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
- Third-party component libraries
  - Material UI

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

ENGINEERING

CS 732 / SE 750 Lecture 02

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.

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

ENGINEERING

CS 732 / SE 750 Lecture 02

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 as an npm package using the following command:

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

- This package adds React components (`Router`, `Link`, `Switch`, `Route`) which:
  - Correctly allow the generation of hyperlinks (`<a>`)
  - Define routes
  - Abstract away the challenges working with the History API

Simple example

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

export default function App() {
  return (
    <Router>
      <div>
        <Switch>
          <Route path="/page1">
            <p>Page One</p>
          </Route>
          <Route path="/page2">
            <p>Page Two</p>
          </Route>
          <Route path="*">
            <p>404 Not Found!!</p>
          </Route>
        </Switch>

      </div>
    </Router>
  );
}
```

Import all necessary components

Surround entire app with <Router></Router>

All <Route>s inside this <Switch> will be evaluated, in the order they're written. The content inside the first one that matches 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 `<Switch>` will be rendered for all routes
  - Can be useful for adding page headers / footers / etc.

```
<Router>
   <div>

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

     <Switch> … </Switch>

   </div>
</Router>
```

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

Navigation

- We can also use `NavLink`, which functions identically but lets us specify a CSS class to apply when its route is active

```
<header>
   <h1>My Website</h1>
   <nav>
      <NavLink to="/page1" activeClassName={styles.activeLink}>Page One</NavLink>
      <NavLink to="/page2" activeClassName={styles.activeLink}>Page Two</NavLink>
   </nav>
</header>
```

Redirection

- Using the `Redirect` component, we can specify that when we navigate to a certain Route, we automatically redirect to an alternative URL.

```
<Switch>
  <Route path="/articles">
    <ArticlesPage … />
  </Route>

  <Route path="/gallery">
    <GalleryPage … />
  </Route>

  <Route path="*">
    <Redirect to="/articles" />
  </Route>
</Switch>
```

If we navigate to `/articles`, render the `ArticlesPage`.

Otherwise, if we navigate to `/gallery`, render the `GalleryPage`.

Otherwise, redirect the user to `/articles`.

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:

```
<Link to={`/articles/1`}>First article</Link>
…
<Route path={`/articles/:id`}>
   <ArticleView />
</Route>
```

These names must match.

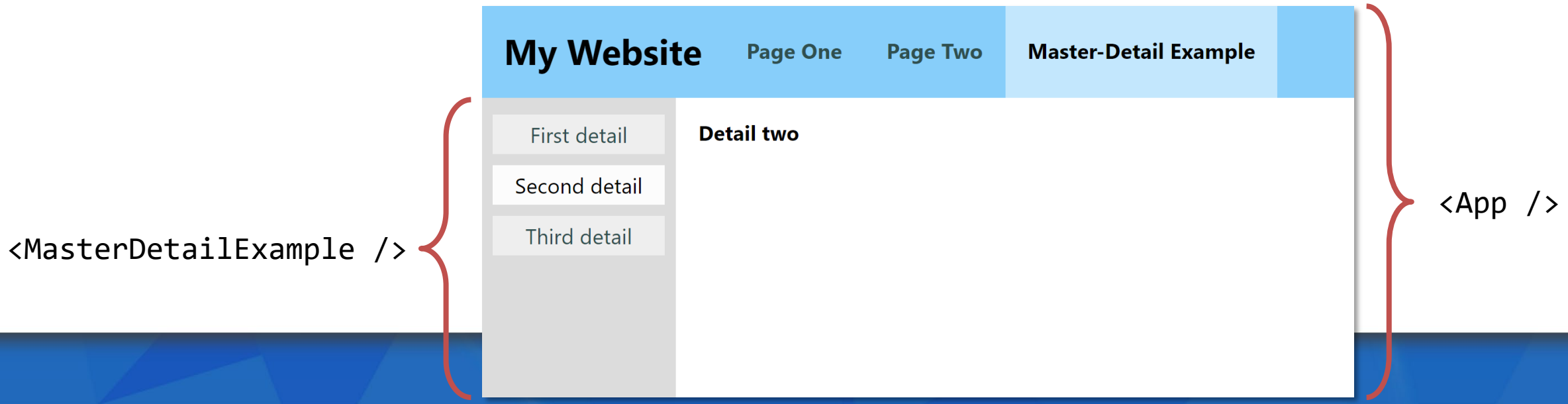
- Then, we can access the supplied value using the `useParams()` hook. For example:

```
function ArticleView() {
    const { id } = useParams();
    return <h3>Article {id}!</h3>;
}
```

If the "First article" link above is clicked, this component will render the text "Article 1".

Nested routes

- We can import and use the `useRouteMatch()` hook to allow us to build `<Route>` and `<Link>` paths which are relative to a parent route.
  - Allows us to build complex hierarchies with nested routes
- For example: Consider an application with a main navbar. Each menu item on the navbar links to a different "page". In one or more of those "pages", a sidebar is presented, allowing the user to select from a number of "sub-pages"…



Nested routes

- When we're rendering our **functional component** with nested routes, we can obtain the link / url information about the parent route as follows:

This code at the top of the file:

```
import { NavLink, useRouteMatch, Switch, Route } from 'react-router-dom';
```

…

This code in our component function:

```
const { path, url } = useRouteMatch();
```

Nested routes

- We can then use the `url` variable to help build our `<Link>`s, and the `path` variable to help build our `<Route>`s.
- Assuming the parent path here is `/master-detail`…

```
<aside>
   <NavLink to={`${url}/detail1`} activeClassName={styles.activeLink}>First detail</NavLink>
   <NavLink to={`${url}/detail2`} activeClassName={styles.activeLink}>Second detail</NavLink>
   <NavLink to={`${url}/detail3`} activeClassName={styles.activeLink}>Third detail</NavLink>
</aside>
<main>
   <Switch>
      <Route exact path={path}>
         <h3>Please select an item on the left</h3>
      </Route>
      <Route path={`${path}/detail1`}>
         <h3>Detail one</h3>
      </Route>
      …
   </Switch>
</main>
```

These will link to `/master-detail/detail1`, etc.

This will match exactly `/master-detail`, with no additional path

This will match `/master-detail/detail1`

Programmatic navigation with useHistory()

- Sometimes, we want to be able to programmatically navigate through our webapp, rather than relying on user interaction or `<Redirect>`s.
- To do this, we can use the `useHistory()` hook:

```
const history = useHistory();
history.push("/path/to/navigate");
history.replace("/path/to/navigate");
history.goBack();
```

Obtain the history object from the hook

As if the user manually navigated to the given path

Replaces the current URL with this one – essentially removes the current page from the history stack.

As if the user pressed the browser's back button

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

ENGINEERING

CS 732 / SE 750 Lecture 02

React portals

React portals

- Occasionally we might want to render a React component *outside* its usual place in the DOM tree
  - Example: Rendering a dialog box over the top of all other page elements
- To achieve this, we can use a React portal
- When rendering, rather than returning raw JSX, we return the result of the function `ReactDOM.createPortal()`:
  - Two arguments. The first is something to render
  - The second is the HTML DOM element in which to render it

React portals

In index.html:

```
<div id="root"></div>
<div id="modal-root"></div>
```

In our react code:

```
const modalRoot = document.querySelector('#modal-root');

…

function Modal(…) {
    return ReactDOM.createPortal(
        <div className="dialog">
            …
        </div>
        , modalRoot
    );
}
```

Obtain a reference to a DOM element other than the default React component root.

Render the given React component (the in this example) in the given DOM element (modalRoot),

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

ENGINEERING

CS 732 / SE 750 Lecture 02

Global state with React's Context mechanism

Global state

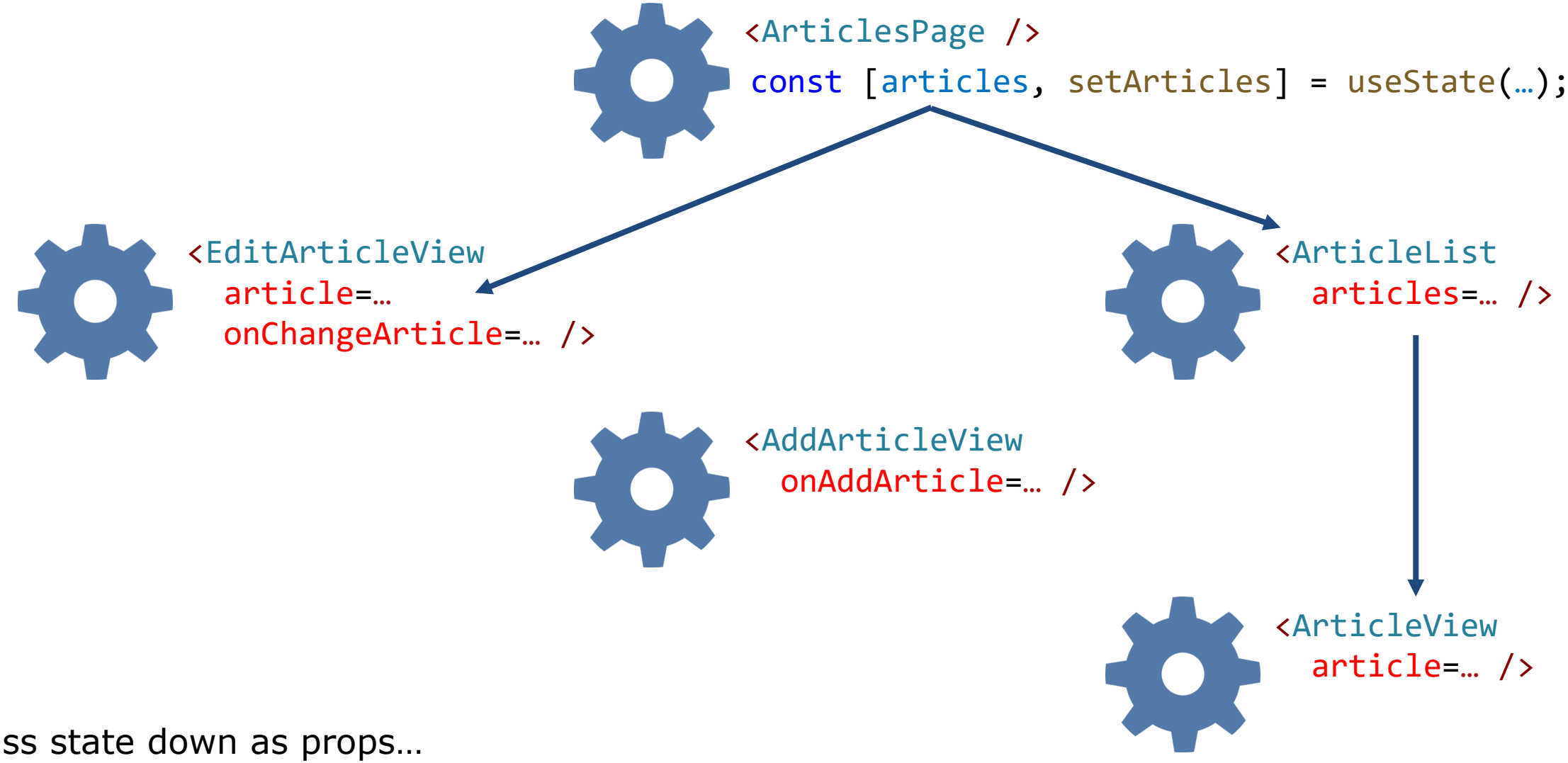
- We have learned how to give components *local* state – using the `useState()` hook
- What if we have state which we need to share with large parts of our application?
  - E.g. a list of articles / to-do items / calendar events
  - Would need to be accessed from (at minimum) the view / add / edit pages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

Models for global state – Top-level storage

- Also known as "moving state up"

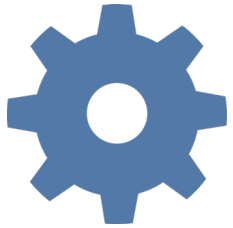
1. Store state at a level in the component hierarchy, such that all components needing to access that state are descendants of the stateful component
2. Pass state "down" to child components as props
3. Pass mutations "up" to parents as events

Models for global state – Top-level storage

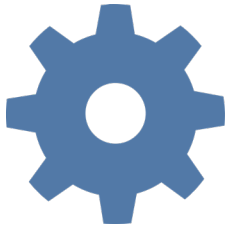


Pass state down as props…

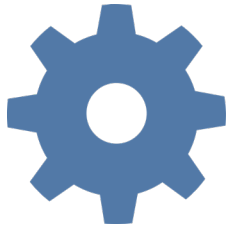
Models for global state – Top-level storage



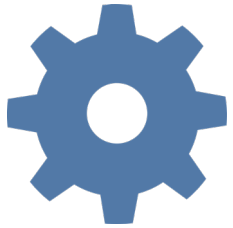
```
<ArticlesPage />
const [articles, setArticles] = useState(…);
```



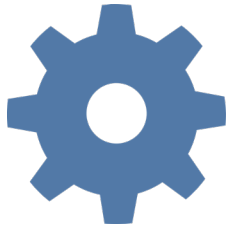
```
<EditArticleView
  article=…
  onChangeArticle=… />
```



```
<ArticleList
  articles=… />
```



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



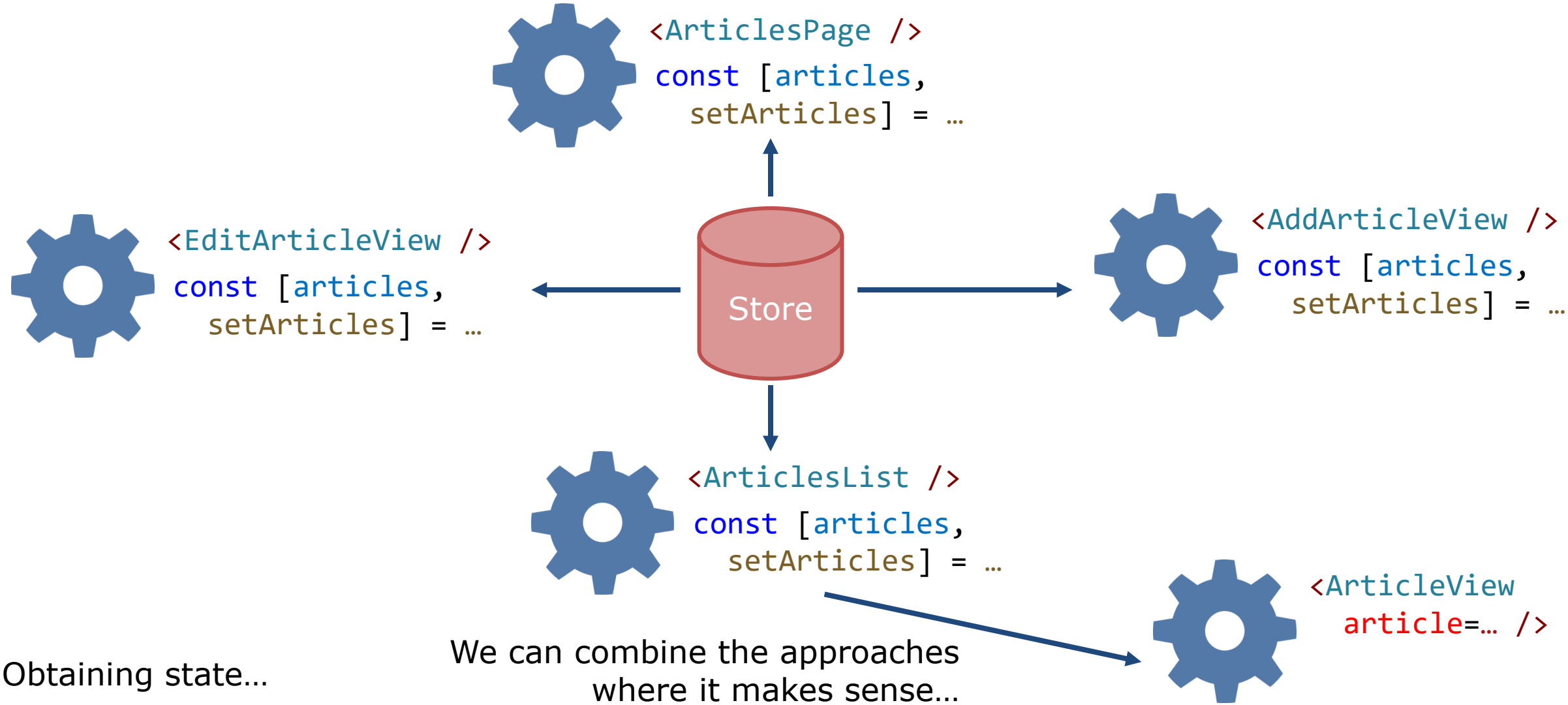
```
<ArticleView
  article=… />
```

… 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

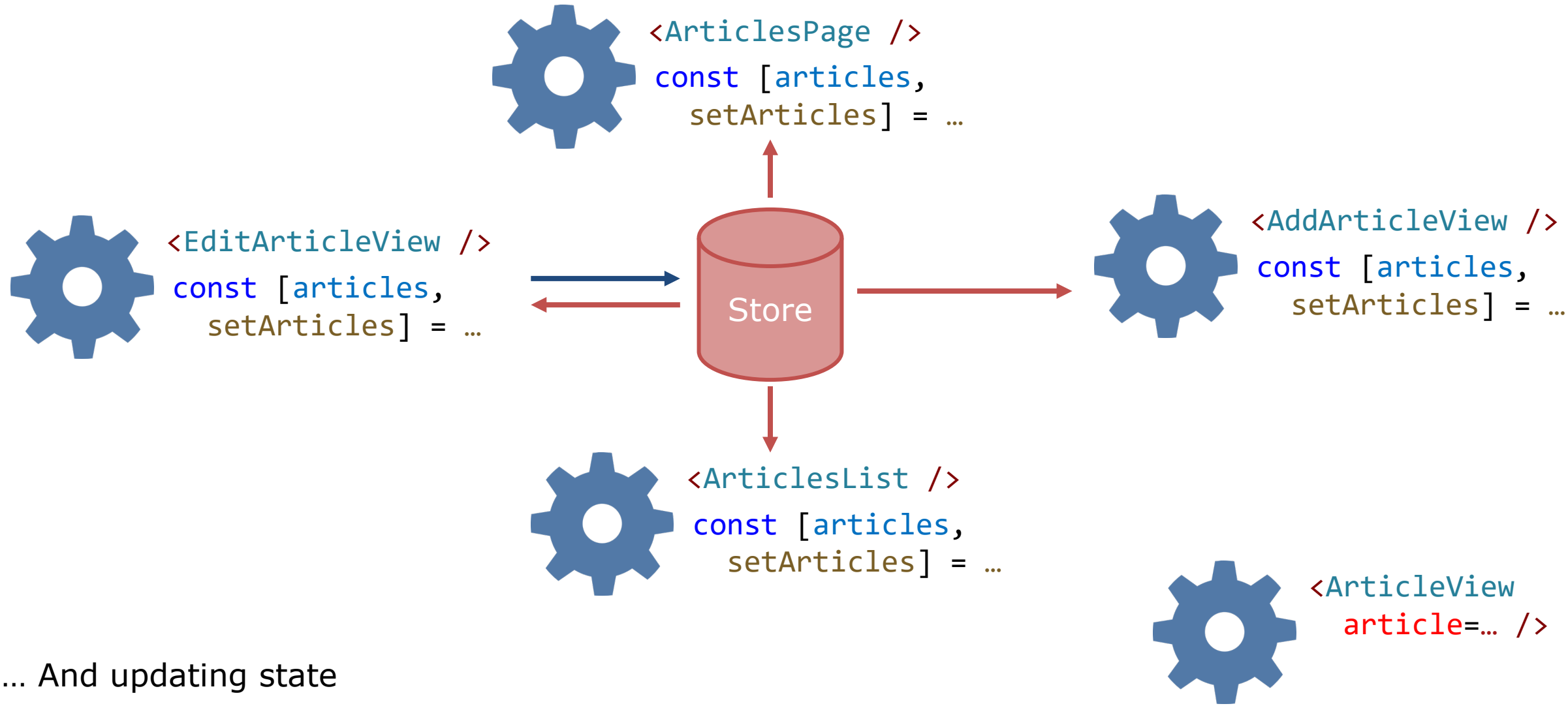
Models for global state – Centralized storage



Obtaining state…

We can combine the approaches where it makes sense…

Models for global state – Centralized storage



… 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


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

```
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 >
  );
}
```

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

```
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 >
  );
}
```

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

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

```
export default function UserInfoPage() {

    const user = useContext(AuthContext);

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

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)
- **Example 18** 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
  - Was taught in CS732/SE750 last year!
  - Slides and examples available for reference
  - Can do much of the same thing with the Context API
- Use local browser storage
  - Provides persistent state across page refreshes / reloads
  - Ideally need to account for different app versions

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

ENGINEERING

CS 732 / SE 750 Lecture 02

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)


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

```
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>
    );
}
```

Local storage use in React

- Issue: Code on the previous slide won't properly propagate updates to local storage to any component other than the one causing the update
- Solutions:
  - We can store the values returned by useLocalStorage() in Context. Or,
  - We can use a third-party package which addresses this issue and more

Package use-persisted-state

- A third-party npm package. Source available here
- Install to our projects as follows:

```
npm i -S use-persisted-state
```

- Benefits over our own `useLocalStorage()` hook:
  - Updates to a local storage value with a given key will propagate to all components using that key
  - This includes components *in other browser tabs / windows!*
  - If the structure of our data changes (e.g. new app version), the package will automatically clear old incompatible data

Package use-persisted-state


```
import createPersistedState from 'use-persisted-state';
const useAuthState = createPersistedState('auth');

export default function UserInfoPage() {
    const [user, setUser] = useAuthState();
    return (
        <h1>{user ? `Welcome, ${user.username}!` : 'You are not logged in!'}</h1>
    );
}
```

We can use the `createPersistedState()` function to create a hook we can use for getting / setting some data in local storage

Wherever we use the same key, all components using that key will be re-rendered together when the value changes

```
import createPersistedState from 'use-persisted-state';
const useAuthState = createPersistedState('auth');

export default function LoginPage() {
    const [user, setUser] = useAuthState();
    return (
        <div>
            <button onClick={() => setUser({ username: 'Bob' })}>Log in</button>
            <button onClick={() => setUser(null)}>Log out</button>
        </div>
    );
}
```

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

ENGINEERING

CS 732 / SE 750 Lecture 02

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)

Material UI

- Material UI is one React component library giving developers access to many React components conforming to Google's Material Design language
- Install as follows:

```
npm install @material-ui/core
npm install @material-ui/icons
```

- 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://material-ui.com/getting-started/installation/

Material UI – Example project

- Check out **example 17** in the examples repository to see some of what Material UI can do!

Online resources

- React router
- React portals
- React context API
- Use-persisted-state package
- Material-UI