Lab 1: Create a New React App

Goals:

- Create a new React app by using **create-vite**
- Open the app's code in **Visual Studio Code**
- Use the **development server** to test the app

Step by step instructions:

1) Open a command prompt and navigate to a directory where you would like the new React app to be created (a new subdirectory for the app will be created automatically).

2) Run the command to create a new app using Vite.

```
npm create vite@latest
```

3) If told that an additional package needs to be installed, press **enter** to acknowledge that it is okay to proceed.

4) For the questions asked by create-vite:
   - Project name: **license-plate-store**
   - Select a framework: **React**
   - Select a variant: **TypeScript**
   - Use rolldown-vite: **No**
   - Install with npm and start now? **No**

5) Once the creation process is complete, change to the newly created **directory** and **install** the app's package dependencies.

```
cd license-plate-store
npm install
```

6) Move (or copy) the **lab-snippets** folder into the **license-plate-store** directory. This will make it easier to copy code that will be needed later.

7) Open the project folder in **Visual Studio Code** (you can open Visual Studio Code and then select [**File > Open Folder…**] from the main menu or you can use the Visual Studio Code command-line tool to open the current directory by typing "`code .`")

8) If not already visible in Visual Studio Code, open the embedded **terminal** by selecting [**Terminal > New Terminal**] from the main menu.

9) In the embedded terminal, execute the command to start the **development server**.
   ```
   npm run dev
   ```

10) In the console, type [**h + enter]** to see the available commands.

11) Execute the command to open the app in a **broswer**. You should see a simple page with the text "Vite + React".

    Note: You can leave the development server running and most changes to the code will be reflected automatically. For larger changes, you may sometimes need to restart the development server.

Lab 2: Dynamic Content

Goals:

- Modify the app to display a JavaScript **constant**
- Display the **current time**

Step by step instructions:

1) Comment-out the current call of **createRoot** in **main.tsx**

2) After the import statements, declare a **const** called **name** and initialize it with your name.
   ```
   const name = “Bill”;
   ```

3) Declare another **const** named **root** and initialize with a call to **createRoot** that retrieves the element with an id of **root** (similar to the original code).
   ```
   const root = createRoot(document.getElementById(‘root’)!);
   ```

4) Call the **render** method of the root element so that an **h1** element is rendered that includes the value of the **name** constant.
   ```
   root.render(<h1>Hello, {name}</h1>);
   ```

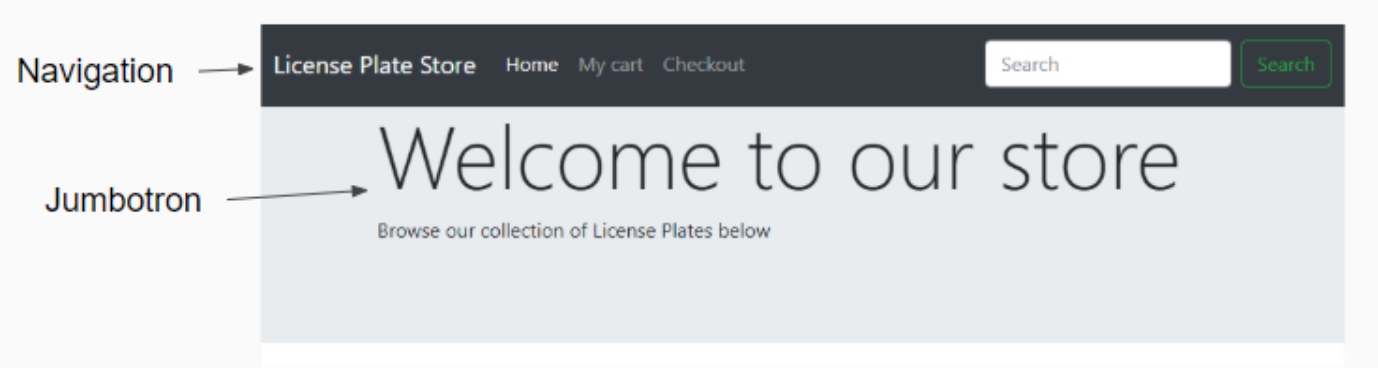
5) Check the results in the browser (you might need to start the development server with **npm run dev** if it is not still running)

6) Modify the call to render so that the current time is shown in an **h3** below the existing h1 and check the results in the browser.
   ```
   <h3>The time is {new Date().toLocaleTimeString()}</h3>
   ```

Lab 3: Defining Components

Goals:

- Configure the app to use Bootstrap for styling
- Build a **navigation** component
- Build a **jumbotron** component



Step by step instructions:

1) Add Bootstrap to the entire app by adding a link in the **<head>** section of **index.html**

```
<link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@4.6.2/dist/css/bootstrap.min.css">
```

2) Since we will not be using the default styles provided by the starter template, delete the contents of **index.css** and **App.css**

3) Create a new directory within the **`src`** folder called **`navigation`**

4) Create a new file within the **`navigation`** directory called **`Navigation.tsx`**.

5) Add code to **Navigation.tsx** that defines a simple **function component** named Navigation that returns an empty <nav> element.

```
export function Navigation() {
  return (
    <nav></nav>
  );
}
```

6) Copy the contents of **navigation.html** in the **lab-snippets** folder replace the empty <nav> element with the contents of that file. Notice that this content uses the class attribute in many places. This is a common change that must be made when brining in HTML for use as the content of a component.

7) Use the **Find and Replace** feature of Visual Studio code to replace all the **class** attributes with **className**.

8) Modify **main.tsx** so that it uses the **App** component like it did originally.

```
createRoot(document.getElementById('root')!).render(
  <StrictMode>
    <App />
  </StrictMode>
)
```

9) Open **App.tsx** in Visual Studio Code and add an **import** statement so that we can use the **Navigation** component within the App component.

```
import { Navigation } from './navigation/Navigation.tsx';
```

10) Replace all the code inside the **App** function with code that returns a **Navigation** component.

```
function App() {
  return (
    <Navigation />
  );
}
```

11) Check the results in the browser. You should see a **Navigation** component at the top of the page.

12) Add a **jumbotron** folder under the **src** folder and add a **new file** in the jumbotron folder named **Jumbotron.tsx**

13) Add a function component to the **Jumbotron.tsx** file named **Jumbotron** that displays some placeholder text for the **title** and **description**.

```
export function Jumbotron() {
  return (
    <div className="jumbotron">
      <div className="container">
        <h1 className="display-3">Title</h1>
        <p>Description</p>
      </div>
    </div>
  );
}
```

14) Modify the **App** component so that the **Jumbotron** is shown on the page. Don't forget to add the necessary **import** statement.

```
function App() {
  return (
    <div>
      <Navigation/>
      <main>
        <Jumbotron/>
      </main>
    </div>
  );
}
```

15) Check the results in the browser.

16) Modify the **Jumbotron** component so that is receives a **props** object.

```
export function Jumbotron(props: any) {
```

17) Modify the Jumbotron component so that it uses **title** and **description** from the props object instead of the current placeholder text.

18) In the **App** component provide a **title** and a **description** to the **Jumbotron** component so that the app looks like the image shows at the start of this lab.

Lab 4: Using Typescript with React

Goals:

- Define an **interface** for **JumbotronProps**
- **Refactor** the Jumbotron component

Step by step instructions:

1) Within **Jumbotron.tsx**, define an interface for **JumbotronProps**

```
export interface JumbotronProps {
  title: string;
  description: string;
}
```

2) Change the **type** of **props** from any to be **JumbotronProps**

```
export function Jumbotron(props: JumbotronProps)
```

3) Confirm that everything still works as before.

4) Experiment with writing code that does not match the **type annotation**. For example, in the App component, spell **title** incorrectly when providing it to the Jumbotron.

Lab 5: Events and Props

Goals:

- Define a **LicensePlate** component
- Use **props** to pass information to the component
- Listen for button **click** event and display an alert

Step by step instructions:

1) Copy (or move) the **mock-data.ts** file in the **lab-snippets** directory to app's **src** directory. This file contains some sample license plate data.

2) Add a **license-plate** directory under **src** and add a file named **LicensePlate.tsx**

3) Add a **function** to **LicensePlate.tsx** named **LicensePlate** that accepts a **props** object. The props object will contain a plate object and some text for the component's button.

```
export function LicensePlate(props: any) { }
```

4) For the first line of the function, decompose the **props** object into **two separate constants**.

```
const {plate, buttonText} = props;
```

5) Write code that uses **plate** and **buttonText** to return some HTML. The HTML is available in **lab-snippets/license-plate.html** if you would like to copy it from there.

```
return (
  <>
    <h2>{plate.title}</h2>
    <img src={plate.picture} className="img-fluid" />
    <p>{plate.description}</p>
    <div>
      <h2 className="float-left">${plate.price}</h2>
      <button className="btn btn-primary float-right"
              role="button">{buttonText}</button>
    </div>
  </>
);
```

6) Open **src/App.tsx** and import the **LicensePlate** component and some license plate data from **mock-data.ts**

```
import { LicensePlate } from './license-plate/LicensePlate.tsx'
import { CALIFORNIA_PLATE, LICENSE_PLATES } from './mock-data.ts'
```

7) Immediately below the **<Jumbotron>** element (still inside of the <main> element), add some HTML that uses some bootstrap styles and renders a **LicensePlate** component.

```
<div className="container" >
  <div className="row" >
    <div className="col-md-4">
      <LicensePlate plate={CALIFORNIA_PLATE}
                    buttonText="Add to cart"/>
    </div>
  </div>
</div>
```

8) Check the results in the browser. You should see a single license plate.

9) Add an event handler function within **LicensePlate.tsx** (separate from the component function)

```
const buttonClicked = () => {
  alert("Plate added to cart");
}
```

10) Attach that function to the **onClick** event for the button.

```
onClick={buttonClicked}
```

11) Check the behavior of the button in the browser.

BONUS - If you have time and want to explore more advanced concepts on your own:

Add type information for the props of the **LicensePlate** component and use that interface as the type of the props object received by the **LicensePlate** component. You can look at the contents of **mock-data.ts** for some help with how the interface should look.

Lab 6: Lists and Keys

Goal:

- Use JavaScript's **`Array.map`** function to render all the license plates stored in the **`LICENSE_PLATES`** constant
- Render every other plate with a light gray background (HTML color code **#F5F5F5**)

Step by step instructions:

1) Within **App.tsx,** find the ** ** element with a className of **row** and surround the contents inside of that with code that calls the map function on the LICENSE_PLATES array.

```
<div className="row">
  {LICENSE_PLATES.map((licensePlate, index) => (
    <div className="col-md-4">
      <LicensePlate plate={CALIFORNIA_PLATE}
                    buttonText="Add to cart"/>
    </div>
  ))}
</div>
```

2) Check the results in the browser. You should see eight identical instances of the LicensePlate component (since we are still using CALIFORNIA_PLATE each time)

3) Add a **key** attribute and modify the plate attribute to use the value of **licensePlate**.

```
<div key={licensePlate._id} className="col-md-4">
  <LicensePlate plate={licensePlate} buttonText="Add to cart"/>
</div>
```

4) Check the results in the browser again. You should now see eight different plates.

5) Use the **index** variable from the map function to specify a different **background color** for every other license plate. This should be applied to the ** ** element that contains the LicensePlate component. Double braces are needed when generating an inline style.

```
style={{backgroundColor: (index % 2 == 0) ? '#F5F5F5' : ''}}
```

6) Check the results in the browser.

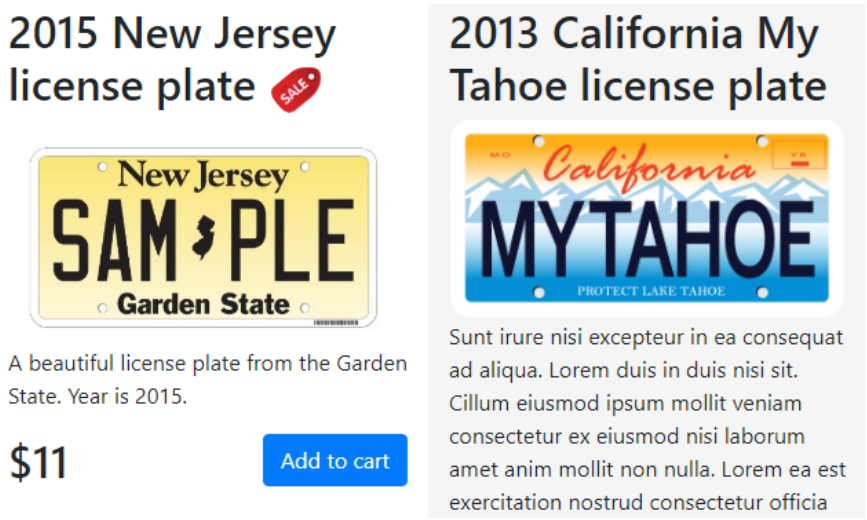
BONUS - If you have time and want to explore more advanced concepts on your own:

Instead of using inline “hard coded” CSS to style the background color, create a specific CSS class in **App.css** and use it to apply the proper styling here.

Lab 7: Conditional Rendering

Goal:

- Some license plates are on sale (property **`onSale = true`**). We want to showcase those products by adding an image next to their title.
- Using the file **`sale.png`** and update your license plate component so that products on sale appear as follows:



Step by step instructions:

1) Move (or copy) **sale.png** from the **lab-snippets** folder to the **src/assets** folder.
2) Add an **import** statement to the top of **LicensePlate.tsx** for **sale.png**

   ```
   import saleIcon from '../assets/sale.png'
   ```

3) Add an **image element** for **sale.png** immediately after where the **title** is displayed. This will cause the sale image to appear for every plate.

   ```
   <h2>{plate.title}<img src={saleIcon}/></h2>
   ```

4) Modify the code so that the sale image only appears for license plates where **onSale** is **true** and check the results in the browser.

Lab 8: Using State

Goals:

- Add some state to the **App** component
- Explore the **React Dev Tools** in the browser

Step by step instructions:

1) Edit **`src/App.tsx`** to add license plates to the state of that component.

```
export function App() {
   const [licensePlates, setPlates] = useState(LICENSE_PLATES);
```

2) In the JSX, replace the code that uses LICENSE_PLATES with code that uses the state variabl.

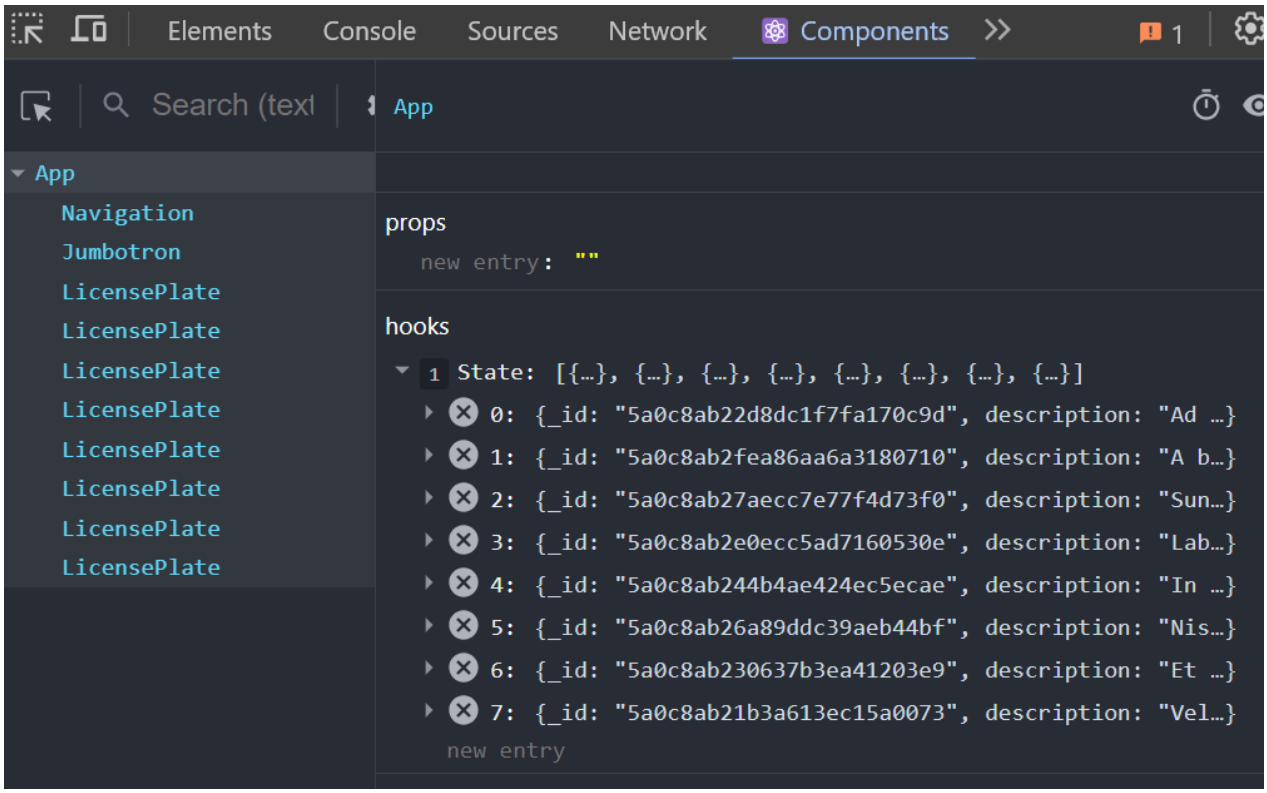
```
{licensePlates.map((licensePlate, index) =>
```

3) Our component is now using a local state to render its list of license plates. This is an improvement because calling setPlates() in the component will allow us to update the data with fresh information downloaded from a server.

4) If you don't already have it installed, install the React Developer Tools extension by doing a Google search for: "React Developer Tools" and clicking on the Chrome Web Store link.

5) Once the extension installed, open the browser dev tools (by pressing **F12** or **right click** on our React app web page then **Inspect**)

6) Click on the "Components" tab in the dev tools and explore the component hierarchy where state and props are visible, which makes it easier to debug React applications:



BONUS STEP - if you have time and want to explore more advanced concepts on your own:

- Add type information for the state of the **App** component.

Lab 9: Class Component Migration and useState

Goals:

- Turn a **CurrencyDropdown** class component into a function component that uses Hooks to store its state.

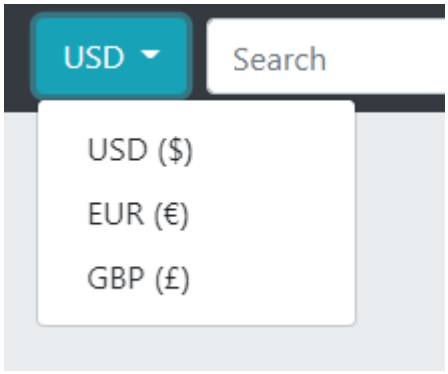
Step by step instructions:

1) Copy (or move) the **currency** folder from the **lab-snippets** folder into the project's **src** folder.

2) Open **`src/navigation/Navigation.tsx`** in your editor, add the **CurrencyDropdown** component as an import, and place the component just above the form element:

```
import { CurrencyDropdown } from "../currency/CurrencyDropdown";
…
<CurrencyDropdown/>
<form className="form-inline my-2 my-lg-0">
```

3) Check the user interface in your browser and make sure the dropdown can be used to make a new currency selection.



4) Now, let's turn the **CurrencyDropdown** class component into a function component.

5) First, replace the class declaration with a function declaration.

```
export function CurrencyDropdown() {
```

6) Next, remove the **render** method and have the CurrencyDropdown function **return** the JSX directly.

7) Replace the constructor code with **useState** hooks. Don't forget to also add the necessary **import**.

```
let [showItems, setShowItems] = useState(false);
let [currency, setCurrency] = useState("USD");
```

8) Modify the JSX code to use the new state variables.

9) Test the component and make sure that everything is still working as expected.

Lab 10: useEffect and Updating State

Goals:

- In this lab, we want to add a promo banner with a countdown that gets updated every second.
- Use the **PromoBanner** component and complete it so the countdown works.

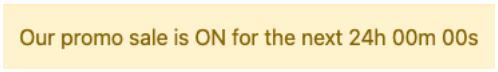
Step by step instructions:

1) Copy (or move) the **currency** folder from the **lab-snippets** folder into the project's **src** folder.

2) In **App.tsx**, add a **PromoBanner** component between the Jumbotron and the license plates. VS Code can add the import statement automatically when performing auto-complete.

```
<Jumbotron … />
<PromoBanner />
<div className="container">
```

3) Check the results in the browser. The banner should appear but the countdown timer does not update.



4) Add code in **PromoBanner.tsx** (before the return statement) that takes advantage of **useEffect** to update **timeLeft**.

```
useEffect(() => {
  const id = setInterval(() => {
    setTimeLeft(timeLeft - 1);
  }, 1000)
}, []);
```

5) Check the results in the browser. There is a problem! The banner may update once but then not update again. Why?

6) To fix the issue, we need to use the version of **setTimeLeft** that uses a **function** with the current state as a parameter.

```
setTimeLeft(time => time - 1);
```

7) Check the results in the browser. Something is still wrong.

   The countdown timer is decreasing by at least two seconds each time. This is happening because our React application is running in strict mode and strict mode runs every component of the application twice to help us identify when we're doing something wrong. What are we doing wrong, then? We never stop our interval timer, and since React runs the render method twice, we decrease our number of seconds twice. That's React's way of telling us we must cancel our timer when a component is unmounted.

8) Add code to **useEffect** that returns a function to **clean up** and stop the timer.

```
useEffect(() => {
  const id = setInterval(() => {
    setTimeLeft(time => time - 1);
  }, 1000);
  return () => clearInterval(id);
}, []);
```

9) Check the result in the browser. You may need to manually refresh the browser once, but the banner should now be working properly.

   This simple example emphasizes how important it is to update state using the right approach and to clean-up any timers/subscriptions created in effects.

Lab 11: Making HTTP Requests with fetch

Goals:

- Connect the **App** component to an **HTTP server**
- Use the **fetch** API to download license plate data

Step by step instructions:

1) Stop the app if it is currently running (type **q + enter** in the VS Code terminal).

2) In the VS Code terminal, make sure you are in the same directory as package.json and then run the following command to install a package for the license plate server.

   ```
   npm install lp-store-server
   ```

3) Open a separate terminal window, change to the root project directory (with the node_modules folder), and run the following command to start the license plate server.

   ```
   node node_modules/lp-store-server
   ```

   You should see the message "License Plate server listening on 8000". If you get an errors when trying to star the server, it's likely port 8000 is already in use on your machine. In that case, you can change the port number by setting an environment variable on the command line. For example, from the command line run: **`export PORT=8001 ; node node_modules/lp-store-server.`**

4) Before adding code to the app, if you would like to see the data provided by the server, you can use a browser to make a request to http://localhost:8000/data

5) Leave the server running in the separate terminal windows and return to VS Code.

6) In **`App.tsx`**, initialize the state to an **empty array** instead of using the mock data. You also remove the **import** statement for **mock-data.ts**

   ```
   const [licensePlates, setPlates] = useState([]);
   ```

7) Before the return statement, add a call to **useEffect** with code that uses **fetch** to make a request to the HTTP server and updates the state.

```
useEffect(() => {
  fetch('http://localhost:8000/data')
    .then(response => response.json())
    .then(data => setPlates(data));
}, []);
```

- The **`useEffect`** function runs **`after`** the component gets rendered to the DOM for the first time. The REST API is "fetched" at this point.

- The **`setPlates`** method will cause the component to re-render. So once **`fetch`** resolves with data from the API, the component will repaint with new data.

8) Run the app again with **npm run dev** and check the results. If you would like, you can use the network tab of the browser developer tools to see the request being made to the HTTP server.

BONUS STEP - if you have time and want to explore more advanced concepts on your own:

- Implement a "spinner" or "loader" that gets displayed while the API data is being fetched. You can use the **Spinner** component in the **lab-snippets** folder.

Lab 12: Using a Controlled Component Form

Goals:

- Implement a **CheckoutForm** component as a **controlled component**
- Implement some form **validation** with feedback to the user
- Submit the form to an HTTP server

Step by step instructions:

1) Copy (or move) the **checkout** folder from the **lab-snippets** folder into the project's **src** folder.

2) Add the **CheckoutForm** component to **App.tsx** immediately after the **Jumbotron** component.

3) Open **CheckoutForm.tsx** and make sure you understand how this component is currently set up, especially the **handleChange** function. Notice that only the **firstname** and **lastname** inputs are currently set up as controlled components.

4) Apply a similar "recipe" to all the other inputs to make them controlled components. Here is an example for the **street** input:

```
<input type="text" className="form-control"
  placeholder="Street" name="street" required
  value={street} onChange={e => handleChange(e, setStreet)} />
```

5) An easy way to validate your work is to check the current value of your state using the React dev tools component tab. The state value should always be in sync with the values entered by the user in the form.

6) Find the zip input, and edit it to specify that the zip is now required and should follow the given pattern (a 5-digit number):

```
<input type="text" className="form-control" value={zip}
  placeholder="Zip" name="zip" required pattern="[0-9]{5}" />
```

7) Next, let's add a new function in **CheckoutForm.tsx** to check the validation state of our zip code and update the component state accordingly.

```
const checkZipCodeValidity = (event: any) => {
  handleChange(event, setZip);
  if (event.target.validationMessage != "") {
    setZipValid(false);
  } else {
    setZipValid(true);
  }
}
```

8) Call the new function every time the zip input value is changed and display an error message when the zip value is invalid:

```
<div className="col-lg-6">
  {! zipValid && <div className="alert alert-danger">
    Please enter a 5-digit zipcode
    </div>
  }
  <div className="input-group">
    <input type="text" className="form-control"
      placeholder="Zip" name="zip" value={zip}
      onChange={checkZipCodeValidity}
      required pattern="[0-9]{5}"/>
  </div>
</div>
```

9) To handle the submit of the form, add an **onSubmit** function right after the **handleChange** function in **CheckoutForm.tsx**.

```
const onSubmit = (event: any) => {
  event.preventDefault();
  fetch('http://localhost:8000/checkout', {
    method: 'POST',
    headers: {'Content-Type': 'application/json'},
    body: JSON.stringify({firstname, lastname,
      street, city, zip, state, cc })
  })
};
```

10) Ensure the function is called when the form is submitted.

```
<form id="checkoutForm" onSubmit={onSubmit}>
```

11) Fill out the form and try to submit it. Explore the browser devtools to make sure that the HTTP request to http://localhost:8000/checkout is happening as expected.

BONUS STEPS - if you have time and want to explore more advanced concepts on your own:

- Add a dropdown to pick a US state from a list.
- Use the following data to load the full list of states with an HTTP request: **http://localhost:8000/states**

Lab 13: Component Router

Goals:

- We would like three different views: A store view, a checkout view, and a cart view
- **StoreViewComponent** should be the default route
- **CartViewComponent** should be at **/cart**
- **CheckoutViewComponent** should be at **/checkout**

Step by step instructions:Store

1) Create a new component called **`StoreView`**. Create a new directory called **`src/store-view`**, and add the function component there.

   ```
   export function StoreView() { }
   ```

   This new component will handle most of the existing functionality in **`App.tsx`**.

2) Move the **state** and **useEffect** hook into **`StoreView`**.

3) Add a return statement to **StoreView** and move the JSX content that is currently within the **<main>** element in **App.tsx** into the return statement. Since there are multiple top-level elements, you will need to put the content inside of a **React fragment**. You will also need to add the necessary **import** statements.

   `StoreView` is our new main view with all the license plates. **`App.tsx`** is will host that view with the navigation component above it.

4) Remove the **<CheckoutForm>** component from **StoreView.tsx** since that will be on its own page now.

5) So, that we can use **BrowserRouter**, add the package for **React Router**.

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

6) Add a **<BrowserRouter>** element to **App.tsx** and add the necessary **import** statement.

```
return (
  <div>
    <BrowserRouter>
      <Navigation/>
      <main>
      </main>
    </BrowserRouter>
  </div>
);
```

7) Add a **<Routes>** element inside of the **<main>** element to specify what should appear for the default route.

```
<main>
  <Routes>
    <Route path="/" element={<StoreView />} />
  </Routes>
</main>
```

8) Run the application (if it's not currently running) and check the browser to confirm that the page looks correct (same as before but without the checkout form).

9) Copy (or move) the **cart-view, checkout-view,** and **cart-service** folders from **lab-snippets** into the project's **src** folder. You may also need to copy **license-plate-date.type.ts** into the **src** folder if you didn't do that earlier.

10) Add two more routes to the **<Routes>** element in **App.tsx** for the two new components.

```
<Route path="/cart" element={<CartView />} />
<Route path="/checkout" element={<CheckoutView />} />
```

11) Open **Navigation.tsx** and find the <a> elements for "**Home**", "**Cart**" and "**Checkout**".

12) Modify each of them to use a **<Link>** element. Here is the code for one of them (cart):

```
<Link to="/cart" className="nav-link">Cart</Link>
```

13) Return to the browser and test the navigation.

Lab 14: Component Communication - Lifting State Up

Goals:

- Update our application so that when a user selects a new currency with the **CurrencyDropdown** component, the change is propagated to the **App** component

- We are going to "Lift the state up", which will allow us to make our **LicensePlate** components change their currency symbol when the currency switcher is used.

Step by step instructions:

At this point, only the **CurrencyDropdown** component knows about the current currency value. To convey this information to each **LicensePlate**, we will "lift state up" from the **CurrencyDropdown** to the **App** component. Changes to the **App** state (via **setState**) will then trigger changes in any components below it and cause them to re-render as needed. Specifically, all **LicensePlates** need to re-render with the correct currency information.

1) Open **`CurrencyDropdown.tsx`** and delete the state for currency.

   ~~`let [currency, setCurrency] = useState("USD");`~~

2) Above the function, define a new **interface** that represents what the component should be given (the current currency and a function to call to update the currency).

```
export interface CurrencyDropdownProps {
  currency: Currency;
  onCurrencyChange: (newCurrency: Currency) => void
}
```

3) Modify the component function to receive an item with that interface as its props.

```
export function CurrencyDropdown(props: CurrencyDropdownProps) {
```

4) Update the code that access currency to use the props object.

```
{props.currency}
```

5) In the three places where **setCurrency( )** is called, update the code to call **props.onCurrencyChange( )** instead. Here is an example for USD:

```
props.onCurrencyChange('USD'); } }>
```

6) Add state for the current currency to **App.tsx** and give it an initial value of "**USD**".

```
const [currency, setCurrency] = useState('USD');
```

The only thing still left to do is to pass this state down into the CurrencyDropdown along with a function that the CurrencyDropdown should use to update the state. The parent component of the CurrencyDropdown is the **Navigation** component.

7) Modify **Navigation.tsx** so that it accepts props.

```
export function Navigation(props: any) {
```

8) Set the props for the **CurrencyDropdown** from the props that are passed in.

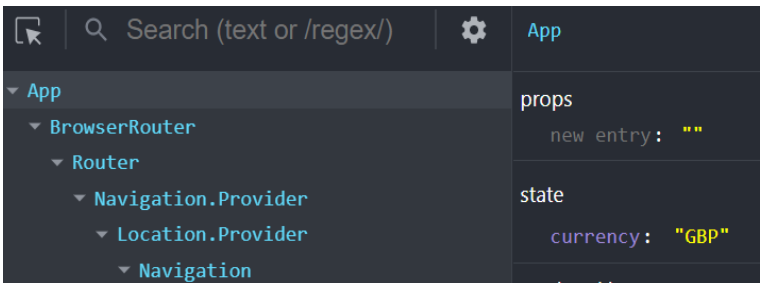
```
<CurrencyDropdown onCurrencyChange={props.onCurrencyChange}
                  currency={props.currency} />
```

9) Update **App.tsx** so it passes the current currency and the currency setter method as props to the **Navigation** component.

```
<Navigation onCurrencyChange={setCurrency}
            currency={currency} />
```

10) Check the behavior in the browser. The currency symbols for the license plates don't update. Why?

Some more changes are required. For now, you can use the React Developer Tools to confirm that the state in **App.tsx** is being updated.



Lab 15: Component Communication

Goals:

- Complete the propagation of our state update so that the currency set in the **App** component is displayed properly by each **LicensePlate** component

Step by step instructions:

Like the previous exercise, we will work from the bottom up. The **LicensePlate** component needs to know the current currency, so we'll start there.

1) Modify **LicencePlate.tsx** to use a value from its props for the currency symbol.

```
<h2 className="float-left">{props.currency} {plate.price}</h2>
```

2) Modify **StoreView.tsx** so that it receives props.

```
export function StoreView(props: any) {
```

3) Make sure the currency is passed to the **LicensePlate** component when it is used in **StoreView**.

```
<LicensePlate currency={props.currency} plate={licensePlate}
  buttonText="Add to cart"/>
```

4) Modify **App.tsx** so the current currency is also passed to the **StoreView** component.

```
<Route path="/" element={<StoreView currency={currency} />} />
```

5) Check the behavior in the browser. Something is not quite right. The three-letter abbreviation is being used instead of the proper currency symbol.

6) Add a mapping object in **LicensePlate.tsx**

```
const CURRENCIES: Record<string, string> = {EUR: "€", USD: "$",
GBP: "£"};

export function LicensePlate(props: any) {
```

7) Use the mapping object to display the correct currency symbol.

```
<h2 className="float-left">
  {CURRENCIES[props.currency]}{plate.price}
</h2>
```

8) Test the behavior in the browser.

BONUS STEPS - if you have time and want to explore more advanced concepts on your own:

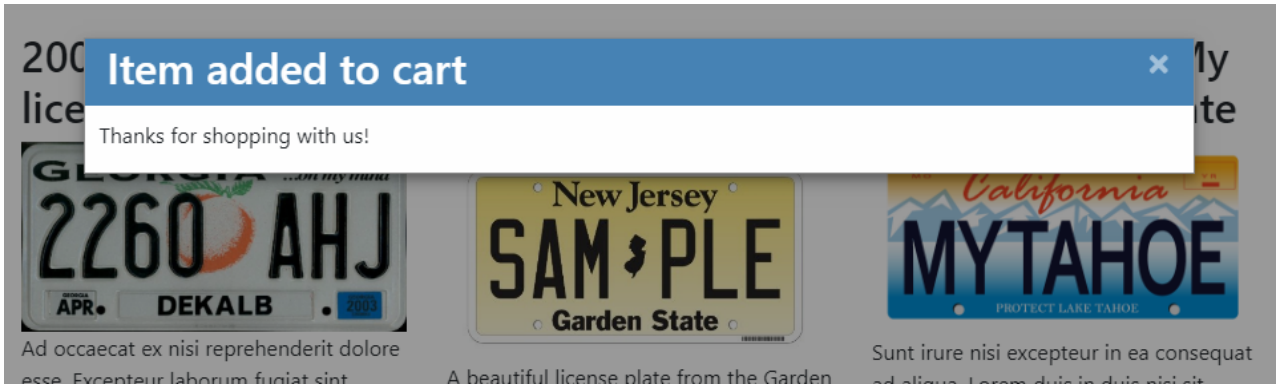
- Use the following back-end URL to get the exchange rates for each currency: http://localhost:8000/rates (those numbers are the exchange rates to use from a USD amount)

- Apply those exchange rates accordingly when a new currency is selected.

Lab 16: “Add to cart” Feature with a Hook

Goals:

- Use the **`cart-hook.ts`** hook features to add a license plate to the user’s cart.
- When the API returns successfully, display a **`PopupWindow`** to confirm success



Step by step instructions:

1) Take some time to review the contents of **`src/cart-service/cart-hook.ts`** This uses cart-service to interact with the REST API that manages the shopping cart contents.

2) Copy (or move) the **popup** directory from the **lab-snippets** directory to the project’s **src** directory.

3) Make sure that the backend, **`lp-store-server`**, is running.

4) Import **PopupWindow** in **StoreView.tsx**

```
import { PopupWindow } from "../popup/PopupWindow";
```

5) Define a state variable in **StoreView** named **showPopup** and initialize it to **false**.

```
const [showPopup , setShowPopup ] = useState<boolean>(false);
```

6) Add a method **addPlateToCart** and show the popup window when the item is successfully added to the cart.

```
const addPlateToCart = (plate: LicensePlateData) => {
  addToCart(plate).then(() => {
    setShowPopup(true);
  });
}
```

7) Add the **PopupWindow** at the end of the JSX code.

```
  <PopupWindow show={showPopup} title="Plate added to cart"
    onClose={() => setShowPopup(false) }>
    Thanks for shopping with us!
  </PopupWindow>
</>
```

Notice that **PopupWindow** is passed two props: a Boolean property **show**, and a callback function property, **onClose**.

8) Add a prop that passes the **addPlateToCart** function to the **LicensePlate** component.

```
<LicensePlate currency={props.currency}
 plate={licensePlate} buttonText="Add to cart"
 onButtonClicked={addPlateToCart} />
```

9) In **LicensePlate.tsx**, use the prop to invoke the function**.**

```
const buttonClicked = () => {
  props.onButtonClicked(props.plate);
}
```

10) Test the behavior in the browser. You can check the cart contents by navigating to the **Cart** view.

BONUS STEP - if you have time and want to explore more advanced concepts on your own:

- Implement the “Remove from cart” feature in the **CartView** component.

Lab 17: React Production Build

Goal:

- Create a production build for React: **`npm run build`**

Step by step instructions:

1. Open a terminal in our labs folder **`react-training-start-master`**
2. Run the **`npm run build`** command in that terminal to compile the code of the entire project.
3. Once the build completes, examine the contents of the **dist** folder. That's your build output with all the compiled, minified, and obfuscated JavaScript code.