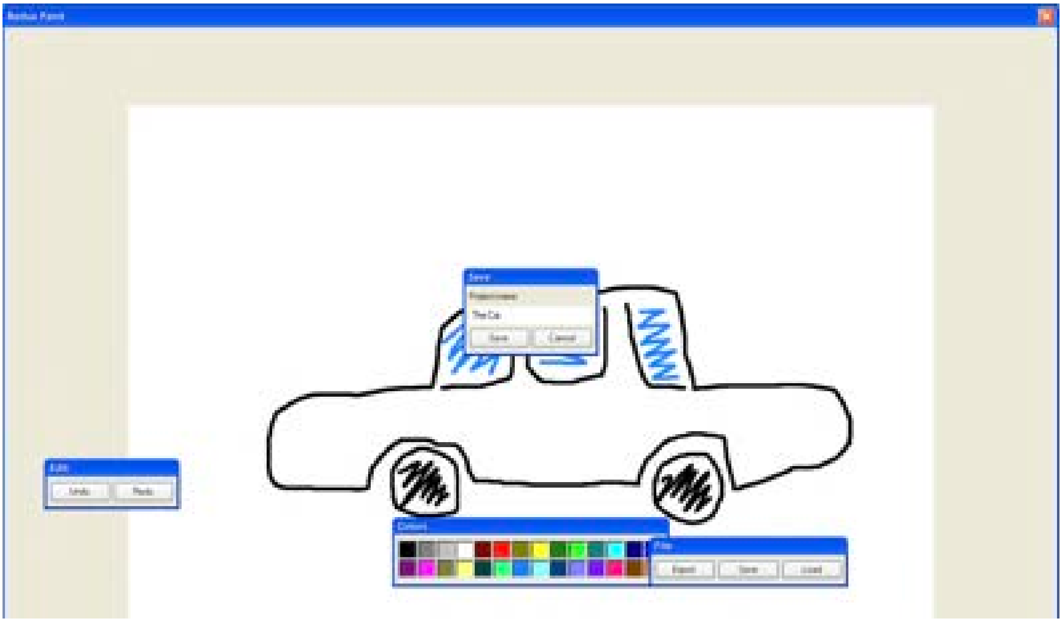


Empty canvas

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.

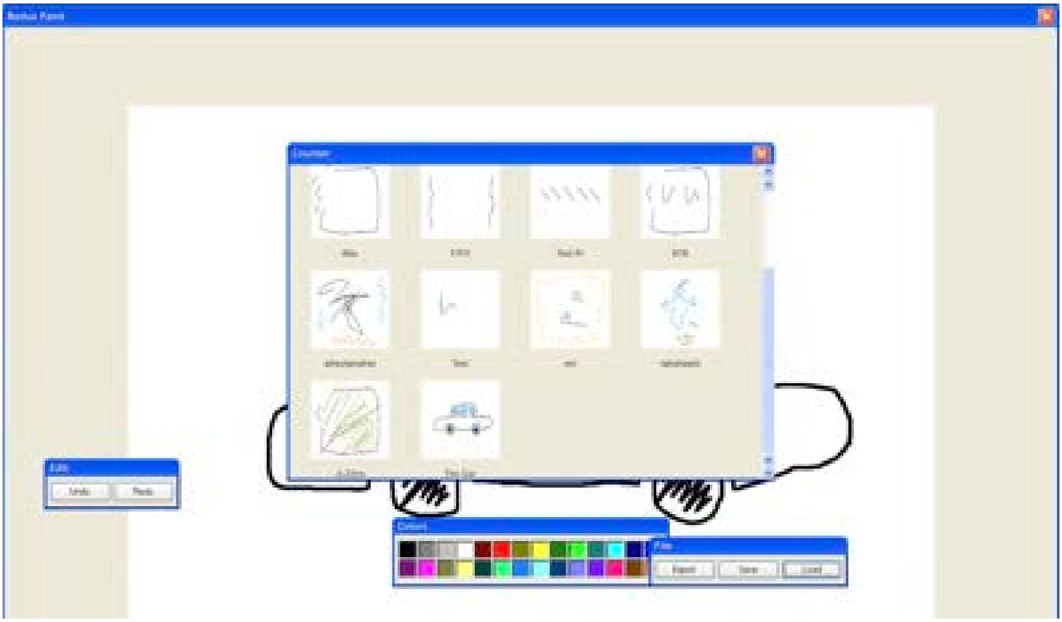


Saving the project

Pick a name for your project and press the *Save* button.

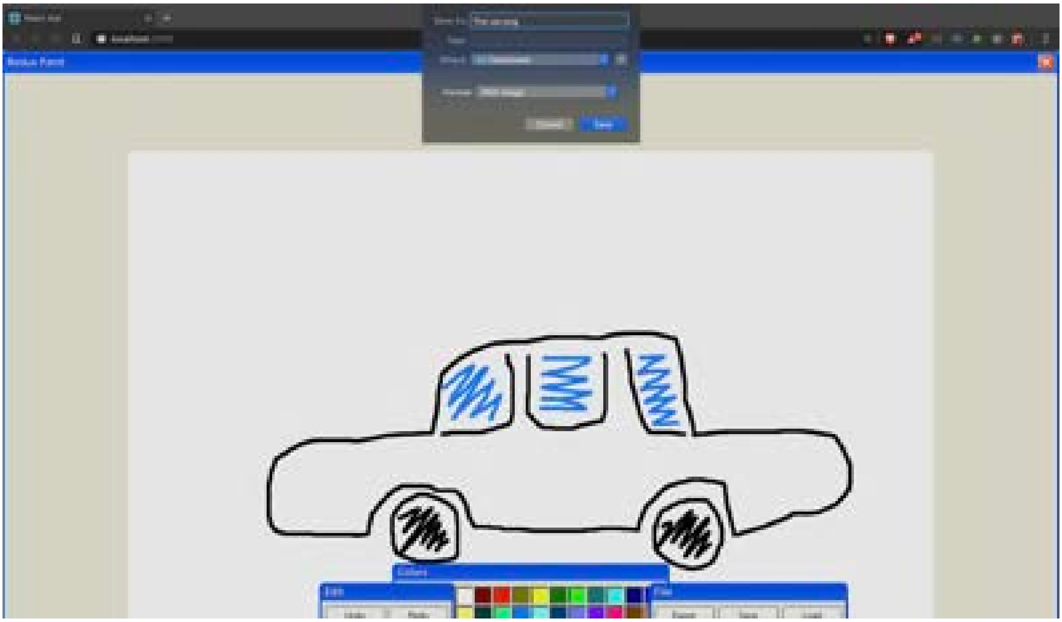
Now you can load this project and continue drawing. The changes in history will be preserved.

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.

04-redux/redux-example/index.ts

```
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.

04-redux/redux-example/index.ts

```
const store = createStore(reducer, "Initial Value")
```

The store provides a `subscribe` method that allows us to subscribe to the store updates.

04-redux/redux-example/index.ts

```
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:

04-redux/redux-example/index.ts

```
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[143]

Usually, instead of creating actions in place, people define action creator functions:

04-redux/redux-example/index.ts

```
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.

[143] 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[144] 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[145] to handle that. Also, there are tools like Redux Thunk[146] 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.

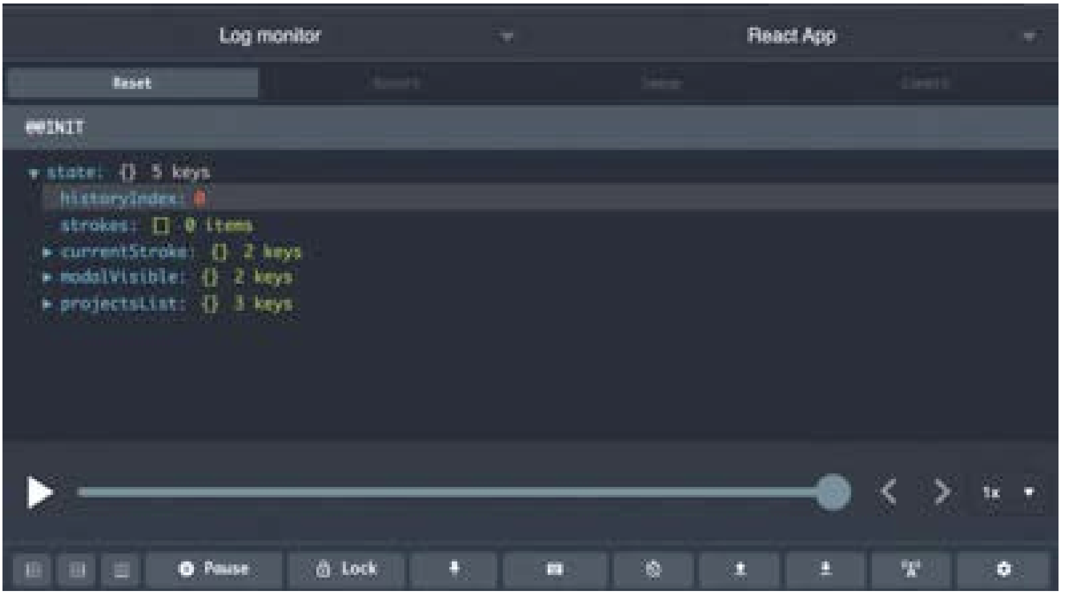
Patterns and Code Organization. Redux is well-studied and there are recipes for most of the problems. There is a methodology called Ducks[147] that you can use to organize the Redux code.

[144] https://github.com/reduxjs/redux-devtools
[145] https://redux.js.org/advanced/middleware
[146] https://github.com/reduxjs/redux-thunk
[147] 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[148] and Firefox[149].

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:

[148] https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd
[149] 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:

04-redux/step1/src/rootReducer.ts

```
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.

04-redux/step1/src/store.ts

```
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.[150]

Then go to `src/index.tsx` and import `Provider` from `react-redux`:

04-redux/step1/src/index.tsx

```
import {Provider} from 'react-redux'
```

Wrap your `App` component into the `Provider`:

04-redux/step1/src/index.tsx

```
ReactDOM.render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>,
  document.getElementById('root')
);
```

[150] https://redux.js.org/advanced/middleware

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:

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:

04-redux/step1/src/store.ts

```
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.

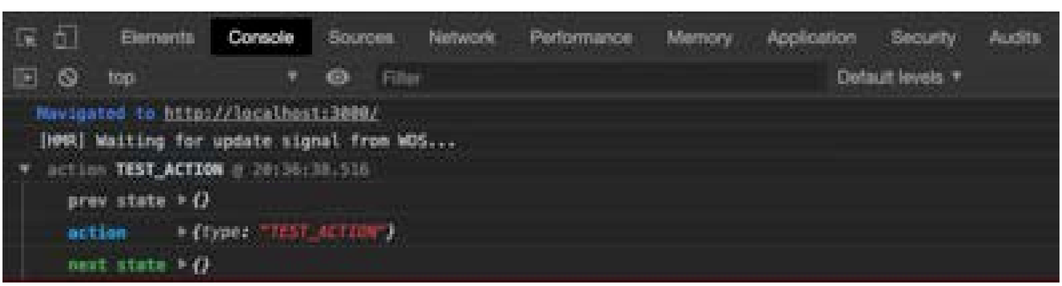
> Read more about applying middlewares to your Redux store in the Redux Documentation[151]

Temporarily add the following code to dispatch an action:

04-redux/step1/src/store.ts

```
store.dispatch({type: "TEST_ACTION"})
```

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:

[151] 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[152] by Adam Hammad[153] for our styles.

Install it:

```
yarn add xp.css
```

And import it in `src/index.css`:

04-redux/step1/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.

Working With Canvas

We will use the Canvas API[154] to handle drawing.

We will need to render the canvas and get a reference to it. Add the following code in `src/App.tsx`:

[152] https://botoxparty.github.io/XP.css/
[153] https://github.com/botoxparty
[154] https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API

04-redux/step1/src/App.tsx

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

function App() {
  const canvasRef = useRef<HTMLCanvasElement>(null)

  return <canvas ref={canvasRef} />
}

export default App
```

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`.

Handling Canvas Events

We need 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

Add the following event handlers:

04-redux/step1/src/App.tsx

```
function App() {
  const canvasRef = useRef<HTMLCanvasElement>(null)

  const startDrawing = () => { }
  const endDrawing = () => { }
  const draw = () => { }

  return (
    <canvas
      onMouseDown={startDrawing}
      onMouseUp={endDrawing}
      onMouseOut={endDrawing}
      onMouseMove={draw}
      ref={canvasRef}
    />
  )
}
```

Every time the user presses, moves, or releases the mouse, we'll dispatch an action.

For example, we will dispatch a `MOUSE_MOVE` action inside the `draw` callback. This action will save new points in the store.

In this component, we will subscribe to the store changes and draw on the `canvas` each time the state is updated.

Before we can do this, we need to define our state.

Define The Store Types

Create a new file `src/types.d.ts`.

> In typescript `*.d.ts` files are used to contain the types declarations exclusively. You can import types from such files just like you import values from the regular modules.

Inside this file let's define the type for our state:

04-redux/step1/src/type.d.ts

```
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:

04-redux/step1/src/type.d.ts

```
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:

04-redux/step1/src/type.d.ts

```
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:

04-redux/step1/src/actions.ts

```
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/types.d.ts`:

04-redux/step1/src/actions.ts

```
import { Point } from "./types"
```

Define the `Action` type:

04-redux/step1/src/actions.ts

```
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:

04-redux/step1/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 endStroke = () => {
  return { type: END_STROKE }
}
```

Add The Reducer Logic

Go to `src/rootReducer.ts`. Import the `RootState` from `src/types.d.ts` and the `Action` type from the `src/actions.ts`.

04-redux/step1/src/rootReducer.ts

```
import { RootState } from './types'
import { Action } from './actions'
```

Then we need to define the initial state:

04-redux/step1/src/rootReducer.ts

```
const initialState: RootState = {
  currentStroke: { points: [], color: "#000" },
  strokes: [],
  historyIndex: 0
}
```

Remake the `rootReducer` to this:

04-redux/step1/src/rootReducer.ts

```
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`:

04-redux/step1/src/rootReducer.ts

```
    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 we need to process the `UPDATE_STROKE` action:

04-redux/step1/src/rootReducer.ts

```
    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[155]. 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`:

[155] https://redux.js.org/recipes/structuring-reducers/immutable-update-patterns

04-redux/step1/src/rootReducer.ts

```
    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.

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.

Define The First Selector

When you work with Redux, it is a good idea to separate the data retrieval logic from the rendering logic. This way your components won't depend on the form of your state. It will allow you to refactor your application more easily.

This separation is achieved using selectors.

Selectors are functions that accept the `state` as an argument and then return some specific value from it.

Let's define our first selector.

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

04-redux/step1/src/selectors.ts

```
import { RootState } from "./types";

export const currentStrokeSelector = (state: RootState) => state.curren\
tStroke
```

This selector returns an array of points of the current stroke.

Use The Selector

Go to `src/App.tsx`. Import the `useSelector` hook from `react-redux` and the `currentStrokeSelec` from the `src/selectors.ts`:

04-redux/step1/src/App.tsx

```
import { useSelector } from "react-redux"
import { currentStrokeSelector } from './selectors'
```

Get the `currentStroke` value from the state. Add this code after the `canvasRef` definition:

04-redux/step1/src/App.tsx

```
  const currentStroke = useSelector(currentStrokeSelector)
```

Now our component will be re-rendered every time the `currentStroke` gets updated.

Dispatch Actions

Still in `src/App.tsx`, import the `useDispatch` from `react-redux`:

04-redux/step1/src/App.tsx

```
import { useSelector, useDispatch } from "react-redux"
```

Get the `dispatch` function from the `useDispatch` - add this line after the `useSelector` call:

04-redux/step1/src/App.tsx

```
  const dispatch = useDispatch()
```

Now let's edit the mouse press event handler. Make it dispatch the `BEGIN_STROKE` action.

04-redux/step1/src/App.tsx

```
  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[156] 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:

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

04-redux/step1/src/App.tsx

```
const draw = ({
  nativeEvent
}: React.MouseEvent<HTMLCanvasElement>) => {
  if (!isDrawing) {
    return
  }
  const { offsetX, offsetY } = nativeEvent

  dispatch(updateStroke(offsetX, offsetY))
}
```

Here we need to check that the mouse is pressed - this is why we check the `isDrawing` flag.

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 `currentStroke` selector:

04-redux/step1/src/App.tsx

```
const isDrawing = !!currentStroke.points.length
```

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:

04-redux/step1/src/App.tsx

```
const endDrawing = () => {
  if (isDrawing) {
    dispatch(endStroke())
  }
}
```

Here we dispatch the `END_STROKE` action.

The `endDrawing` function will also trigger when the mouse leaves the canvas area. This is why we check the `isDrawing` flag to dispatch the `endStroke` action only if we were drawing the stroke.

Draw The Current Stroke

We dispatch the actions to update the state when we interact with the canvas.

The actions will trigger the state updates.

Now let's observe the state and render the strokes on the canvas.

To draw on the canvas we need to get the canvas drawing context. Let's define a function that will get the context from the canvas reference.

Below the `isDrawing` flag, define the `getCanvasWithContext` function:

04-redux/step1/src/App.tsx

```
const getCanvasWithContext = (canvas = canvasRef.current) => {
  return { canvas, context: canvas?.getContext("2d") }
}
```

This function will return both the canvas and its `2d` drawing context.

Still in the `src/App.tsx`, define a side-effect to handle the `currentStroke` updates.

04-redux/step1/src/App.tsx

```
  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 color.

Let's define the `drawStroke` method in a separate module. Create a new file `src/canvasUtils` and import `Point` from the `types` module:

04-redux/step1/src/canvasUtils.ts

```
import { Point } from "./types"
```

Now define and export the `drawStroke` method:

04-redux/step1/src/canvasUtils.ts

```
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.

First, 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.

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. 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:

04-redux/step2/src/actions.ts

```
export const SET_STROKE_COLOR = "SET_STROKE_COLOR"
```

Expand the `Action` type definition with this block:

04-redux/step2/src/actions.ts

```
  | {
      type: typeof SET_STROKE_COLOR
      payload: string
    }
```

And then add a new action creator:

04-redux/step2/src/actions.ts

```
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 add a new reducer block:

04-redux/step2/src/rootReducer.ts

```
    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/ColorPanel.tsx`. First we need to import `React`, `useDispatch`, and `setStrokeColor` action:

04-redux/step2/src/ColorPanel.tsx

```
import React from "react"
import { useDispatch } from "react-redux"
import { setStrokeColor } from "./actions"
```

Define the list of colors:

04-redux/step2/src/ColorPanel.tsx

```
const COLORS = [
  "#000000",
  "#808080",
  "#c0c0c0",
  "#ffffff",
  "#800000",
  //... Full list in completed example
]
```

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:

04-redux/step2/src/ColorPanel.tsx

```
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:

04-redux/step2/src/ColorPanel.tsx

```
  const dispatch = useDispatch()

  const onColorChange = (color: string) => {
    dispatch(setStrokeColor(color))
  }
```

Then go to `src/App.tsx` and add the `ColorPanel` to the layout.

04-redux/step2/src/App.tsx

```
      <ColorPanel />
      <canvas
        onMouseDown={startDrawing}
        onMouseUp={endDrawing}
        onMouseOut={endDrawing}
        onMouseMove={draw}
        ref={canvasRef}
      />
```

Add it right above the `canvas` element.

Launch the app.

Picking the colors

You should now be able to select colors.

Implement Undo and Redo

Now let's implement the *undo* functionality.

First, let's add the **Undo** and **Redo** buttons.

Create a new file `src/EditPanel.tsx`. Import `React`, `useDispatch` and undo/redo actions:

04-redux/step3/src/EditPanel.tsx

```
import React from "react"
import { useDispatch } from "react-redux"
import { undo, redo } from "./actions"
```

and define the `EditPanel` component there:

04-redux/step3/src/EditPanel.tsx

```
export const EditPanel = () => {
  return (
    <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"
          >
            Undo
          </button>
          <button
            className="button undo"
          >
            Redo
          </button>
        </div>
      </div>
    </div>
  )
}
```

Get the `dispatch` function using the `useDispatch` hook from `react-redux`.

04-redux/step3/src/EditPanel.tsx

```
  const dispatch = useDispatch()
```

Add this line right above the component layout.

Then add event listeners to the buttons and dispatch the `UNDO` and `REDO` actions:

04-redux/step3/src/EditPanel.tsx

```
          <button
            className="button redo"
            onClick={() => dispatch(undo())}
          >
            Undo
          </button>
          <button
            className="button undo"
            onClick={() => dispatch(redo())}
          >
            Redo
          </button>
```

Now go to `src/App.tsx`.

Add the `EditPanel` to the layout:

04-redux/step3/src/App.tsx

```
      <EditPanel/>
      <ColorPanel />
      <canvas
        onMouseDown={startDrawing}
        onMouseUp={endDrawing}
        onMouseOut={endDrawing}
        onMouseMove={draw}
        ref={canvasRef}
      />
```

The new element should be right above the `ColorPanel`.

We also need to redraw the screen when we undo or redo the strokes.

Add a new `useEffect` block:

04-redux/step3/src/App.tsx

```
  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)
      })
    })
```

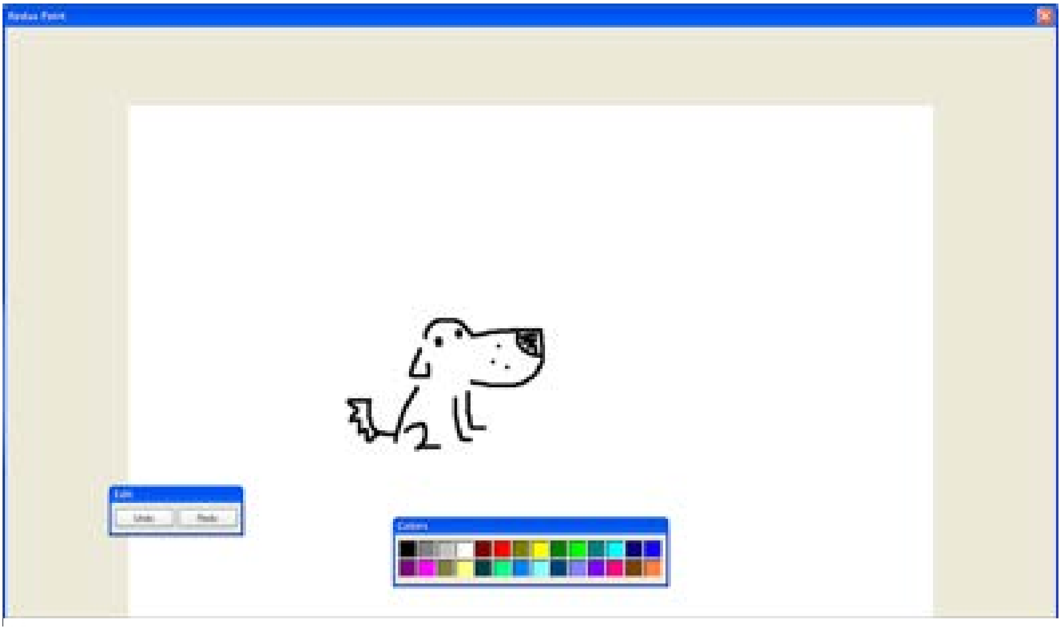
Every time the `historyIndex` gets updated we clear the screen and then draw only the strokes that weren't undone.

Open `src/canvasUtils.ts` and add the `clearCanvas` method:

04-redux/step3/src/canvasUtils.ts

```
export const clearCanvas = (canvas: HTMLCanvasElement) => {
  const context = canvas.getContext("2d")
  if (!context) {
    return
  }
  context.fillStyle = "white"
  context.fillRect(0, 0, canvas.width, canvas.height)
}
```

To clear the canvas we set the fill color to white and draw the rectangle the size of the canvas.



Redux Paint with undo and redo

Launch your app. You should now be able to undo and redo the strokes.

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 `currentStroke` 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.

04-redux/step4/src/modules/historyIndex/actions.ts

```
import { Stroke } from "../../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; historyLimit: number }
    }

export const undo = (undoLimit: number) => {
  return { type: UNDO, payload: undoLimit }
}

export const redo = () => {
  return { type: REDO }
}
```

Create a new file `src/modules/historyIndex/reducer.ts`. Import the actions and the `RootState` type:

04-redux/step4/src/modules/historyIndex/reducer.ts

```
import { RootState } from "../../types"
import { HistoryIndexAction, UNDO, REDO, END_STROKE } from "./actions"
```

Now define the reducer with the following contents:

04-redux/step4/src/modules/historyIndex/reducer.ts

```
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 a new file `src/modules/historyIndex/selectors.ts`.

04-redux/step4/src/modules/historyIndex/selectors.ts

```
import { RootState } from "../../types";

export const historyIndexSelector = (state: RootState) => state.history\
Index
```

Separate The Current Stroke

Create a new folder `src/modules/currentStroke`.

Create a new file `src/modules/currentStroke/actions.ts`. Import the `Point` and `Stroke` types:

04-redux/step4/src/modules/currentStroke/actions.ts

```
import { Point, Stroke } from "../../types"
```

Move the `BEGIN_STROKE`, `UPDATE_STROKE`, and `SET_STROKE_COLOR` types there.

04-redux/step4/src/modules/currentStroke/actions.ts

```
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:

04-redux/step4/src/modules/currentStroke/actions.ts

```
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; historyLimit: number }
    }
```

And finally, move the action creators from the `src/actions.ts` file to it.

04-redux/step4/src/modules/currentStroke/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 = (historyLimit: number, stroke: Stroke) => {
```

```
  return { type: END_STROKE, payload: { historyLimit, stroke } }
}
```

Create a new file `src/modules/currentStroke/reducer.ts`.

Import the actions and the root state type:

04-redux/step4/src/modules/currentStroke/reducer.ts

```
import {
  Action,
  UPDATE_STROKE,
  BEGIN_STROKE,
  END_STROKE,
  SET_STROKE_COLOR,
} from "./actions"
import { RootState } from "../../types"
```

Define the initial state:

04-redux/step4/src/modules/currentStroke/reducer.ts

```
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.

04-redux/step4/src/modules/currentStroke/reducer.ts

```
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/reducer.ts` to `src/modules/currentStroke/selecto`

04-redux/step4/src/modules/currentStroke/selectors.ts

```
import { RootState } from "../../types";

export const currentStrokeSelector = (state: RootState) => state.curren\
tStroke
```

Separate The Strokes List

Create a new folder `src/modules/strokes`.

Then create the `src/modules/strokes/actions.ts` file and add the `END_STROKE` action type and action creator there:

04-redux/step4/src/modules/strokes/actions.ts

```
import { Stroke } from "../../types"

export const END_STROKE = "END_STROKE"

export type Action = {
  type: typeof END_STROKE
  payload: { stroke: Stroke; historyLimit: number }
}

export type HistoryIndexAction = {
  type: typeof END_STROKE
  payload: { stroke: Stroke; historyLimit: number }
}

export const endStroke = (historyLimit: number, stroke: Stroke) => {
  return { type: END_STROKE, payload: { historyLimit, stroke } }
}
```

Create a new file `src/modules/strokes/reducer.ts`.

Add the `END_STROKE` action handler from our root reducer to this file.

04-redux/step4/src/modules/strokes/reducer.ts

```
import { RootState } from "../../types"
import { Action, END_STROKE } from "./actions"

export const reducer = (
  state: RootState["strokes"] = [],
  action: Action
) => {
  switch (action.type) {
    case END_STROKE: {
      const { historyLimit, stroke } = action.payload
      if (!stroke.points.length) {
        return state
      }
      return [...state.slice(0, state.length - historyLimit), stroke]
    }
    default:
      return state
  }
}
```

Note that here we don't modify the `historyIndex` state. We have a separate `END_-STROKE` action handler in the `historyIndex` reducer.

Move the `strokes` selector from `src/reducer.ts` to `src/modules/strokes/selectors.ts`.

04-redux/step4/src/modules/strokes/selectors.ts

```
import { RootState } from "../../types";

export const strokesLengthSelector = (state:RootState) => state.strokes\
.length

export const strokesSelector = (state:RootState) => state.strokes
```

Join The Reducers Using combineReducers

Now we can remove the `src/reducer.ts`.

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:

04-redux/step5/src/store.ts

```
import { configureStore, getDefaultMiddleware, combineReducers } from "\
@reduxjs/toolkit"
import {reducer as historyIndex} from './modules/historyIndex/reducer'
import {reducer as currentStroke} from './modules/currentStroke/reducer'
import {reducer as strokes} from './modules/strokes/reducer'
import logger from 'redux-logger'

const middleware = [...getDefaultMiddleware(), logger]

export const store = configureStore({ reducer: combineReducers({
  historyIndex,
  currentStroke,
  strokes,
}), middleware })
```

We import our reducers separately. Then we pass an object with our reducers as fields to the `combineReducers` method.

Launch the application to check that it works.

Exporting An Image

Let's allow exporting the picture to a file.

Create a new file `src/shared/FilePanel.tsx`. This panel will have the *Export* button.

Make the necessary imports:

04-redux/step5/src/shared/FilePanel.tsx

```
import React from "react"
import { useCanvas } from "../CanvasContext"
import { saveAs } from "file-saver"
import { getCanvasImage } from "../canvasUtils"
```

Define the `FilePanel` component:

04-redux/step5/src/shared/FilePanel.tsx

```
import React from "react"
import { useCanvas } from "../CanvasContext"
import { saveAs } from "file-saver"
import { getCanvasImage } from "../canvasUtils"

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>
  )
}
```

When the user clicks the button we'll generate the `Blob` from the canvas and then save it to the disk using the `file-saver` package.

Install the `file-saver`:

```
yarn file-saver @types/file-saver
```

Now add the `getCanvasImage` function to canvas utils:

04-redux/step5/src/canvasUtils.ts

```
export const getCanvasImage = (
  canvas: HTMLCanvasElement | null
): Promise<null | Blob> => {
  return new Promise((resolve, reject) => {
    if (!canvas) {
      return reject(null)
    }
    canvas.toBlob(resolve)
  })
}
```

We'll need to pass the reference to the canvas to this function. To make the canvas available from the `FilePanel`, let's move it to the context provider.

Create a new file `src/CanvasContext.tsx` with the following contents:

04-redux/step5/src/CanvasContext.tsx

```
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)
```

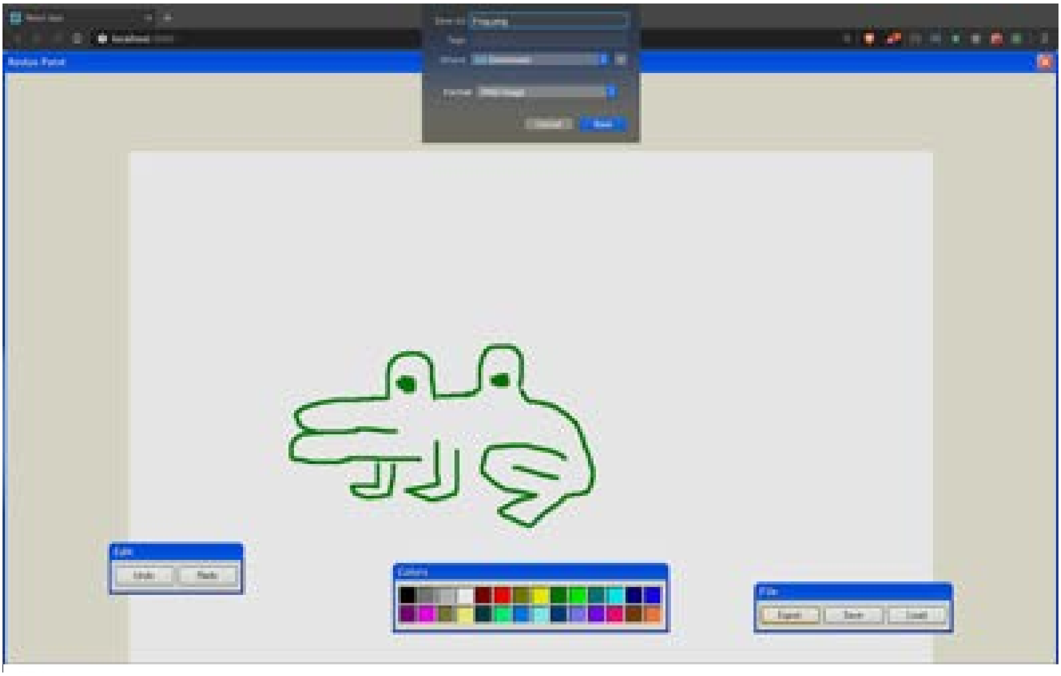
This provider will store the reference to the context. Go to `src/App.tsx` and change the call to `useRef` to `useCanvas` hook.

04-redux/step5/src/App.tsx

```
  const dispatch = useDispatch()
```

Now inside `FilePanel`, we can get the reference to the canvas and pass it to the `getCanvasImage` function.

Launch your application, draw something, and try to export it as a file.



Exporting an image

Using Redux Toolkit

Redux Toolkit[157] 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` and `react-redux` packages.

[157] https://redux-toolkit.js.org/

```
yarn remove redux react-redux
```

Configuring The Store

The first change is how you initialize your store. Now it's done using the configure-Store[158] method.

Open `src/store.ts` and remake it like this:

04-redux/step6/src/store.ts

```
import {
  configureStore,
  getDefaultMiddleware,
  Action
} from "@reduxjs/toolkit"
import { currentStroke } from './modules/currentStrokeSlice'
import { historyIndex } from './modules/historyIndexSlice'
import { strokes } from './modules/strokesSlice'
import logger from "redux-logger"
import { RootState } from "./utils/types"

const middleware = [...getDefaultMiddleware(), logger]

export const store = configureStore({
  reducer: {
    historyIndex,
    strokes,
    currentStroke,
  },
  middleware
})
```

Now we don't have to combine middleware, we can provide them as a list.

[158] https://redux-toolkit.js.org/api/configureStore

We use `getDefaultMiddleware` to use the default middlewares provided by `redux-toolkit`.

Currently, the list of returned middlewares contains the following:

- Immutability Check Middleware[159] - this middleware checks that you don't mutate the state in your reducers. It will throw an error if you do.
- Serializability check middleware[160] - 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.

Using createAction

Right now we have to define a type constant and an action creator for each action in our project.

Redux Toolkit provides the createAction[161] 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.

Go to `src/modules/historyIndex/actions.ts` and make it look like this:

[159] https://github.com/reduxjs/redux-toolkit/blob/master/docs/api/immutabilityMiddleware.md
[160] https://github.com/reduxjs/redux-toolkit/blob/master/docs/api/serializabilityMiddleware.md
[161] https://redux-toolkit.js.org/api/createAction

04-redux/step6/src/modules/historyIndex/actions.ts

```
import { createAction } from "@reduxjs/toolkit"
import { Stroke } from "../../utils/types"

export const endStroke = createAction<{
  stroke: Stroke
  historyIndex: number
}>("endStroke")

export const undo = createAction<number>("UNDO")

export const redo = createAction("REDO")
```

Then go to `src/modules/currentStroke/actions.ts` and remake it like this:

04-redux/step6/src/modules/currentStroke/actions.ts

```
import { createAction } from "@reduxjs/toolkit"
import { Stroke, Point } from "../../utils/types"

export const beginStroke = createAction<Point>("BEGIN_STROKE")

export const updateStroke = createAction<Point>("UPDATE_STROKE")

export const setStrokeColor = createAction<string>("SET_STROKE_COLOR")

export const endStroke = createAction<{
  stroke: Stroke
  historyIndex: number
}>("endStroke")
```

Update the `src/modules/strokes/actions.ts` to look like this:

04-redux/step6/src/modules/currentStroke/actions.ts

```
import { createAction } from "@reduxjs/toolkit"
import { Stroke } from "../../utils/types"

export const endStroke = createAction<{
  stroke: Stroke
  historyIndex: number
}>("endStroke")
```

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[162] 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`:

04-redux/step6/src/modules/currentStroke/reducer.ts

```
import { createReducer } from "@reduxjs/toolkit"
```

Now update the reducer to look like this:

[162] https://immerjs.github.io/immer/docs/introduction

04-redux/step6/src/modules/currentStroke/reducer.ts

```
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, action) => {
    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

Now go to `src/modules/strokes/reducer.ts`. Rewrite the code to use `createReducer`:

04-redux/step6/src/modules/strokes/reducer.ts

```
import { RootState } from "../../utils/types"
import { createReducer } from "@reduxjs/toolkit"
import { endStroke } from "../../actions"

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)
    }
  })
})
```

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`:

04-redux/step6/src/modules/historyIndex/reducer.ts

```
import {
  endStroke, 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, action) => {
    return Math.max(state - 1, 0)
  })
  builder.addCase(endStroke, (state, action) => {
    return 0
  })
})
```

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.[163]

Launch the application and make sure it works.

[163] https://immerjs.github.io/immer/docs/pitfalls

Using Slices

Currently, we have to 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 make the necessary imports:

04-redux/step7/src/modules/historyIndex/slice.ts

```
import { createSlice, PayloadAction } from "@reduxjs/toolkit"
```

Now remake the reducer into slice:

04-redux/step7/src/modules/historyIndex/slice.ts

```
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)
    }
  }
})
```

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`.

First let's add it to shared actions. Create the `src/modules/sharedActions.ts` file with the following contents:

04-redux/step7/src/modules/sharedActions.ts

```
import { createAction } from "@reduxjs/toolkit";
import { Stroke } from "../utils/types";

export const endStroke = createAction<{
  stroke: Stroke
  historyIndex: number
}>("endStroke")
```

As the `END_STROKE` action is shared, we need to define it in `extraReducers`:

04-redux/step7/src/modules/historyIndex/slice.ts

```
  extraReducers: (builder) => {
    builder.addCase(endStroke, () => {
      return 0
    })
  }
```

Add this block to the slice definition below the `reducers` field.

Export the reducer and the actions from the slice:

04-redux/step7/src/modules/historyIndex/slice.ts

```
export default historyIndex.reducer

export const { undo, redo } = historyIndex.actions
```

Remove the `src/modules/historyIndex/actions.ts` file.

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:

04-redux/step7/src/modules/strokes/slice.ts

```
import { createSlice } from "@reduxjs/toolkit"
import { RootState } from "../../utils/types"
import { endStroke } from "../sharedActions"
```

Now we need to define the initial state.

04-redux/step7/src/modules/strokes/slice.ts

```
const initialStrokes: 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:

04-redux/step7/src/modules/strokes/slice.ts

```
const strokes = createSlice({
  name: "strokes",
  initialState: initialStrokes,
  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:

04-redux/step7/src/modules/strokes/slice.ts

```
export default strokes.reducer
```

CurrentStroke Slice

Open `src/modules/currentStroke/reducer.ts`. Let's remake it to slice as well.

First remake the imports:

04-redux/step7/src/modules/currentStroke/slice.ts

```
import { createSlice, PayloadAction } from "@reduxjs/toolkit"
import { RootState, Point } from "../../utils/types"
import { endStroke } from "../sharedActions"
```

Then define the initial state:

04-redux/step7/src/modules/currentStroke/slice.ts

```
const initialState:RootState["currentStroke"] = {color: "#000", points:\
 []}
```

Now let's remake the reducer into a slice:

04-redux/step7/src/modules/currentStroke/slice.ts

```
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
    }
  }
})
```

This slice has three reducers 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 need to handle the `END_STROKE` shared action:

04-redux/step7/src/modules/currentStroke/slice.ts

```
  extraReducers: (builder) => {
    builder.addCase(endStroke, (state) => {
      state.points = []
    })
  }
```

In this extra reducer, we'll reset the `currentStroke` points array.

Export the reducers and actions:

04-redux/step7/src/modules/currentStroke/slice.ts

```
export const currentStroke = slice.reducer;

export const { beginStroke, updateStroke, setStrokeColor } = slice.acti\
ons;
```

Remake The Imports

Go to `src/store.ts`. Remake the imports, so that we import reducers from the slices:

04-redux/step7/src/store.ts

```
import strokes from './modules/strokes/slice'
import logger from "redux-logger"
```

Go to `src/App.tsx` and update the action imports there:

04-redux/step7/src/App.tsx

```
import {
  beginStroke,
  updateStroke,
} from "./modules/currentStroke/slice"
import { endStroke } from "./modules/sharedActions"
import { useCanvas } from "./CanvasContext"
import { ColorPanel } from "./shared/ColorPanel"
import { FilePanel } from "./shared/FilePanel"
```

Update the action imports in the `src/EditPanel.tsx`:

04-redux/step7/src/shared/EditPanel.tsx

```
import { strokesLengthSelector } from "../modules/strokes/selectors"
```

Update the `src/ColorPanel.tsx`:

04-redux/step7/src/shared/ColorPanel.tsx

```
import { setStrokeColor } from "../modules/currentStroke/slice"
```

Now our application uses slices - congratulation! Launch the app and verify that everything works.

Save And Load Data Using Thunks

Right now we can only export our drawings as `*.png` images. It would be cool to be able to save them as projects, and preserve the history of edits.

We also need to learn how to work with side-effects in Redux Toolkit.

We'll save the projects on the backend. To do this we'll use the server that comes with the code examples.

Copy the server from `code/04-redux/server` to your application root folder.

You'll also need to install a few dependencies for it to work:

```
yarn add --dev concurrently cors express lowdb nanoid ts-node
```

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 @types/express @types/lowdb
```

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 dev
```

I recommend stopping your app if it's running and relaunching it using the `start:server` script:

```
yarn start:server
```

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:

04-redux/step8/src/modules/modals/slice.ts

```
import { createSlice, PayloadAction } from "@reduxjs/toolkit"
```

Define the `ModalState` type:

04-redux/step8/src/modules/modals/slice.ts

```
export type ModalState = {
  isShown: boolean
  modalName: string | null
}
```

Then define the initial state with this type:

04-redux/step8/src/modules/modals/slice.ts

```
const initialState: ModalState = {
  isShown: true,
  modalName: null
};
```

Now we can define the slice:

04-redux/step8/src/modules/modals/slice.ts

```
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 and the actions:

04-redux/step8/src/modules/modals/slice.ts

```
export const modalVisible = slice.reducer

export const { show, hide } = slice.actions
```

Go to `src/store.ts` and import the new reducer:

04-redux/step8/src/store.ts

```
import {modalVisible} from './modules/modals/slice'
```

Add the reducer to the combined store:

04-redux/step8/src/store.ts

```
export const store = configureStore({
  reducer: {
    historyIndex,
    strokes,
    currentStroke,
    modalVisible,
    projectsList
  },
  middleware
})
```

Add The Modal Manager Component

Now we can use the created slice to show the windows.

Create a new file `src/ModalLayer.tsx` with the following content:

04-redux/step8/src/ModalLayer.tsx

```
import React from "react"
import { useSelector } from "react-redux"
import { ProjectsModal } from "./ProjectsModal"
import { ProjectSaveModal } from "./ProjectSaveModal"
import { modalNameSelector } from "./modules/modals/selectors"

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.

Now render this component inside the `src/App.tsx` layout. Add it above all the panels we render there.

Add a Window Component

Create a new file `src/ProjectSaveModal.tsx`.

Begin with the imports:

04-redux/step8/src/ProjectSaveModal.tsx

```
import React, { useState, ChangeEvent } from "react"
import { useDispatch } from "react-redux"
import { hide } from "./modules/modals/slice"
import { getCanvasImage } from "./utils/canvasUtils"
import { useCanvas } from "./CanvasContext"
import { getBase64Thumbnail } from "./utils/scaler"
import { saveProject } from "./modules/strokes/saveProject"
```

Define the component:

04-redux/step8/src/ProjectSaveModal.tsx

```
export const ProjectSaveModal = () => {
  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-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>
  )
}
```

This component has an input for the project name and a button that will dispatch the save project action on click.

Define the state to hold the project name state. Add this line to the beginning of your component:

04-redux/step8/src/ProjectSaveModal.tsx

```
const [projectName, setProjectName] = useState("")
```

Then get the `dispatch` method:

04-redux/step8/src/ProjectSaveModal.tsx

```
const dispatch = useDispatch()
```

We'll also need the `canvas` reference:

04-redux/step8/src/ProjectSaveModal.tsx

```
const canvasRef = useCanvas()
```

Define the `projectNameChange` handler:

04-redux/step8/src/ProjectSaveModal.tsx

```
const onProjectNameChange = (e: ChangeEvent<HTMLInputElement>) => {
  setProjectName(e.target.value)
}
```

Here we handle the `ChangeEvent` to update the `projectName` state.

Define the `onProjectSave` handler:

04-redux/step8/src/ProjectSaveModal.tsx

```
  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())
  }
```

Save The Project Using Thunks

The official way to handle side-effects in Redux Toolkit is Thunks[164].

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.

Define the type for our thunk:

04-redux/step8/src/store.ts

```
export type AppThunk = ThunkAction<void, RootState, unknown, Action<str\
ing>>
```

Create the file `src/modules/strokes/saveProject/thunk.ts` and define the `saveProject` thunk there:

[164] https://github.com/reduxjs/redux-thunk

04-redux/step8/src/modules/strokes/saveProject.ts

```
import { AppThunk } from "../../store"
import { newProject } from "./api"

export const saveProject = (
  projectName: string,
  thumbnail: string
): AppThunk => async (dispatch, getState) => {
  try {
    const response = await newProject(
      projectName,
      getState().strokes,
      thumbnail
    )
    console.log(response)
  } catch (err) {
    console.log(err.message)
  }
}
```

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.

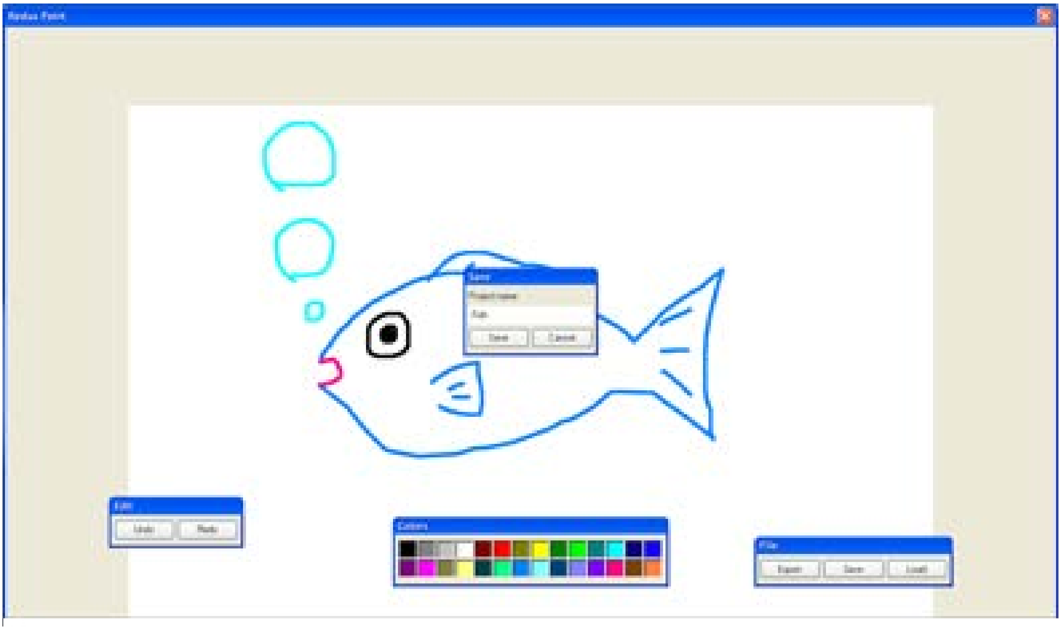
Here we are using the `newProject` function from the `api` module. Let's define it.

Create a new file `src/modules/strokes/api.ts` and define the `newProject` function there:

04-redux/step8/src/modules/strokes/api.ts

```
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())
```

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

To load the project we'll first need to present the user with the list of saved projects.

Create a new file `src/ProjectsModal.tsx`.

Make these imports:

04-redux/step8/src/ProjectsModal.tsx

```
import React, { useEffect } from "react"
import { useDispatch, useSelector } from "react-redux"
import { hide } from "./modules/modals/slice"
import { loadProject } from "./modules/strokes/loadProject"
import { getProjectsList } from "./modules/projectsList/getProjectsList"
import { projectsListSelector } from "./modules/projectsList/selectors"
```

Define the `ProjectsModal` component:

04-redux/step8/src/ProjectsModal.tsx

```
export const ProjectsModal = () => {
  const projectList:any = []

  return (
    <div className="window modal-panel">
      <div className="title-bar">
        <div className="title-bar-text">Counter</div>
        <div className="title-bar-controls">
          <button
            aria-label="Close"
            onClick={() => dispatch(hide())}
          />
        </div>
      </div>
      <div className="projects-container">
        {(projectsList.projects || []).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>
      </div>
  )
}
```

For now, we hardcode the `projectsList` to be an empty array. We'll get the actual products list from the backend a bit later.

Now define the `useEffect` with the following contents before the layout:

04-redux/step8/src/ProjectsModal.tsx

```
  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.

We'll define this thunk in a minute.

Define the `onLoadProject` event handler:

04-redux/step8/src/ProjectsModal.tsx

```
  const onLoadProject = (projectId: string) => {
    dispatch(loadProject(projectId))
    dispatch(hide())
  }
```

Define The ProjectsList Module

Create a new folder `src/modules/projectsList`.

First, let's define the slice. Create the `src/modules/projectList/slice.ts` file.

First add the imports:

04-redux/step8/src/modules/projectsList/slice.ts

```
import { createSlice, PayloadAction } from "@reduxjs/toolkit"
import { Project } from "../../utils/types"
```

Then define the state type:

04-redux/step8/src/modules/projectsList/slice.ts

```
type ProjectsListState = {
  error: string | null
  pending: boolean
  projects: Project[]
}
```

Define the initial state:

04-redux/step8/src/modules/projectsList/slice.ts

```
const initialState: ProjectsListState = {
  error: null,
  pending: true,
  projects: []
}
```

Define the slice:

04-redux/step8/src/modules/projectsList/slice.ts

```
const slice = createSlice({
  name: "projectsList",
  initialState,
  reducers: {
    getProjectsListSuccess: (
      state,
      action: PayloadAction<Project[]>
    ) => {
      state.error = null
```

```
      state.pending = false
      state.projects = action.payload
    },
    getProjectsListFailed: (state, action: PayloadAction<string>) => {
      state.error = action.payload
      state.pending = false
      state.projects = []
    }
  }
})
```

Here we define two reducers, one to handle successful data fetching, and another to handle errors.

Export the reducer and the actions:

04-redux/step8/src/modules/projectsList/slice.ts

```
export const projectsList = slice.reducer

export const {
  getProjectsListFailed,
  getProjectsListSuccess
} = slice.actions
```

Add the reducer to the store:

04-redux/step8/src/store.ts

```
import {
  configureStore,
  getDefaultMiddleware,
  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"

const middleware = [...getDefaultMiddleware(), logger]

export const store = configureStore({
  reducer: {
    historyIndex,
    strokes,
    currentStroke,
    modalVisible,
    projectsList
  },
  middleware
})

export type AppThunk = ThunkAction<void, RootState, unknown, Action<str\
ing>>
```

Let's define the API. Create the `src/modules/projectsList/api.ts` file. It should have the `fetchProjectsList` function defined there:

04-redux/step8/src/modules/projectsList/api.ts

```
export const fetchProjectsList = () =>
  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.

Now we can define the thunk that will fetch the projects list. Create a new file `src/modules/projectsList/getProjectsList.ts`.

Add the following there:

04-redux/step8/src/modules/projectsList/getProjectsList.ts

```
import { AppThunk } from "../../store"
import { Project } from "../../utils/types"
import {
  getProjectsListSuccess,
  getProjectsListFailed
} from "./slice"
import { fetchProjectsList } from "./api"

export const getProjectsList = (): AppThunk => async (dispatch) => {
  try {
    const projectsList: Project[] = await fetchProjectsList()
    dispatch(getProjectsListSuccess(projectsList))
  } catch (err) {
    dispatch(getProjectsListFailed(err.toString()))
  }
}
```

Here we call the `api` and then if we get the data, `dispatch` it through the `getProjectListSuccess` action.

Now let's define the selector. Create the `src/modules/projectsList/selectors.ts` file with the following contents:

04-redux/step8/src/modules/projectsList/selectors.ts

```
import { RootState } from "../../utils/types"

export const projectsListSelector = (state: RootState) =>
  state.projectsList
```

After you have the selector, go back to the `src/ProjectsModal.tsx` and use the new selector instead of the hardcoded data:

04-redux/step8/src/ProjectsModal.tsx

```
  const projectsList = useSelector(projectsListSelector)
```

Now we need to define the `loadProject` thunk.

Create the `src/modules/strokes/loadProject.ts` file:

04-redux/step8/src/modules/strokes/loadProject.ts

```
import { AppThunk } from "../../store";
import { getProject } from "./api";
import { setStrokes } from "./slice";

export const loadProject = (projectId: string): AppThunk => async (
  dispatch) => {
  try {
    const { project } = await getProject(projectId)
    dispatch(setStrokes(project.strokes));
  }
  catch (err) {
    console.log(err.message);
  }
};
```

Here we use the `getProject` API method to load the project data.

Create the `api.ts` inside the `src/modules/strokes` folder:

04-redux/step8/src/modules/strokes/api.ts

```
export const getProject = (projectId: string) =>
  fetch(`http://localhost:4000/projects/${projectId}`).then((res) =>
    res.json()
  )
```

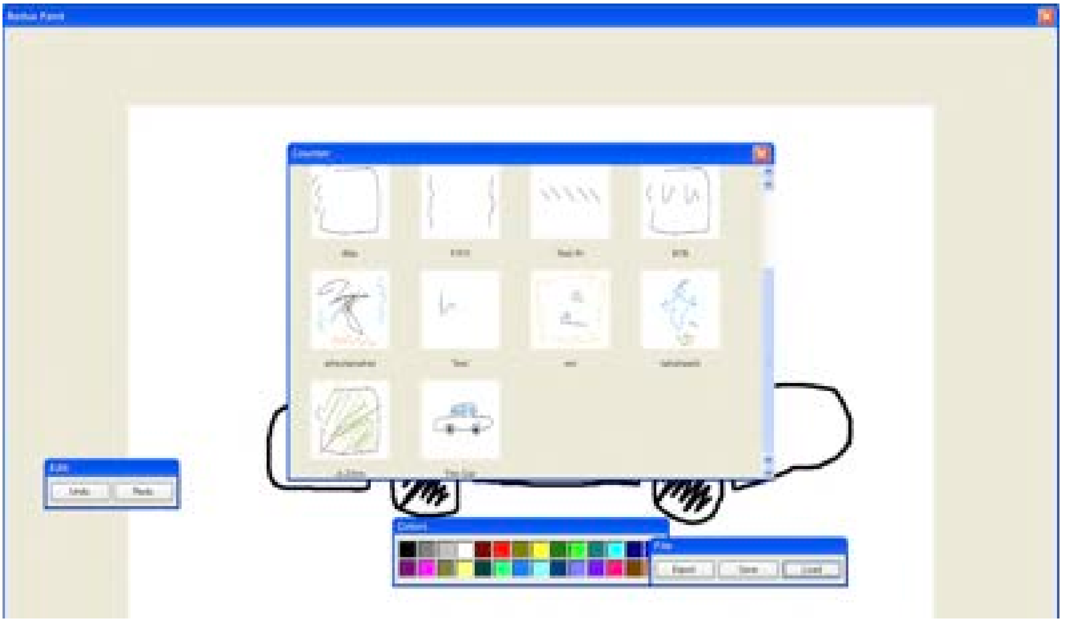
Note that our `loadProject` thunk dispatches the `setStrokes` action with the loaded strokes.

Let's define the reducer to process it.

04-redux/step8/src/modules/strokes/slice.ts

```
  reducers: {
    setStrokes: (state, action: PayloadAction<Stroke[]>) => {
      return action.payload
    }
  },
```

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[165], known as SPAs. They are so called because of the way that the page refresh goes: our application would not reload the whole page, but it would fetch new data and re-render only the parts of the page that should be updated instead. Since all this happens on the same page, they are called SPAs.

There is a caveat in this flow, though. Say, we want all the pages of our application to be detectable by search engines. It cannot be done if all the data fetching and re-rendering happens only in a user's browser. The vast majority of search robots wouldn't wait until the real content of an application appears. They would instead read the content of the HTML we serve them at the start, which is almost empty.

For an application that relies hugely on its content, such as a blog platform or a news site, this is not acceptable. Here the pre-rendering[166] comes in.

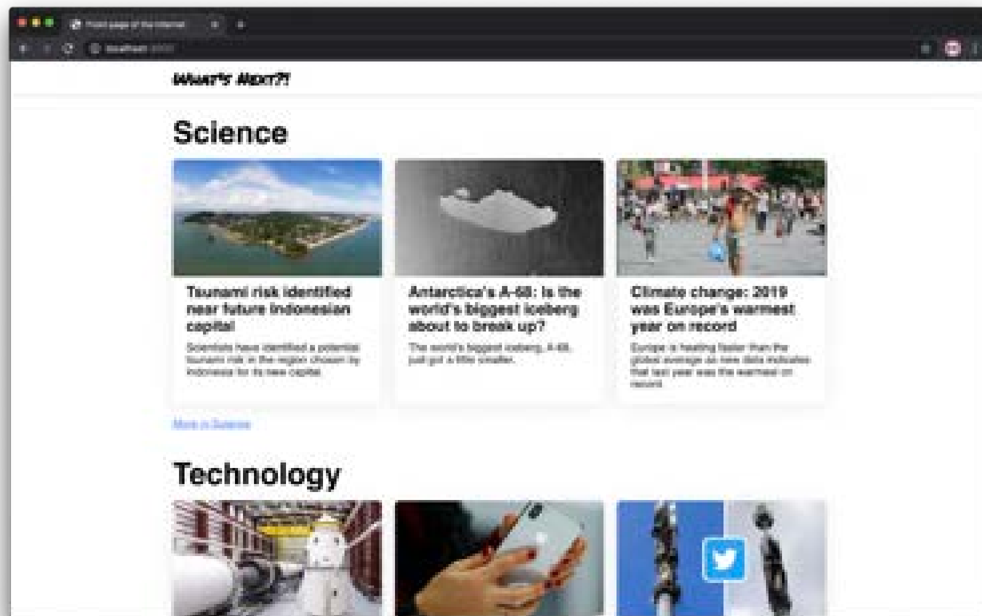
What We're Going to Build

To fully understand all the advantages of pre-rendering, we have to create an application that has a lot of text content. With that in mind, we're going to create a news site. We will take the BBC website[167] as a source of news and images and create an application with pre-rendered pages with content on them.

[165]https://en.wikipedia.org/wiki/Single-page_application
[166]https://nextjs.org/docs/basic-features/pages#pre-rendering
[167]https://www.bbc.com

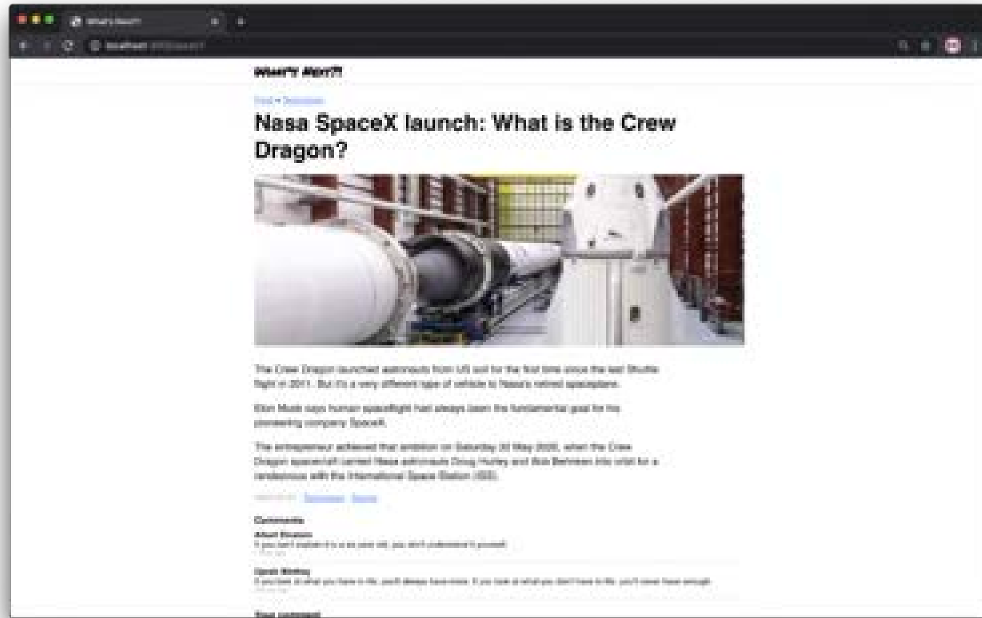
We will both statically generate some pages and use pre-rendering on a server. Our final app will use static generation for pages with post categories and the front page and pre-render for single post pages. Also, we will create a comment form that will be connected to the Redux store, and hydrate the store when using on a client.

The main page of the completed application will look like this:



A completed news site

And a post page will look like this:



A post page of the application

A complete code example is located in `code/05-next-ssg/completed`.

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

```
cd code/05-next-ssg/completed
```

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

```
yarn && yarn dev
```

This should open the app in the browser. If it doesn't, navigate to http://localhost:3000[168] and open it manually.

[168]http://localhost:3000

Pre-Rendering

As we said earlier, for an application that relies so much on its content, serving empty pages is not acceptable. Here, we would want to pre-render pages of an application to serve them with the content.

The two main ways to pre-render pages are Server-Side Rendering and Static Site Generation.

Server-Side Rendering

Server-Side Rendering[169], or SSR, is a technique where a server renders real HTML for every page request it gets. For our application, it would mean that 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 a server. Instead, we could have 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 is associated with the minimum JavaScript code necessary for that page. When a page is loaded by the browser, its JavaScript code runs and makes the page interactive. Thus, an application that was "frozen" resurrects and runs from the point at which it was "frozen". This process is called hydration[170].

Static Site Generation

Static Site Generation[171], or SSG, means that pages' HTML is generated at build time once. So, technically this means that we will have all the real HTML files for each page.

The advantage of this technique is that SSG responds faster, since it doesn't need to render each 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.

[169]https://nextjs.org/docs/basic-features/pages#server-side-rendering
[170]https://nextjs.org/docs/basic-features/pages#pre-rendering
[171]https://nextjs.org/docs/basic-features/pages#static-generation-recommended

We will use both SSG and SSR. We will explore the difference between them a bit later.

Next.js

We're going to use Next.js.[172]

Next is a framework for creating React applications. We chose Next because it has a clean API and all the features we're going to need for our purposes, SSG included. Also, it has great documentation and tutorials.

Setting Up a Project

First of all, we have to set up a project. Next has a set of instructions[173] for getting started, however, we want to walk through the setting up step by step.

For starters, let's create a directory in which our project will be located.

```
mkdir news-site
```

Inside, we have to create two more directories, `pages` and `public`. The first is a directory in which Next will search for pages[174] of our application (we will talk about pages in detail a bit later). The second one is a directory for static resources[175] like images, stylesheets, etc.

```
cd news-site
mkdir pages
mkdir public
```

Then, let's initialize a project and add all the dependencies we're going to need:

[172] https://github.com/zeit/next.js/
[173] https://nextjs.org/docs/getting-started
[174] https://nextjs.org/docs/basic-features/pages
[175] https://nextjs.org/docs/basic-features/static-file-serving

```
yarn init -y
yarn add next react react-dom
```

Once initialized, we want to update the `scripts` section of our `package.json` file and add the following scripts:

05-next-ssg/step-1/package.json

```
  "scripts": {
    "dev": "next",
    "build": "next build",
    "start": "next start"
  },
```

Among those scripts: - `dev`, runs a development environment - we will use this the most often - `build`, will build our application and generate rendered pages - `start`, we won't use in this chapter, but this script is used in production environments on servers when an application is started

Adding TypeScript

By default, Next uses JavaScript, not TypeScript. To integrate TypeScript we have to set it up as well.

First, we're going to add all of the development dependencies.

```
yarn add --dev typescript @types/react @types/node
```

Then, we will create an empty `tsconfig.json` file in the root directory of the project:

```
touch tsconfig.json
```

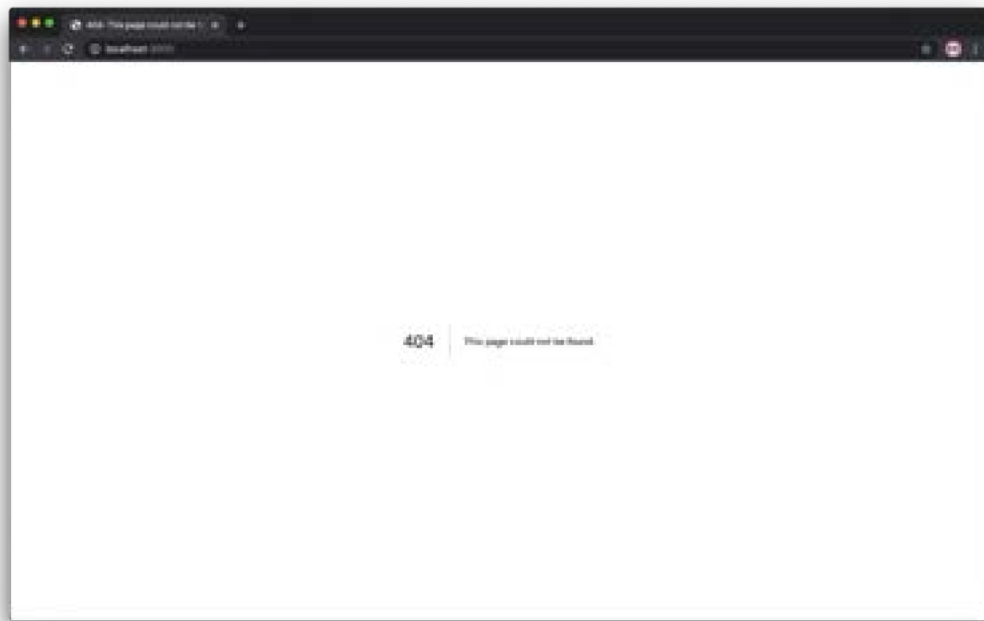
Notice that we don't populate it with any content. Next will do this for us automatically when we run:

```
yarn dev
```

This command should open the app in the browser. If it doesn't, navigate to http://localhost:3000[176] and open it manually.

Creating A First Page

When opened, the application should show a 404 error.



By default there is a "Not found" error

This is fine. Next renders a 404 error because we haven't created any pages yet. So, let's fix that!

A page[177] in Next is a React Component exported from a `.js`, `.jsx`, `.ts`, or `.tsx` file in the `pages` directory. That's why we created that folder - to populate it with page components.

[176] http://localhost:3000
[177] https://nextjs.org/docs/basic-features/pages

To create our first page we need to create the file `pages/index.tsx` and export a React Component from it:

05-next-ssg/step-1/pages/index.tsx

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

First of all, notice that we use a default export here. That's because Next requires page components to be default-exported.

Another interesting thing is a `Head` component imported from `next/head`. This is a component that injects everything we pass as children inside of the `head` element on an HTML page. In our case, we pass the `title` element with the page title to update it.

When the file is created, Next should notice that there is a new page and refresh the browser, whereupon we should see the message "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 a `Header` component.

Header Component

05-next-ssg/step-2/components/Header/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>
  )
}
```

Here, we declare a `Header` component that uses a couple of dependencies, such as `Head` component and `style.ts`. For styles, we're using `styled-components`, and as we know, in order to use them we have to install them first. So, let's do that:

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

After installation, this package can be used in our code. First of all, we want to create a `Container` for our `Header` component which will stick to the page top and contain all the component's content.

05-next-ssg/step-2/components/Header/style.ts

```
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 an `h1` element. It uses props to get access to the theme, which we will cover a bit later in this section.

05-next-ssg/step-2/components/Header/style.ts

```
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};
  }
`
```

Next's Link

The next dependency we used in `Header` is a `Link` component[178] imported from `next/link`. This is a component that enables client-side transition between routes of our app - basically, between pages[179].

Please pay attention to the structure of the `Link` we created. At the top level, we use the `Link` component and provide an `href` attribute to it, and inside we use an `a` element in which we place the link contents.

`Link` requires exactly one element to passed as a child. In cases when we cannot pass an `a` element for some reason, we can use different elements or components and force[180] `Link` to pass an `href` prop further. It will be useful later when we use styled links.

Center Component

Another component that we will use across the whole project is a `Center` component. It is a styled component that does only one thing - it aligns itself at the center of the page.

05-next-ssg/step-2/components/Center/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;
  }
`
```

[178] https://nextjs.org/docs/api-reference/next/link
[179] https://nextjs.org/docs/routing/introduction
[180] https://nextjs.org/docs/api-reference/next/link#if-the-child-is-a-custom-component-that-wraps-an-a-tag

We will use this component to center content in many other places. That's why we didn't place it in `Header/style.ts` but located it in `components/Center/style.ts` instead.

Footer Component

Finally, we create a `Footer` component which we will use at the bottom of the application pages.

05-next-ssg/step-2/components/Footer/Footer.tsx

```
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:

05-next-ssg/step-2/components/Footer/style.ts

```
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 contain a current year and a link to Newline.co site. Notice that here we use not a `Link` component, but an ordinary `a` element instead. That's because `Link` should be used only for navigation between application routes, and not for links to "outer" resources. Otherwise Next will throw an error.

Custom App Component

Once we've created all of the components we're going to need, we want to use them in the app layout.

One possibility for how to use them is to include components in `pages/index.tsx` right away. That would work, but then we would have to include those components in the code of every new page we're going to create. This is not convenient and it violates the DRY principle (Don't Repeat Yourself).

For this problem, Next has a solution. We can create a component that will be like a wrapper for every page Next is going to render. This component is `App`[181].

Next uses the `App` component to initialize pages. We can override it and control the page initialization. It may be useful for: - Persisting layout between page changes - Keeping state when navigating pages - Injecting additional data into pages - Adding global CSS

Let's create one and see how we can use it in our app. First of all, let's decide what we want to import and use in this component.

[181] https://nextjs.org/docs/advanced-features/custom-app

05-next-ssg/step-2/pages/_app.tsx

```
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"
```

We will use `Head` from `next/head` to override page title, `ThemeProvider` from `styled-components` for using the theme which we will create in `shared/theme` shortly, and all the components we created earlier.

Then, we create a component `MyApp` and export it. Notice the props of `MyApp`: `Component` and `pageProps` - those are the props that Next injects 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 the page.

We render `Component` inside and pass `pageProps` to it using spreading. In other words, we render a current page and pass all the props required for it.

Also, we use `Head` and `title` elements to set a default page title and `Header` and `Footer` components to create a layout. Finally, we wrap all of this in `ThemeProvider` to provide access to the theme for every styled component.

05-next-ssg/step-2/pages/_app.tsx

```
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>
  )
}
```

Application Theme

Now it is time to create a theme for our application!

First of all, we declare an object `theme` with the fonts and colors we're going to use.

05-next-ssg/step-2/shared/theme.ts

```
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 want to create global styles for all the pages. We declare a new type `MainThemeProps` which will be used in `createGlobalStyle()` generic function on the next line.

05-next-ssg/step-2/shared/theme.ts

```
export type MainThemeProps = ThemeProps<typeof theme>
export const GlobalStyle = createGlobalStyle<MainThemeProps>`
```

Next we create some basic global styles for `body`, headings, links and `.main` block.

05-next-ssg/step-2/shared/theme.ts

```
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);
  }
`
```

This `GlobalStyle` component we use in `MyApp` to inject those styles into pages' code.

From now on we will focus more on the components' code and the integration with Next, and less on the styles' code. You can find all the styles in sources besides the corresponding components.

Custom Document Component

So far we have created global styles and the theme, but if we look closely at our theme we can find that the accent font is defined as `"Permanent Marker"` font-family. This is not a font that every device has, so we have to include it.

We can use Google Fonts to get this font, however, it is not yet clear where we can place a `link` element with a link to a stylesheet with this font. We could include it in `MyApp` component, but Next has another option called custom `Document` component[182].

Next's `Document` component not only encapsulates `html` and `body` declarations but can also include initial props[183] for expressing asynchronous server-rendering data requirements. In our case, initial props would be the styles across the application.

But why not just render styled components as we usually do? That's a tricky question because since we want to create an application that is being rendered on a server and then gets "hydrated" on a client, we have to make sure that page's markup from a server and markup on a client are the same. Otherwise, we would get an error that some properties are not the same.

To make the markup consistent, we have to make styles and class names consistent as well. And that is what custom `Document` is going to help us to do.

To see the difference between `App` and `Document` let's compare them:

	App	Document
Shared logic and layout	Yes	Not recommended[184]
Global styles	Yes	Not recommended
Renders on…	Client and Server	Server
Event handlers like `onClick`	Will work	Won't work
Need to restart dev-server after change	Yes	Yes
Styled-components sheet collection	No	Yes[185]
Global middleware	Page-level only	App level, request level

[182] https://nextjs.org/docs/advanced-features/custom-document
[183] https://nextjs.org/docs/api-reference/data-fetching/getInitialProps#context-object
[184] https://nextjs.org/docs/advanced-features/custom-document#caveats
[185] https://github.com/vercel/next.js/tree/master/examples/with-styled-components

Also, custom `getInitialProps()` in `App` will disable Automatic Static Optimization in pages without Static Generation. And custom `getInitialProps()` in `Document` is not called during client-side transitions, nor when a page is statically optimized.

Now let's create a blueprint for the custom `Document` component. Here, we import `ServerStyleSheet` from `styled-components` which will help us to collect all the styles needed to be sent to a client, and a bunch of things from `next/document`. We will cover them in detail a bit later, but now let's pay attention to `Document`.

05-next-ssg/step-2/pages/_document.tsx

```
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` which extends Next's `Document` component. Then, we define a `render()` method.

05-next-ssg/step-2/pages/_document.tsx

```
  render() {
    const description = "The Next generation of a news feed"
    const fontsUrl =
      "https://fonts.googleapis.com/css2?family=Permanent+Marker&displa\
y=swap"

    return (
      <Html>
        <Head>
```

```
        <meta name="description" content={description} />
        <link href={fontsUrl} rel="stylesheet" />
        {this.props.styles}
      </Head>

      <body>
        <Main />
        <NextScript />
      </body>
    </Html>
  )
}
```

Notice that we don't use an `html` element, but we use an `Html` component imported from `next/document` instead. This is because `Html`, `Head`, `Main` and `NextScript` are required for the page to be properly rendered. `Html` is a root element, `Main` is a component which will render pages, and `NextScript` is a service component required for Next to work correctly.

Inside of a `Head` we create a `meta` element with description and a `link` element with a link to fonts from Google Fonts. This is the place where we keep links to external resources like fonts. Then, we render `this.props.styles` - those are the styles collected using `ServerStyleSheet`. We collect them in `getInitialProps()` method.

05-next-ssg/step-2/pages/_document.tsx

```
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()
    }
  }
```

This method is `static` which means that it can be called on a `class` (without creating an instance of it) like this: `Document.getInitialProps()`. This method takes a Next's `DocumentContext` as an argument. This is an object that contains many useful things[186], such as `pathname` of a page URL, `req` for request, `res` for response and error object `err` for any error encountered during the rendering.

Here, we kind of extend it with our `styles` prop, to make them accessible in `render()` method later. We create a `sheet` which is an instance of a `ServerStyleSheet` - that way we will be able to collect styles from the whole application. Next, we "remember" `ctx.renderPage()` method in a constant `originalRenderPage` to "override" original `ctx.renderPage()` inside of `try-finally` clause.

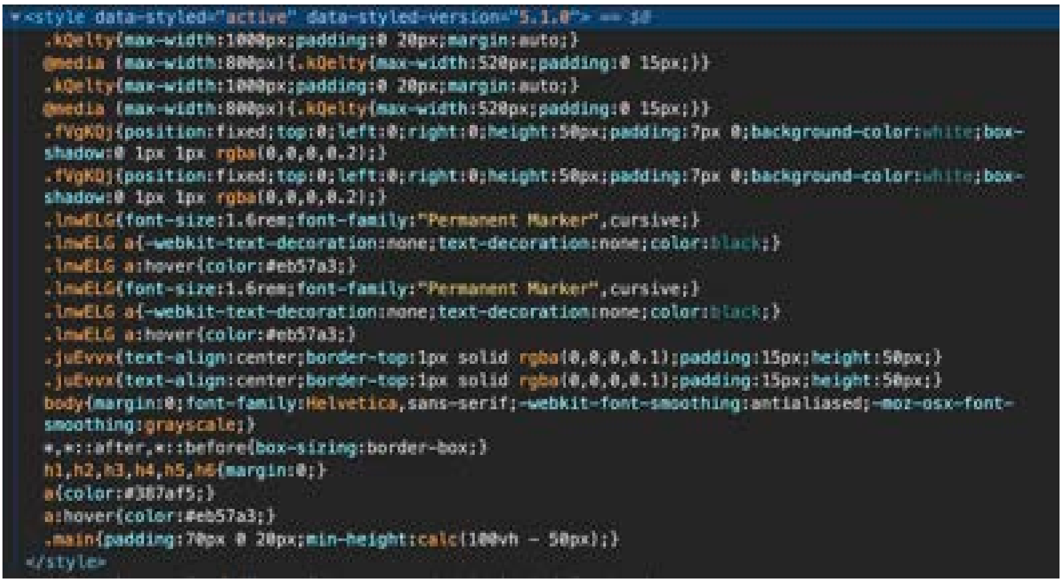
When overriding it we use `sheet.collectStyles()`[187] method and pass the whole rendered application as an argument. It will gather all the styles so that we will be able to extract them by calling `sheet.getStyleElement()` later.

Then, we "remember" original `initialProps` by calling `Document.getInitialProps()`. Notice that we call it like a static method. That's why we had to make our `getInitialProps()` static as well - to make sure that we don't break compatibility.

[186] https://nextjs.org/docs/api-reference/data-fetching/getInitialProps#context-object
[187] https://styled-components.com/docs/advanced#example

As a result, we return from this method an object that contains all of the original `initialProps` and a `styles` prop which contains a component with `style` elements that contain all the styles that are required to be sent along with the page markup.

In the browser it should look like a `style` element filled with app styles:



Final collected styles

After all that, in a `finally` clause we call `sheet.seal()` method. Thus, we make sure that the `sheet` object is available for garbage collection[188].

Site Front Page

Now, we've prepared everything to create our first page and fix that `404`. Let's start with a front page.

On the front page of the site, we will have a `Feed` with `Post` cards inside. Let's update our `Front` component and include `Feed` in the `main` element.

[188] https://styled-components.com/docs/advanced#example

05-next-ssg/step-3/pages/index.tsx

```
      <main>
        <Feed />
      </main>
```

News Feed

Then, we want to create a `Feed` component. Our `Feed` would contain three sections with post cards inside. Those sections would represent news categories such as science, technology, and arts.

05-next-ssg/step-3/components/Feed/Feed.tsx

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

export const Feed = () => {
  return (
    <>
      <Section title="Science" />
      <Section title="Technology" />
      <Section title="Arts" />
    </>
  )
}
```

News Section

For now, each `Section` component's props would require only a `title`. We will update it later.

05-next-ssg/step-3/components/Section/Section.tsx

```
import { Post } from "../Post"
import { Grid, Title } from "./style"

type SectionProps = {
  title: string
}
```

`Section` itself will contain a `Title` and a `Grid` with a bunch of `Post` cards inside (hardcoded for now).

05-next-ssg/step-3/components/Section/Section.tsx

```
export const Section = ({ title }: SectionProps) => {
  return (
    <section>
      <Title>{title}</Title>
      <Grid>
        <Post />
        <Post />
        <Post />
      </Grid>
    </section>
  )
}
```

In this project, we're not using `FunctionComponent<>` type since none of our components, except pages, don't 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)`.

A `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[189].

[189] https://stackoverflow.com/questions/18744164/flex-box-align-last-row-to-grid

05-next-ssg/step-3/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;
  }
```

Also, we use `@media` to define adaptive styles for our grid.

05-next-ssg/step-3/components/Section/style.ts

```
  @media (max-width: 800px) {
    &:after,
    & > * {
      width: 100%;
    }
  }
```

Single Post

Now, let's create a `Post` card. This component will play the role of a preview for a full post and will contain an image, a title, and a short text description.

05-next-ssg/step-3/components/Post/Post.tsx

```
import { Card, Figure, Title, Content } from "./style"
import Link from "next/link"

export const Post = () => {
  return (
    <Link href="/post/example" passHref>
      <Card>
        <Figure>
          <img alt="Post photo" src="/image1.jpg" />
        </Figure>
        <Title>Post title!</Title>
        <Content>
          <p>
            Lorem ipsum dolor sit amet, consectetur adipiscing elit,
            sed do eiusmod tempor incididunt ut labore et dolore magna
            aliqua.
          </p>
        </Content>
      </Card>
    </Link>
  )
}
```

A couple of interesting things here. First of all, notice the `passHref` prop on the `Link` component - that is the way that we tell Next to provide `href` prop further on a child of `Link`. This is because we don't pass an `a` element to a `Link` but we pass a `Card` instead.

`Card` is a styled `a` element, so it is technically not an `a`, but an `a` wrapped in some other thing. Without this prop, an `a` element won't have a `href` attribute, which can affect SEO.

Next, we need to define the `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[190] 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 might result in missing pre-rendering of that page
<Link href={`/posts/${post.id}`} />
```

Since Next 10 there is no need[191] to specify the `as` prop anymore. So we can safely use just `href` in our `Card` component.

Lastly, notice the `src="/image1.jpg"` on `img` element. This is the path for an image from our `public` directory. By default, Next serves everything from `public` and makes it accessible right from `/` path. Thus, if we want to render an image we use `src` prop with a path to an image respectively to the `public` folder's root.

K> Later in this chapter we will optimize images with the `next/image` component that was introduced in the Next 10.

Now, on the main page, you should see three `Section` components with three `Post` cards in each of them. However, if we click on any of the `Post` cards we will see the default 404 page. So, before we create a post page, let's update 404 a bit.

Page 404

To create a custom 404 page[192] we're going to need to create a file called `404.tsx`.

In that file, we create a component `NotFound` which we're going to export by default.

[190] https://nextjs.org/docs/routing/dynamic-routes
[191] https://nextjs.org/blog/next-10#automatic-resolving-of-href
[192] https://nextjs.org/docs/advanced-features/custom-error-page

05-next-ssg/step-3/pages/404.tsx

```
const NotFound = () => {
  return (
    <Container>
      <Main>404</Main>
      Oops! The page not found!
    </Container>
  )
}

export default NotFound
```

Also, in that exact file, we define styles for our `404`.

05-next-ssg/step-3/pages/404.tsx

```
import styled from "styled-components"

const Container = styled.div`
  display: flex;
  flex-wrap: wrap;
  justify-content: center;
  align-items: center;
  text-align: center;
`

const Main = styled.h2`
  font-size: 10rem;
  line-height: 11rem;
  font-family: ${(p) => p.theme.fonts.accent};
  width: 100%;
`
```

We keep them in the same file because Next requires all the pages to export by default a component that is a page. So we cannot create, say, a directory `404` with file `404/style.ts` and extract the styles in that file. If we do that while building a project we will get an error:

> 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 them in some kind of shared code, but since the styles code is not huge we can keep it here just to gather everything about this page in one place.

And finally, we are ready to create a post page.

Post Page Template

As our first approach to this page, we won't render any content for now. 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 a dynamic route segment[193], we should add brackets to a page file name.

In our case, a new file will be called `[id].tsx` and will be located in `pages/post` directory.

<<05-next-ssg/step-3/pages/post/[id].tsx[194]

Nothing special inside so far. But let's examine more closely a `useRouter()` hook[195]. It is a hook that provides access to a `router` object[196].

In that object there are two values that we are interested in: - `pathname` - current route, the path of the page in `pages` directory. - `query` - the query string parsed to an object.

A `query` object will contain the `id` of a current post. So, we access it and use it for loading data later on.

---

[193] https://nextjs.org/docs/routing/dynamic-routes
[194] ./code/05-next-ssg/step-3/pages/post/.examples/id.tsx
[195] https://nextjs.org/docs/api-reference/next/router#userouter
[196] 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. This pre-rendering should happen at build time once, and then generated pages should be sent as responses to requests.

In order to be able to generate those pages, we need data to inject into them. We can get this data in many different ways: - from the file system (as .md files for example) - from a remote database directly - from a backend server's API

Next has a great example[197] on working with the file system. We, however, will create a backend server and fetch data from its API.

First of all, let's install the required dependencies:

```
yarn add body-parser concurrently cors express node-fetch ts-node
```

And then, update our scripts section a bit:

```
"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 `serve` script which sets up a server, and updated the `dev` script to run `serve` and `next` at the same time. The `serve` script will run a node.js server using a `server/index.ts` file. Let's create one.

[197] https://nextjs.org/docs/basic-features/data-fetching#simple-example

05-next-ssg/step-4/server/index.ts

```
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())
```

We import all the packages we're going to use and data as well. We could use a database (like MongoDB for example), but for simplicity we will read data straight from json files. You can find them in `05-next-ssg/step-4/server` directory.

We use the `cors` package to make sure that we can send requests from a different localhost port to the server. Also, we use `body-parser` to more conveniently parse data from the body of the request in the future.

Post Data and Type

Let's take a quick look at `posts.json` and see what kind of structure a single post will have. A post is an object with id, some meta information, text content, and 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 post entity with TypeScript first, to be able to use this type later in both client and server codebases. We create a file called `types.ts` in `shared` directory.

05-next-ssg/step-4/shared/types.ts

```
export type UriString = string
export type UniqueString = string
export type EntityId = number | UniqueString

export type Category = "Technology" | "Science" | "Arts"
export type DateIsoString = string
```

Inside we create some common type aliases (like `UriString`, `UniqueString`, `EntityId`, and `DateIsoString`) and a `Category` union. We use type aliases to create more readable types, that can better describe the intent of our code. When created, we use them to describe a `Post` type:

05-next-ssg/step-4/shared/types.ts

```
export type Post = {
  id: EntityId
  date: DateIsoString
  category: Category
  title: string
  lead: string
  content: string
  image: UriString
  source: UriString
}
```

API Endpoints

Now, we want to create API endpoints to make data accessible via `GET` requests.

05-next-ssg/step-4/server/index.ts

```
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}!`)
)
```

Here we set up a port `4000` for this server and create two endpoints - `/posts` (so that when a client sends a request on `http://localhost:4000/posts` it would get a list of posts as a response), and `/categories`.

Frontend API Client

When we have created a server API, we can create a frontend client for that API. Let's create a directory `api` with two files in it: `config.ts` and `summary.ts`.

The `config.ts` will contain configuration settings for our requests. A `baseUrl` setting will help us to reduce duplication across our request functions.

05-next-ssg/step-4/api/config.ts

```
export const config = {
  baseUrl: "http://localhost:4000"
}
```

`summary.ts` will have functions for fetching data for the main page from our server.

05-next-ssg/step-4/api/summary.ts

```
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()
}

export async function fetchCategories(): Promise<Category[]> {
  const res = await fetch(`${config.baseUrl}/categories`)
  return await res.json()
}
```

Notice that we use the `node-fetch` package here. This is because when Next builds a project it will run outside of the browser's environment, so it won't have access to the `fetch()` function. This package creates a function like `fetch()` available in node.

Then there are `fetchPosts()` and `fetchCategories()` functions. Both are `async` and return a `Promise`. The first one requests `/posts` and returns a promise of `Post[]`, and the second one`/categories` and `Category[]` respectively. These functions we will use for fetching and pre-fetching data on the main page.

Updating The Main Page

When the functions for data fetching are done, we can use them to fetch data on the main page. First, let's make our page dependent on posts and categories that will be passed as props.

05-next-ssg/step-4/pages/index.tsx

```
type FrontProps = {
  posts: Post[]
  categories: Category[]
}
```

Here, we create a type `FrontProps` and use it in `Front` component:

05-next-ssg/step-4/pages/index.tsx

```
export default function Front({ posts, categories }: FrontProps) {
  return (
    <>
      <Head>
        <title>Front page of the Internet</title>
      </Head>

      <main>
        <Feed posts={posts} categories={categories} />
      </main>
    </>
  )
}
```

Also, we change `Feed` component's API as well to make it accept posts and categories as props. We will update it a bit later, but now let's take a look at how we can pre-render this page.

Fetching Data

Next has a concept of static props[198]. Those are the props that Next will inject at build time into a page component. In our case, those props would be categories and posts for the main page.

In order to tell Next that we want to fetch some data and pre-render a page, we have to export an `async` function called `getStaticProps()`.

[198]https://nextjs.org/docs/basic-features/data-fetching#getstaticprops-static-generation

05-next-ssg/step-4/pages/index.tsx

```
export async function getStaticProps() {
  const categories = await fetchCategories()
  const posts = await fetchPosts()
  return { props: { posts, categories } }
}
```

In this function we make two 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 those `categories` and `posts`.

This object is going to be injected as `Front` component's props, so that we will have access to them, inside of a component. We should be aware that `getStaticProps()` runs only on the server-side. It will never be run on the client-side. It won't even be included in the bundle for the browser.

Updating Feed

Then, it is time to update the `Feed` component, since we want to pass the props from the `Front` page.

05-next-ssg/step-4/components/Feed/Feed.tsx

```
import { Section } from "../Section"
import { Post, Category } from "../../shared/types"

type FeedProps = {
  posts: Post[]
  categories: Category[]
}
```

We start by declaring a type `FeedProps` and accessing them inside of a component.

05-next-ssg/step-4/components/Feed/Feed.tsx

```
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. After, we render a `Section` for each category and pass a `title` and `posts` for this category as props.

Updating Section

Now, the `Section` component needs to be updated as well.

Again, we start by declaring a type `SectionProps` and accessing them inside of a component.

05-next-ssg/step-4/components/Section/Section.tsx

```
import { Post as PostType } from "../../shared/types"
import { Post } from "../Post"
import { Grid, Title } from "./style"

type SectionProps = {
  title: string
  posts: PostType[]
}
```

Then, we render a `Title` and `Grid` with `Post` cards inside.

05-next-ssg/step-4/components/Section/Section.tsx

```
export const Section = ({ title, posts }: SectionProps) => {
  return (
    <section>
      <Title>{title}</Title>
      <Grid>
        {posts.map((post) => (
          <Post key={post.id} post={post} />
        ))}
      </Grid>
    </section>
  )
}
```

Updating Post Card

And finally, we want to update a `Post` card component.

05-next-ssg/step-4/components/Post/Post.tsx

```
import Link from "next/link"
import { Post as PostType } from "../../shared/types"
import { Card, Figure, Title, Lead } from "./style"

type PostProps = {
  post: PostType
}
```

We declare a type `PostProps` with a `post` field. Then we render a `Link` and pass an `href` prop with a path to our `post/[id].tsx` page, `as` prop which specifies how this URL should look in the browser, and a `passHref` prop to force Next to pass `href` further on a child component.

05-next-ssg/step-4/components/Post/Post.tsx

```
  return (
    <Link href={`/post/${post.id}`} passHref>
      <Card>
```

We use `post.id` in `as` prop to make our URLs look pretty, so that when we render a post with `"id": "some-post"`, the URL would look like `/posts/some-post/`.

The last thing we have to do now is to render every piece of information from `post` in the card.

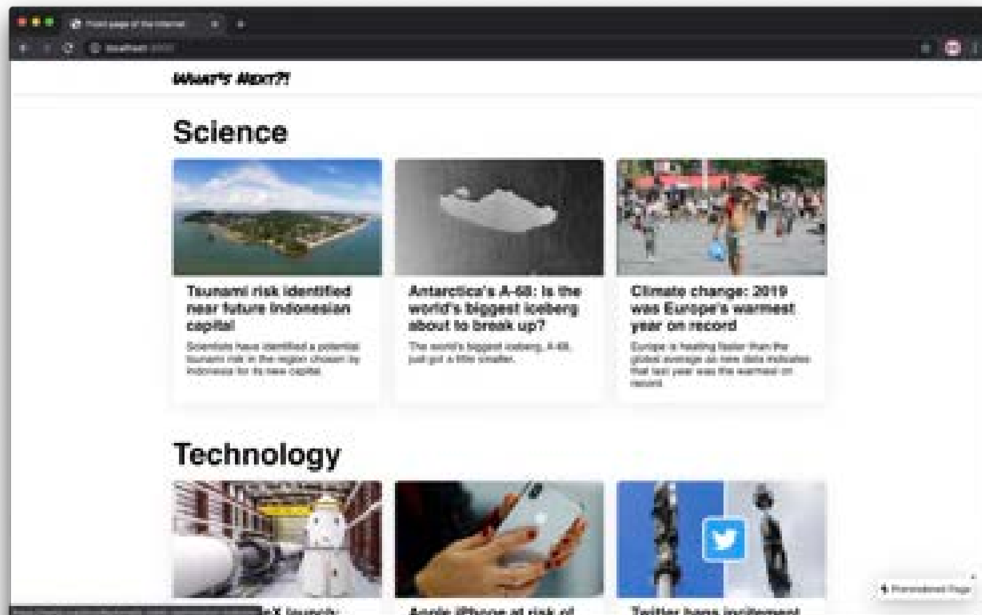
05-next-ssg/step-4/components/Post/Post.tsx

```
export const Post = ({ post }: PostProps) => {
  return (
    <Link href={`/post/${post.id}`} passHref>
      <Card>
        <Figure>
          <img alt={post.title} src={post.image} />
        </Figure>
        <Title>{post.title}</Title>
        <Lead>{post.lead}</Lead>
```

```
      </Card>
    </Link>
  )
}
```

We render an image, a title and a lead text.

After we do this, we can run `yarn dev` and see the result!



Statically generated front page

Here, we see the front page with categories fetched from the server, each of which contains a list of posts for that category also fetched from our Backend API.

Notice the "pre-rendered page indicator" in the bottom right corner of the page. It appears[199] on pages that Next statically generated.

[199] https://nextjs.org/docs/api-reference/next.config.js/static-optimization-indicator

Pre-Render Post Page

Post API

The first thing for us to do is to create an API endpoint for getting single post info.

05-next-ssg/step-5/server/index.ts

```
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)
})
```

Here, we create an endpoint for `/posts/:id`, extract the `id` of a needed post, then search for a post with the same `id` from the list of all posts and return the found one.

Then, we create a function to fetch that data.

05-next-ssg/step-5/api/post.ts

```
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!

Post Page Static Props

For a post page, we also want to declare a props type since this component will accept data via props.

<<05-next-ssg/step-5/pages/post/[id].tsx[200]

Then, since this page is also going to be pre-rendered, we create a `getStaticProps()` function.

<<05-next-ssg/step-5/pages/post/[id].tsx[201]

Notice the `if` statement. Here we check if the type of `params.id` is equal to string. We have to do it because Next gives us an object where `params.id` can be either `string` or `string[]`. Our function and server can only handle `string`, so we need to check the type of a given value.

We don't necessarily have to throw an error at that point - we could gracefully render a message for the user. In our case, for simplicity we use the `throw` operator.

We import `GetStaticProps` from `next` package to declare the types of this function's arguments and returned result. Notice that this time we use an argument that is being passed into this function. This argument is a `context` object[202].

It contains a `params` object, which contains the 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 that is equal to the `id` of a current post, which we will use to fetch data.

Static Paths

There is another exported function, called `getStaticPaths()`. This function determines[203] which paths should be rendered to HTML at build time.

<<05-next-ssg/step-5/pages/post/[id].tsx[204]

Here, we see that this function returns an object with two fields. The first one is `fallback`, which is `true`. When it's `false` any paths not returned by `getStaticPaths()` will result in a 404 page. When `true`, Next will return the "fallback" version of those paths.

[200]./code/05-next-ssg/step-5/pages/post/.examples/id.tsx
[201]./code/05-next-ssg/step-5/pages/post/.examples/id.tsx
[202]https://nextjs.org/docs/basic-features/data-fetching#getstaticprops-static-generation
[203]https://nextjs.org/docs/basic-features/data-fetching#getstaticpaths-static-generation
[204]./code/05-next-ssg/step-5/pages/post/.examples/id.tsx

In our case, we use `router.isFallback` property to render the `Loader` component (which we will cover a bit later). When a user requests a page that is not yet rendered but has a "fallback", they will see a `Loader`. Meanwhile in the background, Next will statically generate the requested path HTML and JSON. The browser will then receive that HTML and JSON and swap from a "fallback" page to a rendered one.

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 `shared/staticPaths.ts` file.

05-next-ssg/step-5/shared/staticPaths.ts

```
import { EntityId } from "./types"

type PostStaticParams = {
  id: EntityId
}

type PostStaticPath = {
  params: PostStaticParams
}

const staticPostsIdList: EntityId[] = [1, 2, 3, 4, 5, 6, 7, 8, 9]

export const postPaths: PostStaticPath[] = staticPostsIdList.map(
  (id) => ({
    params: { id: String(id) }
  })
)
```

There, we generate a list of objects with structure `{params: { id: post.id }}` for each post. That way we're telling Next the ids of posts it should pre-render.

Then we finish our `Post` page component.

<<05-next-ssg/step-5/pages/post/[id].tsx[205]

Inside we use the `useRouter()` hook to get access to the `router` object. Then we check

[205] ./code/05-next-ssg/step-5/pages/post/.examples/id.tsx

if `router.isFallback` is `true`. If so, it means that this post hasn't been pre-rendered, so we render a `Loader` component. If not we render a `PostBody` component.

Loader Component

For loader we use a block with `Loading...` text inside.

05-next-ssg/step-5/components/Loader/Loader.tsx

```
import { Container } from "./style"

export const Loader = () => {
  return <Container>Loading...</Container>
}
```

And the styles for it:

05-next-ssg/step-5/components/Loader/style.ts

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

export const Container = styled.div`
  font-family: ${(p) => p.theme.fonts.accent};
`;
```

PostBody Component

To render the whole post we create a `PostBody` component. It will take `post` as a prop.

05-next-ssg/step-5/components/Post/PostBody.tsx

```
import Link from "next/link"
import { Post } from "../../shared/types"
import { Title, Figure, Content, Meta } from "./PostBodyStyle"

type PostBodyProps = {
  post: Post
}
```

...and return a block with main post info first:

05-next-ssg/step-5/components/Post/PostBody.tsx

```
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 post meta info last:

05-next-ssg/step-5/components/Post/PostBody.tsx

```
      <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>
  )
}
```

We use `dangerouslySetInnerHTML` on `Content` component only for simplicity's sake. 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 or other security vulnerabilities.

In `Meta` we also create a link to the category page. This is the page we're going to create next. For now, let's try and run `yarn dev` to see what a post page will look like.

Statically generated post page

And it's done!

Category Page

The final step before our application is done is to create a category page. It will contain a list of posts from a given category. Again, we will start with an API.

Category API

Here, we create a new endpoint for `/categories/:id` URL. We use `id` as a category identifier and search for posts that have a `category` field with the same value.

05-next-ssg/step-6/server/index.ts

```
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 use a list of found posts three times, just to make it a bit bigger than it is, to make the example simpler. In a real-world API, we would make a request to a database instead and pull out a list of category posts from there.

Next, we create a function for fetching that data in `api/category.ts`.

05-next-ssg/step-6/api/category.ts

```
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 an `EntityId` which is a category identifier, and returns a `Promise` of `Post` items list. And that's how we make our API ready!

Category Page Component

Next, we want to create a `Category` page component. First of all, let's design props for it. The `Category` component will take a list of `Post` items as a `posts` prop.

<<05-next-ssg/step-6/pages/category/[id].tsx[206]

Since we want this page to be pre-rendered as well, we create a `getStaticProps()` function. In that function, we `fetchPosts` and return a props object with `posts` property.

<<05-next-ssg/step-6/pages/category/[id].tsx[207]

As well as creating `getStaticProps()` we want to create `getStaticPaths()` function. Again, we make the `fallback` property equal to `true` just to make sure that no page returns 404 when it is not pre-rendered.

<<05-next-ssg/step-6/pages/category/[id].tsx[208]

Static paths for this page will be a list of objects with `{params: { id: category }}`. By default, we include three categories to pre-render which are listed in `categoriesToPreRender`.

05-next-ssg/step-6/shared/staticPaths.ts

```
const categoriesToPreRender: Category[] = [
  "Science",
  "Technology",
  "Arts"
]

export const categoryPaths: CategoryStaticPath[] = categoriesToPreRende\
r.map(
  (category) => ({ params: { id: category } })
)
```

[206] ./code/05-next-ssg/step-6/pages/category/.examples/id.tsx
[207] ./code/05-next-ssg/step-6/pages/category/.examples/id.tsx
[208] ./code/05-next-ssg/step-6/pages/category/.examples/id.tsx

And finally, we check if the page is not pre-rendered and render `Loader` component, or render `Section` otherwise.

<<05-next-ssg/step-6/pages/category/[id].tsx[209]

Updating Section

Now, we use our `Section` component both on the main page and on a category page. On the main page, there are only three post cards. Let's create a link "More in this section" for the main page so that a user would be able to go to a section page right away.

Firstly, let's update `SectionProps` and append `isCompact` optional field. It will determine whether to render the "More" link or not.

05-next-ssg/step-6/components/Section/Section.tsx

```
import Link from "next/link"
import { Post as PostType } from "../../shared/types"
import { PostCard } from "../Post"
import { Grid, Title, MoreLink } from "./style"

type SectionProps = {
  title: string
  posts: PostType[]
  isCompact?: boolean
}
```

Then, we access this prop:

[209] ./code/05-next-ssg/step-6/pages/category/.examples/id.tsx

05-next-ssg/step-6/components/Section/Section.tsx

```
export const Section = ({
  title,
  posts,
  isCompact = false
}: SectionProps) => {
```

And conditionally render a `Link` component which leads to a given category.

05-next-ssg/step-6/components/Section/Section.tsx

```
  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>
  )
```

Again, we use `passHref` to force the `Link` component to pass `href` further on a `MoreLink`, which is a styled link.

05-next-ssg/step-6/components/Section/style.ts

```
export const MoreLink = styled.a`
  margin: -20px 0 30px;
  display: inline-block;
  vertical-align: top;
`
```

Now, when `isCompact` is not `true` we won't see this link. However, it is not done yet, because we have to update `Feed` to make sure that this link is being rendered on the main page. Let's do that!

05-next-ssg/step-6/components/Feed/Feed.tsx

```
  return (
    <>
      {categories.map((category) => {
        const inSection = posts.filter(
          (post) => post.category === category
        )

        return (
          <Section
            key={category}
            title={category}
            posts={inSection}
            isCompact
          />
        )
      })}
    </>
  )
```

Here, we append `isCompact` prop on `Section` components inside of `map()`. Thus, all the sections in `Feed` would render `MoreLink` and a user would have access to a category page.

Adding Breadcrumbs

The last thing we would want to show to our users is `Breadcrumbs` on a post page. It is a component that contains a "links path" from the main page to a current page. In our case, it will have a link to the main page, and a link to a category that the current post is in.

Let's create a new component. We start with a type `BreadcrumbsProps` and getting access to `post` prop.

05-next-ssg/step-6/components/Breadcrumbs/Breadcrumbs.tsx

```
import Link from "next/link"
import { Post } from "../../shared/types"
import { Container } from "./style"

type BreadcrumbsProps = {
  post: Post
}
```

Then we render a `Container` (styled `nav` element) inside of which we place a couple of links.

05-next-ssg/step-6/components/Breadcrumbs/Breadcrumbs.tsx

```
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>
  )
}
```

And the styles for it:

05-next-ssg/step-6/components/Breadcrumbs/style.ts

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

export const Container = styled.nav`
  & > * {
    margin-right: 0.3em;
  }
`;
```

Then we want to render it in the `PostBody` component, right above the post title.

05-next-ssg/step-6/components/Post/PostBody.tsx

```
    <div>
      <Breadcrumbs post={post} />
      <Title>{post.title}</Title>
```

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. But what if we wanted to use some dynamic content on our pages, like, say, comments?

First of all, we couldn't use SSG anymore, because users can write comments after we build our site, and we would lose some data. That brings in Server-Side Rendering, SSR.

Updates on Each Request

As we remember, with SSR, pages get updated on each request[210]. 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 time those pages that have comments on them won't just be rendered once at build time. Instead, they will be rendered "live", at request time on a server.

[210] 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
}
```

It contains: - `id` - the comment id - `author` - the name of the author of the comment - `content` - comment text - `time` - string with relative time (in a real API it would be a timestamp or ISO string, but for our example, just a string will be fine) - `post` - the post id which this comment is written for

In our `server/index.ts` we create another endpoint for getting comments for a given post.

05-next-ssg/step-7/server/index.ts

```
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 the post id from a URL and filter through the `comments` array which we import above.

05-next-ssg/step-7/server/index.ts

```
const comments = require("./comments.json")
```

Comment Type

Now when the server API is ready let's create client code. First of all, we want to describe comments in TypeScript terms. For that, we create a new type in `types.ts` called `Comment`.

05-next-ssg/step-7/shared/types.ts

```
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 types, and refers to two new types: - `Person` - in our case this is just a `string`, but it could be a more complicated data structure as well - `RelativeTime` - again, for this example just a `string`

When we described types, we can create a `fetchComments()` function, which will take a `postId` as an argument and return a `Promise<Comment[]>`.

05-next-ssg/step-7/api/comments.ts

```
export async function fetchComments(
  postId: EntityId
): Promise<Comment[]> {
  const res = await fetch(`${config.baseUrl}/comments/${postId}`)
  return await res.json()
}
```

Add Comments to Page

Then, let's create components to render our comments on the page. We will need three things: - `Comment` component for an actual single comment - `CommentForm` for letting users send new comments - `Comments` container which will wrap those

Single Comment Component

Let's again start with imports:

05-next-ssg/step-7/components/Comment/Comment.tsx

```
import React from "react"
import { Comment as CommentType } from "../../shared/types"
import { Container, Author, Body, Meta } from "./style"

type CommentProps = {
  comment: CommentType
}
```

A `Comment` component will take a comment as a prop. The markup for it will contain the author's name, comment text, and the date it was created.

05-next-ssg/step-7/components/Comment/Comment.tsx

```
export const Comment: React.FC<CommentProps> = ({ comment }) => {
  return (
    <Container>
      <Author>{comment.author}</Author>
      <Body>{comment.content}</Body>
      <Meta>{comment.time}</Meta>
    </Container>
  )
}
```

We will use this code to style our comments.

05-next-ssg/step-7/components/Comment/style.ts

```
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;
`
```

Comment Form

Next, we want to create a form for our users to send comments. For that, we create another component called `CommentForm`. As props, we pass a `post` id to figure out which post should have this comment attached later.

05-next-ssg/step-7/components/CommentForm/CommentForm.tsx

```
import React, { useState, FormEvent } from "react"
import { EntityId } from "../../shared/types"
import { Form } from "./style"
import { submitComment } from "../../api/comments"

type CommentFormProps = {
  post: EntityId
}
```

Inside we create three fields for the local state: loading, name, and value. The `name` is the author's name, `value` is the comment text itself, and `loading` is the flag that is `true` if a comment is being submitted at the time.

05-next-ssg/step-7/components/CommentForm/CommentForm.tsx

```
export const CommentForm: React.FC<CommentFormProps> = ({ post }) => {
  const [loading, setLoading] = useState<boolean>(false)
  const [value, setValue] = useState<string>("")
  const [name, setName] = useState<string>("")
```

From this component, we return a `form` element with an `input` and a `textarea` inside.

05-next-ssg/step-7/components/CommentForm/CommentForm.tsx

```
  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>
  )
```

Also, we create an `async` function which should be called when the form is submitted. We first prevent default behavior using `e.preventDefault()`, which prevents the form from being submitted the "classic" way via HTTP. Then we set the `loading` flag to be true, which replaces the submit button with "Submitting..." label, and `submitComment()`.

After we get a response from the server we check if the status equals `201` (meaning that something has been created) and if so we refresh the page to get fresh comments.

05-next-ssg/step-7/components/CommentForm/CommentForm.tsx

```
  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()
    }
  }
```

We will do it without reloading the page later. Right now let's focus on creating the API for submitting comments.

API for Adding Comments

Our function `submitComment()` looks like:

05-next-ssg/step-7/api/comments.ts

```
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 specify the `postId` in the URL of the endpoint that we send the request to.

On the backend, we create a new comment object and response at this endpoint with `201` status. Right now the code for creating a comment is more mock than real code. In the real API, we would save the comment in the database, but for this example, we keep the `comments` array in memory and `push()` a new value to it when we submit a comment.

05-next-ssg/step-7/server/index.ts

```
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 on Page

To inject comments on a page we want to create a wrapper for the comments section. Let's create a `Comments` component. For starters, we create the `CommentsProps` type. A `comments` field defines an array of comments to render and a `post` field contains a current post id.

05-next-ssg/step-7/components/Comments/Comments.tsx

```
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 a form below that list.

05-next-ssg/step-7/components/Comments/Comments.tsx

```
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>
  )
}
```

We use this code to style this component.

05-next-ssg/step-7/components/Comments/style.ts

```
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);
`
```

Now we're ready to add a comments section on the page. We change the `PostProps` type for the post page to make it contain the `comments` field, like so:

<<05-next-ssg/step-7/pages/post/[id].tsx[211]

Then, we change the component itself to render the `Comments` component. We access `comments` prop from props and pass them as a prop to `Comments`.

<<05-next-ssg/step-7/pages/post/[id].tsx[212]

We provide an `id` prop, to make sure that when a user submits a comment, their browser would scroll right to this section after the reload.

The last thing to do is to convert this page from being statically generated to being rendered on a server.

Converting Statically Generated Page to Rendered on Server

In order to make a page SSR-ed we have to export[213] a `getServerSideProps()` function.

We cannot[214] use it along with the `getStaticPaths()` function, so we have to remove `getStaticPaths()`.

Then we create the `getServerSideProps()` function. Notice that it is typed with `GetServerSideProps` type. Inside, we not only fetch the current post, but `fetchComments()` as well.

<<05-next-ssg/step-7/pages/post/[id].tsx[215]

Thus, comments will be fetched on every page request and there will not be any missing data.

[211] ./code/05-next-ssg/step-7/pages/post/.examples/id.tsx
[212] ./code/05-next-ssg/step-7/pages/post/.examples/id.tsx
[213] https://nextjs.org/docs/basic-features/data-fetching#getserversideprops-server-side-rendering
[214] https://nextjs.org/docs/basic-features/data-fetching#use-together-with-getstaticprops
[215] ./code/05-next-ssg/step-7/pages/post/.examples/id.tsx

Connecting Redux

Now the post page reloads after a user submits a comment. Let's try to make it work without reloads. In order to do that we would need some kind of store on a client. For this purpose, we will use Redux.

There is a package[216] called `next-redux-wrapper` which can help us connect Redux with Next more easily.

First, let's add all the packages needed:

```
yarn add next-redux-wrapper react-redux @types/react-redux
```

We don't add `redux` itself, because it is included in the dependencies for `next-redux-wrapper`, but it requires[217] `react-redux` as a peer dependency, so we have to install it separately.

Next, we can configure our store.

Configuring Store

Let's take a look at the `store/index.ts` file:

05-next-ssg/step-8/store/index.ts

```
import { 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
}
```

[216] https://github.com/kirill-konshin/next-redux-wrapper
[217] https://github.com/kirill-konshin/next-redux-wrapper#installation

```
const combinedReducer = combineReducers({ post, comments })
const makeStore: MakeStore<State> = () => createStore(combinedReducer)

export const store = createWrapper<State>(makeStore, {
  debug: true
})
```

First of all, there is a `State` type, which defines the structure of our future state. In our case, we will only need a store for `comments` and a current `post` since only a post page is dynamic.

`PostState` is an `Optional<Post>` from `store/post.ts`. It is optional because later we will use it in reducer and default state cannot be any post yet, thus we will define it as `null`.

05-next-ssg/step-8/store/post.ts

```
export type PostState = Optional<Post>
```

We need `Optional` type, so let's create it:

05-next-ssg/step-8/shared/types.ts

```
export type Optional<TEntity> = TEntity | null
```

`CommentsState` is an array of `Comment` items from `store/comments.ts`:

05-next-ssg/step-8/store/comments.ts

```
export type CommentsState = Comment[]
```

Then, there is a `combinedReducer`, which contains the definition for `post` and `comments` reducers. We will cover them shortly.

`makeStore()` is a function which creates a redux-store. Notice the `MakeStore` type there - this type will help `createWrapper()` function create a wrapper that we will be able to use with our components.

Actions for Comments

Let's define types for reducer and actions for `comments` state.

05-next-ssg/step-8/store/comments.ts

```
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 an `UpdateCommentsAction` interface which extends `AnyAction` from `redux`. We set the `type` field to be a type of `UPDATE_COMMENTS_ACTION` constant. The second field in this action is `comments` which is an array of `Comment`.

K> Notice that 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 this `UpdateCommentsAction` or `HydrateAction`, which is defined in `store/hydrate.ts`:

05-next-ssg/step-8/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`, which is imported from `next-redux-wrapper` package. This is a special action that must be used[218], in order to properly reconcile the hydrated state on top of the existing state.

Each reducer must have a handler for this action. Because each time when pages that have `getServerSideProps` are opened by a user the `HYDRATE` action will be dispatched.

Reducer for Comments

With that in mind, let's create our `comments()` reducer.

05-next-ssg/step-8/store/comments.ts

```
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
  }
}
```

[218] https://github.com/kirill-konshin/next-redux-wrapper#usage

Notice the `HYDRATE` case in it. Inside we see the familiar optional chaining[219] operator `?`, but later there is `??`. This is nullish coalescing[220].

When the whole expression `action.payload?.comments` is `null` or `undefined` nullish coalescing will tell TypeScript to use a fallback value - in our case an empty array.

It is okay here to simply replace the whole state with the fresh one when hydration happens because we need to load the new comments for the new post. However, sometimes this is not the case, and you should consider comparing states and merging them[221].

The second case handles the `UpdateCommentsAction` calls. It replaces comments with those in the payload.

As a default value for the state, we provide an empty array.

Reducer for Post

Next, the `post()` reducer.

05-next-ssg/step-8/store/post.ts

```
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
```

[219] https://www.typescriptlang.org/docs/handbook/release-notes/typescript-3-7.html#optional-chaining
[220] https://www.typescriptlang.org/docs/handbook/release-notes/typescript-3-7.html#nullish-coalescing
[221] https://github.com/kirill-konshin/next-redux-wrapper#state-reconciliation-during-hydration

The `UpdatePostAction` interface extends `AnyAction` and defines `type` field to be type of `UPDATE_POST_ACTION`, and `post` to be type of `Post`. A union `PostAction` contains either `HydrateAction` or `UpdatePostAction`.

The reducer again contains a case for `HYDRATE` action, and for `UPDATE_POST_ACTION`. When hydration happens we either take the `post` from `action.payload`, or set `null` as a value for the state. Besides we provide `null` as a default value for the state - that's why we needed `Optional<>` type.

05-next-ssg/step-8/store/post.ts

```
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
  }
}
```

On an `UpdatePostAction` call we replace the current value with the new one to render a freshly loaded post.

Changing Custom App Component

When our store is created we can connect it to the Next `_app`. First of all, we don't default export `MyApp()` function anymore. Instead we default export a wrapped version of it:

05-next-ssg/step-8/pages/_app.tsx

```
export default store.withRedux(MyApp)
```

This `store` is the wrapper which we created earlier:

05-next-ssg/step-8/pages/_app.tsx

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

The `MyApp()` function itself stays the same, but we now need[222] to specify[223] `MyApp.getInitialProps()` static method.

05-next-ssg/step-8/pages/_app.tsx

```
MyApp.getInitialProps = async ({ Component, ctx }: AppContext) => ({
  pageProps: {
    ...(Component.getInitialProps
      ? await Component.getInitialProps(ctx)
      : {})
  }
})
```

Here we call page-level `getInitialProps()`. This is required to correctly collect the data from the store.

Updating Post Page

Now we need to update post page. Since we want to store comments and post data in the redux-store, we need to connect this page to the store.

For accessing the store we're going to use the `useSelector()` hook from `react-redux` package. The whole page component will look like this:

<<05-next-ssg/step-8/pages/post/[id].tsx[224]

We access the whole state and destructure it into `post` and `comments` objects, which then pass as props further. Since the post data can be `null` we render the `Loader` component if there is no post yet to show.

[222] https://github.com/kirill-konshin/next-redux-wrapper#app-and-getserversideprops-or-getstaticprops-at-page-level
[223] https://github.com/kirill-konshin/next-redux-wrapper#pagegetinitialprops
[224] ./code/05-next-ssg/step-8/pages/post/.examples/id.tsx

Right now if we start our project it won't work, because Next doesn't yet 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.

<<05-next-ssg/step-8/pages/post/[id].tsx[225]

Here we use `store.getServerSideProps()` function which takes a callback inside of which 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 right in `Post` component's props we `dispatch()` actions, that update our store with this data.

Notice that `Post` now doesn't take any props at all. All the data it renders it gets from the store accessed via `useSelector()` hook.

Making Comment Form Work Without Reloads

For the comment form to work without page reloads we need to `dispatch()` some action that will update the store instead of reloading the page. Let's imagine how it should work.

When we submit a comment on a server, we want to get the data to refresh the comments section on a page. Let's make our server respond not with the status `201`, but return a list of comments for a current post instead.

> In the more canonical version of REST API `post` requests should return `201` and an ID of the created entity. In our case, we make our response less canonical, but more convenient for us to work with by returning the whole list of comments instead.

So in our `server/index.ts` we need to update the `return` statement in the post method. We return all the comments for the post with the given `postId`.

[225] ./code/05-next-ssg/step-8/pages/post/.examples/id.tsx

05-next-ssg/step-8/server/index.ts

```
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))
})
```

In the `CommentForm` component we use the `useDispatch()` hook to get access to `dispatch()` function. This `dispatch()` is going to be used to dispatch actions when the request has finished.

05-next-ssg/step-8/components/CommentForm/CommentForm.tsx

```
  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 (
```

Here from the `response` from the server, we access all the `comments`. Then we use `setValue()` and `setName()` to clear the form, and if the request succeeded we dispatch `UPDATE_COMMENTS_ACTION` with the list of `comments` as a payload. This will update the comments store and re-render the comments section on this 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[226] 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:

05-next-ssg/step-9/components/Post/PostBody.tsx

```
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:

[226] https://nextjs.org/docs/api-reference/next/image

05-next-ssg/step-9/components/Post/PostBody.tsx

```
      <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.

K> 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`:

05-next-ssg/step-9/components/Post/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[227]. Create a file called `next.config.js` in the root of the project directory and add this configuration:

05-next-ssg/step-9/next.config.js

```
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.

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

[227] https://nextjs.org/docs/api-reference/next.config.js/introduction

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:

05-next-ssg/step-9/components/Post/PostCard.tsx

```
import Link from "next/link"
import Image from "next/image"
import { Post as PostType } from "../../shared/types"
import { Card, Figure, Title, Lead } from "./PostCardStyle"
```

Then replace the old `img` with the new component:

05-next-ssg/step-9/components/Post/PostCard.tsx

```
        <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`:

05-next-ssg/step-9/components/Post/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 though, we won't see any build artifacts in a project directory. That's because by default Next puts those in a `.next` directory.

Next offers an option to export generated code[228] in `out` directory via `next export` script, though we would want to change the build destination directory to ours—`build`.

Notice that `next/image` works only with a next application live-running on a server[229] via `next start`. If we want to export our app as a static site we need to either specify a loader[230] 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.

[228] https://nextjs.org/docs/advanced-features/static-html-export
[229] https://github.com/vercel/next.js/issues/18356
[230] https://nextjs.org/docs/basic-features/image-optimization#loader

One of the configuration options is `distDir`[231] - it is the name to use for a custom build directory. In our case, we want to use `build` for that:

05-next-ssg/step-10/next.config.js

```
module.exports = {
  distDir: "build"
}
```

Now, we can run `yarn serve` in one terminal window to set up a backend server and `yarn build` in another. After the project is built you will see a bunch of files in `build` directory.

Notice the `BUILD_ID` file - it contains a hash of a current build. This hash is the name of a directory inside of `build/server/static` which contains current build artifacts like pages' HTML and JSON.

Notice that all the pages that could be statically generated (`Section`, `Front`) have `.html` files associated with them. Although pages that can only be rendered on a server (`Post`) have only `.js` files.

Conclusion

In this chapter, we learned how to create applications using the Next.js framework and how to use Static Site Generation for pre-rendering pages. We connected the app to the Redux store and learned how to optimize images using built in Next components.

[231]https://nextjs.org/docs/api-reference/next.config.js/setting-a-custom-build-directory

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 specify exactly the fields of data you want to get from the backend.

Let's say you work with a Pokemon API and you want to fetch the information about a pokemon.

You would send the query containing the fields you are interested in:

```
query {
  pokemon(name: "Pikachu") {
    id
    number
    name
  }
}
```

In response you would get an object where the fields will be filled with data:

```
{
  "data": {
    "pokemon": {
      "id": "UG9rZW1vbjowMjU=",
      "number": "025",
      "name": "Pikachu"
    }
  }
}
```

To be able to use GraphQL you need both the backend and frontend of your application to support it.

For the frontend there are a bunch of libraries available - all of them have React bindings:

- Relay[232] - is a library by Facebook released alongside GraphQL. It has quite a steep learning curve and might require some time to learn.
- Apollo[233] - is a platform that has client libraries for all the popular web frameworks and mobile platforms. It is popular and has an easy-to-learn API. We will use it in this chapter.
- URQL[234] - a GraphQL library by Formidable labs. Also has a nice and easy to learn API.

All of them provide a convenient wrapper to make GraphQL requests. But you can also perform GraphQL requests manually. After all, GraphQL is based on HTTP protocol.

For example, try to run this `cURL` script in the terminal:

[232] https://relay.dev/
[233] https://www.apollographql.com/
[234] https://formidable.com/open-source/urql/docs/

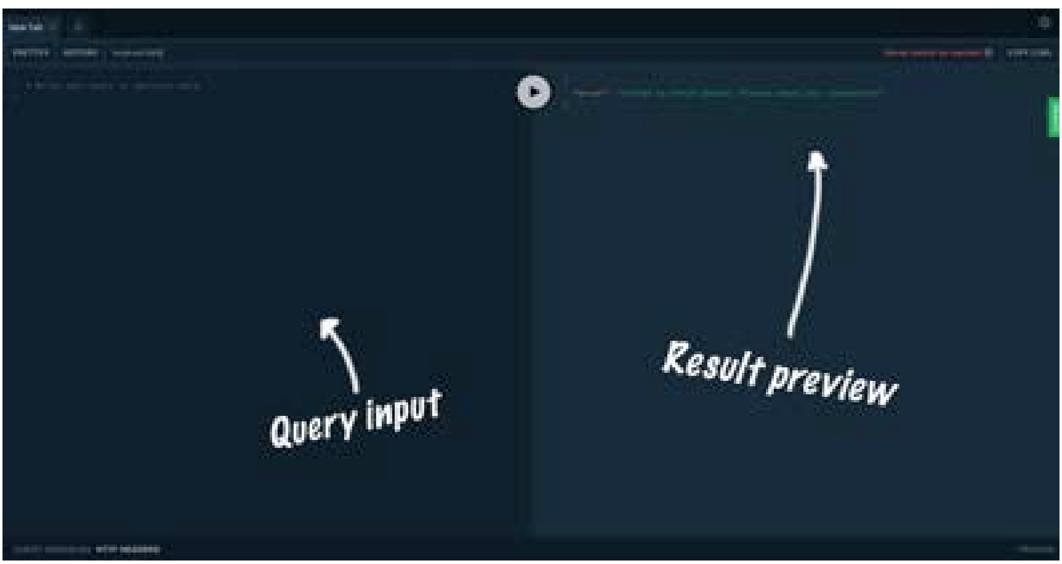
```
curl 'https://graphql-pokemon2.vercel.app/?' \
  -H 'content-type: application/json' \
  --request POST \
  --data '{"query":"query { pokemon(name: \"Pikachu\") { id number name\
 } }","variables":null}' \
```

The server will respond with a JSON formatted object.

```
{"data":{"pokemon":{"id":"UG9rZW1vbjowMjU=","number":"025","name":"Pika\
chu"}}}
```

Almost all the GraphQL server implementations also provide a schema explorer.

For example, when you launch Apollo GraphQL server you'll have a `__graphql` endpoint where you'll see the following interface:



Apollo GraphQL Schema Explorer

Here you can enter a query on the left, press the execute button, and get the result on the right pane.

This feature allows you to explore the provided GraphQL schema easily.

You can play with the Pokemon example API here[235].

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 defined by Roy Fielding, a computer scientist, who presented the REST principles in his Ph.D. dissertation[236] in 2000.

Here are the key characteristics of a REST API:

- Client-server architecture[237] Client-server architecture means that the user interface concerns should be separated from the data storage concerns to improve the user interface portability across multiple platforms.
- Statelessness[238] A stateless server does not persist any information about the user who uses the API.
- Cacheability[239] REST API responses must define themselves as cacheable or non-cacheable to prevent clients from providing any inappropriate data that can be used in future requests.
- Layered System[240] A Layered system means that if a proxy or load balancer is placed between client and server, the connections between them shouldn't be affected and the client won't know if he's connected to the end server or not.
- Uniform Interface[241] A Uniform interface suggests that should be a uniform way of interacting with a given server despite the application type (website, mobile app). The main guideline is that each individual resource has to be identified on request.

When you create a REST API you define HTTP endpoints for each of your resources. For example, if you want to be allowed to Create, Read, Update and Delete users in your application then it would look like this:

[235] https://graphql-pokemon2.vercel.app/?
[236] https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
[237] https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_1_2
[238] https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_1_3
[239] https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_1_4
[240] https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_1_6
[241] 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 new user
GET http://api.example/users/:id  //Get the user by id
PUT http://api.example/users/:id  //Update the user by id
DELETE http://api.example/users/:id  //Delete the user by id
```

If you have some associated data, for example, if your users 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 the repositories of 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 with their associated repositories
- make two subsequent calls to the API to first fetch the users and then their repositories.

As you can see this creates overhead - you have to write more code to extend your API.

This is why in 2015 Facebook started developing GraphQL[242].

GraphQL allows the client to specify what data you need to get from the server.

When you use GraphQL, you need to define the complete schema on the backend and implement special functions - resolvers - that will fill the schema with data.

This approach allows you to make fewer assumptions about the client's needs. You won't have to define additional endpoints when your client needs more data.

It also fixes the problem of over-fetching. Now your client can specify if it needs additional data in the query.

Overall GraphQL requires less work to define a decent API, and also it is easier to maintain.

[242] https://engineering.fb.com/core-data/graphql-a-data-query-language/

Currently, a lot of services provide GraphQL versions for their APIs. Here are a few for example:

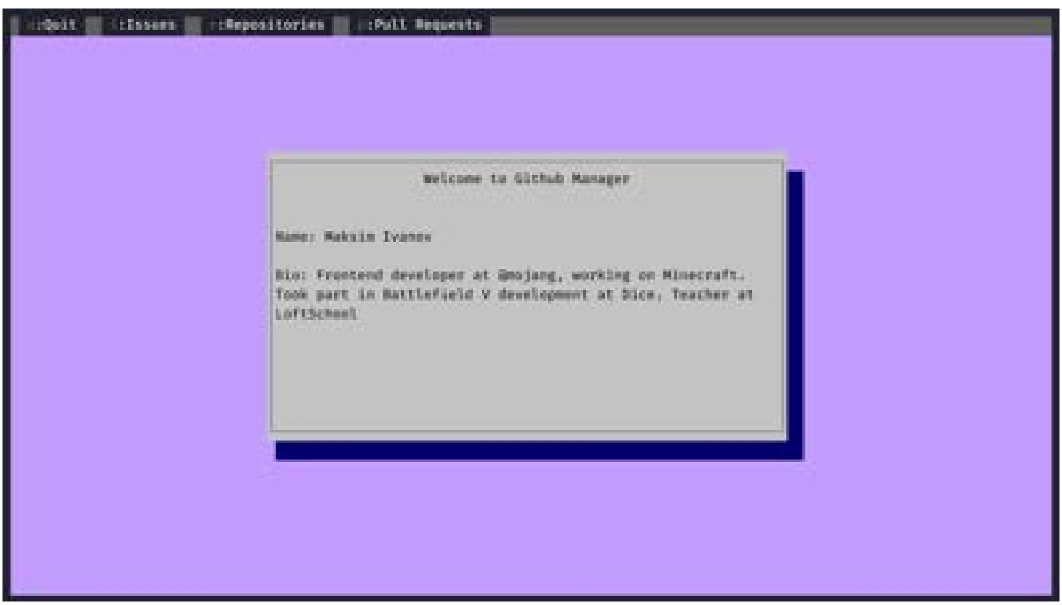
- Facebook
- Instagram
- Github

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 the owned repositories, issues, and pull requests.

The app will have a graphical interface made using the curses library.

On the main screen, you can see the information about the currently logged-in user.



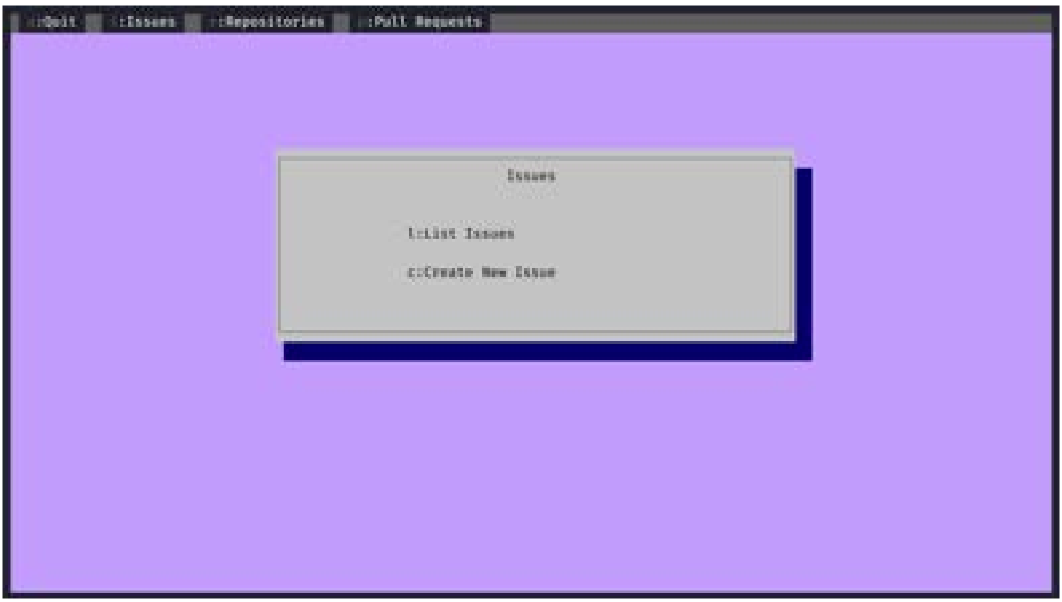
main screen image

There is a navigation bar on the top with a list of resources you can perform operations with:

- Repositories
- Issues
- Pull Requests

You can switch between the screens by pressing the associated letters on the keyboard.

For example, you can open the Issues tab by pressing `i`.



Issues screen

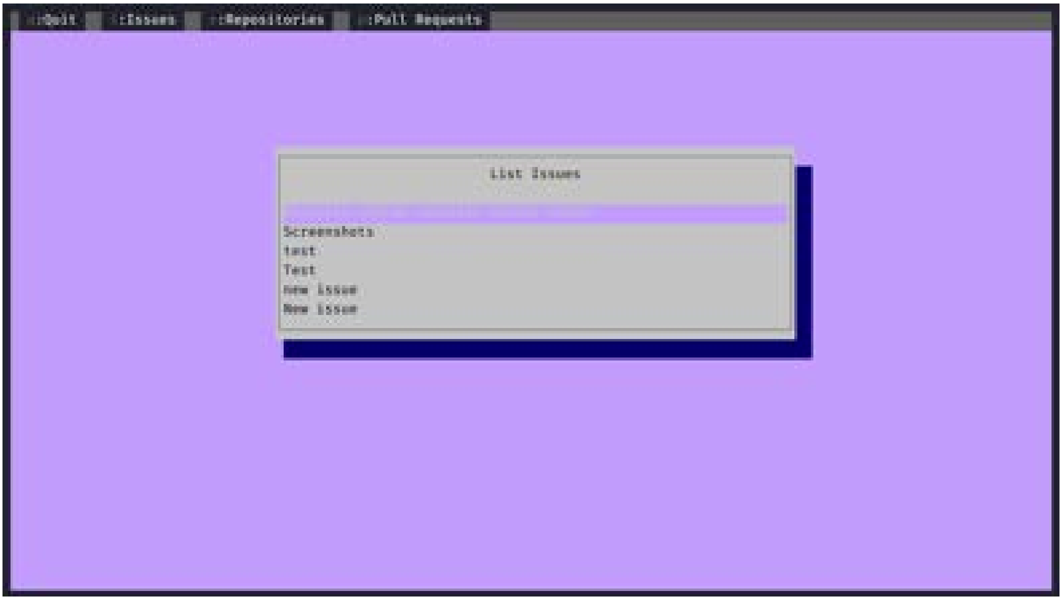
You will be presented with a window giving you two options:

- Press `c` to create a new issue
- Press `l` to see the list of existing issues

If you press `c`, it will open a form with the new issue title and description. Every issue belongs to a specific repository, so you'll also have to specify the repository name.

Create Issue Screen

If you press `l` you will see a list of available issues. You can select the issue using the mouse, arrow keys, or `j` and `k` letters like in Vim. After you've selected an issue you can press `Enter` or click on it to open the browser and navigate to the selected issue.

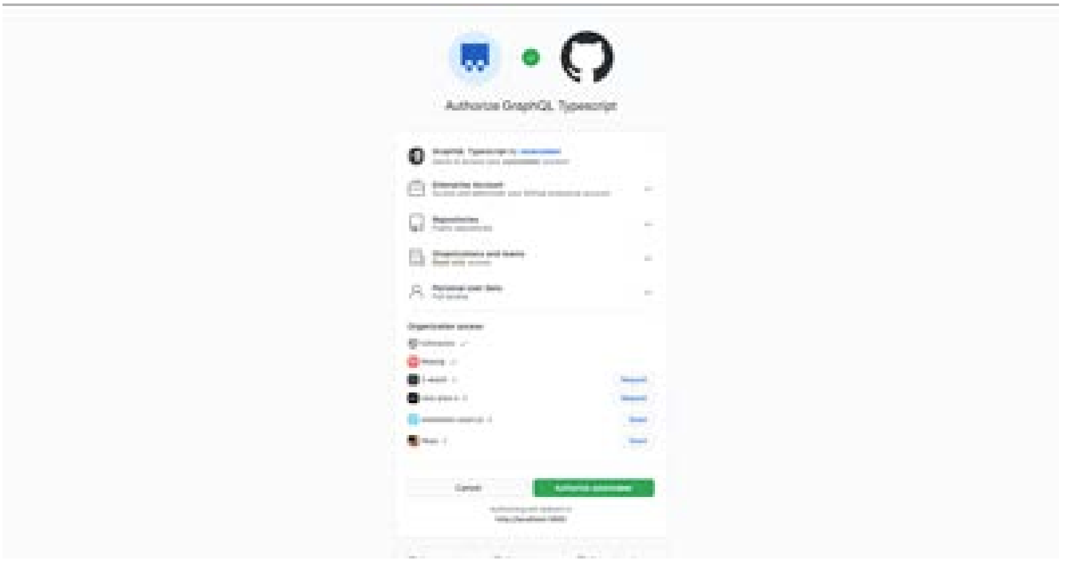


List Issues Screen

Similarly, you can operate Pull Requests and Repositories.

Github requires authentication to make the API calls. In our app, we'll be using the OAuth2 authentication flow.

When you launch your application for the first time, it will open the browser and present you with the GitHub authentication page:



GitHub Authentication Screen

After you authenticate, it will store the authentication token and won't require you to repeat this process unless you remove it from the 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

Preview The Final Result

A complete code example is located in `code/06-graphql/completed`.

Unzip the archive that comes with this book and `cd` to the app folder.

```
cd code/06-graphql/completed
```

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

```
yarn && yarn start
```

It will open the browser window where you'll need to log in to GitHub and authorize the app to get access to your GitHub resources.

After that's done you can try to create some issues, pull-requests, or repositories.

Setting Up The Project

Unlike the projects in the previous chapters, this one runs in the terminal and not in the browser.

It will be a NodeJS application that we'll write in Typescript. We'll use the `react-blessed` custom React renderer to be able to render the text-based GUI in the terminal.

Begin by creating a new folder for the project. Let's call it `github-client`:

```
mkdir github-client
cd github-client
```

After you create the folder and open it, run `npm init` to generate the `package.json` file.

```
npm init -y
```

Running Typescript in The Console

There are two major ways to run TypeScript in the console:

- Precompile it 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.

Install `babel-node` as a dev dependency:

```
yarn add babel-node
```

Add the `start` script that will launch the `babel-node` with inspector enabled. We'll need it to be able to use the debugger and see the logs in the console:

06-graphql/step-1/package.json

```
  "scripts": {
    "start": "babel-node --inspect src/index.tsx --extensions \".js,.ts\
,.jsx,.tsx,\""
  },
```

Here we pass the `--inspector` param to enable the debugger.

Now we can install the dependencies:

```
yarn add apollo apollo-boost @apollo/react-hooks react react-blessed re\
act-devtools react-router ws open keytar graphql form-data dotenv cross\
-fetch blessed babel-plugin-transform-class-properties @babel/core @bab\
el/preset-env @babel/preset-react @babel/preset-typescript @babel/regis\
ter
```

We also need to install the types for some of the packages:

```
yarn add @types/react-blessed @types/react-router
```

Authenticating in GitHub

The first thing we need to do to be able to use the GitHub API is authenticate.

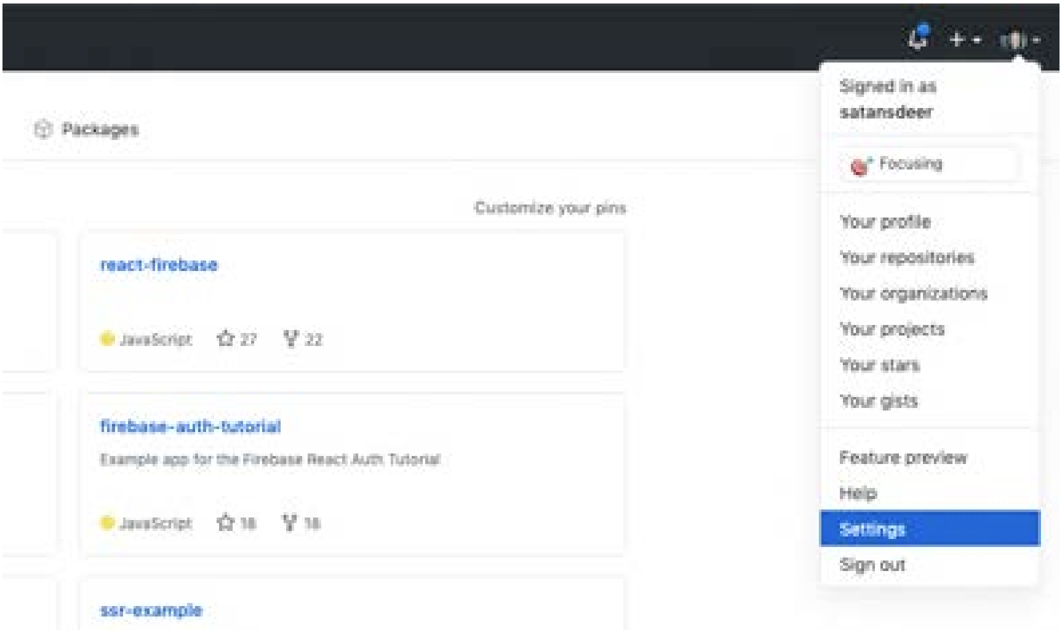
To communicate with the GraphQL server we'll need the OAuth token with the right scopes. We will follow the https://docs.github.com/en/developers/apps/authorizing-oauth-apps#web-application-flow[243].

[243] webapplicationflow

To enable the web authentication flow in our application we need to get the `client_id` and `client_secret`.

To get them you need to go to your GitHub profile and generate a new key.

First, click on your avatar in the top right corner, and then click the settings link:

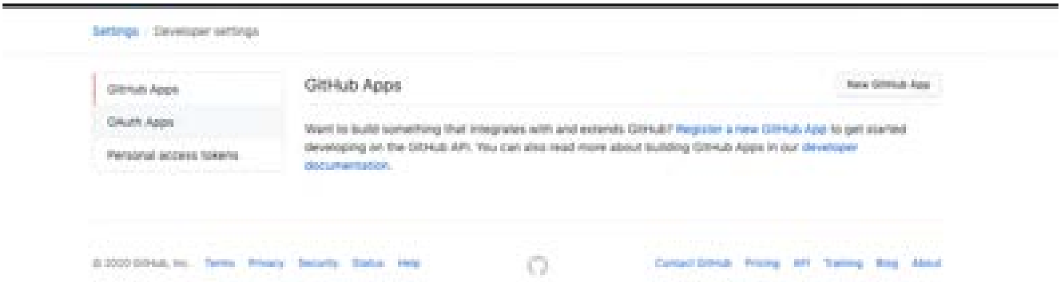


Profile Dropdown

Then on the "Settings" page go to "Developer Settings":

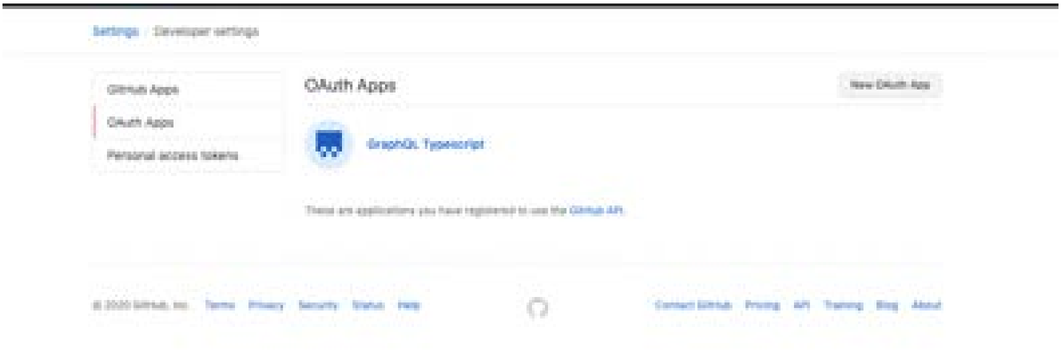
Developer Settings

On “Developer Settings” page select the “OAuth Apps”:



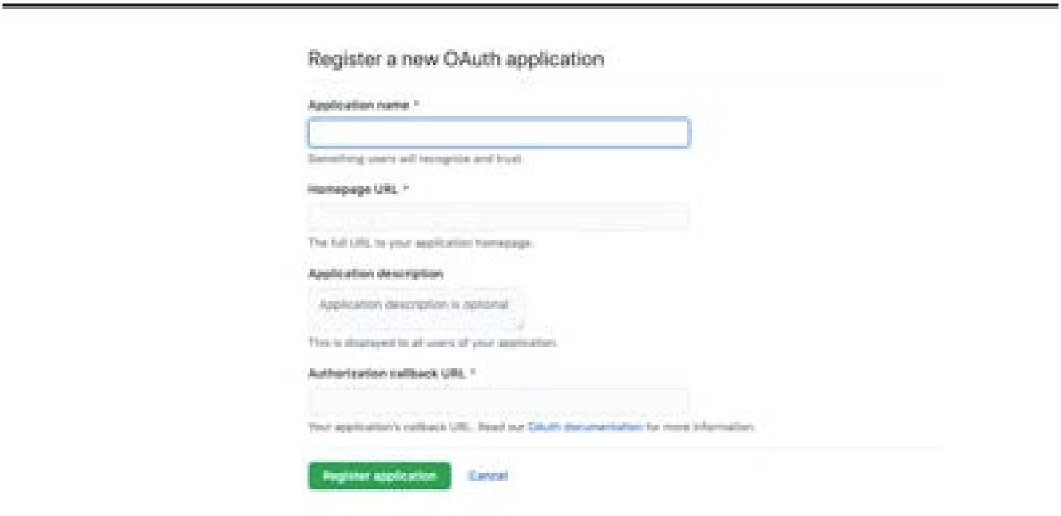
OAuth Applications

There click on the “New Github App” button:



New OAuth App

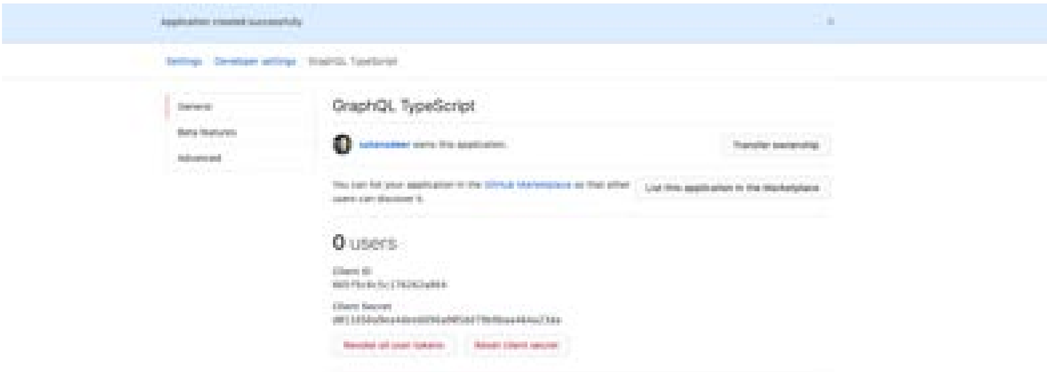
Now fill in all the data about your application:



New App 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 it in the keychain.



Application Keys

Now we can construct the url, so we can start writing the authentication code.

Create a new file `.env` and store your key there:

<<06-graphql/step-1/.env[244]

Initializing The Application

Create the `src` folder and define `index.tsx` file there.

First add the imports:

06-graphql/step-1/src/index.tsx

```
import React from "react"
import blessed from "blessed"
import { render } from "react-blessed"
import * as dotenv from "dotenv"
import { App } from "./App"
import { ErrorBoundary } from "./ErrorBoundary"
import { MemoryRouter } from "react-router"
```

Then we need to load the environment variables from the `.env` file:

[244] ./code/06-graphql/step-1/.env

06-graphql/step-1/src/index.tsx

```
dotenv.config()
```

Initialise the `blessed.screen`:

06-graphql/step-1/src/index.tsx

```
const screen = blessed.screen({
  autoPadding: true,
  smartCSR: true,
  sendFocus: true,
  title: "Github Manager",
  cursor: {
    color: "black",
    shape: "underline",
    artificial: true,
    blink: true
  }
})
```

Add the key press event listeners to be able to exit the application:

06-graphql/step-1/src/index.tsx

```
screen.key(["q", "C-c"], () => process.exit(0))
```

Now render the app:

06-graphql/step-1/src/index.tsx

```
const component = render(
  <ErrorBoundary>
    <MemoryRouter>
      <App />
    </MemoryRouter>
  </ErrorBoundary>,
  screen
)
```

Note that we don't have the `App` component yet - let's fix that. Create a new file `src/App.tsx` with the following code:

06-graphql/step-1/src/App.tsx

```
import React from "react"

export const App = () => {
  return (
    <blessed-box
      style={{
        bg: "#0000ff"
      }}
    >
      Hello React-Blessed
    </blessed-box>
  )
}
```

Make sure that you can launch the app. Run `yarn start`.

Authentication Context

Create the `src/auth` folder, and then inside it create a new file called `ClientProvider` with the following content.

First we make the imports:

06-graphql/step-1/src/auth/ClientProvider.tsx

```
import React, { FC, PropsWithChildren } from "react"
import { useState } from "react"
import ApolloClient from "apollo-boost"
import { ApolloProvider } from "react-apollo-hooks"
```

Define the `GITHUB_BASE_URL`:

06-graphql/step-1/src/auth/ClientProvider.tsx

```
const GITHUB_BASE_URL = "https://api.github.com/graphql"
```

Then we need to initialize the `ApolloClient`:

06-graphql/step-1/src/auth/ClientProvider.tsx

```
export const ClientProvider: FC<PropsWithChildren<{}>> = ({
  children
}) => {
  const [token, setToken] = useState<string>()

  const client = new ApolloClient({
    uri: GITHUB_BASE_URL,
    request: (operation) => {
      operation.setContext({
        headers: {
          authorization: `Bearer ${token}`
        }
      })
    }
  })
```

Here we need to get the authorization token and then provide it to the whole application through the context.

We'll use the `useEffect` hook to get the token. Add this after the `useState` hook:

06-graphql/step-1/src/auth/ClientProvider.tsx

```
  const [token, setToken] = useState<string>()

  useEffect(() => {
    const getToken = async () => {
      let key: any = await keytar.getPassword(
        "github",
        process.env.CLIENT_ID!
      )
      if (!key) {
        key = await getCode()
      }
      setToken(key)
    }
```

As you can see we are using the `getCode` function here. Let's define it. Create a new file `src/auth/getCode`. First add this import block there:

06-graphql/step-1/src/auth/getCode.ts

```
import * as http from "http"
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 the server that will handle our return url for GitHub auth:

06-graphql/step-1/src/auth/getCode.ts

```
const PORT = 3000
```

Define the `getCode` function:

06-graphql/step-1/src/auth/getCode.ts

```
export const getCode = () => {
  return new Promise((resolve, reject) => {
    http
      .createServer(function (req, res) {
        if (!req.url) {
          return
        }
        const { code } = url.parse(req.url, true).query
        res.writeHead(200, { "Content-Type": "text/plain" })
        res.write(`The code is: ${code}`)
        res.end()
      })
      .listen(PORT)
  })
}
```

Here we launch the server that will serve the return url for GitHub. We get the authentication code from the query params and store it in the `code` constant.

Now we need to send the `code` along with the `CLIENT_ID` and `CLIENT_SECRET` to the GitHub `login` endpoint in a `POST` request.

Define this self-invoking `async` function right after the server code:

06-graphql/step-1/src/auth/getCode.ts

```
        (async () => {
          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");

          fetch("https://github.com/login/oauth/access_token", {
            method: "POST",
```

```
            body: data,
            headers: {
              Accept: "application/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 you received as a response on our return url
- `state` - the random string we provided when starting the authentication
- `redirect_url` - the URL to send the user to after the authentication

Then we call the `fetch` method with the form data and set the `Accept` header to be `application/json`.

Now we'll add the code that will get the `access_token`:

06-graphql/step-1/src/auth/getCode.ts

```
          fetch("https://github.com/login/oauth/access_token", {
            method: "POST",
            body: data,
            headers: {
              Accept: "application/json",
            },
          })
            .then((res: any) => res.json())
            .then(async (data: any) => {
              await keytar.setPassword(
                "github",
                process.env.CLIENT_ID!,
                data.access_token
              );
              resolve(data.access_token);
```

```
        });
    })();
```

We get the JSON representation of the response and then we save the data.access_-token field to keytar.

keytar automatically detects what key storage is available in the system. On Mac OS it will use the Keychain Access app.

After we saved the password we resolve the promise object with the access_token.

Now we need to open the authentication page. Add this code after the server-launching code:

06-graphql/step-1/src/auth/getCode.ts

```
    open(
      `https://github.com/login/oauth/authorize?client_id=${process.env\
.CLIENT_ID}&scope=user%20read:org%20public_repo%20admin:enterprise&stat\
e=abc`
    );
```

Here we ask the user to allow us to fetch the user data, create new repositories, issues, and pull requests.

Now open src/index.tsx and import ClientProvider:

06-graphql/step-1/src/index.tsx

```
import { ClientProvider } from "./auth/ClientProvider"
```

Wrap the app into the ClientProvider:

06-graphql/step-1/src/index.tsx

```
const component = render(
  <ErrorBoundary>
    <MemoryRouter>
      <ClientProvider>
        <App />
      </ClientProvider>
    </MemoryRouter>
  </ErrorBoundary>,
  screen
)
```

Authenticating The ApolloClient

Open the `src/auth/ClientProvider.tsx` and import the `getCode` function:

06-graphql/step-1/src/auth/ClientProvider.tsx

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

Also add this check for the `token` before we return the layout:

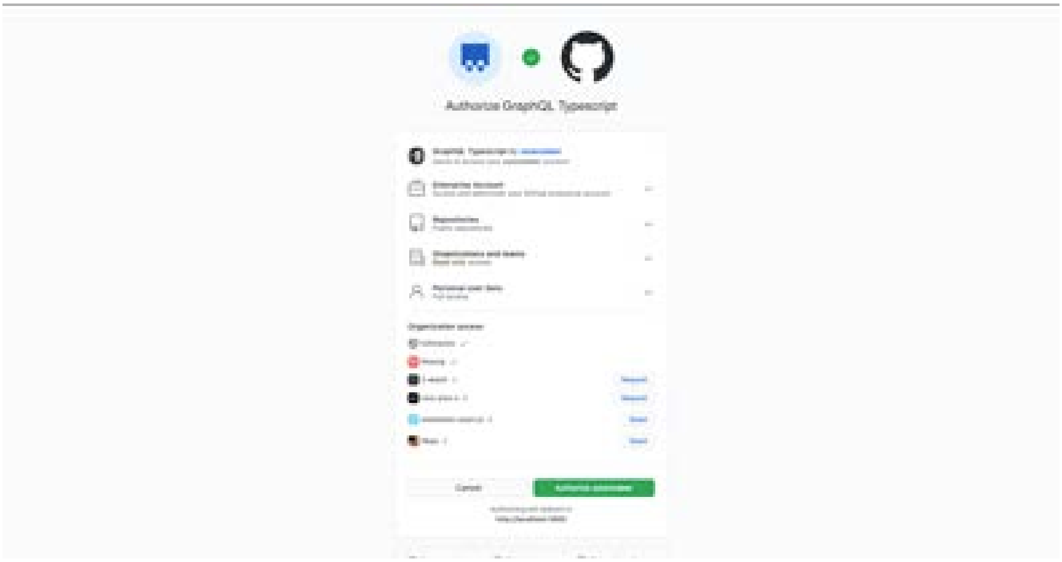
06-graphql/step-1/src/auth/ClientProvider.tsx

```
  if (!token) {
    return <>Loading...</>
  }
```

Run the app:

```
yarn start
```

You should see the following page in your browser.



Authentication page

Click the authentication button.

GraphQL Queries - Getting The User Data

Let's make our first query.

Create a new file `src/WelcomeWindow.tsx` - here we'll define the `WelcomeWindow` component.

In this component, we want to load the currently authenticated user data and present it in a window.

First make the imports:

06-graphql/step-1/src/WelcomeWindow.tsx

```
import React from "react"
import { gql } from "apollo-boost"
import { useQuery } from "react-apollo-hooks"
```

Then define a constant for the user info query:

06-graphql/step-1/src/WelcomeWindow.tsx

```
const GET_USER_INFO = gql`
  query getUserInfo {
    viewer {
      name
      bio
    }
  }
`
```

If you go to GitHub API documentation[245] - you'll see that this query returns an object with the field `viewer` that contains user data. We'll use the fields `name` and `bio`. Let's define a type for this query:

06-graphql/step-1/src/WelcomeWindow.tsx

```
type UserInfoData = {
  viewer: {
    name: string
    bio: string
  }
}
```

Now define the component:

245

06-graphql/step-1/src/WelcomeWindow.tsx

```
export const WelcomeWindow = () => {
  const { loading, data } = useQuery<UserInfoData>(GET_USER_INFO, {
    notifyOnNetworkStatusChange: true,
    pollInterval: 0,
    fetchPolicy: "no-cache"
  })

  if (loading) {
    return null
  }

  return 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` we get three variables:

- isLoading - is a boolean flag that shows if we are still waiting for the server response
- data - is our data. You can provide the type argument to `useQuery` hook to specify the type of it.
- error - if something goes wrong this object will contain the information about the error

We show the loader while the `isLoading` flag is true and show the values from the `data` object after it's loaded.

For now, we just render the parsed JSON of the data that we got from the GitHub API.

Open `src/App.tsx` and render the `WelcomeWindow` component:

06-graphql/step-1/src/App.tsx

```
import React from "react"
import { WelcomeWindow } from "./WelcomeWindow"

export const App = () => {
  return (
    <blessed-box
      style={{
        bg: "#0000ff"
      }}
    >
        <WelcomeWindow />
    </blessed-box>
  )
}
```

Try to launch the app and make sure that you get the data:

```
yarn start
```

You should see something like this:

Getting user data from GitHub

If everything is ok, we can add a proper layout.

Add The Panel Component

If you've launched the app from the example folder, you saw that we render a window or a panel on each screen. Let's define a component for it.

Create a new file `src/Panel.tsx` and make the imports:

06-graphql/step-1/src/Panel.tsx

```
import React, { PropsWithChildren, FC } from "react"
import { forwardRef } from "react"
```

Then define the type for the component props:

06-graphql/step-1/src/Panel.tsx

```
type PanelProps = {
  top?: number | string
  left?: number | string
  right?: number | string
  bottom?: number | string
  width?: number | string
  height?: number | string
}
```

And finally we define the layout:

06-graphql/step-1/src/Panel.tsx

```
export const Panel = forwardRef<any, PropsWithChildren<PanelProps>>(
  ({ children, ...rest }, ref) => {
    return (
      <blessed-box
        ref={ref}
        draggable
        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>
    )
  }
)
```

Define The WelcomeWindow Layout

Go back to `src/WelcomeWindow.tsx` and add the following lines to the layout:

06-graphql/step-1/src/WelcomeWindow.tsx

```
  return (
    <Panel height={10} top="25%" left="center">
      <blessed-text
        left="center"
        bg="white"
        fg="black"
        content="Welcome to Github Manager"
      />
      <blessed-text
        top={3}
        bg="white"
        fg="black"
        content={`Name: ${data?.viewer.name}`}
      />
      <blessed-text
        top={5}
        bg="white"
        fg="black"
        content={`Bio: ${data?.viewer.bio}`}
      />

    </Panel>
  )
```

Now if you launch the app again you should see this:

Main screen

Getting GitHub GraphQL Schema

Ok, we just wrote our first query. The problem was that we had to provide the types for it manually.

The type information is already contained in the GraphQL schema - to be able to use it with typescript you just need to extract it.

To extract the type information you first need to obtain the full GraphQL schema definition.

To do this run this command in the terminal:

```
yarn run apollo schema:download --header="Authorization: Bearer c554482\
33ba17de366e633fb59a39733dcb3536f" --endpoint=https://api.github.com/gr\
aphql graphql-schema.json
```

Here we pass the following options:

- `header` - provides the authentication token
- `endpoint` - the url providing the schema
- `graphql-schema.json` - the file where you want to store the output

This script will download the schema and save it to a JSON file.

Generating The Types

Now we can calculate the TypeScript types from it.

Apollo provides a special CLI util to get the TypeScript types from the GraphQL schema.

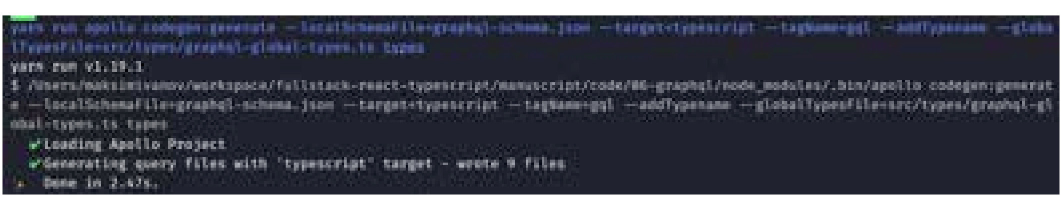
Run it like this:

```
yarn run apollo codegen:generate --localSchemaFile=graphql-schema.json \
--target=typescript --tagName=gql --addTypename --globalTypesFile=src/t\
ypes/graphql-global-types.ts types
```

Here 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 well you should see something like this:



Types generated successfully

Also, you should see that you've got a new folder `src/types`. If you open it you'll see the type definitions for the `getUserInfo` query.

Every time we write new GraphQL queries or mutations, we'll run this code generator to get the types for those queries.

Now let's update our code to use the automatically generated types instead of our custom ones.

Open `src/WelcomeWindow` and import the types:

06-graphql/step-2/src/WelcomeWindow.tsx

```
import { getUserInfo } from "./types/getUserInfo"
```

And change the call to `useQuery` to this:

06-graphql/step-2/src/WelcomeWindow.tsx

```
  const { loading, data } = useQuery<getUserInfo>(GET_USER_INFO, {
    notifyOnNetworkStatusChange: true,
    pollInterval: 0,
    fetchPolicy: "no-cache"
  })
```

Adding Navigation

Right now we have only one window - the one that greets the user and shows profile information.

We need to let the user navigate between different pages. To do this we'll use the `react-router` library.

Go to `src/App.tsx` and add the following imports:

06-graphql/step-2/src/App.tsx

```
import { Switch, Route, useHistory } from "react-router"
```

We'll use `Switch` and `Route` to define the routing and the `useHistory` hook to navigate between the pages.

Call the `useHistory` hook inside the `App` component:

06-graphql/step-2/src/App.tsx

```
  const history = useHistory()
```

Now define the `Switch` with routes inside the `blessed-box` element:

06-graphql/step-2/src/App.tsx

```
      <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've defined the routes for the repositories, issues, and pull request pages.

Create three new folders, one for each resource type. Create an `index.ts` file inside of each folder. The file structure should look like this:

- src
  - /Issues
    - index.ts
  - /Repositories
    - index.ts
  - /PullRequests
    - index.ts

Inside each `index.ts` file define a component matching the folder name.

For example the `src/Issues/index.ts` will look like this:

06-graphql/step-2/src/Issues/index.tsx

```
import React from "react"
import { Panel } from "../Panel"

export const Issues = () => {
  return (
    <Panel height={10} top="25%" left="center">
      <blessed-text
        left="center"
        bg="white"
        fg="black"
        content="Issues" // Use proper text for each page
      />
    </Panel>
  )
}
```

Repeat for the other resources.

After you've done that, import these components in the `src/App.tsx`.

Now we can define the navigation panel. To do this `blessed` has a special component called `blessed-listbar`. It allows you to render a list of options with associated keys. When the user presses the key, it triggers an associated callback.

Add the following code to `src/App.tsx` above the `Switch` element.

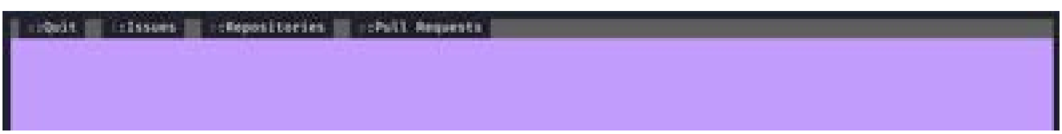
06-graphql/step-2/src/App.tsx

```
      <blessed-listbar
        height={1}
        items={{
          Quit: {
            keys: "q"
          },
          Issues: {
            keys: "i",
            callback: () => history.push("/issues")
```

```
          },
          Repositories: {
            keys: "r",
            callback: () => history.push("/repositories")
          },
          "Pull Requests": {
            keys: "p",
            callback: () => history.push("/pull-requests")
          }
        }}
        style={{
          bg: "grey",
          height: 1
        }}
      />
```

Here we define three callbacks, one for each page. As we are using the `react-router` library we can use the `history` object to perform the navigation programmatically.

Launch the app and make sure you can navigate between the pages.



Navigation Bar

Try pressing the corresponding keys to see if the navigation works.

Working With GitHub Repositories

In our app, the user will be able to list the existing repositories and create new ones.

Open the `src/Repositories/index.tsx` and add the following code:

06-graphql/step-3/src/Repositories/index.tsx

```
import React from "react"
import { Route, Switch, useRouteMatch } from "react-router"

const RepositoriesMain = () => <>Repositories Main</>
const NewRepository = () => <>New Repository</>
const ListRepositories = () => <>List Repositories</>

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={ListRepositories}
      />
    </Switch>
  )
}
```

Here we've defined some nested routes specific to repositories. We have three routes:

- `RepositoriesMain` - this component will show links to two other routes
- `NewRepository` - this component will contain the form to create new repositories
- `ListRepositories` - this will show a scrollable list of existing repos.

Let's start with the main repositories page component. Create a new file `src/Repositories/Repos`

First add the imports:

06-graphql/step-3/src/Repositories/RepositoriesMain.tsx

```
import React from "react"
import { useHistory, useRouteMatch } from "react-router"
import { useRef } from "react"
import { Panel } from "../Panel"
```

Then define the component with the following layout:

06-graphql/step-3/src/Repositories/RepositoriesMain.tsx

```
export const RepositoriesMain = () => {
  const ref = useRef<any>()

  return (
    <Panel ref={ref} height={10} top="25%" left="center">
      <blessed-text
        left="center"
        bg="white"
        fg="black"
        content="Issues"
      />
      <blessed-button
        left="center"
        top={3}
        bg="white"
        fg="black"
        content="l:List Repositories"
      />
      <blessed-button
        left="center"
        top={5}
        bg="white"
        fg="black"
        content="c:Create New Repository"
      />
    </Panel>
```

```
  )
}
```

Here we render the instructions on how to navigate to other pages.

We also get the reference to the panel, so we can have screen-specific event listeners.

Add this code before the layout:

06-graphql/step-3/src/Repositories/RepositoriesMain.tsx

```
  const history = useHistory()
  const match = useRouteMatch()
  const ref = useRef<any>()

  React.useEffect(() => {
    ref.current.key("c", () => history.push(`${match.url}/new`))
    ref.current.key("l", () => history.push(`${match.url}/list`))
  }, [])
```

Import the main component to the `src/repositories/index.tsx` and render it instead of the stub:

06-graphql/step-3/src/Repositories/index.tsx

```
import React from "react"
import { Route, Switch, useRouteMatch } from "react-router"
import { RepositoriesMain } from "./RepositoriesMain"

const NewRepository = () => <>New Repository</>
const ListRepositories = () => <>List Repositories</>

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={ListRepositories}
      />
    </Switch>
  )
}
```

Define The List Component

In the next section, we'll get the list of repositories and we'll need a way to render them.

Let's define the `List` helper component. Create a new file `src/List.tsx`.

Add the following imports:

06-graphql/step-3/src/List.tsx

```
import React, { FC, forwardRef } from "react"
```

Then define the type for the component props:

06-graphql/step-3/src/List.tsx

```
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[]
}
```

Define and export the component:

06-graphql/step-3/src/List.tsx

```
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}
      />
    )
  }
)
```

Here we use the `blessed-list` with a bunch of props predefined.

Getting The Repositories List

Now we can navigate to the repositories list page. Let's define this component. Create a new file `src/repositories/ListRepositories.tsx`.

Add the following imports:

06-graphql/step-3/src/Repositories/ListRepositories.tsx

```
import React, { useRef } from "react"
import { Panel } from "../Panel"
import { useEffect } from "react"
import open from "open"
import { gql } from "apollo-boost"
import { useQuery } from "react-apollo-hooks"
import { List } from "../List"
```

Let's define a query that will fetch the list of available repositories:

06-graphql/step-3/src/Repositories/ListRepositories.tsx

```
const LIST_REPOSITORIES = gql`
  query listRepositories {
    viewer {
      repositories(first: 100) {
        nodes {
          name
          url
        }
      }
    }
  }
```

Now we can run the code generator to get the types for this query:

```
yarn run apollo codegen:generate --localSchemaFile=graphql-schema.json \
--target=typescript --tagName=gql --addTypename --globalTypesFile=src/t\
ypes/graphql-global-types.ts types
```

You should see a new folder: `src/Repositories/types`. Import the generated query type:

06-graphql/step-3/src/Repositories/ListRepositories.tsx

```
import { listRepositories } from "./types/listRepositories"
```

Run the `useQuery` hook with the query that we've just defined:

06-graphql/step-3/src/Repositories/ListRepositories.tsx

```
export const ListRepositories = () => {
  const { loading, error, data } = useQuery<listRepositories>(LIST_REPO\
SITORIES, {
    notifyOnNetworkStatusChange: true,
    pollInterval: 0,
    fetchPolicy: "no-cache"
  })

  return (
    null
  )
}
```

Here we've provided the types that we've generated from the query.

Let's make sure that we get the data correctly. Render the JSON:

06-graphql/step-3/src/Repositories/ListRepositories.tsx

```
export const ListRepositories = () => {
  const { loading, error, data } = useQuery<listRepositories>(LIST_REPO\
SITORIES, {
    notifyOnNetworkStatusChange: true,
    pollInterval: 0,
    fetchPolicy: "no-cache"
  })

  if(loading){
    return <>Loading...</>
  }
```

```
  return (
    JSON.stringify(data)
  )
}
```

You should see something like this:

Loading the data

Now let's define the layout. We're going to use the `blessed-list` component. It will automatically handle the keyboard and mouse navigation.

First define the `listRef` and the `repos` array that we'll get from the `data` object:

06-graphql/step-3/src/Repositories/ListRepositories.tsx

```
  const listRef = useRef<any>()
  const repos = data?.viewer.repositories.nodes
```

Now define the layout:

06-graphql/step-3/src/Repositories/ListRepositories.tsx

```
  return (
    <Panel height={10} top="25%" left="center">
      <blessed-text
        left="center"
        bg="white"
        fg="black"
        content="List Repositories"
      />
      <List
        ref={listRef}
        top={2}
        onAction={(el) =>
          open(
            repos?.find((repo) => repo?.name === el.content)
              ?.url || ""
          )
        }
        items={repos?.map((repo) => repo?.name || "") || []}
      />
    </Panel>
  )
```

We pass the `listRef` to the `List` element here so that we can trigger the focus event on it on mount. Add the following `useEffect` before the layout:

06-graphql/step-3/src/Repositories/ListRepositories.tsx

```
  useEffect(() => {
    listRef.current.focus()
  }, [data])
```

We've also added an `onAction` callback that will open the browser when the user selects the repo on the list.

Open the `src/Repositories/index.tsx` and use the real `ListRepositories` component:

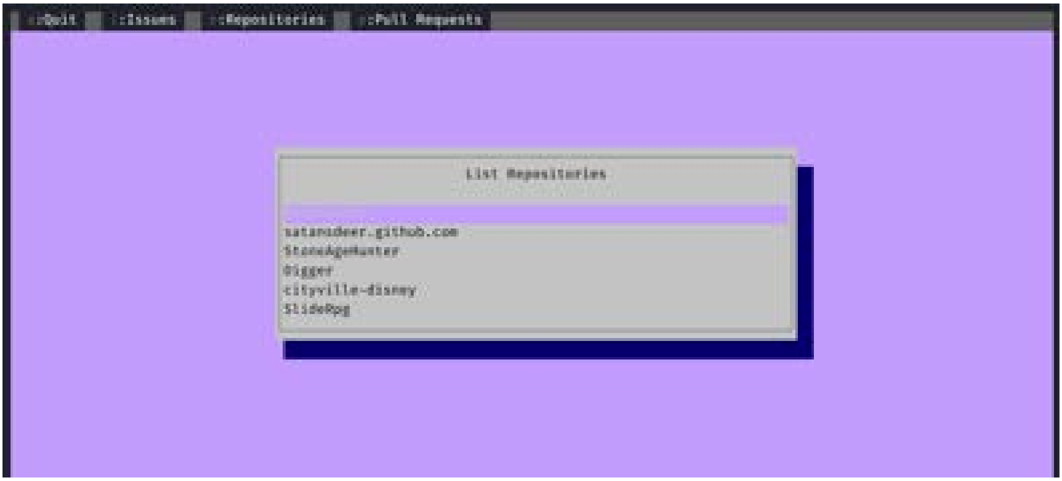
06-graphql/step-3/src/Repositories/index.tsx

```
import React from "react"
import { Route, Switch, useRouteMatch } from "react-router"
import { RepositoriesMain } from "./RepositoriesMain"
import { ListRepositories } from "./ListRepositories"

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={ListRepositories}
      />
    </Switch>
  )
}
```

Run the app and make sure it works.

```
yarn start
```

It should look something like this:



Repositories List

Define Form Helper Components

In the next section, we are going to use our first mutation and we'll collect the user input for it. To do this we'll need to implement the `Form` and `Field` components.

Let's start with the form. Create the new file `src/Form.tsx` and add these imports:

06-graphql/step-4/src/Form.tsx

```
import React, { PropsWithChildren, FC, ReactNode, useRef } from "react"
```

Then let's define the types for our form:

06-graphql/step-4/src/Form.tsx

```
export type FormValues = {
  textbox: string[]
}

type FormProps = {
  onSubmit(values: FormValues): void
  children(triggerSubmit: () => void): ReactNode
}
```

Here we define the `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 the form `onSubmit` automatically when its inputs are submitted, so we have to have this hack here.

Now we can define the `Form` component:

06-graphql/step-4/src/Form.tsx

```
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>
  )
}
```

Here we define the trigger submit function that will call the `submit` method on our form when triggered.

Also, we define the `useEffect` to automatically focus the form when the component is mounted.

Then in the `Form` layout, we render the `children` function passing the `triggerSubmit` to it.

Now let's define the `Field`. Create the new file `src/Field.tsx`. Begin with the imports:

06-graphql/step-4/src/Field.tsx

```
import React from "react"
import { FC } from "react"
import { TextBox } from "./TextBox"
import { forwardRef } from "react"
```

Then define the type for the props:

06-graphql/step-4/src/Field.tsx

```
type FieldProps = {
  label: string
  top?: number | string
  onSubmit(): void
}
```

- `label` - will be shown before the input
- `top` - the offset from the top
- `onSubmit` - input submit handler, triggers on `Enter` keypress

Finally define the `Field` component:

06-graphql/step-4/src/Field.tsx

```
export const Field: FC<FieldProps> = ({label, top, onSubmit}) => {
  return (
    <>
      <blessed-text
        width={label.length}
        content={label}
        style={{
          bg: "white",
          fg: "black"
        }}
        top={top}
      />
      <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 it defined as a reusable component.

GraphQL Mutations - Creating The Repositories

So far we've only been fetching the data. Time to write our first mutation and create some repos.

Create a new file `src/Repositories/NewRepository.tsx`, and add these imports:

06-graphql/step-4/src/Repositories/NewRepository.tsx

```
import { gql } from "apollo-boost"
import React, { useState } from "react"
import { useMutation } from "react-apollo-hooks"
import { Field } from "../Field"
import { Form, FormValues } from "../Form"
import { Panel } from "../Panel"
import { NewRepositorySuccess } from "./NewRepositorySuccess"
```

Then let's define the mutation:

06-graphql/step-4/src/Repositories/NewRepository.tsx

```
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 the types:

```
yarn run apollo codegen:generate --localSchemaFile=graphql-schema.json \
--target=typescript --tagName=gql --addTypename --globalTypesFile=src/t\
ypes/graphql-global-types.ts types
```

Import the generated types:

06-graphql/step-4/src/Repositories/NewRepository.tsx

```
import { createNewRepository_createRepository_repository, createNewRepo\
sitory, createNewRepositoryVariables } from "./types/createNewRepositor\
y"
import { RepositoryVisibility } from "../types/graphql-global-types"
```

Define the component:

06-graphql/step-4/src/Repositories/NewRepository.tsx

```
export const NewRepository = () => {
  const onSubmit = async (values: FormValues) => {
    const [name, description] = values.textbox
  }

  return (
    <Panel top="25%" left="center" height={10}>
      <blessed-text
        left="center"
        bg="white"
        fg="black"
```

```
        content="New repository"
      />
      <Form onSubmit={onSubmit}>
        {(triggerSubmit) => {
          return (
            <>
              <Field
                top={0}
                label="Name: "
                onSubmit={triggerSubmit}
              />
              <Field
                top={1}
                label="Description: "
                onSubmit={triggerSubmit}
              />
            </>
          )
        }}
      </Form>
    </Panel>
  )
}
```

Here we have a form and an `onSubmit` handler that for now just extracts the values from the form.

Use the mutation - add this code to the beginning of the component:

06-graphql/step-4/src/Repositories/NewRepository.tsx

```
  const [createrepository] = useMutation<
    createNewRepository,
    createNewRepositoryVariables
  >(CREATE_REPOSITORY)
```

Here we've used the `useMutation` hook from `react-apollo`.

Now let's call the `createRepositoryMutation` inside the `onSubmit` callback:

06-graphql/step-4/src/Repositories/NewRepository.tsx

```
    const result = await createrepository({
      variables: {
        name,
        description,
        visibility: RepositoryVisibility.PUBLIC
      }
    })
```

Make sure that `onSubmit` is an `async` function.

We've provided the automatically generated types to it so we'll get the correct data in return. Also, we are getting the type-suggestions when we pass the variables to it:

Type suggestions

Now after we get the `result` from the mutation, we want to store it in the state. Define the `repository` state:

06-graphql/step-4/src/Repositories/NewRepository.tsx

```
const [
  repository,
  setRepository
] = useState<createNewRepository_createRepository_repository | null>()
```

Save the `result` from the mutation call using this state:

06-graphql/step-4/src/Repositories/NewRepository.tsx

```
  setRepository(result.data?.createRepository?.repository)
```

The `onSubmit` callback should look like this:

06-graphql/step-4/src/Repositories/NewRepository.tsx

```
const onSubmit = async (values: FormValues) => {
  const [name, description] = values.textbox

  const result = await createrepository({
    variables: {
      name,
      description,
      visibility: RepositoryVisibility.PUBLIC
    }
  })

  setRepository(result.data?.createRepository?.repository)
}
```

Now let's add an early return if we have the `repository` in the state:

06-graphql/step-4/src/Repositories/NewRepository.tsx

```
  if (repository) {
    return <NewRepositorySuccess repository={repository} />
  }
```

Add this code right above the layout. Here we render the success screen.

Create the `src/Repositories/NewRepositorySuccess.tsx` file. Add the imports:

06-graphql/step-4/src/Repositories/NewRepositorySuccess.tsx

```
import open from "open"
import React, { useRef, useEffect, FC } from "react"
import { Panel } from "../Panel"
import { createNewRepository_createRepository_repository } from "./type\
s/createNewRepository"
```

Then add the props types:

06-graphql/step-4/src/Repositories/NewRepositorySuccess.tsx

```
type NewIssueSuccessProps = {
  repository: createNewRepository_createRepository_repository;
}
```

Define the component:

06-graphql/step-4/src/Repositories/NewRepositorySuccess.tsx

```
export const NewRepositorySuccess:FC<NewIssueSuccessProps> = ({reposito\
ry}) => {
  const ref = useRef<any>()

  useEffect(() => {
    ref.current.key("o", () => open(repository.url))
  }, [])
```

```
  return (
    <Panel ref={ref} top="25%" left="center" height={10}>
      <blessed-text
        left="center"
        bg="white"
        fg="black"
        content="Repository Created"
      />

      <blessed-text
        left="center"
        top={3}
        bg="white"
        fg="black"
        content="o: Open Repository in Browser"
      />
    </Panel>
  )
}
```

Here we add a keypress listener inside the `useEffect`. When the user presses the letter `o`, we'll open the `repository.url` in the browser.

Go back to `src/Repositories/NewRepository.tsx`.

Let's add the navigation instructions to our form view. Add this inside the `Panel` children, right after the `Form` element:

06-graphql/step-4/src/Repositories/NewRepository.tsx

```
      <blessed-text
        left="center"
        bg="white"
        fg="black"
        bottom={1}
        content="Tab: Next Field"
      />
      <blessed-text
        left="center"
        bg="white"
        fg="black"
        bottom={0}
        content="Enter: Submit"
      />
```

Go to `src/index.tsx` and import the real `NewRepository` component:

06-graphql/step-4/src/Repositories/index.tsx

```
import React from "react"
import { Route, Switch, useRouteMatch } from "react-router"
import { ListRepositories } from "./ListRepositories"
import { NewRepository } from "./NewRepository"
import { RepositoriesMain } from "./RepositoriesMain"

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={ListRepositories} />
    </Switch>
  )
}
```

Launch the app to see if it renders correctly:

Create repository form

Try to create a new repository and navigate to it.

Getting The Repository ID

Before we move to other resources we'll need to create a shared query that will allow us to get the repository id by its name.

Create a new file `src/queries/getRepository.ts` with the following code:

06-graphql/step-5/src/queries/getRepository.ts

```
import { gql } from "apollo-boost";

export const GET_REPOSITORY = gql`
  query getRepository($owner: String!, $name: String!) {
    repository(owner: $owner, name: $name) {
      id
    }
  }
`
```

Run the code generator to get the types for it.

```
yarn run apollo codegen:generate --localSchemaFile=graphql-schema.json \
--target=typescript --tagName=gql --addTypename --globalTypesFile=src/t\
ypes/graphql-global-types.ts types
```

Make sure that you've got the `src/queries/types` folder with the types for this query.

Working With GitHub Issues

Now we can start working on GitHub Issues. Issues are basically discussions bound to specific repos.

Open the `src/Issues/index.tsx` and add this navigation code:

06-graphql/step-6/src/Issues/index.tsx

```
import React from "react";
import { Route, Switch, useRouteMatch } from "react-router";

const IssuesMain = () => <>Main</>
const NewIssue = () => <>New Issue</>
const ListIssues = () => <>List Issues</>

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={ListIssues} />
    </Switch>
  )
}
```

As you can see it has the same structure as the repository's index component.

Define the main issues page component. Create a new file `src/Issues/IssuesMain.tsx`.

First add the imports:

06-graphql/step-6/src/Issues/IssuesMain.tsx

```
import React from "react"
import { useHistory, useRouteMatch } from "react-router"
import { useRef } from "react"
import { Panel } from "../Panel"
```

Then define the component with the following layout:

06-graphql/step-6/src/Issues/IssuesMain.tsx

```
export const IssuesMain = () => {
  const ref = useRef<any>()

  return (
    <Panel ref={ref} height={10} top="25%" left="center">
      <blessed-text
        left="center"
        bg="white"
        fg="black"
        content="Issues"
      />
      <blessed-button
        left="center"
        top={3}
        bg="white"
        fg="black"
        content="l:List Issues"
      />
      <blessed-button
        left="center"
        top={5}
        bg="white"
        fg="black"
        content="c:Create New Issue"
      />
    </Panel>
  )
}
```

Here we render the instructions on how to navigate to other pages.

We also get the reference to the panel, so we can have screen-specific event listeners.

Add this code before the layout:

06-graphql/step-6/src/Issues/IssuesMain.tsx

```
  const history = useHistory()
  const match = useRouteMatch()
  const ref = useRef<any>()

  React.useEffect(() => {
    ref.current.key("c", () => history.push(`${match.url}/new`))
    ref.current.key("l", () => history.push(`${match.url}/list`))
  }, [])
```

Import the main component to the `src/Issues/index.tsx` and render it instead of the stub:

06-graphql/step-6/src/Issues/index.tsx

```
import React from "react";
import { Route, Switch, useRouteMatch } from "react-router";
import { IssuesMain } from "./IssuesMain"

const NewIssue = () => <>New Issue</>
const ListIssues = () => <>List Issues</>

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={ListIssues} />
    </Switch>
  )
}
```

Getting The List Of Issues

Create a new component called `src/Issues/ListIssues.tsx`.

Begin by defining the imports:

06-graphql/step-6/src/Issues/ListIssues.tsx

```
import React, { useRef } from "react"
import { Panel } from "../Panel"
import { useEffect } from "react"
import open from "open"
import { gql } from "apollo-boost"
import { useQuery } from "react-apollo-hooks"
import { List } from "../List"
```

Now let's define the query:

06-graphql/step-6/src/Issues/ListIssues.tsx

```
const LIST_ISSUES = gql`
  query listIssues {
    viewer {
      issues(first: 100) {
        nodes {
          title
          url
        }
      }
    }
  }
`
```

And then run the code generator to get the types:

```
yarn run apollo codegen:generate --localSchemaFile=graphql-schema.json \
--target=typescript --tagName=gql --addTypename --globalTypesFile=src/t\
ypes/graphql-global-types.ts types
```

After you've got the types, add this to imports:

06-graphql/step-6/src/Issues/ListIssues.tsx

```
import { listIssues } from "./types/listIssues"
```

Now define the component:

06-graphql/step-6/src/Issues/ListIssues.tsx

```
export const ListIssues = () => {
  const { loading, error, data } = useQuery<listIssues>(LIST_ISSUES, {
    notifyOnNetworkStatusChange: true,
    pollInterval: 0,
    fetchPolicy: "no-cache"
  })
  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
        top={2}
        onAction={(el) =>
          open(
            issues?.find((issue) => issue?.title === el.content)
              ?.url || ""
          )
```

```
        }
        items={issues?.map((issue) => issue?.title || "") || []}
      />
    </Panel>
  )
}
```

Here we've called `useQuery` to get the data, just like we did to get the repositories list. Then we passed the `issues` array to the `List` component.

One last thing - define the `listRef` and the `useEffect`:

06-graphql/step-6/src/Issues/ListIssues.tsx

```
  const listRef = useRef<any>()
  useEffect(() => {
    listRef.current.focus()
  }, [data])
```

Pass the `listRef` as a `ref` to the `List`:

06-graphql/step-6/src/Issues/ListIssues.tsx

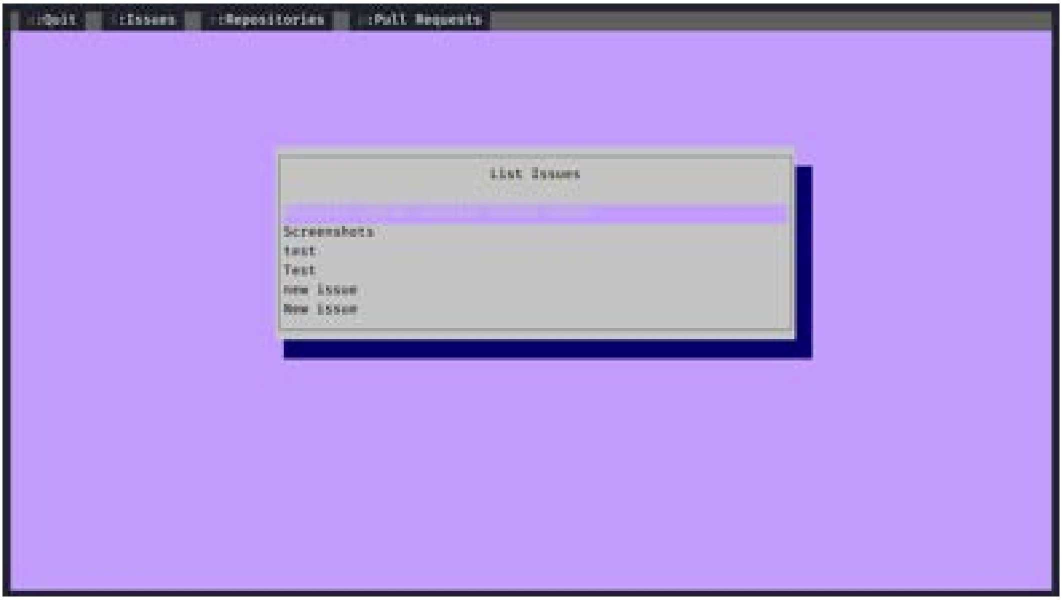
```
      <List
        ref={listRef}
        top={2}
        onAction={(el) =>
          open(
            issues?.find((issue) => issue?.title === el.content)
              ?.url || ""
          )
        }
        items={issues?.map((issue) => issue?.title || "") || []}
      />
```

Now go to `src/Issues/index.tsx` and make sure you use the real `ListIssues` component.

06-graphql/step-6/src/Issues/index.tsx

```
import React from "react";
import { Route, Switch, useRouteMatch } from "react-router";
import { IssuesMain } from "./IssuesMain"
import { ListIssues } from "./ListIssues"

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={ListIssues} />
    </Switch>
  )
}
```

Launch the app and make sure you can get the issues list.



List Issues screen

You should also be able to open the selected issue in the browser.

Creating An Issue

Create a new file `src/Issues/NewIssue.tsx`. Add the imports:

06-graphql/step-6/src/Issues/NewIssue.tsx

```
import { gql } from "apollo-boost"
import React, { useState } from "react"
import { useApolloClient, useMutation } from "react-apollo-hooks"
import { Field } from "../Field"
import { Form, FormValues } from "../Form"
import { Panel } from "../Panel"
import { NewIssueSuccess } from "./NewIssueSuccess"
import { GET_REPOSITORY } from "../queries/getRepository"
```

Now let's add the query:

06-graphql/step-6/src/Issues/NewIssue.tsx

```
const CREATE_ISSUE = gql`
  mutation createNewIssue(
    $title: String
    $body: String
    $repository: ID
  ) {
    createIssue(
      input: { title: $title, body: $body, repositoryId: $repository }
    ) {
      issue {
        title
        url
      }
    }
  }
`
```

Generate the types:

```
yarn run apollo codegen:generate --localSchemaFile=graphql-schema.json \
--target=typescript --tagName=gql --addTypename --globalTypesFile=src/t\
ypes/graphql-global-types.ts types
```

Import the generated types:

06-graphql/step-6/src/Issues/NewIssue.tsx

```
import { getRepository, getRepositoryVariables } from "../queries/types\
/getRepository"
import {
  createNewIssue,
  createNewIssueVariables,
  createNewIssue_createIssue_issue
} from "./types/createNewIssue"
```

Now define the component:

06-graphql/step-6/src/Issues/NewIssue.tsx

```
export const NewIssue = () => {
  const onSubmit = async (values: FormValues) => {
    const [repo, title, body] = values.textbox
    const [owner, name] = repo.split("/")
  }

  return (
    <Panel top="25%" left="center" height={10}>
      <blessed-text
        left="center"
        bg="white"
        fg="black"
        content="New Issue"
      />
    </Panel>
  )
}
```

Here we prepared an `onSubmit` handler that will get the input values from the form. Now below the `blessed-text` element define the form:

06-graphql/step-6/src/Issues/NewIssue.tsx

```
      <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>
```

Here we have three inputs:

- Repository name - we need this value to get the repository id. The repository id is a mandatory field when you want to create a new issue
- Issue title - this is also a mandatory field
- Issue description - this is an optional field that you can use to provide some additional information about the issue

Define the `createIssue` mutation and get the reference to the apollo client using the `useApolloClient` hook.

06-graphql/step-6/src/Issues/NewIssue.tsx

```
const [createIssue] = useMutation<
  createNewIssue,
  createNewIssueVariables
>(CREATE_ISSUE)

const client = useApolloClient()
```

We'll use the client to perform the queries directly.

Add the following code to the `onSubmit` handler:

06-graphql/step-6/src/Issues/NewIssue.tsx

```
  const { data } = await client.query<
    getRepository,
    getRepositoryVariables
  >({
    query: GET_REPOSITORY,
    variables: {
      owner,
      name
    }
  })

  if (!data || !data.repository) {
    return
  }
```

Here we manually perform a query to get the repository ID by its name.

If we don't get the repository in the response we just return from the callback.

Now we want to perform the mutation. Add this code after the `if` block inside the `onSubmit` handler:

06-graphql/step-6/src/Issues/NewIssue.tsx

```
    const result = await createIssue({
      variables: {
        title,
        body,
        repository: data.repository.id
      }
    })

    setIssue(result.data?.createIssue?.issue)
```

We call the mutation and then store the result in the state.

Define the `issue` state at the beginning of the component:

06-graphql/step-6/src/Issues/NewIssue.tsx

```
  const [
    issue,
    setIssue
  ] = useState<createNewIssue_createIssue_issue | null>()
```

Now we want to check if we have the `issue` in the state and render the success screen. Add the following code right before the layout:

06-graphql/step-6/src/Issues/NewIssue.tsx

```
  if (issue) {
    return <NewIssueSuccess issue={issue} />
  }
```

Create a new file `src/Issues/NewIssueSuccess.tsx`. Add the following imports:

06-graphql/step-6/src/Issues/NewIssueSuccess.tsx

```
import open from "open"
import React, { useRef, useEffect, FC } from "react"
import { Panel } from "../Panel"
import { createNewIssue_createIssue_issue } from "./types/createNewIssu\
e"
```

Then define the type for the props:

06-graphql/step-6/src/Issues/NewIssueSuccess.tsx

```
type NewIssueSuccessProps = {
  issue: createNewIssue_createIssue_issue;
}
```

Define and export the `NewIssueSuccess` component:

06-graphql/step-6/src/Issues/NewIssueSuccess.tsx

```
export const NewIssueSuccess:FC<NewIssueSuccessProps> = ({issue}) => {
  const ref = useRef<any>()

  useEffect(() => {
    ref.current.key("o", () => open(issue.url))
  }, [])

  return (
    <Panel ref={ref} top="25%" left="center" height={10}>
      <blessed-text
        left="center"
        bg="white"
        fg="black"
        content="Issue Created"
      />

      <blessed-text
        left="center"
```

```
        top={3}
        bg="white"
        fg="black"
        content="o: Open Issue in Browser"
      />
    </Panel>
  )
}
```

Import the `NewIssueSuccess` inside the `src/Issues/NewIssue.tsx`.

Then go to `src/Issues/index.tsx` and make sure you are using the real `NewIssue` component:

06-graphql/step-6/src/Issues/index.tsx

```
import React from "react";
import { Route, Switch, useRouteMatch } from "react-router";
import { IssuesMain } from "./IssuesMain";
import { NewIssue } from "./NewIssue";
import { ListIssues } from "./ListIssues";

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={ListIssues} />
    </Switch>
  )
}
```

Now launch the app and make sure everything works:

New Issue screen

Working With Github Pull Requests

The pull requests are very similar to issues as they also are bound to specific repositories.

Begin by adding the navigation code to the `src/PullRequests/index.tsx`:

06-graphql/step-7/src/PullRequests/index.tsx

```
import React from "react";
import { Route, Switch, useRouteMatch } from "react-router";

const PullRequestsMain = () => <>Main</>
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>
  )
}
```

Define the main pull requests page component. Create a new file `src/PullRequests/PullRequest`

First add the imports:

06-graphql/step-7/src/PullRequests/PullRequestsMain.tsx

```
import React from "react"
import { useHistory, useRouteMatch } from "react-router"
import { useRef } from "react"
import { Panel } from "../Panel"
```

Then define the component with the following layout:

06-graphql/step-7/src/PullRequests/PullRequestsMain.tsx

```
export const PullRequestsMain = () => {
  const ref = useRef<any>()

  return (
    <Panel ref={ref} height={10} top="25%" left="center">
      <blessed-text
        left="center"
        bg="white"
        fg="black"
        content="Pull Requests"
      />
      <blessed-button
        left="center"
        top={3}
```

```
      bg="white"
      fg="black"
      content="l:List Pull Requests"
    />
    <blessed-button
      left="center"
      top={5}
      bg="white"
      fg="black"
      content="c:Create New Pull Request"
    />
  </Panel>
 )
}
```

Here we render the instructions on how to navigate to other pages.

We also get the reference to the panel, so we can have screen-specific event listeners.

Add this code before the layout:

06-graphql/step-7/src/PullRequests/PullRequestsMain.tsx

```
  const history = useHistory()
  const match = useRouteMatch()
  const ref = useRef<any>()

  React.useEffect(() => {
    ref.current.key("c", () => history.push(`${match.url}/new`))
    ref.current.key("l", () => history.push(`${match.url}/list`))
  }, [])
```

Import the main component to the `src/PullRequests/index.tsx` and render it instead of the stub:

06-graphql/step-7/src/PullRequests/index.tsx

```
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>
  )
}
```

Now let's get the list of pull requests.

Getting The Pull Requests List

Create a new component `src/PullRequests/ListPullRequests.tsx` with the following imports:

06-graphql/step-7/src/PullRequests/ListPullRequests.tsx

```
import React, { useRef } from "react"
import { Panel } from "../Panel"
import { useEffect } from "react"
import open from "open"
import { gql } from "apollo-boost"
import { useQuery } from "react-apollo-hooks"
import { List } from "../List"
```

Then we define the query:

06-graphql/step-7/src/PullRequests/ListPullRequests.tsx

```
const LIST_PULL_REQUESTS = gql`
  query listPullRequests {
    viewer {
      pullRequests(first: 100) {
        nodes {
          title
          url
        }
      }
    }
  }
`
```

Run the code generator to get the types:

```
yarn run apollo codegen:generate --localSchemaFile=graphql-schema.json \
--target=typescript --tagName=gql --addTypename --globalTypesFile=src/t\
ypes/graphql-global-types.ts types
```

Now define the component:

06-graphql/step-7/src/PullRequests/ListPullRequests.tsx

```
export const ListPullRequests = () => {
  const pullRequests = []
  const listRef = useRef<any>()
  useEffect(() => {
    listRef.current.focus()
  }, [data])

  return (
    <Panel height={10} top="25%" left="center">
      <blessed-text
        left="center"
        bg="white"
        fg="black"
        content="List Pull Requests"
      />
      <List
        ref={listRef}
        top={2}
        onAction={(el) =>
          open(
            pullRequests?.find((pullRequest) => pullRequest?.title === \
el.content)
              ?.url || ""
          )
        }
        items={pullRequests?.map((pullRequest) => pullRequest?.title ||\
 "") || []}
      />
    </Panel>
  )
}
```

For now, we'll hardcode the `pullRequests` as an empty array.

We've created a `listRef` and passed it to the `List` element to be able to trigger the `focus` method in the `useEffect` on component mount.

Now let's use the query. Add this to the beginning of the component:

06-graphql/step-7/src/PullRequests/ListPullRequests.tsx

```
  const { loading, error, data } = useQuery<listPullRequests>(LIST_PULL\
_REQUESTS, {
    notifyOnNetworkStatusChange: true,
    pollInterval: 0,
    fetchPolicy: "no-cache"
  })
```

Now let's get the `pullRequests` from the data. Change the hardcoded value to this:

06-graphql/step-7/src/PullRequests/ListPullRequests.tsx

```
  const pullRequests = data?.viewer.pullRequests.nodes
```

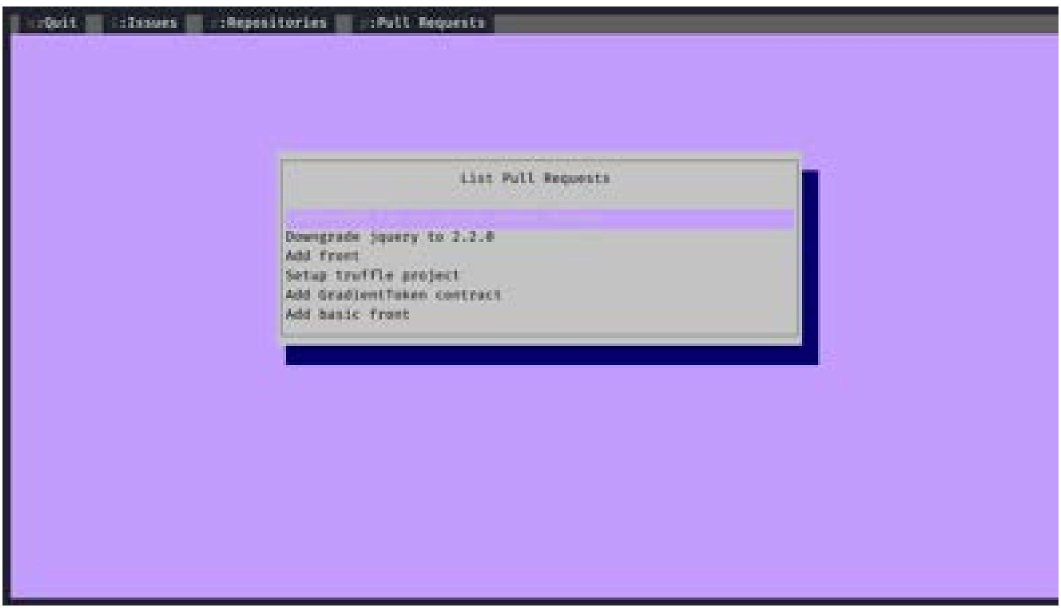
Now go to `src/PullRequests/index.tsx` and import the `ListPullRequests` component:

06-graphql/step-7/src/PullRequests/index.tsx

```
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 app again and verify that you can see the list of pull requests and that it opens the selected pull request in the browser.



List of pull requests

Creating A New Pull Request

Create a new file `src/PullRequests/NewPullRequest.tsx` with the following imports:

06-graphql/step-7/src/PullRequests/NewPullRequest.tsx

```
import { gql } from "apollo-boost"
import React, { useState } from "react"
import { useApolloClient, useMutation } from "react-apollo-hooks"
import { Field } from "../Field"
import { Form, FormValues } from "../Form"
import { Panel } from "../Panel"
import { getRepository, getRepositoryVariables } from "../queries/types\
/getRepository"
import { GET_REPOSITORY } from "../queries/getRepository"
```

Next we define the GraphQL query to create the pull request:

06-graphql/step-7/src/PullRequests/NewPullRequest.tsx

```
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 codegen to generate the types:

```
yarn run apollo codegen:generate --localSchemaFile=graphql-schema.json \
--target=typescript --tagName=gql --addTypename --globalTypesFile=src/t\
ypes/graphql-global-types.ts types
```

After that's done, import the generated types:

06-graphql/step-7/src/PullRequests/NewPullRequest.tsx

```
import {
  createNewPullRequest,
  createNewPullRequestVariables,
  createNewPullRequest_createPullRequest_pullRequest
} from "./types/createNewPullRequest"
```

Then define the component:

06-graphql/step-7/src/PullRequests/NewPullRequest.tsx

```
export const NewPullRequest = () => {
  const onSubmit = async (values: FormValues) => {
    const [repo, title, body, baseRefName, headRefName] = values.textbox
    const [owner, name] = repo.split("/")
  }

  return (
    <Panel top="25%" left="center" height={12}>
      <blessed-text
        left="center"
        bg="white"
        fg="black"
        content="New Pull Request"
```

```
      />
      // We'll add the Form here
    </Panel>
  )
}
```

Define the form layout inside the `Panel`:

06-graphql/step-7/src/PullRequests/NewPullRequest.tsx

```
      <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>
```

Define the mutation and get the `client` instance. Add this code to the beginning of the component:

06-graphql/step-7/src/PullRequests/NewPullRequest.tsx

```
import { gql } from "apollo-boost"
import React, { useState } from "react"
import { useApolloClient, useMutation } from "react-apollo-hooks"
import { Field } from "../Field"
import { Form, FormValues } from "../Form"
import { Panel } from "../Panel"
import { getRepository, getRepositoryVariables } from "../queries/types\
/getRepository"
import { GET_REPOSITORY } from "../queries/getRepository"
import { NewPullRequestSuccess } from "./NewPullRequestSuccess"
import {
  createNewPullRequest,
  createNewPullRequestVariables,
  createNewPullRequest_createPullRequest_pullRequest
} from "./types/createNewPullRequest"

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
      }
    }
  }
`

export const NewPullRequest = () => {
  const [
    pullRequest,
    setPullRequest
  ] = useState<createNewPullRequest_createPullRequest_pullRequest | nul\
l>()
  const [createPullRequest] = useMutation<
    createNewPullRequest,
    createNewPullRequestVariables
  >(CREATE_PULL_REQUEST)

  const client = useApolloClient()

  const onSubmit = async (values: FormValues) => {
    const [repo, title, body, baseRefName, headRefName] = values.textbox
    const [owner, name] = repo.split("/")

    const { data } = await client.query<
      getRepository,
      getRepositoryVariables
```

```
    >({
      query: GET_REPOSITORY,
      variables: {
        owner,
        name
      }
    })

    if (!data || !data.repository) {
      return
    }

    const result = await createPullRequest({
      variables: {
        title,
        body,
        repositoryId: data.repository.id,
        baseRefName,
        headRefName
      }
    })

    setPullRequest(result.data?.createPullRequest?.pullRequest)
  }

  if (pullRequest) {
    return <NewPullRequestSuccess pullRequest={pullRequest} />
  }

  return (
    <Panel top="25%" left="center" height={12}>
      <blessed-text
        left="center"
        bg="white"
        fg="black"
        content="New Pull Request"
```

```
    />
    <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>
    <blessed-text
      left="center"
```

```
        bg="white"
        fg="black"
        bottom={1}
        content="Tab: Next Field"
      />
      <blessed-text
        left="center"
        bg="white"
        fg="black"
        bottom={0}
        content="Enter: Submit"
      />
    </Panel>
  )
}
```

Now we can get the repository ID in the `onSubmit` handler:

06-graphql/step-7/src/PullRequests/NewPullRequest.tsx

```
    const { data } = await client.query<
      getRepository,
      getRepositoryVariables
    >({
      query: GET_REPOSITORY,
      variables: {
        owner,
        name
      }
    })

    if (!data || !data.repository) {
      return
    }
```

Here we get the repository ID and if we fail we just return from the handler.

Next we can call the mutation:

06-graphql/step-7/src/PullRequests/NewPullRequest.tsx

```
    const result = await createPullRequest({
      variables: {
        title,
        body,
        repositoryId: data.repository.id,
        baseRefName,
        headRefName
      }
    })

    setPullRequest(result.data?.createPullRequest?.pullRequest)
```

Here we run the mutation and then save the result in the component state.

Define the state for created pull request:

06-graphql/step-7/src/PullRequests/NewPullRequest.tsx

```
  const [
    pullRequest,
    setPullRequest
  ] = useState<createNewPullRequest_createPullRequest_pullRequest | nul\
l>()
```

After we create the repo we update the state to show the success screen. Add this code right before the layout:

06-graphql/step-7/src/PullRequests/NewPullRequest.tsx

```
  if (pullRequest) {
    return <NewPullRequestSuccess pullRequest={pullRequest} />
  }
```

Let's define the `NewPullRequestSuccess` component. Create a new file `src/PullRequests/NewPul`

Add the imports:

06-graphql/step-7/src/PullRequests/NewPullRequestSuccess.tsx

```
import open from "open"
import React, { FC, useEffect, useRef } from "react"
import { Panel } from "../Panel"
import { createNewPullRequest_createPullRequest_pullRequest } from "./t\
ypes/createNewPullRequest"
```

Define the type for the component props:

06-graphql/step-7/src/PullRequests/NewPullRequestSuccess.tsx

```
type NewIssueSuccessProps = {
  pullRequest: createNewPullRequest_createPullRequest_pullRequest;
}
```

Define the component:

06-graphql/step-7/src/PullRequests/NewPullRequestSuccess.tsx

```
export const NewPullRequestSuccess:FC<NewIssueSuccessProps> = ({pullReq\
uest}) => {
  const ref = useRef<any>()

  useEffect(() => {
    ref.current.key("o", () => open(pullRequest.url))
  }, [])

  return (
    <Panel ref={ref} top="25%" left="center" height={10}>
      <blessed-text
        left="center"
        bg="white"
        fg="black"
        content="Pull Request Created"
      />
```

```
      <blessed-text
        left="center"
        top={3}
        bg="white"
        fg="black"
        content="o: Open Pull Request in the Browser"
      />
    </Panel>
  )
}
```

Go back to `src/PullRequests/NewPullRequest.tsx` and import the `NewPullRequestSuccess` component:

06-graphql/step-7/src/PullRequests/NewPullRequest.tsx

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

Then open the `src/PullRequests/index.tsx` and use the real `NewPullRequest` component.

06-graphql/step-7/src/PullRequests/index.tsx

```
import React from "react";
import { Route, Switch, useRouteMatch } from "react-router";
import { ListPullRequests } from "./ListPullRequests";
import { NewPullRequest } from "./NewPullRequest";
import { PullRequestsMain } from "./PullRequestsMain";

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 app and make sure you can create pull requests.

Creating a Pull Request

Conclusion

In this chapter, we've learned to work with GraphQL and TypeScript combined. It is a great duo as GraphQL allows us to preserve the type-information while communicating with the backend.

One of the great advantages of using GraphQL on your backend is that you can provide the full schema definition to your clients, just like GitHub does it.

Another great benefit of using GraphQL is that you can generate the types from the GraphQL schema. It makes using queries and mutations super easy, as you now have the autocomplete based on the actual schema information.

Hope you liked working on this fun project and good luck in your next endeavors!

Appendix

Changelog

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