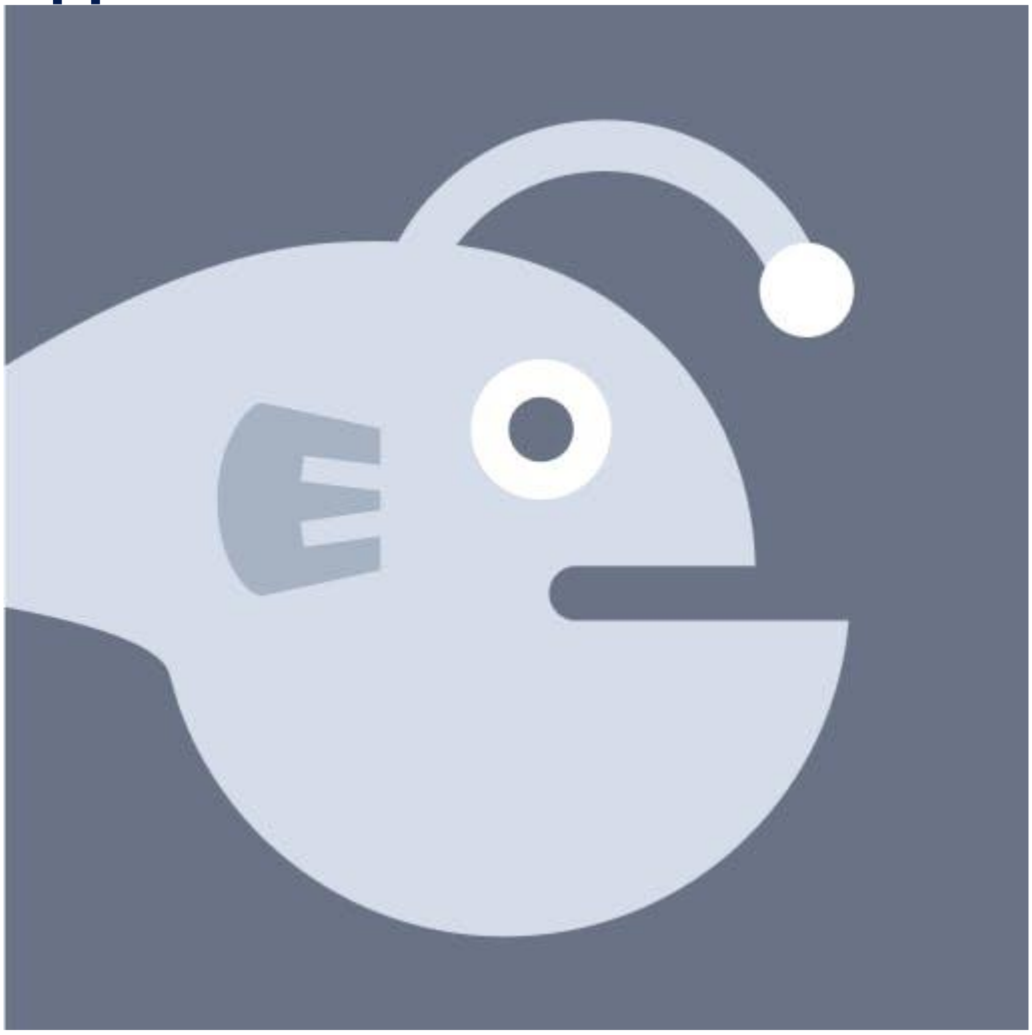
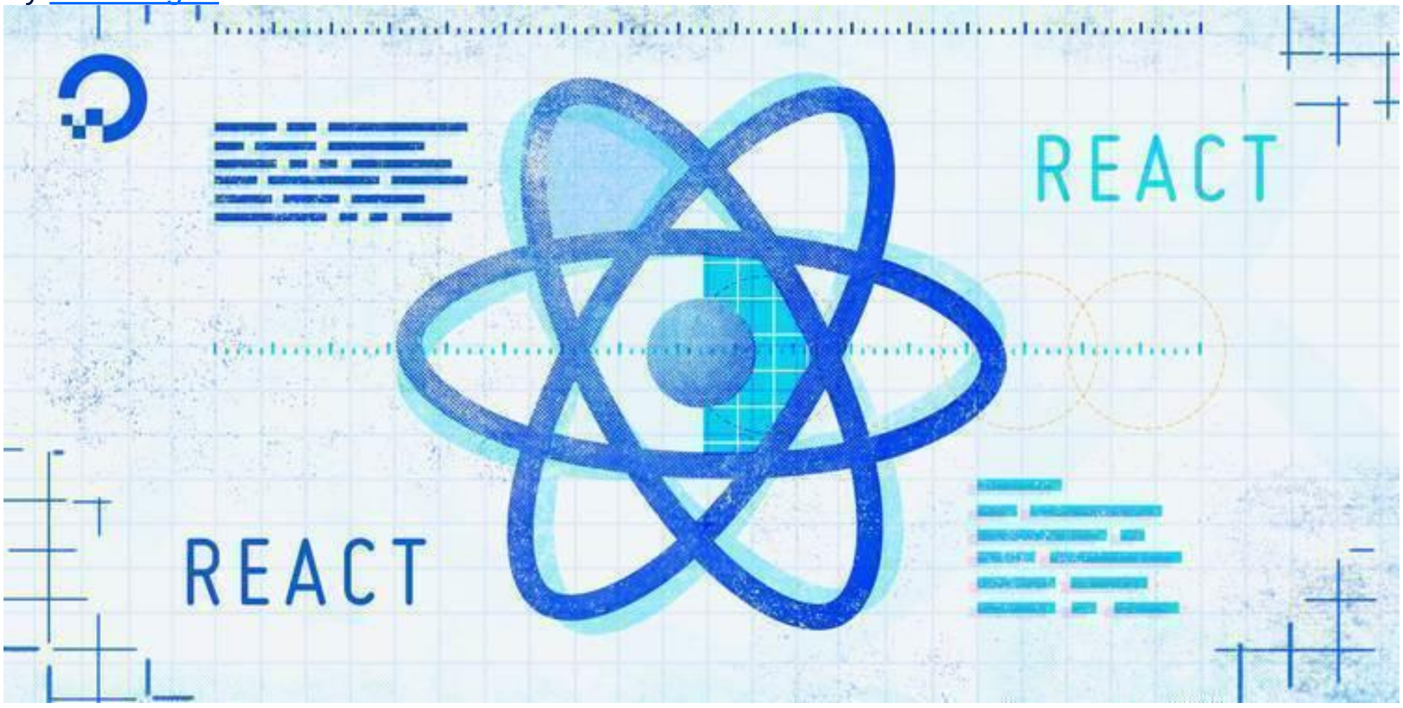
How To Add Login Authentication to React Applications



By Joe Morgan



Introduction

Many web applications are a mix of public and private pages. Public pages are available to anyone, while a private page requires a user login. You can use *authentication* to manage which users have access to which pages.

Your React application will need to handle situations where a user tries to access a private page before they are logged in, and you will need to save the login information once they have successfully authenticated.

In this tutorial, you'll create a React application using a token-based authentication system. You'll create a mock API that will return a user token, build a login page that will fetch the token, and check for authentication without rerouting a user. If a user is not authenticated, you'll provide an opportunity for them to log in and then allow them to continue without navigating to a dedicated login page. As you build the application, you'll explore different methods for storing tokens and will learn the security and experience trade-offs for each approach. This tutorial will focus on storing tokens in `localStorage` and `sessionStorage`.

By the end of this tutorial, you'll be able to add authentication to a React application and integrate the login and token storage strategies into a complete user workflow.

Need to deploy a React project quickly? Check out DigitalOcean App Platform and deploy a React project directly from GitHub in minutes.

Prerequisites

- You will need a development environment running Node.js; this tutorial was tested on Node.js version 10.22.0 and npm version 6.14.6. To install this on macOS or Ubuntu 18.04, follow the steps in How to Install Node.js and Create a Local Development Environment on macOS or the **Installing Using a PPA** section of How To Install Node.js on Ubuntu 18.04.
- A React development environment set up with Create React App, with the non-essential boilerplate removed. To set this up, follow **Step 1 — Creating an Empty Project** of the How To Manage State on React Class Components tutorial. This tutorial will use `auth-tutorial` as the project name.
- You will be fetching data from APIs using React. You can learn about working with APIs in How To Call Web APIs with the useEffect Hook in React.
- You will also need a basic knowledge of JavaScript, HTML, and CSS, which you can find in our How To Build a Website With HTML series, How To Style HTML with CSS, and in How To Code in JavaScript.

Step 1 — Building a Login Page

In this step, you'll create a login page for your application. You'll start by installing React Router and creating components to represent a full application. Then you'll render the login page on any route so that your users can login to the application without being redirected to a new page.

By the end of this step, you'll have a basic application that will render a login page when a user is not logged into the application.

To begin, install react router with `npm`. There are two different versions: a web version and a native version for use with React Native. Install the web version:

```
npm install react-router-dom

```

Copy

The package will install and you'll receive a message when the installation is complete. Your message may vary slightly:

```
Output
...
+ react-router-dom@5.2.0
added 11 packages from 6 contributors, removed 10 packages and audited 1945
packages in 12.794s
...
```

Next, create two components called `Dashboard` and `Preferences` to act as private pages. These will represent components that a user should not see until they have successfully logged into the application.

First, create the directories:

```
mkdir src/components/Dashboard

mkdir src/components/Preferences
```

Copy

Then open `Dashboard.js` in a text editor. This tutorial will use nano:

```
nano src/components/Dashboard/Dashboard.js
```

Copy

Inside of `Dashboard.js`, add an `<h2>` tag with the content of `Dashboard`:

auth-tutorial/src/components/Dashboard/Dashboard.js

```
import React from 'react';

export default function Dashboard() {
  return(
    <h2>Dashboard</h2>
  );
}
```

Copy

Save and close the file.

Repeat the same steps for `Preferences`. Open the component:

```
nano src/components/Preferences/Preferences.js
```

Copy

Add the content:

auth-tutorial/src/components/Preferences/Preferences.js

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

```
export default function Preferences() {
  return(
    <h2>Preferences</h2>
  );
}
```
Copy

Save and close the file.

Now that you have some components, you need to import the components and create routes inside of `App.js`. Check out the tutorial [How To Handle Routing in React Apps with React Router](#) for a full introduction to routing in React applications.
To begin, open `App.js`:

```
nano src/components/App/App.js
```
Copy

Then import `Dashboard` and `Preferences` by adding the following highlighted code:

auth-tutorial/src/components/App/App.js

```
import React from 'react';
import './App.css';
import Dashboard from '../Dashboard/Dashboard';
import Preferences from '../Preferences/Preferences';

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

export default App;
```
Copy

Next, import `BrowserRouter`, `Switch`, and `Route` from `react-router-dom`:

auth-tutorial/src/components/App/App.js

```
import React from 'react';
import './App.css';
import { BrowserRouter, Route, Switch } from 'react-router-dom';
import Dashboard from '../Dashboard/Dashboard';
import Preferences from '../Preferences/Preferences';

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

export default App;
```

Copy

Add a surrounding `<div>` with a `className` of `wrapper` and an `<h1>` tag to serve as a template for the application. Be sure that you are importing `App.css` so that you can apply the styles.
Next, create routes for the `Dashboard` and `Preferences` components. Add `BrowserRouter`, then add a `Switch` component as a child. Inside of the `Switch`, add a `Route` with a `path` for each component:

tutorial/src/components/App/App.js

```
import React from 'react';
import './App.css';
import { BrowserRouter, Route, Switch } from 'react-router-dom';
import Dashboard from '../Dashboard/Dashboard';
import Preferences from '../Preferences/Preferences';

function App() {
  return (
    <div className="wrapper">
      <h1>Application</h1>
      <BrowserRouter>
        <Switch>
          <Route path="/dashboard">
            <Dashboard />
          </Route>
          <Route path="/preferences">
            <Preferences />
          </Route>
        </Switch>
      </BrowserRouter>
    </div>
  );
}

export default App;
```

Copy

Save and close the file.

The final step is to add some padding to the main `<div>` so your component is not directly at the edge of the browser. To do this, you will change the CSS.
Open `App.css`:

```
nano src/components/App/App.css
```

Copy

Replace the contents with a class of `.wrapper` with `padding` of `20px`:

auth-tutorial/src/components/App/App.css

```
.wrapper {
    padding: 20px;
}
```
Copy

Save and close the file. When you do, the browser will reload and you'll find your basic components:

Check each of the routes. If you visit `http://localhost:3000/dashboard`, you'll find the dashboard page:

Your routes are working as expected, but there is a slight problem. The route `/dashboard` should be a protected page and should not be viewable by an unauthenticated user. There are different ways to handle a private page. For example, you can create a new route for a login page and use [React Router to redirect if the user is not logged in](). This is a fine approach, but the user would lose their route and have to navigate back to the page they originally wanted to view.
A less intrusive option is to generate the login page regardless of the route. With this approach, you'll render a login page if there is not a stored user token and when the user logs in, they'll be on the same route that they initially visited. That means if a user visits `/dashboard`, they will still be on the `/dashboard` route after login.
To begin, make a new directory for the `Login` component:

```
mkdir src/components/Login

```
Copy

Next, open `Login.js` in a text editor:

```
nano src/components/Login/Login.js

```
Copy

Create a basic form with a submit `<button>` and an `<input>` for the username and the password. Be sure to set the input type for the password to `password`:

auth-tutorial/src/components/Login/Login.js

```
import React from 'react';

export default function Login() {
  return(
    <form>
      <label>
        <p>Username</p>
        <input type="text" />
      </label>
      <label>
        <p>Password</p>
        <input type="password" />
      </label>
      <div>
        <button type="submit">Submit</button>
      </div>
    </form>
  )
}
```
Copy

For more on forms in React, check out the tutorial How To Build Forms in React.
Next, add an `<h1>` tag asking the user to log in. Wrap the `<form>` and the `<h1>` in a `<div>` with a `className` of `login-wrapper`. Finally, import `Login.css`:

auth-tutorial/src/components/Login/Login.js

```
import React from 'react';
import './Login.css';

export default function Login() {
  return(
    <div className="login-wrapper">
      <h1>Please Log In</h1>
      <form>
        <label>
          <p>Username</p>
          <input type="text" />
        </label>
        <label>
          <p>Password</p>
          <input type="password" />
        </label>
        <div>
          <button type="submit">Submit</button>
        </div>
      </form>
    </div>
  )
}
```
Copy

Save and close the file.

Now that you have a basic `Login` component, you'll need to add some styling. Open `Login.css`:

```
nano src/components/Login/Login.css
```
Copy

Center the component on the page by adding a `display` of `flex`, then setting the `flex-direction` to `column` to align the elements vertically and adding `align-items` to `center` to make the component centered in the browser:

auth-tutorial/src/components/Login/Login.css

```
.login-wrapper {
    display: flex;
    flex-direction: column;
    align-items: center;
}
```
Copy

For more information on using Flexbox, see our CSS Flexbox Cheatsheet

Save and close the file.

Finally, you'll need to render it inside of `App.js` if there is no user token. Open `App.js`:

```
nano src/components/App/App.js
```
Copy

In **Step 3**, you'll explore options for storing the token. For now, you can store the token in memory using the `useState` Hook.
Import `useState` from `react`, then call `useState` and set return values to `token` and `setToken`:

auth-tutorial/src/components/App/App.js

```
import React, { useState } from 'react';
import { BrowserRouter, Route, Switch } from 'react-router-dom';
import './App.css';
import Dashboard from '../Dashboard/Dashboard';
import Preferences from '../Preferences/Preferences';

function App() {
  const [token, setToken] = useState();
  return (
    <div className="wrapper">
      <h1>Application</h1>
      <BrowserRouter>
        <Switch>
          <Route path="/dashboard">
            <Dashboard />
          </Route>
          <Route path="/preferences">
            <Preferences />
          </Route>
        </Switch>
      </BrowserRouter>
    </div>
  );
}

export default App;
```
Copy

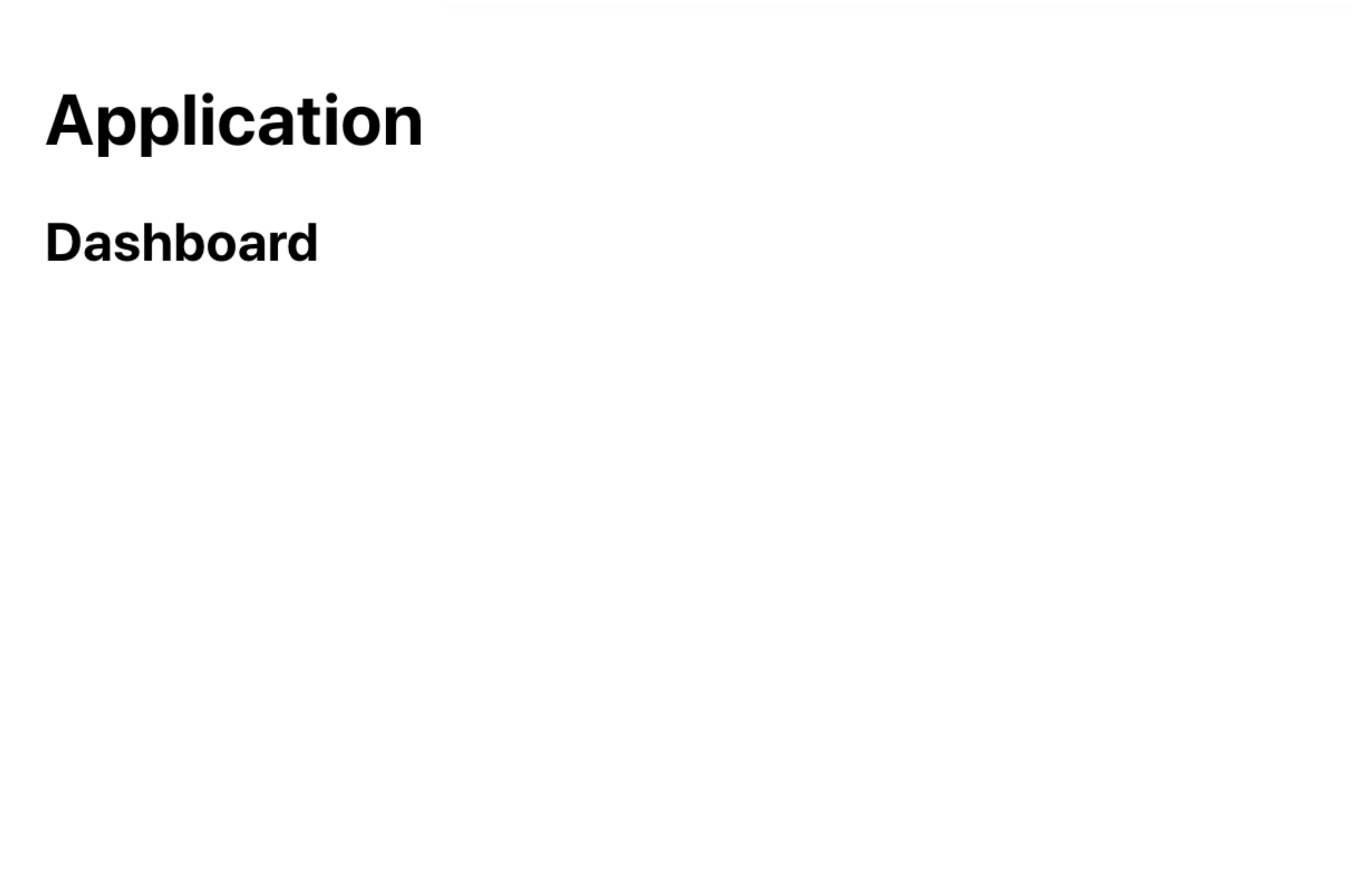
Import the `Login` component. Add a conditional statement to display `Login` if the `token` is falsy.
Pass the `setToken` function to the `Login` component:

auth-tutorial/src/components/App/App.js

```
import React, { useState } from 'react';
import { BrowserRouter, Route, Switch } from 'react-router-dom';

import './App.css';
import Dashboard from '../Dashboard/Dashboard';
import Login from '../Login/Login';
import Preferences from '../Preferences/Preferences';

function App() {
  const [token, setToken] = useState();

  if(!token) {
    return <Login setToken={setToken} />
  }

  return (
    <div className="wrapper">
      <h1>Application</h1>
      <BrowserRouter>
        <Switch>
          <Route path="/dashboard">
            <Dashboard />
          </Route>
          <Route path="/preferences">
            <Preferences />
          </Route>
        </Switch>
      </BrowserRouter>
    </div>
  );
}

export default App;
```
Copy

For now, there is no token; in the next step, you'll call an API and set the token with the return value.

Save and close the file. When you do, the browser will reload and you'll see the login page. Notice that if you visit `http://localhost:3000/dashboard`, you'll still find the login page since the token has not yet been set:


Application

Dashboard

In this step, you created an application with private components and a login component that will display until you set a token. You also configured routes to display the pages and added a check to display the `Login` component on every route if the user is not yet logged into the application.
In the next step, you'll create a local API that will return a user token. You'll call the API from the `Login` component and save the token to memory on success.

Step 2 — Creating a Token API

In this step, you'll create a local API to fetch a user token. You'll build a mock API using Node.js that will return a user token. You'll then call that API from your login page and render the component after you successfully retrieve the token. By the end of this step, you'll have an application with a working login page and protected pages that will only be accessible after login.
You are going to need a server to act as a backend that will return the token. You can create a server quickly using Node.js and the Express web framework. For a detailed introduction to creating an Express server, see the tutorial Basic Express Server in Node.js.
To start, install `express`. Since the server is not a requirement of the final build, be sure to install as a `devDependency`.
You'll also need to install `cors`. This library will enable cross origin resource sharing for all routes.

Warning: Do not enable CORS for all routes in a production application. This can lead to security vulnerabilities.

```
npm install --save-dev express cors
```
Copy

When the installation is complete, you'll receive a success message:

```
Output
...
+ cors@2.8.5
+ express@4.17.1
removed 10 packages, updated 2 packages and audited 2059 packages in 12.597s
...
```
Next, open a new file called `server.js` in the root of your application. Do not add this file to the `/src` directory since you do not want it to be part of the final build.

```
nano server.js
```
Copy

Import `express`, then initialize a new app by calling `express()` and saving the result to a variable called `app`:

auth-tutorial/server.js
```
const express = require('express');
const app = express();
```
Copy

After creating the `app`, add `cors` as a middleware. First, import `cors`, then add it to the application by calling the `use` method on `app`:

auth-tutorial/server.js
```
const express = require('express');
const cors = require('cors');
const app = express();

app.use(cors());
```
Copy

Next, listen to a specific route with `app.use`. The first argument is the path the application will listen to and the second argument is a callback function that will run when the application serves the path. The callback takes a `req` argument, which contains the request data and a `res` argument that handles the result.
Add in a handler for the `/login` path. Call `res.send` with a JavaScript object containing a token:

auth-tutorial/server.js

```
const express = require('express');
const cors = require('cors')
const app = express();

app.use(cors());

app.use('/login', (req, res) => {
  res.send({
    token: 'test123'
  });
});
```

Copy

Finally, run the server on port `8080` using `app.listen`:

auth-tutorial/server.js

```
const express = require('express');
const cors = require('cors')
const app = express();

app.use(cors());

app.use('/login', (req, res) => {
  res.send({
    token: 'test123'
  });
});

app.listen(8080, () => console.log('API is running on
http://localhost:8080/login'));
```

Copy

Save and close the file. In a new terminal window or tab, start the server:

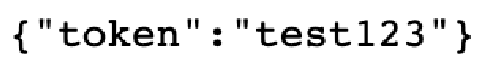
```
1. node server.js
2.
```

Copy

You will receive a response indicating that the server is starting:

```
Output
API is running on http://localhost:8080/login
```

Visit `http://localhost:8080/login` and you'll find your JSON object.



```
{"token":"test123"}
```

When you fetch the token in your browser, you are making a `GET` request, but when you submit the login form you will be making a `POST` request. That’s not a problem. When you set up your route with `app.use`, Express will handle all requests the same. In a production application, you should be more specific and only allow certain request methods for each route.

Now that you have a running API server, you need to make a request from your login page. Open `Login.js`:

```
nano src/components/Login/Login.js
```

Copy

In the previous step, you passed a new prop called `setToken` to the `Login` component. Add in the `PropType` from the new prop and destructure the props object to pull out the `setToken` prop.

auth-tutorial/src/components/Login/Login.js

```
import React from 'react';
import PropTypes from 'prop-types';

import './Login.css';

export default function Login({ setToken }) {
  return(
    <div className="login-wrapper">
      <h1>Please Log In</h1>
      <form>
        <label>
          <p>Username</p>
          <input type="text" />
        </label>
        <label>
          <p>Password</p>
          <input type="password" />
        </label>
        <div>
          <button type="submit">Submit</button>
        </div>
      </form>
    </div>
  )
}

Login.propTypes = {
  setToken: PropTypes.func.isRequired
}
```
Copy

Next, create a local state to capture the `Username` and `Password`. Since you do not need to manually set data, make the `<inputs>` uncontrolled components. You can find detailed information about uncontrolled components in How To Build Forms in React.

auth-tutorial/src/components/Login/Login.js

```
import React, { useState } from 'react';
import PropTypes from 'prop-types';

import './Login.css';

export default function Login({ setToken }) {
  const [username, setUserName] = useState();
  const [password, setPassword] = useState();
  return(
    <div className="login-wrapper">
      <h1>Please Log In</h1>
      <form>
        <label>
          <p>Username</p>
          <input type="text" onChange={e => setUserName(e.target.value)}/>
        </label>
        <label>
          <p>Password</p>
          <input type="password" onChange={e => setPassword(e.target.value)}/>
        </label>
        <div>
          <button type="submit">Submit</button>
        </div>
      </form>
    </div>
  )
}

Login.propTypes = {
  setToken: PropTypes.func.isRequired
};
```
Copy

Next, create a function to make a `POST` request to the server. In a large application, you would add these to a separate directory. In this example, you'll add the service directly to the component. Check out the tutorial How To Call Web APIs with the useEffect Hook in React for a detailed look at calling APIs in React components.
Create an `async` function called `loginUser`. The function will take `credentials` as an argument, then it will call the `fetch` method using the `POST` option:

auth-tutorial/src/components/Login/Login.js

```
import React, { useState } from 'react';
import PropTypes from 'prop-types';
import './Login.css';

async function loginUser(credentials) {
 return fetch('http://localhost:8080/login', {
   method: 'POST',
   headers: {
     'Content-Type': 'application/json'
   },
   body: JSON.stringify(credentials)
 })
   .then(data => data.json())
}

export default function Login({ setToken }) {
...
```
Copy

Finally, create a form submit handler called `handleSubmit` that will call `loginUser` with the `username` and `password`. Call `setToken` with a successful result.
Call `handleSubmit` using the `onSubmit` event handler on the `<form>`:

auth-tutorial/src/components/Login/Login.js

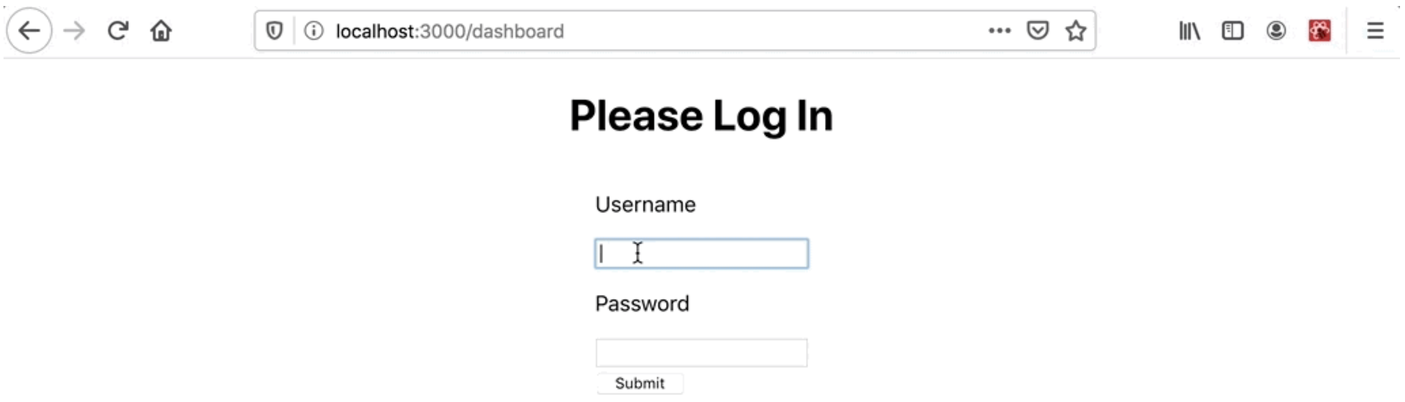
```
import React, { useState } from 'react';
import PropTypes from 'prop-types';
import './Login.css';
```

```
async function loginUser(credentials) {
 return fetch('http://localhost:8080/login', {
   method: 'POST',
   headers: {
     'Content-Type': 'application/json'
   },
   body: JSON.stringify(credentials)
 })
   .then(data => data.json())
}

export default function Login({ setToken }) {
  const [username, setUserName] = useState();
  const [password, setPassword] = useState();

  const handleSubmit = async e => {
    e.preventDefault();
    const token = await loginUser({
      username,
      password
    });
    setToken(token);
  }

  return(
    <div className="login-wrapper">
      <h1>Please Log In</h1>
      <form onSubmit={handleSubmit}>
        <label>
          <p>Username</p>
          <input type="text" onChange={e => setUserName(e.target.value)} />
        </label>
        <label>
          <p>Password</p>
          <input type="password" onChange={e => setPassword(e.target.value)} />
        </label>
        <div>
          <button type="submit">Submit</button>
        </div>
      </form>
    </div>
  )
}

Login.propTypes = {
  setToken: PropTypes.func.isRequired
};
```
Copy

Note: In a full application, you'll need to handle situations where the component unmounts before a Promise resolves. Check out the tutorial How To Call Web APIs with the useEffect Hook in React for more information.

Save and close the file. Make sure that your local API is still running, then open a browser to `http://localhost:3000/dashboard`.

You will see the login page instead of the dashboard. Fill out and submit the form and you will receive a web token then redirect to the page for the dashboard.



You now have a working local API and an application that requests a token using a username and password. But there is still a problem. The token is currently stored using a local state, which means that it is stored in JavaScript memory. If you open a new window, tab, or even just refresh the page, you will lose the token and the user will need to login again. This will be addressed in the next step.

In this step you created a local API and a login page for your application. You learned how to create a Node server to send a token and how to call the server and store the token from a login component. In the next step, you'll learn how to store the user token so that a session will persist across page refreshes or tabs.

Step 3 — Storing a User Token with `sessionStorage` and `localStorage`

In this step, you'll store the user token. You'll implement different token storage options and learn the security implications of each approach. Finally, you'll learn how different approaches will change the user experience as the user opens new tabs or closes a session.

By the end of this step, you'll be able to choose a storage approach based on the goals for your application.

There are several options for storing tokens. Every option has costs and benefits. In brief the options are: storing in JavaScript memory, storing in `sessionStorage`, storing in `localStorage`, and storing in a cookie. The primary trade-off is security. Any information that is stored outside of the memory of the current application is vulnerable to Cross-Site Scripting (XSS) attacks. The danger is that if a malicious user is able to load code into your application, it can access `localStorage`, `sessionStorage`, and any cookie that is also accessible to your application. The benefit of the non-memory storage methods is that you can reduce the number of times a user will need to log in to create a better user experience.
This tutorial will cover `sessionStorage` and `localStorage`, since these are more modern than using cookies.

Session Storage

To test the benefits of storing outside of memory, convert the in-memory storage to `sessionStorage`. Open `App.js`:

```
nano src/components/App/App.js
```

Copy

Remove the call to `useState` and create two new functions called `setToken` and `getToken`. Then call `getToken` and assign the results to a variable called `token`:

auth-tutorial/src/components/App/App.js

```
import React from 'react';
import { BrowserRouter, Route, Switch } from 'react-router-dom';

import './App.css';
import Dashboard from '../Dashboard/Dashboard';
import Login from '../Login/Login';
import Preferences from '../Preferences/Preferences';

function setToken(userToken) {
}

function getToken() {
}

function App() {
  const token = getToken();

  if(!token) {
    return <Login setToken={setToken} />
  }
  return (
    <div className="wrapper">
     ...
    </div>
  );
}

export default App;
```
Copy

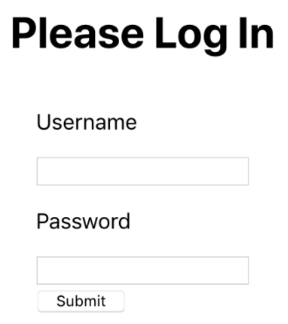
Since you are using the same function and variable names, you will not need to change any code in the `Login` component or the rest of the `App` component.
Inside of `setToken`, save the `userToken` argument to `sessionStorage` using the `setItem` method. This method takes a key as a first argument and a string as the second argument. That means you'll need to convert the `userToken` from an object to a string using the `JSON.stringify` function. Call `setItem` with a key of `token` and the converted object.

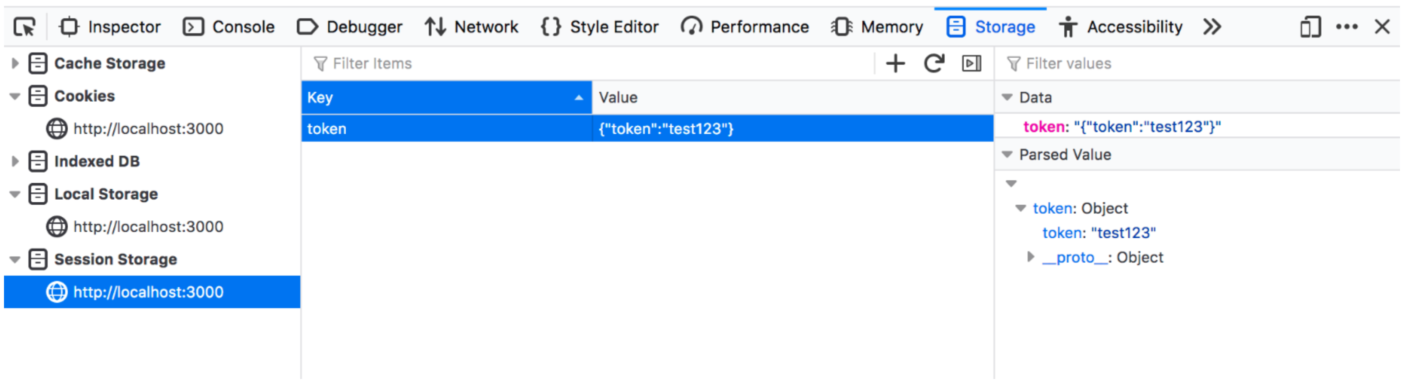
auth-tutorial/src/components/App/App.js

```
import React from 'react';
import { BrowserRouter, Route, Switch } from 'react-router-dom';

import './App.css';
import Dashboard from '../Dashboard/Dashboard';
import Login from '../Login/Login';
import Preferences from '../Preferences/Preferences';

function setToken(userToken) {
  sessionStorage.setItem('token', JSON.stringify(userToken));
}

function getToken() {
}

function App() {
  const token = getToken();

  if(!token) {
    return <Login setToken={setToken} />
  }

  return (
    <div className="wrapper">
      ...
    </div>
  );
}
```

```
export default App;
```
Copy

Save the file. When you do the browser will reload. If you type in a username and password and submit, the browser will still render the login page, but if you look inside your browser console tools, you'll find the token is stored in `sessionStorage`. This image is from [Firefox](), but you'll find the same results in [Chrome]() or other modern browsers.





http://localhost:3000

Key	Value
token	{"token":"test123"}

Data
token: "{"token":"test123"}"
Parsed Value
token: Object
token: "test123"
__proto__: Object

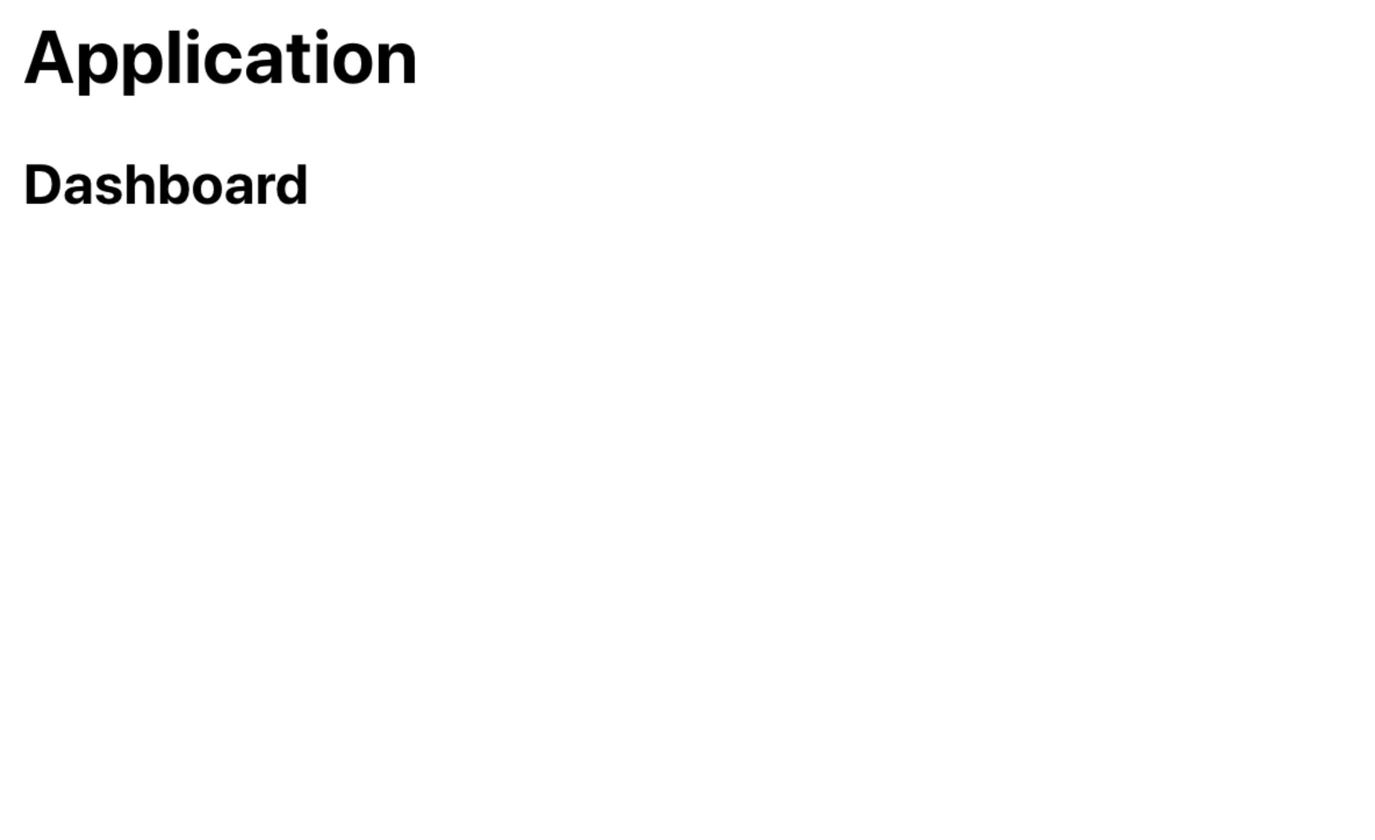
Now you need to retrieve the token to render the correct page. Inside the `getToken` function, call `sessionStorage.getItem`. This method takes a `key` as an argument and returns the string value. Convert the string to an object using `JSON.parse`, then return the value of `token`:

auth-tutorial/src/components/App/App.js

```
import React from 'react';
import { BrowserRouter, Route, Switch } from 'react-router-dom';
import './App.css';
import Dashboard from '../Dashboard/Dashboard';
import Login from '../Login/Login';
import Preferences from '../Preferences/Preferences';

function setToken(userToken) {
  sessionStorage.setItem('token', JSON.stringify(userToken));
}

function getToken() {
  const tokenString = sessionStorage.getItem('token');
  const userToken = JSON.parse(tokenString);
  return userToken?.token
}

function App() {
  const token = getToken();

  if(!token) {
    return <Login setToken={setToken} />
  }

  return (
    <div className="wrapper">
      ...
    </div>
  );
}

export default App;
```
Copy

You need to use the optional chaining operator—`?.`—when accessing the `token` property because when you first access the application, the value of `sessionStorage.getItem('token')` will be `undefined`. If you try to access a property, you will generate an error.

Save and close the file. In this case, you already have a token stored, so when the browser refreshes, you will navigate to the private pages:

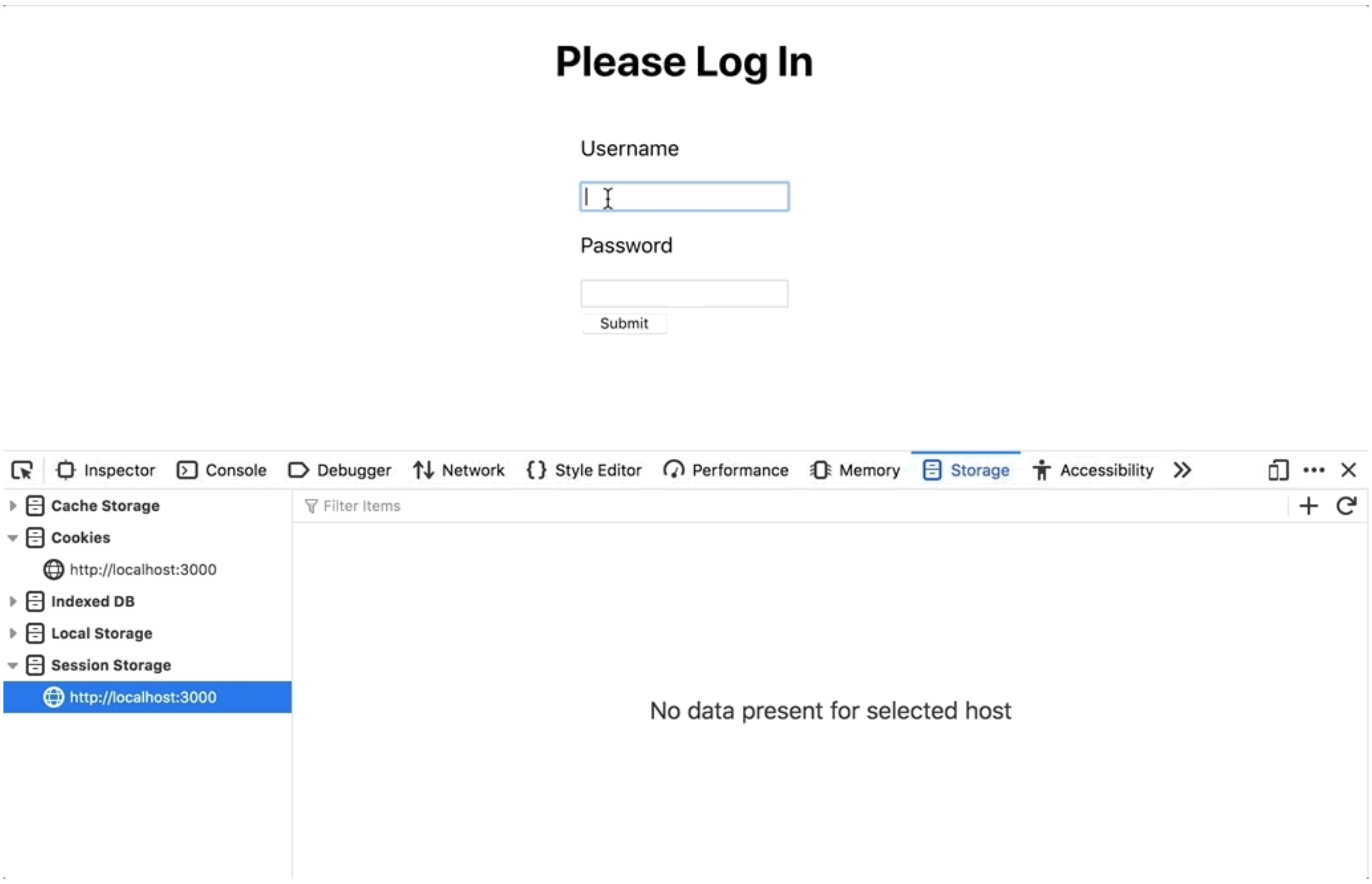


Application

Dashboard

Clear out the token by either deleting the token in the **Storage** tab in your developer tools or by typing `sessionStorage.clear()` in your developer console.

There’s a little problem now. When you log in, the browser saves the token, but you still see the login page.



The problem is your code never alerts React that the token retrieval was successful. You'll still need to set some state that will trigger a re-render when the data changes. Like most problems in React, there are multiple ways to solve it. One of the most elegant and reusable is to create a custom Hook.

Creating a Custom Token Hook

A custom Hook is a function that wraps custom logic. A custom Hook usually wraps one or more built-in React Hooks along with custom implementations. The primary advantage of a custom Hook is that you can remove the implementation logic from the component and you can reuse it across multiple components.

By convention, custom Hooks start with the keyword `use*`.
Open a new file in the `App` directory called `useToken.js`:

```
nano src/components/App/useToken.js

```
Copy

This will be a small Hook and would be fine if you defined it directly in `App.js`. But moving the custom Hook to a different file will show how Hooks work outside of a component. If you start to reuse this Hook across multiple components, you might also want to move it to a separate directory.
Inside `useToken.js`, import `useState` from `react`. Notice that you do not need to import `React` since you will have no JSX in the file. Create and export a function called `useToken`. Inside this function, use the `useState` Hook to create a `token` state and a `setToken` function:

auth-tutorial/src/components/App/useToken.js

```
import { useState } from 'react';
```

```

export default function useToken() {
  const [token, setToken] = useState();

}
```
Copy

Next, copy the `getToken` function to `useHook` and convert it to an [arrow function](), since you placed it inside `useToken`. You could leave the function as a standard, named function, but it can be easier to read when top-level functions are standard and internal functions are arrow functions. However, each team will be different. Choose one style and stick with it.
Place `getToken` before the state declaration, then initialize `useState` with `getToken`. This will fetch the token and set it as the initial state:

auth-tutorial/src/components/App/useToken.js

```
import { useState } from 'react';

export default function useToken() {
  const getToken = () => {
    const tokenString = sessionStorage.getItem('token');
    const userToken = JSON.parse(tokenString);
    return userToken?.token
  };
  const [token, setToken] = useState(getToken());

}
```
Copy

Next, copy the `setToken` function from `App.js`. Convert to an arrow function and name the new function `saveToken`. In addition to saving the token to `sessionStorage`, save the token to state by calling `setToken`:

auth-tutorial/src/components/App/useToken.js

```

import { useState } from 'react';

export default function useToken() {
  const getToken = () => {
    const tokenString = sessionStorage.getItem('token');
    const userToken = JSON.parse(tokenString);
    return userToken?.token
  };

  const [token, setToken] = useState(getToken());

  const saveToken = userToken => {
    sessionStorage.setItem('token', JSON.stringify(userToken));
    setToken(userToken.token);
  };
}
```
Copy

Finally, return an object that contains the `token` and `saveToken` set to the `setToken` property name. This will give the component the same interface. You can also return the values as an array, but an object will give users a chance to destructure only the values they want if you reuse this in another component.

auth-tutorial/src/components/App/useToken.js

```
import { useState } from 'react';

export default function useToken() {
  const getToken = () => {
    const tokenString = sessionStorage.getItem('token');
    const userToken = JSON.parse(tokenString);
    return userToken?.token
  };

  const [token, setToken] = useState(getToken());

  const saveToken = userToken => {
    sessionStorage.setItem('token', JSON.stringify(userToken));
    setToken(userToken.token);
  };

  return {
    setToken: saveToken,
    token
  }
}
```
Copy

Save and close the file.

Next, open `App.js`:

```
nano src/components/App/App.js

```
Copy

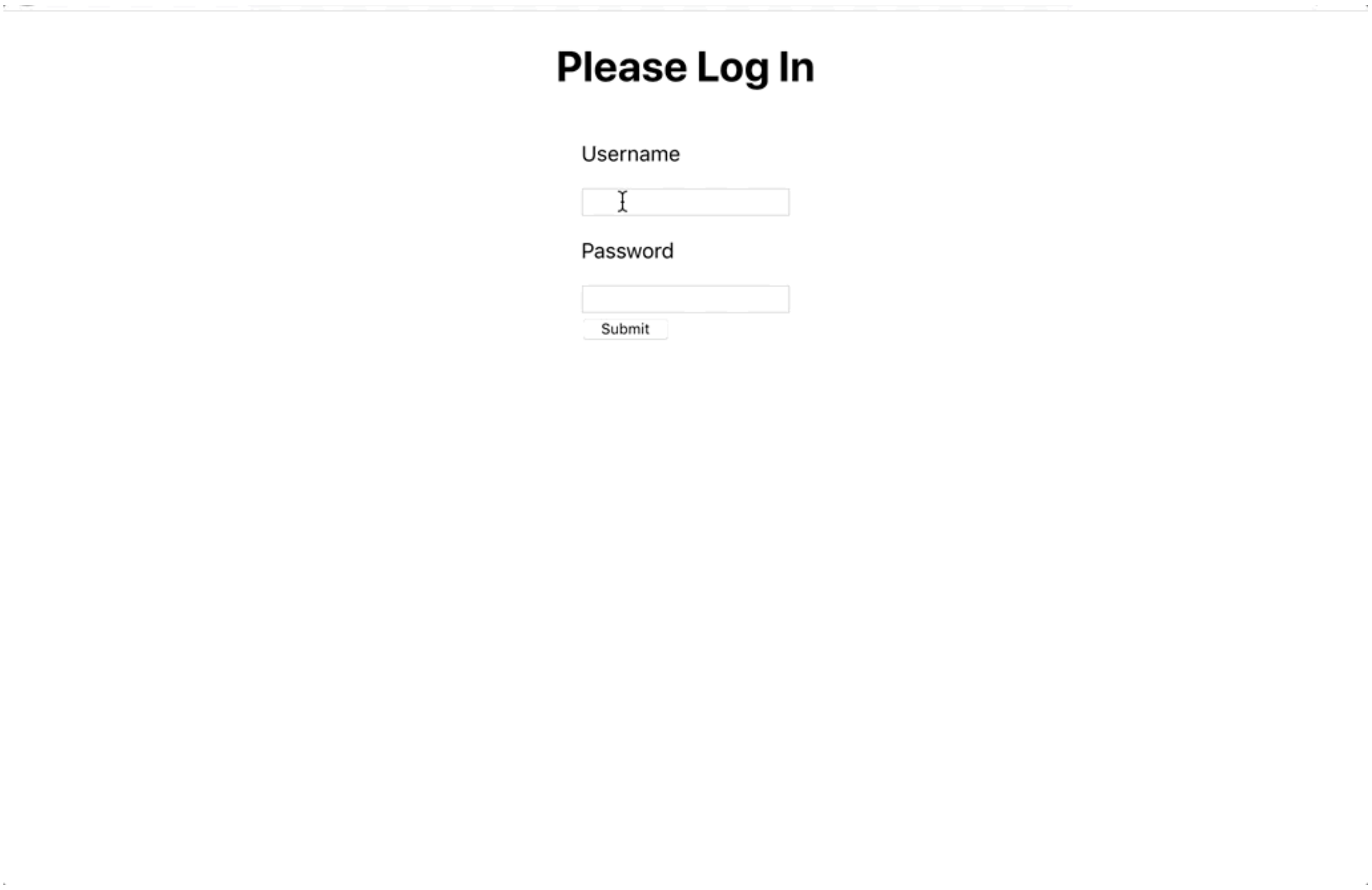
Remove the `getToken` and `setToken` functions. Then import `useToken` and call the function destructuring the `setToken` and `token` values. You can also remove the import of `useState` since you are no longer using the Hook:

auth-tutorial/src/components/App/App.js

```
import React from 'react';
import { BrowserRouter, Route, Switch } from 'react-router-dom';
import './App.css';
import Dashboard from '../Dashboard/Dashboard';
import Login from '../Login/Login';
import Preferences from '../Preferences/Preferences';
import useToken from './useToken';
```

```
function App() {

  const { token, setToken } = useToken();

  if(!token) {
    return <Login setToken={setToken} />
  }

  return (
    <div className="wrapper">
      <h1>Application</h1>
      <BrowserRouter>
        <Switch>
          <Route path="/dashboard">
            <Dashboard />
          </Route>
          <Route path="/preferences">
            <Preferences />
          </Route>
        </Switch>
      </BrowserRouter>
    </div>
  );
}

export default App;
```
Copy

Save and close the file. When you do, the browser will refresh, and when you log in, you will immediately go to the page. This is happening because you are calling `useState` in your custom Hook, which will trigger a component re-render:



You now have a custom Hook to store your token in `sessionStorage`. Now you can refresh your page and the user will remain logged in. But if you try to open the application in another tab, the user will be logged out. `sessionStorage` belongs only to the specific window session. Any data will not be available in a new tab and will be lost when the active tab is closed. If you want to save the token across tabs, you'll need to convert to `localStorage`.

Using `localStorage` to Save Data Across Windows

Unlike `sessionStorage`, `localStorage` will save data even after the session ends. This can be more convenient, since it lets users open multiple windows and tabs without a new login, but it does have some security problems. If the user shares their computer, they will remain logged in to the application even though they close the browser. It will be the user's responsibility to explicitly log out. The next user would have immediate access to the application without a login. It's a risk, but the convenience may be worth it for some applications.

To convert to `localStorage`, open `useToken.js`:

```
nano src/components/App/useToken.js

```

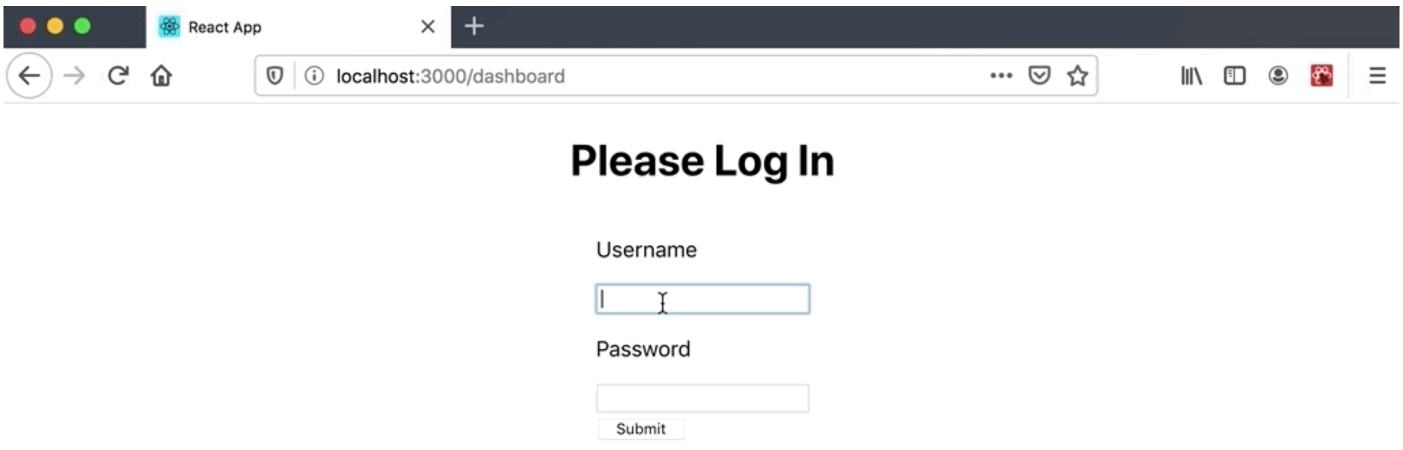
Copy

Then change every reference of `sessionStorage` to `localStorage`. The methods you call will be the same:

auth-tutorial/src/components/App/useToken.js

```
import { useState } from 'react';

export default function useToken() {
  const getToken = () => {
    const tokenString = localStorage.getItem('token');
    const userToken = JSON.parse(tokenString);
    return userToken?.token
  };

  const [token, setToken] = useState(getToken());

  const saveToken = userToken => {
    localStorage.setItem('token', JSON.stringify(userToken));
    setToken(userToken.token);
  };

  return {
    setToken: saveToken,
    token
  }
}
```
Copy

Save the file. When you do, the browser will refresh. You will need to log in again since there is no token yet in `localStorage`, but after you do, you will remain logged in when you open a new tab.



Please Log In

Username

Password

Submit

In this step, you saved tokens with `sessionStorage` and `localStorage`. You also created a custom Hook to trigger a component re-render and to move component logic to a separate function. You also learned about how `sessionStorage` and `localStorage` affect the user's ability to start new sessions without login.

Conclusion

Authentication is a crucial requirement of many applications. The mixture of security concerns and user experience can be intimidating, but if you focus on validating data and rendering components at the correct time, it can become a lightweight process.

Each storage solution offers distinct advantages and disadvantages. Your choice may change as your application evolves. By moving your component logic into an abstract custom Hook, you give yourself the ability to refactor without disrupting existing components.