THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND
ENGINEERING

COMPSCI 732

SOFTENG 750

Single-Page Applications with React

Agenda

- Single-Page Apps (SPAs) and routing
- React portals
- Global application state
- Persistent application state
- Aside: Third-party component libraries
  - Material UI

CS 732 / SE 750 – Module Two

Single-Page Applications

Single-Page Applications

A Single-Page Application (SPA) is "*a web application that requires only a single page load in a web browser*".

Single-Page Applications

- Web browsers fully load an SPA only once, when a user first navigates to the site.
- Any required updates to the page after this point are handled by JavaScript code
- Resources (HTML / CSS / JS) are loaded once – only data is transmitted back and forth.

Comparison with traditional (multi-page) apps

Benefits

- **Fast and responsive** – Usually much faster to load and use compared to traditional webapps, as only data is transferred during usage, rather than resources.
- **Caching** – As the entire functionality of the website is script-based, these webapps can function offline after the initial load. Data received from the server can be cached, and updated when web connectivity resumes.
- **Debugging** – Purpose-built browser tools such as React Dev Tools allow for an experience more like an IDE, which isn't possible for more traditional webapps.

Drawbacks

- **Search Engine Optimization (SEO)** – Web crawlers are optimized for traditional web pages – SPA's may not be indexed correctly.
- **Browser history** – Careful programming is required to maintain a user's history of interaction through a site, and to allow correct use of the "back" button.
- **Security** – the more functionality is handled by the client, the more care needs to be taken not to provide clients with functionality they're not permitted to use.

CS 732 / SE 750 – Module Two

Routing with React Router

Routing

- **Routing** refers to the mapping of a URL entered into the browser, to a specific webpage or endpoint
  - **Server-side routing** – the browser sends a request to a URL, the server routes that request to the appropriate endpoint based on the URL path
  - **Client-side routing** – A URL change does *not* result in a server request; the page contents are updated in JavaScript

Routing

- SPAs require **both** kinds of routing to be effective
  - No page reloads during normal operation → client-side routing required
  - The "refresh" button requires a page reload; users may wish to jump to a specific app point via URL → server-side routing required
- One approach to this problem:
  - All server-side requests route to, e.g. *index.html*
  - The remaining routing is all handled client-side via examining and modifying the URL using the history API
  - Works well with React, which only necessitates a single HTML template being loaded

React router

- There are several ways we can achieve client-side routing with React.
- React Router is one of the most popular approaches, and can be installed using the following yarn command:

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

- This package adds React components and hooks (`BrowserRouter`, `Link`, `NavLink`, `Routes`, `Route`, `useLocation`, and `useNavigate`, amongst others) which:
  - Correctly allow the generation of hyperlinks (`<a>`)
  - Define routes
  - Abstract away the challenges working with the History API
- **Note:** This course covers React Router **version 6** (the latest version as of January 2022). The code is not backwards compatible with older versions!

Simple example

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

export default function App() {
  return (
    <BrowserRouter>
      <div>
        <Routes>

          <Route path="/page1"
            element={<p>Page One</p>} />

          <Route path="/page2"
            element={<p>Page Two</p>} />

          <Route path="*"
            element={<p>404 Not Found!!</p>} />
        </Routes>
      </div>
    </BrowserRouter>
  );
}
```

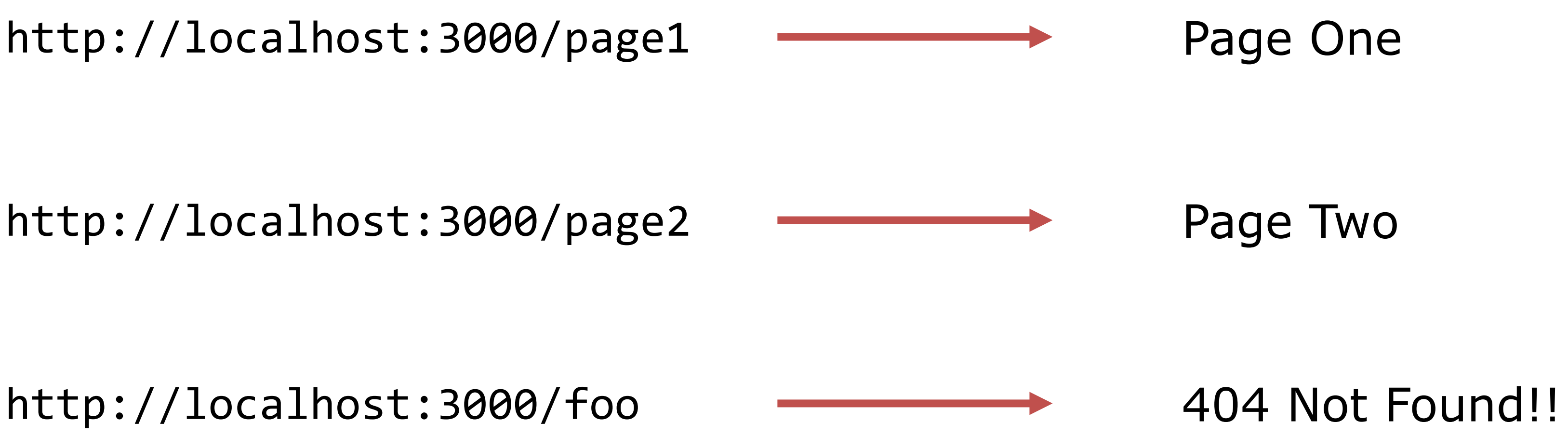
Import all necessary components

Surround entire app with `<BrowserRouter></BrowserRouter>`

All `<Route>`s inside this `<Routes>` block will be evaluated, in the order they're written. The **`element` prop** of the first matching route will be rendered.

* matches anything, so is good practice to have a default in-case of a user entering an invalid URL

Simple example



`http://localhost:3000/page1` → Page One

`http://localhost:3000/page2` → Page Two

`http://localhost:3000/foo` → 404 Not Found!!

Shared content

- Any components *outside* of the `<Routes>` block will be rendered for all routes
  - Can be useful for adding page headers / footers / etc.

```
<BrowserRouter>
   <div>

      <header>
        <h1>My Website</h1>
      </header>

      <Routes> … </Routes>

   </div>
</BrowserRouter>
```

This `<header>` will be rendered no matter the route

Navigation

- To allow the user to navigate between pages, we use the `Link` component

```
<header>
   <h1>My Website</h1>
      <nav>
         <Link to="/page1">Page One</Link>
         <Link to="/page2">Page Two</Link>
      </nav>
</header>
```

- The `Link` component will render a hyperlink (`<a>`) in the browser which, when clicked, will cause client-side navigation to the path specified with the `to` property

Absolute vs Relative paths

- Any path / to props in React Router can be either absolute or relative:
- **Absolute** paths start with / and are relative to the application root.
  - Example: A link to `/hello`, *anywhere* in a webapp served from http://localhost:3000/, would navigate to exactly http://localhost:3000/hello.
- **Relative** paths don't start with `/`, and are relative to *their nearest ancestor React Router component*.
  - Example:

```
<Routes>
  <Route path="/page1"
    element={<Link to="hello">Click me!</Link>} />
  <Route path="/page2"
    element={<p>Page Two</p>} />
  <Route path="*"
    element={<p>404 Not Found!!</p>} />
</Routes>
```

This relative link to "`hello`", will actually link to `/page1/hello`, because it's inside the "`/page1`" route.

Navigation

- We can also use `NavLink`, which is intended to allow us to see a visual difference between an “active” and “inactive” link.
  - Active links are those which match the user’s current location within the website
  - Inactive links are those which are not active.
  - `NavLink`’s `className` and `style` props can accept a **function** rather than just a simple string (for `className`) or JS object (for `style`). The function takes an object with an isActive property as an argument, and returns the class name / style to apply. Therefore, we can return different values depending on whether a link is active or not.

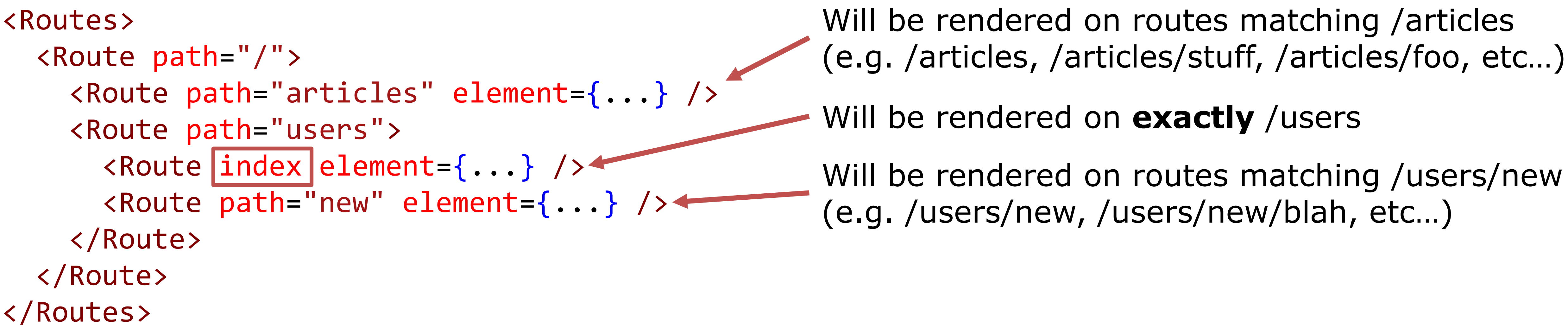
```
<NavLink to="articles" className={({ isActive }) => isActive ? "active" : "inactive"} />
```

We can use the isActive boolean to conditionally change the class name / style to visually alter our links when required.

```
<NavLink to="about" style={({ isActive }) => ({ color: isActive ? 'red' : 'black' })} />
```

Nesting routes

- Within our `<Routes>` block, we can nest `<Route>` components inside each other.
  - The matching path, causing that route's `element` to be rendered, will be constructed from that route's path, and all its parent paths
  - A route can be marked as the `index` route, in which case it will be rendered only if its parent route exactly matches.



```
<Routes>
  <Route path="/">
    <Route path="articles" element={...} />
    <Route path="users">
      <Route index element={...} />
      <Route path="new" element={...} />
    </Route>
  </Route>
</Routes>
```

Will be rendered on routes matching /articles (e.g. /articles, /articles/stuff, /articles/foo, etc…)

Will be rendered on **exactly** /users

Will be rendered on routes matching /users/new (e.g. /users/new, /users/new/blah, etc…)

Nesting routes – rendering parent elements

- If parents of matching routes have element props defined, they will be rendered too! The elements of child routes will be rendered *inside* the elements of parent routes.
- This can allow us to incrementally build up the final webpage that appears to the user, based on the route.
- For example, in the following code, if the user navigates to `/users/new`:

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

    <Route path="articles" element={<ArticlesPage />} />

    <Route path="users" element={<UsersPage />}>
      <Route path="new" element={<NewUserForm />} />
    </Route>
  </Route>
</Routes>
```

<PageWithNavbar> will be rendered

<UsersPage> will be rendered *inside* the <PageWithNavbar>

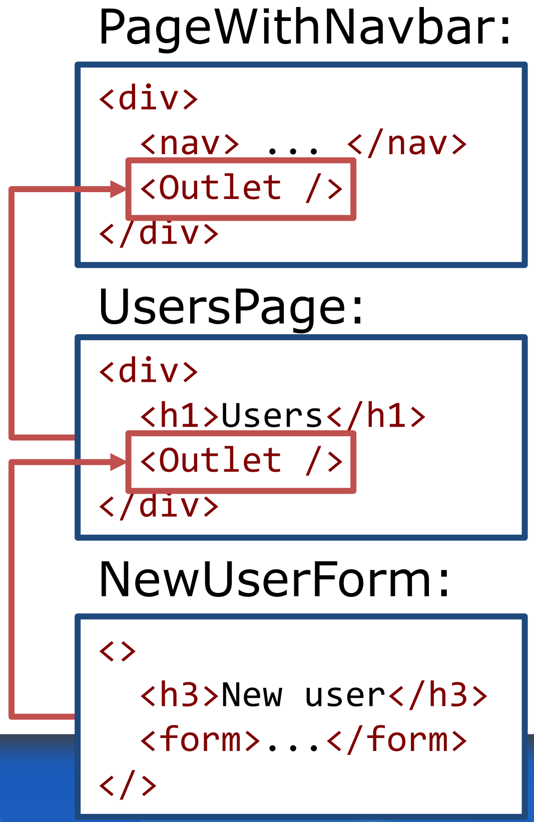
<NewUserForm> will be rendered *inside* the <UsersPage>

But wait: How does React Router know *where* within the parent elements to render the child elements?

Nesting routes - <Outlet>

- To properly enable rendering like on the previous slide, we give the parent routes an `<Outlet>` component. React Router will render the matching child route (if any) inside that outlet.

- Continuing from the `/users/new` example:



PageWithNavbar:

```
<div>
  <nav> ... </nav>
  <Outlet />
</div>
```

UsersPage:

```
<div>
  <h1>Users</h1>
  <Outlet />
</div>
```

NewUserForm:

```
<>
  <h3>New user</h3>
  <form>...</form>
</>
```

Final HTML:

```
<div>
  <nav>...</nav>
  <div>
    <h1>Users</h1>
    <h3>New user</h3>
    <form>...</form>
  </div>
</div>
```

Programmatic navigation – <Navigate />

- Using the `Navigate` component, we can specify that when we navigate to a certain Route, we automatically redirect to an alternative URL.
- In the following example, if we navigate to exactly **/**, automatically redirect the user to `/articles`.

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

    <Route index element={<Navigate to="articles" />} />

    <Route path="articles" element={<ArticlesPage />} />
    <Route path="users" element={<UsersPage />}>
      <Route path="new" element={<NewUserForm />} />
    </Route>
  </Route>
</Routes>
```

Programmatic navigation – <Navigate />

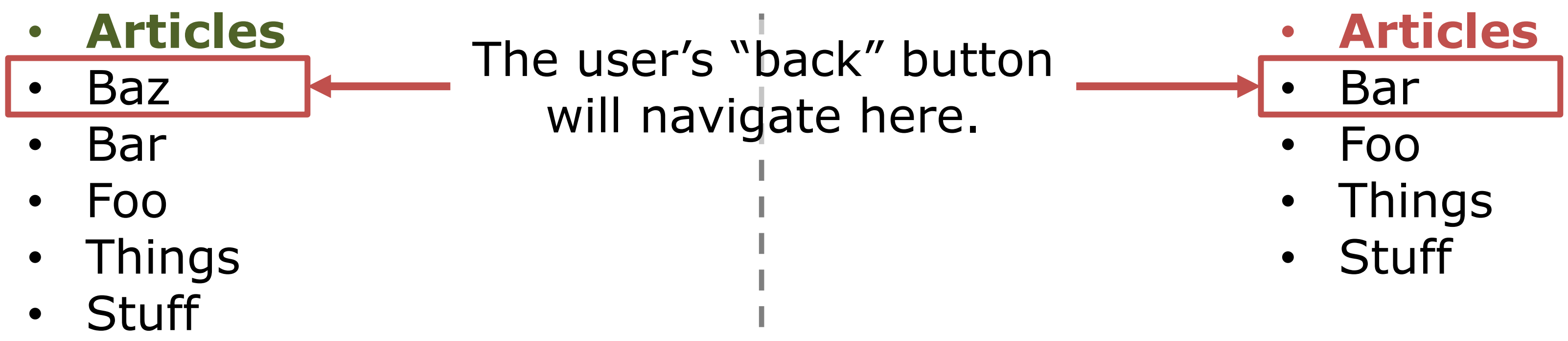
- `<Navigate>` has a boolean prop called replace (defaults to false).
- If set to true, the new URL will *replace* the current URL in the browser history stack, rather than being pushed onto the stack as usual
  - This affects what will happen when the user presses the “back” button on the browser:

```
<Navigate to="articles" />
```

```
<Navigate to="articles" replace />
```

User’s browser history:

User’s browser history:



- **Articles**
- Baz
- Bar
- Foo
- Things
- Stuff

The user’s “back” button will navigate here.

- **Articles**
- Bar
- Foo
- Things
- Stuff

Programmatic navigation – useNavigate()

- Sometimes, we want to be able to programmatically navigate through our webapp, rather than relying on user interaction or `<Navigate>`.
- To do this, we can use the `useNavigate()` hook.
  - The hook gives us back a function that we can use to perform navigation.
  - The function takes two arguments:
    - The first argument specifies where to navigate – similar to <Navigate>'s to prop
    - The second argument specifies config options, including whether to replace the top of the user's browser history stack (identical to the example on the previous slide).

```
const navigate = useNavigate();

navigate("articles");

navigate("articles", { replace: true });
```

Path parameters

- It is common for us to want to use a placeholder for part of a URL, and use the value that's supplied to that placeholder later
  - For example, we might want `/articles/1`, `/articles/2`, etc. to map to the same route, and then use the supplied value to grab the data for a particular article.
- To do this, we use **path parameters**. These begin with a colon (e.g. :id), and will match anything at that point in the URL. For example:

```
<Routes>
  <Route path="/" element={<PageWithNavbar />}>
    <Route index element={<Navigate to="articles" />} />
    <Route path="articles" element={<ArticlesPage />}>

      <Route path=":id" element={<ArticleView />} />

    </Route>
  </Route>
</Routes>
```

Will match anything at this point in the route hierarch, e.g. `/articles/`**`1`**, `/articles/`**`2`**, etc. The values supplied in the path (1, 2, etc.) will be saved as the path param named "**id**".

Path parameters

- Within any component rendered as a descendent of a `Route` with a path param, we can access the value of those path params using the `useParams()` hook.

```
        …
          <Route path=":id" element={<ArticleView />} />
```

These names must match.

```
function ArticleView() {
  const { id } = useParams();
  const article = articles.find((article) => article.id == id);
  return (
    <>
      <h3>{article.title}</h3>
      <p>{article.content}</p>
    </>
  );
}
```

Now, if we navigate to `/articles/`**`1`**, this `ArticleView` will be rendered, and the value of "`id`" will be **1**.

CS 732 / SE 750 – Module Two

Global state with React’s Context mechanism

Global state

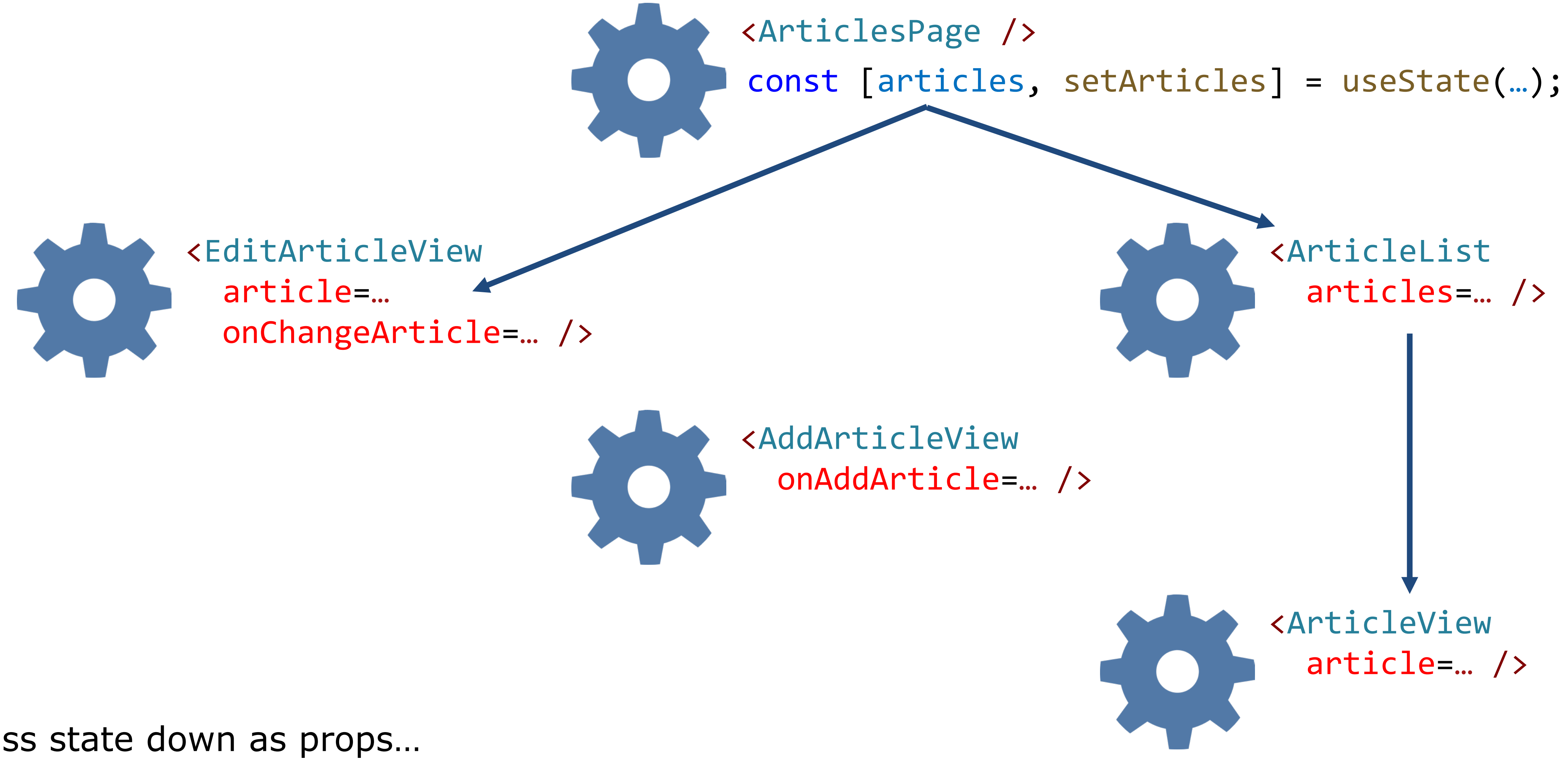
- We have learned how to give components *local* state – using the `useState()` hook
- What if we have state which we need to share with large parts of our application?
  - E.g. a list of articles / to-do items / calendar events
  - Would need to be accessed from (at minimum) the view / add / edit pages for those items…

Models for global state – Top-level storage

- Also known as “moving state up”

1. Store state at a level in the component hierarchy, such that all components needing to access that state are descendants of the stateful component
2. Pass state “down” to child components as props
3. Pass mutations “up” to parents as events

Models for global state – Top-level storage



```
<ArticlesPage />
const [articles, setArticles] = useState(…);
```

```
<EditArticleView
  article=…
  onChangeArticle=… />
```

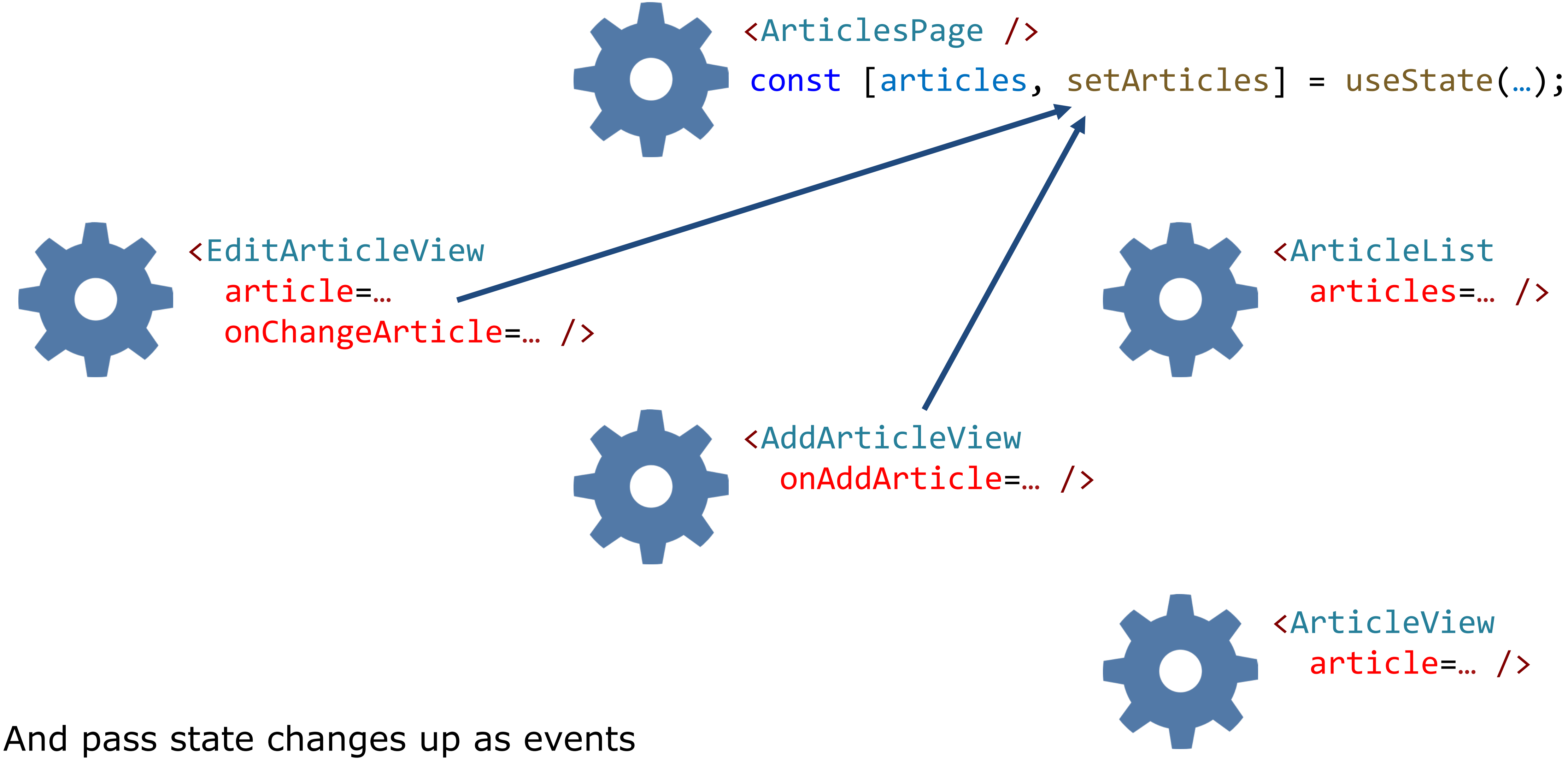
```
<ArticleList
  articles=… />
```

```
<AddArticleView
  onAddArticle=… />
```

```
<ArticleView
  article=… />
```

Pass state down as props…

Models for global state – Top-level storage



```
<ArticlesPage />
const [articles, setArticles] = useState(…);
```

```
<EditArticleView
  article=…
  onChangeArticle=… />
```

```
<ArticleList
  articles=… />
```

```
<AddArticleView
  onAddArticle=… />
```

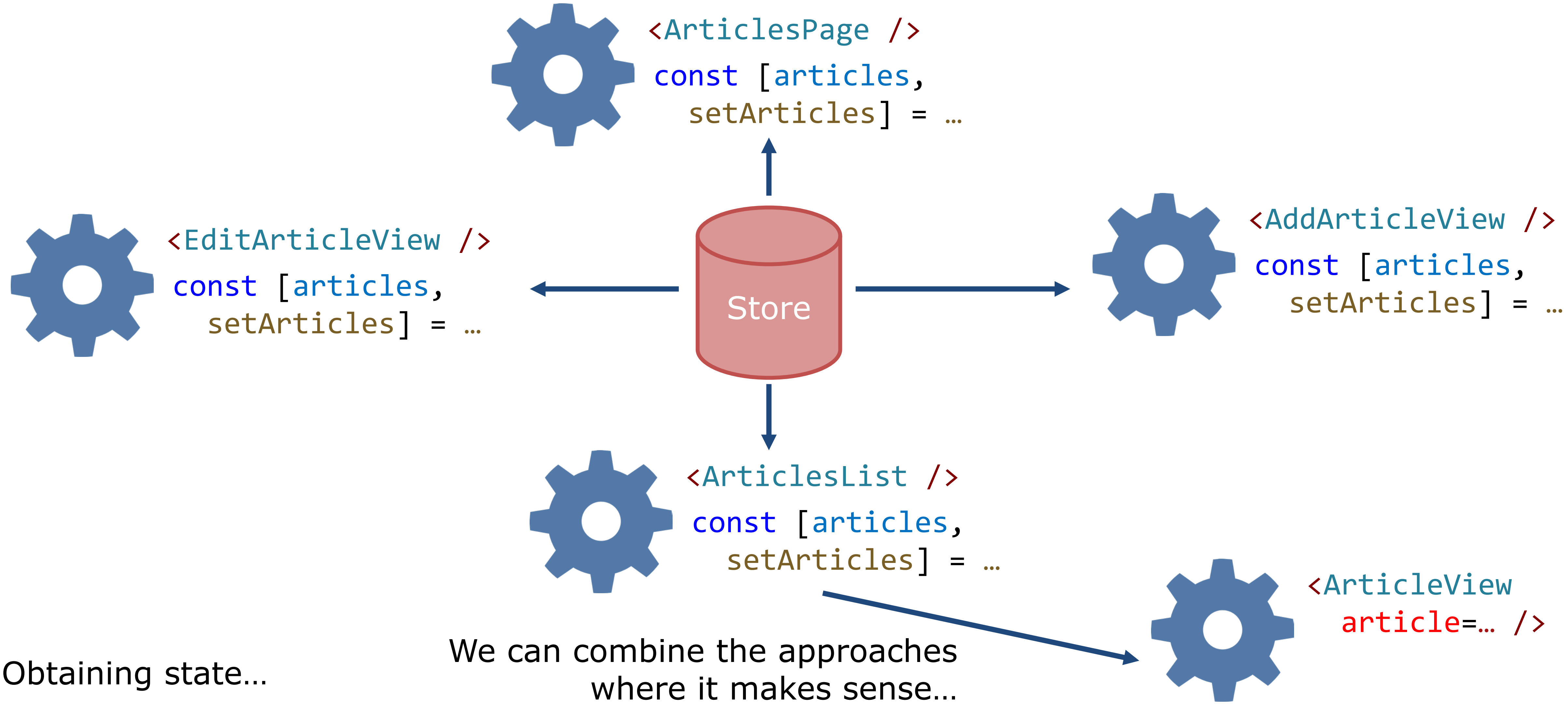
```
<ArticleView
  article=… />
```

… And pass state changes up as events

Models for global state – Centralized storage

1. State is held in a central “store”, accessible from all components
2. State changes are dispatched to the store, which then notifies all observers to update themselves

Models for global state – Centralized storage



```
<ArticlesPage />
const [articles,
  setArticles] = …
```

```
<EditArticleView />
const [articles,
  setArticles] = …
```

```
<AddArticleView />
const [articles,
  setArticles] = …
```

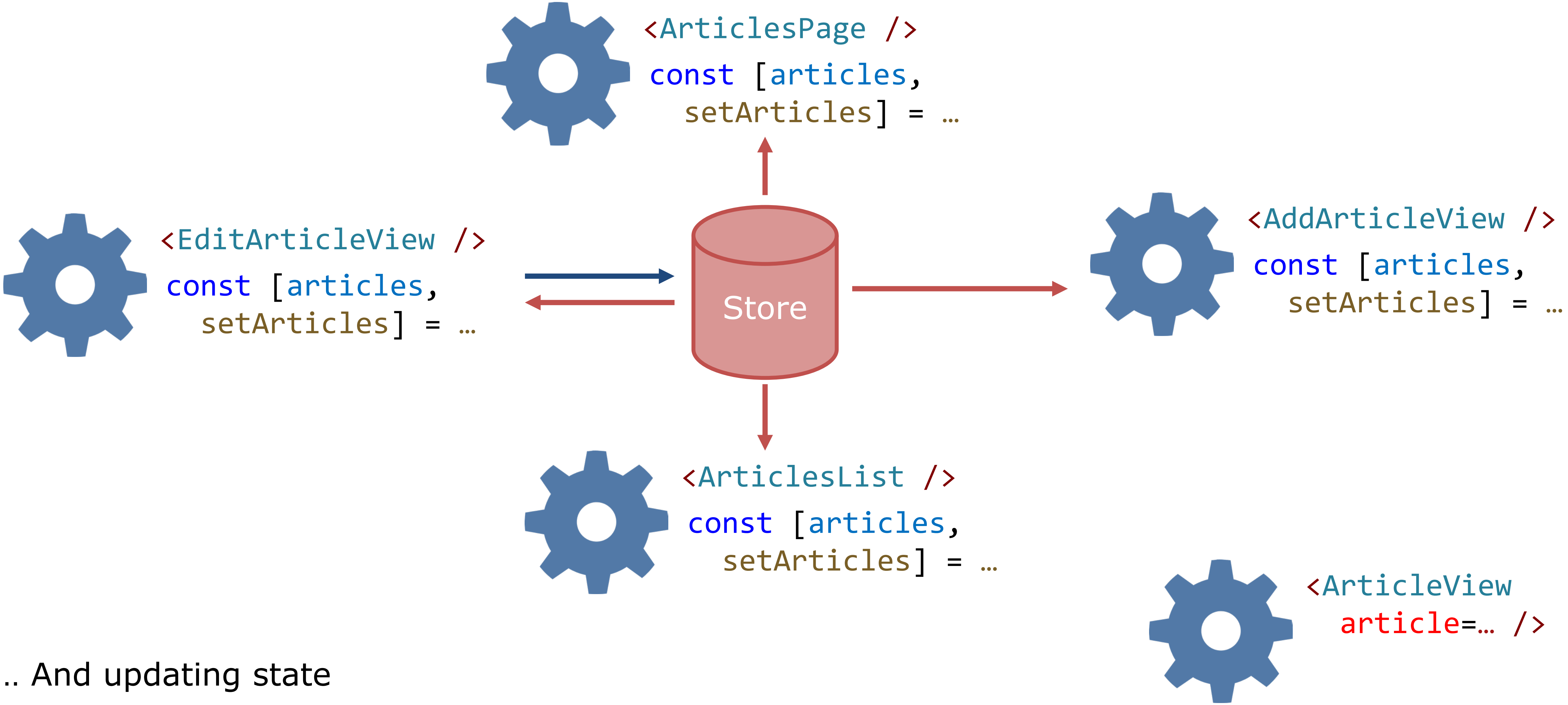
```
<ArticlesList />
const [articles,
  setArticles] = …
```

```
<ArticleView
  article=… />
```

We can combine the approaches
where it makes sense…

Obtaining state…

Models for global state – Centralized storage



```
<ArticlesPage />
const [articles,
  setArticles] = …
```

```
<EditArticleView />
const [articles,
  setArticles] = …
```

```
<AddArticleView />
const [articles,
  setArticles] = …
```

```
<ArticlesList />
const [articles,
  setArticles] = …
```

```
<ArticleView
  article=… />
```

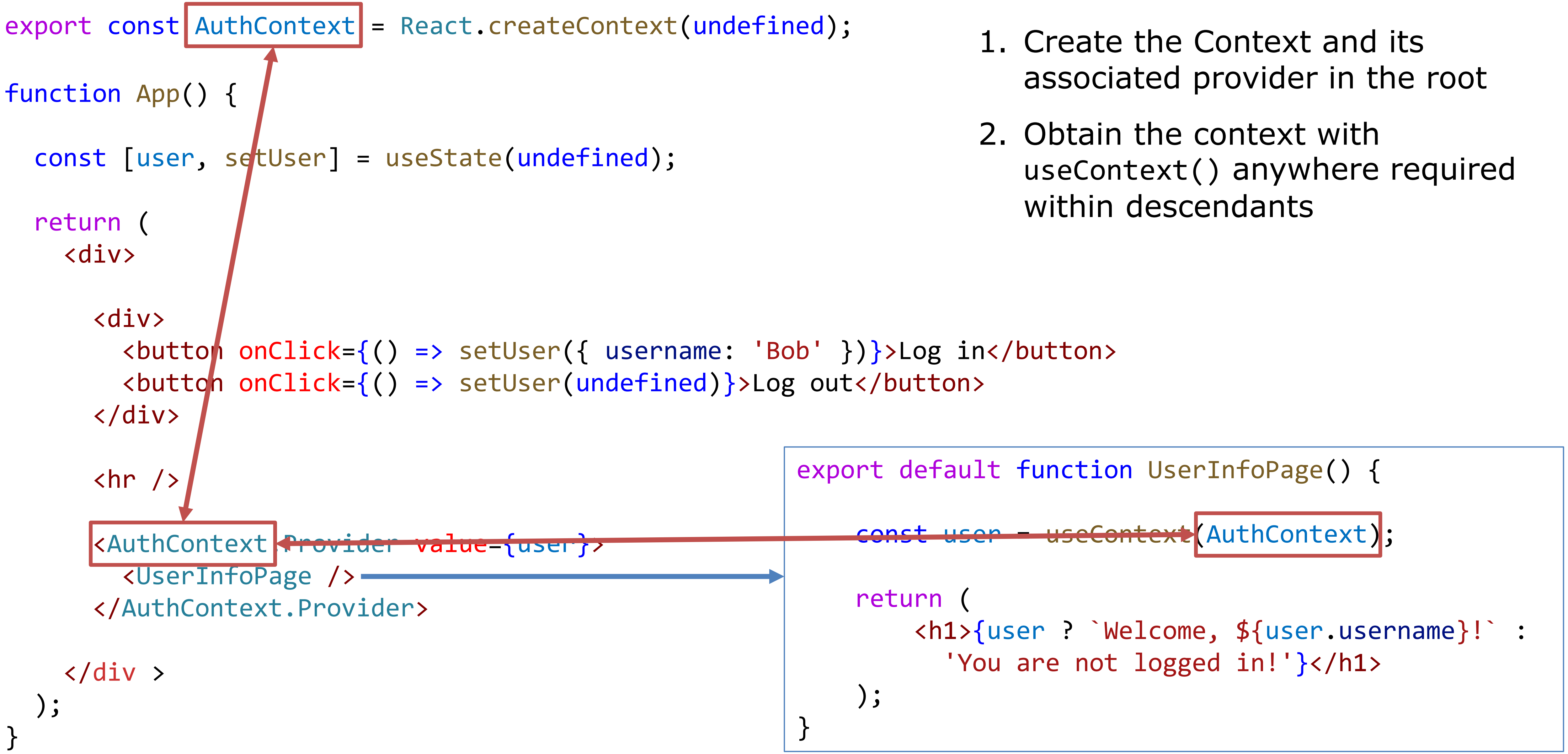
… And updating state

React Context

- How it works:
  1. Create a Context object using `React.createContext()`
  2. Wrap our React components in a `<Context.Provider>`, supplying some value for the context
  3. Any descendants of that Provider will be able to access the context value without having it passed to them as props
  4. Whenever the Provider’s value changes, it (and all descendants) will be re-rendered, giving them access to the new value

Example 1 – Modifying context from root

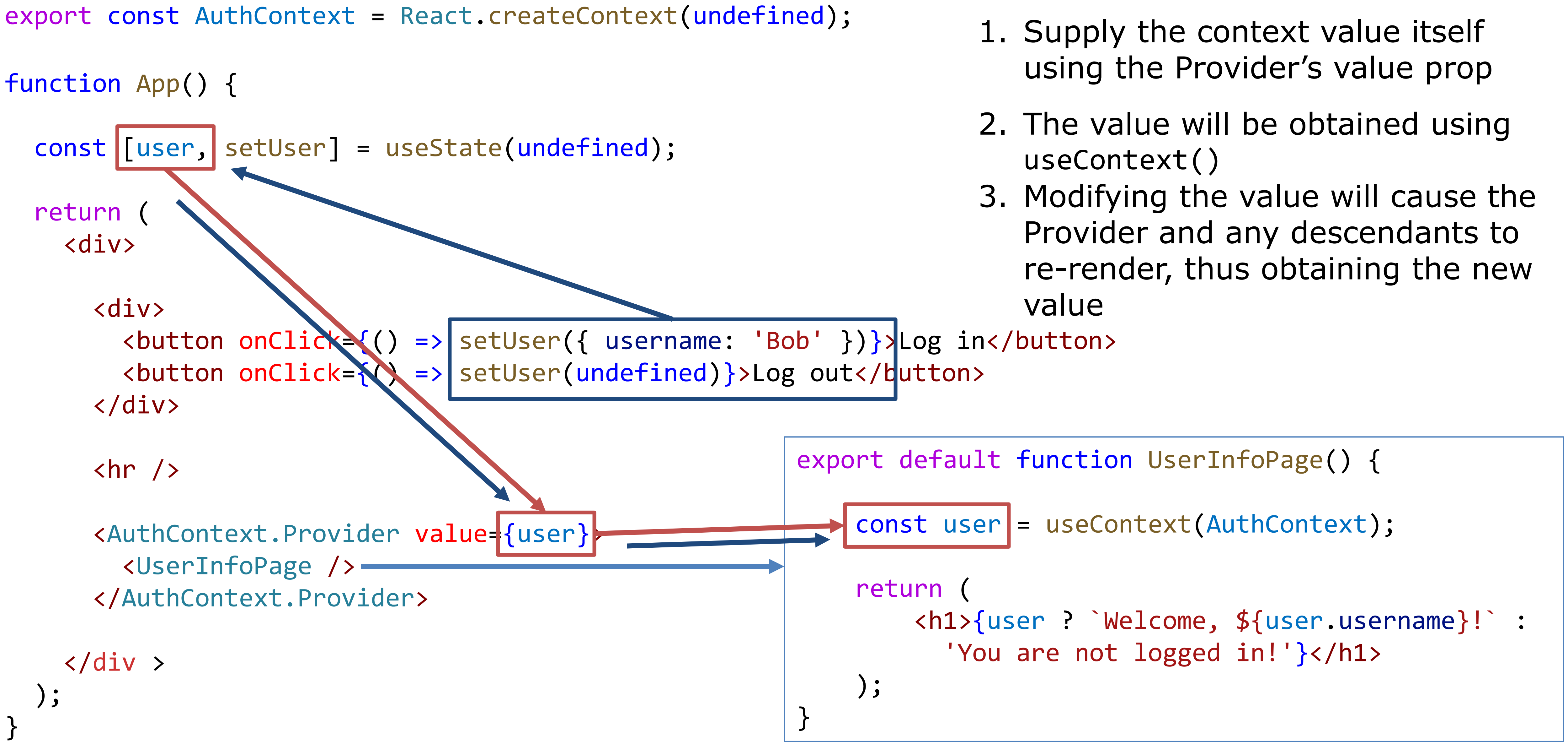
1. Create the Context and its associated provider in the root
2. Obtain the context with useContext() anywhere required within descendants



```
export const AuthContext = React.createContext(undefined);

function App() {

  const [user, setUser] = useState(undefined);

  return (
    <div>

      <div>
        <button onClick={() => setUser({ username: 'Bob' })}>Log in</button>
        <button onClick={() => setUser(undefined)}>Log out</button>
      </div>

      <hr />

      <AuthContext.Provider value={user}>
        <UserInfoPage />
      </AuthContext.Provider>

    </div >
  );
}
```

```
export default function UserInfoPage() {

    const user = useContext(AuthContext);

    return (
        <h1>{user ? `Welcome, ${user.username}!` :
          'You are not logged in!'}</h1>
    );
}
```

Example 1 – Modifying context from root



```
export const AuthContext = React.createContext(undefined);

function App() {

  const [user, setUser] = useState(undefined);

  return (
    <div>

      <div>
        <button onClick={() => setUser({ username: 'Bob' })}>Log in</button>
        <button onClick={() => setUser(undefined)}>Log out</button>
      </div>

      <hr />

      <AuthContext.Provider value={user}>
        <UserInfoPage />
      </AuthContext.Provider>

    </div >
  );
}
```

```
export default function UserInfoPage() {

    const user = useContext(AuthContext);

    return (
        <h1>{user ? `Welcome, ${user.username}!` :
          'You are not logged in!'}</h1>
    );
}
```

1. Supply the context value itself using the Provider’s value prop
2. The value will be obtained using useContext()
3. Modifying the value will cause the Provider and any descendants to re-render, thus obtaining the new value

Example 1 – Modifying context from root

```
export const AuthContext = React.createContext(undefined);

function App() {

  const [user, setUser] = useState(undefined);

  return (
    <div>

      <div>
        <button onClick={() => setUser({ username: 'Bob' })}>Log in</button>
        <button onClick={() => setUser(undefined)}>Log out</button>
      </div>

      <hr />

      <AuthContext.Provider value={user}>
        <UserInfoPage />
      </AuthContext.Provider>

    </div >
  );
}
```

1. Supply the context value itself using the Provider's value prop
2. The value will be obtained using useContext()
3. Modifying the value will cause the Provider and any children to re-render, thus obtaining the new value

```
export default function UserInfoPage() {

  const user = useContext(AuthContext);

  return (
    <h1>{user ? `Welcome, ${user.username}!` :
      'You are not logged in!'}</h1>
  );
}
```

Question: What if we want to *modify* the user from within a descendant component, not just access it?

Example 2 – Modifying context from a descendant

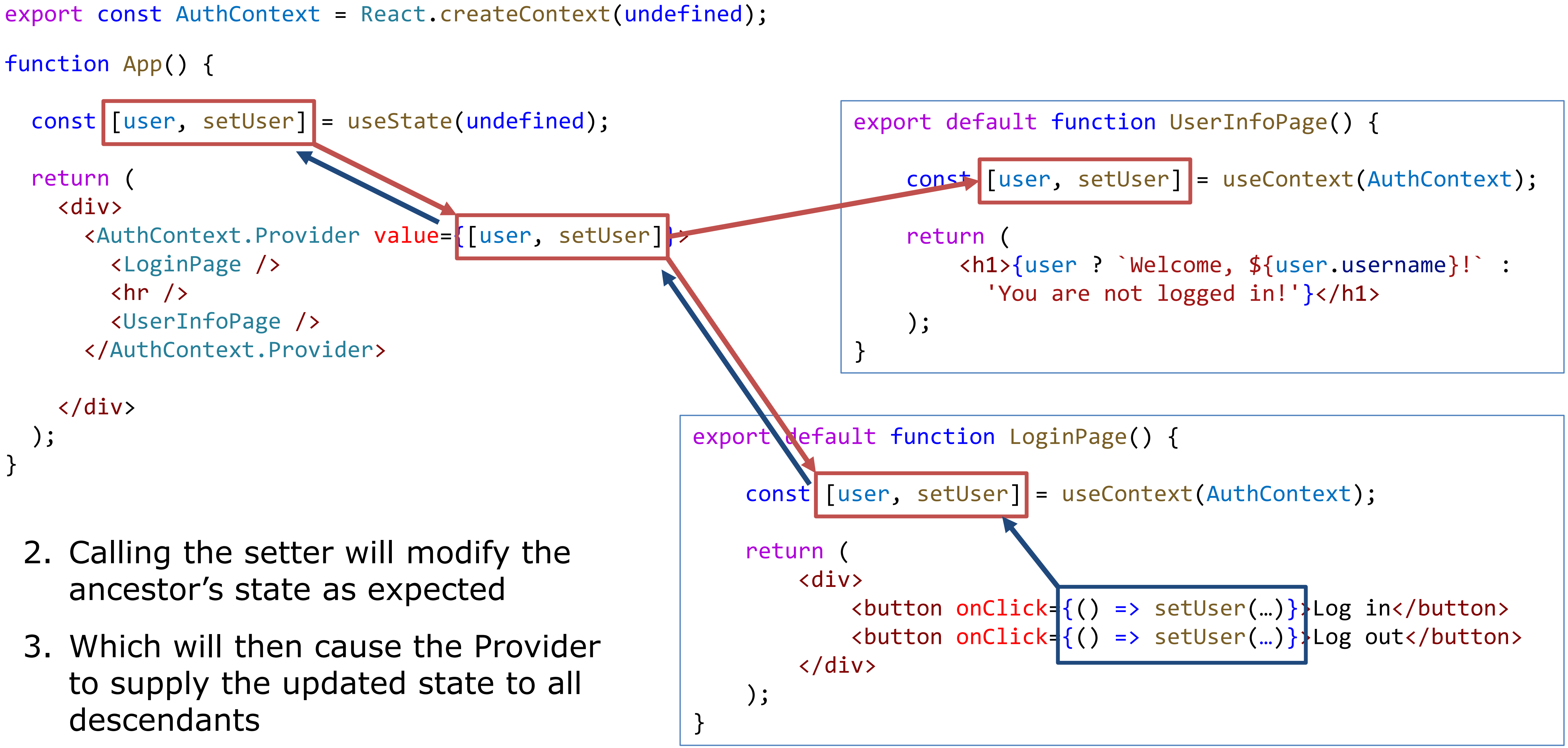
```
export const AuthContext = React.createContext(undefined);

function App() {

  const [user, setUser] = useState(undefined);

  return (
    <div>
      <AuthContext.Provider value={[user, setUser]}>
        <LoginPage />
        <hr />
        <UserInfoPage />
      </AuthContext.Provider>

    </div>
  );
}
```

```
export default function UserInfoPage() {

    const [user, setUser] = useContext(AuthContext);

    return (
        <h1>{user ? `Welcome, ${user.username}!` :
          'You are not logged in!'}</h1>
    );
}
```

```
export default function LoginPage() {

    const [user, setUser] = useContext(AuthContext);

    return (
        <div>
            <button onClick={() => setUser(…)}>Log in</button>
            <button onClick={() => setUser(…)}>Log out</button>
        </div>
    );
}
```

1. Supply context information through the Provider’s value prop as before – but this time, additionally supply the setter function

Example 2 – Modifying context from a descendant



```
export const AuthContext = React.createContext(undefined);

function App() {

  const [user, setUser] = useState(undefined);

  return (
    <div>
      <AuthContext.Provider value={[user, setUser]}>
        <LoginPage />
        <hr />
        <UserInfoPage />
      </AuthContext.Provider>

    </div>
  );
}
```

```
export default function UserInfoPage() {

    const [user, setUser] = useContext(AuthContext);

    return (
        <h1>{user ? `Welcome, ${user.username}!` :
          'You are not logged in!'}</h1>
    );
}
```

```
export default function LoginPage() {

    const [user, setUser] = useContext(AuthContext);

    return (
        <div>
            <button onClick={() => setUser(…)}>Log in</button>
            <button onClick={() => setUser(…)}>Log out</button>
        </div>
    );
}
```

2. Calling the setter will modify the ancestor’s state as expected

3. Which will then cause the Provider to supply the updated state to all descendants

“Clean” approach to using context

- There are many ways we could organize our use of context, state, hooks to provide the functionality we desire.
- It can be good practice (and “clean code”) to **encapsulate** the context for an app (both the stateful values and the functions to modify those values) in a *wrapper component* (or higher-order component)
- One of the provided examples (“encapsulating state”) shows one possible way of organizing this.
  - Check out, in particular, the `AppContextProvider` component

When to use local state vs context?

- **Local state:** Use when the state doesn't need to be shared with any other component
  - E.g. the state of a textbox in a form
- **Context:** Use when the state is required by many disparate components, to avoid passing props everywhere
  - E.g. user preferences, themes, authentication information
- **For most state:** Can use either method, depending on specific requirements & preferences
  - E.g. of these two methods, there's no right answer as to how we should be storing our articles list…

Other mechanisms for storing state

- Use a global state management system like Redux
  - Still very popular
  - Can do much of the same thing with the Context API, but
  - Can be better performance – avoids even more unnecessary re-renders
- Use local browser storage
  - Provides persistent state across page refreshes / reloads
  - Ideally need to account for different app versions

ENGINEERING

CS 732 / SE 750 – Module Two

Utilizing local browser storage

Local browser storage

- All modern browsers have *local storage*
  - A set of key-value pairs
  - Storage is local to a particular *origin* (protocol / hostname / port combination) – e.g. my app running at http://localhost:3000 can't access the local storage of https://www.google.com/.
- Can be accessed in Javascript through the `window.localStorage` global (or just `localStorage` for short)
- There is also `window.sessionStorage`
  - Works the same, except data stored within is local to a particular *browser tab*, and is cleared when that tab is closed

Usage in plain JavaScript (no React)

```html
<p>This page has been visited <span id="numVisits"></span> time(s) before!</p>
```

```javascript
const span = document.querySelector('#numVisits');

let numVisits = JSON.parse(localStorage.getItem('numVisits'));

if (!numVisits) {
    numVisits = 0;
}

span.innerText = numVisits;

numVisits++;
localStorage.setItem('numVisits', JSON.stringify(numVisits));
```

1. Gets the value with the given key, as a **string**

2. Converts the string to actual data

3. If the value didn’t exist in local storage, it will be null. We should account for this in our code.

4. Convert our data to save into a string

5. Save our string value to local storage with the given key

Local storage usage in React

- We can access local storage in React, exactly as with the previous slide!
  - Problem: If we update local storage, React won’t detect the change and thus will not re-render any component relying on it
- We can combine `localStorage` with `useState()` and `useEffect()` to allow React’s own state management to hook into local storage

Local storage usage in React

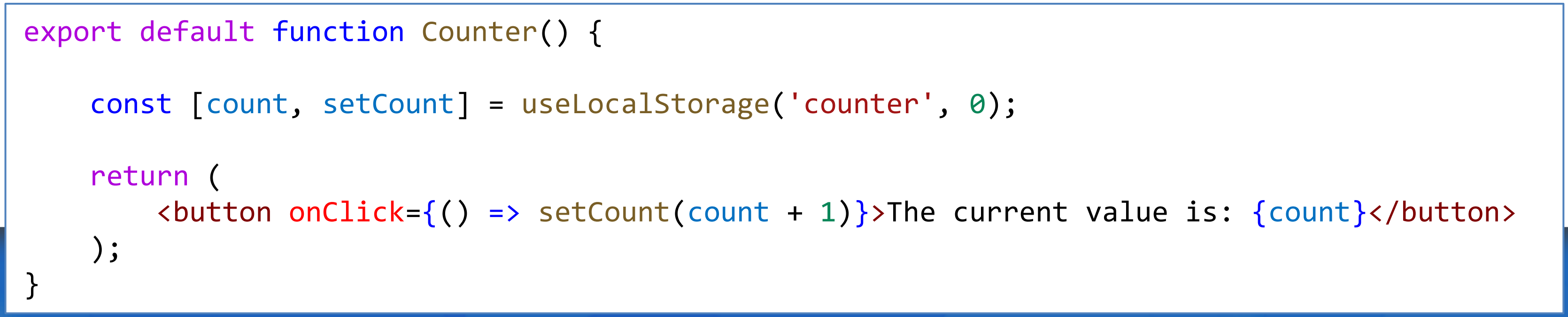
```
export function useLocalStorage(key, initialValue = null) {

    const [value, setValue] = useState(() => {
        try {
            const data = window.localStorage.getItem(key);
            return data ? JSON.parse(data) : initialValue;
        } catch {
            return initialValue;
        }
    });

    useEffect(() => {
        window.localStorage.setItem(key, JSON.stringify(value));
    }, [value, setValue])

    return [value, setValue];

}
```

1. Defining a custom hook for ease of reuse

2. This function will be run the first time this state is initialized; it will load the initial value from local storage if it's already there, or use the given initialValue if not.

3. As a side-effect, save whatever is the current value to local storage.

4. Usage of our custom hook is very similar to `useState()` itself.



```
export default function Counter() {

    const [count, setCount] = useLocalStorage('counter', 0);

    return (
        <button onClick={() => setCount(count + 1)}>The current value is: {count}</button>
    );
}
```

CS 732 / SE 750 – Module Two

Third-party component libraries

Third-party component libraries

- Many libraries exist offering a plethora of third-party React components we can use
- Install via npm
- Can offer:
  - Integration with other libraries, e.g. Redux providers
  - Standardized UI/UX experience without writing lots of custom CSS, e.g. Material UI, Ant Design…
- Many are free / open source! (though some are paid)

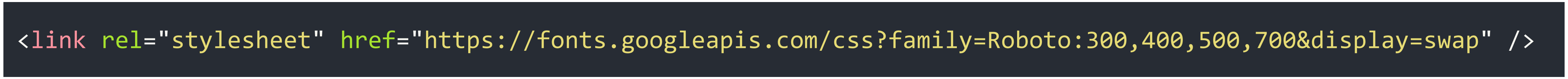
MUI

- MUI is one React component library giving developers access to many React components conforming to Google’s Material Design language

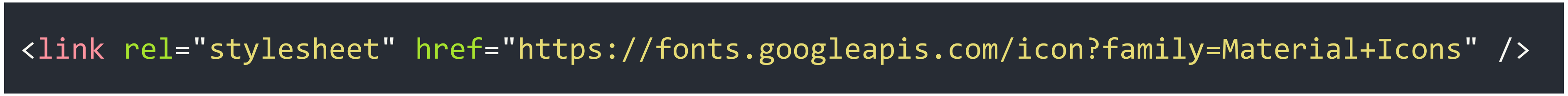
- Install as follows:

```
npm install @mui/material @emotion/react @emotion/styled
npm install @mui/icons-material
```

- Require Roboto and Icons fonts:



```
<link rel="stylesheet" href="https://fonts.googleapis.com/css?family=Roboto:300,400,500,700&display=swap" />
```



```
<link rel="stylesheet" href="https://fonts.googleapis.com/icon?family=Material+Icons" />
```

- Excellent resources available at: https://mui.com/getting-started/usage/

MUI – Example project

- Check out the MUI example in the examples repository to see some of what MUI can do!

Online resources

- React router
- React context API
- Local storage API