FULLSTACK REACT WITH TYPESCRIPT

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

MAKSIM IVANOV
ALEX BESPOYASOV

Fullstack React with TypeScript

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

Written by Maksim Ivanov and Alex Bespoyasov
Edited by Nate Murray

Published by \newline

Contents

Book Revision

Revision 1p - 2020-05-19

EARLY DRAFT VERSION

This version of the book is an early draft. Our expectation is that the code works, but some of the manuscript has not been through final edits.

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 setup 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***: There 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 it's 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***: 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*** Building Medium with SSG. React can be rendered server-side. It allows 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 NextJS framework. The example application will be blogging platform (like Medium).

State Management With Redux and Typescript. (coming soon – Summer 2020) Some React applications are so complex that they require using 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. (coming soon – Summer 2020) GraphQL is a query language that allows to create flexible APIs. Facebook, Github, Twitter and a lot of other companies provide GraphQL APIs. Typescript works pretty well with GraphQL. In this section we will build a Github issue viewer.

We recommend you to read the book in linear order, from start to finish. The sections are arranged from basic topics to more complex. 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 assumed that you have at least the following skills:

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

Here we mostly focus on 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

In the beginning of each section you will find instructions of 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`, it should output your current NodeJS version:

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

Here are instructions for installing NodeJS on different systems:

Windows

To work with 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 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 the 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` word in the end:

```
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 same name but without suffix to see a completed version of it.

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

Reporting Issues

We did 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:

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

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

Ideally also provide a link to a git repository where we can reproduce an 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 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 - Typescript typechecker will give us an error.

Typescript typechecker can tell if we have an error in our code just by analysing 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 will 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 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 name.

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

How can we make sure that this function recieves the input of 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.

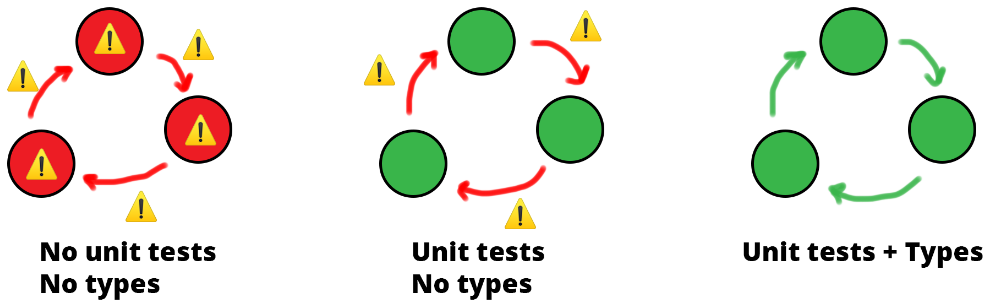
```
greet({firstName: "Maksim", lastName: "Ivanov"}) // Returns "Hello Maks\
im Ivanov!"
```

If we'll 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 the parts of our program, so we can be sure that they interact correctly. It means they will have clear contracts of communicating with each other which will significantly reduce the amount of bugs.



Typescript contracts by which parts of your programm 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% less bugs in your code. There is also an interesting paper about unit tests[15] stating that products where TDD was applied had bettween 40% and 90% decrease in pre-relese 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

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

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 multipler simpler components.

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

This is where Typescript becomes very handy. You can use it to define interfaces of your components, so that you can be sure that your component gets only correct inputs.

If you worked with React before you probably know that you can specify 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 could do with `prop-types` - why would you need Typescript?

For 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 just 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 powerfull. 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 you examples of it 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 a 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/GYVwdgxgLglg9mABAWwKYGd0FUAOAVAC1QEEAnUgQwE8AKC8gLkTMqoB50pSYwBzRAD6IwIZACNUpAHwBKJgDc4MACaI
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 the data obtained through network requests or loaded from the file system.

During this book we demonstrate the techniques that allow 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 tell you what is 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.

By the end of this chapter, we will have the application layout. In the next chapter, we'll add the drag-and-drop and the business logic to it.

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`, it's 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, it 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 examples, I will use Yarn[22]. You can use npm[23] if you want.

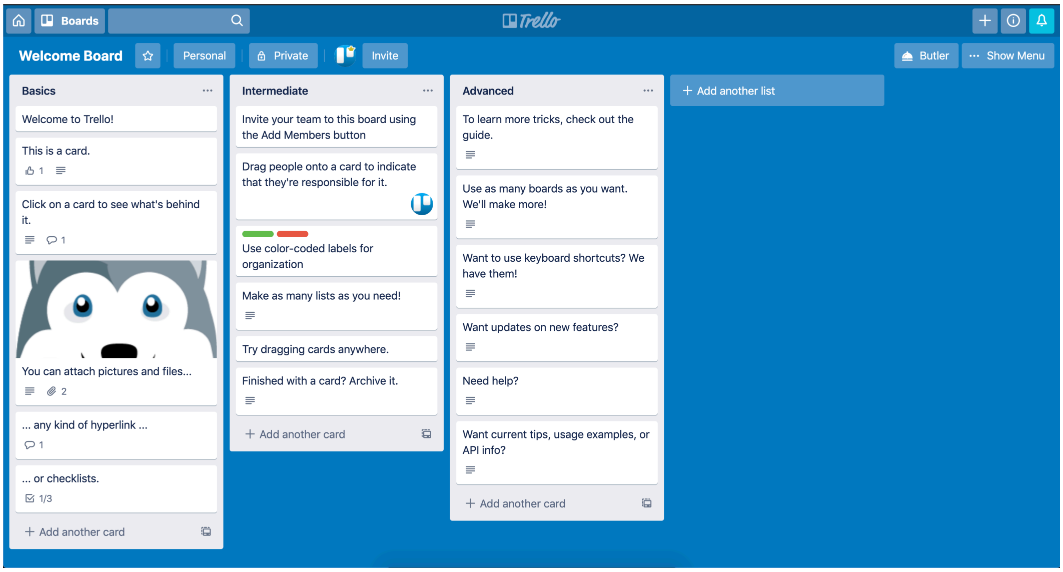
All the examples for this chapter contain `yarn.lock` files, remove them if you want to use npm to install dependencies.

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 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/creationix/nvm#installation
[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



Basics

Welcome to Trello!

This is a card.

Click on a card to see what's behind it.

You can attach pictures and files...

... any kind of hyperlink ...

... or checklists.

Intermediate

Invite your team to this board using the Add Members button

Drag people onto a card to indicate that they're responsible for it.

Use color-coded labels for organization

Make as many lists as you need!

Try dragging cards anywhere.

Finished with a card? Archive it.

Advanced

To learn more tricks, check out the guide.

Use as many boards as you want. We'll make more!

Want to use keyboard shortcuts? We have them!

Want updates on new features?

Need help?

Want current tips, usage examples, or API info?

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`.

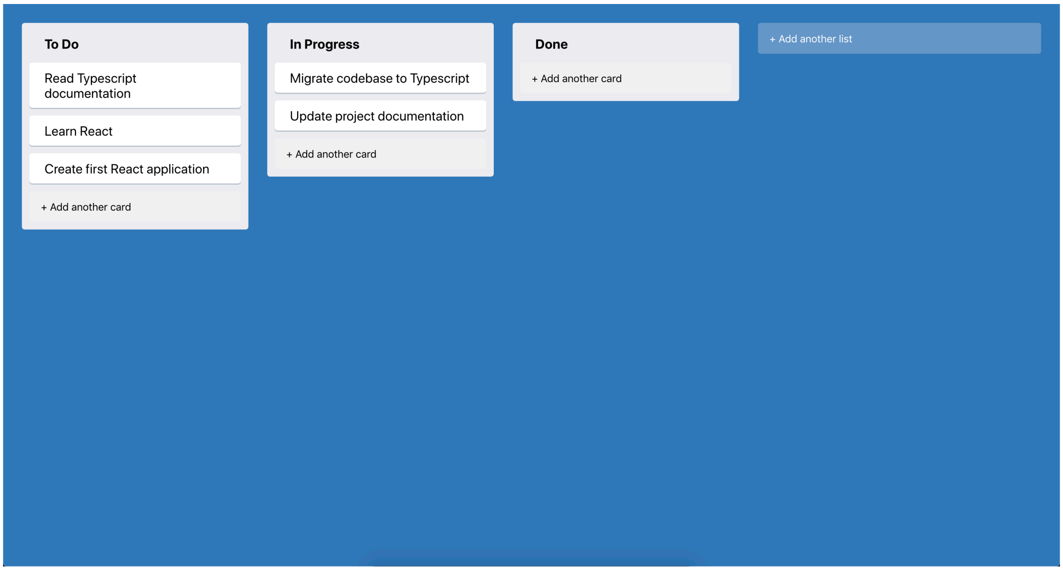
Unzip the archive and `cd` to the app folder.

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

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

```
yarn && yarn start
```

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


To Do
Read Typescript documentation
Learn React
Create first React application
+ Add another card
In Progress
Migrate codebase to Typescript
Update project documentation
+ Add another card
Done
+ Add another card
+ Add another list

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 the column and between them.

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

For now, our app doesn't persist any state and we don't send data to a server, but we'll add these features later on.

Try to create new cards and columns and drag them around.

How to Bootstrap React + Typescript App Automatically?

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

In this chapter, we will use an automatic CLI tool to generate our project 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 is a lot of smaller things. Setting up a test runner, adding vendor prefixes to your CSS rules, setting up linter, 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. 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
│   ├── serviceWorker.ts
│   └── setupTests.ts
├── node_modules
│   └── ...
├── README.md
├── package.json
├── tsconfig.json
└── yarn.lock
```

Let's go through the files and see why do we need them there. We'll make a short overview, and then we'll get 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 npm website[28]

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

01-first-app/step1/package.json

```
  "dependencies": {
    "@testing-library/jest-dom": "^4.2.4",
    "@testing-library/react": "^9.3.2",
    "@testing-library/user-event": "^7.1.2",
    "@types/jest": "^24.0.0",
    "@types/node": "^12.0.0",
    "@types/react": "^16.9.0",
    "@types/react-dom": "^16.9.0",
    "react": "^16.12.0",
    "react-dom": "^16.12.0",
```

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

```
    "react-scripts": "3.3.1",
    "typescript": "~3.7.2"
  },
```

Now, most packages that we use have a corresponding `@types/*` package.

I'm showing only the `dependencies` block because this is where type definitions are installed. You won't find any 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?

Problem with Javascript is that a lot of times it's impossible to tell what types will the code work with. Let's say we have a Javascript code where we have 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 is the `data` attribute 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 Typescript compiler will see it, usually, it's `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 knew 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[29] 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 `^16.12.0` and `@types/react-dom` is `^16.9.0`.

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

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

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

These are all the files that we can 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 `public` folder in Create React App documentation[30].

[29]http://definitelytyped.org/
[30]https://create-react-app.dev/docs/using-the-public-folder/

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

manifest.json. It provides application metadata for Progressive Web Apps[31]. For example, this file allows installing 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[32]

favicon.ico, logo192.png, logo512.png. These are icons for your application. There is `favicon.ico`, it's a favicon, 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. It tells crawlers what resources they shouldn't access. By default it allows everything.

> Read more about `robots.txt` on robotstxt website[33]

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 React Docs[34]

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

[31]https://web.dev/progressive-web-apps/
[32]https://developer.mozilla.org/en-US/docs/Web/Manifest
[33]https://www.robotstxt.org/robotstxt.html
[34]https://reactjs.org/docs/introducing-jsx.html

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

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

Typescript has *type assertion operator* that uses angle brackets:

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

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

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

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

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

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

You can read about it in Typescript Documentation[35].

index.tsx

Most important file in `/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:

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

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

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import * as serviceWorker from './serviceWorker';

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

// If you want your app to work offline and load faster, you can change
// unregister() to register() below. Note this comes with some pitfalls.
// Learn more about service workers: https://bit.ly/CRA-PWA
serviceWorker.unregister();
```

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

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

```
ReactDOM.render(<App />, 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 `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.

Then there is an interesting syntax to import `serviceWorker`:

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

```
import * as serviceWorker from './serviceWorker';
```

Our `serviceWorker.ts` file exports two functions: `register` and `unregister`. It doesn't have the default export so we have to use an asterisk syntax to save both functions to one `serviceWorker` variable.

Alternatively we could import both functions individually, like this: `import { register, unregister } from './serviceWorker'`.

What's important here is that when the module doesn't have a default export - you can use an asterisk to assign all the individual exports to one variable. In the next section, we'll discuss how can it affect you when you use Typescript.

App.tsx

Let's open `src/App.tsx`. If you use modern `create-react-app` this file won't have any difference from the regular Javascript version.

In older versions, React was imported differently.

Instead of:

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

```
import React from 'react';
```

You would see:

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

To explain this I will have to tell 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[36]. React exports an object full of different classes and functions:

[36]https://github.com/facebook/react/blob/master/packages/react/index.js

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

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

But if you've used React with Javascript before - you've noticed that React is always imported there like it has the default export.

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

It's 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*. And second - you most likely use React with some bundler like Webpack, and it's smart enough to check that if no default property is set in the export, 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 def\
ault-imported using the 'allowSyntheticDefaultImports' flag  TS1259

  > 1 | import React from 'react'
      |        ^
    2 | import logo from './logo.svg';
    3 | import './App.css';
    4 |
```

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 Typescript 2.7 release notes[37].

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, it's needed for libraries that were originally written in Javascript.

This file containes the following code:

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 Typescript documentation[38]

By default, it 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 it's `package.json`. So we end up referencing types from `./node_modules/react-scripts/lib/rea`

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?

Thing is that 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:

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

```
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.

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

```
declare module "*.png" {
  const src: string
  export default src
}
```

This declaration tells Typescript that when we import stuff from modules that have names ending with `.png` - we will get the default export of type `string`.

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

And Webpack 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`
- `serviceWorker.ts`.

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

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

Also open `src/index.tsx` and remove all the `serviceWorker` mentions. Your index file should look like this:

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(<App />, document.getElementById('root'));
```

Add Global Styles

We need to have some styles to be applied 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 browser to include elements `padding` and `border` in it's `width` and `height` calculations.

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

How To Style React Elements

There are several ways to style React elements:

- Regular CSS files, including CSS-modules.
- Manually specifying element `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 import `index.css` in `index.tsx` file.

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

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

Passing CSS Rules Through Style Prop

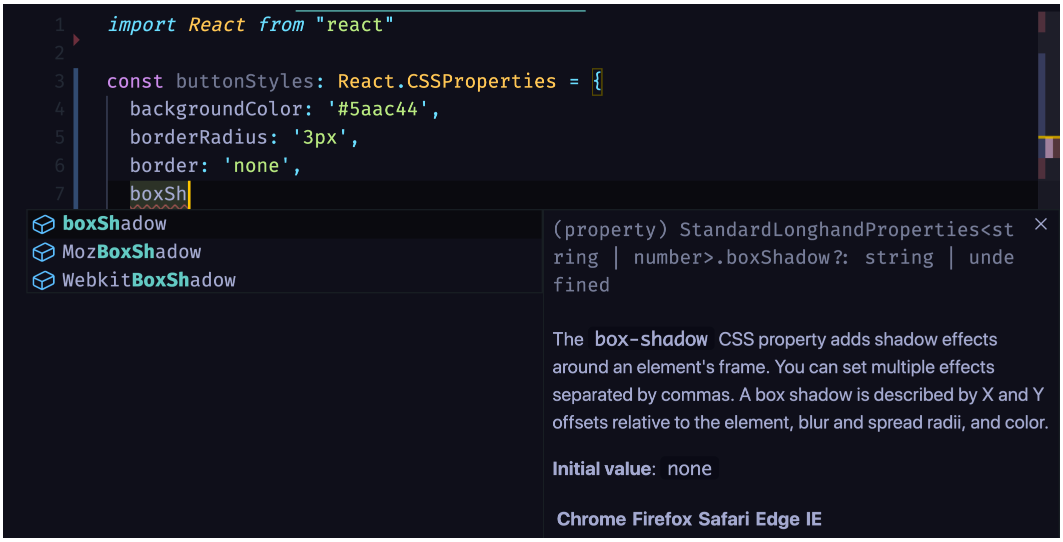
Another option is to pass an object with styling rules through `style` property. You can declare the object inline, then you won't need to specify type for it:

```
<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.



```
import React from "react"

const buttonStyles: React.CSSProperties = {
  backgroundColor: '#5aac44',
  borderRadius: '3px',
  border: 'none',
  boxSh
```

(property) StandardLonghandProperties<string | number>.boxShadow?: string | undefined

The box-shadow CSS property adds shadow effects around an element's frame. You can set multiple effects separated by commas. A box shadow is described by X and Y offsets relative to the element, blur and spread radii, and color.

Initial value: none

Chrome Firefox Safari Edge IE

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[39].

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

```
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 moment of writing this book, Styled Components has **28.4k** stars on Github. It also has Typescript support.

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

Now we are ready to start working on our app layout. Here we'll get to create our first functional components. We'll also define the data structure for our app.

First, we'll create the components that will be our cards and columns. Then we'll arrange them on the screen. During this step, we won't add any interactivity.

We'll need to provide styles for our components. I will use the `styled-components` library. It allows you to create components that only hold styles.

Install `styled-components`:

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

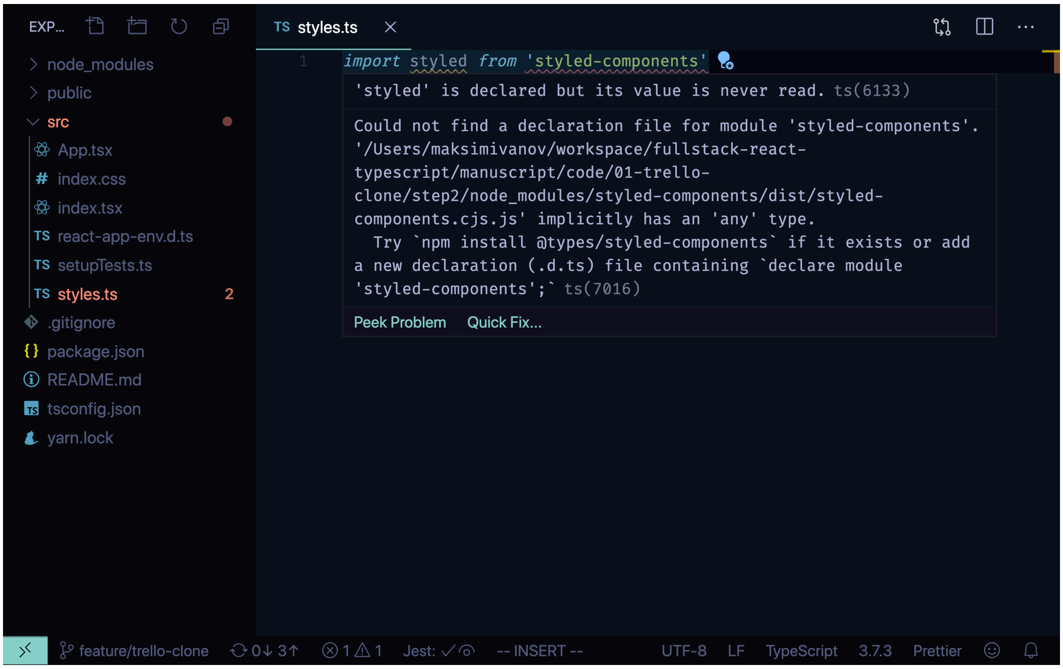
```
yarn add styled-components
```

For convenience, we'll put all the components generated by `styled-components` to `styles.ts`.

Create the `styles.ts` file. Now try to import `styled` from `styled-components`:

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

You'll get a Typescript error.



```
import styled from 'styled-components'
```

```
'styled' is declared but its value is never read. ts(6133)

Could not find a declaration file for module 'styled-components'.
'/Users/maksimivanov/workspace/fullstack-react-
typescript/manuscript/code/01-trello-
clone/step2/node_modules/styled-components/dist/styled-
components.cjs.js' implicitly has an 'any' type.
  Try `npm install @types/styled-components` if it exists or add
a new declaration (.d.ts) file containing `declare module
'styled-components';` ts(7016)
```

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
```

Now we are ready to define our first styled-components.

Prepare Styled Components

We will create a bunch of container elements:

- `AppContainer` - arrange columns horizontally
- `ColumnContainer` - set the grey background and rounded corners
- `ColumnTitle` - make column title bold and add paddings
- `CardContainer`

Let's go one by one.

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 components 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 of 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

```
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 we 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 React documentation[40]. Make sure you know what are props, what is state and how do lifecycle events work.

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 it by using special triangle brackets syntax:

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

```
interface CounterProps {
  message: string;
};
interface CounterState {
  count: number;
};

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

  render() {
    return (
      <div>
        {this.props.message} {this.state.count}
      </div>
    );
  }
}
```

`React.Component` is a generic type that accepts *type variables* for props and state. I will talk more about generics later.

You can find a working class-component example in `code/01-trello/class-components`.

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>
}
```

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

Create Column Component

Time to create our first functional component.

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

```
import React from "react"

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

Update Column Layout

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

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

```
import React from "react"
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.

In Typescript, you need to provide 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. A lot of their features overlap. We'll get to some differences later in this chapter.

This being said I usually define props as an `interface`:

01-first-app/step2/src/Column3.tsx

```
import React from "react"
import { ColumnContainer } from "./styles"

interface ColumnProps {
  text: string
}

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

Here we define an `interface` called `ColumnProps`.

> Sometimes you can see the code where all the interfaces start with the capital `I`. For example `ColumntProps` would be `IColumnProps`. When I first wrote the example code for this chapter I prefixed all interfaces with the capital `I`. But then I read a discussion on github[41] and decided to not use the prefix.

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

[41]https://github.com/typescript-eslint/typescript-eslint/issues/374

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

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

```
interface 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.

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 modify the component:

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

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

Here we make use of `React.PropsWithChildren` type that can enhance your props interface and add a definition for `children` there.

Alternatively, we could manually add `children?: React.ReactNode` to our `ColumnProps` interface, but I think that `React.PropsWithChildren` approach is cleaner.

Here is the `React.PropsWithChildren` type definition:

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

The letter `P` in angle brackets is a *generic type*. It serves as a placeholder for an actual type that we can pass there. It doesn't necessarily have to be `P`, but the convention is to use capital Latin letters.

When we used `React.PropsWithChildren` we've passed our `ColumnProps` interface 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 it's called a *type intersection*.

Create The Card Component

After it'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 React from "react"
import { CardContainer } from "./styles"

interface CardProps {
  text: string
}

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

It will also accept only the text prop. Define the `CardProps` interface for the props

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

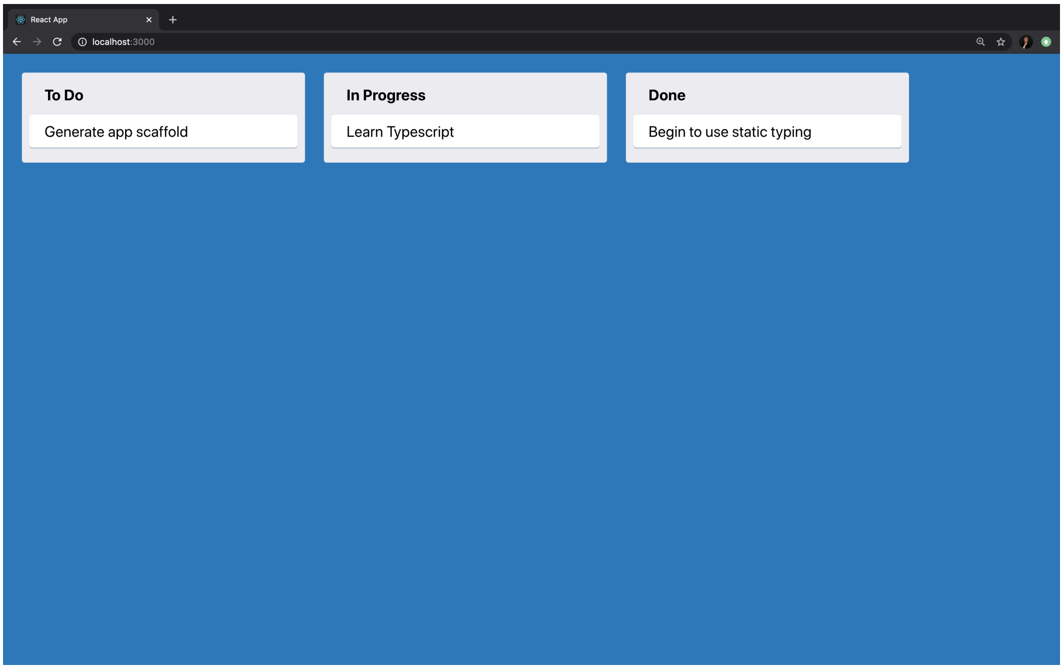
Now and change the layout code to this:

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

```
const App = () => {
  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>
    </AppContainer>
  )
}
```

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

Run `yarn start` and open the browser, you should see this:

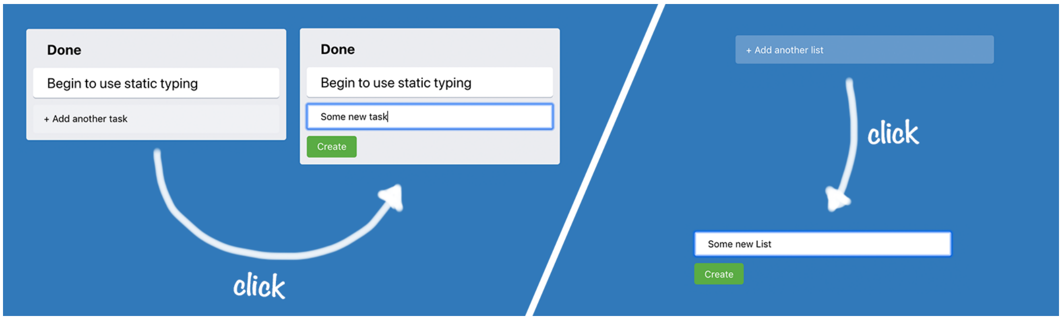


Rendering columns and cards

The only component missing here is a button to create new tasks and lists.

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

Before we move 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 we'll trigger the callback function that we'll pass as a prop.

Prepare Styled Componenets

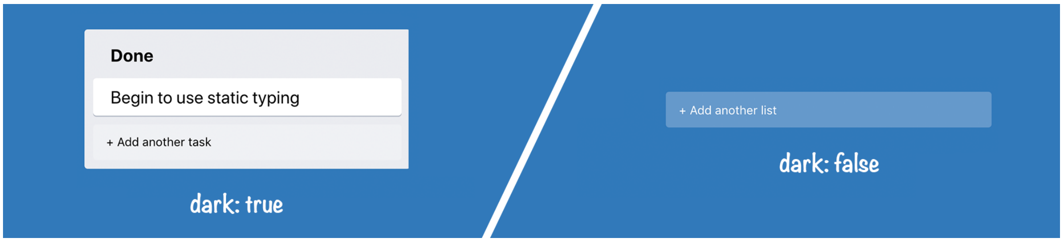
Styles For The Button

Open `src/styles.ts` and define an interface for `AddItemButtonProps`.

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

```
interface AddItemButtonProps {
  dark?: boolean
}
```

We'll use the `AddNewItemButton` component for both lists and tasks. When we'll 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 want the text to have black color.



Button on light and dark background

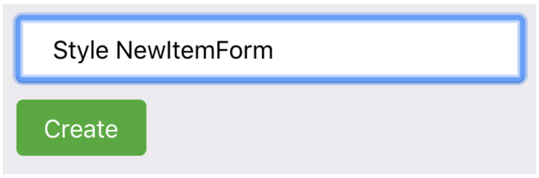
Now define the `AddNetItemButton` styled component:

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;
  }
`
```

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`, import `React` and `AddItemButton` styles:

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

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

This component will accept an item type and some text props for it's buttons. Define an interface for it's props:

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

```
interface 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:

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 `React` with `useState` hook and styled components:

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

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

Define the `NewItemFormProps` interface:

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

```
interface NewItemFormProps {
  onAdd(text: string): void
}
```

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

Now define the `NewItemForm` component:

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

```
  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 - we update the `text` state.

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

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

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

```
  const [showForm, setShowForm] = useState(false)
  const { onAdd, toggleButtonText } = props

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

  return <button onClick={() => setShowForm(true)}>{toggleButtonText}</\
button>
}
```

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

```
const App = () => {
  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>
  )
}
```

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

Adding New Tasks

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

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

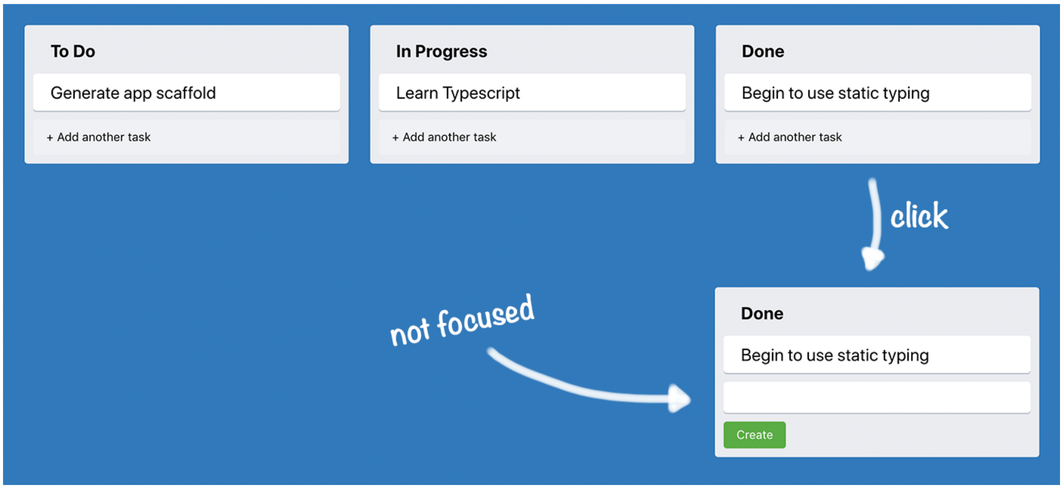
```
export const Column = ({
  text,
  children
}: React.PropsWithChildren<ColumnProps>) => {
  return (
    <ColumnContainer>
      <ColumnTitle>{text}</ColumnTitle>
      {children}
      <AddNewItem
        toggleButtonText="+ Add another task"
        onAdd={console.log}
        dark
      />
  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

Verify That It Works

Let’s launch the app and verify that everything works:

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 React feature called refs.

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

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 will be the element type. So we provide the actual type to it. In our case, we work with input so it's `HTMLInputElement`.

When I need to know what is the name of some element type I usually check @types/react/global.d.ts[42] file. It contains type definitions for types that have to be exposed globally (not in `React` namespace).

Now let's use it in our `NewItemForm`. 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.

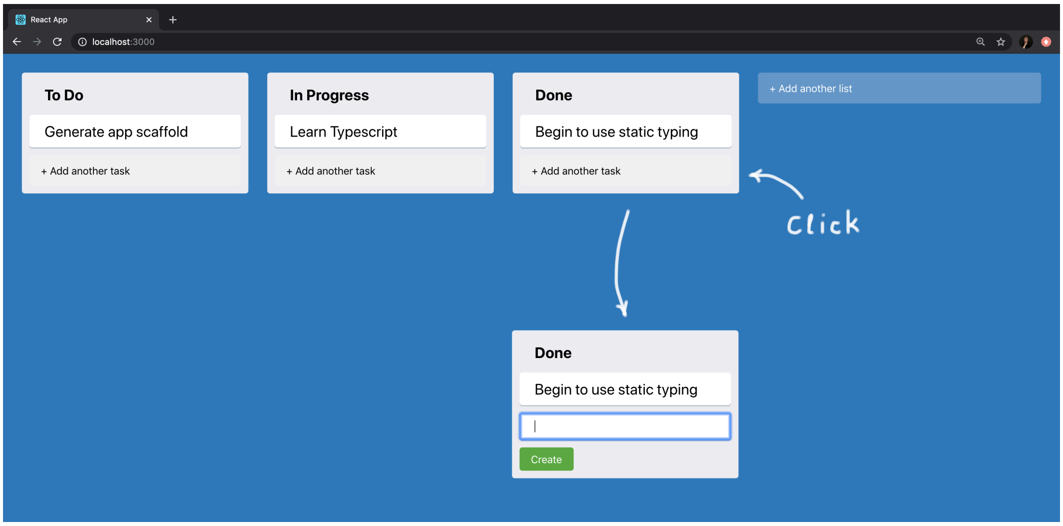
01-first-app/step2/src/NewItemForm.tsx

```
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>
  )
}
```

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

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

Add Global State And Business Logic

In this chapter add interactivity to our application.

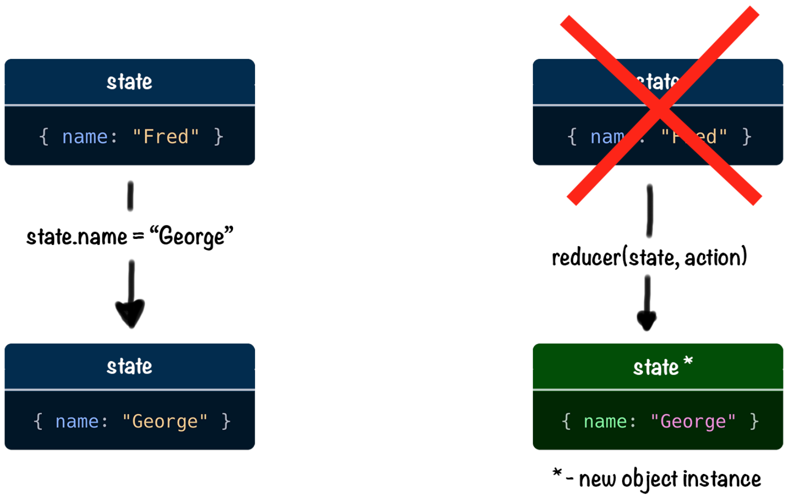
We'll implement drag-and-drop using the React DnD library. And we will add state management. We won't use any external framework like Redux or Mobx. Instead, we'll throw together a poor man's version of Redux using `useReducer` hook and React context API.

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

Using useReducer

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

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

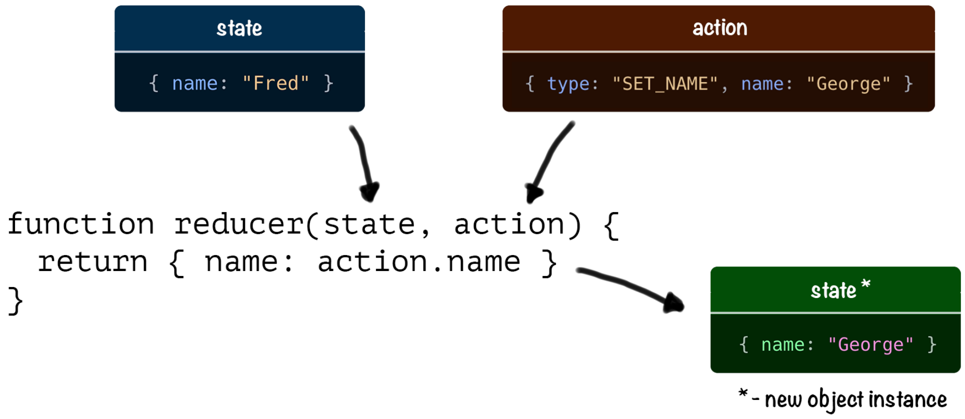


Instead of mutating the object we create a new instance

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

What Is Reducer?

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


state
{ name: "Fred" }
action
{ type: "SET_NAME", name: "George" }
function reducer(state, action) {
  return { name: action.name }
}
state *
{ name: "George" }
* - new object instance

Reducer

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

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 `name` field:

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

How to Call useReducer

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

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.

It's Typescript so we must provide types for `state` and `action` attributes.

We'll define the `State` interface with `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 it's called a *union type*.

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 this chapter.

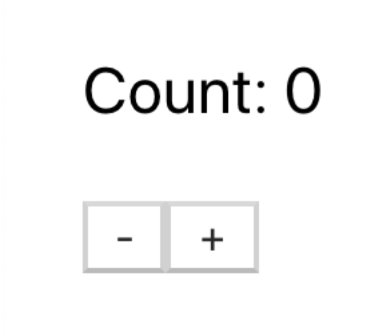
For now let's see how can you use this in your components. Here is a counter component that will use the reducer we've defined previously:

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` function inside of `onClick` handlers. With each `dispatch` call we send an `Action` object. And then we calculate the new state in our counter reducer.

If you launch the app you should see a counter with two buttons:



counter app

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

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/AppStateContext.tsx`. Define the application data, for now let's hardcode it:

01-first-app/step3/src/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" }]
    }
  ]
}
```

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/AppStateContext.tsx

```
interface Task {
  id: string
  text: string
}

interface List {
  id: string
  text: string
  tasks: Task[]
}
```

```
export interface AppState {
  lists: List[]
}
```

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

Now let's use `createContext` to define `AppStateContext`. Import `createContext` from `react`. Import `React` as well, because we'll define a provider component soon:

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

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

Use `createContext` to define the `AppStateContext`.

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

```
interface AppStateContextProps {
  state: AppState
}
```

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 it 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/AppStateContext.tsx

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

Now let's define the `AppStateProvider`. It will pass the hardcoded `appData` through the `AppStateContext.Provider`:

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

```
export const AppStateProvider = ({ children }: React.PropsWithChildren<\
{}>) => {
  return (
    <AppStateContext.Provider value={{ state: appData }}>
      {children}
    </AppStateContext.Provider>
  )
}
```

Our component will only accept `children` as a prop. We use `React.propsWithChildren` type. It requires one generic argument, but we don't want to have any other props so we pass an empty object to it.

Go to `src/index.tsx` and wrap the `<App/>` component into `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 "./AppStateContext"

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

Now we'll be able to get `state` and `dispatch` from any component.

To make it easier to access them - let's create a custom hook.

Using Data From Global Context. Implement Custom Hook

Go back to `src/AppStateContext.tsx` and import `useContext`:

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

```
import React, { createContext, useReducer, useContext } from "react"
```

Then define a new function called `useAppState`:

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

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

Inside of this function, we retrieve the value from `AppStateContext` using `useContext` hook and return the result.

Get The Data From AppStateContext

Go to `src/App.tsx`. Let's use our `useAppState` hook to retrieve the `state`.

Import the hook:

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

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

Then update the layout to use the `appData`:

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

```
const App = () => {
  const {state} = useAppState()

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

If you check the type of the `state` constant - you'll see that it is `AppState`. Typescript derived this type automatically because we've already provided it when we called `createContext`.

If we make a typo and instead of `list.text` we'll white `list.test` - Typescript will correct us and show a list of available fields.

In `src/App.tsx` we started to pass an `index` prop to our columns. We'll use it to retrieve a list of cards to render.

Update the `Column` component. Remove `children` prop from the props and add `index: number` prop:

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

```
interface ColumnProps {
  text: string
  index: number
}
```

Import the `useAppState` hook:

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

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

Change the layout. We call `useAppState` to get the data. Then we get the column by `index`. This is why we are passing it as a prop to the `Column` component. Then we iterate over the cards and render the `Card` components.

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

```
export const Column = ({ text, index }: ColumnProps) => {
  const { state } = useAppState()

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

Now all our components can get app data from the context. Time to make it possible to update the data. Let’s add some actions and reducers.

Adding Items. Typescript Interfaces Vs Types

In this chapter, we'll define 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.

Define Actions

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

Open `src/AppStateContext` and define a new type:

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

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

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

We've defined a type `Action` and then we've passed two interfaces separated by a vertical line to it. It means that `Action` now can resolve to one of the forms that we've passed.

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

For example here is an `if` statement:

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

Here Typescript already knows that `action.payload` can only be a `string`. The interface that has `type: "ADD_LIST"` has a `payload` field defined as `string`. We can use it do define our reducers.

Define appStateReducer

Inside `src/AppStateContext.tsx` define `appStateReducer`, it should look like this:

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

```
const appStateReducer = (state: AppState, action: Action): AppState => {
  switch (action.type) {
    case "ADD_LIST": {
      // Reducer logic here...
      return {
        ...state
      }
    }
    case "ADD_TASK": {
      // Reducer logic here...
      return {
        ...state
      }
    }
    default: {
      return state
    }
  }
}
```

We don't have to define constants for our action types. Typescript will give you an error if you try to compare `action.type` to something it cannot be.

Here is also another catch, note that we use curly brackets to define the block scope for our `case` statements. Without those brackets, our constants would be visible across the whole `switch` block.

Let's say you have your reducer defined like this, without curly brackets:

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

```
const appStateReducer = (state: AppState, action: Action): AppState => {
  switch (action.type) {
    case "ADD_LIST":
      const visibilityExample = "Too visible"
      return {
        ...state
      }
    case "ADD_TASK":
      const visibilityExample = "Too visible"
      return {
        ...state
      }
    default: {
      return state
    }
  }
}
```

Typescript will give you an error:

```
Cannot redeclare block-scoped variable 'visibilityExample'.ts(2451)
```

So don't forget to use the curly brackets.

Provide Dispatch Through The Context

Open the `src/AppStateContext.tsx` and update the `AppStateProvider`:

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

```
export const AppStateProvider = ({ children }: React.PropsWithChildren<\
{}>) => {
  const [state, dispatch] = useReducer(appStateReducer, appData)

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

Now we provide the `state` value from our `appStateReducer` instead of using hardcoded `appData`.

Adding Lists

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

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

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

New column has `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 `uuid` to generate new identifiers.

We need to install this library. It doesn't include type definitions so we install them as well:

```
yarn add uuid @types/uuid
```

Now import `uuid` inside `src/AppStateContext`:

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

```
import uuid from 'uuid'
```

Adding Tasks

Adding tasks is a bit more complex because they need to be added to specific lists `tasks` array. We'll need to find the list by it's `id`. Let's add `findItemIndexById` method.

Create a new file `src/utils/findItemIndexById`. We are going to define a function that will accept any object that has a field `id: string`.

Define a new interface `Item`.

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

```
interface Item {
  id: string
}
```

Now we will use generic type `T` that extends `Item`. That means that we constrained our generic to have the fields that are defined on the `Item` interface. In this case the `id` field.

Define the function:

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

```
export const findItemIndexById = <T extends Item>(items: T[], id: strin\
g) => {
  return items.findIndex((item: T) => item.id === id)
}
```

Now go back to `src/AppStateContext` and add the code for `ADD_TASK` block:

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

```
    case "ADD_TASK": {
      const targetLaneIndex = findItemIndexById(
        state.lists,
        action.payload.taskId
      )
      state.lists[targetLaneIndex].tasks.push({
        id: uuid(),
        text: action.payload.text
      })

      return {
        ...state
      }
    }
```

Here we first find the target list index and save it to `targetListIndex` constant.

Then we push a new task object to the list with that index.

And then we return a new object, created from the old state using object spread syntax.

Dispatching Actions

Go to `src/App.tsx` and update the code. Now we also get the `dispatch` function from the `useAppState` hook.

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

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

  return (
    <AppContainer>
      {state.lists.map((list, i) => (
        <Column id={list.id} text={list.text} key={list.id} index={i}/>
      ))}
      <AddNewItem
        toggleButtonText="+ Add another list"
        onAdd={text => dispatch({ type: "ADD_LIST", payload: text })}
      />
    </AppContainer>
  )
}
```

Also update the `AddNewItem` `onAdd` function. Now we'll call the `dispatch` method there, passing the `text` as a `payload`.

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

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

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

  return (
    <ColumnContainer>
      <ColumnTitle>{text}</ColumnTitle>
      {state.lists[index].tasks.map((task, i) => (
        <Card text={task.text} key={task.id} index={i} />
      ))}
      <AddNewItem
        toggleButtonText="+ Add another card"
        onAdd={text =>
          dispatch({ type: "ADD_TASK", payload: { text, taskId: id } })
        }
        dark
      />
    </ColumnContainer>
  )
}
```

Here we also call the `dispatch` function. We pass the `taskId` alongside `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.

Moving Items

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

Moving Columns

Add a new `Action` type:

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

```
type Action =
  // ... Previously defined actions
  {
    type: "MOVE_LIST"
    payload: {
      dragIndex: number
      hoverIndex: number
    }
  }
```

We've added a `MOVE_LIST` action. This action has `dragIndex` and `hoverIndex` in its payload. When we start dragging the column - we remember the original position of it and then pass it as `dragIndex`. When we hover other columns we take their positions and use them as a `hoverIndex`.

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

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

```
const appStateReducer = (state: AppState, action: Action): AppState => {
  switch (action.type) {
    // ... Previously defined 'case' blocks

    case "MOVE_LIST": {
      const { dragIndex, hoverIndex } = action.payload
      state.lists = moveItem(state.lists, dragIndex, hoverIndex)
      return { ...state }
    }
    default: {
      return state
    }
  }
}
```

Here we take `dragIndex` and `hoverIndex` from the action payload. Then we calculate the new value for the `lists` array. To do this we use the `moveItem` function, which takes the source array, and two indices that it will swap.

Create a new file `src/moveItem.ts` that will hold this function:

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

```
export const moveItem = <T>(array: T[], from: number, to: number) => {
  const startIndex = to < 0 ? array.length + to : to;
  const item = array.splice(from, 1)[0]
  array.splice(startIndex, 0, item)
  return array
}
```

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

Then we calculate the `startIndex`. We make sure that it's always a positive number. If our destination index is smaller than zero - we use array length plus the destination index. We do this because if you pass a negative index to `splice` function it will begin that many elements from the end. So we can end up adding an item to the wrong spot.

After we've calculated the `startIndex` that is always a positive number we can move items around. First, we remove the item with the `from` index and store it in the `item` const. Then we insert that item at `startIndex` position.

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 react-dnd-html5-backend
```

`react-dnd` has type definitions included, 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 Backend from "react-dnd-html5-backend"
import { AppStateProvider } from "./AppStateContext"

ReactDOM.render(
  <DndProvider backend={Backend}>
    <AppStateProvider>
      <App />
    </AppStateProvider>
  </DndProvider>,
  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 some 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 the `type` field that for now will be `COLUMN`. We'll also pass the column's `id`, `text` and `index` that we'll get from the `Column` component.

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

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

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

export type DragItem = ColumnDragItem
```

Later we will add a `CardDragItem` to it.

Store The Dragged Item In State

Unfortunately, you can only access currently dragged item data from `react-dnd` hooks callbacks.

It's not enough for us. For example, when we drag the column `react-dnd` will create a drag preview that we'll move around with our cursor. This drag preview will look like the component that we started to drag. If we don't hide the original component - it will look like we are dragging a duplicate.

To fix it we need to hide the item that we are currently dragging. To do this we need to know what kind of item are we dragging. We need to know the `type`, to know if it's a card or a column. And we need to know the `id` of this particular item.

Let's store the dragged item in our app state. Create a new action `SET_DRAGGED_ITEM`:

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

```
type Action =
  | {
      type: "SET_DRAGGED_ITEM"
      payload: DragItem | undefined
    }
```

It will hold the `DragItem` that we defined earlier. We need to be able to set it to `undefined` if we are not dragging anything.

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

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

```
    case "SET_DRAGGED_ITEM": {
      return { ...state, draggedItem: action.payload }
    }
```

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

Define 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 `undefined` as payload.

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

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

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

export const useItemDrag = (item: DragItem) => {
  const { dispatch } = useAppState()
  const [, drag ] = useDrag({
    item,
    begin: () =>
      dispatch({
        type: "SET_DRAGGED_ITEM",
        payload: item
      }),
    end: () => dispatch({ type: "SET_DRAGGED_ITEM", payload: undefined \
})
  })
  return { drag }
}
```

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

- `item` - contains the data about the dragged item
- `begin` - is called when we start dragging an item
- `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 `undefined`.

Drag Column

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

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

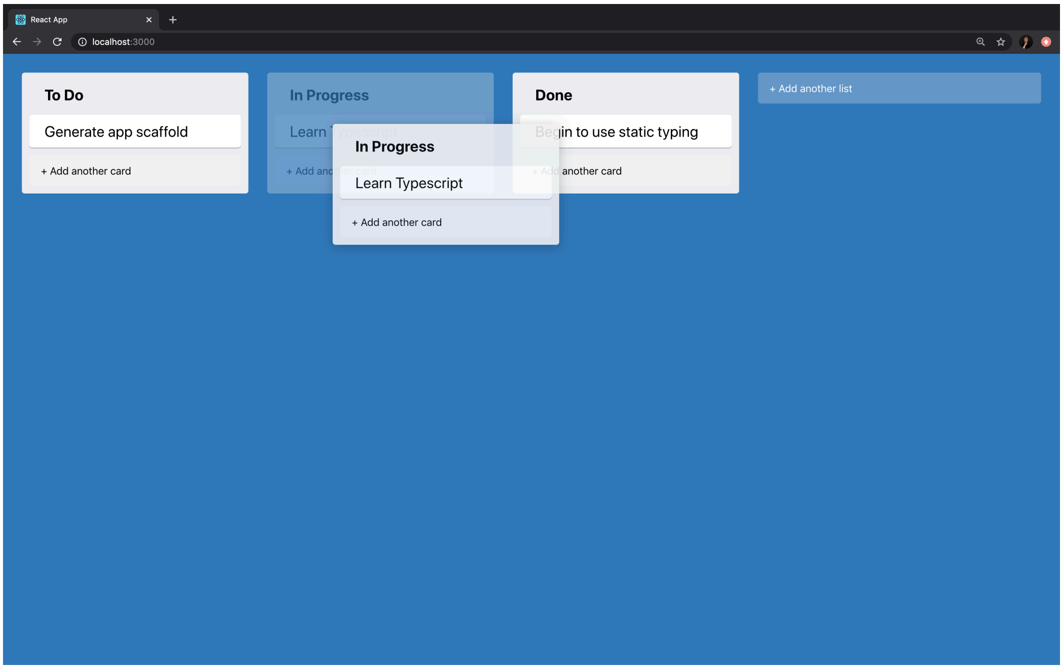
```
export const Column = ({ text, index, id }: ColumnProps) => {
  const { state, dispatch } = useAppState()
  const ref = useRef<HTMLDivElement>(null)

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

  drag(ref)

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

We need a `ref` to specify as a 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 `ColumnContaner`.

Then we call our `useItemDrag` hook. We pass an object that will represent the dragged item. We 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 the 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 import `useDrop` from `react-dnd`:

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

```
import { useDrop } from "react-dnd"
```

Now add this code in the beginning of the `Column` component:

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

```
  const [, drop] = useDrop({
    accept: "COLUMN",
    hover(item: DragItem) {
      const dragIndex = item.index
      const hoverIndex = index

      if (dragIndex === hoverIndex) {
        return
      }

      dispatch({ type: "MOVE_LIST", payload: { dragIndex, hoverIndex } \
})
      item.index = hoverIndex
    }
  })
```

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 Column

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`.

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;
`
```

Calculate 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 | undefined,
  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 update the layout. We now pass the result of `isHidden` function to `isHidden` prop of our `ColumnContainer`:

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

```
    <ColumnContainer ref={ref} isHidden={isHidden(state.draggedItem, "C\
OLUMN", id)}>
      <ColumnTitle>{text}</ColumnTitle>
      {state.lists[index].tasks.map((task, i) => (
        <Card text={task.text} key={task.id} index={i} />
      ))}
      <AddNewItem
        toggleButtonText="+ Add another card"
        onAdd={text =>
          dispatch({ type: "ADD_TASK", payload: { text, columnId: id } \
})
        }
        dark
      />
    </ColumnContainer>
  )
```

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

Implement 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 viewport.

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

01-first-app/step6/src/stylestsx

```
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 import `useDragLayer` from `react-dnd`:

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

```
import { useDragLayer } from "react-dnd"
```

Create a `CustomDragLayer` component:

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

```
const CustomDragLayer: React.FC = () => {
  const { isDragging, item } = useDragLayer(monitor => ({
    item: monitor.getItem(),
    isDragging: monitor.isDragging()
  }))

  return isDragging ? (
    <CustomDragLayerContainer>
      <Column
        id={item.id}
        text={item.text}
        index={item.index}
      />
    </CustomDragLayerContainer>
  ) : null
}
```

Here we use `useDragLayer` to obtain `isDragging` flag and currently dragged `item` object. Then we render our layout if `isDragging` is true, otherwise, we return `null` and render nothing.

We use an actual `Column` component to render a preview. We pass it `id`, `index` and `text` from the `item` object.

Move The Dragged Item Preview

Right now we just render the preview component. We need to write some extra code to make it follow the cursor.

We will write a function that will get the dragged item coordinates from `react-dnd` and generate the styles with the `transform` attribute to move the preview around.

In this function, we'll need to use `XYCoord` type from `react-dnd`. Import it from the library.

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

```
import { XYCoord, useDragLayer } from "react-dnd"
```

Here is the function to generate new styles:

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

```
function getItemStyles(currentOffset: XYCoord | null): React.CSSPropert\
ies {
  if (!currentOffset) {
    return {
      display: "none"
    }
  }

  const { x, y } = currentOffset

  const transform = `translate(${x}px, ${y}px)`
  return {
    transform,
    WebkitTransform: transform
  }
}
```

We can manually set the return value of this function to be `React.CSSProperties`. It's not required, but can be useful, because then if you will make a mistake - you'll get an error inside the function instead of the place where you pass the resulting style as a prop to your component.

This function accepts a `currentOffset` argument that has the `XYCoord` type. It contains a currently dragged item position. We take `x` and `y` fields from the `currentOffset` and generate the value for CSS `transform` property.

Add a wrapping `div` element around the `Column` preview. Now we can use the `getItemStyles` function to specify the styles for our wrapping `div`.

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

```
const CustomDragLayer: React.FC = () => {
  const { isDragging, item, currentOffset } = useDragLayer(monitor => ({
    item: monitor.getItem(),
    currentOffset: monitor.getSourceClientOffset(),
    isDragging: monitor.isDragging()
  }))

  return isDragging ? (
    <CustomDragLayerContainer>
      <div style={getItemStyles(currentOffset)}>
        // ...Dragged item preview
      </div>
    </CustomDragLayerContainer>
  ) : null
}
```

Here we also get the `currentOffset` value from the `useDragLayer` hook. Pass this value to our `getItemStyles` function.

After we create our `CustomDragLayer` component we need to do two things. First, we need to mount the component inside the `App` layout, and then we'll need to hide the default drag preview.

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

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

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

  return (
    <AppContainer>
      <CustomDragLayer />
      {state.lists.map((list, i) => (
        <Column id={list.id} text={list.text} key={list.id} index={i}/>
      ))}
      <AddNewItem
        toggleButtonText="+ Add another list"
        onAdd={text => dispatch({ type: "ADD_LIST", payload: text })}
      />
    </AppContainer>
  )
}
```

Hide The Default Drag Preview

To hide the default drag preview we'll have to modify the `useItemDrag` hook.

Open `src/useItemDrag.ts`. We'll use `getEmptyImage` function to create the preview that won't be rendered. Import the function from `react-dnd-html5-backend`:

<<01-first-app/step6/src/useItemDrag.ts[43]

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

[43] ./code/01-first-app/step6/src/useItemDrag.ts

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

```
export const useItemDrag = (item: DragItem) => {
  const { dispatch } = useAppState()
  const [, drag, preview ] = useDrag({
    item,
    begin: () =>
      dispatch({
        type: "SET_DRAGGED_ITEM",
        payload: item
      }),
    end: () => dispatch({ type: "SET_DRAGGED_ITEM", payload: undefined \
})
  })
  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`.

Launch the app - you'll see that now the default preview is not visible. Problem is that our custom preview is not rendered either.

Make The Custom Preview Visible

Our custom preview is hidden because it uses the same `id` and `index` as the currently dragged column. We need to add `isPreview` condition to our `isHidden` function.

Open `src/utils/isHidden`, add a new `boolean` argument `isPreview`:

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

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

Now we need to add this argument to our `Column` component. First add it to `ColumnProps` interface:

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

```
interface ColumnProps {
  text: string
  index: number
  id: string
  isPreview?: boolean
}
```

Now pass this prop to `isHidden` function call.

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

```
export const Column = ({ text, index, id, isPreview }: ColumnProps) => {
  // ... the rest of the code

  return (
    <ColumnContainer
      ref={ref}
      isHidden={isHidden(isPreview, state.draggedItem, "COLUMN", id)}
    >
      // ... Column layout
    </ColumnContainer>
  )
}
```

We used to have the dragged column opacity to be `0.3`, it was a hack to keep the preview visible before we created a custom preview component. Open `src/styles.ts` and set the hidden state `opacity` to `0`

01-first-app/step6/src/styles0.ts

```
export const DragPreviewContainer = styled.div<DragPreviewContainerProp\
s>`
  opacity: ${props => (props.isHidden ? 0 : 1)};
`
```

Tilt The Custom Preview

Add a new `isPreview` property to our `DragPreviewContainer` component to rotate it a few degrees.

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

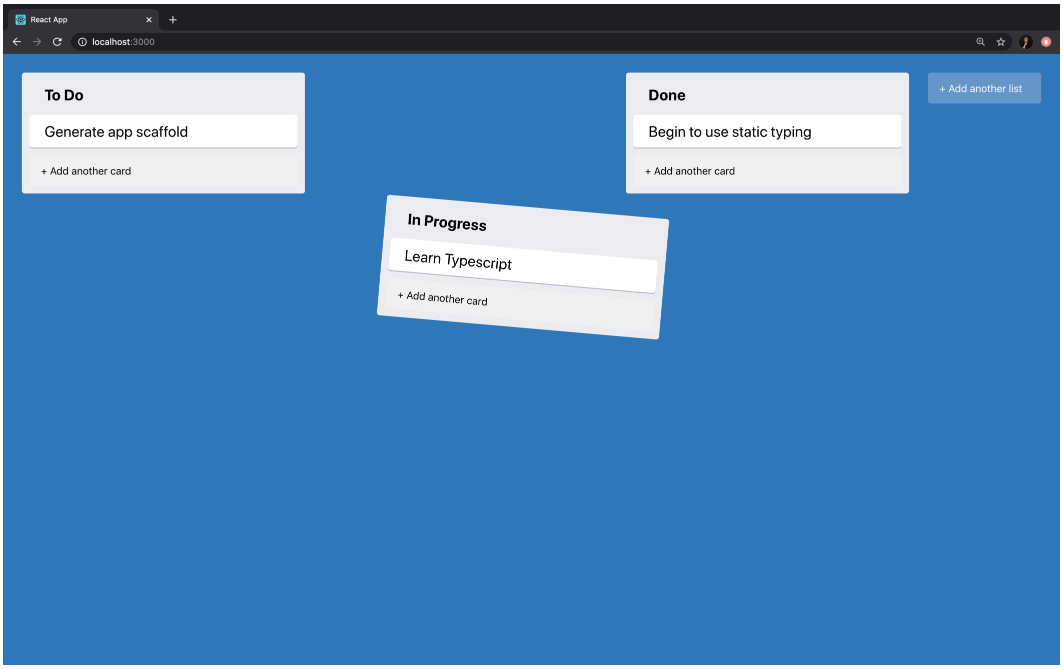
```
export const DragPreviewContainer = styled.div<DragPreviewContainerProp\
s>`
  transform: ${props => (props.isPreview ? "rotate(5deg)" : undefined)};
  opacity: ${props => (props.isHidden ? 0 : 1)};
`
```

Now go back to `src/Column.tsx` and pass `isPreview` as a prop to `ColumnContainer`:

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

```
export const Column = ({ text, index, id, isPreview }: ColumnProps) => {
  // ... the rest of the code

  return (
    <ColumnContainer
      isPreview={isPreview}
      ref={ref}
      isHidden={isHidden(isPreview, state.draggedItem, "COLUMN", id)}
    >
      // ... Column layout
    </ColumnContainer>
  )
}
```

Launch the app, now you can drag columns around and they will have this nice little tilt to them.



Tilted column drag-preview

Drag Cards

Time to drag cards around. Open `src/DragItem.ts` and add the `CardDragItem` type.

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

```
export type CardDragItem = {
  index: number
  id: string
  columnId: string
  text: string
  type: "CARD"
}

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

export type DragItem = CardDragItem | ColumnDragItem
```

Also, update the `DragItem` type to be either a `CardDragItem` or a `ColumnDragItem`.

Update The Reducer

First we need to add a new `Action` type. Open `src/AppStateContext.tsx` and add `MOVE_TASK` action:

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

```
type Action =
  // ... previously defined actions
  {
    type: "MOVE_TASK"
    payload: {
      dragIndex: number
      hoverIndex: number
      sourceColumn: string
      targetColumn: string
    }
  }
```

This action accepts `dragIndex` and `hoverIndex` just like `MOVE_LIST`, but it also needs to know between which columns do we drag the card. So it also contains `sourceColumn` and `targetColumn` attributes that hold source and target column ids.

Also we need to add a new `case` block to our reducer:

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

```
const appStateReducer = (state: AppState, action: Action): AppState => {
  switch (action.type) {
    // ... other 'case' blocks

    case "MOVE_TASK": {
      const {
        dragIndex,
        hoverIndex,
        sourceColumn,
        targetColumn
      } = action.payload
      const sourceLaneIndex = findItemIndexById(state.lists, sourceColu\
mn)
      const targetLaneIndex = findItemIndexById(state.lists, targetColu\
mn)
      const item = state.lists[sourceLaneIndex].tasks.splice(dragIndex,\
 1)[0]
      state.lists[targetLaneIndex].tasks.splice(hoverIndex, 0, item)
      return { ...state }
    }
    default: {
      return state
    }
  }
}
```

Our `sourceColumn` and `targetColumn` are column ids so first, we find their corresponding indices in `columns` array. Then we use `splice` to remove the card from the source column and then another `splice` to add it to the target column.

Implement The useDrop

Next we need to make our cards to be drop targets. Open `src/Card.tsx` and add this `useDrop` block:

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

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

      const dragIndex = item.index
      const hoverIndex = index
      const sourceColumn = item.columnId
      const targetColumn = columnId

      dispatch({
        type: "MOVE_TASK",
        payload: { dragIndex, hoverIndex, sourceColumn, targetColumn }
      })
      item.index = hoverIndex
      item.columnId = targetColumn
    }
  })
```

Now just like in the `Column` component, wrap your ref into the `drop` call.

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 `dragIndex` and `sourceColumn` from the dragged item, and `hoverIndex` and `targetColumn` from the hovered card.

We dispatch those values inside the `MOVE_TASK` action payload.

The last thing we do - we set the dragged item's `index` and `columnId` to match the fields of the hovered card.

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

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

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

Now launch the app and enjoy dragging the cards around. Pretty soon you might notice that after you've moved all the cards from some column - you can't move them back. Let's fix it.

Drag a Card 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 is the actual type of our dragged item. The `item` has `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 `item.type` is `COLUMN` - then we do what we did before. Just leave the previous logic there. Otherwise, we will calculate the `hoverIndex` differently. Remember - we are hovering a column and its index is not very useful when we are dragging the card. So we just set the `hoverIndex` to `0`.

Then we store `item.columnId` as `sourceColumn`. Here we just prepare a `const` to match the field name of our `action.payload`.

Next, we take the `id`, we do it inside a `Column` component so it's not `columnId` and store it as `targetColumn`.

We check that the source and the target columns are different and if that's the case - we dispatch an action.

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

```
    hover(item: DragItem) {
      if (item.type === "COLUMN") {
        // ... draggin column
      } else {
        const dragIndex = item.index
        const hoverIndex = 0
        const sourceColumn = item.columnId
        const targetColumn = id

        if (sourceColumn === targetColumn) {
          return
        }

        dispatch({
          type: "MOVE_TASK",
          payload: { dragIndex, hoverIndex, sourceColumn, targetColumn }
        })
        item.index = hoverIndex
        item.columnId = targetColumn
      }
    }
```

Last thing is to update the dragged item's `index` and `columnId` to match the new values.

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 chapters, 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 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 backend works correctly by manually sending cURL requests. There are two endpoints available. One for storing data and one for retrieving.

Here is a command to store the data:

```
curl --header "Content-Type: application/json" \
  --request POST \
  --data '{"lists":"[]"}' \
  http://localhost:4000/api/login
```

And here is the one to retrieve:

```
curl http://localhost:4000/load
```

Every time you `POST` a JSON object to `/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.

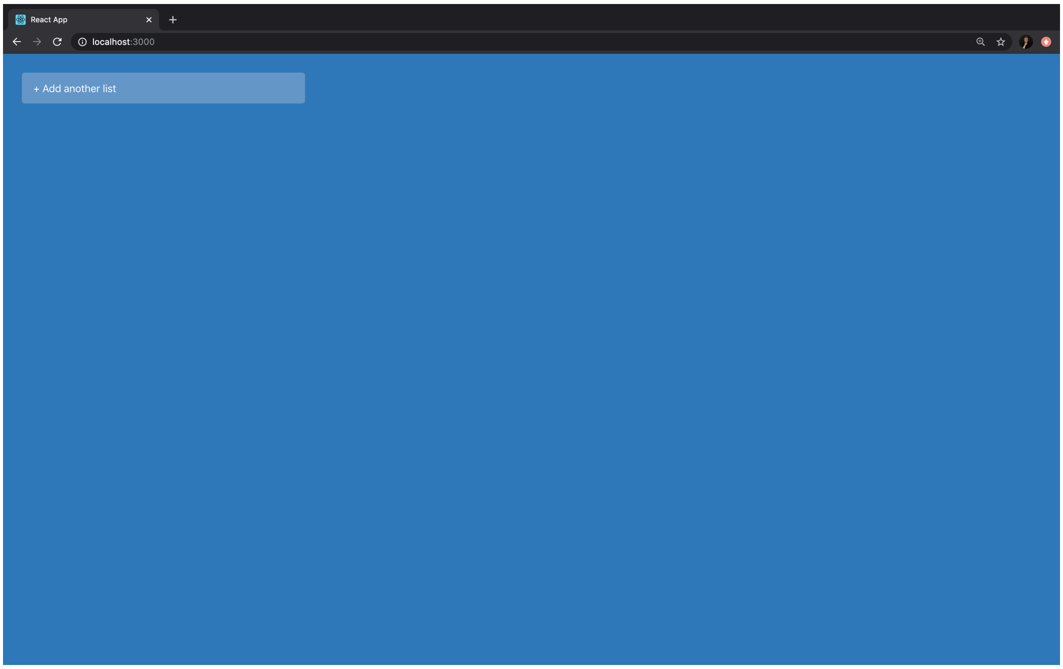
Launch the sample backend in a separate terminal tab:

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

Completed example for this chapter is located in `code/02-trello-clone/step9`, `cd` to this folder and launch the app:

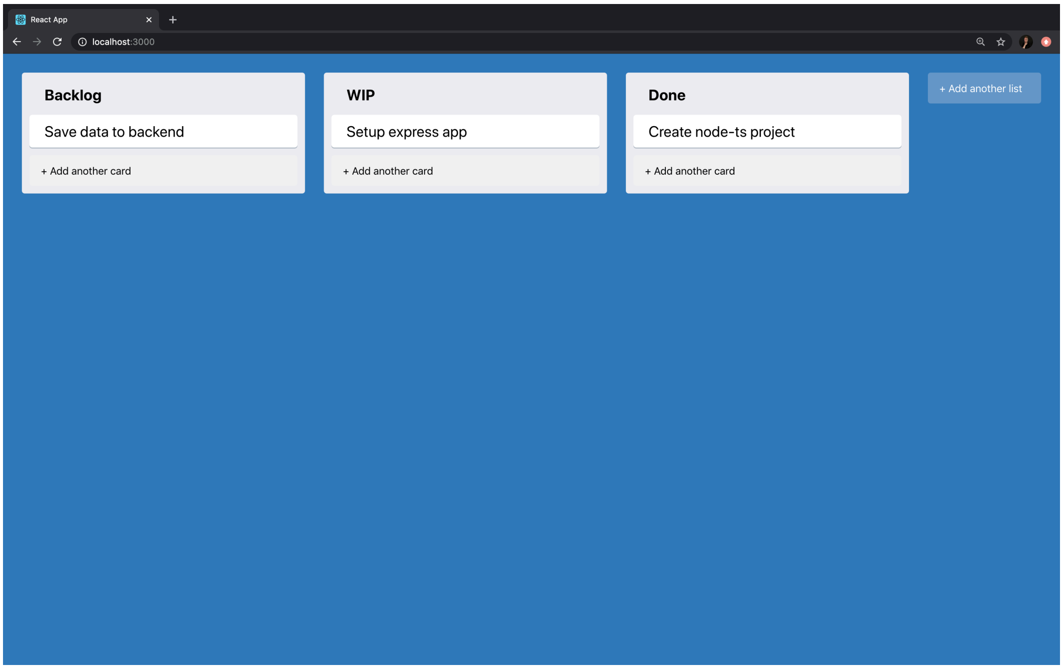
```
cd code/01-first-app/step9
yarn && yarn start
```

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 where you've finished.

If you didn't follow the previous chapters - then you can use the `code/02-trello-clone/step9` as your starting point. Copy the folder somewhere in your working projects directory.

Using Fetch With Typescript

Browser Javascript has a built-in `fetch` method that allows making network requests. Here is 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, among 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 Typescript Playground[44].

Here I specified the return value of `json()` function call to be have 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) => {
    return response.json() as Promise<T>;
  })
}
```

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.

It allows me to use this function like this:

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

```
const { user_search_url } = await githubAPI<{user_search_url: string}>(\
);
```

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.

This approach has a bunch of benefits:

- **We are not bound to a specific `fetch` implementation.** If you want to switch to axios[45] - 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[46]

Now create a new file `api.ts` and define the `save` function:

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

```
import { AppState } from "./AppStateContext"

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),
  })
```

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

```
    .then((response) => {
      return response.json()
    })
    .catch(console.log)
}
```

This function will accept the current state and send it to the backend as JSON.

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) => {
      return response.json() as Promise<AppState>
    }
  )
}
```

This function will load the previously saved data from the backend. We cast the JSON parsing result to the `AppState` type.

Ok, now you have an API with two functions:

- `save` function makes a `POST` request and sends a JSON representation of our application state to the backend.
- `load` function makes a `GET` request to retrieve previously saved state.

Saving The State

We want to save our application state every time it changes. It 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 backend we'll use a `useEffect` hook.

Add the following code to `src/AppStateContext.tsx` right before the `AppStateProvider` return statement:

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

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

Don't forget to import the `useEffect` hook from React.

The useEffect[47] hook allows to run side effect callbacks on some value change.

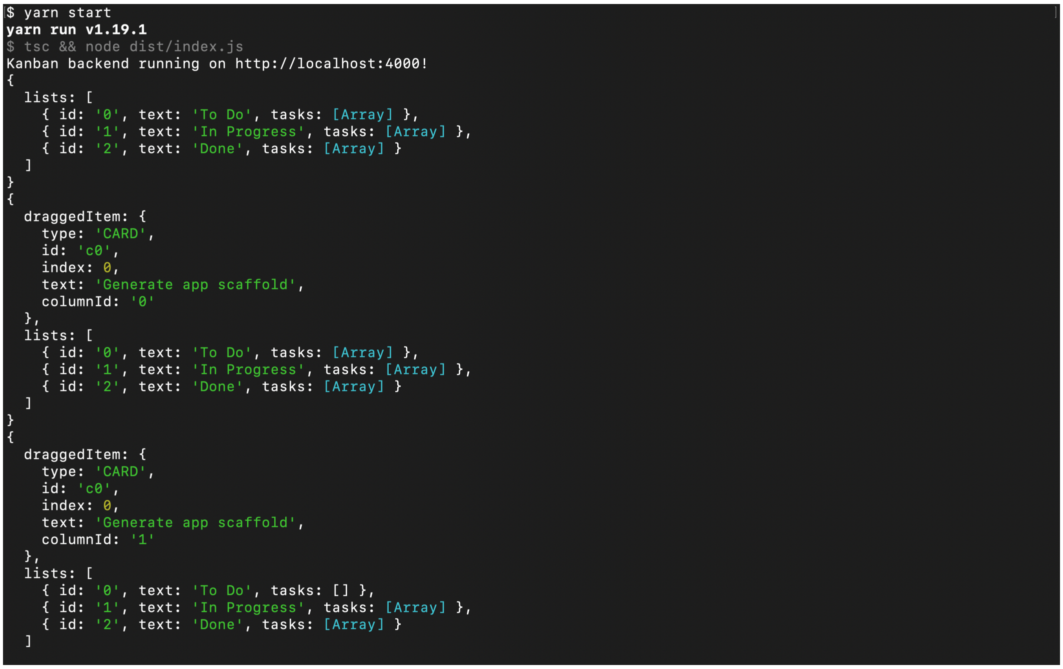
It accepts a callback function and a dependency array. Then it triggers the callback function every time the variables in the dependency array get updated.

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 console.

Try to drag the items around and then check the backend console output. It should look like this:

47



```
$ yarn start
yarn run v1.19.1
$ tsc && node dist/index.js
Kanban backend running on http://localhost:4000!
{
  lists: [
    { id: '0', text: 'To Do', tasks: [Array] },
    { id: '1', text: 'In Progress', tasks: [Array] },
    { id: '2', text: 'Done', tasks: [Array] }
  ]
}
{
  draggedItem: {
    type: 'CARD',
    id: 'c0',
    index: 0,
    text: 'Generate app scaffold',
    columnId: '0'
  },
  lists: [
    { id: '0', text: 'To Do', tasks: [Array] },
    { id: '1', text: 'In Progress', tasks: [Array] },
    { id: '2', text: 'Done', tasks: [Array] }
  ]
}
{
  draggedItem: {
    type: 'CARD',
    id: 'c0',
    index: 0,
    text: 'Generate app scaffold',
    columnId: '1'
  },
  lists: [
    { id: '0', text: 'To Do', tasks: [] },
    { id: '1', text: 'In Progress', tasks: [Array] },
    { id: '2', text: 'Done', tasks: [Array] }
  ]
```

Backend console output

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 being mounted once when we render our application. Problem is that we can't load the data directly inside of it because then our application will first initialize with the default data, then we will get the data from the backend but it wouldn't matter as our reducer would already be initialized.

The solution is to have a wrapper component that will load the data for us and then will 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 of it. But I propose to create a more generic solution using the HOC pattern.

What is HOC

HOC is a React pattern where 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.

Create a new file `src/withData.tsx`.

Make necessary imports:

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

```
import React, { PropsWithChildren, ComponentType } from "react"
import { AppState } from "./AppStateContext"
```

Then define and export our `withData` HOC:

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

```
export const withData = (
  WrappedComponent: ComponentType<PropsWithChildren<{ initialState: App\
State }>>
) => {
  return ({ children }: PropsWithChildren<{}>) => {
    const initialState: AppState = { lists: [], draggedItem: undefined }

    // Here will go the logic of our HOC

    return (
      <WrappedComponent initialState={initialState}>
        {children}
      </WrappedComponent>
    )
  }
}
```

Let's go line by line. First we define a `withData` function that accepts `WrappedComponent` argument. This `WrappedComponent` has a complex type declaration:

```
ComponentType<PropsWithChildren<{ initialState: AppState }>>
```

Here we composed the type of our `WrappedComponent` from several *generic* types. What we meant here is that we want to have a React component, that accepts children and also additional `initialState` prop of type `AppState`.

Then inside of our function, we return a nameless component that accepts `children` prop. Our `AppStateProvider` will accept the `initialState` prop. This nameless component will play the role of modified `AppStateProvider` that only accepts the `children`.

Inside of the nameless internal component we have hardcoded the initial state.

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

Here you go. 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 the advanced React patterns, where we talk in more detail about them.

Load Data Inside HOC

Inside this function, we create a functional component that uses `useEffect` hook to load data.

Call a `useEffect` hook and call our `load` function from the `API` module when it gets triggered.

We will have three different states:

- **Pending**. We have this state when we've started loading data but didn't finish yet. We need to render some kind of loader.
- **Success**. The data is loaded successfully. We can render our app.
- **Failure**. We got a network error. We need to render the error message.

Remove the following line:

```
const initialState: AppState = { lists: [], draggedItem: undefined }
```

Define states for `pending`, `success`, and `failure` statuses.

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

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

Now we'll need to update those states when the status of our request changes.

Add a `useEffect` call:

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

```
    React.useEffect(() => {
      const fetchInitialState = async () => {
        try {
          const data = await load()
          setInitialState(data)
        } catch (e) {
          setError(e)
        }
        setIsLoading(false)
      }
      fetchInitialState()
    }, [])
```

Here we call the `useEffect` hook. We an empty array to it as a second argument so that it triggers the callback only when our component mounts.

> Read more about `useEffect` hook in React Documentation[48]

Inside of our `useEffect` callback, we defined the `fetchInitialState` asynchronous function. We did it so that we could use the `async/await` syntax.

Inside of 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/withData.tsx

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

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

    return (
      <WrappedComponent initialState={initialState}>
        {children}
      </WrappedComponent>
    )
```

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/AppStateContext.tsx.`:

[48]https://reactjs.org/docs/hooks-effect.html

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

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

And wrap the `AppStateContext` into it.

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

```
export const AppStateProvider = withData(({ children, initialState }: R\
eact.PropsWithChildren<{initialState: AppState}>) => {
  const [state, dispatch] = useReducer(appStateReducer, initialState)

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

  return (
    <AppStateContext.Provider
      value={{ state, 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.

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 React testing library[49] because it has simple API, is easy to set up and is recommended by React team. Oh, and of course it supports Typescript.

How to test a front-end application isn't always obvious, 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.

The 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
```

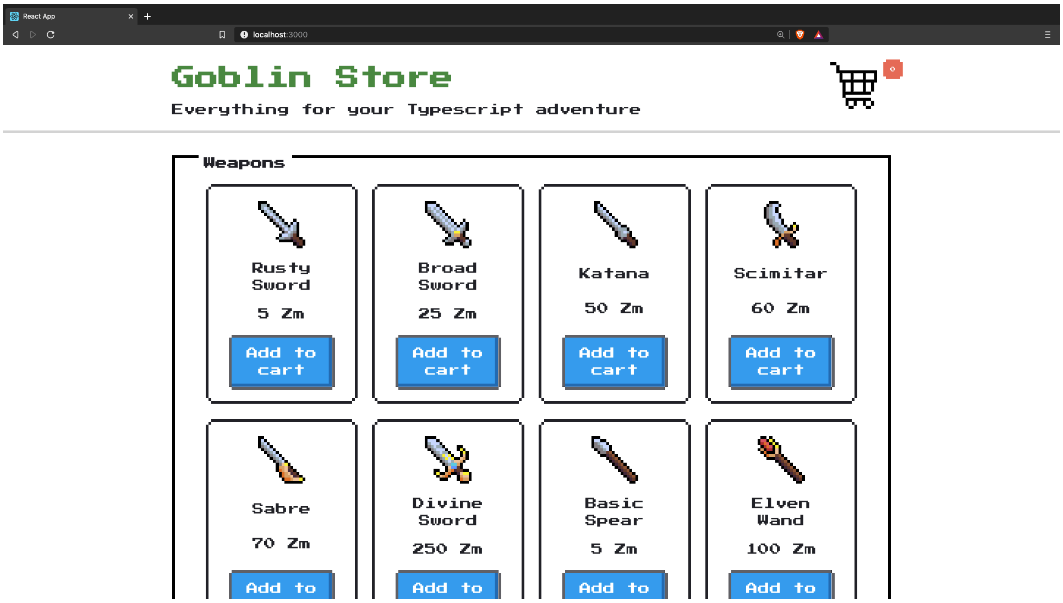
When you are there - install the dependencies and launch the app:

[49] 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[50] to launch two scripts at the same time. You can check `src/package.json` to see how we do it.

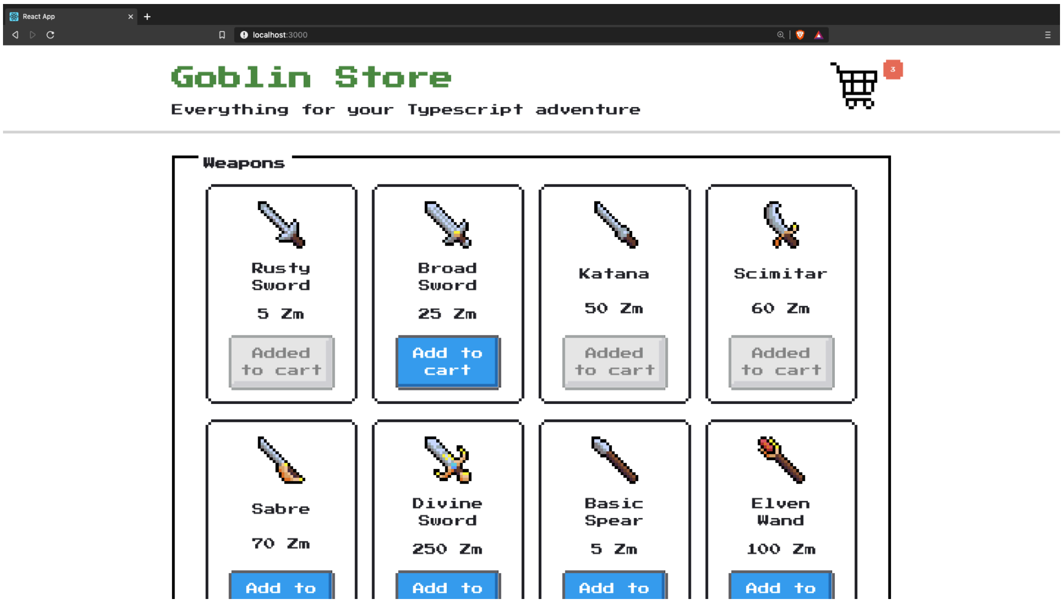
It should also open the app in the browser. If it didn't happen - navigate to `http://localhost:3000` and open it manually.



Main screen

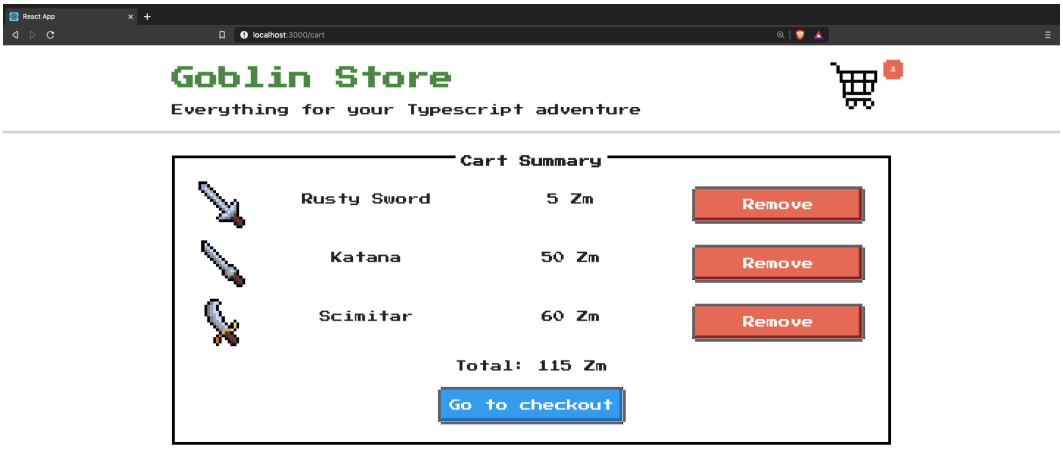
You should see a list of hero equipment: weapons, armor, potions. Click the **Add to cart** buttons to add items to the cart.

[50]https://www.npmjs.com/package/concurrently



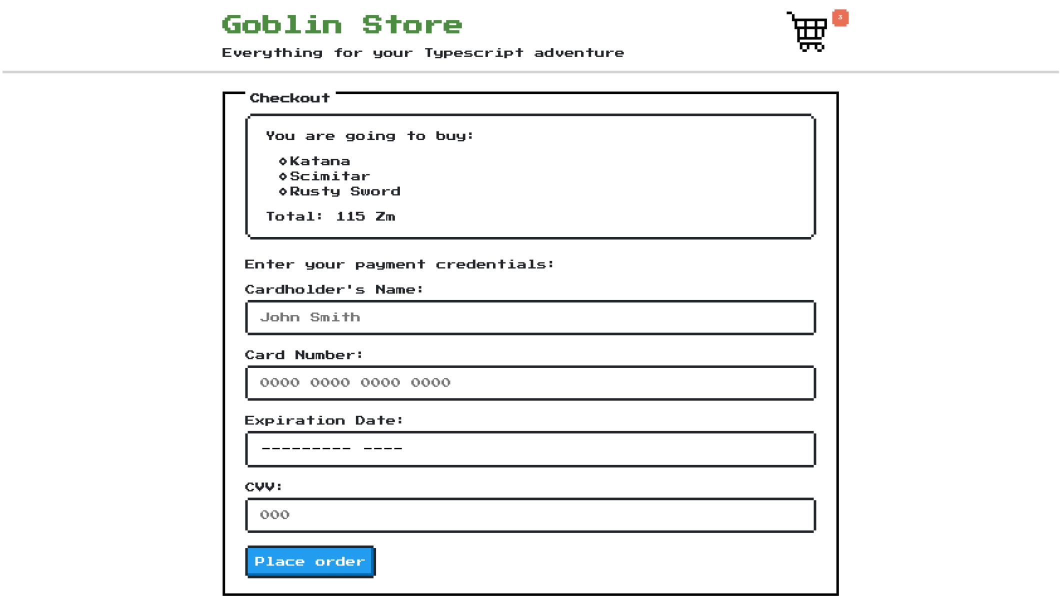
Selected items

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 *Cart Summary* page. Here you can review the cart and remove the items if you don't want to buy them anymore. Click the **Go to checkout** button.


Goblin Store
Everything for your Typescript adventure

Checkout

You are going to buy:

- Katana
- Scimitar
- Rusty Sword

Total: 115 Zm

Enter your payment credentials:

Cardholder's Name:
John Smith

Card Number:
0000 0000 0000 0000

Expiration Date:
--------- ----

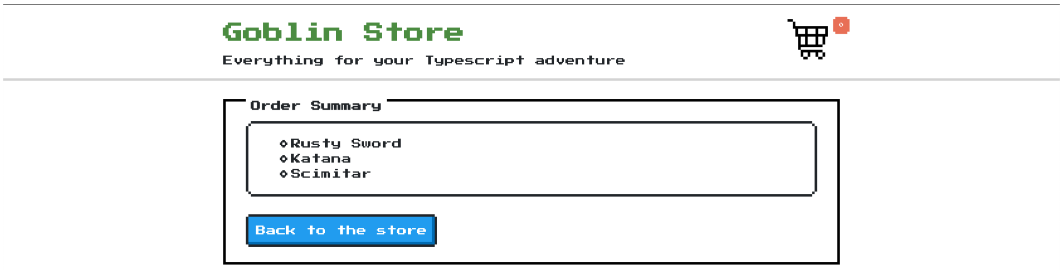
CVV:
000

Selected items

Now you are on *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 see the checkout form. Fill in the fields. If you try to skip the fields or input the 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.


Goblin Store
Everything for your Typescript adventure

Order Summary

- Rusty Sword
- Katana
- Scimitar

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 it's 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:

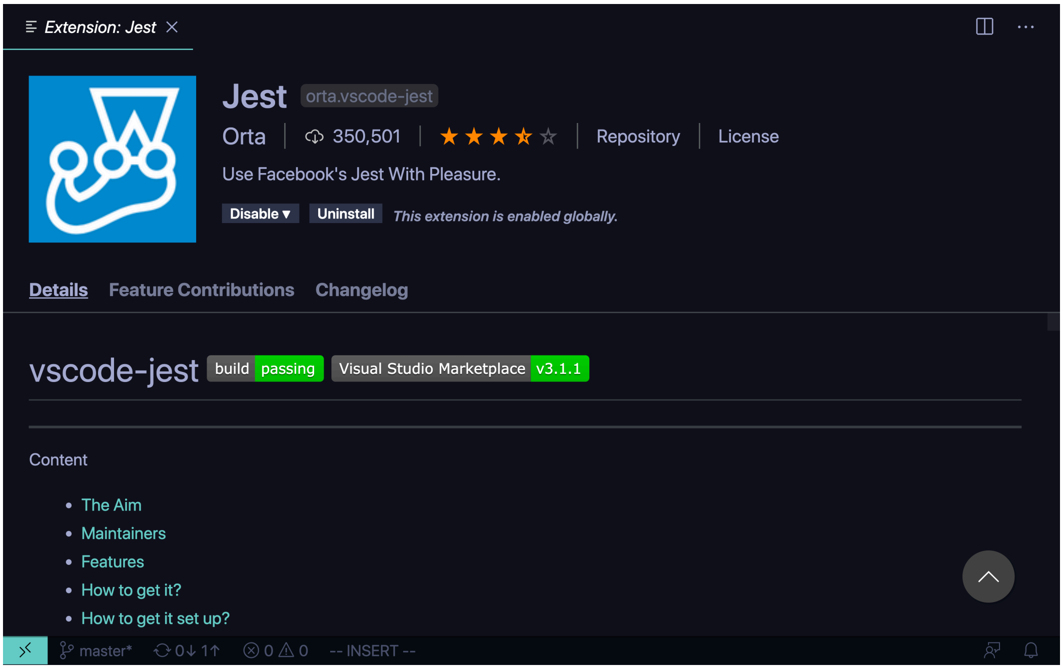
```
yarn && yarn test
```

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

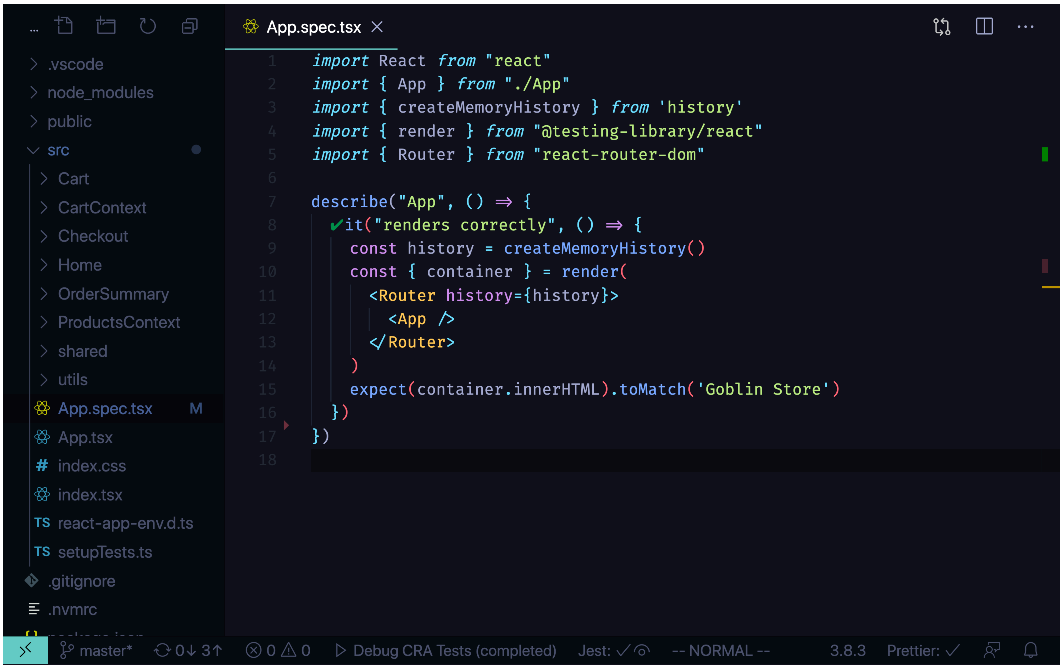
If you are using VSCode - you can install a useful Jest plugin[51] that automatically runs the tests and displays the test results right in the text editor.

[51]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:

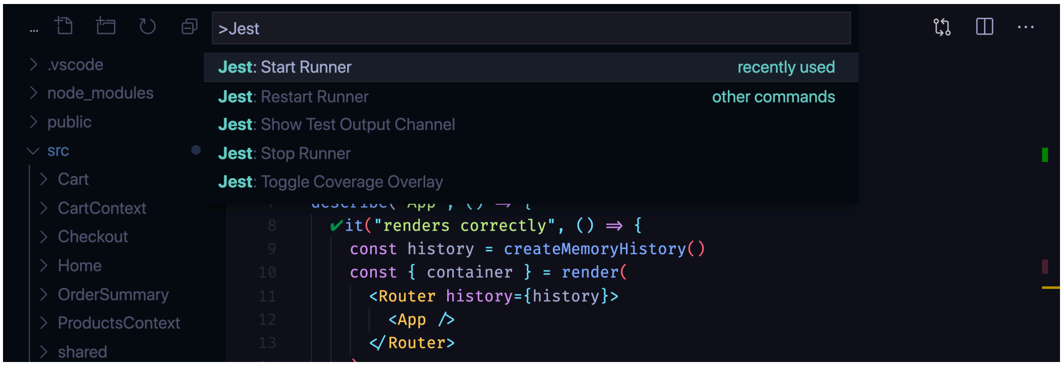


```
import React from "react"
import { App } from "./App"
import { createMemoryHistory } from 'history'
import { render } from "@testing-library/react"
import { Router } from "react-router-dom"

describe("App", () => {
  it("renders correctly", () => {
    const history = createMemoryHistory()
    const { container } = render(
      <Router history={history}>
        <App />
      </Router>
    )
    expect(container.innerHTML).toMatch('Goblin Store')
  })
})
```

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
Jest: Start Runner recently used
Jest: Restart Runner other commands
Jest: Show Test Output Channel
Jest: Stop Runner
Jest: Toggle Coverage Overlay

Jest VSCode plugin

Troubleshooting

If your VSCode Jest plugin doesn't seem to work, check the "Output" console on 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 here[52]

Enable Debugging Tests

Before we begin there is one more thing that is good to know. How to 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",
```

[52] 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 an equivalent of running the following in your terminal:

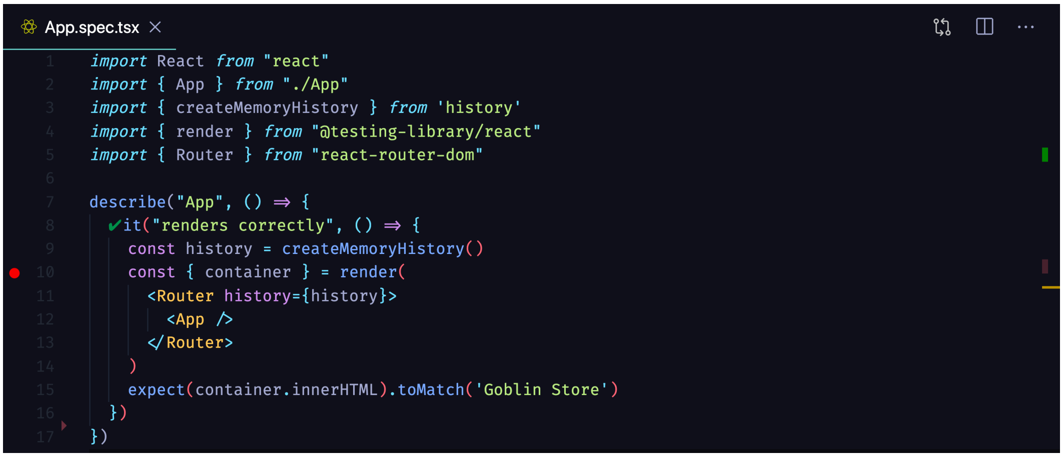
```
yarn test --runInBand --no-cache --watchAll=false
```

- `--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 of 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

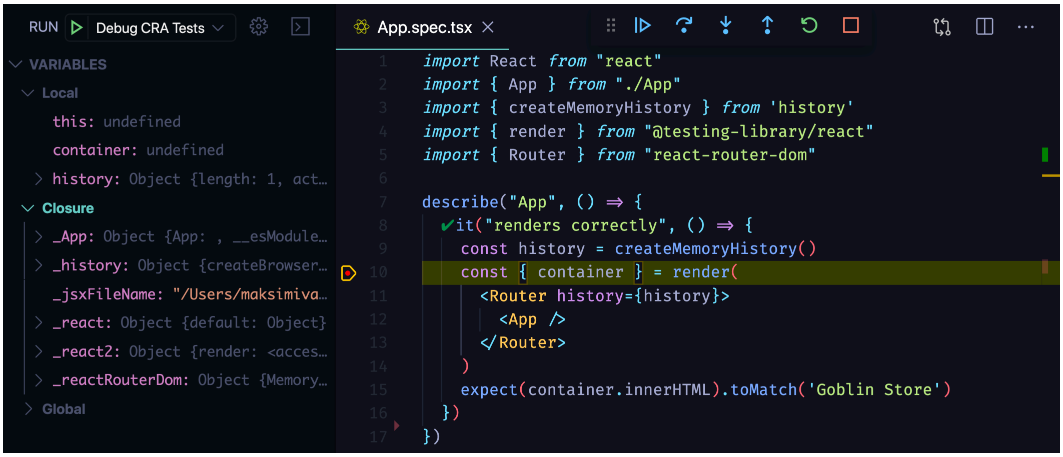
Let's verify our debugging configuration. Open `src/App.spec.tsx` and place a breakpoint:



```
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 correctly", () => {
    const history = createMemoryHistory()
    const { container } = render(
      <Router history={history}>
        <App />
      </Router>
    )
    expect(container.innerHTML).toMatch('Goblin Store')
  })
})
```

Jest VSCode plugin

Now open the `Command Palette` (`View -> Command Palette`) and select `Debug: Select and Start Debugging` and the `Debug CRA Tests`.



```
VARIABLES
  Local
    this: undefined
    container: undefined
  > history: Object {length: 1, act…
  Closure
  > _App: Object {App: , __esModule…
  > _history: Object {createBrowser…
    _jsxFileName: "/Users/maksimiva…
  > _react: Object {default: Object}
  > _react2: Object {render: <acces…
  > _reactRouterDom: Object {Memory…
  Global
```

```
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 correctly", () => {
    const history = createMemoryHistory()
    const { container } = render(
      <Router history={history}>
        <App />
      </Router>
    )
    expect(container.innerHTML).toMatch('Goblin Store')
  })
})
```

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 tests execution step by step and observe the values of all the variables in your code. We will use this functionality later in this chapter, 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 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 path="/checkout">
            <Checkout />
          </Route>
          <Route path="/cart">
            <Cart />
          </Route>
          <Route path="/order">
            <OrderSummary />
          </Route>
          <Route path="/">
```

```
            <Home />
          </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 test that verifies that App component at least renders successfully. Open src/App.spec.tsx and add the following code:

02-testing/step1/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 of 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 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/step1/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/step1/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[53]. It means that some functionality can be missing or work differently from the browser environment.

One of the missing things is a History API[54]. So to use routing we'll have to install an additional package that will provide us the History API functionality.

Install `history` as dev dependency:

[53]https://www.npmjs.com/package/jsdom
[54]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/step1/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 are 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[55] and get the `container` instance. Container represents the containing DOM node of the rendered React component.

Expectation. We call the `expect` method provided by Jest[56]. 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 the `src/App.spec.tsx` import the `Home` component and then call `jest.mock` to mock this module:

02-testing/step1/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 by a fake double that mimics its behavior. You can also spy on mocked

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

objects and functions to track how your code is using them. But we'll get back to it 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 component was rendered.

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

02-testing/step1/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 to not 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/step1/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 `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 routes 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/step1/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 `history` object.

We've defined the `renderWithRouter` function on the `global` object. The `global` object is a global namespace object in node[57].

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 it.

First define the type for our function:

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

02-testing/step1/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 it's `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/step1/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 Typescript documentation[58].

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:

[58]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` in 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[59] type as well. It is a variable that refers to the global scope.

02-testing/step1/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 />, "/")
```

Now let's make it available in our test files. Go to `src/setupTests.ts` and import the `src/testHelpers.tsx`:

02-testing/step1/src/setupTests.ts

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

Writing The Tests

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

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

02-testing/step1/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 name `routing` and move our root route test there. Remake it so that it uses `renderWithRouter`:

02-testing/step1/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/step1/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 alse render correctly:

02-testing/step1/src/App.spec.tsx

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

Here we check that with some 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

`Header` component renders the title of the store and also the cart widget. Cart widget is defined in a separate component, so we'll mock it and test `Header` in isolation.

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

02-testing/step1/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/step1/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/step1/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 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 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/step1/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[60] 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 `link` role. You can find complete list of default `aria-role` values on WHATWG site[61].

So in our test, we click the link element and then check if we ended up on the `/cart` route.

Now let's test that `CartWidget` renders the number of products in the cart correctly.

`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` though the `useCartContext` hook.

Open the `CartWidget` component code. It's located in `src/shared/CartWidget/CartWidget.tsx`:

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

02-testing/step1/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/step1/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 - 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/step1/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/step1/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/step1/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

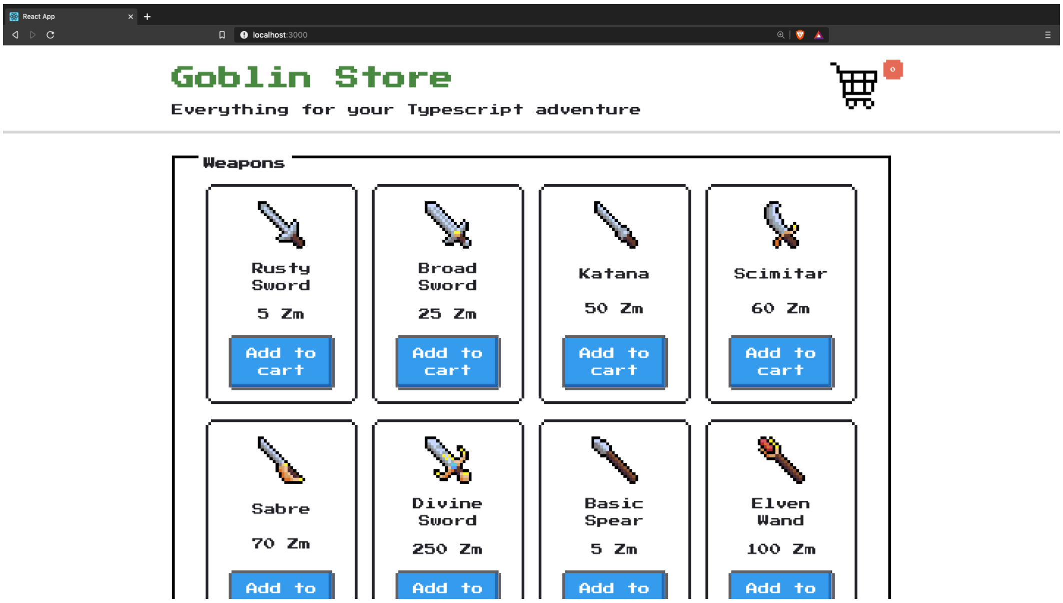
Our `Loader` component does not contain any logic. In our test we'll only make sure that it renders correctly:

02-testing/step1/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 here. 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'll decide to reuse it we'll have to move it to `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 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 got 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:

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 got 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 `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 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 it in the `ProductCart` component tests.

ProductCart Component

Moving on to the `ProductCard` component. Let's see what do 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/step1/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 correctly:

- 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 `addToCart` function:

02-testing/step1/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 `Home` component tests.

Now let's test how our component works when it's product is not in the cart, add this code to "when product is not in the cart" `describe` block:

02-testing/step1/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:

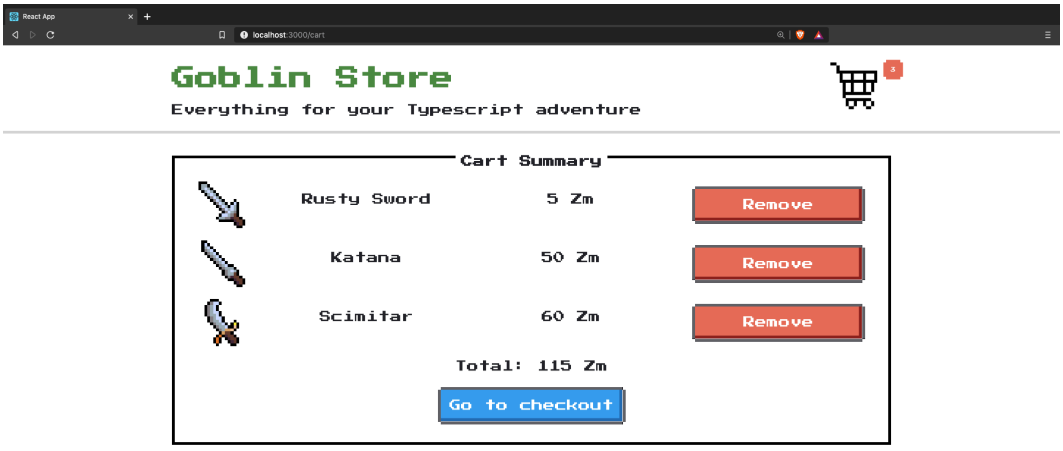
We fire the `click` event on our button and then we check that the `addToCart` function was called with the product data.

We are done testing the `Home` page components. We'll test the `useProducts` hook later, for now, move on to other pages.

We'll continue with the `Cart` page.

Cart Page

This page renders the list of items that you've added to cart.



Rusty Sword	5 Zm	Remove
Katana	50 Zm	Remove
Scimitar	60 Zm	Remove

Total: 115 Zm

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 anymore.

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'", () => {
      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"

describe("Cart", () => {
  describe("without products", () => {
    it.todo("renders empty cart message")
  })

  describe("with products", () => {
    it.todo("renders cart products list")
    describe("on 'go to checkout' click", () => {
      it.todo("redirects to '/checkout'")
    })
  })
})
```

Now let's check that if we click the link - we'll get redirected to main page. First lets hardcode the cart value with empty products array inside the `without products` block:

02-testing/step1/src/Cart/Cart.spec.tsx

```
    const stubCartHook = () => ({
      products: [],
      removeFromCart: () => {},
      totalPrice: () => 0
    })
```

Still inside products block write the test that will check that our component will render `Your cart is empty` message:

02-testing/step1/src/Cart/Cart.spec.tsx

```
    it("renders empty cart message", () => {
      const { container } = renderWithRouter(() => (
        <Cart useCartHook={stubCartHook} />
      ))
      expect(container.innerHTML).toMatch(
        "Your cart is empty."
      )
    })
```

Time to check that if we click the `Back to main page` button we'll 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/step1/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 `Back to main page` text on it, click it and then verify that ended up on root route.

Now let's verify that cart with products also renders correctly. Inside the `with products` block hardcode an array of products:

02-testing/step1/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/step1/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 in the beginning of our test file:

02-testing/step1/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/step1/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/step1/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 removing the product from the cart. So 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/step1/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/step1/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/step1/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 it's `alt` attribute and it has correct `src`.

Let's move on and test that when user clicks the `Remove` button we call the function passed through the `removeFromCart` prop. Add this code inside the `on 'Remove' click` block:

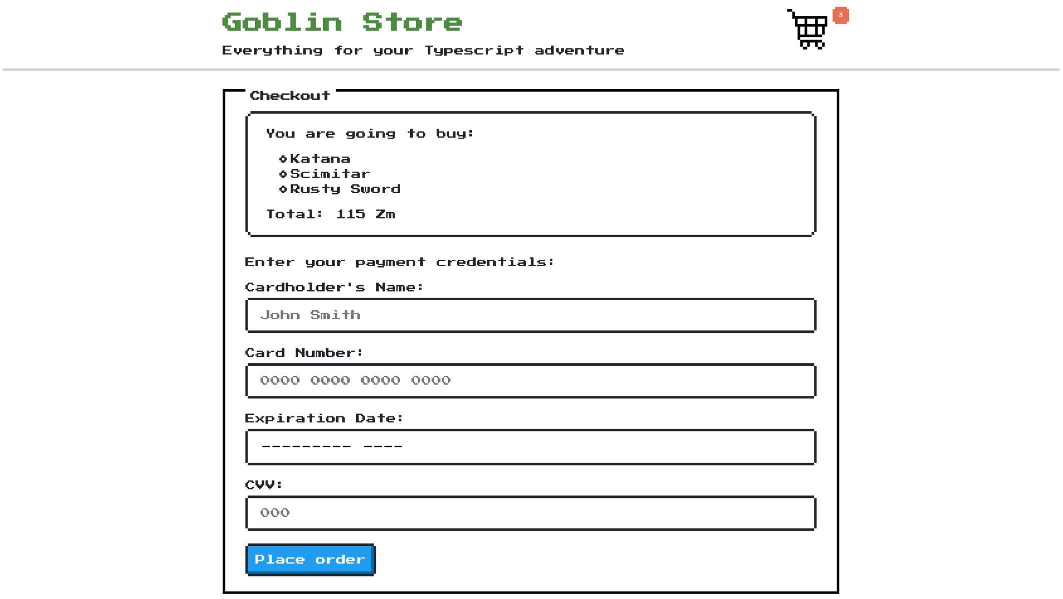
02-testing/step1/src/Cart/CartItem.spec.tsx

```
    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 the payment credentials and perform the order.



Goblin Store

Everything for your Typescript adventure

Checkout

You are going to buy:

- Katana
- Scimitar
- Rusty Sword

Total: 115 Zm

Enter your payment credentials:

Cardholder's Name: John Smith

Card Number: 0000 0000 0000 0000

Expiration Date: --------- ----

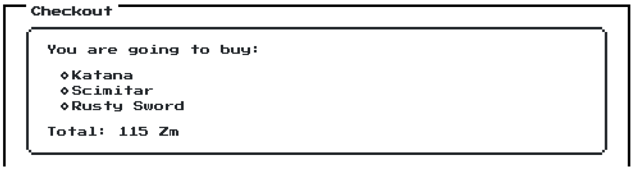
CVV: 000

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

You are going to buy:

- Katana
- Scimitar
- Rusty Sword

Total: 115 Zm

Checkout list

This component also uses `CartContext` through the `useCart` hook.

It has one task, so it better does 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:

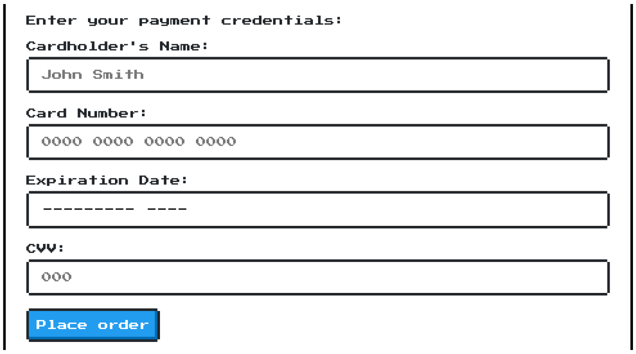
02-testing/completed/src/Checkout/CheckoutList.spec.tsx

```
  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 messages
- When the input values are valid
  - When you click the `Order` button
    * The submit function is called

Create a the test file with the following contents:

02-testing/step1/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 `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[62] to manage the inputs. It means that the inputs are controlled[63] 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[64].

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:"), {
        target: { value: "123" }
      })
    })
```

And then click the `Place order` button. Technically we could put it into the same `act` block, but I decided that it is more clear that first we create specific conditions and then we perform an action:

[62]https://react-hook-form.com/
[63]https://reactjs.org/docs/forms.html#controlled-components
[64]https://reactjs.org/docs/test-utils.html#act

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/step1/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/step1/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, it has the correct `name` value and doesn’t have 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 ou `FormField` - it will render the error message:

02-testing/step1/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:

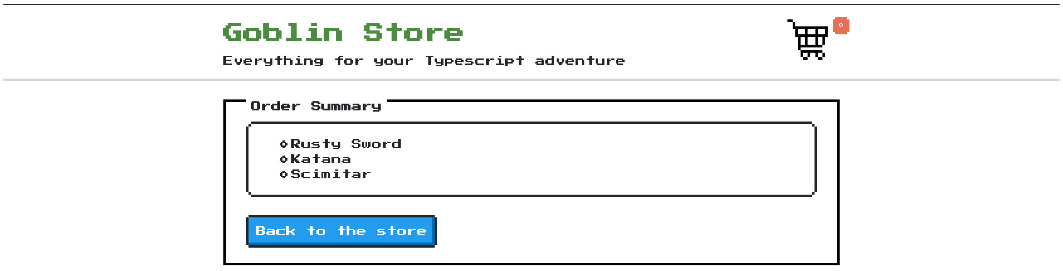
02-testing/step1/src/Checkout/FormField.spec.tsx

```
    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 `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.



◇Rusty Sword
◇Katana
◇Scimitar

Order summary

It gets the `orderId` from the current location query params. And makes a request to backend using the `api` module.

02-testing/step1/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 loading state we'll render `Loader`. First, let's mock the `Loader` component.

02-testing/step1/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/step1/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 order is loaded successfully we render the products list from it. Hardcode the `useOrder` hook inside the `when order is loaded` block:

02-testing/step1/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/step1/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 renaed a link to the main page. Let's write a test for it as well:

02-testing/step1/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 lest's test that if the order data could not be loaded we render a failure message:

02-testing/step1/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. Time to test the hooks.

Testing React Hooks

Let's go back to our `Home` page and test how do 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/step1/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 `useProducts` hook will start fetching data when it is mounted:

02-testing/step1/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/step1/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/step1/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/step1/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/step1/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 next update before we can test our expectations:

02-testing/step1/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/step1/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 correct value:

02-testing/step1/src/Home/useProducts.spec.ts

```
      const { result, waitForNextUpdate } = renderHook(() =>
        useProducts(mockApiGetProducts)
      )

      await act(() => waitForNextUpdate())
```

And finally we check our expectations:

02-testing/step1/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/step1/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[65] where function tests are called with pound sign prefix: `#functionName`.

Let's go one by one, first, we need to make sure that when this hook is mounted it loads the data from the `localStorage`. Let's start by mocking the `localStorage`:

Define the `localStorage` constant:

02-testing/step1/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 `window` object using `Object.assign` method:

[65] https://rspec.rubystyle.guide/

02-testing/step1/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/step1/src/CartContext/useCart.spec.ts

```
  afterEach(() => {
    localStorageMock.clear()
  })
```

Now we are ready to test that our hook will load its initial state from `localStorage`:

02-testing/step1/src/CartContext/useCart.spec.ts

```
    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/step1/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 `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 method 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/step1/src/CartContext/useCart.spec.ts

```
      const product: Product = {
        name: "Product foo",
        price: 0,
        image: "image.jpg"
      }
      localStorageMock.setItem(
        "products",
        JSON.stringify([product])
      )
```

Next render our hook:

02-testing/step1/src/CartContext/useCart.spec.ts

```
      const { result } = renderHook(useCart)
```

Now call the `removeFromCart` method. Remember to wrap this call into `act` because it alters the state of the hook:

02-testing/step1/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/step1/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/step1/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 `totalPrice` function is forty-two.

The last method we'll test is `clearCart`.

02-testing/step1/src/CartContext/useCart.spec.ts

```
  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 then check that the cart is empty.

Congratulations

If you read up until 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

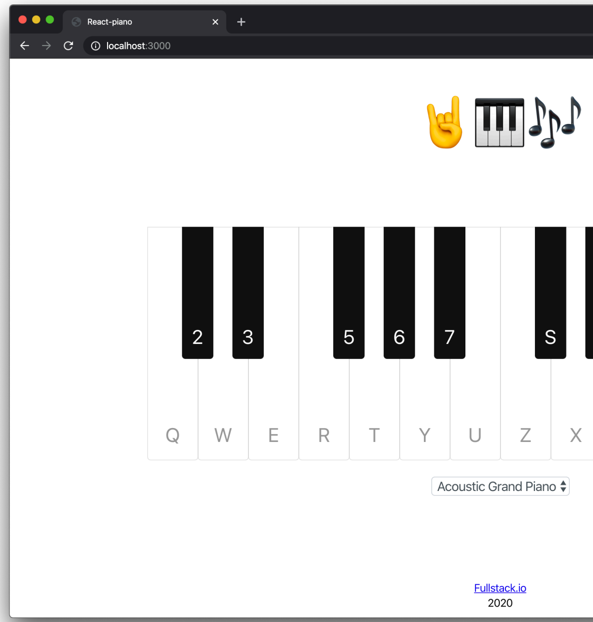
Particularly, we will talk about React-specific patterns such as *Render-Props* and *Higher Order Component*, and how they are connected to a more general concepts.

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 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 didn't, navigate to http://localhost:3000[66] and open it manually.

[66]http://localhost:3000

In the browser, at the center of the screen, you will see a keyboard with letter labels on each key and a `select` under with a default instrument.

Go ahead and try it out! You will hear the musical notes played on an acoustic grand piano. Let's build it!

First Steps and Basic Application Layout

First, let's 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 `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 `logo.svg` import along with the corresponding file—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 React, { FunctionComponent } from "react"
import "./style.css"

export const Footer: FunctionComponent = () => {
  const currentYear = new Date().getFullYear()

  return (
    <footer className="footer">
      <a href="https://fullstack.io">Fullstack.io</a>
      <br />
      {currentYear}
    </footer>
  )
}
```

Notice that our component imports a stylesheet, so let's create a file called `style.css` beside out `Footer.tsx` and fill it up with these styles.

03-react-piano/step-1/src/components/Footer/style.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 2nd line of stylesheet: there we declare that component's height should be equal to a value of a custom property[67] (a.k.a CSS variable).

In CSS `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 that 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;
```

A visibility scope of our variable is `:root`. That means that our variable is visible across all the elements on a page. We could also define it in some selector, so that it would be hidden from other element. For 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:

[67]https://developer.mozilla.org/en-US/docs/Web/CSS/--*

03-react-piano/step-1/src/components/Logo/Logo.tsx

```
import React, { FunctionComponent } from "react"
import "./style.css"

export const Logo: FunctionComponent = () => {
  return (
    <h1 className="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 emoji 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 symbol in a `span` with a `role="image"` attribute. It will help screen readers to correctly parse the content of our app. Then, we create a stylesheet for our `Logo` component.

03-react-piano/step-1/src/components/Logo/style.css

```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`[68]. This is a relative unit, that refers to the value of `font-size` property on `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 single `font-size` value on `html` element instead.

When created `Footer` and `Logo` and styles for them, we're going to import and render it in an `App.tsx`, so that it will look like this:

03-react-piano/step-1/src/App.tsx

```tsx
import React from "react"
import { Footer } from "./components/Footer"
import { Logo } from "./components/Logo"
import "./App.css"

export const App = () => {
  return (
    <div className="app">
      <Logo />
      <main className="app-content" />
      <Footer />
    </div>
```

[68]https://developer.mozilla.org/en-US/docs/Web/CSS/font-size

```
  )
}
```

Now, let's finish with global styles which will be applied to the whole project:

03-react-piano/step-1/src/index.css

```css
*,
*::after,
*::before {
  box-sizing: border-box;
}
```

Here we define `box-sizing: border-box` to every element on a page. It will help us to calculate elements' geometry more easily. Also, we declare that page should have height at least 100% of a screen height. Since our keyboard will be placed in the center of a screen it will be convenient to do that.

And finally, let's style our `App` component to ensure that `Footer` will be placed at the bottom of the page and `Logo` component—at the top.

03-react-piano/step-1/src/App.css

```css
.app {
  min-height: 100vh;
}

.app-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 a page but without `Footer` and `Logo` components' heights. It ensures that content area is at least a size of a 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 in which way we want to represent musical notes in our application. Nowadays it is considered standard to use MIDI Notes Numbers[69] for that.

Long story short, MIDI Note Number is a number that represents a given note in range from -1st octave to 9th. Octave is a set of 12 semitones that are different from each other by half of a tone (hence semitone).

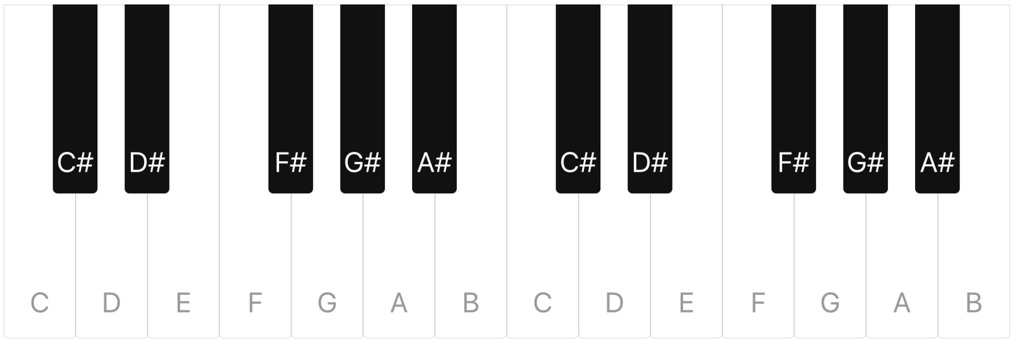
Notes in an octave start from C and go up to B like this:

```
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 sake we will focus on and use sharps. 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 that 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.

[69]http://www.flutopedia.com/octave_notation.htm



Notes location on a musical keyboard

Coding Music Rules

With all that said let's try to formalize these rules and express them in TS:

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, let's talk about domain. In software domain[70] is a target subject of a program. This term has roots in domain driven design[71]—the concept of how to structure applications.

In our case domain refers to sound, notes generation, notes notation and real keyboard layout.

Inside of domain directory we create a file called `note.ts`—here we describe everything about notes that we want to express in TypeScript.

For example inside we create new custom union 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 among. In our case `NoteType` is a set of possible notes types like: natural, sharp or flat. Despite the fact that we're going

[70]https://en.wikipedia.org/wiki/Domain_(software_engineering)
[71]https://en.wikipedia.org/wiki/Domain-driven_design

to use only sharps it is a good practice to keep union types as full as possible to make it clear what can be used in general.

Then, `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 signatures of our future functions more clear.

03-react-piano/step-2/src/domain/note.ts

```
export type MidiValue = number
export type PitchIndex = number
```

Here, we describe a `MidiValue` type which is basically a number from Octave Notation above. And a `PitchIndex` which is also a number and represents an 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.

Interface of a Note

We're going to create an interface of our `Note` entity.

Firstly, what is an interface? An interface[72] is an abstract description of some entity, in our case of an object. It is a shared boundary across which two or more separate components of a computer system exchange information.

In TypeScript, interfaces fill the role of naming custom types[73], and are a powerful way of defining contracts within our code as well as contracts with code outside of our project.

They are a powerful tool to make code components less dependent on each other and make our code reusable and less error prone.

So let's go ahead and create our `Note` interface:

[72]https://en.wikipedia.org/wiki/Interface_(computing)
[73]https://www.typescriptlang.org/docs/handbook/interfaces.html

03-react-piano/step-2/src/domain/note.ts

```
export interface Note {
  midi: MidiValue
  type: NoteType

  pitch: NotePitch
  index: PitchIndex
  octave: OctaveIndex
}
```

We describe a shape of a note object which is going to be used later in our code. A `Note` contains 5 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 field `type` accepts values with type of `NoteType`. That means that the value for field `type` can only be one of those described earlier in `NodeType`. So we can only assign `"natural"`, `"sharp"` or `"flat"` to field `type` and nothing more.

If we try to do that TS type checker will tell us that:

```
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 of 1st octave. It has a `MidiValue` of 24, we will save it in `C1_MIDI_NUMBER` constant to use later.

Similarly we create constraints for our keyboard range. A start note will be `C4_MIDI_NUMBER` and finish note — `B5_MIDI_NUMBER`. Also we're going to need count of halfsteps in an octave so we keep it in `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
```

And now, we can create some kind of a 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 an `PitchIndex` as a key and `NotePitch` as a value.

You may notice that its type is `Record<PitchIndex, NotePitch>`. Types with "arguments" like this one are called generics[74]. Those are types which allow us to

[74]https://www.typescriptlang.org/docs/handbook/generics.html

create a program component that can work over a variety of types rather than a single one.

`Record<K, T>` type constructs[75] 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`.

We will cover generics in more detail later when creating our own.

Notes Generation

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 of. Finally we define what `type` this note of, and return a created note object.

This function 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.

[75] https://www.typescriptlang.org/docs/handbook/utility-types.html#recordkt

03-react-piano/step-2/src/domain/note.ts

```
interface NotesGeneratorSettings {
  fromNote?: MidiValue
  toNote?: MidiValue
}

export function generateNotes({
  fromNote = LOWER_NOTE,
  toNote = HIGHER_NOTE
}: NotesGeneratorSettings = {}): Note[] {
  return Array(toNote - fromNote + 1)
    .fill(0)
    .map((_, index: number) => fromMidi(fromNote + index))
}

export const notes = generateNotes()
```

Here, we create a `generateNotes()` function which takes a settings object of type `NotesGeneratorSettings`. It describes what settings we can use in our function to generate notes. Question mark (`?`) at 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 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 an access to `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 note in 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 some third-party API to create a sound. So let's talk a bit about integration of those APIs.

Web Audio API

For starters let's figure out what's required to create a sound in a browser in a first place. Modern web browsers have support `Audio API`[76].

It uses an `AudioContext` which allows us to handle audio operations such as playing musical tracks, creating oscillators etc. This `AudioContext`[77] has nothing to do with `React.Context` that we saw earlier. Those only have similar names, but `AudioContext` we're going to use is an interface that provides access to 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 an assumption that user's browser has it.

So we have to ensure that 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`:

[76]https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API
[77]https://developer.mozilla.org/en-US/docs/Web/API/AudioContext

03-react-piano/step-2/src/domain/types.ts

```
export type Optional<TEntity> = TEntity | null
```

Our `Optional` type is a generic type, which tells that it represents a union of a given type `TEntity` or a `null`. Basically we're building an "assumption" type, and 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 bit more verbose one—we use `TEntity` instead of `T`. This is not mandatory. We will use this only for readability sake, because later on, when we will be building complex interfaces and generic functions, we will be needing a way to describe what our type arguments are and what they are for.

So, this `Optional` type is useful when we need to make sure that we cover all the possible cases when some 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 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, this is a constructor function. To make TypeScript understand what type we want to declare we explicitly define it as `typeof AudioContext`.

Below, we can see an extension for `Window` interface, which includes field `webkitAudioContext` with a type of `AudioContextType`. This is required for now because TypeScript by default doesn't include[78] some vendor properties and methods on `window`.

So we have to extend standard window interface to get an access to this field because in some browsers `AudioContext` is accessible via `AudioContext` property and in some—via `webkitAudioContext`.

And 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` otherwise. If a browser doesn't support any of them then we want to return `null`, just to be able to determine later that we cannot access `Audio API`.

Soundfont

Next, it is time to introduce a third party API which we're going to use — Soundfont[79]. It is a framework agnostic loader and player which has a pack of pre-rendered sounds of many instruments. And also it comes with typings for integration with TypeScript project!

We prefer Soundfont over MIDI.js[80] 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 `DEFAULT_INSTRUMENT` constant of type `InstrumentName` which comes with `soundfont-player` package. One of the coolest things about integration third-party APIs which have TypeScript declarations is that we can use

[78]https://github.com/microsoft/TypeScript/issues/31686
[79]https://www.npmjs.com/package/soundfont-player
[80]https://github.com/mudcube/MIDI.js

our IDE's autocomplete to scroll through possible options for union types. Here, we can select across 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. A good news is that many of those problems are solved for us long time ago. The solutions for typical software development problems are called *patterns*.

Adapter or Provider Pattern

An Adapter[81] pattern (or sometimes 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*[82] (or *provides*) third party API to us and make it usable in our application code.

React-Specific Patterns

In our case we want to use Provider pattern to make Soundfont's functionality accessible for our application. Also, it will be useful to connect `Audio API` to our code.

Using React we can implement Provider pattern using multiple techniques, such as Render-Props and Higher Order Components. Those are also called patterns, however to distinguish these from 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.

[81]https://en.wikipedia.org/wiki/Adapter_pattern
[82](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 on 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 `Main` component.

03-react-piano/step-3/src/components/Main/Main.tsx

```
import React, { FunctionComponent } from "react"
import { Keyboard } from "../Keyboard"
import { NoAudioMessage } from "../NoAudioMessage"
import { useAudioContext } from "../AudioContextProvider"

export const Main: FunctionComponent = () => {
  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[83]—`useAudioContext()`.

Custom Hook for Accessing Audio

Intentionally Hooks in React let us use state and other features without writing a class. Writings hooks has its rules[84] and limitations. For example all hooks' names should start with `use*` prefix. It allows linter to check if a hook's source code satisfies all the limitations, which are:

[83]https://reactjs.org/docs/hooks-intro.html
[84]https://reactjs.org/docs/hooks-rules.html

- We can call hooks only at the top level of our components, and never — conditionally.
- We can call hooks only inside of 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 `useRef()` hook[85] to "remember" the value that our `accessContext()` function is going to return.

The `useState()` hook is a hook which allows us to create a local state in functional component. It returns a tuple of a value and an update-function. Since we don't need an update function, we only use the value from the local state.

As an argument `useState()` takes an initial value for the state. We can also pass a function, since `useState()` checks the arguments and if a given argument is a function, `useState()` will call it automatically and will keep the returned result as a state value.

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 `NoAudioMessage` component otherwise. Now it's time to look at both of them.

[85]https://reactjs.org/docs/hooks-reference.html#useref

Handling Missing Audio Context

Let’s look at the `NoAudioMessage` component first. It is basically a `div` with some text in it. It doesn’t do much, only renders a message for a user.

03-react-piano/step-3/src/components/NoAudioMessage/NoAudioMessage.tsx

```
import React, { FunctionComponent } from "react"

export const NoAudioMessage: FunctionComponent = () => {
  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` components however is a bit more interesting.

03-react-piano/step-3/src/components/Keyboard/Keyboard.tsx

```
import React, { FunctionComponent } from "react"
import { OctavesRange, selectKey } from "../../domain/keyboard"
import { notes } from "../../domain/note"
import { Key } from "../Key"
import "./style.css"

export const Keyboard: FunctionComponent = () => {
  return (
    <div className="keyboard">
```

```
      {notes.map(({ midi, type, index, octave }) => {
        const label = selectKey(octave as OctavesRange, index)
        return <Key key={midi} type={type} label={label} />
      })}
    </div>
  )
}
```

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 into `midi`, `type`, `index`, and `octave`. For each note we render a `Key` component, we will look at it 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 type OctavesRange = Extract<OctaveIndex, 4 | 5>

export const TOP_ROW: Keys = Array.from("q2w3er5t6y7u")
export const BOTTOM_ROW: Keys = Array.from("zsxdcvgbhnjm")

export function selectKey(
  octave: OctavesRange,
  index: PitchIndex
): Key {
  const keysRow = octave < 5 ? TOP_ROW : BOTTOM_ROW
  return keysRow[index]
}
```

In `keyboard.ts` we create 3 custom types:

- `Key`, a type-alias for representing letter key label
- `Keys`, an array of those labels
- and an `OctavesRange`, a type which uses `Extract` utility type.

`Extract<T, U>` constructs[86] a type by extracting from type `T` all properties that are assignable to type `U`. Thus, it makes an `OctavesRange` to be a type that contains only octaves that can fit into our not very wide keyboard.

By design, `OctavesRange` is possible to fit only `2` octaves. It sets up a constraint on what octaves can be used here. In our case there are only `4`-th and `5`-th. `Extract` takes a union (`OctaveIndex`) and keeps only a set of values given as a second argument.

Then, we create `2` arrays of letters, that will label our keys. If those letters are pressed on a real keyboard we will play a sound of a key with corresponding label. We use `Array.from()`[87] to create an array of characters from a string.

And finally, `selectKey()` is a function which takes an octave index that we are choosing a key for and a pitch index to select among 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.

03-react-piano/step-3/src/components/Key/Key.tsx

```
import clsx from "clsx"
```

First of all, we want to use `clsx` package[88] to compose a component's `className` with others in the future.

[86]https://www.typescriptlang.org/docs/handbook/utility-types.html#extracttu
[87]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/from
[88]https://www.npmjs.com/package/clsx

03-react-piano/step-3/src/components/Key/Key.tsx

```
interface KeyProps {
  type: NoteType
  label: string
  disabled?: boolean
}

export const Key: FunctionComponent<KeyProps> = (props) => {
  const { type, label, ...rest } = props
  return (
    <button
      className={clsx(`key key--${type}`)}
      type="button"
      {...rest}
    >
      {label}
    </button>
  )
}
```

Then, let's pay attention to a type definition of a component — it is a `FunctionComponent<KeyProp`

We could write this without `FunctionComponent` at all, it wouldn't be a mistake:

```
export const Key = ({ type, label, ...rest }: KeyProps) => /*...*/
```

However, let's try to use `FunctionComponent` as well. First of all, `FunctionComponent` is a generic type[89] from 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 IDE, because when IDE knows what props a component can have it can help us with suggestions of what we can or must provide when using it.

In our case these argument-props are described with an interface `KeyProps`. Inside we define:

[89] https://www.typescriptlang.org/docs/handbook/jsx.html#function-component

- `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
- and `disabled`, an optional `boolean`—if `true` it will disable the key from pressing.

Also we want to use `clsx` package[90] to compose a component's `className` with others in the future.

As a base for our component we use `button` element. To ensure that all the browsers render our keys more or less equally we want to reset default button styles.

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;

  -webkit-font-smoothing: inherit;
  -moz-osx-font-smoothing: inherit;
}
```

Here, we drop default styles and make button look like a text item.

Then, in styles for `Key` component we describe how keys should look like. The whole stylesheet can be found in `src/components/Key/style.css`, here, we focus only on the difference between black and white keys.

[90]https://www.npmjs.com/package/clsx

03-react-piano/step-3/src/components/Key/style.css

```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 `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/style.css

```css
.key--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;
}

.key--sharp,
.key--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;
}
```

Playing a Sound

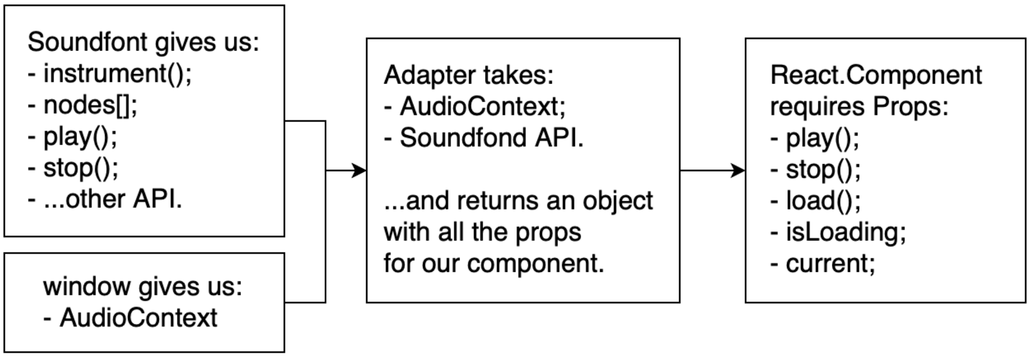
All right, it seems like everything is ready to actually play some sounds in our app. Before we begin, let's add a new custom type called `SoundfontType` in our `.d.ts`. It is going to be useful when we will create an adapter for Soundfont.

03-react-piano/step-4/src/react-app-env.d.ts

```
type SoundfontType = typeof Soundfont
```

Soundfont Adapter

Let's examine what we want from adapter to do. It should take what Soundfont provides as 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 specify what we need as dependencies and as a result.

03-react-piano/step-4/src/adapters/Soundfont/useSoundfont.ts

```
interface 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` interface describes what our `useSoundfont()` adapter hook requires as arguments—in our case we want an `AudioContext` constructor. Then, `Adapted` interface specifies what kind of object we're going to return from our hook.

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. A `current` field contains a current instrument.

Functions `load()`, `play()` and `stop()` are functions which handle loading instrument sounds set, starting playing a note and finishing playing a note respectively. They all are asynchronous, since the `Audio API` is asynchronous by itself.

Async functions in TS 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

```
  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`[91] items. Those are a general interfaces to handling sound operations. Soundfont uses them to store a state of played notes. Notice that 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 with 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 when `.stop()` method is being called on a non-existing node or on a node with an invalid audio state. So, we update this registry directly using 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

[91]https://developer.mozilla.org/ru/docs/Web/API/AudioNode

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[92] to keep a reference to an instance of an `AudioContext` that we create when component mounts. To access this instance we will have to use `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)
  }
```

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 thing's first, we set `loading` state to `true` to indicate that sounds set is being loaded. Then, we call `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 2 more functions: `load()` and `stop()`. Let's build them.

[92] https://reactjs.org/docs/hooks-reference.html#useref

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 `stop()` function is a non-null assertion operator[93]. 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.

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 in what state `audio` is in right now. If it is `suspended`[94] 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.

[93]https://www.typescriptlang.org/docs/handbook/release-notes/typescript-2-0.html#non-null-assertion-operator
[94]https://developer.mozilla.org/en-US/docs/Web/API/AudioContext/suspend

To handle the case when state of `audio` wasn't `suspended`, we use `Promise.resolve()`[95]. 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, then we don't do anything. Otherwise, we create an active `audioNode` by calling `player.play()` method. Then, we save that 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 s `stop()` method on it.

And that is how we created our first sound provider!

Connecting to a Keyboard

In order to use our adapter we have to tweak props of our `Keyboard` and `Key` components a bit. First, let's look at the keyboard.

03-react-piano/step-4/src/components/Keyboard/Keyboard.tsx

```
export interface KeyboardProps {
  loading: boolean
  play: (note: MidiValue) => Promise<void>
  stop: (note: MidiValue) => Promise<void>
}

export const Keyboard: FunctionComponent<KeyboardProps> = ({
  loading,
  stop,
  play
}) => (
  <div className="keyboard">
    {notes.map(({ midi, type, index, octave }) => {
      const label = selectKey(octave as OctavesRange, index)
      return (
```

[95] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/resolve

```
        <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 `loading` flag to disable keys to forbid user to press them while keyboard is not ready. Also, we use `onDown()` and `onUp()` methods to handle key press events.

03-react-piano/step-4/src/components/Key/Key.tsx

```
  onUp: ReactEventHandler<HTMLButtonElement>
  onDown: ReactEventHandler<HTMLButtonElement>
```

Those methods are described now in `KeyProps` and we use them in `onMouseDown()` and `onMouseUp()` props for `button` element.

03-react-piano/step-4/src/components/Key/Key.tsx

```
      onMouseDown={onDown}
      onMouseUp={onUp}
```

Now, we only have to actually connect our `Keyboard` to Soundfont provider, and we're there!

03-react-piano/step-4/src/components/Keyboard/WithInstrument.tsx

```
import React, { FunctionComponent } from "react"
import { useAudioContext } from "../AudioContextProvider"
import { useSoundfont } from "../../adapters/Soundfont"
import { useMount } from "../../utils/useMount"
import { Keyboard } from "../Keyboard"
import "./style.css"

export const KeyboardWithInstrument: FunctionComponent = () => {
  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 exclamation mark to tell type checker that we are sure that `useAudioContext()` returns not `null`. That is because we know that this component will be rendered only if the browser supports Audio API, because we tested it earlier.

Finally, the only thing we have to do is to update our `Main` component to include our connected `KeyboardWithInstrument`.

03-react-piano/step-4/src/components/Main/Main.tsx

```
import React, { FunctionComponent } from "react"
import { KeyboardWithInstrument } from "../Keyboard"
import { NoAudioMessage } from "../NoAudioMessage"
import { useAudioContext } from "../AudioContextProvider"

export const Main: FunctionComponent = () => {
  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 like we want to. In our case we want to subscribe to `keyPress` events.

Let's again start with designing API.

03-react-piano/step-5/src/components/PressObserver/usePressObserver.ts

```
type IsPressed = boolean
type EventCode = string

interface Settings {
  watchKey: KeyLabel
  onStartPress: Function
  onFinishPress: Function
}
```

`IsPressed` is a type alias for `boolean`, it helps us determining if a user pressed a key or not. `EventCode` is a type alias for `event.code`, we will use it to figure out what key is pressed. In `Settings` we use `KeyLabel` to define which key is to be observed. Functions `onStartPress()` and `onFinishPress()` are handlers for when user pressed a key and lift 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. A state (`pressed` or not) we will keep 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

```
  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])
```

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 finished pressing the key we call `onFinishPress()` inside of `handlePressFinish()` handler.

We use `document.addEventListener()` to connect events and our named handler functions, and `document.removeEventListener()` inside of a cleanup function which is returned from the `useEffect()`[96] hook. This is important to remove event listeners in cleanup-function to prevent memory leaks and unwanted event handlers calls.

When `usePressObserver()` is ready, we connect it to our `Key` component.

03-react-piano/step-5/src/components/Key/Key.tsx

```
  const pressed = usePressObserver({
    watchKey: label,
    onStartPress: onDown,
    onFinishPress: onUp
  })

  return (
    <button
      className={clsx(`key key--${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 returned `pressed` value to assign an active `className` to our `button`.

[96] https://reactjs.org/docs/hooks-effect.html

Instruments List

Last thing to do before we dive in *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 among 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

```
export interface 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 `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 an access to the context.

03-react-piano/step-6/src/state/Instrument/Provider.tsx

```
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 local state and exposes `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 get access to the `InstrumentContext` from anywhere.

Instruments Selector

Now, let's actually try to update a current instrument. For that we create a new component called `InstrumentSelector`.

03-react-piano/step-6/src/components/InstrumentSelector/InstrumentSelector.tsx

```
export const InstrumentSelector: FunctionComponent = () => {
  const { instrument, setInstrument } = useInstrument()
  const updateValue = ({ target }: ChangeEvent<HTMLSelectElement>) =>
    setInstrument(target.value as InstrumentName)

  return (
    <select
      className="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 its updating. Then, we create an event handler called `updateValue()` which takes a `ChangeEvent<HTMLSelectElement>` as an argument and calls `setInstrument()` with new `InstrumentName`.

`ChangeEvent` is a generic type that tells React that this function takes some change event of some element, in our case this element is `select`, hence `ChangeEvent<HTMLSelectElement`

Notice how we set `onChange()` property to have a value of `updateValue()`. That is how we connect our `Context` to a component in UI. That is where all the changes affect our state.

Later we render the `select` element, filled with `options` list. We import `options` list from other file besides.

03-react-piano/step-6/src/components/InstrumentSelector/options.ts

```
interface 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 for our case 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`. And `label` is a string that we will put inside of `option` elements to render them visible for users.

A function `normalizeList()` converts instrument names provided by Soundfont into readable ones. You see, Soundfont gives us a list of instruments that typed like `"acoustic_grand_piano"`, and 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 an access to our `InstrumentContext` we have to expose it via `InstrumentContextProvider`.

03-react-piano/step-6/src/components/Playground/Playground.tsx

```
export const Playground: FunctionComponent = () => {
  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, it is not necessary. In our case there are only 2 components that actually use `InstrumentContext`: `Keyboard` and `InstrumentSelector`, so we can wrap only 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

```
export const Main: FunctionComponent = () => {
  const AudioContext = useAudioContext()
  return !!AudioContext ? <Playground /> : <NoAudioMessage />
}
```

We're almost there! The only this 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

03-react-piano/step-6/src/components/Keyboard/WithInstrument.tsx

```
export const KeyboardWithInstrument: FunctionComponent = () => {
  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 instruments sounds set, instead of loading it once when mounted.

Also, we check if an instrument is really changed and load new one only if so. For that we use `current` value which is provided by `useSoundfont()` hook earlier. We compare a `current` instrument in Soundfont provider and a wanted `instrument` from our `Context`. And 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 technics to achieve the same result. One of those technics is a React-pattern called *Render-Props*.

The key idea of this technic is reflected in the title. A component with a render prop[97] takes a function that returns a React element and calls it instead of implementing its own render logic. This technic makes it possible and convenient to share internal logic of a component with another.

Let's try to imagine how a component with `render`-function would look like. Its usage would look somewhat like this:

```
<ExampleRenderPropsComponent
  render={(name: string) => <div>Hello, {name}!</div>}
/>
```

If we look closely to `render` we would notice that it takes a function, that returns another React component. However, it renders not just some 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` kind of connects internal values of `ExampleRenderPropsComponent` with outside world. We, sort of, expose this internal value to outer world. The coolest

[97] https://reactjs.org/docs/render-props.html

thing is that we can decide what to share with outer world and what don't. We could have 100 internal values inside of `ExampleRenderPropsComponent`, but expose only 1. Only one that is needed to be exposed.

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 `name` value to `Greetings` and `Farewell`. We don't recreate all the operations that 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*.

Creating Render-Props With Functional Components

There are 2 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 require to be passed.

03-react-piano/step-7/src/adapters/Soundfont/SoundfontProvider.ts

```
interface ProviderProps {
  instrument?: InstrumentName
  AudioContext: AudioContextType
  render(props: ProvidedProps): ReactElement
}
```

We would require an optional `instrument` prop to specify what instrument we want to load, and an `AudioContext` to work with. But most importantly we would require `render` prop that is a function that takes `ProvidedProps` as an argument and returns a `ReactElement`. `ProvidedProps` is an interface with values that we would provide to outside world, we would describe it like this:

03-react-piano/step-7/src/adapters/Soundfont/SoundfontProvider.ts

```
interface 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 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} />
}

;<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:

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 internal state would be the same as it was in `useSoundfont()` hook. Except that we add loading and reloading sounds when `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 new sounds set for that instrument. However we don't call `load()` function, instead we call a memoized version[98] of it—this is possible because of `useCallback()` hook.

You may notice that this is the logic that we implemented in `KeyboardWithInstrument` component previously, and you will 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 outside world. For that we use `render()`:

[98]https://reactjs.org/docs/hooks-reference.html#usecallback

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 of it an object with all the values and functions that we promised to pass in `ProvidedProps`.

Now the only thing that we have to update for the application to work is to tweak code of `KeyboardWithInstrument` component a bit.

03-react-piano/step-7/src/components/Keyboard/WithInstrument.tsx

```
export const KeyboardWithInstrument: FunctionComponent = () => {
  const AudioContext = useAudioContext()!
  const { instrument } = useInstrument()

  return (
    <SoundfontProvider
      AudioContext={AudioContext}
      instrument={instrument}
      render={(props) => <Keyboard {...props} />}
    />
  )
}
```

Here, we pass the `AudioContext` and an `instrument` as props to `SoundfontProvider` and then pass to `render()` a function that takes `loading`, `play()` and `stop()`, 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 technic but based on a `class`.

`ProvidedProps` would still be the same, because we don't change the public API. `ProviderProps`, on the other hand, will change. This time `instrument` field will not be optional.

03-react-piano/step-7/src/adapters/Soundfont/SoundfontProviderClass.ts

```
interface ProviderProps {
  instrument: InstrumentName
  AudioContext: AudioContextType
  render(props: ProvidedProps): ReactElement
}
```

That's because we will use `defaultProps`[99] 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 `useState()` hook is not available in class components. Hooks can be used only inside of functional components. So, let's introduce the `ProviderState` interface.

03-react-piano/step-7/src/adapters/Soundfont/SoundfontProviderClass.ts

```
interface ProviderState {
  loading: boolean
  current: Optional<InstrumentName>
}
```

Here, we declare that our local state should contain a `loading` field which is a `boolean` and a `current` which is a `Optional<InstrumentName>`. Those are the parts that should cause re-render when changed.

[99] 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 2 types into `Component<>` type. First one describes props and second one describes a state. Also, we created a couple of 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 a good practice to mark everything that is not `public` as `private` or `protected`.

> The difference[100] 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 class as well.

Notice, `defaultProps` there. We declare them as a `static` field on a class.

[100]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[101] 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[102] `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.

Seems pretty well. Now, let's imagine our component's life cycle. What should be done when, so to speak. 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 a new instrument is different to current one and reload it if so.

Technically we described 4 life cycle[103] methods here:

- `constructor()`, which we discussed before

[101] https://www.typescriptlang.org/docs/handbook/classes.html#classes
[102] https://overreacted.io/why-do-we-write-super-props/
[103] https://reactjs.org/docs/state-and-lifecycle.html

- `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 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
    if (instrument && instrument !== prevInstrument)
      this.load(instrument)
  }
```

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`.

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.

Now, we have to implement `this.load()` method for loading sounds.

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—it will be provided later to a component in `render()`. Also, notice that this method is `public`, since we want to expose it to outer world. However, make sure to mark `load()` method private, since we don't want it to be exposed to outer world in any way.

There are 2 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 change not local variables but private class fields instead. All the signatures, API and implementation are the same.

> That is what makes abstraction 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()
}
```

...And expose the methods and values to `render()` function. We access that function from `this.props`, take it and pass as an argument the object with all the values and method 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 `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`[104].

Using a Render-Props 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* 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

And we can consider as cons that a *Render-Props* Provider:

- Adds 1-2 nesting levels to a component which uses it.
- Needs a render to be called.

[104]https://reactjs.org/docs/render-props.html

Higher Order Components

The next React-Pattern we're going to explore is called *Higher Order Component* or HOC. Let's first break down this name to understand what it means.

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 a 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 order *higher that 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 that first*, hence the name—*higher order function*[105].

This kind of functions is useful for *composition.* This term[106] 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:

[105]https://en.wikipedia.org/wiki/Higher-order_function
[106]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 number 3 or 5 times. Also, hardcoded logic is not good in general. And, finally, if we zoom into `twice()` function we can notice similarities with our `incrementTwice()` function.

They both call some function 2 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 `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, variable `fn` starts carrying the value of `increment` function. So, after 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 2 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()` 2 times when called, and it takes some arguments that will pass into `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))
}
```

And 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 1 argument and now it takes an array of arguments. It is a side effect of a fact that now we can use function `twice()` with any other function to repeat it!

```
function sayHello(name: string): void {
  console.log(`Hello, ${name}!`);
}

const sayHelloTwice = twice(sayHello);
sayHelloTwice('Alex');

// Hello, Alex!
// Hello, Alex!
```

Notice that we didn't implement this logic again from scratch. We used a *higher order function* `twice()` to build 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. ...Hmm, but let's define a component.

How it is put in official docs[107], 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 higher order function either takes a function or returns a function or both, we can assume that higher order component is a function that takes a component and returns another one as a result. And this is what the official docs tell us[108].

Whereas 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.

Th public API would stay the same as it was before, however `ProvidedProps` we would call `InjectedProps` now since we would inject them into a component which is going to be enhanced. `ProviderProps` and `ProviderState` are the same as before, but without `render()` method.

03-react-piano/step-8/src/adapters/Soundfont/withInstrument.tsx

```
interface InjectedProps {
  loading: boolean
  play(note: MidiValue): Promise<void>
  stop(note: MidiValue): Promise<void>
}

interface ProviderProps {
  AudioContext: AudioContextType
  instrument: InstrumentName
}

interface ProviderState {
  loading: boolean
```

[107] https://reactjs.org/docs/components-and-props.html
[108] https://reactjs.org/docs/higher-order-components.html

```
  current: Optional<InstrumentName>
}
```

Then, we create a function `withInstrument()` that takes a component needed to be enhanced. We make this function generic, to tell 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>) {
```

Please, pay attention to `extends` keyword in type arguments declaration. This is a generic constraint[109]. We use it to define that `TProps` must have properties that describes in `InjectedProps` type, otherwise TypeScript should give us an error.

Also notice, that by default we define `TProps` to be `InjectedProps` type using = sign. This is 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[110] for debugging. You see, a container component that we're going to create is going to show up in developer tools like any other component. So, we 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`.

[109] https://www.typescriptlang.org/docs/handbook/generics.html#generic-constraints
[110] 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 class `static`[111] 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 a class is the same as it was in `SoundfontProviderClass` from the 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 {...this.props} {...(injected as TProps)} />
      )
    }
```

Here, instead of calling `this.props.render()` and passing an object with values and methods into it, we render our `WrappedComponent` and inject these values and method on it.

[111]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 after.

Also, there is an issue[112] in TypeScript that forces us to explicitly cast `injected` props to `InjectedProps` type, when we use `as InjectedProps`.

Using HOC with Keyboard

When created it can be used to enhance our `Keyboard` component to connect it to Soundfont.

03-react-piano/step-8/src/components/Keyboard/WithInstrument.tsx

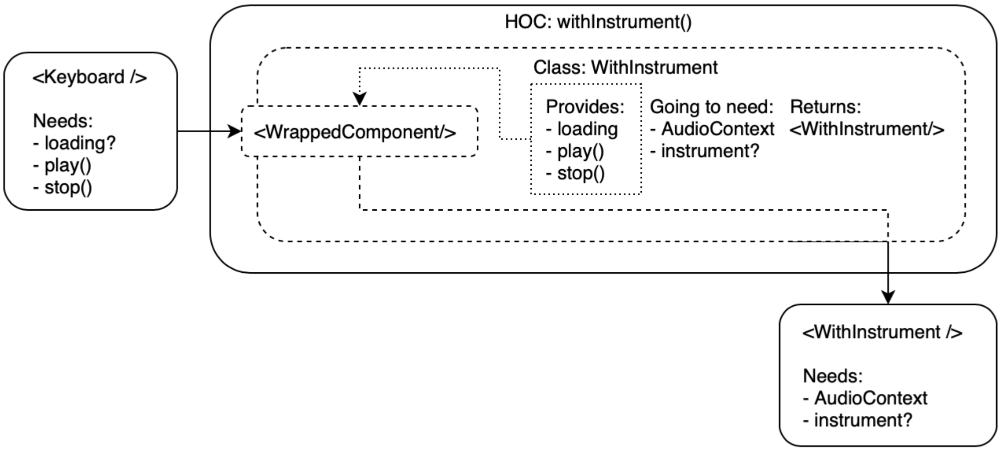
```
const WrappedKeyboard = withInstrument(Keyboard)

export const KeyboardWithInstrument: FunctionComponent = () => {
  const AudioContext = useAudioContext()!
  const { instrument } = useInstrument()

  return (
    <WrappedKeyboard
      AudioContext={AudioContext}
      instrument={instrument}
    />
  )
}
```

Here, we can see how `withInstrument()` is being used: it takes a `Keyboard` component that requires `loading`, `play()` and `stop()` as props and returns a `WrappedKeyboard` that requires `AudioContext` and optional `instrument` props.

This is possible because a `Keyboard` becomes `WrappedComponent` when we call `withInstrument()`. Basically, `WrappedKeyboard` is a `WithInstrument` class that renders out a `Keyboard` with "remembered" injected props.

[112] https://github.com/Microsoft/TypeScript/issues/28938#issuecomment-450636046

At the moment when we render `WrappedComponent` it already has `loading`, `play()` and `stop()`, since they have been injected as `InjectedProps` earlier. And what it requires is `ProviderProps` that were specified on `Component<ProviderProps, ProviderState>`.



Props flow in HOC

You may notice that this is almost exactly like in the example with functions, when `fn` became `increment` and an anonymous function was "remembering" it.

Caveats

We cannot[113] 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[114] over.

There may be a situation when some props that provided by a HOC have the same

[113]https://reactjs.org/docs/higher-order-components.html#dont-use-hocs-inside-the-render-method
[114]https://reactjs.org/docs/higher-order-components.html#static-methods-must-be-copied-over

names as props from other HOC or wrapper. The name collision can lead us to accidentally overridden props.

Passing Refs Through

Refs[115] provide a way to access DOM nodes or React elements created in the render method.

By default refs aren't passed through[116], and for "true" reusability we have to also consider exposing[117] a ref for our HOC. For that we can use[118] `forwardRef()` function.

The base of our HOC will still be the same, however we have to declare some "runtime" types inside of `withInstrument()`.

03-react-piano/step-8/src/adapters/Soundfont/withInstrumentForwardedRef.tsx

```
type ComponentInstance = InstanceType<typeof WrappedComponent>
type WithForwardedRef = ProviderProps & {
  forwardedRef: Ref<ComponentInstance>
}
```

First, we create a `ComponentInstance` type. It is a type[119] consisting of the instance type of a component. We need it to pass into `Ref<>` type to specify a ref of which component it would be. This we put into a `WithForwardRef` type which extends `ProviderProps` type and where `forwardedRef` is a ref that we want to forward further into an enhanced component.

Basically, the root cause of a 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 an 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`.

[115]https://reactjs.org/docs/refs-and-the-dom.html
[116]https://reactjs.org/docs/higher-order-components.html#refs-arent-passed-through
[117]https://reactjs.org/docs/forwarding-refs.html
[118]https://github.com/typescript-cheatsheets/react-typescript-cheatsheet/blob/master/HOC.md
[119]https://www.typescriptlang.org/docs/handbook/utility-types.html#instancetypet

03-react-piano/step-8/src/adapters/Soundfont/withInstrumentForwardedRef.tsx

```
class WithInstrument extends Component<
  WithForwardedRef,
  ProviderState
> {
```

In `render()` method we access `forwardedRef` from props and pass it as `ref` props onto a `WrappedComponent`.

03-react-piano/step-8/src/adapters/Soundfont/withInstrumentForwardedRef.tsx

```
    public render() {
      const { forwardedRef, ...rest } = this.props
      const injected = {
        loading: this.state.loading,
        play: this.play,
        stop: this.stop
      } as InjectedProps

      return (
        <WrappedComponent
          ref={forwardedRef}
          {...rest}
          {...(injected as TProps)}
        />
      )
    }
```

The rest of 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 the other props. And in order to get an access to a `ref`, we have to call a special `forwardRef()` function.

As an argument for it we provide another anonymous function which return our `WithInstrument` component. Notice that this function receives 2 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 when we don't need to change an instrument in runtime, and we want to specify it once. In that 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? There is! It is called static composition.

So far we worked with, as they call it, dynamic composition—when 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 on of those!

Again let's determine what a 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: FunctionComponent = () => {
  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. So, we can create as many functions as we want beforehand and use them co connect components to Soundfont after!

From Hooks to HOCs

Since HOCs are just functions that return components, we can reckon that they can be based on hooks as well.

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 sounds sets inside of `WithInstrumentComponent` and expose to outside only `ProviderProps`. However, all the logic of this 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 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.
- Not obvious 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, comparing to functional components.
- HOCs may become too verbose.

Conclusion

Congratulations! We have created the piano keyboard that plays sounds of many instruments! But most importantly we now can solve problems with sharing logic and reducing duplications using different technics such as *Render-Props* and *Higher Order Components*.

Next.js and Static Site Generation: Building a Medium-like Blog

Introduction

So far we have been creating Single Page Applications[120], as known as SPA. They got this name because of a way that page refresh goes: our application would not reload the whole page, but it would fetch new data and re-render only parts of the page that should be updated instead. Since all this happens on the same page, they are called SPA.

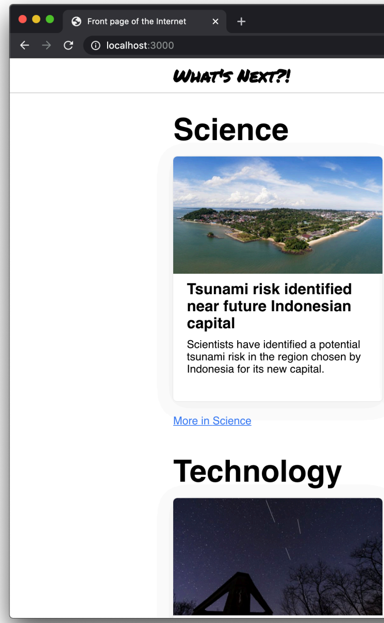
There is a caveat in this flow, though. Say, we want all the pages of our application to be detectable by search engines. It cannot be done if all the data fetching and re-render happens only in a user's browser. The vast majority of search robots wouldn't wait until the real content of an application would appear. They would instead read the content of the HTML we serve them at start, which is almost empty.

For an application that hugely rely on its content, like say a blog platform or a news site, it is not acceptable. And here the pre-rendering[121] comes in.

What we're going to build

In order 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 a BBC site[122] as a source of news and images and create an application which will have pre-rendered pages with content on them.

[120]https://en.wikipedia.org/wiki/Single-page_application
[121]https://nextjs.org/docs/basic-features/pages#pre-rendering
[122]https://www.bbc.com



Science

Tsunami risk identified near future Indonesian capital

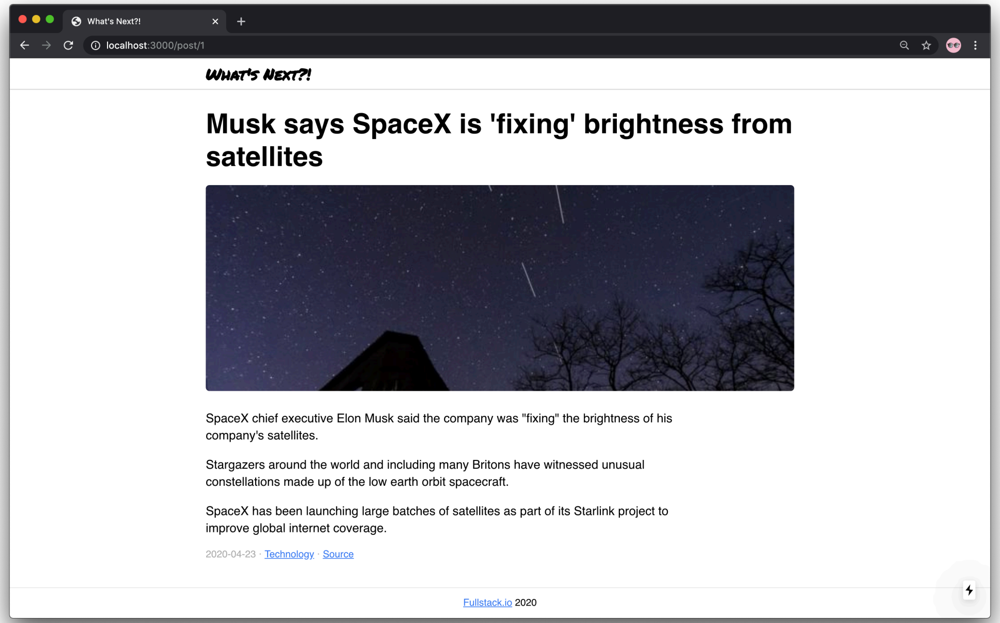
Scientists have identified a potential tsunami risk in the region chosen by Indonesia for its new capital.

More in Science

Technology

The main page of the completed application will look like this:

And a post page will look like this:



Musk says SpaceX is 'fixing' brightness from satellites

SpaceX chief executive Elon Musk said the company was "fixing" the brightness of his company's satellites.

Stargazers around the world and including many Britons have witnessed unusual constellations made up of the low earth orbit spacecraft.

SpaceX has been launching large batches of satellites as part of its Starlink project to improve global internet coverage.

2020-04-23 · Technology · Source

Fullstack.io 2020

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
```

It should open the app in the browser. If it didn't, navigate to http://localhost:3000[123] and open it manually.

[123] http://localhost:3000

Pre-rendering

As we said earlier, for an application that hugely rely 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 2 major ways to pre-render pages are: Server Side Rendering and Static Site Generation.

Server Side Rendering

Server Side Rendering[124], or SSR is a technic when a server renders real HTML for every page request it gets. For our application it would mean that a server would render HTML for each post page, section page, etc.

SSR doesn't require us us to have to store each page as an HTML-file on a server, not at all. 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 a client.

Each page is associated with minimal 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 "freezed" resurrects and runs from a point which it was "freezed" at. This process is called hydration[125].

Static Site Generation

Static Site Generation[126], or SSG, on the other hand means that pages' HTML is generated at build time once. So, technically it means that we will have all the real HTML files for each page.

The advantage of this technic 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.

[124]https://nextjs.org/docs/basic-features/pages#server-side-rendering
[125]https://nextjs.org/docs/basic-features/pages#pre-rendering
[126]https://nextjs.org/docs/basic-features/pages#static-generation-recommended

In our case SSG is perfect, because we can predict which content should appear on which page.

Next.js

Since, we're focusing on SSG, we're going to use Next.js[127] (from now on and later—Next).

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 a great documentation and tutorials to learn.

Setting up a project

First of all, we have to set up a project. Next has a set of instructions[128] 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 05-ssr-and-ssg
```

Inside, we have to create 2 more directories `pages` and `public`. First one is a directory in which Next will search for pages[129] of our application, we will talk about pages in detail a bit later. Second one is a directory for static resources[130] like images, stylesheets etc.

```
cd 05-ssr-and-ssg
mkdir pages
mkdir public
```

Then, let's initialize a project and add all the dependencies we're going to need:

[127]https://github.com/zeit/next.js/
[128]https://nextjs.org/docs/getting-started
[129]https://nextjs.org/docs/basic-features/pages
[130]https://nextjs.org/docs/basic-features/static-file-serving

```
yarn init -y
yarn add next react react-dom
```

Once initialized, we want to update `scripts` section of our `package.json` file and add following scripts:

05-next-ssg/step-1/package.json

```
  "scripts": {
    "dev": "next",
    "build": "next build",
    "start": "next start"
  },
```

Among those scripts: - `dev`, it will run a development environment, we will use it the most often. - `build`, it will build our application and generate rendered pages. - `start`, we won't use it in this chapter, but this script is being used in production environments on servers when application is started.

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 a project:

```
touch tsconfig.json
```

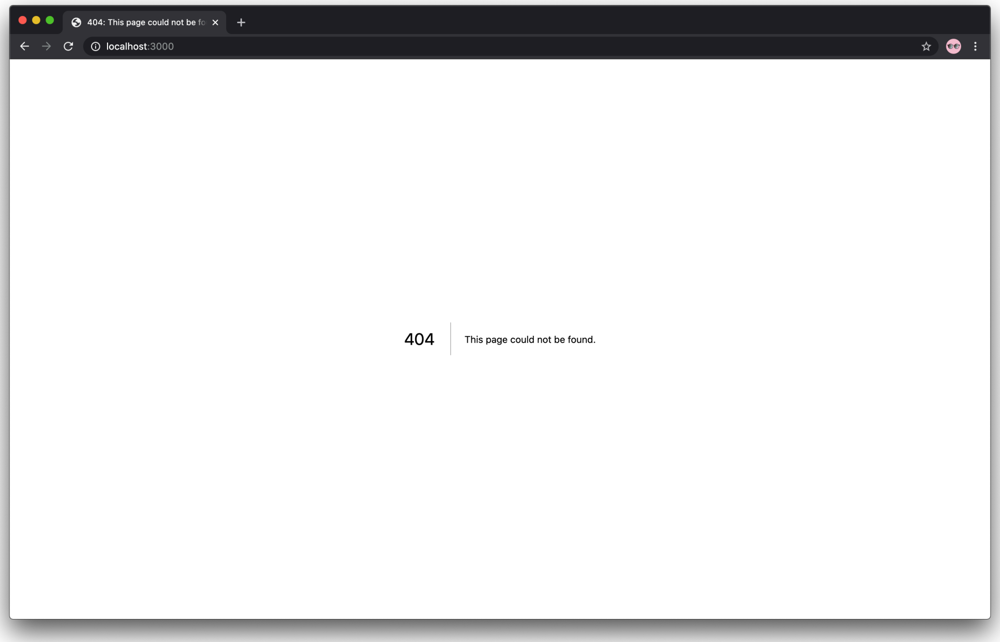
Notice that we don't populate it with any content, Next will do it for us automatically when we run:

```
yarn dev
```

This command should open the app in the browser. If it didn't, navigate to http://localhost:3000[131] and open it manually.

First page

When opened the application should show 404 error.



404 | This page could not be found.

By default there is "Not found" error

This is fine. Next couldn't render anything because we haven't created any page yet. So, let's fix that!

A page[132] 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 before—to populate it with pages components.

[131]http://localhost:3000
[132]https://nextjs.org/docs/basic-features/pages

To create our first page we need to create a 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.

Other interesting thing is a `Head` component imported from `next/head`. This is a component that injects everything we pass as children inside of `head` element on an HTML-page. In our case we pass there `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, where 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 React, { FunctionComponent } from "react"
import Link from "next/link"
import { Center } from "../Center"
import { Container, Logo } from "./style"

export const Header: FunctionComponent = () => {
  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 installed, 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 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 `Link` component[133] imported from `next/link`. This is a component which enables client-side transition between routes of our app—basically, between pages[134].

Please, pay attention to a structure of a `Link` we created. At the top level we use `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 for some reasons cannot pass an `a` element, we can use different elements or components and force[135] `Link` to pass an `href` prop further. It will be useful later, when we will use styled links.

`Center` component

Another component that we will use across the whole project is a `Center` component. It is a styled component which does only 1 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;
  }
`
```

[133]https://nextjs.org/docs/api-reference/next/link
[134]https://nextjs.org/docs/routing/introduction
[135]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 React, { FunctionComponent } from "react"
import { Center } from "../Center"
import { Container } from "./style"

export const Footer: FunctionComponent = () => {
  const currentYear = new Date().getFullYear()

  return (
    <Container>
      <Center>
        <a href="https://fullstack.io">Fullstack.io</a> {currentYear}
      </Center>
    </Container>
  )
}
```

It will contain a current year and a link to Fullstack.io site. Notice that here we use not a `Link` component, but an ordinary `a` element instead. That' 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`

When we created all of the components we're going to need, we want to use them in the app layout.

First thought of 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 code of every new page we're going to create. This is not convenient and it violates DRY principle (Don't Repeat Yourself).

For this problem Next has a solution. We can create a component which will be like a wrapper for every page Next is going to render. This component is `App`[136].

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.

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 theme which we will create in `shared/theme` in a minute, 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.

[136]https://nextjs.org/docs/advanced-features/custom-app

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` element 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 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 fonts and colours 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>`
```

And then we create some basic global styles for `body`, headings, links and `.main` block.

05-next-ssg/step-2/shared/theme.ts

```
  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 in 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 according components.

Custom `_document`

So far we created global styles and theme, but if we look closely at our theme we can find that accent font is defined to be `"Permanent Marker"` font-family. This is not the 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 when 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[137].

Next's `Document` component not only encapsulates `html` and `body` declarations, but can also include initial props[138] 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

[137]https://nextjs.org/docs/advanced-features/custom-document
[138]https://nextjs.org/docs/api-reference/data-fetching/getInitialProps#context-object

server and markup on a client are the same. Otherwise we would get an error that some properties are not the same.

In order 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.

First of all, 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, 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 `html` element, but we use `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 when 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[139], such as `pathname` of a page URL, `req` for request, `res` for response and and error object `err` if 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.

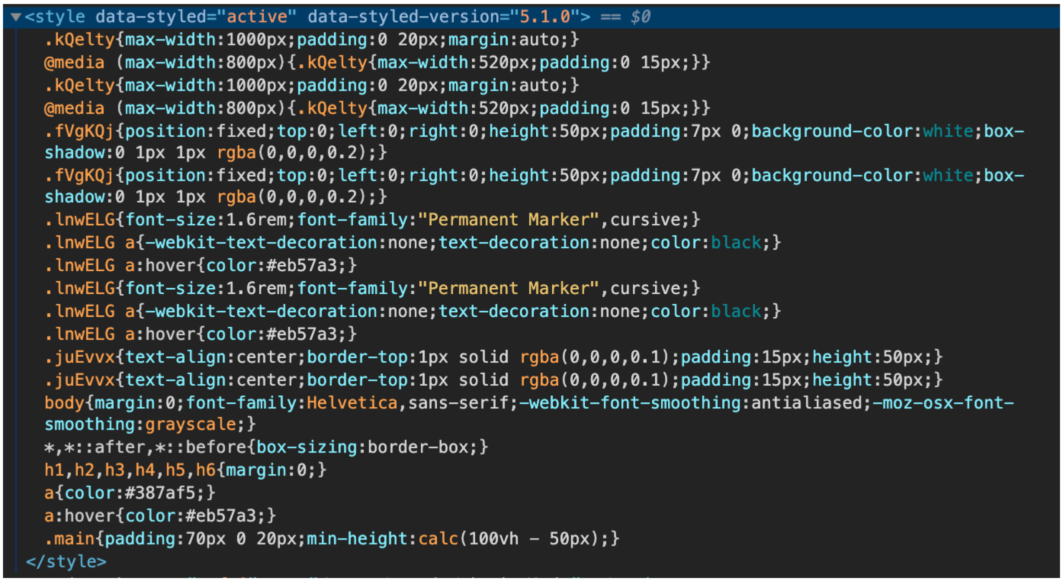
When overriding it we use `sheet.collectStyles()`[140] method and pass the whole rendered application as an argument. It will gather all the styles so that we will be able to extract them by calling `sheet.getStyleElement()` later.

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.

[139]https://nextjs.org/docs/api-reference/data-fetching/getInitialProps#context-object
[140]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 required to be sent along with the page markup.

In the browser it should look like a `style` element filled with app styles:



```
▼<style data-styled="active" data-styled-version="5.1.0"> == $0
   .kQelty{max-width:1000px;padding:0 20px;margin:auto;}
   @media (max-width:800px){.kQelty{max-width:520px;padding:0 15px;}}
   .kQelty{max-width:1000px;padding:0 20px;margin:auto;}
   @media (max-width:800px){.kQelty{max-width:520px;padding:0 15px;}}
   .fVgKQj{position:fixed;top:0;left:0;right:0;height:50px;padding:7px 0;background-color:white;box-
   shadow:0 1px 1px rgba(0,0,0,0.2);}
   .fVgKQj{position:fixed;top:0;left:0;right:0;height:50px;padding:7px 0;background-color:white;box-
   shadow:0 1px 1px rgba(0,0,0,0.2);}
   .lnwELG{font-size:1.6rem;font-family:"Permanent Marker",cursive;}
   .lnwELG a{-webkit-text-decoration:none;text-decoration:none;color:black;}
   .lnwELG a:hover{color:#eb57a3;}
   .lnwELG{font-size:1.6rem;font-family:"Permanent Marker",cursive;}
   .lnwELG a{-webkit-text-decoration:none;text-decoration:none;color:black;}
   .lnwELG a:hover{color:#eb57a3;}
   .juEvvx{text-align:center;border-top:1px solid rgba(0,0,0,0.1);padding:15px;height:50px;}
   .juEvvx{text-align:center;border-top:1px solid rgba(0,0,0,0.1);padding:15px;height:50px;}
   body{margin:0;font-family:Helvetica,sans-serif;-webkit-font-smoothing:antialiased;-moz-osx-font-
   smoothing:grayscale;}
   *,*::after,*::before{box-sizing:border-box;}
   h1,h2,h3,h4,h5,h6{margin:0;}
   a{color:#387af5;}
   a:hover{color:#eb57a3;}
   .main{padding:70px 0 20px;min-height:calc(100vh - 50px);}
 </style>
```

Final collected styles

After all, in `finally` clause we call `sheet.seal()` method. Thus, we make sure that `sheet` object is available for garbage collector[141].

Site front page

On a front page we will have a `Feed` with `Post` cards in side. Let's update our `Front` component and include `Feed` in `main` element.

[141] https://styled-components.com/docs/advanced#example

05-next-ssg/step-3/pages/index.tsx

```
      <main>
        <Feed />
      </main>
```

Feed

Then, we want to create a `Feed` component. Our `Feed` would contain 3 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 React, { FunctionComponent } from "react"
import { Section } from "../Section"

export const Feed: FunctionComponent = () => {
  return (
    <>
      <Section title="Science" />
      <Section title="Technology" />
      <Section title="Arts" />
    </>
  )
}
```

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

```
interface SectionProps {
  title: string
}
```

And `Section` itself will contain a `Title` and a `Grid` with a bunch of (hardcoded for now) `Post` cards inside.

05-next-ssg/step-3/components/Section/Section.tsx

```
export const Section: FunctionComponent<SectionProps> = ({ title }) => {
  return (
    <section>
      <Title>{title}</Title>
      <Grid>
        <Post />
        <Post />
        <Post />
      </Grid>
    </section>
  )
}
```

A `Grid` component is a styled component which uses `display: flex` to line up the content inside. The `:after` pseudo-element is required to prevent elements in last row from wrong positioning[142].

[142]https://stackoverflow.com/questions/18744164/flex-box-align-last-row-to-grid

05-next-ssg/step-3/components/Section/style.ts

```
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%;
    }
  }
```

Post

Now, let's create a `Post` card. This component will play a 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

```
export const Post: FunctionComponent = () => {
  return (
    <Link href="/post/[id]" as="/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, se\
d 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 `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.

Then, we define `href` and `as` props on `Link`. When we work with dynamic routes[143] in Next we use "[]" to specify dynamic part of a route. In our case `[id]` is a dynamic part of a post route. It will represent a post id that we want to load.

The `href` is the name of the page in the pages directory. And the `as` is the url that will

[143]https://nextjs.org/docs/routing/dynamic-routes

be shown in the browser. We will use it later as well to create pretty urls for every post in `Feed`.

Also, `as` prop helps Next determine which pages to pre-render. Therefore **it is possible to miss pre-rendering of some pages** when using dynamic segments in `href` right away like this:

```
// this is alright
<Link href="/posts/[id]" as={`/posts/${post.id}`} />
```

…and not like this:

```
// this may result in missing pre-rendering of that page
<Link href={`/posts/${post.id}`} />
```

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 make 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.

Now, on the main page you should see a three `Section` components with `3` `Post` cards in each of them. However, if we click on any of `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[144] 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.

[144]https://nextjs.org/docs/advanced-features/custom-error-page

05-next-ssg/step-3/pages/404.tsx

```
const NotFound: FunctionComponent = () => {
  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

```
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 i\
nfo.
```

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 server later.

To create a page that is responsible for a path with dynamic route segment[145], 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[146]

Nothing special inside so far. But let's examine more closely a `useRouter()` hook[147]. It is a hook that provides access to a `router` object[148].

In that object there are 2 values that we are interested in: - `pathname`, current route. This is 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 caccess it and use for loading data later on.

[145]https://nextjs.org/docs/routing/dynamic-routes
[146]./code/05-next-ssg/step-3/pages/post/[id].tsx
[147]https://nextjs.org/docs/api-reference/next/router#userouter
[148]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 a 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 in them. We can get this data in many different ways: - from file system (as .md files for example); - from a remote data base directly; - from a backend server's API.

Next has a great example[149] on working with file system. We, however, will create a "backend server" and fetch data from it's API.

First of all, let's install 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 nodejs server using a `server/index.ts` file. Let's create one.

[149] 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 some DB (like MongoDB for example), but for simplicity sake we will read data right from json files. You can find them in `05-next-ssg/step-4/server` directory.

Post data and interface

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 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. Then we use them to describe a `Post` interface:

05-next-ssg/step-4/shared/types.ts

```
export interface 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 setup a port 4000 for this server and create 2 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 the same for `/categories`.

Frontend API client

Now, when we have created a server API, we can create a frontend client for that API. Let's create a directory `api` with 2 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"
}
```

And `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 `node-fetch` package here. This is because when Next builds a project it will run outside of browser's environment, so it won't have access to `fetch()` function. This package creates a function alike `fetch()` available in node.

Then there are `fetchPosts()` and `fetchCategories()` functions. Both are `async` and return `Promise`. First one requests `/posts` and returns a promise of `Post[]`. Second one—`/categories` and `Category[]` respectively. These functions we will use for fetching and pre-fetching data on main page.

Updating main page

When 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

```
interface FrontProps {
  posts: Post[]
  categories: Category[]
}
```

Here, we cerate an interface `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, now let's take a look at how we can pre-render this page.

Static props

Next has a concept of static props[150]. Those are the props that Next will inject at build time in 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()`.

[150]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 2 requests to our backend API: `fetchCategories()` fetches categories for the main page, and `fetchPosts()` fetches posts. Then, we return an object with `props` that contain 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

```
interface FeedProps {
  posts: Post[]
  categories: Category[]
}

export const Feed: FunctionComponent<FeedProps> = ({ posts, categories \
}) => {
```

We start with declaring an interface `FeedProps` and accessing them inside of a component.

05-next-ssg/step-4/components/Feed/Feed.tsx

```
  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 with declaring an interface `SectionProps` and accessing them inside of a component.

05-next-ssg/step-4/components/Section/Section.tsx

```
interface SectionProps {
  title: string
  posts: PostType[]
}

export const Section: FunctionComponent<SectionProps> = ({ title, posts\
 }) => {
```

Then, we render a `Title` and `Grid` with `Post` cards inside.

05-next-ssg/step-4/components/Section/Section.tsx

```
  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

```
interface PostProps {
  post: PostType
}

export const Post: FunctionComponent<PostProps> = ({ post }) => {
```

We declare an interface `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 tells 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/[id]" as={`/post/${post.id}`} passHref>
```

We use `post.id` in `as` prop to make our urls look pretty. So that when we render a post with `"id": "some-post"` url would look like `/posts/some-post/`.

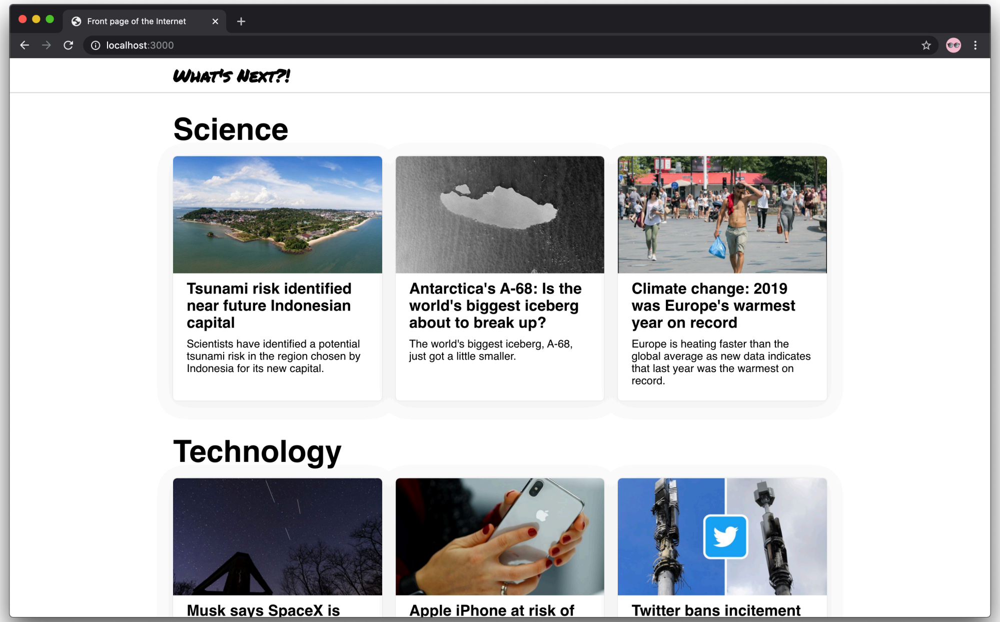
The last thing we have to do now is to render every piece of information from `post` in the card.

05-next-ssg/step-4/components/Post/Post.tsx

```
    <Link href="/post/[id]" as={`/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!



Science

Tsunami risk identified near future Indonesian capital

Scientists have identified a potential tsunami risk in the region chosen by Indonesia for its new capital.

Antarctica's A-68: Is the world's biggest iceberg about to break up?

The world's biggest iceberg, A-68, just got a little smaller.

Climate change: 2019 was Europe's warmest year on record

Europe is heating faster than the global average as new data indicates that last year was the warmest on record.

Technology

Musk says SpaceX is

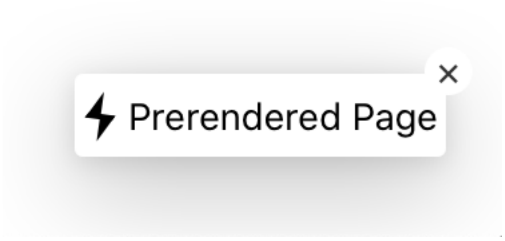
Apple iPhone at risk of

Twitter bans incitement

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 right bottom corner of a page. It appears[151] on pages that Next statically generated.



Pre-rendered page indicator

[151]https://nextjs.org/docs/api-reference/next.config.js/static-optimization-indicator

Pre-render post page

Now, let's create a pre-rendered post page.

API

First thing for us to do is to create an API endpoint for getting a 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 an `id` of a needed post, then search for post with the same `id` among the list of all posts and return 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 and static paths

For a post page we also want to declare a props interface since this component will accept data via props.

<<05-next-ssg/step-5/pages/post/[id].tsx[152]

Then, since this page is also going to be pre-rendered, we create `getStaticProps()` function.

<<05-next-ssg/step-5/pages/post/[id].tsx[153]

We import `GetStaticProps` from `next` package to declare 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[154].

It contains `params` object, which contains the route parameters for pages that use dynamic routes. Since our page has dynamic segment (`[id]`) this object has an `id` property with a value that is equal to an `id` of a current post, which we will use to fetch data.

Static paths

There is another exported function though, called `getStaticPaths()`. This function determines[155] which paths should be rendered to HTML at build time.

<<05-next-ssg/step-5/pages/post/[id].tsx[156]

Here, we see that this function returns an object with 2 fields. 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 "fallback" version of those paths.

In our case we use `router.isFallback` property to render `Loader` component (will cover a bit later). When a user requests a page which is not yet rendered, but has a "fallback", they will see a `Loader`. At that time in the background Next will statically

[152]./code/05-next-ssg/step-5/pages/post/[id].tsx
[153]./code/05-next-ssg/step-5/pages/post/[id].tsx
[154]https://nextjs.org/docs/basic-features/data-fetching#getstaticprops-static-generation
[155]https://nextjs.org/docs/basic-features/data-fetching#getstaticpaths-static-generation
[156]./code/05-next-ssg/step-5/pages/post/[id].tsx

generate the requested path HTML and JSON. A browser then will receive those HTML and JSON and swap from a "fallback" page to a rendered one.

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

```
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 posts with which ids it should pre-render.

Then we finish our `Post` page component.

<<05-next-ssg/step-5/pages/post/[id].tsx[157]

Inside we use `useRouter()` hook to get access to `router` object. Then we check 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.

[157] ./code/05-next-ssg/step-5/pages/post/[id].tsx

05-next-ssg/step-5/components/Loader/Loader.tsx

```
import React, { FunctionComponent } from "react"
import { Container } from "./style"

export const Loader: FunctionComponent = () => {
  return <Container>Loading...</Container>
}
```

`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

```
interface PostBodyProps {
  post: Post
}

export const PostBody: FunctionComponent<PostBodyProps> = ({ post }) =>\
 {
```

...And return a block with main post info first:

05-next-ssg/step-5/components/Post/PostBody.tsx

```
  return (
    <div>
      <Title>{post.title}</Title>
      <Figure>
        <img src={post.image} alt={post.title} />
      </Figure>

      <Content dangerouslySetInnerHTML={{ __html: post.content }} />
```

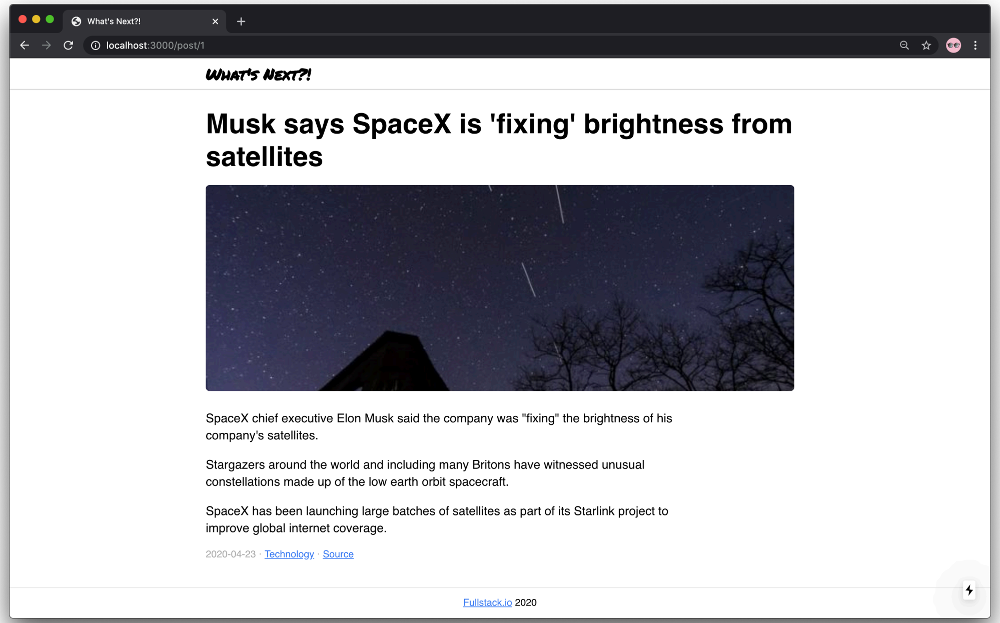
...And post meta info last:

05-next-ssg/step-5/components/Post/PostBody.tsx

```
      <Meta>
        <span>{post.date}</span>
        <span>&middot;</span>
        <Link href="/category/[id]" as={`/category/${post.category}`}>
          <a>{post.category}</a>
        </Link>
        <span>&middot;</span>
        <a href={post.source}>Source</a>
      </Meta>
```

We use `dangerouslySetInnerHTML` on `Content` component only for simplicity sake. Since our posts have HTML markup in their `content` fields we render them right away. In real world application we should consider text preprocessing to avoid XSS or other security vulnerabilities.

In `Meta` we also create a link to 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.



Musk says SpaceX is 'fixing' brightness from satellites

SpaceX chief executive Elon Musk said the company was "fixing" the brightness of his company's satellites.

Stargazers around the world and including many Britons have witnessed unusual constellations made up of the low earth orbit spacecraft.

SpaceX has been launching large batches of satellites as part of its Starlink project to improve global internet coverage.

2020-04-23 · Technology · Source

Fullstack.io 2020

Statically generated post page

And it is 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.

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 3 times, just to make it a bit bigger, than it is. We do it only to make an example simpler. In real world API we would make a request to a data base 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 our API is ready!

`Category` page component

Next, we want to create a `Category` page component. First of all, let's design a props for it. `Category` component will take a list of `Post` items as a `posts` prop.

<<05-next-ssg/step-6/pages/category/[id].tsx[158]

[158] ./code/05-next-ssg/step-6/pages/category/[id].tsx

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[159]

As well as we created `getStaticProps()` we want to create `getStaticPaths()` function. Again, we make `fallback` property equal to `true` just to make sure that no page would return 404 when it is not pre-rendered.

<<05-next-ssg/step-6/pages/category/[id].tsx[160]

Static paths for this page will be a list of objects with `{params: { id: category }}`. By default we include 3 categories to pre-render which a listed in `categoriesToPreRender`.

05-next-ssg/step-6/shared/staticPaths.ts

```
const categoriesToPreRender: Category[] = ["Science", "Technology", "Ar\
ts"]

export const categoryPaths: CategoryStaticPath[] = categoriesToPreRende\
r.map(
  (category) => ({ params: { id: category } })
)
```

And finally, we check if page is not pre-rendered and render `Loader` component, or render `Section` otherwise.

<<05-next-ssg/step-6/pages/category/[id].tsx[161]

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 3 post cards, though. 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 "More link" or not.

[159]./code/05-next-ssg/step-6/pages/category/[id].tsx
[160]./code/05-next-ssg/step-6/pages/category/[id].tsx
[161]./code/05-next-ssg/step-6/pages/category/[id].tsx

05-next-ssg/step-6/components/Section/Section.tsx

```
interface SectionProps {
  title: string
  posts: PostType[]
  isCompact?: boolean
}
```

Then, we access this prop:

05-next-ssg/step-6/components/Section/Section.tsx

```
export const Section: FunctionComponent<SectionProps> = ({
  title,
  posts,
  isCompact = false
}) => {
```

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 `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 (
          <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. 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 an interface `BreadcrumbsProps` and getting access to `post` prop.

05-next-ssg/step-6/components/Breadcrumbs/Breadcrumbs.tsx

```
interface BreadcrumbsProps {
  post: Post
}

export const Breadcrumbs: FunctionComponent<BreadcrumbsProps> = ({ 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

```
  return (
    <Container>
      <Link href="/">
        <a>Front</a>
      </Link>
      <span>▯</span>
      <Link href="/category/[id]" as={`/category/${post.category}`}>
        <a>{post.category}</a>
      </Link>
    </Container>
  )
```

And then we want to render it in `PostBody` component right above the post title.

05-next-ssg/step-6/components/Post/PostBody.tsx

```
  return (
    <div>
      <Breadcrumbs post={post} />
```

Building a project

Now it is finally time to build our project. For that we have a script `yarn build`.

However, if we run it right now, 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[162] in `out` directory via `next export` script. Though we would want to change the build destination directory to ours—`build`.

In order to do that we have to create a file called `next.config.js`. This is a configuration file[163] for Next framework.

One of the configuration options is `distDir`[164]—it is a name to use for a custom build directory.

In our case we want to use `build` for that:

05-next-ssg/step-7/next.config.js

```
module.exports = {
  distDir: "build"
}
```

Now, we can run `yarn serve` in one terminal window to setup a "backend server" and `yarn build` in another. After 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 a name of a directory inside of `build/server/static` which contains a current build artifacts like pages' HTML and JSON.

Conclusion

In this chapter we learned how to create applications using Next.js framework and hot to use Static Site Generation for pre-rendering pages.

162 https://nextjs.org/docs/advanced-features/static-html-export
163 https://nextjs.org/docs/api-reference/next.config.js/introduction
164 https://nextjs.org/docs/api-reference/next.config.js/setting-a-custom-build-directory

Using Redux and TypeScript - (COMING SOON)

This chapter is coming soon (Summer 2020)

GraphQL, React, and TypeScript (COMING SOON)

This chapter is coming soon (Summer 2020)

Appendix

Changelog

Revision 1p (05-20-2020)

First “Early Draft” Release