3

Setting Up React and TypeScript

In this chapter, we will learn how to use both React and TypeScript together. We will start by going through the steps for creating a React and TypeScript project using a tool called webpack. Then we will create another project, but this time using a tool called Create React App to show you how to speed up the process of creating a React and TypeScript project.

This chapter will then cover how to use TypeScript to make React props and states type-safe, extending the alert component built in the first chapter. Lastly, we will learn how to debug your app with React's DevTools.

In this chapter, we'll cover the following topics:

- Creating a project with webpack
- Creating a project with Create React App
- Creating a React and TypeScript component

Technical requirements

We will use the following technologies in this chapter:

- **Node.js** and **npm**: React and TypeScript are dependent on these. You can install them from `https://nodejs.org/en/download/`.
- **Visual Studio Code**: We'll use this editor to write code and execute terminal commands. You can install it from `https://code.visualstudio.com/`.

All the code snippets in this chapter can be found online at `https://github.com/PacktPublishing/Learn-React-with-TypeScript-2nd-Edition/tree/main/Chapter3`.

Creating a project with webpack

Setting up a React and TypeScript project is tricky because both JSX and TypeScript code needs to be transpiled into JavaScript. In this section, we will cover how to set up a React and TypeScript project step by step, with the help of a tool called webpack.

Introducing webpack

Webpack is a tool that bundles JavaScript source code files together. It can also bundle CSS and images. It can run other tools such as Babel to transpile React and the TypeScript type checker as it scans the files. It is a mature and incredibly popular tool used in the React community that powers many React projects.

Webpack is incredibly flexible but, unfortunately, it requires a lot of configuration. We will witness this as we create our project with webpack.

It is important to understand that webpack isn't a project creation tool. For example, it won't install React or TypeScript – we have to do that separately. Instead, webpack brings tools such as React and TypeScript together once installed and configured. So, we won't use webpack until later in this section.

Creating the folder structure

We will start by creating a simple folder structure for the project. The structure will separate the project's configuration files from the source code. Carry out the following steps to do this:

1. Open Visual Studio Code in the folder where you want the project to be.
2. In the **Explorer** panel, create a folder called `src`. A folder can be created by right-clicking in the **Explorer** panel and choosing **New Folder**. Note that `src` is short for source code.

So, the `src` folder will hold all the source code for the app. The project configuration files will be placed at the root of the project.

Next, we will define the critical information about the project.

Creating package.json

The `package.json` file defines our project name, description, npm scripts, dependent npm modules, and much more.

Create a `package.json` file at the root of the project with the following content:

```
{
  "name": "my-app",
  "description": "My React and TypeScript app",
```

```
  "version": "0.0.1"
}
```

This file contains minimal information at the moment. However, it will eventually contain other details, such as React and TypeScript as the app's dependencies.

> **Note**
>
> More information can be found on `package.json` at the following link: `https://docs.npmjs.com/cli/v8/configuring-npm/package-json`.

Next, we will add the web page that will host the React app.

Adding a web page

An HTML page is going to host the app. In the `src` folder, create a file called `index.html` with the following content:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>My app</title>
  </head>
  <body>
    <div id="root"></div>
  </body>
</html>
```

This code snippet can be copied and pasted from `https://github.com/PacktPublishing/Learn-React-with-TypeScript-2nd-Edition/blob/main/Chapter3/Section1-Creating-a-project-with-Webpack/src/index.html`.

The React app will be injected into the `div` element with an `id` attribute value of `"root"`. We will cover the injection of the React app in a later section, *Adding React*.

Adding TypeScript

Next, we will install TypeScript into the project. To do this, carry out the following steps:

1. Start by opening the Visual Studio Code terminal by opening the **Terminal** menu and clicking **New Terminal**.

2. We know from the last chapter that using `npm install` without specifying any options will install the dependencies listed inside `package.json`. The `install` command has options for the specific packages to be installed that aren't in `package.json` yet. Execute the following command in the terminal to install `typescript`:

```
npm install --save-dev typescript
```

We have also included a `--save-dev` option to specify that `typescript` should be installed as a **development-only** dependency. This is because TypeScript is only required during development and not at runtime.

3. After the command has finished, open `package.json`. You will see that `typescript` is now listed as a development dependency in the `devDependencies` section:

```
{
  "name": "my-app",
  "description": "My React and TypeScript app",
  "version": "0.0.1",
  "devDependencies": {
    "typescript": "^4.6.4"
  }
}
```

Note that the version (`4.6.4`) of `typescript` in the preceding code snippet will probably be different in your example. This is because `npm install` installs the latest version of the dependency unless a version is specified in the command.

4. Next, we will create a TypeScript configuration file. Note that we aren't going to configure TypeScript to do any transpilation – we will use Babel for that, which is covered later. So, the TypeScript configuration will be focused on type checking.

To do this, create a file called `tsconfig.json` in the `root` folder and enter the following content into it:

```
{
  "compilerOptions": {
    "noEmit": true,
    "lib": [
      "dom",
      "dom.iterable",
      "esnext"
    ],
    "moduleResolution": "node",
```

```
    "allowSyntheticDefaultImports": true,
    "esModuleInterop": true,
    "jsx": "react",
    "forceConsistentCasingInFileNames": true,
    "strict": true
  },
  "include": ["src"],
  "exclude": ["node_modules", "dist"]
}
```

This code snippet can be copied and pasted from `https://github.com/PacktPublishing/Learn-React-with-TypeScript-2nd-Edition/blob/main/Chapter3/Section1-Creating-a-project-with-Webpack/tsconfig.json`.

Here is an explanation of each setting that has been configured that wasn't explained in the last chapter:

- Setting `noEmit` to `true` suppresses the TypeScript compiler from doing any transpilation.
- Setting `allowSyntheticDefaultImports` and `esModuleInterop` to `true` allows React to be imported as a default import, like the following:

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

  - Without these settings set to `true`, React would have to be imported like this:

```
import * as React from 'react'
```

- Setting `forceConsistentCasingInFileNames` to `true` enables the type-checking process to check the casing of referenced filenames in import statements are consistent.

> **Note**
>
> For more information on the TypeScript compiler options, see the following link: `https://www.typescriptlang.org/docs/handbook/compiler-options.html`.

Adding React

Next, we will install React and its TypeScript types into the project. We will then add a React root component. To do this, carry out the following steps:

1. Execute the following command in the terminal to install React:

```
npm install react react-dom
```

Reacts comes in two libraries:

- The core library is called `react`, which is used in all variants of React.
- The specific React variant, which is the variant used to build web apps, is called `react-dom`. An example of a different variant would be the React Native variant used to build mobile apps.

2. React doesn't include TypeScript types – instead, they are in a separate npm package. Let's install these now:

```
npm install --save-dev @types/react @types/react-dom
```

3. The root component will be in a file called `index.tsx` in the `src` folder. Create this file with the following content:

```
import React, { StrictMode } from 'react';
import { createRoot } from 'react-dom/client';

const root = createRoot(
  document.getElementById('root') as HTMLElement
);

function App() {
  return <h1>My React and TypeScript App!</h1>;
}

root.render(
  <StrictMode>
    <App />
  </StrictMode>
);
```

This code snippet can be copied and pasted from `https://github.com/PacktPublishing/Learn-React-with-TypeScript-2nd-Edition/blob/main/Chapter3/Section1-Creating-a-project-with-Webpack/src/index.tsx`.

The structure of this file is similar to the `index.js` file in the alert component project from the first chapter. It injects the React app into a DOM element with an `id` of `'root'`. The app is straightforward – it displays a heading called **My React and TypeScript App!**.

Notice that the file extension for `index` is `.tsx` rather than `.js`. This allows Babel and TypeScript to detect TypeScript files containing JSX in the transpilation and type-checking processes. A `.ts` extension can be used for TypeScript code that doesn't contain any JSX.

Also, notice `as HTMLElement` in the call to `createRoot`:

```
const root = createRoot(
  document.getElementById('root') as HTMLElement
);
```

This is called a **type assertion**, which tells TypeScript what the type should be. Without the type assertion, TypeScript will infer the type as `HTMLElement | null` because `document.getElementById` may not find an element and return `null`. However, we are confident that the element will be found because we specified it in the `index.html` file, so it is safe to narrow the type to `HTMLElement` using a type assertion.

React is now installed in the project, and the project also contains a simple React app.

Adding Babel

As mentioned earlier, Babel will transpile both React and TypeScript code into JavaScript in this project. Carry out the following steps to install and configure Babel:

1. Start by installing the core Babel library using the following command in Visual Studio Code's terminal:

   ```
   npm install --save-dev @babel/core
   ```

   Babel is installed as a development dependency because it is only needed during development to transpile code and not when the app runs.

   A shortened version of this command is as follows:

   ```
   npm i -D @babel/core
   ```

   `i` is short for `install`, and `-D` is short for `--save-dev`.

2. Next, install a Babel plugin called `@babel/preset-env` that allows the latest JavaScript features to be used:

   ```
   npm i -D @babel/preset-env
   ```

3. Now, install a Babel plugin called `@babel/preset-react` that enables React code to be transformed into JavaScript:

   ```
   npm i -D @babel/preset-react
   ```

4. Similarly, install a Babel plugin called `@babel/preset-typescript` that enables TypeScript code to be transformed into JavaScript:

```
npm i -D @babel/preset-typescript
```

5. The last two plugins to install allow the use of the async and await features in JavaScript:

```
npm i -D @babel/plugin-transform-runtime @babel/runtime
```

6. Babel can be configured in a file called `.babelrc.json`. Create this file at the root of the project with the following content:

```
{
  "presets": [
    "@babel/preset-env",
    "@babel/preset-react",
    "@babel/preset-typescript"
  ],
  "plugins": [
    [
      "@babel/plugin-transform-runtime",
      {
        "regenerator": true
      }
    ]
  ]
}
```

This code snippet can be found at `https://github.com/PacktPublishing/Learn-React-with-TypeScript-2nd-Edition/blob/main/Chapter3/Section1-Creating-a-project-with-Webpack/.babelrc.json`.

The preceding configuration tells Babel to use the installed plugins. Babel is now installed and configured.

> **Note**
> For more information about Babel, see the following link: `https://babeljs.io/`.

Next, we will glue everything together with webpack.

Adding webpack

Webpack is a popular tool that primarily bundles JavaScript source code files together. It can run other tools, such as Babel, as it scans the files. So, we will use webpack to scan all the source files and transpile them into JavaScript. The output from the webpack process will be a single JavaScript bundle that is referenced in `index.html`.

Installing webpack

Carry out the following steps to install webpack and its associated libraries:

1. Start by installing webpack using the following command in the terminal in Visual Studio Code:

   ```
   npm i -D webpack webpack-cli
   ```

   This installs the core webpack library as well as its command-line interface.

   Webpack has TypeScript types in the `webpack` package, so we don't need to install them separately.

2. Next, run the following command to install webpack's development server:

   ```
   npm i -D webpack-dev-server
   ```

   Webpack's development server is used during developments to host the web app and automatically updates as changes are made to the code.

3. A webpack plugin is required to allow Babel to transpile the React and TypeScript code into JavaScript. This plugin is called `babel-loader`. Install this using the following command:

   ```
   npm i -D babel-loader
   ```

4. Webpack can create the `index.html` file that hosts the React app. We want webpack to use the `index.html` file in the `src` folder as a template and add the React app's bundle to it. A plugin called `html-webpack-plugin` is capable of doing this. Install this plugin using the following command:

   ```
   npm i -D html-webpack-plugin
   ```

Webpack and its associated libraries are now installed.

Configuring webpack

Next, we will configure webpack to do everything we need. Separate configurations for development and production can be created because the requirements are slightly different. However, we will focus on a configuration for development in this chapter. Carry out the following steps to configure webpack:

1. First, install a library called `ts-node`, which allows the configuration to be defined in a TypeScript file:

   ```
   npm i -D ts-node
   ```

2. Now, we can add the development configuration file. Create a file called `webpack.dev.config.ts` in the project root. The code to go in this file is lengthy and can be copied and pasted from the following link: `https://github.com/PacktPublishing/Learn-React-with-TypeScript-2nd-Edition/blob/main/Chapter3/Section1-Creating-a-project-with-Webpack/webpack.dev.config.ts`. In the following steps, we will explain the different parts of this file.

3. The configure file starts with various import statements and a type for the configuration object:

```
import path from 'path';
import HtmlWebpackPlugin from 'html-webpack-plugin';
import {
  Configuration as WebpackConfig,
  HotModuleReplacementPlugin,
} from 'webpack';
import {
  Configuration as WebpackDevServerConfig
} from 'webpack-dev-server';

type Configuration = WebpackConfig & {
  devServer?: WebpackDevServerConfig;
}
```

Let's review the most important points:

- The `path` node library will tell webpack where to place the bundle.
- `HtmlWebpackPlugin` will be used to create `index.html`.
- The webpack configuration TypeScript types come from both the `webpack` and `webpack-dev-server` packages. So, we combine them using an intersect type, creating a type called `Configuration`.

4. The configuration object is then defined as follows:

```
const config: Configuration = {
  mode: 'development',
  output: {
    publicPath: '/',
  },
  entry: './src/index.tsx',
  ...
```

```
};

export default config;
```

Let's review the most important points here:

- The `mode` property tells webpack the configuration is for development, meaning that the React development tools are included in the bundle
- The `output.publicPath` property is the root path in the app, which is important for deep linking in the dev server to work correctly
- The `entry` property tells webpack where the React app's entry point is, which is `index.tsx` in our project
- Webpack expects the configuration object to be a default export, so we export the `config` object as a default export

5. Further configuration is highlighted by the following code:

```
const config: Configuration = {
  ...,
  module: {
    rules: [
      {
        test: /\.(ts|js)x?$/i,
        exclude: /node_modules/,
        use: {
          loader: 'babel-loader',
          options: {
            presets: ['@babel/preset-env', '@babel/
              preset-react', '@babel/preset-typescript'],
          },
        },
      },
    ],
  },
  resolve: {
    extensions: ['.tsx', '.ts', '.js'],
  }
};
```

The `module` property informs webpack how different modules should be processed. We need to tell webpack to use `babel-loader` for files with `.js`, `.ts`, and `.tsx` extensions.

The `resolve.extensions` property tells webpack to look for TypeScript files and JavaScript files during module resolution.

6. Next, a couple of plugins are defined:

```
const config: Configuration = {
  ...,
  plugins: [
    new HtmlWebpackPlugin({
      template: 'src/index.html',
    }),
    new HotModuleReplacementPlugin(),
  ]
};
```

As mentioned earlier, `HtmlWebpackPlugin` creates the HTML file. It has been configured to use `index.html` in the `src` folder as a template.

`HotModuleReplacementPlugin` allows modules to be updated while an application is running, without a full reload.

7. Lastly, the following properties complete the configuration:

```
const config: Configuration = {
  ...,
  devtool: 'inline-source-map',
  devServer: {
    static: path.join(__dirname, 'dist'),
    historyApiFallback: true,
    port: 4000,
    open: true,
    hot: true,
  }
};
```

The `devtool` property tells webpack to use full inline source maps, which allow the original source code to be debugged before transpilation.

The `devServer` property configures the webpack development server. It configures the web server root to be the `dist` folder and to serve files on port `4000`. Now, `historyApiFallback` is required for deep links to work, and we have also specified to open the browser after the server has been started.

The development configuration is now complete. But before we try to run the app in development mode, we need to create an npm script to run webpack.

8. First, open `package.json` and add a `scripts` section with a `start` script:

```
{
  ...,
  "scripts": {
    "start": "webpack serve --config webpack.dev.config.
      ts"
  }
}
```

9. Now, we can run the app in development mode by running the following command in the terminal:

```
npm run start
```

The `npm run` command executes a script in the `scripts` section of `package.json`. The `start` script is commonly used to run a program in development mode. It is so common that npm recognizes this without the `run` part. So, the command can be shortened to the following:

```
npm start
```

After a few seconds, the app opens in the default browser:

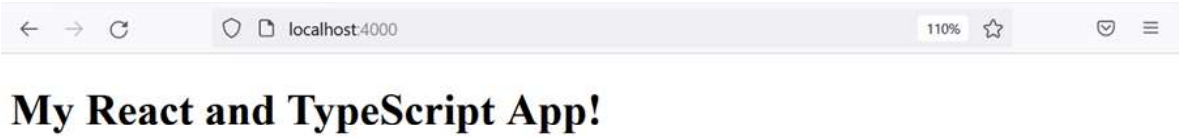

Figure 3.1 – The app running in a browser in development mode

10. Leave the app running and open the `index.tsx` file. Change the content of the `h1` element to something slightly different:

```
function App() {
  return <h1>My Super React and TypeScript App!</h1>;
}
```

When the file is saved, notice that the running app automatically refreshes:

Figure 3.2 – The app is automatically refreshed

That completes the setup of the React and TypeScript project using webpack. Here's a recap of the key points of the setup:

- An HTML file is required to host the React app
- Webpack transpiles the app's React and TypeScript code into JavaScript with the help of Babel and then references it in the HTML file
- Webpack has a development server that automatically refreshes the app as we write code

> **Note**
> More information about webpack is available at the following link: `https://webpack.js.org/`.

That was an incredible amount of work to set up a React and TypeScript app, and it only does a fraction of what we will need to build a real app. For example, CSS can't be used, and the setup doesn't support unit testing. Luckily, there is a much easier way to create React and TypeScript projects, which we will learn in the next section. What you have learned in this section is really important though, because the tool we are about to use also uses webpack under the hood.

Creating a project with Create React App

Create React App is a popular tool for creating React projects. It is based on webpack, so the knowledge from the last section will give you an understanding of how Create React App works. In this section, we will use Create React App to create a React and TypeScript project.

Using Create React App

Unlike the setup in the last section, Create React App generates a React and TypeScript project with all the common tools we will likely require, including CSS and unit testing support.

To use Create React App, open Visual Studio Code in a blank folder of your choice and run the following command:

```
npx create-react-app myapp --template typescript
```

`npx` allows npm packages to temporarily be installed and run. It is a common method of running project scaffolding tools such as Create React App.

`create-react-app` is the package for the Create React App tool that creates the project. We have passed it the app name, `myapp`. We have also specified that the `typescript` template should be used to create the project.

It takes a minute or so for the project to be created, but this is much quicker than creating it manually with webpack!

When the command has finished creating the project, reopen the project in Visual Studio Code in the `myapp` folder. Note that your directory might be slightly different if you called the app something different. It is important to be in the app name folder; otherwise, the dependencies may be installed in the wrong location.

Next, we will understand what linting is and add an extension for it into Visual Studio Code.

Adding linting to Visual Studio Code

Linting is the process of checking code for potential problems. It is common practice to use linting tools to catch problems early in the development process as code is written. **ESLint** is a popular tool that can lint React and TypeScript code. Fortunately, Create React App has already installed and configured ESLint in our project.

Editors such as Visual Studio Code can be integrated with ESLint to highlight potential problems. Carry out the following steps to install an ESLint extension into Visual Studio Code:

1. Open up the **Extensions** area in Visual Studio Code. The **Extensions** option is in the **Preferences** menu in the **File** menu on Windows *or* in the **Preferences** menu in the **Code** menu on a Mac.
2. A list of extensions will appear on the left-hand side and the search box above the extensions list can be used to find a particular extension. Enter `eslint` into the extensions list search box.

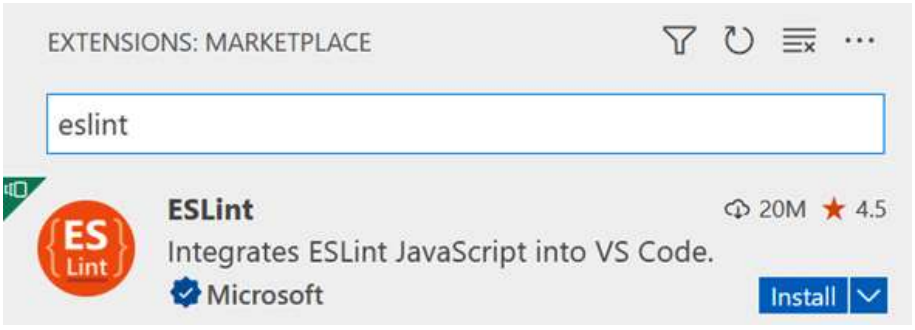

Figure 3.3 – The Visual Studio Code ESLint extension

An extension by Microsoft called ESLint should appear at the top of the list.

3. Click the **Install** button to install the extension.
4. Now, we need to make sure the ESLint extension is configured to check React and TypeScript. So, open the **Settings** area in Visual Studio Code. The **Settings** option is in the **Preferences** menu in the **File** menu on Windows or in the **Preferences** menu in the **Code** menu on a Mac.
5. In the settings search box, enter `eslint: probe` and select the **Workspace** tab:

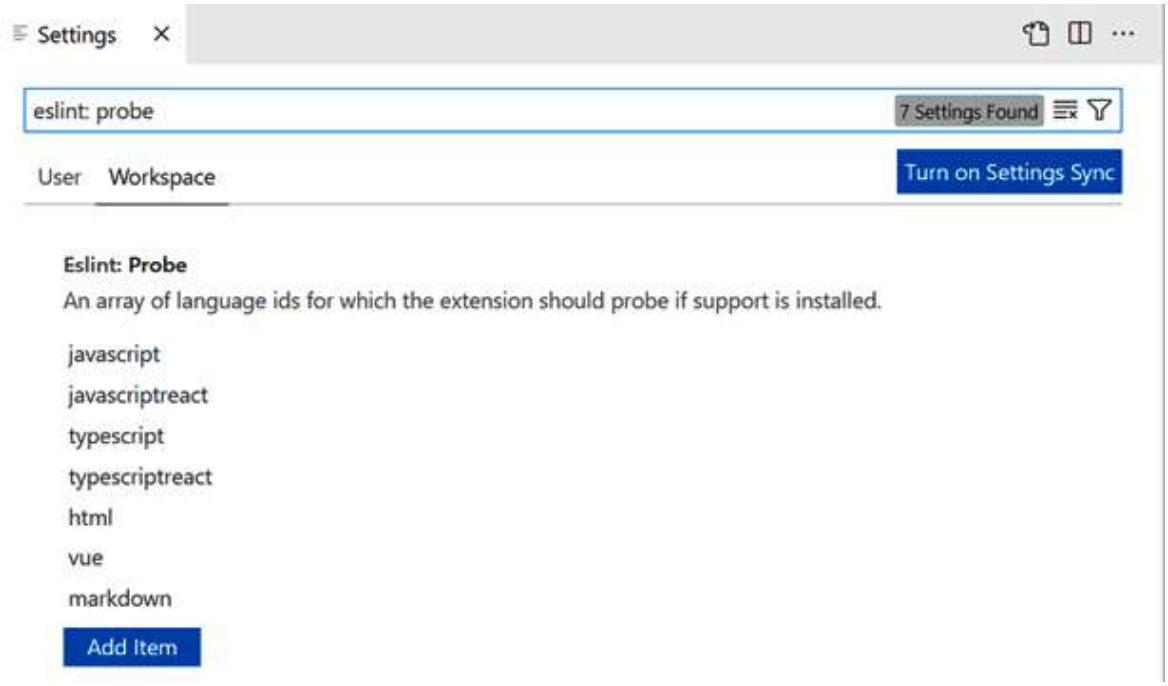

Figure 3.4 – Visual Studio Code ESLint Probe settings

   This setting defines the languages to use when ESLint checks code.
6. Make sure that **typescript** and **typescriptreact** are on the list. If not, add them using the **Add Item** button.

The ESLint extension for Visual Studio Code is now installed and configured in the project.

> **Note**
> For more information about ESLint, see the following link: `https://eslint.org/`.

Next, we will add automatic code formatting to the project.

Adding code formatting

The next tool we will set up automatically formats code. Automatic code formatting ensures code is consistently formatted, which helps its readability. Having consistently formatted code also helps developers see the important changes in a code review – rather than differences in formatting.

Prettier is a popular tool capable of formatting React and TypeScript code. Unfortunately, Create React App doesn't install and configure it for us. Carry out the following steps to install and configure Prettier in the project:

1. Install Prettier using the following command in the terminal in Visual Studio Code:

```
npm i -D prettier
```

   Prettier is installed as a development dependency because it is only used during development time and not at runtime.

2. Prettier has overlapping style rules with ESLint, so install the following two libraries to allow Prettier to take responsibility for the styling rules from ESLint:

```
npm i -D eslint-config-prettier eslint-plugin-prettier
```

   `eslint-config-prettier` disables conflicting ESLint rules, and `eslint-plugin-prettier` is an ESLint rule that formats code using Prettier.

3. The ESLint configuration needs to be updated to allow Prettier to manage the styling rules. Create React App allows ESLint configuration overrides in an `eslintConfig` section in `package.json`. Add the Prettier rules to the `eslintConfig` section in `package.json` as follows:

```
{
  ...,
  "eslintConfig": {
    "extends": [
      "react-app",
      "react-app/jest",
      "plugin:prettier/recommended"
    ]
  },
  ...
}
```

4. Prettier can be configured in a file called `.prettierrc.json`. Create this file with the following content in the root folder:

```
{
  "printWidth": 100,
  "singleQuote": true,
  "semi": true,
  "tabWidth": 2,
  "trailingComma": "all",
  "endOfLine": "auto"
}
```

We have specified the following:

- Lines wrap at 100 characters
- String qualifiers are single quotes
- Semicolons are placed at the end of statements
- The indentation level is two spaces
- A trailing comma is added to multi-line arrays and objects
- Existing line endings are maintained

> **Note**
> More information on the configuration options can be found at the following link: `https://prettier.io/docs/en/options.html`.

Prettier is now installed and configured in the project.

Visual Studio Code can integrate with Prettier to automatically format code when source files are saved. So, let's install a Prettier extension into Visual Studio Code:

1. Open the **Extensions** area in Visual Studio Code and enter `prettier` into the extensions list search box. An extension called **Prettier – Code formatter** should appear at the top of the list:

Figure 3.5 – The Visual Studio Code Prettier extension

2. Click the **Install** button to install the extension.
3. Next, open the **Settings** area in Visual Studio Code. Select the **Workspace** tab and make sure the **Format On Save** option is ticked:

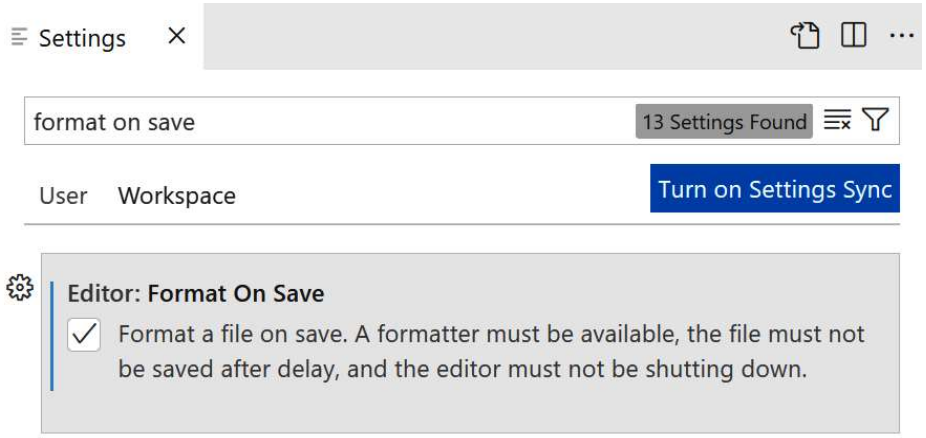

Figure 3.6 – Visual Studio Code "Format On Save" setting

This setting tells Visual Studio Code to automatically format code in files that are saved.

4. There is one more setting to set. This is the default formatter that Visual Studio Code should use to format code. Click the **Workspace** tab and make sure **Default Formatter** is set to **Prettier - Code formatter**:

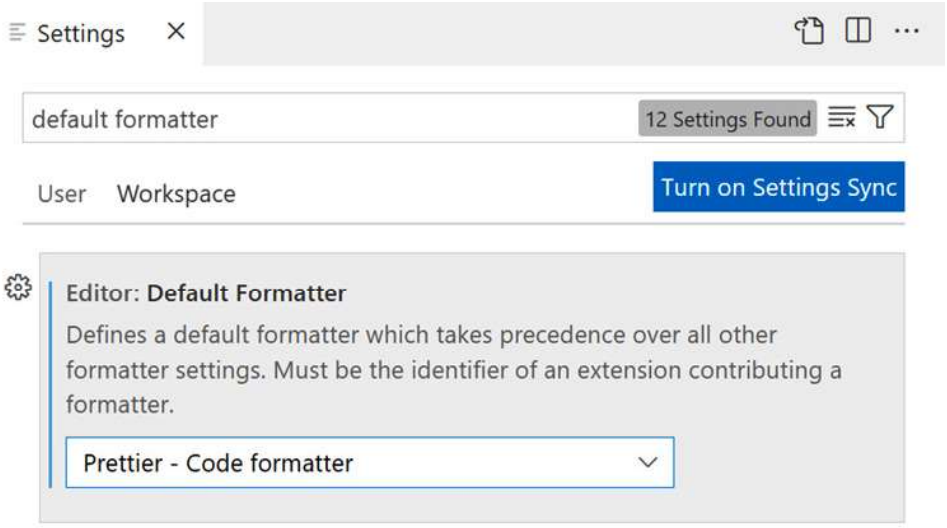

Figure 3.7 – Setting Default Formatter to Prettier - Code formatter

The Prettier extension for Visual Studio Code is now installed and configured in the project. Next, we will run the app in development mode.

Starting the app in development mode

Carry out the following steps to start the app in development mode:

1. Create React App has already created an npm script called `start`, which runs the app in development mode. Run this script in the terminal as follows:

```
npm start
```

After a few seconds, the app will appear in the default browser:

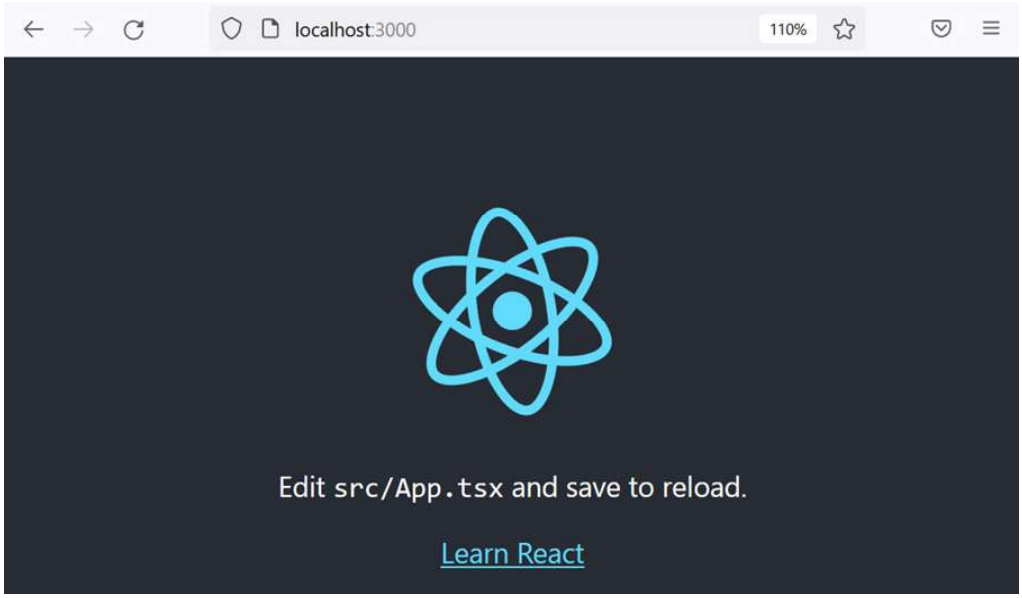

Figure 3.8 – The React app running in development mode

If your app reports Prettier formatting issues, open the file in question and save it. This will correctly format the file and resolve the errors.

2. Open `App.tsx` and change the **Learn React** link to **Learn React and TypeScript**:

```
<a
  className="App-link"
  href="https://reactjs.org"
  target="_blank"
  rel="noopener noreferrer"
>
  Learn React and TypeScript
</a>;
```

After the file is saved, the running app is automatically refreshed with the updated link text:

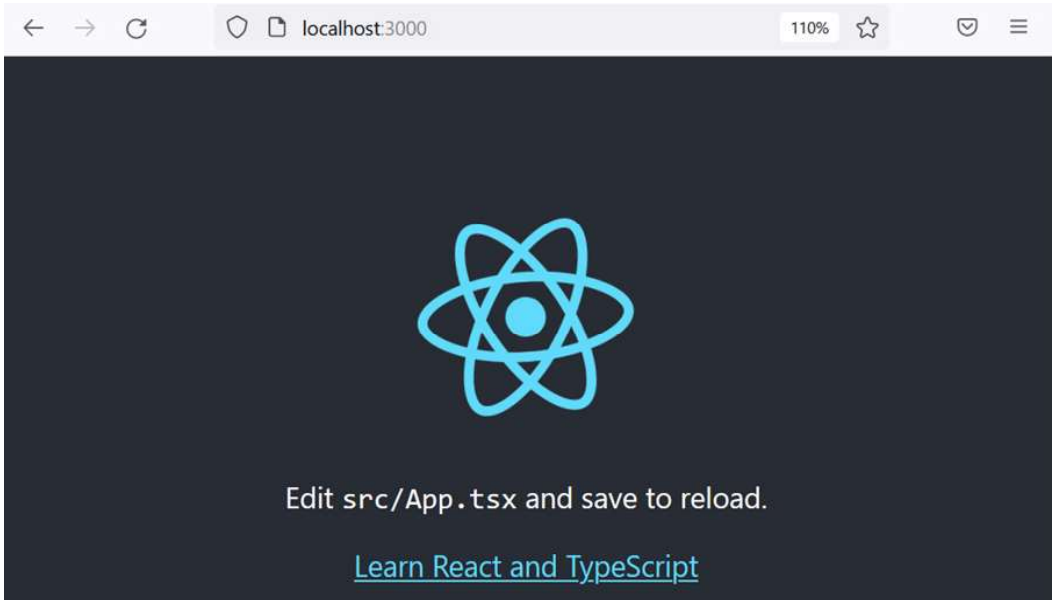

Figure 3.9 – The updated React app

3. Experiment with the code in `App.tsx`. Add a used variable and pass an invalid prop in a JSX element:

```
function App() {
  const unused = 'something';
  return (
    <div className="App" invalidProp="something">
      ...
    </div>
  );
}
```

As expected, the problems are caught and reported in Visual Studio Code. The issues are also reported in the browser.

Figure 3.10 – Problems caught by Visual Studio Code

4. Remove the invalid code and stop the app from running before continuing. The shortcut key for stopping the app is *Ctrl* + *C*.

We have now seen how Create React App provides a productive development experience. Next, we will produce a production build.

Producing a production build

Carry out the following steps to produce a build of the app that can be deployed into production:

1. Create React App has already created an npm script called `build` that produces all the artifacts for deployment to production. Run this script in the terminal as follows:

```
npm run build
```

After a few seconds, the deployment artifacts are placed in a `build` folder.

2. Open the `build` folder – it contains many files. The root file is `index.html`, which references the other JavaScript, CSS, and image files. All the files are optimized for production with whitespace removed and the JavaScript minified.

This completes the production build and the React and TypeScript project set up with Create React App. Here's a recap of the key points of the setup:

- The `npx` tool can execute the Create React App library, specifying the `typescript` template to create a React and TypeScript project.

- Create React App sets up many useful project features, such as linting, CSS support, and SVG support.
- Create React App also sets up npm scripts to run the app in development mode and produce a production build.
- One feature that Create React App doesn't set up is automatic code formatting. However, Prettier can be installed and configured manually to provide this capability.

Keep this project safe because we will continue to use it in the next section.

Next, we will learn how to create a React component that uses TypeScript for type checking.

Creating a React and TypeScript component

In *Chapter 1, Introducing React*, we built an alert component using React. In this section, we will use TypeScript to make the component strongly typed and experience the benefits. We start by adding a type to the alert component's props and then experiment with defining a type for its state. After completing the alert component, we will inspect the component using React's DevTools.

Adding a props type

We will continue using the React and TypeScript project created in the last section with Create React App. Take the following steps to add a strongly typed version of the alert component:

1. Open the project in Visual Studio Code if it isn't already open. Make sure you open the project in the app name folder so that `package.json` is at the root.
2. Create a new file in the `src` folder called `Alert.tsx`. Paste in the JavaScript version of the alert component, which can be found on GitHub at `https://github.com/PacktPublishing/Learn-React-with-TypeScript-2nd-Edition/blob/main/Chapter1/Section7-Using-events/Alert.js`.

   Notice that type errors are reported on some of the props because they can only be inferred as having the `any` type.
3. Add the following type just above the component. This will be the type for the component props:

```
type Props = {
  type?: string;
  heading: string;
  children: React.ReactNode;
  closable?: boolean;
  onClose?: () => void;
};
```

The `heading` and `children` props are required but the rest of the props are optional.

The `children` prop is given a special type called `React.ReactNode`. This allows it to accept JSX elements as well as strings.

The name of the type can be anything, but it is common practice to call it `Props`.

Remember from the *Creating interfaces* section in *Chapter 2, Introducing TypeScript*, that the interface syntax can be used to create types as an alternative to type aliases. An interface version of the `Props` type is as follows:

```
interface Props {
  type?: string;
  heading: string;
  children: React.ReactNode;
  closable?: boolean;
  onClose?: () => void;
}
```

As mentioned in the last chapter, it is largely personal preference whether you choose type aliases or interfaces for the type of component props.

4. Now, assign the `Props` type to the alert component after the destructed parameters:

```
export function Alert({
  type = "information",
  heading,
  children,
  closable,
  onClose,
}: Props) {
  ...
}
```

The alert props are now strongly typed.

5. Open `App.tsx` and replace the header element with the alert component. Don't forget to import the alert component before using it in the JSX. Don't pass any props into `Alert` to test the type checking:

```
import React from 'react';
import './App.css';
import { Alert } from './Alert';
```

```
function App() {
  return (
    <div className="App">
      <Alert />
    </div>
  );
}

export default App;
```

As expected, a type error is raised on `Alert`:

Figure 3.11 – Type error on the Alert component

6. Pass in a `header` prop to `Alert`, and give it some content:

```
<Alert heading="Success">Everything is really good!</
Alert>
```

The type errors will disappear.

7. Start the app in development mode if not already running (`npm start`). After that, the app component appears on the page as expected.

Next, we will learn how to explicitly give the React component state a type.

Adding a state type

Follow these steps to experiment with the `visible` state type in the alert component:

1. Open `Alert.tsx` and hover over the `visible` state variable to determine its inferred type. It has been inferred to be `boolean` because it has been initialized with the `true` value. The `boolean` type is precisely what we want.

2. As an experiment, remove the initial value of `true` passed into `useState`. Then, hover over the `visible` state variable again. It has been inferred to be `undefined` because no default value has been passed into `useState`. This obviously isn't the type we want.
3. Sometimes, the `useState` type isn't inferred to be the type we want, like in the previous step. In these cases, the type of the state can be explicitly defined using a **generic argument** on `useState`. Explicitly give the `visible` state a `boolean` type by adding the following generic argument:

```
const [visible, setVisible] = useState<boolean>();
```

> **Note**
>
> A generic argument is like a regular function argument but defines a type for the function. A generic argument is specified using angled brackets after the function name.

4. Restore the `useState` statement to what it originally was, with it initialized as `true` and no explicit type:

```
const [visible, setVisible] = useState(true);
```

5. Stop the app from running by pressing *Ctrl* + *C*.

In summary, always check the inferred state type from `useState` and use its generic argument to explicitly define the type if the inferred type is not what is required.

Next, we will learn how to use the React browser development tools.

Using React DevTools

React DevTools is a browser extension available for Chrome and Firefox. The tools allow React apps to be inspected and debugged. The links to the extensions are as follows:

- Chrome: `https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi/`
- Firefox: `https://addons.mozilla.org/en-GB/firefox/addon/react-devtools/`

To install the extension, click the **Add to Chrome** or **Add to Firefox** button. You will need to reopen the browser for the tools to be available.

Carry out the following steps to explore the tools:

1. In Visual Studio Code, start the app in development mode by running `npm start` in a terminal. After a few seconds, the app will appear in the browser.
2. Open the browser's development tools by pressing *F12*. React DevTools adds two panels called **Components** and **Profiler**.
3. First, we will explore the **Components** panel, so select this panel:

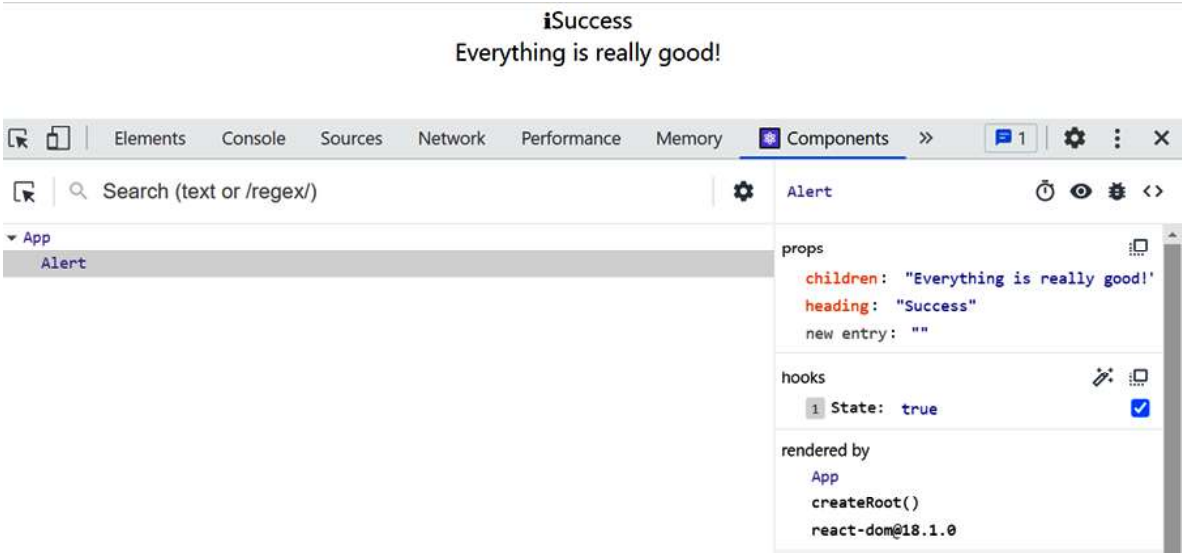

Figure 3.12 – The React DevTools Components panel

The React component tree appears on the left-hand side. Selecting a React component reveals the current props and state values on the right-hand side.

4. Notice that the state isn't named – it has a generic name, **State**. Click the wand icon at the right of the **hooks** section.

   The name of the state now appears in brackets:

Figure 3.13 – State variable name after wand is clicked

5. In Visual Studio Code, open `App.tsx` and pass the `closable` prop to `Alert`:

```
<Alert heading="Success" closable>
  Everything is really good!
</Alert>
```

The app refreshes and the close button appears.

6. Click the close button and notice that the `visible` state changes to `false` in DevTools.

   The **Components** panel is useful when debugging a large component tree to quickly understand the values of the props and state.

7. Refresh the browser so that the alert appears again. While still in the **Components** panel in React DevTools, open the settings by clicking the cog icon. Tick the **Highlight updates when components render.** option in the **General** section, if not already ticked.

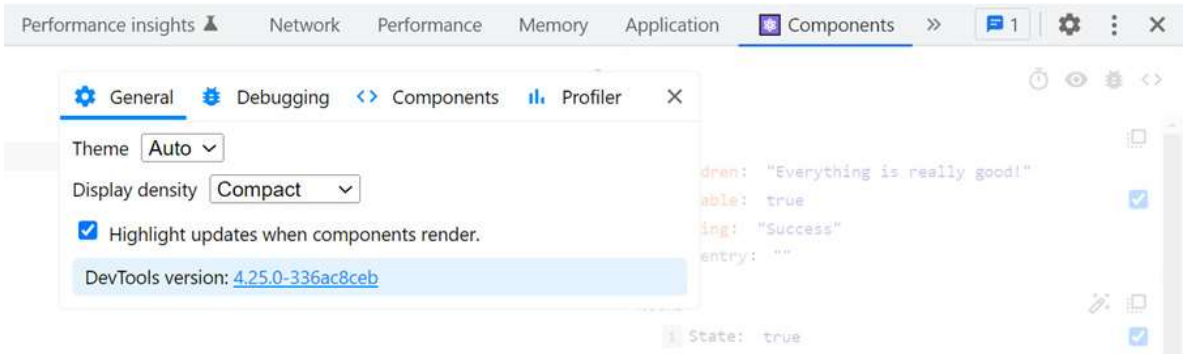

Figure 3.14 – Re-render highlight option

This option will highlight when a component is re-rendered.

8. Before we try the re-render highlight, open `Alert.tsx` and update it to render **Gone!** when the `visible` state is `false`:

```
if (!visible) {
  return <div>Gone!</div>;
}
```

Before this change, the re-render highlight would have no element to highlight.

9. Now, click the close button on the alert in the browser. The re-rendered alert component will be highlighted with a green border:

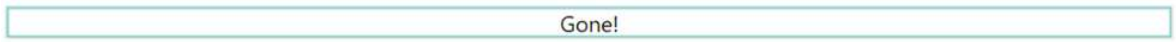

Figure 3.15 – Re-render highlight

This completes our exploration of the **Components** panel. Press *F5* to refresh the browser so that the alert component reappears before continuing.

10. Now, we will explore the **Profiler** panel, so select this panel. This tool allows interactions to be profiled, which is useful for tracking performance problems.

11. Click the **Start profiling** option, which is the blue circle icon.
12. Click the close button in the alert.
13. Click the **Stop profiling** option, which is the red circle icon. A timeline appears of all the component re-renders:

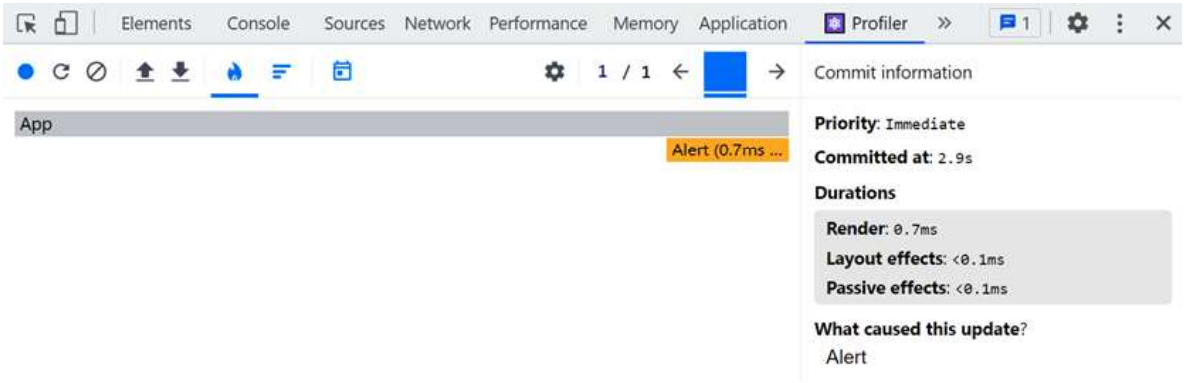

Figure 3.16 – React DevTools components

This shows that `Alert` was re-rendered when the close button was clicked, taking 0.7 milliseconds.

This tool is helpful in quickly spotting the slow components for a particular user interaction.

14. This completes our exploration of React DevTools. Undo the change we made to the `Alert` component so that it renders `null` again when invisible:

```
if (!visible) {
  return null;
}
```

That completes this section. Here's a recap:

- A type can be added to component props to make them type-safe
- A state can be inferred from its initial value but can be explicitly defined using a generic argument on `useState`
- React DevTools can be installed in a browser to inspect the component tree in a running app and help track down performance problems

That brings us to the end of the chapter.

Summary

We started the chapter by creating a React and TypeScript project with the help of webpack and Babel. Lots of steps were involved and we only set up a fraction of what we would need for a real project. For example, we have not set up the ability to produce an optimized bundle for production.

We moved on to use Create React App for creating a React and TypeScript project. We saw how this approach is a much faster and more thorough way of creating a project. Then, we used ESLint for linting and Prettier for automatic code formatting. As a result of that exercise, we now understand how TypeScript, ESLint, and Prettier can be used together to create a high-quality React and TypeScript project environment.

In the final section, we learned how to create React components with strongly typed props and states. We experienced how this helps to catch problems quickly. We also played with React's DevTools, which allows a React component tree to be inspected and the performance profiled.

In the next chapter, we will learn about React's common hooks.

Questions

The following questions will check what you have learned in this chapter:

1. What type will the `name` prop have in the following component, which has no type annotation?

```
export function Name({ name }) {
  return <span>{name}</span>;
}
```

2. What type will the `firstName` state have in the following `useState` statement?

```
const [firstName, setFirstName] = useState("");
```

3. A `ContactDetails` component has the following type for its props:

```
type Props = {
  firstName?: string;
  email: string;
};
export function ContactDetails({ firstName, email }:
Props) {
  ...
}
```

The preceding component is referenced in another component's JSX as follows:

```
<ContactDetails email="fred@somewhere.com" />
```

Will a type error be raised?

4. A `status` state variable can hold the `"Good"` and `"Bad"` values and is initially `"Good"`. It is defined in the following code:

```
const [status, setStatus] = useState("Good");
```

What is the type given to `status`? How can its type be narrowed to only `"Good"` or `"Bad"`?

5. A `FruitList` component takes in an array of fruit names and displays them in a list. It is referenced in another component's JSX as follows:

```
<FruitList fruits={["Banana", "Apple", "Strawberry"]} />;
```

What type would you define for the `FruitList` component?

6. An `email` state variable can hold `null` or an email address and is initially `null`. How would you define this state using the `useState` hook?

7. The following component allows the user to agree:

```
export function Agree({ onAgree }: Props) {
  return (
    <button onClick={() => onAgree()}>
      Click to agree
    </button>
  );
}
```

What would you define as the type definition for `Props`?

Answers

Here are the answers to the questions in the preceding section.

1. The `name` prop will have the `any` type.
2. The `firstName` state will be given the `string` type because `string` will be inferred from the initial value `""`.
3. There will be no type error even though `firstName` is not passed because it is defined as optional.

4. The inferred type of `status` is `string`. An explicit type can be defined for the state using a generic type argument as follows:

```
const [status, setStatus] = useState<'Good' |
'Bad'>('Good');
```

5. The type for the `FruitList` component could be as follows:

```
type Props = {
  fruits: string[];
}
```

Alternatively, it could be defined using an interface as follows:

```
interface Props {
  fruits: string[];
}
```

6. The `email` state could be defined as follows:

```
const [email, setEmail] = useState<string | null>(null);
```

An explicit type needs to be defined; otherwise, an initial value of `null` would give `email` a type of `null`.

7. The type for `Props` could be as follows:

```
type Props = {
  onAgree: () => void;
};
```

Alternatively, it could be defined using an interface as follows:

```
interface Props {
  onAgree: () => void;
}
```