Microsoft / TypeScript-React-Starter

Dismiss

Join GitHub today

GitHub is home to over 20 million developers working together to host and review code, manage projects, and build software together.

Sign up

A starter template for TypeScript and React with a detailed README describing how to use the two together.

19 commits | 3 branches | 0 releases | 9 contributors | MIT

Branch: master | New pull request | Find file | Clone or download

DanielRosenwasser committed on GitHub Merge pull request #16 from tugberkugurlu/update-package ... Latest commit 96fb623 on 17 May

public	initial commit	6 months ago
src	initial commit	6 months ago
.gitignore	initial commit	6 months ago
LICENSE	Initial commit	6 months ago
README-CRA.md	initial commit	6 months ago
README.md	Update README.md	6 months ago
package.json	correct the version of @types/react	6 months ago
tsconfig.json	initial commit	6 months ago
tslint.json	initial commit	6 months ago
yarn.lock	correct the version of @types/react	6 months ago

README.md

TypeScript React Starter

This quick start guide will teach you how to wire up TypeScript with React. By the end, you'll have

- a project with React and TypeScript
- linting with TSLint
- testing with Jest and Enzyme, and
- state management with Redux

We'll use the create-react-app tool to quickly get set up.

We assume that you're already using Node.js with npm. You may also want to get a sense of the basics with React.

Install create-react-app

We're going to use the create-react-app because it sets some useful tools and canonical defaults for React projects. This is just a command-line utility to scaffold out new React projects.

```
npm install -g create-react-app
```

Create our new project

We'll create a new project called `my-app`:

```
create-react-app my-app --scripts-version=react-scripts-ts
```

react-scripts-ts is a set of adjustments to take the standard create-react-app project pipeline and bring TypeScript into the mix.

At this point, your project layout should look like the following:

```
my-app/
├─ .gitignore
├─ node_modules/
├─ public/
├─ src/
│  └─ ...
├─ package.json
├─ tsconfig.json
└─ tslint.json
```

Of note:

- `tsconfig.json` contains TypeScript-specific options for our project.
- `tslint.json` stores the settings that our linter, TSLint, will use.
- `package.json` contains our dependencies, as well as some shortcuts for commands we'd like to run for testing, previewing, and deploying our app.
- `public` contains static assets like the HTML page we're planning to deploy to, or images. You can delete any file in this folder apart from `index.html`.
- `src` contains our TypeScript and CSS code. `index.tsx` is the entry-point for our file, and is mandatory.

Running the project

Running the project is as simple as running

```
npm run start
```

This runs the `start` script specified in our `package.json`, and will spawn off a server which reloads the page as we save our files. Typically the server runs at `http://localhost:3000`, but should be automatically opened for you.

This tightens the iteration loop by allowing us to quickly preview changes.

Testing the project

Testing is also just a command away:

```
npm run test
```

This command runs Jest, an incredibly useful testing utility, against all files whose extensions end in `.test.ts` or `.spec.ts`. Like with the `npm run start` command, Jest will automatically run as soon as it detects changes. If you'd like, you can run `npm run start` and `npm run test` side by side so that you can preview changes and test them simultaneously.

Creating a production build

When running the project with `npm run start`, we didn't end up with an optimized build. Typically, we want the code we ship to users to be as fast and small as possible. Certain optimizations like minification can accomplish this, but often take more time. We call builds like this "production" builds (as opposed to development builds).

To run a production build, just run

```
npm run build
```

This will create an optimized JS and CSS build in `./build/static/js` and `./build/static/css` respectively.

You won't need to run a production build most of the time, but it is useful if you need to measure things like the final size of your app.

Creating a component

We're going to write a `Hello` component. The component will take the name of whatever we want to greet (which we'll call `name`), and optionally the number of exclamation marks to trail with (`enthusiasmLevel`).

When we write something like `<Hello name="Daniel" enthusiasmLevel={3} />`, the component should render to something like `<div>Hello Daniel!!!</div>`. If `enthusiasmLevel` isn't specified, the component should default to showing one exclamation mark. If `enthusiasmLevel` is `0` or negative, it should throw an error.

We'll write a `Hello.tsx`:

```tsx
// src/components/Hello.tsx

import * as React from 'react';

export interface Props {
  name: string;
  enthusiasmLevel?: number;
}

function Hello({ name, enthusiasmLevel = 1 }: Props) {
  if (enthusiasmLevel <= 0) {
    throw new Error('You could be a little more enthusiastic. :D');
  }

  return (
    <div className="hello">
      <div className="greeting">
        Hello {name + getExclamationMarks(enthusiasmLevel)}
      </div>
    </div>
  );
}

export default Hello;

// helpers

function getExclamationMarks(numChars: number) {
  return Array(numChars + 1).join('!');
}
```

Notice that we defined a type named `Props` that specifies the properties our component will take. `name` is a required `string`, and `enthusiasmLevel` is an optional `number` (which you can tell from the `?` that we wrote out after its name).

We also wrote `Hello` as a stateless function component (an SFC). To be specific, `Hello` is a function that takes a `Props` object, and destructures it. If `enthusiasmLevel` isn't given in our `Props` object, it will default to `1`.

Writing functions is one of two primary ways React allows us to make components. If we wanted, we *could* have written it out as a class as follows:

```tsx
class Hello extends React.Component<Props, object> {
  render() {
    const { name, enthusiasmLevel = 1 } = this.props;

    if (enthusiasmLevel <= 0) {
      throw new Error('You could be a little more enthusiastic. :D');
    }

    return (
      <div className="hello">
        <div className="greeting">
          Hello {name + getExclamationMarks(enthusiasmLevel)}
        </div>
      </div>
    );
```

```
  }
}
```

Classes are useful when our component instances have some state. But we don't really need to think about state in this example - in fact, we specified it as `object` in `React.Component<Props, object>`, so writing an SFC tends to be shorter. Local component state is more useful at the presentational level when creating generic UI elements that can be shared between libraries. For our application's lifecycle, we will revisit how applications manage general state with Redux in a bit.

Now that we've written our component, let's dive into `index.tsx` and replace our render of `<App />` with a render of `<Hello ... />`.

First we'll import it at the top of the file:

```
import Hello from './components/Hello';
```

and then change up our `render` call:

```
ReactDOM.render(
  <Hello name="TypeScript" enthusiasmLevel={10} />,
  document.getElementById('root') as HTMLElement
);
```

Type assertions

One final thing we'll point out in this section is the line `document.getElementById('root') as HTMLElement`. This syntax is called a *type assertion*, sometimes also called a *cast*. This is a useful way of telling TypeScript what the real type of an expression is when you know better than the type checker.

The reason we need to do so in this case is that `getElementById`'s return type is `HTMLElement | null`. Put simply, `getElementById` returns `null` when it can't find an element with a given `id`. We're assuming that `getElementById` will actually succeed, so we need convince TypeScript of that using the `as` syntax.

TypeScript also has a trailing "bang" syntax (`!`), which removes `null` and `undefined` from the prior expression. So we *could* have written `document.getElementById('root')!`, but in this case we wanted to be a bit more explicit.

Adding style 😎

Styling a component with our setup is easy. To style our `Hello` component, we can create a CSS file at `src/components/Hello.css`.

```
.hello {
  text-align: center;
  margin: 20px;
  font-size: 48px;
  font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
}

.hello button {
  margin-left: 25px;
  margin-right: 25px;
  font-size: 40px;
  min-width: 50px;
}
```

The tools that create-react-app uses (namely, Webpack and various loaders) allow us to just import the stylesheets we're interested in. When our build runs, any imported `.css` files will be concatenated into an output file. So in `src/components/Hello.tsx`, we'll add the following import.

```
import './Hello.css';
```

Writing tests with Jest

We had a certain set of assumptions about our `Hello` component. Let's reiterate what they were:

> - When we write something like `<Hello name="Daniel" enthusiasmLevel={3} />`, the component should render to something like `<div>Hello Daniel!!!</div>`.
> - If `enthusiasmLevel` isn't specified, the component should default to showing one exclamation mark.
> - If `enthusiasmLevel` is `0` or negative, it should throw an error.

We can use these requirements to write a few tests for our components.

But first, let's install Enzyme. Enzyme is a common tool in the React ecosystem that makes it easier to write tests for how components will behave. By default, our application includes a library called jsdom to allow us to simulate the DOM and test its runtime behavior without a browser. Enzyme is similar, but builds on jsdom and makes it easier to make certain queries about our components.

Let's install it as a development-time dependency.

```
npm install -D enzyme @types/enzyme react-addons-test-utils
```

Notice we installed packages `enzyme` as well as `@types/enzyme`. The `enzyme` package refers to the package containing JavaScript code that actually gets run, while `@types/enzyme` is a package that contains declaration files (`.d.ts` files) so that TypeScript can understand how you can use Enzyme. You can learn more about `@types` packages here.

We also had to install `react-addons-test-utils`. This is something `enzyme` expects to be installed.

Now that we've got Enzyme set up, let's start writing our test! Let's create a file named `src/components/Hello.test.tsx`, adjacent to our `Hello.tsx` file from earlier.

```
// src/components/Hello.test.tsx

import * as React from 'react';
import * as enzyme from 'enzyme';
import Hello from './Hello';

it('renders the correct text when no enthusiasm level is given', () => {
  const hello = enzyme.shallow(<Hello name='Daniel' />);
  expect(hello.find(".greeting").text()).toEqual('Hello Daniel!')
});

it('renders the correct text with an explicit enthusiasm of 1', () => {
  const hello = enzyme.shallow(<Hello name='Daniel' enthusiasmLevel={1}/>);
  expect(hello.find(".greeting").text()).toEqual('Hello Daniel!')
});

it('renders the correct text with an explicit enthusiasm level of 5', () => {
  const hello = enzyme.shallow(<Hello name='Daniel' enthusiasmLevel={5} />);
  expect(hello.find(".greeting").text()).toEqual('Hello Daniel!!!!!');
});

it('throws when the enthusiasm level is 0', () => {
  expect(() => {
    enzyme.shallow(<Hello name='Daniel' enthusiasmLevel={0} />);
  }).toThrow();
});

it('throws when the enthusiasm level is negative', () => {
  expect(() => {
    enzyme.shallow(<Hello name='Daniel' enthusiasmLevel={-1} />);
  }).toThrow();
});
```

These tests are extremely basic, but you should be able to get the gist of things.

Adding state management

At this point, if all you're using React for is fetching data once and displaying it, you can consider yourself done. But if you're developing an app that's more interactive, then you may need to add state management.

State management in general

On its own, React is a useful library for creating composable views. However, React doesn't come with any facility for synchronizing data between your application. As far as a React component is concerned, data flows down through its children through the props you specify on each element.

Because React on its own does not provide built-in support for state management, the React community uses libraries like Redux and MobX.

Redux relies on synchronizing data through a centralized and immutable store of data, and updates to that data will trigger a re-render of our application. State is updated in an immutable fashion by sending explicit action messages which must be handled by functions called reducers. Because of the explicit nature, it is often easier to reason about how an action will affect the state of your program.

MobX relies on functional reactive patterns where state is wrapped through observables and passed through as props. Keeping state fully synchronized for any observers is done by simply marking state as observable. As a nice bonus, the library is already written in TypeScript.

There are various merits and tradeoffs to both. Generally Redux tends to see more widespread usage, so for the purposes of this tutorial, we'll focus on adding Redux; however, you should feel encouraged to explore both.

The following section may have a steep learning curve. We strongly suggest you familiarize yourself with Redux through its documentation.

Setting the stage for actions

It doesn't make sense to add Redux unless the state of our application changes. We need a source of actions that will trigger changes to take place. This can be a timer, or something in the UI like a button.

For our purposes, we're going to add two buttons to control the enthusiasm level for our `Hello` component.

Installing Redux

To add Redux, we'll first install `redux` and `react-redux`, as well as their types, as a dependency.

```
npm install -S redux react-redux @types/react-redux
```

In this case we didn't need to install `@types/redux` because Redux already comes with its own definition files (`.d.ts` files).

Defining our app's state

We need to define the shape of the state which Redux will store. For this, we can create a file called `src/types/index.tsx` which will contain definitions for types that we might use throughout the program.

```
// src/types/index.tsx

export interface StoreState {
    languageName: string;
    enthusiasmLevel: number;
}
```

Our intention is that `languageName` will be the programming language this app was written in (i.e. TypeScript or JavaScript) and `enthusiasmLevel` will vary. When we write our first container, we'll understand why we intentionally made our state slightly different from our props.

Adding actions

Let's start off by creating a set of message types that our app can respond to in `src/constants/index.tsx`.

```
// src/constants/index.tsx

export const INCREMENT_ENTHUSIASM = 'INCREMENT_ENTHUSIASM';
export type INCREMENT_ENTHUSIASM = typeof INCREMENT_ENTHUSIASM;
```

```
export const DECREMENT_ENTHUSIASM = 'DECREMENT_ENTHUSIASM';
export type DECREMENT_ENTHUSIASM = typeof DECREMENT_ENTHUSIASM;
```

This `const` / `type` pattern allows us to use TypeScript's string literal types in an easily accessible and refactorable way.

Next, we'll create a set of actions and functions that can create these actions in `src/actions/index.tsx`.

```
import * as constants from '../constants'

export interface IncrementEnthusiasm {
    type: constants.INCREMENT_ENTHUSIASM;
}

export interface DecrementEnthusiasm {
    type: constants.DECREMENT_ENTHUSIASM;
}

export type EnthusiasmAction = IncrementEnthusiasm | DecrementEnthusiasm;

export function incrementEnthusiasm(): IncrementEnthusiasm {
    return {
        type: constants.INCREMENT_ENTHUSIASM
    }
}

export function decrementEnthusiasm(): DecrementEnthusiasm {
    return {
        type: constants.DECREMENT_ENTHUSIASM
    }
}
```

We've created two types that describe what increment actions and decrement actions should look like. We also created a type (`EnthusiasmAction`) to describe cases where an action could be an increment or a decrement. Finally, we made two functions that actually manufacture the actions which we can use instead of writing out bulky object literals.

There's clearly boilerplate here, so you should feel free to look into libraries like redux-actions once you've got the hang of things.

Adding a reducer

We're ready to write our first reducer! Reducers are just functions that generate changes by creating modified copies of our application's state, but that have *no side effects*. In other words, they're what we call *pure functions*.

Our reducer will go under `src/reducers/index.tsx`. Its function will be to ensure that increments raise the enthusiasm level by 1, and that decrements reduce the enthusiasm level by 1, but that the level never falls below 1.

```
// src/reducers/index.tsx

import { EnthusiasmAction } from '../actions';
import { StoreState } from '../types/index';
import { INCREMENT_ENTHUSIASM, DECREMENT_ENTHUSIASM } from '../constants/index';

export function enthusiasm(state: StoreState, action: EnthusiasmAction): StoreState {
  switch (action.type) {
    case INCREMENT_ENTHUSIASM:
      return { ...state, enthusiasmLevel: state.enthusiasmLevel + 1 };
    case DECREMENT_ENTHUSIASM:
      return { ...state, enthusiasmLevel: Math.max(1, state.enthusiasmLevel - 1) };
  }
  return state;
}
```

Notice that we're using the *object spread* (`...state`) which allows us to create a shallow copy of our state, while replacing the `enthusiasmLevel`. It's important that the `enthusiasmLevel` property come last, since otherwise it would be overridden by the property in our old state.

You may want to write a few tests for your reducer. Since reducers are pure functions, they can be passed arbitrary data. For every input, reducers can tested by checking their newly produced state. Consider looking into Jest's toEqual method to accomplish this.

Making a container

When writing with Redux, we will often write components as well as containers. Components are often data-agnostic, and work mostly at a presentational level. *Containers* typically wrap components and feed them any data that is necessary to display and modify state. You can read more about this concept on Dan Abramov's article *Presentational and Container Components*.

First let's update `src/components/Hello.tsx` so that it can modify state. We'll add two optional callback properties to `Props` named `onIncrement` and `onDecrement`:

```
export interface Props {
  name: string;
  enthusiasmLevel?: number;
  onIncrement?: () => void;
  onDecrement?: () => void;
}
```

Then we'll bind those callbacks to two new buttons that we'll add into our component.

```
function Hello({ name, enthusiasmLevel = 1, onIncrement, onDecrement }: Props) {
  if (enthusiasmLevel <= 0) {
    throw new Error('You could be a little more enthusiastic. :D');
  }

  return (
    <div className="hello">
      <div className="greeting">
        Hello {name + getExclamationMarks(enthusiasmLevel)}
      </div>
      <div>
        <button onClick={onDecrement}>-</button>
        <button onClick={onIncrement}>+</button>
      </div>
    </div>
  );
}
```

In general, it'd be a good idea to write a few tests for `onIncrement` and `onDecrement` being triggered when their respective buttons are clicked. Give it a shot to get the hang of writing tests for your components.

Now that our component is updated, we're ready to wrap it into a container. Let's create a file named `src/containers/Hello.tsx` and start off with the following imports.

```
import Hello from '../components/Hello';
import * as actions from '../actions/';
import { StoreState } from '../types/index';
import { connect, Dispatch } from 'react-redux';
```

The real two key pieces here are the original `Hello` component as well as the `connect` function from react-redux. `connect` will be able to actually take our original `Hello` component and turn it into a container using two functions:

- `mapStateToProps` which massages the data from the current store to part of the shape that our component needs.
- `mapDispatchToProps` which creates callback props to pump actions to our store using a given `dispatch` function.

If we recall, our application state consists of two properties: `languageName` and `enthusiasmLevel`. Our `Hello` component, on the other hand, expected a `name` and an `enthusiasmLevel`. `mapStateToProps` will get the relevant data from the store, and adjust it if necessary, for our component's props. Let's go ahead and write that.

```
export function mapStateToProps({ enthusiasmLevel, languageName }: StoreState) {
  return {
    enthusiasmLevel,
    name: languageName,
  }
}
```

Note that `mapStateToProps` only creates 2 out of 4 of the properties a `Hello` component expects. Namely, we still want to pass in the `onIncrement` and `onDecrement` callbacks. `mapDispatchToProps` is a function that takes a dispatcher function. This dispatcher function can pass actions into our store to make updates, so we can create a pair of callbacks that will call the dispatcher as necessary.

```
export function mapDispatchToProps(dispatch: Dispatch<actions.EnthusiasmAction>) {
  return {
    onIncrement: () => dispatch(actions.incrementEnthusiasm()),
    onDecrement: () => dispatch(actions.decrementEnthusiasm()),
  }
}
```

Finally, we're ready to call `connect`. `connect` will first take `mapStateToProps` and `mapDispatchToProps`, and then return another function that we can use to wrap our component. Our resulting container is defined with the following line of code:

```
export default connect(mapStateToProps, mapDispatchToProps)(Hello);
```

When we're finished, our file should look like this:

```
// src/containers/Hello.tsx

import Hello from '../components/Hello';
import * as actions from '../actions/';
import { StoreState } from '../types/index';
import { connect, Dispatch } from 'react-redux';

export function mapStateToProps({ enthusiasmLevel, languageName }: StoreState) {
  return {
    enthusiasmLevel,
    name: languageName,
  }
}

export function mapDispatchToProps(dispatch: Dispatch<actions.EnthusiasmAction>) {
  return {
    onIncrement: () => dispatch(actions.incrementEnthusiasm()),
    onDecrement: () => dispatch(actions.decrementEnthusiasm()),
  }
}

export default connect(mapStateToProps, mapDispatchToProps)(Hello);
```

Creating a store

Let's go back to `src/index.tsx`. To put this all together, we need to create a store with an initial state, and set it up with all of our reducers.

```
import { createStore } from 'redux';
import { enthusiasm } from './reducers/index';
import { StoreState } from './types/index';

const store = createStore<StoreState>(enthusiasm, {
  enthusiasmLevel: 1,
  languageName: 'TypeScript',
});
```

`store` is, as you might've guessed, our central store for our application's global state.

Next, we're going to swap our use of `./src/components/Hello` with `./src/containers/Hello` and use react-redux's `Provider` to wire up our props with our container. We'll import each:

```
import Hello from './containers/Hello';
import { Provider } from 'react-redux';
```

and pass our `store` through to the `Provider`'s attributes:

```
ReactDOM.render(
  <Provider store={store}>
    <Hello />
  </Provider>,
  document.getElementById('root') as HTMLElement
);
```

Notice that `Hello` no longer needs props, since we used our `connect` function to adapt our application's state for our wrapped `Hello` component's props.

Ejecting

If at any point, you feel like there are certain customizations that the create-react-app setup has made difficult, you can always opt-out and get the various configuration options you need. For example, if you'd like to add a Webpack plugin, it might be necessary to take advantage of the "eject" functionality that create-react-app provides.

Simply run

```
npm run eject
```

and you should be good to go!

As a heads up, you may want to commit all your work before running an eject. You cannot undo an eject command, so opting out is permanent unless you can recover from a commit prior to running an eject.

Next steps

create-react-app comes with a lot of great stuff. Much of it is documented in the default `README.md` that was generated for our project, so give that a quick read.

If you still want to learn more about Redux, you can check out the official website for documentation. The same goes for MobX.

If you want to eject at some point, you may need to know a little bit more about Webpack. You can check out our React & Webpack walkthrough here.

At some point you might need routing. There are several solutons, but react-router is probably the most popular for Redux projects, and is often used in conjunction with react-router-redux.