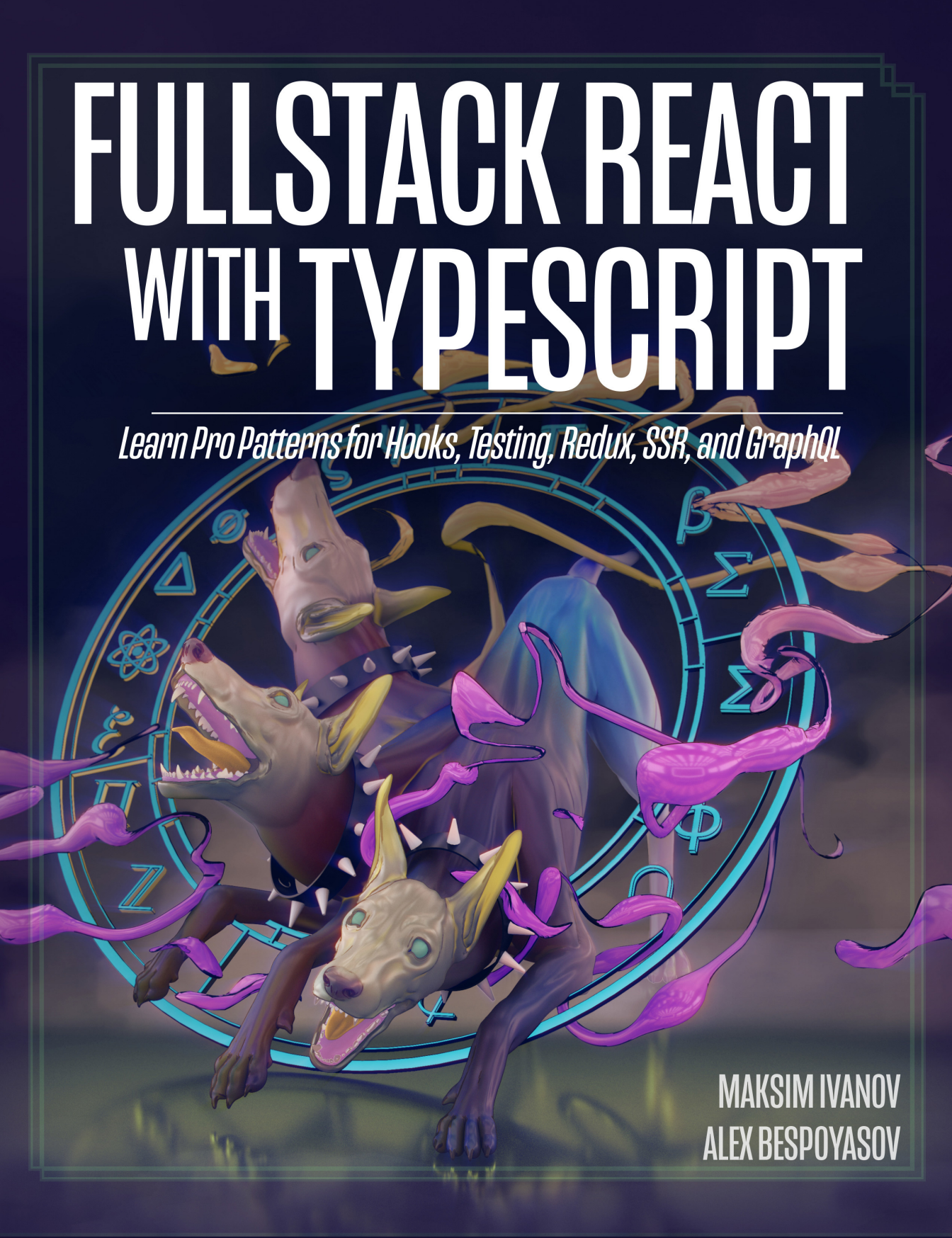
FULLSTACK REACT WITH TYPESCRIPT

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 r7 - 2020-12-01

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!"
```

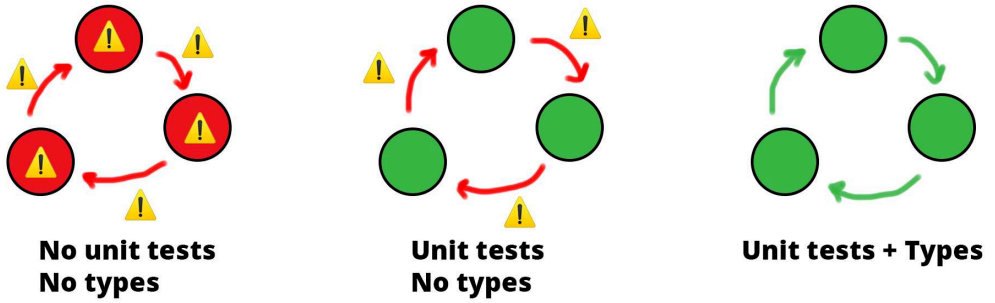
If we'll try to pass something else - we'll get an error.

```
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 between 40% and 90% decrease in pre-release bug density.

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

Why Use TypeScript With React

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

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

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 as callbacks. So if something happens in child component it can notify its parent by calling the passed callback function.

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

If you 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 powerful. It gives you more options to define the types and then it allows you to use this type information in many different ways. We will demonstrate 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 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/GYVwdgxgLglg9mABAWwKYGd0FUAOAVAC1QEEAnUgQwE8AKC8gLkTMqoB50pSYwBzRAD6IwIZACNUpAHwBKJgDc4MACaIAfCE1HrzoYQBM6U4ucAA2qIZdcLyFhlBwADJwAO6SAMIU6Kg0MjLaQA

```
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.

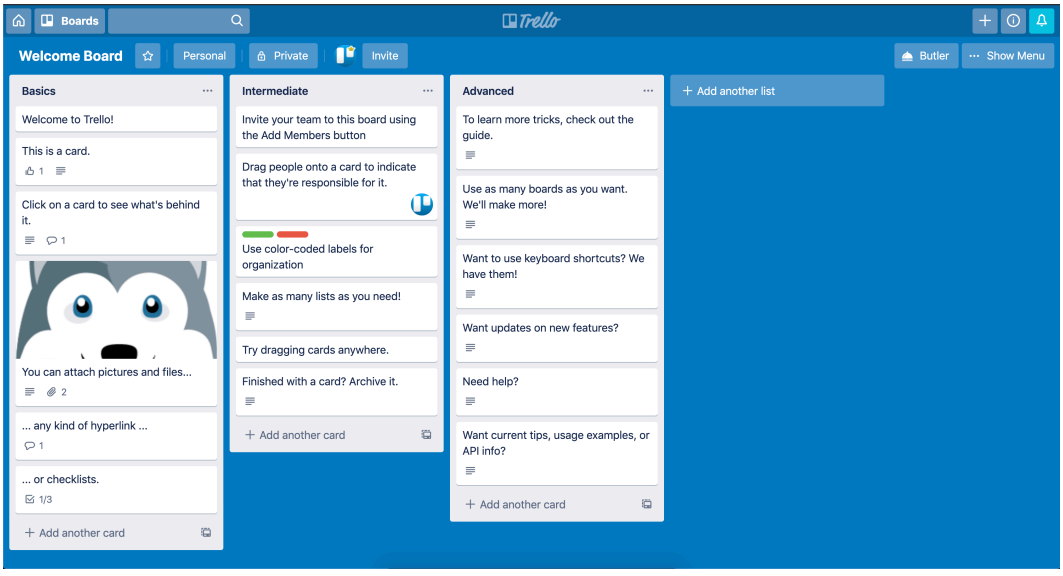
All the examples for this chapter contain `yarn.lock` files, remove them if you want to use npm to install dependencies.

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



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`.

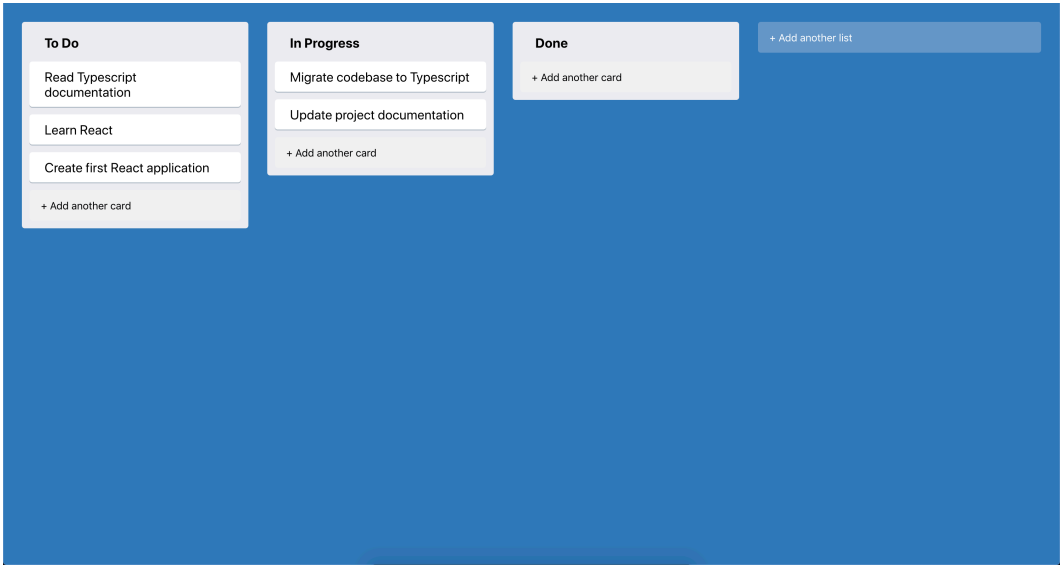
Unzip the archive and `cd` to the app folder.

```
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.



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
│   ├── reportWebVitals.ts
│   └── setupTests.ts
├── node_modules
│   └── ...
├── README.md
├── package.json
```

```
22 ├── tsconfig.json
23 └── 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": "^5.11.4",
    "@testing-library/react": "^11.1.0",
    "@testing-library/user-event": "^12.1.10",
    "@types/jest": "^26.0.15",
    "@types/node": "^12.0.0",
    "@types/react": "^16.9.53",
    "@types/react-dom": "^16.9.8",
    "react": "^17.0.1",
    "react-dom": "^17.0.1",
```

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

```
    "react-scripts": "4.0.0",
    "typescript": "^4.0.3",
    "web-vitals": "^0.2.4"
  },
```

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

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

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

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 `^17.0.1` and `@types/react-dom` is `^16.9.8`.

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 reportWebVitals from './reportWebVitals';

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

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

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

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

```
ReactDOM.render(
```

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.

After that we import `reportWebVitals`, this module can be useful if you want to measure your app performance. It is explained in more detail here[36].

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

Then we render the `App` using the `ReactDOM.render` method. Note that by default the `App` component is wrapped into the `React.StrictMode` component.

This component mostly checks that no deprecated methods are being used. All those checks are performed only in development mode, and it is a good practice to wrap your app into this component.

Check the documentation[37] for the updated list of the `StrictMode` functionality.

App.tsx

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

Currently in JavaScript apps generated with `create-react-app` you don't need to import React at all. Read more here[38].

In older versions, React was imported differently.

Instead of:

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

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

You would see:

[36] https://create-react-app.dev/docs/measuring-performance/
[37] https://reactjs.org/docs/strict-mode.html
[38] https://reactjs.org/blog/2020/09/22/introducing-the-new-jsx-transform.html

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

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

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

But if you've used React with JavaScript before - you'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:

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

```
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[40].

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 contains 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[41]

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

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

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/react-app.d.ts
```

Instead of looking up the file in the `node_modules` folder you can check the react-scripts GitHub repo[42].

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:

```
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`.

[42] https://github.com/facebook/create-react-app/blob/master/packages/react-scripts/package.json#L29

```
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`

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

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

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

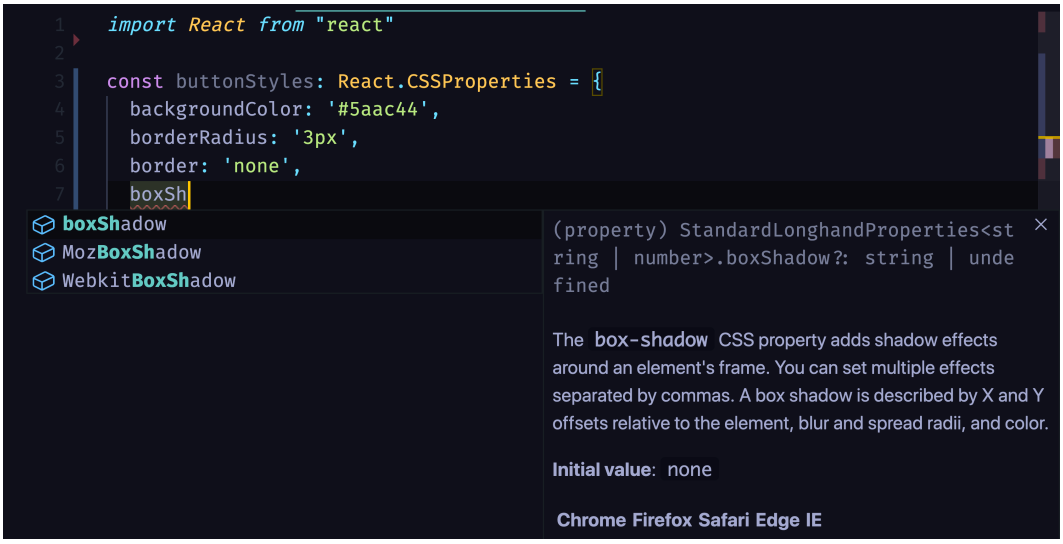
Another option is to pass an object with styling rules through `style` property. You can declare the object inline, then you won't need to specify type for it:

```
<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 auto-completion hints for CSS property names.



TypeScript provides nice CSS autocompletion

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

Using External Styling Libraries

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

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

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

```
import styled from "styled-components"

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

Then you can use them as regular React components:

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

At the 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`:

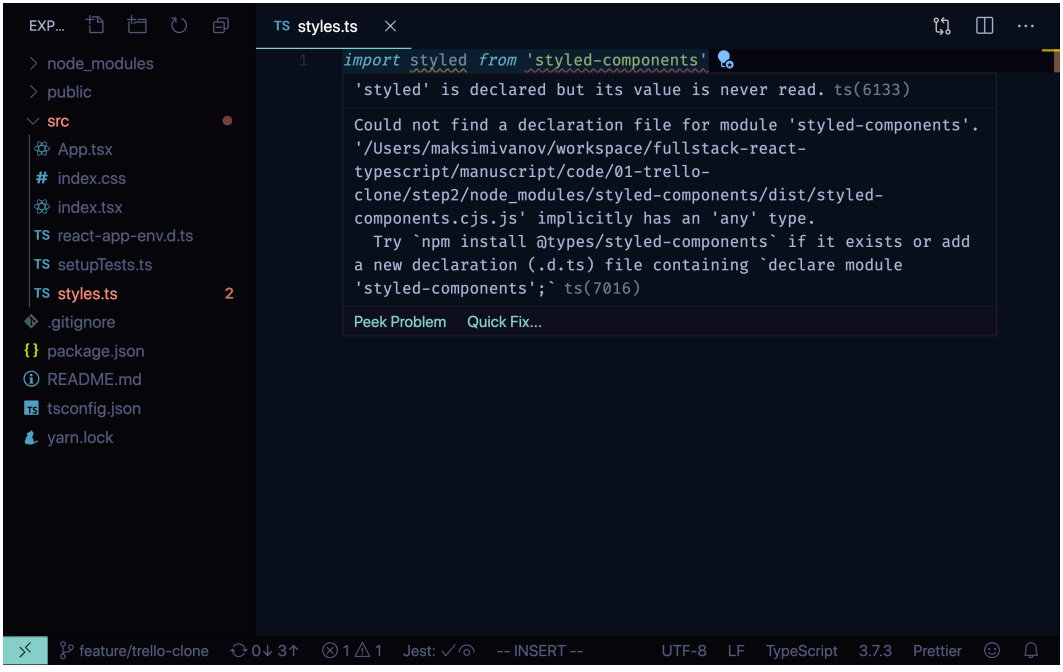
```
yarn add styled-components
```

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

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

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

You'll get a TypeScript error.



Missing @types for styled-components

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

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

Install the missing types:

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

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[44]. Make sure you know what props are, state is and how 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:

```
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`.

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

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/Column.tsx

```
import React from "react"
import { ColumnContainer, ColumnTitle } 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 `ColumnProps` 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[45] 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.

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.

Create The Card Component

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

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

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 with the field `text` of type `string` there.

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*.

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"
```

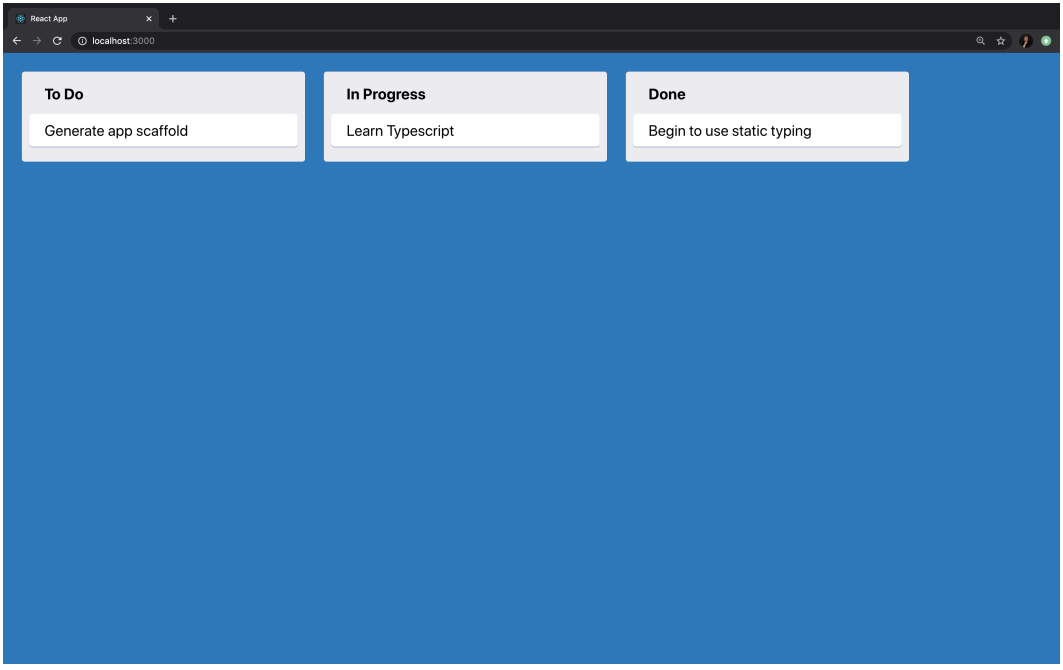
Now and change the layout code to this:

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:

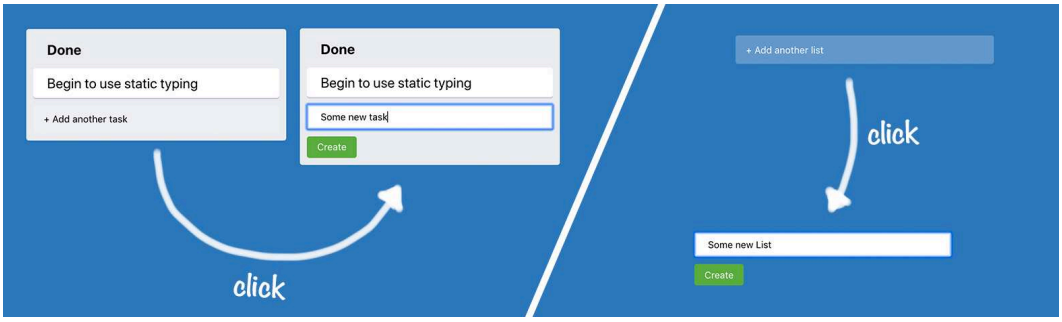


Rendering columns and cards

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

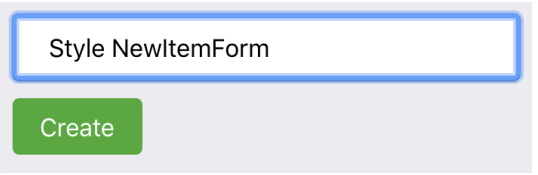
Now define the `AddNetItemButton` styled component:

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"
```

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

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

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

The component uses a controlled input we'll store the value for it in the `text` state. Whenever you type in the text inside this input - 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

Import `NewItemForm`:

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

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

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

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

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

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

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

Use AddNewItem Component

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

Adding New Lists

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

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

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

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

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

```
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

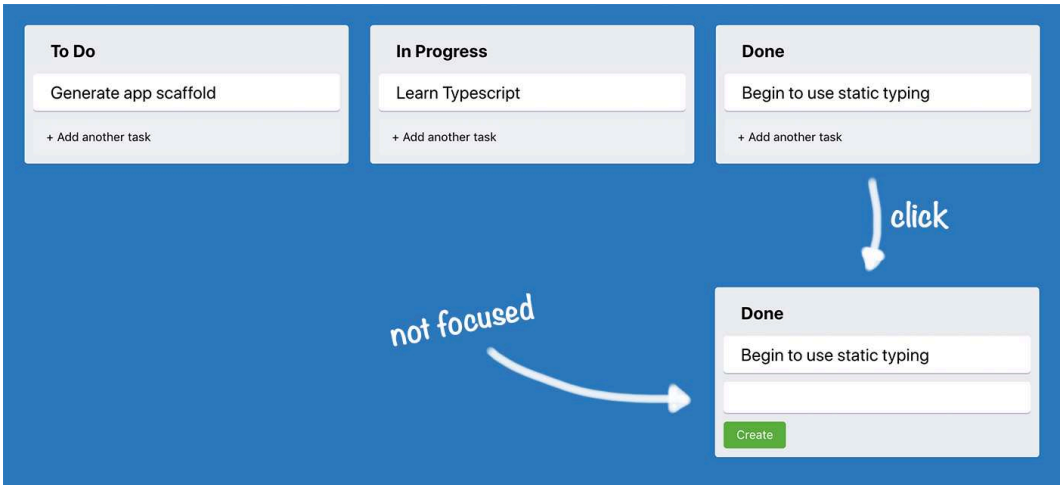
```
export const Column = ({
  text,
  children
}: React.PropsWithChildren<ColumnProps>) => {
  return (
    <ColumnContainer>
      <ColumnTitle>{text}</ColumnTitle>
      {children}
      <AddNewItem
        toggleButtonText="+ Add another task"
        onAdd={console.log}
        dark
      />
    </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[46] 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.

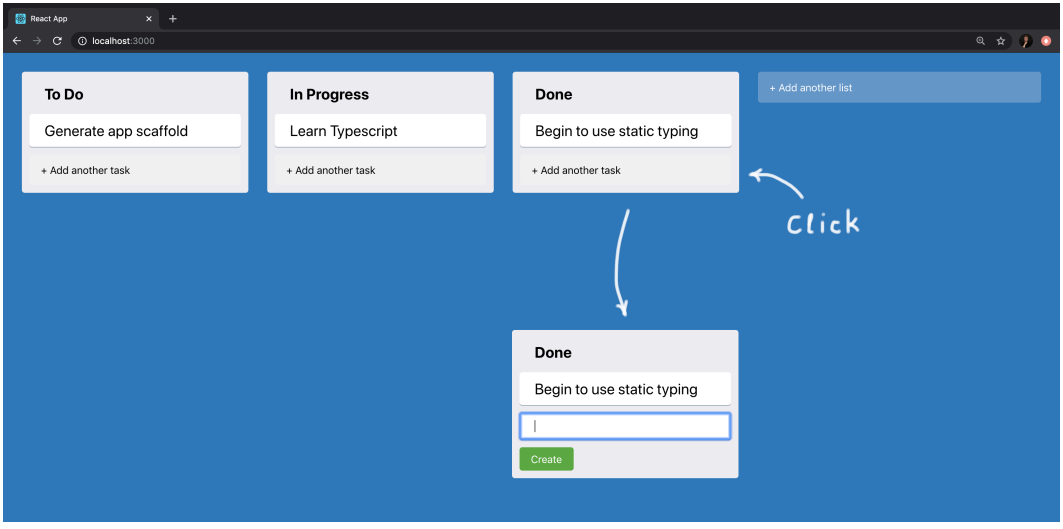
01-first-app/step2/src/NewItemForm.tsx

```
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>
  )
}
```

[46] 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

Requested Feature - Submit on Enter Press

Some readers requested the `NewItemForm` component to submit the input on `Enter` key press as well. So that the items could be created by pressing the `Enter` key instead of clicking the `Create` button. Let's implement it.

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

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

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

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

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

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

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

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

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

Add Global State And Business Logic

In this chapter add interactivity to our application.

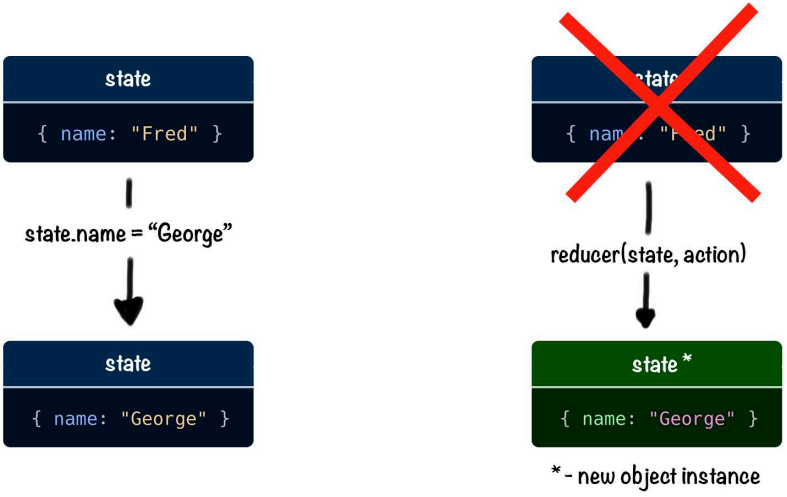
We'll implement drag-and-drop using the React DnD library. And we will add state management. We won't use any external framework like Redux or Mobx. Instead, we'll throw together a poor man's version of Redux using `useReducer` hook and React context API.

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.

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

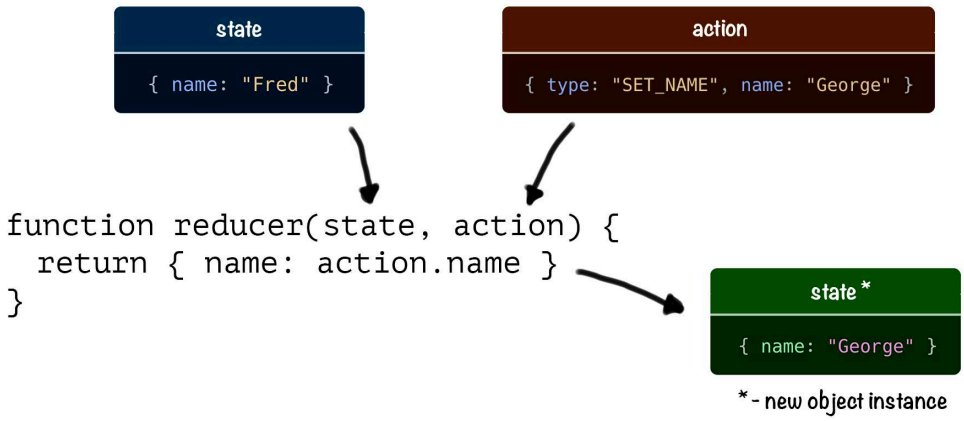


Instead of mutating the object we create a new instance

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

What Is a Reducer?

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



Reducer

Reducer must be a pure function. It means it shouldn't 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 the `React.PropsWithChildren` wrapping the `ColumnProps`. Add `index: number` field to the props:

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.tsx` 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; listId: string }
    }
```

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

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 add the dispatch method to the `AppStateContextProps` definition:

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

```
interface AppStateContextProps {
  state: AppState
  dispatch: React.Dispatch<Action>
}
```

Then 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 hard-coded `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: nanoid(), 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 nanoid[47] to generate new identifiers.

We need to install this library:

```
yarn add nanoid
```

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

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

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

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 its `id`. Let's add `findItemIndexById` method.

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

Define a new interface `Item`.

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

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

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

It will allow us to get the item position in the array knowing its `id`. We'll use it to find the index of the list we are going to modify.

Now we could already write the `ADD_TASK` handler like this:

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

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

      return {
        ...state
      }
    }
```

Where we would find the `targetLaneIndex` and then `push` the new task to it. But this would be wrong, as it mutates the original state object.

Read about the Immutable Update Patterns[48] to learn how to update the state in Redux without mutating it.

Instead of using `push` we will add the new task in two steps.

First we'll create a new list object with the new task added to it. Then we'll use a utility function `overrideItemAtIndex` to override the target list with its updated version.

Let's define the `overrideItemAtIndex` first. Still in `src/utils/arrayUtils.ts` add the following code:

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

```
export function overrideItemAtIndex<T>(
  array: T[],
  newItem: T,
  targetIndex: number
) {
  return array.map((item, index) => {
    if (index !== targetIndex) {
      return item
    }

    return newItem
  })
}
```

Here we defined a function that takes in `array`, `newItem` and `targetIndex` and then generates a new array where the object at index `targetIndex` is overriden with the `newItem` value.

We've used a type variable `T` to represent the type of the items in the array. In this example all the items in our array should have the same type.

[48] https://redux.js.org/recipes/structuring-reducers/immutable-update-patterns

To generate the updated array we use the method `map` that allows us to traverse the items in the array. This method can pass two arguments to the mapper function - the traversed item and its index. So we can check if the items `index` matches the `targetIndex` to return the `newItem`.

Now go back to `src/AppStateContext` and import the `findItemIndexById` and `overrideItemAtIn` functions.

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

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

Then add the code for the `ADD_TASK` block:

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

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

      const targetList = state.lists[targetListIndex]

      const updatedTargetList = {
        ...targetList,
        tasks: [
          ...targetList.tasks,
          { id: nanoid(), text: action.payload.text }
        ]
      }

      return {
        ...state,
        lists: overrideItemAtIndex(
```

```
          state.lists,
          updatedTargetList,
          targetListIndex
        )
      }
    }
```

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

Then we get the `targetList` and create a new object overriding the `tasks` field. We use the spread syntax to append the new task to the end of it.

And then we return the new state object where we override the `lists` field using our `overrideItemAtIndex` function. This function takes the `updatedTargetList` and puts it to the `targetListIndex` position.

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. First add the `id` field to the props:

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

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

Then update the component:

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, listId: id } })
        }
        dark
      />
    </ColumnContainer>
  )
}
```

Here we also call the `dispatch` function. We pass the `listId` 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

Open `src/AppStateContext.tsx`. Add a new action type to the `Action` union type:

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

```
  | {
      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

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

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.

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

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

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

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

Then we want to get the value at the `from` position and store it in the `item` constant.

Then we want to move the item without modifying the original array. So we use the `removeItemAtIndex` to remove the item from it's original position and then we call `insertItemAtIndex` on the resulting array.

Let's define the `removeItemAtIndex` first:

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

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

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

Go back to `src/AppStateContext.tsx` and import the `moveItem` function:

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

```
import { findItemIndexById, overrideItemAtIndex, moveItem } from "./uti\
ls/arrayUtils"
```

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 { HTML5Backend as 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.

First update the `AppState` type:

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

```
export interface AppState {
  lists: List[]
  draggedItem: DragItem | undefined;
}
```

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

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

```
  | {
      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

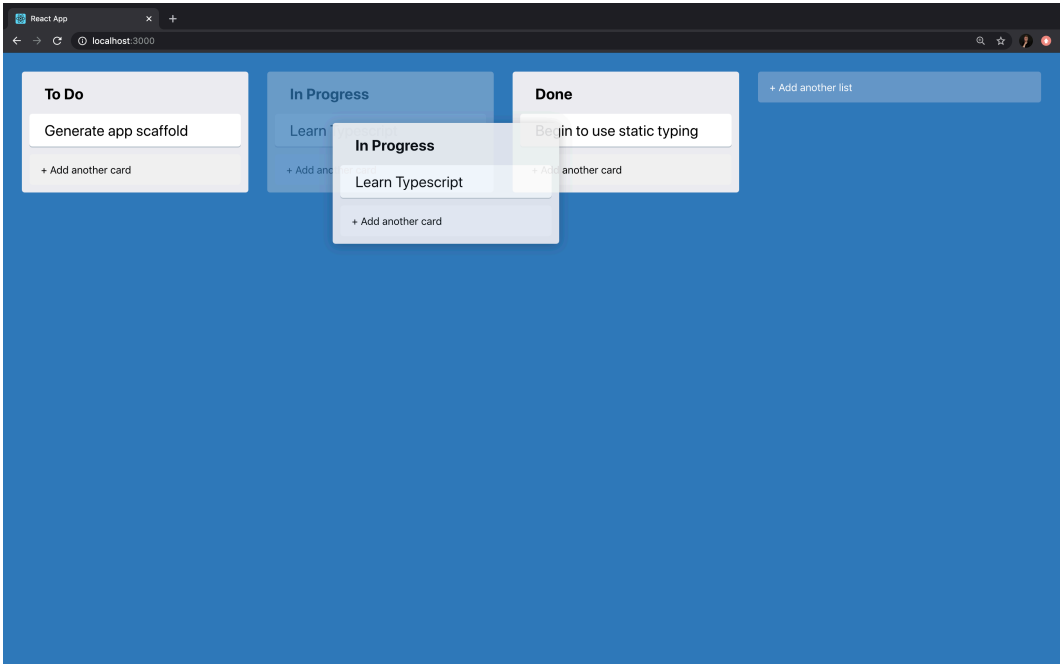
```
export const Column = ({ text, index, id }: ColumnProps) => {
  const { state, dispatch } = useAppState()
  const ref = useRef<HTMLDivElement>(null)

  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, listId: 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

```
import CustomDragLayer from "./CustomDragLayer"

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[49]

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

[49] ./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

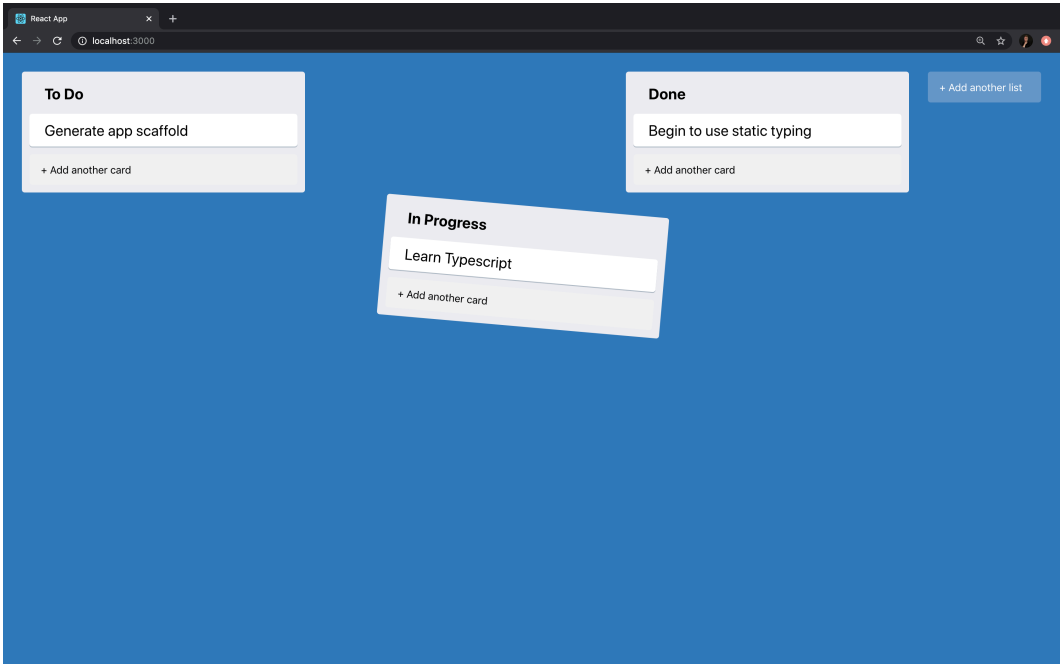
```
export const DragPreviewContainer = styled.div<DragPreviewContainerProp\
s>`
  transform: ${props => (props.isPreview ? "rotate(5deg)" : undefined)};
  opacity: ${props => (props.isHidden ? 0 : 1)};
`
```

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 CustomDragLayer

Open `src/CustomDragLayer` and add an early return if there is no dragged item:

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

```
  if (!isDragging) {
    return null
  }
```

Import the `Card` component:

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

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

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

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

```
  return (
    <CustomDragLayerContainer>
      <div style={getItemStyles(currentOffset)}>
        {item.type === "COLUMN" ? (
          <Column
            id={item.id}
            text={item.text}
            index={item.index}
            isPreview={true}
          />
        ) : (
          <Card
            columnId={item.columnId}
            isPreview={true}
            index={0}
            id={item.id}
            text={item.text}
          />
        )}
      </div>
    </CustomDragLayerContainer>
  )
```

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: "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 `MOVE_TASK` case block to our reducer:

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

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

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

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

```
      const {
        dragIndex,
        hoverIndex,
        sourceColumn,
        targetColumn
      } = action.payload
```

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

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

```
      const sourceListIndex = findItemIndexById(
        state.lists,
        sourceColumn
      )
  // ...
      const targetListIndex = findItemIndexById(
        state.lists,
        targetColumn
      )
```

Now that we have them we can move the task. We have to do it not mutating the original state, so we'll perform to separate steps. First we'll remove the task from the source list, and then add it to the target list.

Add this code to remove the moved task from the source list:

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

```
      const sourceList = state.lists[sourceListIndex]
      const task = sourceList.tasks[dragIndex]

      const updatedSourceList = {
        ...sourceList,
        tasks: removeItemAtIndex(sourceList.tasks, dragIndex)
      }

      const stateWithUpdatedSourceList = {
        ...state,
        lists: overrideItemAtIndex(
          state.lists,
          updatedSourceList,
          sourceListIndex
        )
      }
```

Here we get the source list from the `state.lists` array. Then we get the `task` reference.

Then we generate the `updatedSourceList` by using the `removeItemAtIndex` function we defined earlier. Don't forget to import it from `src/utils/arrayUtils.ts`.

After we have the `updatedSourceList` we can create a new version of `state` where we'll override the old source list with the updated one. We use the `overrideItemAtIndex` to do this, make sure you import it from `src/utils/arrayUtils.ts`.

Then we need to update the `targetList`:

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

```
      const targetList =
        stateWithUpdatedSourceList.lists[targetListIndex]

      const updatedTargetList = {
        ...targetList,
        tasks: insertItemAtIndex(targetList.tasks, task, hoverIndex)
      }
```

Here we get the target list just like we did it with the source list.

Then we generate the `updatedTargetList` by appending the new task to the `tasks` array of the `targetList`.

Finally we return a new state where we override the target list with the updated version:

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

```
import React, { createContext, useReducer, useContext } from "react"
import { nanoid } from "nanoid"
import { findItemIndexById, insertItemAtIndex, moveItem, overrideItemAt\
Index, removeItemAtIndex } from "./utils/arrayUtils"
import { DragItem } from "./DragItem"

interface Task {
  id: string
```

```
  text: string
}

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

export interface AppState {
  draggedItem: DragItem | undefined
  lists: List[]
}

type Action =
  | {
      type: "SET_DRAGGED_ITEM"
      payload: DragItem | undefined
    }
  | {
      type: "ADD_LIST"
      payload: string
    }
  | {
      type: "ADD_TASK"
      payload: { text: string; listId: string }
    }
  | {
      type: "MOVE_LIST"
      payload: {
        dragIndex: number
        hoverIndex: number
      }
    }
  | {
      type: "MOVE_TASK"
```

```
      payload: {
        dragIndex: number
        hoverIndex: number
        sourceColumn: string
        targetColumn: string
      }
    }

interface AppStateContextProps {
  state: AppState
  dispatch: React.Dispatch<Action>
}

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

const appStateReducer = (state: AppState, action: Action): AppState => {
  switch (action.type) {
    case "SET_DRAGGED_ITEM": {
      return { ...state, draggedItem: action.payload }
    }
    case "ADD_LIST": {
      return {
        ...state,
        lists: [
          ...state.lists,
          { id: nanoid(), text: action.payload, tasks: [] }
        ]
      }
    }
    case "ADD_TASK": {
      const targetListIndex = findItemIndexById(
        state.lists,
        action.payload.listId
      )
```

```
      const targetList = state.lists[targetListIndex]

      const updatedTargetList = {
        ...targetList,
        tasks: [
          ...targetList.tasks,
          { id: nanoid(), text: action.payload.text }
        ]
      }

      return {
        ...state,
        lists: overrideItemAtIndex(
          state.lists,
          updatedTargetList,
          targetListIndex
        )
      }
    }
    case "MOVE_LIST": {
      const { dragIndex, hoverIndex } = action.payload
      return {
        ...state,
        lists: moveItem(state.lists, dragIndex, hoverIndex)
      }
    }
    case "MOVE_TASK": {
      const {
        dragIndex,
        hoverIndex,
        sourceColumn,
        targetColumn
      } = action.payload

      const sourceListIndex = findItemIndexById(
```

```
    state.lists,
    sourceColumn
  )

  const targetListIndex = findItemIndexById(
    state.lists,
    targetColumn
  )

  const sourceList = state.lists[sourceListIndex]
  const task = sourceList.tasks[dragIndex]

  const updatedSourceList = {
    ...sourceList,
    tasks: removeItemAtIndex(sourceList.tasks, dragIndex)
  }

  const stateWithUpdatedSourceList = {
    ...state,
    lists: overrideItemAtIndex(
      state.lists,
      updatedSourceList,
      sourceListIndex
    )
  }

  const targetList =
    stateWithUpdatedSourceList.lists[targetListIndex]

  const updatedTargetList = {
    ...targetList,
    tasks: insertItemAtIndex(targetList.tasks, task, hoverIndex)
  }

  return {
    ...stateWithUpdatedSourceList,
```

```
        lists: overrideItemAtIndex(
          stateWithUpdatedSourceList.lists,
          updatedTargetList,
          targetListIndex
        )
      }
    }
    default: {
      return state
    }
  }
}

const appData: AppState = {
  draggedItem: undefined,
  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" }]
    }
  ]
}

export const AppStateProvider = ({ children }: React.PropsWithChildren<\
{}>) => {
```

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

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

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

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
    }
  })
```

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.

Import the `useItemDrag` hook:

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

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

Use it to get get the `drag` function. Add the following code:

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

```
  const { drag } = useItemDrag({ type: "CARD", id, index, text, columnI\
d })
```

Make sure to pass all the fields properly. We need to specify the `type` to be `CARD`.

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

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

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

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

Let's make it possible to move the cards to an empty column.

To implement this functionality we'll use columns as a drop target for our cards as well.

This way if the column is empty and we drag a card over it - the card will be moved to this empty column.

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

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

```
    accept: ["COLUMN", "CARD"],
```

Now inside of our `hover` callback, we'll need to check what 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/save
```

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/01-first-app/step9`, `cd` to this folder and launch the app:

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

Initially, you should see an empty field with the "+ Create new list" button.