Tuesday, July 7, 2020 8:02 PM

Outlook, UDACITY

Data Flow in React Recap

In React, data flows in only one direction, from parent to child. If data is shared between sibling child components, then the data should be stored in the parent component and passed to both of the child components that need it.

The component that stores the data should be the one that updates the data.

Functional Programming

Hot in JS

- Pure functions
- Function composition
- Avoid shared state
- Avoid mutating state
- Avoid side effects

Let's recap on some of the things we covered in this lesson on **why React is great**:

- its compositional model
- its declarative nature
- the way data flows from parent to child
- and that React is really just JavaScript

Lesson Challenge

Read these 3 articles that cover some of the essentials of React: Virtual DOM, The Diffing Algorithm, and How Virtual-DOM and diffing works in React. Answer the following questions (in your own words) and share your answers with your Study Group.

1) What is the “Virtual DOM”?

2) Explain what makes React performant.

3) Explain the Diffing Algorithm to someone who does not have any programming experience.

Rendering UI with React

1. Creating Elements

React's .createElement() method takes in a description of an element and returns a plain JavaScript object. .createElement() Returns One Root Element. JSX Returns One Main Element, Too

React.createElement(/* type */, /* props */, /* content */);
type – either a string of any existing HTML element or a React Component
props – either null or an object of HTML attributes and custom data about the element.

content – null, a string, a React Element, or a React Component

```javascript
import React from 'react'
import ReactDOM from 'react-dom'

const element = React.createElement('div', null, 'hello');
ReactDOM.render(element, document.getElementById('root'));

const element = React.createElement('div',
        {className: 'welcome-message'},
         'hello');
/* element is a js object describing real DOM nodes, not html strings */
const element = React.createElement('div', null,
 React.createElement('strong', null, 'Hello world!')
);

const element = React.createElement('ol', null,
        React.createElement('li', null, 'Tyler'),
        React.createElement('li', null, 'Karen'),
        React.createElement('li', null, 'Richard'),
)
const people = [{name: 'Tyler'}, {{name: 'Karen'}, {name: 'Richard']
const element = React.createElement('ol', null,
        people.map(person => React.createElement('li', {key: person.name}, person.name))
const element = <ol>{people.map(person=><li key={person.name}>{person.name}</li>)}</ol>
/* if you are mapping over an array with React and creating elements for each item in that array, each element needs its own unique key prop */


class ContactList extends React.Component {
        render() {
                const people = []

                return <ol>
                        {people.map(person=><li key={person.name}>{person.name}</li>)}
                </ol>
        }
}
ReactDOM.render(<ContactList />, document.getElementById('root'))

import React, { Component } from 'react';
class ContactList extends Component {
// ...
}
```

```
React allows a lot of HTML attributes to be passed along to the React element.

A great mindset to have when building React apps is to think in components. Components represent the modularity and reusability of React. You can think of your component classes as factories that produce instances of components. These component classes should follow the single responsibility principle and just "do one thing". If it manages too many different tasks, it may be a good idea to decompose your component into smaller subcomponents.

SVG: It turns out that SVG was the one graphic format that most closely responds to current web development demands of scalability, responsiveness, interactivity, programmability, performance, and accessibility.

## Transpiler v Compiler

They're essentially the same: take source code and transform it to something else.

The difference is that compiler usually produces a directly usable artifact (executable binary of some sort). Example: C (produces binary), C# (produces bytecode).

Whereas transpiler produces another form of source code (in another language, for example), which is not directly runnable and needs to be compiled/interpreted. Example: CoffeeScript transpiler, which produces javascript. Opal (converts ruby to javascript)

We typically use a **transpiler** like **Babel** to transpile JSX into regular JavaScript for us. We can run Babel through a **build tool**, like **Webpack** which helps bundle all of our assets (JavaScript files, CSS, images, etc.) for web projects.

JSX v template

## create-react-app

Facebook's create-react-app is a command-line tool that scaffolds a React application. Using this, there is no need to install or configure module bundlers like Webpack, or transpilers like Babel. These come preconfigured (and hidden) with create-react-app, so you can jump right into building your app!

# Favor Composition Over Inheritance

Components are reusable chunks that can be nested inside of each other, like Russian nesting dolls. Each component needs to follow the Single Responsibility Principle - that is, do only 1 thing.

Oftentimes, the number of components an app should have is subjective, but it is always the case that they should follow the Single Responsibility Principle, be reusable, and manageable in size.

## Stateless functional components

For performance reasons, if a component doesn't need to hold state, we'd want to make it a Stateless Functional Component. No internal state, i.e. just
const Email = (props) => (
```

```
 <div>
   {props.text}
 </div>
);
```
a render() method

State

Props refer to attributes from parent components. "read-only" data that are immutable.

A component's state, on the other hand, represents **mutable data** that ultimately affects what is rendered on the page. State is managed internally by the component itself and is meant to **change over time, commonly due to user input** (e.g., clicking on a button on the page).

Avoid initializing that state with props. This is an **error-prone anti-pattern**, since state will only be initialized with props when the component is first created.

```
this.state = {
 user: props.user
}
```
In the above example, if props are ever updated, the current state will not change unless the component is "refreshed." Using props to produce a component's initial state also leads to **duplication of data**, deviating from a dependable "source of truth."

Your UI is a function of your state

State Recap

By having a component manage its own state, any time there are changes made to that state, React will know and automatically make the necessary updates to the page.

This is one of the key benefits of using React to build UI components: when it comes to **re-rendering the page, we just have to think about updating state.** We don't have to keep track of exactly which parts of the page change each time there are updates. We don't need to decide how we will efficiently re-render the page. React compares the previous output and new output, determines what has changed, and makes these decisions for us. This process of determining what has changed in the previous and new outputs is called Reconciliation.

https://reactjs.org/docs/thinking-in-react.html

Figure out the absolute minimal representation of the state your application needs and compute everything else you need on-demand.

1. Is it passed in from a parent via props? If so, it probably isn't state.
2. Does it remain unchanged over time? If so, it probably isn't state.
3. Can you compute it based on any other state or props in your component? If so, it isn't state.

https://en.wikipedia.org/wiki/Don%27t_repeat_yourself

- When the DRY principle is applied successfully, a modification of any single element of a system does not require a change in other

logically unrelated elements.
- Additionally, elements that are logically related all change predictably and uniformly, and are thus kept in sync.
- Violations of DRY are typically referred to as WET solutions, which is commonly taken to stand for "write every time", "write everything twice", "we enjoy typing" or "waste everyone's time".

State management

Props: allow you to pass data into components
Functional components: more intuitive approach to creating components
Controller components: allow you to hook up the forms to your component state

UI+ state: crash/freeze most likely due to state mismanagement

--> improve the predictability of the state, how?
All of the state for application in a single location
- Improve sharing state
- More predictable state changes. Set strict rules for how to update

1. Whenever setState() is called, the component also calls render() with the new state
2. State updates can be merged by passing in an object to setState()
3. State updates can be asynchronous, i.e. setState() can accept a function with the previous state as its first argument

setState() is asynchronous. Do not call setState on one line and assume the state has already changed on the next line.

There are two solutions to this mistake:
- Use the componentDidUpdate() lifecycle method (*recommended by the React team*).
- Pass a callback as a second argument to setState().

From <https://duncanleung.com/avoiding-react-setstate-pitfalls/>

Pitfall 4: Trying to issue multiple setState() calls

Multiple setState() calls during the same cycle may be batched. This is specifically an issue when passing updater an object literal.
- setState() performs a shallow merge of the updater object into the new state.
- Subsequent setState() calls will override values from previous calls in the same cycle.
- If the updater objects have the same keys, the value of the key, of the last object passed to Object.assign(), overrides the previous value.

From <https://duncanleung.com/avoiding-react-setstate-pitfalls/>

Use PropTypes lib to validate that particular element (i.e., an array, a func, a number, etc.), we can also chain isRequired to show a warning if that prop is not provided. PropTypes.string.isRequired

Controller Components
Renders a form but the source of truth for the form state lives inside of the component state, rather than inside of the DOM. React controls the state of the form. So controller components allow you to update your UI based on the form itself, which does not apply to "uncontrolled" components

Benefits:

1. Support instant input validation
2. Conditionally disable or enable buttons
3. Enforce input formats

React Developer Tools

To recap how user input affects the ListContacts component's own state:

1. The user enters text into the input field.
2. The onChange event listener invokes the updateQuery() function.
3. updateQuery() then calls setState(), merging in the new state to update the component's internal state.
4. Because its state has changed, the ListContacts component re-renders.

Any changes to React state will cause a re-render on the page, effectively displaying our live search results.

1. Each update to state has an associated handler function
2. Form elements receive their current value via an attribute
3. Form input values are generally stored in the component's state
4. Event handlers for a controlled element update the component's state

Lifecycle Events

data should not be fetched in the render method! No aync function in render(). So async functions can trigger re-rendering by changing the state or props, but not the other way around.

Ajax requests should only be made in the componentDidMount lifecycle method.

A typical react constructor is used for two purpose -
① Initializing local state by assigning an object to this.state .
② Binding event handler methods to an instance.
Fetching data in constructor is considered to be a side effect and it is recommended to avoid that. As you all know we cannot call setState() in the constructor.

- `componentDidMount()`
  invoked immediately *after* the component is *inserted* into the DOM
- `componentWillUnmount()`
  invoked immediately *before* a component is *removed* from the DOM
- `getDerivedStateFromProps()`
  invoked after a component is instantiated as well as when it receives brand new props

  `shouldComponentUpdate()`
  It returns `true` by default. This means that whenever a component's state (or its parent's state) is updated, the component re-renders.
  This method only exists as a **performance optimization.**

- The default behavior is to re-render on every state change, and in the vast majority of

cases you should rely on the default behavior.
- Do not rely on it to “prevent” a rendering, as this can lead to bugs.
- Consider using the built-in `PureComponent` instead of writing `shouldComponentUpdate()` by hand.
- We do not recommend doing deep equality checks or using `JSON.stringify()` in `shouldComponentUpdate()`. It is very inefficient and will harm performance.

React.PureComponent is similar to React.Component. The difference between them is that React.Component doesn’t implement shouldComponentUpdate(), but React.PureComponent implements it with a shallow prop and state comparison.
If your React component’s render() function renders the same result given the same props and state, you can use React.PureComponent for a performance boost in some cases.

Project 1: https://github.com/udacity/reactnd-project-myreads-starter

React Router

URL controls the page content so it needs to reflect the state of the app. We need to create React applications that offer bookmarkable pages! Keep UI and URL in sync

React Router turns React projects into single-page applications. It does this by providing a number of specialized components that manage the creation of links, manage the app's URL, provide transitions when navigating between different URL locations, and so much more.

React Router is a collection of **navigational components** that compose declaratively with your application.

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

```
class BrowserRouter extends React.Component {
 static propTypes = {
   basename: PropTypes.string,
   forceRefresh: PropTypes.bool,
   getUserConfirmation: PropTypes.func,
   keyLength: PropTypes.number,
   children: PropTypes.node
 }

history = createHistory(this.props)

render() {
   return <Router history={this.history} children={this.props.children} />
 }
```

```
}
```

In summary, for React Router to work properly, you need to wrap your whole app in a `BrowserRouter` component. Also, `BrowserRouter` wraps the history library which makes it possible for your app to be made aware of changes in the URL.

You'll use <Link> in place of anchor tags (`a`) as you're typically used to

```
<Link to="/about">About</Link>
```

```
<Link to={{
 pathname: '/courses',
 search: '?sort=name',
 hash: '#the-hash',
 state: { fromDashboard: true }
}}>
 Courses
</Link>
```

Components wrapped in the Router component will only render when it matches (at least some initial part of) the URL. If the 'exact' flag is set, the path will only match when it exactly matches the URL.

```
npm install --save form-serialize
```

Redux

Github: https://github.com/udacity/reactnd-redux-todos-goals
predictable state management, through having a single state tree that stores the entire state for an app. Reduce duplication, easier to keep state/data in sync

The store: state tree object + 3 ways to interact with it (get the state, update the state and listen for changes)

Rule #1: Only an event can change the state of the store.

1. getState - returns the state
2. subscribe - listens for change

Actions: records of state change. Represent the events that will change the state of our store. All actions are plain JS object and must have a type property. If the state of the app changes, we will know that one of those actions occurred.

it's better practice to pass as little data as possible in each action. That is, prefer passing the index or ID

of a product rather than the entire product object itself.

Example:

```
{
  type: "ADD_PRODUCT_TO_CART",
  productId: 17
}
```

Action Creators are functions that create/return action objects. For example:

```
const addItem = item => ({
  type: ADD_ITEM,
  item
});
```

or in ES5:

```
var addItem = function addItem(item) {
  return {
    type: ADD_ITEM,
    item: item
  };
};
```

Both receivePost functions are *action creators*, which extrapolate the creation of an action object to a function. clearErrors and addSeven, on the other hand, *are* action objects with a valid type key and optional payload. removeComments does not include a type key and is an invalid action.

Rule #2: The function that returns the new state needs to be a pure function

Pure functions are predictable.

1. Return the same result if the same arguments are passed in
2. Depend solely on the arguments passed into them
3. Do not produce side effects, such as API requests and I/O operations (console.log is I/O too)

If we have a function that takes in our state and an action that occurred, the function should (if it's pure!) return the exact same result *every single time*.

Reducer: function that takes in the current state and action and then returns the new state of the app. Must be pure function

```
/* Create A Reducer
 *
 * You need to create a reducer called "appReducer" that accepts two arguments:
 * - First, an array containing information about ice cream
 * - Second, an object with a 'DELETE_FLAVOR' `type` key
 * (i.e., the object contains information to delete the flavor from the state)
 *
 * The action your reducer will receive will look like this:
 * { type: 'DELETE_FLAVOR', flavor: 'Vanilla' }
 *
 * And the initial state will look something like this (as such, refrain
 * from passing in default values for any parameters!):
 * [{ flavor: 'Chocolate', count: 36 }, { flavor: 'Vanilla', count: 210 }];
*/
function appReducer(state, action){
```

```
  if (action.type==="DELETE_FLAVOR"){
    return state.filter((item)=>item.flavor!==action.flavor)
  }
  else {
    return state
  }
}
```

- dispatch() is called with an Action
- the reducer that was passed to createStore() is called with the current state tree and the action...this updates the state tree
- because the state has (potentially) changed, all listener functions that have been registered with the subscribe() method are called

Whenever we want to update the state of the store, we just need to do call `dispatch`, passing it the action which occurred.

- we created a function called createStore() that returns a *store* object
- createStore() must be passed a "reducer" function when invoked
- the store object has three methods on it:
- .getState() - used to get the current state from the store
- .subscribe() - used to provide a listener function the store will call when the state changes
- .dispatch() - used to make changes to the store's state
- the store object's methods have access to the state of the store via closure

- Updates to the store can only be triggered by dispatching actions.
- The store's subscribe() function helps connect React components to the store.

- Reducers must be pure
- Though each reducer handles a different slice of state, we must combine reducers into a single reducer to pass to the store
- createStore() takes only one reducer argument
- Reducers are typically named after the slices of state they manage

Best Practices:

1. Why prefer constants over strings in action types? We can ensure an error will be thrown for misspelled action types
2. We also refactored our .dispatch() calls from passing in unique objects directly to them, to calling special functions that create the action objects - these special functions that create action objects are called **Action Creators**.

Redux has a state management library that has a store, actions and reducers.

DOM manipulation:

- *accessing elements with document.getElementById()*
- *adding listeners with .addEventListener()*
- *accessing the .value property on an element*
- *creating a new element with .createElement()*

- *adding new content with* `appendChild()`
- *etc.*

Redux Middleware

Allows us the hook into the Redux lifecycle and intercept actions before they reach any reducers.

…a third-party extension point between dispatching an action, and the moment it reaches the reducer.

- producing a side effect (e.g., logging information about the store)
- processing the action itself (e.g., making an asynchronous HTTP request)
- redirecting the action (e.g., to another piece of middleware)
- dispatching supplementary actions

```
const store = Redux.createStore( <reducer-function>, <middleware-functions> )
```

Because we set up the Redux store with knowledge of the middleware function, it runs the middleware function between `store.dispatch()` and the invocation of the reducer.

```
applyMiddleware(...middlewares)
```

Note the spread operator on the `middlewares` parameter. This means that we can pass in as many different middleware as we want! Middleware is called in the order in which they were provided to `applyMiddleware()`.

```
const store = Redux.createStore( <reducer-function>, Redux.applyMiddleware(<middleware-functions>) )
```

Redux middleware leverages a concept called **higher-order functions**. A higher-order function is a function that either:

- *accepts* a function as an argument
- *returns* a function

Redux with React

Decouple state from UI

Uncontrolled components

React will call the ref callback with the DOM element when the component mounts, and call it with null when it unmounts. Refs are guaranteed to be up-to-date before componentDidMount or componentDidUpdate fires.

To write an uncontrolled component, instead of writing an event handler for every state update, you can use a ref to get form values from the DOM. You have to 'pull' the value from the field when you need it.

```
class Form extends Component {
  handleSubmitClick = () => {
    const name = this._name.value;
    // do something with `name`
  }
render() {
    return (
      <div>
        <input type="text" ref={input => this._name = input} />
        <button onClick={this.handleSubmitClick}>Sign up</button>
      </div>
    );
  }
}
```

Controlled components
What happens: the value of the input is stored as a state, onChange setState every time the value changes so that it'll be rendered.

1. It starts out as an empty string — ''.
2. You type a and handleNameChange gets an a and calls setState. The input is then re-rendered to have the value of a.
3. You type b. handleNameChange gets the value of ab and sets that to the state. The input is re-rendered once more, now with value="ab".

```
class Form extends Component {
  constructor() {
    super();
    this.state = {
      name: '',
    };
  }
handleNameChange = (event) => {
    this.setState({ name: event.target.value });
  };
render() {
    return (
      <div>
        <input
          type="text"
          value={this.state.name}
          onChange={this.handleNameChange}
        />
      </div>
    );
  }
}
```

The form component can respond to input changes immediately; for example, by:

- in-place feedback, like validations
- disabling the button unless all fields have valid data
- enforcing a specific input format, like credit card numbers

A form element becomes “controlled” if you set its value via a prop.

Element	Value property	Change callback	New value in the callback

<input type="text" />	value="string"	onChange	event.target.value
<input type="checkbox" />	checked={boolean}	onChange	event.target.checked
<input type="radio" />	checked={boolean}	onChange	event.target.checked
<textarea />	value="string"	onChange	event.target.value
<select />	value="option value"	onChange	event.target.value

From <https://goshakkk.name/controlled-vs-uncontrolled-inputs-react/>

feature	uncontrolled	controlled
one-time value retrieval (e.g. on submit)	✓	✓
validating on submit	✓	✓
instant field validation	✗	✓
conditionally disabling submit button	✗	✓
enforcing input format	✗	✓
several inputs for one piece of data	✗	✓
dynamic inputs	✗	✓

From <https://goshakkk.name/controlled-vs-uncontrolled-inputs-react/>

Asynchronous Redux

- `store.dispatch()` calls are made
- if the Redux store was set up with any middleware, those functions are run
- then the reducer is invoked

In the task above, you could've just fetched all of our todos and then all of our Goals, but that's serial and is just making the user wait an unnecessarily long amount of time. Since the API is Promise-based, we can use `Promise.all()` to wait until all Promises have resolved before displaying the content to the user.

the different states our app can be in while getting our remote data:

- before the app has the data
- while the app is fetching the data
- after the data has been received

optimistic updates. Instead of waiting for confirmation from the server, just instantly remove the user from the UI when the user clicks “delete”, then, if the server responds back with an error that the user wasn’t actually deleted, you can add the information back in. This way your user gets that instant feedback from the UI, but, under the hood, the request is still asynchronous.

we're assuming the change will succeed correctly on the server, so we update the UI immediately, and then only roll back to the original state if the API returns an error. Doing optimistic updates is better because it provides a more realistic and dynamic experience to the user.

React-redux

Context API
Context provides a way to pass data through the component tree without having to pass props down manually at every level.

Separation of concerns:

- A connected component (container component) is connected to the Redux store and is responsible for getting data from the store.
- A presentational component should not access the store. It should receive any information it needs as props and then just render a UI.

Connects store and component. Using `connect()`, we can conveniently access the `store` context set by `Provider`. We pass in parts of the state as well as action-dispatches to the components as `props`.

`react-redux` bindings. These bindings give us an API that simplifies the most common interactions between React and Redux.

```
ReactDOM.render(
 <ReactRedux.Provider store={store}>
 <ConnectedApp />
 </ReactRedux.Provider>,
 document.getElementById('app')
);
```

A container component connects the store to MyComponent, giving MyComponent slices of state accessible via props.

```
const ConnectedComponent = connect(mapStateToProps, mapDispatchToProps)(MyComponent)
```

`connect()` connects a React component to the Redux store. The `mapStateToProps()` function allows us to specify which state from the store you want passed to your React component, while the `mapDispatchToProps()` function allows us to bind dispatch to action creators before they ever hit the component.

Real word Redux

Github: https://github.com/udacity/reactnd-chirper-app

A Guide for the Planning Stages of Your Project

1. Identify What Each View Should Look Like
2. Break Each View Into a Hierarchy of Components
3. Determine What Events Happen in the App
4. Determine What Data Lives in the Store

a component should ideally only do one thing. If it ends up growing, it should be decomposed into smaller subcomponents.

1. Components let you split the UI into independent, reusable chunks
2. Each view typically has a component that represents that view
3. Presentational components don't know where their data comes from
4. Components that are connected to the store are called "containers"

When we talk about *state*, we're really talking about *data* - not just any kind of data inside the app, but data that can change based on the events in the app.

the main problems that Redux (and the react-redux bindings!) was meant to solve were:

- Propagation of props through the entire component tree.
- Ensuring consistency and predictability of the state across the app.

How to choose between Redux's store and React's state?
- Whether it's used by multiple components or mutated in a complex way

principles of state normalization, according to Redux documentation:

- Each *type* of data gets its own "table" in the state.
- Each "data table" should store the individual items in an object, with the IDs of the items as keys and the items themselves as the values.
- Any references to individual items should be done by storing the item's ID.
- Arrays of IDs should be used to indicate ordering.

Make API requests in the `componentDidMount()` lifecycle method in React apps, and from asynchronous action creators in React/Redux apps.

React Native

Single UI team with React Native = web team + IOS team + Android team
Write once, run anywhere

React: Learn once, write anywhere. Share principles amongst platforms, such as component composition and declarative UI.

Many of the same principles of the Virtual DOM, **reconciliation**(The end goal of reconciliation is to update the UI based on this new state in the most efficient way possible), and diffing algorithm apply to both.

Create React Native App Pros

1. minimizes the amount of time it takes to create a "hello world" application.
2. allows you to easily develop on your own device. This way, any changes you make in your text editor will instantly show on the app running on your local phone.
3. you just need one build tool. You don't have to worry about Xcode or Android Studio.
4. Lastly, there's no lock in. Just like Create React App, you can "eject" at anytime.

Create React Native Cons

1. First, if you're building an app that's going to be added to an existing native iOS or Android application, Create React Native App won't work.
2. Second, if you need to build your own bridge between React Native and some native API that Create React Native App doesn't expose (which is pretty rare), Create React Native App won't work.

Android Studio
A special variant of the Code Completion feature invoked by pressing Ctrl+Space twice allows you to complete XML tag names from namespaces not declared in the current file. If the namespace is not declared yet the declaration is generated automatically.

You can inject SQL into a string literal (Alt+Enter | Inject language or reference | <SQL dialect>) and then use coding assistance for SQL.

Using the Debugger
1. Refreshing
2. Toggle element inspector
3. Start/stop remote debugging

Web vs. Native

For one, native apps often leverage **animations** to help create a great user experience. Animations such as button effects, screen transitions, and other visual feedback may be subtle, but they support continuity and guidance in the apps you build. They all function to dynamically *tell a story* about how your application works. Without animations, an application can feel like just a collection of static screens. For now, stay tuned; we'll be checking out animations in-depth during Lesson 5.

Another key difference between native and web applications is in **navigation**. Recall that React Router's Route component allows us to map a URL to a specific UI component. In React Native, routers function as a *stack*; that is, individual screens are "pushed" and "popped" as needed. We'll look at routing more closely later in Lesson 4.

Android vs. iOS

Not only are there fundamental differences between *native* apps and *web* apps, you'll also find differences between how native platforms (iOS and Android) *look and feel* as well. Perhaps the most apparent are the distinct design philosophies on each platform: Android apps utilize Google's Material Design, while iOS apps take advantage of Apple's Human Interface Design. When designing mobile applications, it's important to your users that an iOS app *feels* like an iOS app, and an Android app *feels* like an Android app.

Navigation *between* screens feels distinct between Android and iOS as well. Android devices have access to a **navigation bar** at the bottom of the screen, which allows users to go back to the previous screen (among other features). On iOS, the approach is different: there is no such universal navigation bar! When building the UI for an iOS application, it is important to include a back button (perhaps on a custom navigation bar) to help guide users through your app.

One more key difference between Android and iOS involves tab navigation. iOS apps include tab bars at the bottom of the app's screen, allowing for convenient access to different portions of the app. Likewise, Android apps include them as well; however tabs are distinctly located at the top of the screen. Both allow access to high-level content, and we'll explore React Native's **TabNavigator** in closer detail in Lesson 4.