React

Tips

- ReactDom.Render function renders a react element inside an html element
- You can embed any JavaScript expression (a valid unit of code that resolves to a value) in JSX by wrapping it in curly braces.
- When the state (or the props) changes the component responds by re-rendering.
- When a parent component is rendered, its children components are rendered too.
- React components don't modify their props. The React philosophy is that props should be immutable and top-down.
- Only class based components can have a state
- If you want to have an action occur on a child which modifies something on the parent's state, then what you do is pass a callback to the child which it can execute upon the given action. This callback can then modify the parent's state, which in turn sends different props to the child on re-render (and the child id re-rendered if props changed).
- The state is not persistent, it is flushed if the page reloads. For persistent storage you can use: Cookies (Persistent cookies or session cookies) Web storage
- Context provides a way to pass data through the component tree without having to pass props down manually at every level. Whenever the value of the context changes, the subscribed components are rendered. Context is designed to share data that can be considered "global" for a tree of React components, such as the current authenticated user, theme, or preferred language. Apply it sparingly because it makes component reuse more difficult

A GOOD WAY FoR COMPARING the use of libraries is to get the npm downloads history from npm-stat.com for example. In 2018 react is far ahead from Vue and angular (despite the fact that vue has more github stars - that mainly measure hype).

React for everything

There is also react native for mobile, react for VR, react for TVOS soon etc. I have built apps with lots of code reuse on native and web. Abstract all data logic and you can use the exact same code for both apps. This of course does not include your view layer. If you do it correctly, all you have to do is make a new set of view components. Stateless ones. Because you are injecting the state from your non platform specific components.

It makes it extremely fast to build for other platforms. It just takes a few apps to fully understand how to abstract these things perfectly.

The model data in React is represented by props and state. A change to each of those, triggers a re-rendering.

In React, when a component's state changes, it triggers the re-render of the entire component sub-tree, starting at that component as root. To avoid unnecessary re-renders of child components, you need to either use PureComponent or implement shouldComponentUpdate whenever you can. If the shouldComponentUpdate method returns false the component will not be updated.

The virtual dom is composed of react elements.

Babel compiles JSX down to React.createElement() calls which creates an object called React Element

composition

- Many components of our app can have the following behaviour: On mount, add a change listener to DataSource. Inside the listener, call setState whenever the data source changes. On unmount, remove the change listener. We implement this with HOC. Error boundaries also with HOC. But you can use custom hooks instead.
- Components with some common elements can be created with a common component and props.children
- Template blocks like functionality can be implemented with props.sidebar where sidebar renders a component (sidebar is an example could be anything like props.left)

In React, all DOM properties and attributes (including event handlers) should be camelCased. For example, the HTML attribute tabindex corresponds to the attribute tabIndex in React. The exception is aria-* and data-* attributes, which should be lowercased.

Unless you spread components out over multiple files (for example with CSS Modules), scoping CSS in React is often done via CSS-in-JS solutions (e.g. styled-components, glamorous, and emotion)

The virtual DOM (VDOM) is a programming concept where an ideal, or "virtual", representation of a UI is kept in memory and synced with the "real" DOM by a library such as ReactDOM. This process is called reconciliation.

When something changes in your app (for example the state of a component) the whole virtual DOM is created from scratch, not just the changed component. Then React diffs it with the previous one, and apply only the differences to the browser DOM. The virtual DOM is very fast.

You can see the actual virtual DOM in the browser's developer tools using a plugin. When you see something wrong in the UI, you can use React Developer Tools to inspect the props and move up the tree until you find the component responsible for updating the state. This lets you trace the bugs to their source

If you reorder some sibling nodes of a component, react will not recreate them since they exist, it will just reorder the existing instances. This makes it very fast.

React, also uses internal objects called "fibers" to hold additional information about the component tree. Fiber is the new reconciliation engine in React 16. Its main goal is to enable incremental rendering of the virtual DOM.

With Perf.start and Perf.end react prints in the console various information about the rendered components.

render() returns a reference to the component or null for stateless components. However, using this return value is legacy and should be avoided because future versions of React may render components asynchronously in some cases. If you need a reference to the root ReactComponent instance, the preferred solution is to attach a callback ref to the root element.

ReactDOMServer

This object can be used to render components to static markup. For example the ReactDOMServer.renderToString(element) will render a React element to its initial HTML. React will return an HTML string. You can use this method to generate HTML on the server (if you use node.js) and send the markup down on the initial request for faster page loads and to allow search engines to crawl your pages for SEO purposes. For server rendered components you can use ReactDOM.hydrate() instead of

ReactDOM.render(). This will preserve the existing markup and only attach event handlers, allowing you to have a very fast first-load experience. render may change your node if there is a difference between the initial DOM and the current DOM. hydrate will only attach event handlers

With React, typically you only need to bind the methods you pass to other components.

This is because In JavaScript, these two code snippets are not equivalent (probably it means that the "this" would be different): `obj.method();` `var method = obj.method;method();`	Binding methods helps ensure that the second snippet works the same way as the first one.

As your app grows, you can catch a lot of bugs with typechecking. For some applications, you can use JavaScript extensions like Flow or TypeScript to typecheck your whole application. But even if you don't use those, React has some built-in typechecking abilities. To run typechecking on the props for a component, you can assign the special propTypes property. We recommend using Flow or TypeScript instead of PropTypes for larger code bases.

Keep in your source control only the source js files (ES6 etc). Generally, you don't want to keep the generated javascript in your source control, so be sure to add the build folder to your .gitignore.

There are javascript error reporting services. When you catch an error in your js code you can pass it to this service. It can also listen for uncaught error. Then you have an error report.

Create-react-app library. Using it You don't need to install or configure tools like Webpack or Babel. They are preconfigured and hidden so that you can focus on the code.

React is unaware of changes made to the DOM outside of React. It determines updates based on its own internal representation, and if the same DOM nodes are manipulated by another library, React gets confused and has no way to recover. The easiest way to avoid conflicts is to prevent the React component from updating. You can do this by rendering elements that React has no reason to update, like an empty . The element has no properties or children, so React has no reason to update it, leaving the jQuery plugin free to manage that part of the DOM

Code splitting: To avoid winding up with a large bundle, it's good to get ahead of the problem and start "splitting" your bundle. Code-Splitting is a feature supported by bundlers like Webpack and Browserify. Code-splitting your app can help you "lazy-load" just the things that are currently needed by the user, which can dramatically improve the performance of your app.

const { extraProp, ...passThroughProps } = this.props; ???

How to think when building a react app:

- You have the data model (for example a json string from an API) you want to represent with an UI.
- You have a mockup of the UI
- You separate the UI to components and create their hierarchy

- You create a static version of the UI, without any interactivity (so no state). This way you don't have to think a lot on this stage. You create the components bottom up (for big projects) or top-bottom (for small ones). The parent component receives the data and passes it properly to its children through props. If you get an alternative data model and execute React.render to the parent, the UI will be updated.
- Identify the minimal state of your UI. For example, if you're building a TODO list, just keep an array of the TODO items around; don't keep a separate state variable for the count.
  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.
- Decide which components would own the state (and pass it as props to its descendants).
  1. Identify every component that renders something based on that state.
  2. Find a common owner component (a single component above all the components that need the state in the hierarchy).
  3. Either the common owner or another component higher up in the hierarchy should own the state.
  4. If you can't find a component where it makes sense to own the state, create a new component simply for holding the state and add it somewhere in the hierarchy above the common owner component.

JSX

const element = <h1>Hello, world!</h1>;

This funny tag syntax is neither a string nor HTML. It is called JSX, and it is a syntax extension to JavaScript. We recommend using it with React to describe what the UI should look like. JSX may remind you of a template language, but it comes with the full power of JavaScript.

```
const user = {
  firstName: 'Harper',
  lastName: 'Perez'
};
```

Or

```
const element = (
  <h1>    Hello, {formatName(user)}!  </h1>);
```

You can embed any JavaScript expression (a valid unit of code that resolves to a value) in JSX by wrapping it in **curly braces**.

Notice that while it isn't required, when doing this, we also recommend wrapping it in parentheses to avoid the pitfalls of automatic semicolon insertion.

You may use quotes to specify string literals as attributes: const element = ;

You may also use curly braces to embed a JavaScript expression in an attribute: const element = <img src={user.avatarUrl}></img>;

If a tag is empty, you may close it immediately with />, like XML

```
const title = response.potentiallyMaliciousInput;
// This is safe:
const element = <h1>{title}</h1>;
```

By default, React DOM escapes any values embedded in JSX before rendering them. Thus it ensures that you can never inject anything that's not explicitly written in your application. Everything is converted to a string before being rendered. This helps prevent XSS (cross-site-scripting) attacks.

After compilation, JSX expressions become regular JavaScript function calls and evaluate to JavaScript objects. This means that you can use JSX inside of if statements and for loops, assign it to variables, accept it as arguments, and return it from functions

JSX maps to function calls

`return <node> something </node>` This jsx is actually this: `return node(something)`

The <node> tag is a function and the tag's contents are the function's arguments. This means that the tag's contents must be of a type that can be passed as argument to a function.

This will give an error since it is not valid JSX

```
render(){
 return(
  <tbody>
     for (var i=0; i < numrows; i++) {
          <ObjectRow/>
     }
  </tbody>
)}
```

It is as if you do something like this: The <tbody> node is mapped to a function call.

```
return tbody(
     for (var i = 0; i < numrows; i++) {
          ObjectRow()
     } )
```

Which is not valid javascript.

You can solve this by creating an array from the for loop and passing it as an argument to the function. Since the returned jsx object is a function, you can pass whatever arguments you want to it. It would result in something like this:

```
var rows = [];
for (var i = 0; i < numrows; i++) {
     // note: we add a key prop here to allow react to uniquely identify each
     // element in this array. see: https://reactjs.org/docs/lists-and-keys.html
   rows.push(<ObjectRow key={i} />);
   }
return <tbody>{rows}</tbody>;
```

Which results to this:

```
var rows = [];
for (var i = 0; i < numrows; i++) {
    rows.push(ObjectRow());}
return tbody(rows);
```

In general the map function is preferred:

```
return <tbody>
  {items.map(item => <ObjectRow key={item.id} name={item.name} />)}
</tbody>
```

React elements

Unlike browser DOM elements, React elements are plain objects, and are cheap to create.

```
// jsx
const element = (
 <h1 className="greeting">
  Hello, world!
 </h1>);

//transpiled to this
const element = React.createElement(
 'h1',
 {className: 'greeting'},
 'Hello, world!');
```

Babel compiles JSX down to React.createElement() calls which creates an object called React Element. React elements construct the React DOM that takes care of updating the browser DOM to match the React elements.

React.createElement() performs a few checks to help you write bug-free code but essentially it creates an object. These objects are called “React elements”. They are the smallest building blocks of React apps. Components are made of react elements.

`` // which creates this object (which is a react element). // Note: this structure is simplified const element = { type: 'h1', props: { className: 'greeting', children: 'Hello, world' }}; ```	React elements are immutable. Once you create an element, you can't change its children or attributes. An element is like a single frame in a movie: it represents the UI at a certain point in time.
`` const element = Hello, world ; ReactDOM.render(element, document.getElementById('root')); ``` Or ``` function App() { return (<Welcome name="Sara" />);} ReactDOM.render(<App />, // this is how you call it if it is not a variable like before document.getElementById('root')); ```	**Root Nodes** Everything inside it will be managed by React DOM. You may have as many isolated root DOM nodes as you like. To render a React element into a root DOM node, pass both to ReactDOM.render() reactDOM renders a react element inside an html node (the node with id=root in this case). The App is a function that returns a jsx syntax which is transpiled to a react element.

When the render function is called again, a new react element will be created and if the new element that is produced has some differences with the existing element, React will update the browser DOM with the differences. Only the differences. So the concept is that you don't think how to change the UI, but rather how it should look at any given moment.

You could set a time interval and call the render function that will update the DOM. This is not how we normally update the DOM with react.

Components

A react component is a javascript function that accepts an object argument called props and returns a React element which describes what should appear in the UI. They accept arbitrary inputs (called "props") and return a React element describing what should appear on the screen. They are function based or class based.

`` const element = <Welcome name="Sara" />; ``` Welcome is a react component.	When React sees an element representing a user-defined component, **it passes** JSX **attributes** (here name='Sara') **to this component as a single object called "props"**, here props={name: 'Sara'} <Welcome name="Sara"><User /></Welcome> in this example the User component can be accessed within the Welcome component with props.children.

Components can refer to other components in their output.

Class and Function based components

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

ReactDOM.render(
  <Clock />,
  document.getElementById('root'));
```

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
// Wrong! The key should have been specified here:
<ListItem value={number} /> );
  return (
    <ul>
      {listItems}
    </ul>
  );}
```

Typically, new React apps have a single App component at the very top. However, if you integrate React into an existing app, you might start bottom-up with a small component like Button and gradually work your way to the top of the view hierarchy.

A good rule of thumb is that if a part of your UI is used several times (Button, Panel, Avatar), or is complex enough on its own (App, FeedStory, Comment), it is a good candidate to be a reusable component. Don't be afraid to split components to smaller ones. One technique is the single responsibility principle, that is, a component should ideally only do one thing.

All React components must act like pure functions with respect to their props. Pure functions do not modify their inputs and so props must not be modified by the component's code. Therefore components have their own state apart from props. **If they want to modify their output they modify their state, not their input**. Components State allows React components to change their output over time in response to user actions, network responses, and anything else, without violating this rule.

Conditional Rendering

You can normally use if blocks in JSX of course, but there are also some tips to type less code for them. For example:

```
<div>
    <h1>Hello!</h1>
    {unreadMessages.length > 0 &&
      <h2>
        You have {unreadMessages.length} unread messages.
      </h2>
    }
  </div>
```

Inline if with logical && operator
It works because in JavaScript, true && expression always evaluates to expression, and false && expression always evaluates to false

If the render function returns null then the component is not rendered. You can use this with conditional statements if you want. Notice that returning null in render, does not affect the firing of the component's lifecycle methods. For instance, componentWillUpdate and componentDidUpdate will still be called.

Lists and Keys

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <li key={number.toString()}>{number}</li>);

ReactDOM.render(
<ul>{listItems}</ul>,
document.getElementById('root'));
```

Common way of rendering multiple components: using the map function.

All item elements of a list in React must have a unique key attribute, unique among its siblings. Keys help React identify which items have changed, are added, or are removed. The best way to pick a key is to use a string that uniquely identifies a list item among its siblings. Most of the time you will use an id that come with your data. You might use an index of the loop but is not recommended since it might cause issues to the components state if there is a reordering.

```
function ListItem(props) {
  const value = props.value;
  return (    // Wrong! There is no need to specify the key here:
<li key={value.toString()}>     {value}
    </li>
  );}

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
// Wrong! The key should have been specified here:
<ListItem value={number} />  );
  return (
    <ul>
     {listItems}
    </ul>
  );}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root'));
```

Notice:
Keys only make sense in the context of the surrounding array.

Notice: **Keys are not passed to Components**. Here for example key will not be available in the props object of the Post component.

```
const content = posts.map((post) =>
  <Post
    key={post.id}
    id={post.id}
    title={post.title} />);

function Post(props){
    // there is no props.key
    const id = props.id
    const title = props.title
    ...
};
```

Fragments

Notice: The render method can only render a single root html node that contains the other html nodes. From react v16 you can use the Fragments node to return more than one nodes. Actually what you do is to group a list of children together.

A common pattern in React is for a component to return multiple elements. Fragments let you group a list of children without adding extra nodes to the DOM.

The empty tag `<> ... </>` is shorthand for <React.Fragment>

```
class Table extends React.Component {
  render() {
    return (
      <table>
        <tr>
          <Columns />  // this component wants to return many nodes
        </tr>
      </table>
    );
  }}

// the Columns
render() {
  return (
    <React.Fragment>
      <td/>
```

Render many html nodes using Fragments.

Notice:
If you want to return an array of fragments you must define keys for the React.Fragment nodes.

```
function Glossary(props) {
  return (
    <dl>
      {props.items.map(item => (
        // Without the `key`, React will fire a key warning
        <React.Fragment key={item.id}>
          <dt>{item.term}</dt>
          <dd>{item.description}</dd>
        </React.Fragment>
      ))}
```

```
    <td/>
    <td/>
  </React.Fragment>
);}
```

```
  </dl>
);}
```

Portals

Normally the rendered node that is returned from a render method, is mounted into the nearest parent node. Using portals you can mount the rendered node to any existing node in the DOM.

```
render() {
  // React does *not* create a new div. It renders the children into `domNode`.
  // `domNode` is any valid DOM node, regardless of its location in the DOM.
  return ReactDOM.createPortal(
    this.props.children,
    domNode,
    );
  }
```

A typical use case for portals is when a parent component has an overflow: hidden or z-index style, but you need the child to visually "break out" of its container. For example, dialogs, hovercards, and tooltips.

Notice
The portal still exists in the React tree regardless of position in the DOM tree. This includes event bubbling. An event fired from inside a portal will propagate to ancestors in the containing React tree, even if those elements are not ancestors in the DOM tree.

Lifecycle methods

Have in mind that componentDidUpdate(prevProps, prevState) gets two optional arguments

State

React has stateless and stateful components. **Class defined components can have a state available to them while function defined components can't.** This state is local (or encapsulated). It is not accessible to any component other than the one that owns and sets it. No other component can and should know if a certain component is stateless or not. The state is like props, but it is private and fully controlled by the component. If you don't use something in render(), it shouldn't be in the state. **React will preserve this state between re-renders**.

- props are passed from a parent component, but state is managed by the component itself. The most important difference between state and props
- A component cannot change its props, but it can change its state.
- Components should only update their own state.

When the state (or the props) changes the component responds by re-rendering. (If the props changes the component also responds with re-rendering, but props must not change by the component's code)

For each particular piece of changing data, there should be just one component that "owns" it in its state. Don't try to synchronize states of two different components. Instead, lift it up to their closest shared ancestor, and pass it down as props to both of them.

The React philosophy is that props should be immutable and top-down. This means that a parent can send whatever prop values it likes to a child, but the child cannot modify its own props. What you do is react to the incoming props and then, if you want to, modify your child's state based on incoming props.

So you don't ever update your own props, or a parent's props. Ever. You only ever update your own state, and react to prop values you are given by parent.

If you want to have an action occur on a child which modifies something on the parent's state, then what you do is pass a callback to the child which it can execute upon the given action. This callback can then modify the parent's state, which in turn sends different props to the child on re-render (and the child id re-rendered if props changed).

To answer the question of why

In React, props flow downward, from parent to child. This means that when we call ReactDOM.render, React can render the root node, pass down any props, and then forget about that node. It's done with it. It's already rendered. This happens at each component, we render it, then move on down the tree, depth-first. If a component could mutate its props, we would be changing an object that is accessible to the parent node, even after the parent node had already rendered. This could cause all sorts of strange behaviour, for example, a user.name might have one value in one part of the app, and a different value in a different part, and it might update itself the next time a render is triggered.

Modifying Props would be two-way-binding. Mutating props would be a form of two-way binding. We would be modifying values that might be relied on by another component higher up the tree. Angular 1 had this, you could change any data anytime from wherever you were. In order to work, it needed a cyclical $digest. Basically, it would loop around and around, re-rendering the DOM, until all the data had finished propagating. This was part of the reason why Angular 1 was so slow.

The state is not persistent, it is flushed if the page reloads. For persistent storage you can use:

- Cookies (Persistent cookies or session cookies)
- Web storage

See cookies vs web storage (web storage offers a lot of advantages)

Lifecycle hooks

componentDidMount and componentWillUnmount methods of a class component are called after the component output has been rendered to the DOM and before it is removed respectively.

While this.props is set up by React itself and this.state has a special meaning, you are free to add additional fields to the class manually if you need to store something that is not used for the visual output. So you can do this.my_var = whatever.

Sequence of actions explained

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }

  componentWillUnmount() {
    clearInterval(this.timerID);
  }

  tick() {
  this.setState({     date: new Date()    });
```

1. When <Clock /> is passed to ReactDOM.render(), React calls the constructor of the Clock component. Since Clock needs to display the current time, it initializes this.state with an object including the current time. We will later update this state.

2. React then calls the Clock component's render() method. This is how React learns what should be displayed on the screen. React then updates the DOM to match the Clock's render output.

3. When the Clock output is inserted in the DOM, React calls the componentDidMount() lifecycle hook. Inside it, the Clock component asks the browser to set up a timer to call the component's tick() method once a second.

4. Every second the browser calls the tick() method. Inside it, the Clock component schedules a UI update by calling setState()

```
}

 render() {
  return (
   <div>
    <h1>Hello, world!</h1>
    <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
   </div>
  );
 }}

ReactDOM.render(
 <Clock />,
 document.getElementById('root'));
```

with an object containing the current time. Thanks to the setState() call, React knows the state has changed, and calls render() method again to learn what should be on the screen. This time, this.state.date in the render() method will be different, and so the render output will include the updated time. React updates the DOM accordingly.

5. If the Clock component is ever removed from the DOM, React calls the componentWillUnmount() lifecycle hook so the timer is stopped.

The setState function updates the state. It does so by shallow merging the state object with the object that is returned by the setState function. It has two forms. One that accepts an object as argument and one that accepts a function (that returns an object). The first form is suitable for explicitly defining some state attributes. The second one is suitable for updating the state by accessing the previous state too if needed.

Notice about the state

1. don't modify the state directly

```
// Wrong
this.state.comment = 'Hello';
// Correct
this.setState( {comment: 'Hello'} );
```

You must not modify the state directly for example by doing this.state.comment = 'Hello' . This will not trigger the render action! Instead use setState() function

1. State Updates May Be Asynchronous

That's why you need to call setState with a function that has prevState as an argument, if you want to access the previous state.

```
// Wrong
this.setState({
 counter: this.state.counter + this.props.increment,});

// Correct
this.setState((prevState, props) => ({
 counter: prevState.counter + props.increment}));

// Correct
this.setState(function(prevState, props) {
 return {
  counter: prevState.counter + props.increment
 };});
```

React may batch multiple setState() calls into a single update for performance.

In the wrong case you are not sure if this.state.counter is the correct value (the last updated value). Since its update is asynchronous it might have not been updated yet.

So instead you must use the setState with a function as an argument. Either arrow function or a regular function. A function that returns an object.

Why a function, how does it make a difference? Since setState calls are batched, this lets you chain updates and ensure they build on top of each other instead of conflicting

Probably it is stored in a queue and is executed with the proper order. Maybe it has something to do with the redux concept of modifying the state with actions that are functions. But this state here is a local state, specific to this component while for redux it is the global state, of all components and an action can affect many components.

3. State updates are shallow merged

The state is an object. When you want to update it, your action (function) returns an object that is merged with the state. It is a shallow merge (In a shallow merge, the properties of the first object are overwritten with the same property values of the second object).

Unidirectional data flow

One way binding or Unidirectional data flow

Components can't modify their props (which might affect the output of parent components). props only flow in one way, top down. This way you don't have to deal again with a component that has already rendered.
A component may choose to pass its state down as props to its child components: `<FormattedDate date={this.state.date} />`
The FormattedDate component would receive the date in its props and wouldn't know whether it came from the Clock's state, from the Clock's props, or was typed by hand. This is commonly called a "top-down" or "unidirectional" data flow. Any state is always owned by some specific component, and any data or UI derived from that state can only affect components "below" them in the tree. If you imagine a component tree as a waterfall of props, each component's state is like an additional water source that joins it at an arbitrary point but also flows down.

This means that the state or the UI of a component can only affect child components.

Inverse data flow

The children can modify the state of their parent indirectly using callbacks. React makes this data flow explicit to make it easy to understand how your program works, but it does require a little more typing than traditional two-way data binding. This is done like this: The parent passes some callbacks to the children. These callbacks modify the parent's state. They are methods of the Parent component. When the children want to modify the state they execute these callbacks that have been passed to them as props.

Event handling

``// html <button onclick="activateLasers()">Activate Lasers</button> // jsx <button onClick={activateLasers}> Activate Lasers</button>```	Vs html syntax With JSX you pass a function as the event handler, rather than a string. You cannot return false to prevent default behavior in React. You must call preventDefault explicitly.

Synthetic Event

React uses a cross browser wrapper of the browser's native event object, called SyntheticEvent.

With the nativeEvent attribute of the syntheticEvent you can access the underlying browser event.

The SyntheticEvent is pooled for performance (from v17 on, SyntheticEvent is not pooled). Pooling means that when the event callback has been invoked, the same event instance will be reused and for that reason its attributes will be nullified. So you must not access it asynchronously since it will not be the same event.

Normally you add an event listener to an element that already exists in the DOM. In react you can just provide the event listener when the element is initially rendered (it is added the first time the element is rendered).

Event bubbling vs Event Capturing

Event bubbling and capturing are two ways of event propagation in the HTML DOM API, when an event occurs in an element inside another element, and both elements have registered a handle for that event. The event propagation mode determines in which order the elements receive the event.

- With bubbling, the event is first captured and handled by the innermost element and then propagated to outer elements.
- With capturing, the event is first captured by the outermost element and propagated to the inner elements

You can explicitly define which mode you want to use. By default bubbling is used. To register an event handler for the capture phase, append Capture to the event name.

Bind

```
class Toggle extends React.Component {
 constructor(props) {
  super(props);
  this.state = {isToggleOn: true};
  // This binding is necessary to make `this` work in the callback
  this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState (
      prevState => ({     isToggleOn: !prevState.isToggleOn    })
    );
  }

 render() {
  return (
      <button onClick={this.handleClick}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
    </button>
  );
 }}

ReactDOM.render(
 <Toggle />,
 document.getElementById('root'));
```

In JavaScript, class methods are not bound by default. So you have to bind them in order to use this when the function is actually called.. This is not React-specific behavior; it is a part of how functions work in JavaScript.

Generally, if you refer to a method without () after it, such as onClick={this.handleClick}, you should bind that method.

Arrow function expressions preserve this, so you need to bind the event handlers, only when you are not using arrows.

```
// If you want to avoid using bind
class LoggingButton extends React.Component {
 handleClick() {
  console.log('this is:', this);
 }

 render() {
 // This syntax ensures `this` is bound within handleClick
return (
    <button onClick={(e) => this.handleClick(e)}>  Click me
    </button>
  );
 }}
```

Here an arrow function expression invokes the function and preserves this, so this way there is no need to bind the method.

Passing additional arguments to event handlers

```
<button onClick={(e) => this.deleteRow(id, e)}>Delete Row</button>

<button onClick={this.deleteRow.bind(this, id)}>Delete Row</button>
```

Either of these would work, but the second doesn't preserve this inside the deleteRow function. In this

	case if you need this, you have to bind the deleteRow method in the class constructor.
class Toggle extends React.Component { constructor(props) { super(props); this.state = {isToggleOn: true}; } handleClick(id, e) { .... } render() {return (<button onClick={(e) => this.handleClick(id, e)}> Click me </button>); }}	

The dataset property

handleClick(e) { this.setState({ justClicked: e.target.*dataset*.letter }); } // somewhere in the render function <li key={letter} *data-letter={letter}* onClick={this.handleClick}> {letter} 	You use the data-my-var-name html attribute Notice that we use the dataset property to get the data- attributes.

Have in mind that you can prevent a function from being called too quickly or too many times in a row (for example onScroll event handlers) using either **throttling** (Throttling prevents a function from being called more than once in a given window of time.)

When testing your rate limiting code works correctly it is helpful to have the ability to fast forward time.

Forms

Controlled components

Form inputs are stored in React state which is the "single source of truth". The form input controls the react state which controls the components rendering. This is a controlled component. Controlled by the form's input.

An HTML form has the behavior of browsing to a new page when the user submits the form. If you want this behavior in React, it just works. But in most cases, it's convenient to have a JavaScript function that handles the submission of the form and has access to the data that the user entered into the form. The standard way to achieve this is with a technique called "controlled components".

In HTML, form elements such as <input>, <textarea>, and <select> typically maintain their own state and update it based on user input. (for example when you type in an input field the text is appended, it doesn't replace the existing text). In React, mutable state is typically kept in the state property of components, and only updated with setState().

We can combine the two by making the React state be the "single source of truth". Then the React component that renders a form also controls what happens in that form on subsequent user input. An input form element whose value is controlled by React in this way is called a "controlled component".

Textarea: In React, a <textarea> uses a value attribute to define the text while in html the text is the child text of the textarea element.

Select: In React the selected value is not defined with the selected attribute as in html, but by defining a value attribute in the select element.

In the following example we just make the reservation component's state, the single source of truth for the form. There is no code for handling the submission of the form. The point is that whatever we do in the form, is reflected in the component's state. To do so, you have to create a new method which will be a submit listener and would get the form values from the component's state.

```
class Reservation extends React.Component {
 constructor(props) {
  super(props);
  this.state = {
   isGoing: true,
   numberOfGuests: 2
  };
  this.handleInputChange = this.handleInputChange.bind(this);
}

 handleInputChange(event) {
  const target = event.target;
  const value = target.type === 'checkbox' ? target.checked : target.value;
  const name = target.name;
  this.setState({
      [name]: value    });  // this is ES6 computed property name syntax
 }

 render() {
  return (
   <form>
    <label>
     Is going:
     <input
      name="isGoing"
      type="checkbox"
      checked={this.state.isGoing}
      onChange={this.handleInputChange} />
    </label>
    <br />
    <label>
     Number of guests:
     <input
      name="numberOfGuests"
      type="number"
      value={this.state.numberOfGuests}
      onChange={this.handleInputChange} />
    </label>
   </form>
  );
```

Overall, this makes it so that <input type="text">, <textarea>, and <select> all work very similarly - they all accept a **value attribute** that you can use to implement a controlled component.

To select multiple options in a select:
<select multiple={true} value={['B', 'C']}>

Multiple input elements in one form
When you need to handle multiple controlled input elements (with the same submit handler since it is one form), you can add a name attribute to each element and let the handler function choose what to do based on the value of event.target.name.

If a controlled component has taken a value then it is not editable anymore by the browser. You can make it so, if you define its value as {null}

or

```
onChange = e => this.setState({ [e.target.name]:
e.target.value });

onSubmit = e => {
  e.preventDefault();
  console.log("submit");
};
```

}}	

when we have a form, we want each input to be part of the state of the component. This is the controlled component.

If we want to clear the input after a form submission, we do a **this.setState({input_field_1: "", input_field_2: "", etc.})**

Notice that this is the component's state, not the redux global state.

Uncontrolled components

They are based in React Refs. You get a reference to the input DOM element, you take its value and use this value to submit the form instead of using the components state value.

In a controlled component, form data is handled by a React component's state. The alternative is uncontrolled components, where form data is handled by the DOM itself. Since an uncontrolled component keeps the source of truth in the DOM, it is sometimes easier to integrate React and non-React code when using uncontrolled components. Using uncontrolled components might be easier to implement, when you are converting a preexisting codebase to React.

Refs

React supports a special attribute that you can attach to any component. The ref attribute can be an object created by React.createRef() function or a callback function, or a string (in legacy API). When the ref attribute is a callback function, the function receives the underlying DOM element or class instance (depending on the type of element) as its argument. This allows you to have direct access to the DOM element or component instance.

Use refs sparingly. If you find yourself often using refs to "make things happen" in your app, consider getting more familiar with top-down data flow.

class MyComponent extends React.Component { constructor(props) { super(props); this.myRef = React.createRef(); } render() { return ; } }	It is a special React attribute. The ref attribute takes a callback function, and the callback will be executed immediately after the component is mounted or unmounted. When the ref attribute is used on an HTML element, the ref callback receives the underlying DOM element as its argument. This way you can access the underlying browser DOM element and use it. So, I guess that this.myRef references the div node and you can use it in your component as this.myRef. In this example there is no further use of it. You can also use ReactDOMServer.findDOMNode but ref is preferable.

forwardRef

me: we use the forwardRef to define a function component, so that it receives an attribute called ref (that references a DOM node), and use it in our code.

const FancyButton = React.forwardRef((props, ref) => (<button ref={ref} className="FancyButton"> {props.children} </button>)); // You can now get a ref directly to the DOM button: const ref = React.createRef(); <FancyButton ref={ref}>Click me!</FancyButton>;	In the example below, FancyButton uses React.forwardRef to obtain the ref passed to it, and then forward it to the DOM button that it renders. Notice **that this way we can get a reference to the button in another part of the code using the createRef() function. Here the ref variable refers to the actual button element**. This way, components using FancyButton can get a ref to the underlying button DOM node and access it if necessary—just like if they used a DOM button directly.

Lifting state up

In React, sharing state is accomplished by moving it up to the closest common ancestor of the components that need it and passing it down as props. This is called "lifting state up".

In this case for example, the TemperatureInput components are children of the Calculator component. If we want them to "share" some state, then we put this state in the Calculator and pass it to its children components as props. This is the "lifting state up" concept. There should be a single "source of truth" for any data that changes in a React application.

```
const scaleNames = {  c: 'Celsius',  f: 'Fahrenheit'};

class TemperatureInput extends React.Component {
 constructor(props) {
  super(props);
  this.handleChange = this.handleChange.bind(this);
 }

 handleChange(e) {
  this.props.onTemperatureChange(e.target.value);  }

 render() {
  const temperature = this.props.temperature;
  const scale = this.props.scale;
  return (
   <fieldset>
    <legend>Enter temperature in {scaleNames[scale]}:</legend>
    <input value={temperature}
        onChange={this.handleChange} />
   </fieldset>
  );
 }}
```

```
class Calculator extends React.Component {
 constructor(props) {
  super(props);
  this.handleCelsiusChange = this.handleCelsiusChange.bind(this);
  this.handleFahrenheitChange = this.handleFahrenheitChange.bind(this);
  this.state = {temperature: '', scale: 'c'};  }

 handleCelsiusChange(temperature) {
  this.setState({scale: 'c', temperature});  }

 handleFahrenheitChange(temperature) {
  this.setState({scale: 'f', temperature});  }

 render() {
  const scale = this.state.scale;
  const temperature = this.state.temperature;
  const celsius = scale === 'f' ?
     tryConvert(temperature, toCelsius) : temperature;
  const fahrenheit = scale === 'c' ?
     tryConvert(temperature, toFahrenheit) : temperature;
  return (
   <div>
```

We have 2 input elements, one for Celcius and one for Fahrenheit and we want them to be in sync. This is achieved by using a common state, the state of their parent.

Inverse data flow

Notice that the onChange handler is provided as a callback by the parent (as props), since this callback modifies the state which is owned by the parent. The attributes (input values) of the callback though are given from the child.

```
function toCelsius(fahrenheit) {
 return (fahrenheit - 32) * 5 / 9;}

function toFahrenheit(celsius) {
 return (celsius * 9 / 5) + 32;}

function tryConvert(temperature, convert) {
 const input = parseFloat(temperature);
 if (Number.isNaN(input)) {
  return '';
 }
 const output = convert(input);
 const rounded = Math.round(output * 1000) / 1000;
 return rounded.toString();}
```

```
function BoilingVerdict(props) {
 if (props.celsius >= 100) {
  return <p>The water would boil.</p>;
  }
   return <p>The water would not boil.</p>;
}
```

Usually, the state is first added to the component that needs it for rendering. Then, if other components also need it, you can lift it up to their closest common ancestor.

If something can be derived from either props or state, it probably shouldn't be in the state.

```
    <TemperatureInput
      scale="c"
      temperature={celsius}
      onTemperatureChange={this.handleCelsiusChange} />
    <TemperatureInput
      scale="f"
      temperature={fahrenheit}
      onTemperatureChange={this.handleFahrenheitChange} />
    <BoilingVerdict        celsius={parseFloat(celsius)} />
  </div>
 );
}}
```

Composition

In general react favors composition instead of inheritance for its components. They haven't find a use case that inheritance would be necessary. Components may accept arbitrary props, including primitive values, React elements, or functions. If you want to reuse non-UI functionality between components, we suggest extracting it into a separate JavaScript module. The components may import it and use that function, object, or a class, without extending it.

Containment (props.children)

The specialized component is the parent of the generic one and it defines the contents of the generic one as children elements. The generic component access them with props.children. Unlike the other props which are the attributes passed to the used component, the props.children are any components that the used component wraps.

Replacing this

With this

```
const AlbumCard = ({ albumCoverUrl, title, artist, genres, songs }) => {
  return (
    <Card title={title} img={albumCoverUrl} >
      <h2>{artist}</h2>
      <ul>{genres.map(genre => <li>{genre}</li>)}</ul>
      <div>{songs.map(song => <SongCard {...song} />)}</div>
    </Card>
  )



const SongCard = props => {
  return (
    <Card title={props.title}>
      <p>{props.songLength}</p>
    </Card>
  )



const Card = props => {
  return (
    <div className="card">
      {props.img && <img src={props.img} />}
      <h1>{props.title}</h1>
      <div className="card-content">
        {props.children}
      </div>
    </div>
  )
}
```

```jsx
const AlbumCard = props => {
  return (
    <div className="card">
      <img src={props.albumCoverUrl} />
      <AlbumInfo {...props} />
      ...
    </div>
  )
}

const AlbumInfo = ({ title, artist, genres, songs }) => {
  return (
    <div>
      <h1>{title}</h1>
      <h2>{artist}</h2>
      <ul>{genres.map(genre => <li>{genre}</li>)}</ul>
      <SongContainer songs={songs} />
    </div>
  )
}

const SongContainer = props => {
  return (
    <div>
      {props.songs.map(song => <SongCard {...song} />)}
      ...
    </div>
  )
}

const SongCard = props => {
  return (
    <div>
      <h1>{props.title}</h1>
      <p>{props.songLength}</p>
    </div>
  )
}
```

The concept is that we create a more generic Card component that doesn't know its children beforehand. They are passed to it by the more specialized components AlbumCard and SongCard. This is composition. In an inheritance model the specialized components would inherit from the generic one.

```jsx
function FancyBorder(props) {
 return (
  <div className={'FancyBorder FancyBorder-' + props.color}>
    {props.children}
  </div>
 );}

function WelcomeDialog() {
 return (
  <FancyBorder color="blue">
    <h1 className="Dialog-title">      Welcome     </h1>
    <p className="Dialog-message">
      Thank you for visiting our spacecraft!     </p>
  </FancyBorder>
 );}
```

props.children

You can have some components that don't know their children components ahead of time. This is especially common for components like Sidebar or Dialog that represent generic "boxes". In these cases you can use props.children, a special props attribute.

Whatever is inside the FancyBorder tag, is passed as props.children to the FancyBorder container.

```jsx
function SplitPane(props) {
 return (
  <div className="SplitPane">
   <div className="SplitPane-left">
```

Template Blocks like functionality

React elements like <Contacts /> and <Chat /> are just objects, so you can pass them as props like any other data.

```
      {props.left}     </div>
    <div className="SplitPane-right">
      {props.right}     </div>
  </div>
 );}

function App() {
 return (
  <SplitPane
    left={       <Contacts />     }
    right={       <Chat />     } />
 );}
```

Specialization

Sometimes we think about components as being "special cases" of other components. For example, we might say that a WelcomeDialog is a special case of Dialog. You can pass the contents of a component, for example the title and message texts, as props and reuse this component with different messages and titles.

Higher Order components

A higher order component (HOC) is a function that takes a component and returns a new component. HOCs are common in third-party React libraries, such as Redux's connect and Relay's createFragmentContainer.

The concept is that we have some logic (usually non visual logic) that we want to share between various components. Instead of copy pasting code we create a wrapper component that contains the logic and wrap the components with it. The common logic lives in the wrapper component and becomes available as props in the wrapped components. The wrapped components accept the shared logic as props because the wrapper renders them.

In this example Datasource is a service (a 3rd part app, our API etc) that we listen to, for changes in data used by the client.

```
// This function takes a component...
function withSubscription(WrappedComponent, selectData) {
 // ...and returns another component...
 return class extends React.Component {
  constructor(props) {
   super(props);
   this.handleChange = this.handleChange.bind(this);
   this.state = {
    data: selectData(DataSource, props)
   };
  }

  componentDidMount() {
   // ... that takes care of the subscription...
   DataSource.addChangeListener(this.handleChange);
  }

  componentWillUnmount() {
   DataSource.removeChangeListener(this.handleChange);
  }

  handleChange() {
   this.setState({
    data: selectData(DataSource, this.props)
```

Many components of our app can have the following behaviour:

- On mount, add a change listener to DataSource.
- Inside the listener, call setState whenever the data source changes.
- On unmount, remove the change listener.

You can imagine that in a large app, this same pattern of subscribing to DataSource and calling setState will occur over and over again. We want an abstraction that allows us to define this logic in a single place and share them across many components. This is where higher-order components excel. (In inheritance terms, think of them as generic parent Classes)

In this example the withSubscription is a HOC. Each component that uses it takes a different data source call.

When components are rendered they will be passed a data prop (props.data) with the most current data retrieved from DataSource.

So we create a new wrapped component called CommentListWithSubscription which wraps the

```
    });
  }

  render() {
    // ... and renders the wrapped component with the fresh data!
    // Notice that we pass through any additional props
    return <WrappedComponent data={this.state.data} {...this.props} />;
  }
};}
```

```
const CommentListWithSubscription = withSubscription(
  CommentList,
  (DataSource) => DataSource.getComments()
);

const BlogPostWithSubscription = withSubscription(
  BlogPost,
  (DataSource, props) => DataSource.getBlogPost(props.id)
);
```

CommentList component, and we use that. This way the CommentListWithSubscription renders the CommentList component and passes it a props.data (from its wrapper withSubscription). Of course, it can also have any other props that we pass when we render it:

```
<CommentListWithSubscription list-id={list-id}></ >
```

It will pass the props.data and props.list-id to the CommentList and will render whatever the CommentList renders.

Note that a HOC doesn't modify the input component, nor does it use inheritance to copy its behavior. Rather, a HOC composes the original component by wrapping it in a container component. A HOC is a pure function with zero side-effects.

The most common signature for HOCs looks like this:

```
// React Redux's `connect`
const ConnectedComment = connect(commentSelector, commentActions)
(CommentList);

// connect is a function that returns another function
const enhance = connect(commentListSelector, commentListActions);
// The returned function is a HOC, which returns a component that
// is connected to the Redux store
const ConnectedComment = enhance(CommentList);
```

In other words, connect is a higher-order function that returns a higher-order component!

```
// Instead of doing this...
const EnhancedComponent =
 withRouter(connect(commentSelector)(WrappedComponent))

// ... you can use a function composition utility
// compose(f, g, h) is the same as (...args) => f(g(h(...args)))
const enhance = compose(
  // These are both single-argument HOCs
  withRouter,
  connect(commentSelector))

const EnhancedComponent = enhance(WrappedComponent)
```

This form may seem confusing or unnecessary, but it has a useful property. Single-argument HOCs like the one returned by the connect function have the signature Component => Component. Functions whose output type is the same as its input type are really easy to compose together. This is what you do here.

The **compose utility function** is provided by many third-party libraries including lodash (as lodash.flowRight), Redux, and Ramda.

- Don't Use HOCs Inside the render Method
- Static Methods Must Be Copied Over
- If you add a ref to an element whose component is the result of a HOC, the ref refers to an instance of the outermost container component, not the wrapped component.

The render props

The term "render prop" refers to a technique for sharing code between React components using a prop whose value is a function.

It's important to remember that just because the pattern is called "render props" you don't have to use a prop named render to use this pattern. In fact, any prop that is a function that a component uses to know what to render is technically a "render prop". some times it is called "children".

```
class Cat extends React.Component {
  render() {
    const mouse = this.props.mouse;
    return (
      <img src="/cat.jpg" style={{ position: 'absolute', left: mouse.x, top: mouse.y }} />
    );
  }
}

class Mouse extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
    this.state = { x: 0, y: 0 };
  }

  handleMouseMove(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  }

   render() {
     return (
       <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>

         {/*
           Instead of providing a static representation of what <Mouse> renders,
           use the `render` prop to dynamically determine what to render.
         */}
         {this.props.render(this.state)}
       </div>
     );
   }
 }

 class MouseTracker extends React.Component {
   render() {
     return (
       <div>
         <h1>Move the mouse around!</h1>
         <Mouse render={mouse => (
           <Cat mouse={mouse} />
         )}/>
       </div>
     );
   }
 }
```

A component with a render prop takes a function that returns a React element and calls it instead of implementing its own render logic.

You follow the approach on the left instead of

```
render() {
  return (
    <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>

      {/*
        We could just swap out the <p> for a <Cat> here ... but then
        we would need to create a separate <MouseWithSomethingElse>
        component every time we need to use it, so <MouseWithCat>
        isn't really reusable yet.
      */}
      <Cat mouse={this.state} />
    </div>
  );
}
}

class MouseTracker extends React.Component {
render() {
  return (
    <div>
      <h1>Move the mouse around!</h1>
      <MouseWithCat />
    </div>
  );
}
}
```

While in the left, we don't have a MouseWithCat but just a Mouse component that renders whatever receives in its render props. So we could easily add an Elephant component and render a mouse with an elephant picture.

In this example the mouse argument of MouseTracker's Mouse component is actually the this.state in the handleMouseMove.

Error boundaries

```
class ErrorBoundary extends React.Component {
 constructor(props) {
  super(props);
  this.state = { hasError: false };
 }

 componentDidCatch(error, info) {
    // Display fallback UI
   this.setState({ hasError: true });
   // You can also log the error to an error reporting service
   logErrorToMyService(error, info);
  }

 render() {
```

Error boundaries (available from v16) are React components that catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI instead of the component tree that crashed. Error boundaries catch errors during rendering, in lifecycle methods, and in constructors of the whole tree below them.
Note! Error boundaries do not catch errors for:

- Event handlers
- Asynchronous code (e.g. setTimeout or requestAnimationFrame callbacks)
- Server side rendering
- Errors thrown in the error boundary itself (rather than its children)
- Only class components can be error boundaries

```
  if (this.state.hasError) {
    // You can render any custom fallback UI
    return <h1>Something went wrong.</h1>;
  }
  return this.props.children;
 }}

// Then you can use it as a regular component:
<ErrorBoundary><MyWidget /></ErrorBoundary>
```

Where to put them
The granularity of error boundaries is up to you. You may wrap top-level route components to display a "Something went wrong" message to the user, just like server-side frameworks often handle crashes. You may also wrap individual widgets in an error boundary to protect them from crashing the rest of the application. In practice, most of the time you'll want to declare an error boundary component once and use it throughout your application.

Note! As of React 16, errors that are not caught by any error boundary will result in unmounting of the whole React component. tree. They think that it is preferable to show nothing to the user instead of showing wrong info of a broken component.

Its better to use error boundaries than try catch blocks, since the later only works for imperative code. Error boundaries preserve the declarative nature of React.

Error boundaries do not catch errors inside event handlers. React doesn't need error boundaries to recover from errors in event handlers. Unlike the render method and lifecycle hooks, the event handlers don't happen during rendering. So if they throw, React still knows what to display on the screen. For example you can define a try/catch block.

Why error boundaries are not HOCs?

Me: HOCs are used to build individual components. If error boundary was a HOC you would have to use it like MyComponentWithErrorBoundary for each of your components.

Routers

How a router can work:

- It listens to browser navigation events (popstate) in a way that works for all browsers
- When such an event fires, they get the current url and try to match it with the ones (the routes) that you have defined.
- Each route can be an object that has an action attribute which is a function. In case of match, this specific route's action is called.
- The action function calls the api, gets the response data, and returns a jsx component with the data as props like <MyComponent/ {...data}>
- The router renders the given component.

In order to create browser navigation events you use the html5 history api. There are libraries that wrap it and allow you to easily make **the links of your app to push a new browser navigation entry whenever they are clicked**. They usually do this by offering a custom **<link>** component that you use instead of <a> in your app. (This is one of the reasons the custom link element was created).

```
import ReactDOM from 'react-dom';
import history from './history';
import router from './router';
import routes from './routes';
const container = document.getElementById('root');

function renderComponent(component) {
  ReactDOM.render(component, container);
}
```

I guess that something similar is done by the react-router package in the background.

```
function render(location) {
 router.resolve(routes, location) // match the location (the url)
  .then(renderComponent)
  .catch(error => router.resolve(routes, { ...location, error }))
  .then(renderComponent));
}

render(history.getCurrentLocation()); // render the current URL
history.listen(render); // listen to navigation events and call the render function when one happens
```

React Router

the flow in a nutshell

1. You are on the postlists page.
2. You press the "View Post" link of an individual post
3. The url changes
4. The change triggers the wrapped by the Router main App to rerender
5. The new url is matched by the Route to singlePostPage. The part of the url after the /posts/ is matched and stored as a key value pair in the match.params object, where the key is named postId.
6. The singlePostPage receives a match object as a prop, from which it extracts the postId. The component is rendered because its props changes. Since the component is rendered, the useSelector hook runs and collects a new post from the store.

Introduction

this library has three variants:

=> react-router: the core library
=> react-router-dom: a variant of the core library meant to be used for web applications
=> react-router-native: a variant of the core library used with react native in the development of Android and iOS applications.

Both react-router-dom and react-router-native import all the functionality of the core react-router library.

The react-router package includes several routers that we can take advantage of depending on the platform we are targeting. These include BrowserRouter, HashRouter, and MemoryRouter. The BrowserRouter is used for applications which have a dynamic server that knows how to handle any type of URL whereas the HashRouter is used for static websites with a server that only responds to requests for files that it knows about.

```
ReactDOM.render(
  <BrowserRouter>
    <App/>
  </BrowserRouter>,
  document.getElementById('root'));
```

Each Router provider creates a history object that it uses to keep track of the current location and **re-renders the wrapped components (the wrapped main App component) whenever this location changes**. The history object contains the location object. The location object within the history object is shaped like so *{ pathname, search, hash, state }*. The location object properties are derived from the application URL.

The Router low level object

I saw that these two are identical. But material UI uses the second approach with react router version 6. Why? The difference is the use of a custom history object. Why we need a custom one?

(I saw by someone to mention that he needed the custom history object to be able to programmatically navigate your users. But I don't think this is necessary since you can access the history object from the useHistory hook).

```
import { BrowserRouter } from 'react-router-dom'

<BrowserRouter>
  <App />
</BrowserRouter>
```

```
import { Router } from 'react-router-dom'
import { createBrowserHistory } from 'history'

const history = createBrowserHistory()

<Router history={history}>
  <App />
</Router>
```

According to the official documentation The most common use-case for using the low-level <Router> is to synchronize a custom history with a state management lib like Redux or Mobx. Note that this is not required to use state management libs alongside React Router, it's only for deep integration.

With Router and custom history object you can do things like listening to location changes - history.listen((location, action) => {}); probably you can get the history object from the useHistory hook.

The Route component

The <Route/> component of the react-router library is one of the most important building blocks in the React Router package. It renders the appropriate user interface when the current location matches the route's path. When a path is matched, a React component should be rendered so that there's a change in the UI.

```
<Route path="/items"/>
<Route exact path="/items" component={Items} />

<Route exact path="/items"
render={() => (<div>List of Items</div>)} />

const cat = {category: "food"}
<Route exact path="/items"
render={props => <Items {...props} data={cat}/>} />

<Route children={props => <Items {...props}/>}/>
```

It is also worth noting that the **Path-to-RegExp package** is used by the react-router package to turn a path string into a regular expression and matched against the current location.

In the first render example, when the current location matches the path exactly, a React element is created and the string List of Items is rendered in the browser.
In the second example, *data* represents the extra props that are passed to the Items component. Here, cat is passed in as the extra prop.

The <Route/> component provides three props that can be used to determine which component to render:

- component

The component prop defines the React element that will be returned by the Route when the path is matched. This React element is created from the provided component using React.createElement.

- Render

The render prop provides the ability for inline rendering and passing extra props to the element.

- Children

The children prop is similar to the render prop since it always expects a function that returns a React element. The major difference is that the element defined by the child prop is returned for all paths irrespective of whether the current location matches the path or not.

Notice: if more than one route components are matched, they will all be rendered. This is by design, allowing us to compose <Route>s into our apps in many ways, like sidebars and breadcrumbs, bootstrap tabs, etc.

The Switch component

The react-router library also contains a <Switch/> component that is used to wrap multiple <Route/> components. Renders the first child <Route> or <Redirect> that matches the location.

```
let routes = (
  <div>
    <Route path="/about">
      <About />
    </Route>
    <Route path="/:user">
      <User />
    </Route>
    <Route>
      <NoMatch />
    </Route>
  </div>
);
```

How is this different than just using a bunch of <Route>s?
<Switch> is unique in that it renders a route exclusively. In contrast, every <Route> that matches the location renders inclusively.
If the URL is /about, then <About>, <User>, and <NoMatch> will all render because they all match the path. This is by design, allowing us to compose <Route>s into our apps in many ways, like sidebars and breadcrumbs, bootstrap tabs, etc.
Occasionally, however, we want to pick only one <Route> to render. If we're at /about we don't want to also match /:user (or show our "404" page).

The Link component

The react-router package also contains a <Link/> component that is used to navigate the different parts of an application by way of hyperlinks. It is similar to HTML's anchor element but the main difference is that using the Link component does not reload the page but rather, changes the UI and updates the url.

```
export const Home = () => (
  <div>
    Home Component
    <ul>
      <li>
        <Link to="/items">Items</Link>
      </li>
      <li>
        <Link to="/category">Category</Link>
       </li>
    </ul>
  </div>
);
```

The <Link/> component uses "to" as a prop to define the location to navigate to. The to prop can either be a string or a location object. If it is a string, it is converted to a location object. Note that the pathname must be absolute.

Nested routing

When a location and a router's path are successfully matched, a **match object** is created. This object contains information about the URL and the path. This information can be accessed as properties on the match object. Let's take a closer look at the properties:

=> url : A string that returns the *matched* part of the URL
=> path : A string that returns the route's path
=> isExact : A boolean that returns true if the match was exact
=> params : An object containing key-value pairs that were matched by the Path-To-RegExp package.

The following example contains four components (not shown here) one of which is the Category component which demonstrates nested and dynamic routing. Initially we are in the home page url and we press the category link.

```jsx
export const Category = ({match}) => (
  <div>
  <h1>Category Component</h1>
  <h5>Click on a category</h5>
  <ul>
    <li>
      <Link to={`${match.url}/shoes`}>Shoes</Link>
    </li>
    <li>
      <Link to={`${match.url}/food`}>Food</Link>
    </li>
    <li>
      <Link to={`${match.url}/dresses`}>Dresses</Link>
    </li>
  </ul>
);
```

```jsx
<Route
  path={`${match.path}/:categoryName`}
  render={props =>
                (<div>
                  {props.match.params.categoryName} category
                 </div>
                )
          }
/>
```

When the Category link is clicked, a route path is matched and a match object is created and sent as a prop to the Category component. Within the Category component, the match object is destructured in the argument list and links to the three categories are created using match.url. Template literals are used to construct the value of the prop on the Link component to the different /shoes, /food and /dresses URLs.

We also have a Route component so that when the location changes (after we press a specific category) the new location will be matched by the Route element and it will be rendered.

:categoryName is the path parameter within the URL and it catches everything that comes after /category. Passing the value to the path prop in this way saves us from having to hardcode all the different category routes. Also, notice the use of template literals to construct the right path.

A pathname like category/shoes creates a param object {categoryName: "shoes"}

features/posts/SinglePostPage.js

```jsx
import React from 'react'
import { useSelector } from 'react-redux'

export const SinglePostPage = ({ match }) => {
  const { postId } = match.params

  const post = useSelector(state =>
    state.posts.find(post => post.id === postId)
  )

  if (!post) {
    return (
      <section>
        <h2>Post not found!</h2>
      </section>
    )
  }

  return (
    <section>
      <article className="post">
        <h2>{post.title}</h2>
        <p className="post-content">{post.content}</p>
      </article>
    </section>
  )
}
```

We'll use React Router to show this component when the page URL looks like /posts/123, where the 123 part should be the ID of the post we want to show.
React Router will pass in a match object as a prop that contains the URL information we're looking for. When we set up the route to render this component, we're going to tell it to parse the second part of the URL as a variable named postId, and we can read that value from match.params.

```jsx
function App() {
  return (
    <Router>
      <Navbar />
      <div className="App">
        <Switch>
          <Route
            exact
            path="/"
            render={() => (
              <React.Fragment>
                <AddPostForm />
                <PostsList />
              </React.Fragment>
            )}
          />
          <Route exact path="/posts/:postId" component={SinglePostPage} />
          <Redirect to="/" />
        </Switch>
      </div>
    </Router>
  )
}
```

And in the postlist

```jsx
<Link to={`/posts/${post.id}`} className="button muted-button">
  View Post
</Link>
```

So the flow is like this:

1. You are on the postlists page.
2. You press the "View Post" link of an individual post
3. The url changes
4. The change triggers the wrapped by the Router components to rerender
5. The new url is matched by the Route to singlePostPage. The part of the url after the /posts/ is matched and stored as a key value pair in the match.params object, where the key is named postId.

6. The singlePostPage receives a match object as a prop, from which it extracts the postId. The component is rendered since its props changes. Since the component is rendered, the useSelector hook runs and collects a new post from the store.

Have in mind

Nested routing: Prior to v4 you had to use Route inside Route. In v5 you could avoid it with the use of the match object. In v6 it is recommended to use Route within the new Routes component.

In v4 and back

```
<Route path='/topics' component={Topics}>
  <Route path='/topics/:topicId' component={Topic} />
</Route>
```

In v6

https://codesandbox.io/s/react-router-test-jhiol

```
export default function App() {
return (
<BrowserRouter>
 <nav>
  <NavLink to="/">Home</NavLink>
  <NavLink to="Topics">Topics</NavLink>
 </nav>
 <Routes>
  <Route path="/" element={<HomePage />} />
  <Route path="Topics" element={<Topics/>}>
   <Route path="/" element={<TopicsList />} />
   <Route path=":id" element={<Topic/>} />
  </Route>
 </Routes>
</BrowserRouter>
);
}
```

```
function Topics() {
return (
 <div>
  <span>Layout</span>
  <Outlet />
 </div>
);
}
```

```
function TopicsList() {
return (
<div>
 <nav><Link to="2">Topic 2</Link></nav>
 <nav><Link to="3">Topic 3</Link></nav>
</div>
);
}
```

```
function Topic() {
 const { id } = useParams();
 return <div>id: {id}</div>;
}
```

In v5

should become

```
<Route path='/topics' component={Topics} />
```

with

```
const Topics = ({ match }) => (
  <div>
    <h2>Topics</h2>
    <Link to={`${match.url}/exampleTopicId`}>
      Example topic
    </Link>
    <Route path={`${match.path}/:topicId`} component={Topic}/>
  </div>
)
```

Version 6

Routes Component

The Routes component essentially replaces the Switch component.

New features including relative routing and linking, automatic route ranking, and nested routes and layouts.

You can either split routes in different places

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

function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="users/*" element={<Users />} />
      </Routes>
    </BrowserRouter>
  );
}

function Users() {
  /* All <Route path> and <Link to> values in this
     component will automatically be "mounted" at the
     /users URL prefix since the <Users> element is only
     ever rendered when the URL matches /users/*
  */
  return (
    <div>
      <nav>
        <Link to="me">My Profile</Link>
      </nav>

      <Routes>
        <Route path="/" element={<UsersIndex />} />
        <Route path=":id" element={<UserProfile />} />
        <Route path="me" element={<OwnUserProfile />} />
      </Routes>
    </div>
  );
}
```

- Unlike the <Switch> API in v5, all <Route path> and <Link to> values under v6's <Routes> element are automatically relative to the parent route that rendered them. (this way you avoid having to manually interpolate match.path and match.url anymore)
- all <Route> paths **match exactly by default**
- If you want to match more of the URL because you have child routes (see the <Routes> defined in the Users component above), use a trailing * as in <Route path="users/*">.

Or gather them all together

Either Using JSX Routes component	Or Using object based routes (useRoutes hook)

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

function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="users" element={<Users />}>
          <Route path="/" element={<UsersIndex />} />
          <Route path=":id" element={<UserProfile />} />
          <Route path="me" element={<OwnUserProfile />} />
        </Route>
      </Routes>
    </BrowserRouter>
  );
}

function Users() {
  return (
    <div>
      <nav>
        <Link to="me">My Profile</Link>
      </nav>

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

```jsx
import {
  BrowserRouter,
  Link,
  Outlet,
  useRoutes
} from 'react-router-dom';

function App() {
  // We removed the <BrowserRouter> element from App because the
  // useRoutes hook needs to be in the context of a <BrowserRouter>
  // element. This is a common pattern with React Router apps that
  // are rendered in different environments. To render an <App>,
  // you'll need to wrap it in your own <BrowserRouter> element.
  let element = useRoutes([
    // A route object has the same properties as a <Route>
    // element. The `children` is just an array of child routes.
    { path: '/', element: <Home /> },
    {
      path: 'users',
      element: <Users />,
      children: [
        { path: '/', element: <UsersIndex /> },
        { path: ':id', element: <UserProfile /> },
        { path: 'me', element: <OwnUserProfile /> },
      ]
    }
  ]);

  return element;
}

function Users() {
  return (
    <div>
      <nav>
        <Link to="me">My Profile</Link>
      </nav>

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

In the gather all with JSX case you don't need * in users path. We used an **<Outlet>** element as a placeholder. **An <Outlet> in this case is how the Users component renders its child routes** (the three Routes "/", "me" and ":id"). So the <Outlet> will render either a <UsersIndex>, a <UserProfile> or <OwnUserProfile> element respectively depending on the current location. In React Router v5, nested routes have to be defined explicitly.	The ***useRoutes*** hook accepts a (possibly nested) array of JavaScript objects that represent the available routes in your app. Each route has a path, element, and (optionally) children, which is just another array of routes. The object-based route configuration may look familiar if you were using the react-router-config package in v5. Notice that the Users component here acts like a layout for its children components since they are rendered inside it (in their Outlet element).

```jsx
<Route
  path="products/:id"
  element={<ProductPage />}
/>
```

```jsx
const { id } = useParams();
const product = products.find(
  (p) => p.id === id
);
```

The ***useParams*** hook can then be used to retrieve a parameter value from the path

Splitting routes vs having them in one place

In a large app it's nice to be able to spread out your route definitions across multiple <Routes> elements so you can do code splitting more easily. But in a smaller app, or with nested components that are closely related, you may want to just see all of your routes in one place. This can help a lot with code readability.

navigate imperatively

Version 6 is a great chance for us to get the router all ready for the future of React: suspense. Instead of giving you access to the history instance directly (usage of which would introduce subtle bugs in a suspense-enabled app), v6 gives you a useNavigate hook. This is useful any time you need to navigate imperatively, e.g. after the user submits a form or clicks on a button.

```
import React from 'react';
import { useNavigate } from 'react-router-dom';

function App() {
  let navigate = useNavigate();
  let [error, setError] = React.useState(null);

  async function handleSubmit(event) {
    event.preventDefault();
    let result = await submitForm(event.target);
    if (result.error) {
      setError(result.error);
    } else {
      navigate('success');
    }
  }

  return (
    <form onSubmit={handleSubmit}>
      // ...
    </form>
  );
}
```

useNavigate hook
If you need to do a replace instead of a push, use navigate('success', { replace: true }). If you need state, use navigate('success', { state }).

useNavigate is used instead of the useHistory hook and its useHistory.push() method.

Queryset parameters

let [searchParams, setSearchParams] = **useSearchParams**();

Use the useSearchParams to get and set queryset parameters

Lazy loading of component in routes

Login example

The Redirect component and Protected routes

The rationale of having a protected route is that when a user tries to access part of the application without logging in, they are redirected to the login page to sign into the application.

```
<Redirect
  to={{pathname: '/login', state: {from:props.location}}}
/>
```

For this redirect to work as intended, the react-router package provides a <Redirect/> component to serve this purpose. This component has a to prop which is passed to it in form of an object containing the pathname and state as shown below.

Here, the Redirect component replaces the current location in the stack with the pathname provided in the object (here /login) and then stores the location that the user was attempting to visit, in the state property.

Important

The value in state can be accessed from within the Login component using *this.props.location.state* since the location object is { pathname, search, hash, state }. This means that you can access the state of a component's location, from its location object.

Custom routes

```jsx
import React from 'react';
import { Route, Redirect } from "react-router-dom";

const PrivateRoute = ({component: Component, isAuthenticated, ...rest}) => (
  <Route {...rest} render={props => (
    isAuthenticated
      ?
      (<Component {...props}/>)
      :
      (<Redirect to={{pathname: '/login', state: {from: props.location}}}/>)
  )}/>
);
```

```jsx
<Switch><Route exact path="/" component={Home}/>
 <Route path="/items" component={Items}/>
 <Route path="/category" component={Category}/>
 <Route path="/login" component={Login}/>}/>
 <PrivateRoute
   path="/admin"
   component={Admin}
   isAuthenticated={fakeAuth.isAuthenticated}
  />
</Switch>
```

We create a custom route called PrivateRoute and use it in our Switch.

We destructure the props within the argument list and rename component to Component. We use the Route component by passing it the ...rest and render props. Within the render prop, we write logic that determines whether to render a component and which one to render if the user is signed in. Otherwise, the user is redirected to the login page.

```jsx
import React from 'react';
import {Redirect} from 'react-router-dom';

class Login extends React.Component {
  state = {
    redirectToReferrer: false
  };

  login = () => {
    fakeAuth.authenticate(() => {
      this.setState({
        redirectToReferrer: true
      })
    })
  };

  render() {
    const { from } = this.props.location.state || {from: {pathname: '/'}};
    const { redirectToReferrer } = this.state;

    if (redirectToReferrer) {
      return (
        <Redirect to={from}/>
      )
    }

    return (
      <div>
        <p> You must log in to view the content at {from.pathname} </p>
        <button onClick={this.login}> Log in </button>
      </div>
    )
  }
}

/* A fake authentication function */
export const fakeAuth = {
  isAuthenticated: false,
  authenticate(cb) {
    this.isAuthenticated = true;
    setTimeout(cb, 100)
  },
};

export default Login
```

The login component (with a fake authentication function).

Have in mind that with *this.props.location.state* it gets the state from the Redirect component.

When user logins we set the isAuthenticated property of the fakeAuth object to true. The fakeAuth object is exported so it can be accessed by other modules which by checking its isAuthenticated property detect if the user is logged in or not.

The redirectToReferrer state property is set to true when the user is signed in. This triggers a redirect to the route they had intended to visit, or to the '/' path in case they navigated directly to the login route.

Are the location and match objects added as props by default?

this.props.location

this.props.match

Misc

No match component

```jsx
<Switch>
  <Route path="/" exact component={Home}/>
  <Redirect from="/old-match" to="/will-match"/>
  <Route path="/will-match" component={WillMatch}/>
  <Route component={NoMatch}/>
</Switch>
```

Always have a no match route

Context

Context provides a way to pass data through the component tree without having to pass props down manually at every level. Whenever the value of the context changes, the subscribed components are rendered. Context is designed to share data that can be considered "global" for a tree of React components, such as the **current authenticated user**, **theme**, or **preferred language**.

Note: Context is primarily used when some data needs to be accessible by many components at different nesting levels. Apply it sparingly because it makes component reuse more difficult. If you only want to avoid passing some props through many levels, component composition is often a simpler solution than context.

Instead of passing down props

```
class App extends React.Component {
  render() {
    return <Toolbar theme="dark" />;
  }
}

function Toolbar(props) {
  // The Toolbar component must take an extra "theme" prop
  // and pass it to the ThemedButton. This can become painful
  // if every single button in the app needs to know the theme
  // because it would have to be passed through all components.
  return (
    <div>
      <ThemedButton theme={props.theme} />
    </div>
  );
}

class ThemedButton extends React.Component {
  render() {
    return <Button theme={this.props.theme} />;
  }
}
```

You can use Context

```
// Context lets us pass a value deep into the component tree
// without explicitly threading it through every component.
// Create a context for the current theme (with "light" as the default).
const ThemeContext = React.createContext('light');

class App extends React.Component {
  render() {
    // Use a Provider to pass the current theme to the tree below.
    // Any component can read it, no matter how deep it is.
    // In this example, we're passing "dark" as the current value.
    return (
      <ThemeContext.Provider value="dark">
        <Toolbar />
      </ThemeContext.Provider>
    );
  }
}

// A component in the middle doesn't have to
// pass the theme down explicitly anymore.
function Toolbar() {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}

class ThemedButton extends React.Component {
  // Assign a contextType to read the current theme context.
  // React will find the closest theme Provider above and use its value.
  // In this example, the current theme is "dark".
  static contextType = ThemeContext;
  render() {
    return <Button theme={this.context} />;
  }
}
```

Providers

Every Context object (like the ThemeContext in the example above) comes with a Provider React component that allows consuming components (the descendants of this Provider) to subscribe to context changes. Accepts a value prop to be passed to consuming components that are descendants of this Provider. One Provider can be connected to many consumers. Providers can be nested to override values deeper within the tree. All consumers that are descendants of a Provider will re-render whenever the Provider's value prop changes.

useContext hook

Accepts a context object (the value returned from React.createContext) and returns the current context value for that context. The current context value is determined by the value prop of the nearest <MyContext.Provider> above the calling component in the tree.

When the nearest <MyContext.Provider> above the component updates, this Hook will trigger a rerender with the latest context value passed to that MyContext provider

The context value is passed to the children components so why you need this hook? The hook is useful if you want to get the value of the context provider and use it in javascript for any reason, for a function that the child component uses for example.

You can build your own Providers, see Devias kit pro, contexts/SettingsContext. See also the use of the useContext hook.

```jsx
import React from 'react';
import { useFormik } from 'formik';

// Create empty context
const FormikContext = React.createContext({});

// Place all of what's returned by useFormik onto context
export const Formik = ({ children, ...props }) => {
  const formikStateAndHelpers = useFormik(props);
  return (
    <FormikContext.Provider value={formikStateAndHelpers}>
      {typeof children === 'function'
        ? children(formikStateAndHelpers)
        : children}
    </FormikContext.Provider>
  );
};
```

Hooks

Hooks are a new addition in React 16.8. They let you use state and other React features without writing a class. Hooks are functions that let you “hook into” React state and lifecycle features from function components. Hooks don’t work inside classes — they let you use React without classes. React provides a few built-in Hooks like useState. You can also create your own Hooks to reuse stateful behavior between different components. Custom hooks are to replace the need to use Higher order components to share logic between components.

It takes a bit of a mindshift to start “thinking in Hooks”. We intend for Hooks to cover all existing use cases for classes, but we will keep supporting class components for the foreseeable future. You don’t have to rewrite your whole app using hooks. You can just start using them for your new components.

Hook rules

- Only call Hooks at the top level. Don’t call Hooks inside loops, conditions, or nested functions.
- Hooks are only called from React function components (or custom Hooks — which are also only called from React components). Don’t call Hooks from regular JavaScript functions.

Hooks “hook” into react lifecycle features

Whenever you want to use withRepos like functionality (HOC) where the wrapped components need to have access to the most recent data from the server (here the repos) use a custom hook instead of the with Repos component that receives the new repos. See custom hooks later.

Collection of React Hooks

There is a collection of community created custom hooks that you can use in your applications https://react-hooks.org/

Some of the reasons they implemented hooks

https://www.youtube.com/watch?v=eX_L39UvZes

The reasons in a nutshell.

- To avoid classes we have to find a way to use local state in our components. (using the `useState` hook)
- We need to find a way to replace lifecycle methods (using the `useEffect` hook)
- We need to find a way to share non visual (stateful) logic between components without using High order components (using `custom hooks`).

It's hard to reuse stateful logic between components

```
export default withHover(
  withTheme(
    withAuth(
      withRepos(Profile)
    )
  )
)
```

```
<WithHover>
  <WithTheme hovering={false}>
    <WithAuth hovering={false} theme='dark'>
      <WithRepos hovering={false} theme='dark' authed={true}>
        <Profile
          id='JavaScript'
          loading{true}
          repos={[]}
          authed={true}
          theme='dark'
          hovering={false}
        />
      </WithRepos>
    </WithAuth>
  </WithTheme>
</WithHover>
```

Prior to hooks you had to use patterns like higher order components or render props to do this. This could lead to a wrapper hell. With Hooks, you can extract stateful logic from a component so it can be tested independently and reused. Hooks allow you to reuse stateful logic without changing your component hierarchy. This makes it easy to share Hooks among many components or with the community

Complex components become hard to understand

For example, components might perform some data fetching in componentDidMount and componentDidUpdate. However, the same componentDidMount method might also contain some unrelated logic that sets up event listeners, with cleanup performed in componentWillUnmount. Mutually related code that changes together gets split apart, but completely unrelated code ends up combined in a single method. This makes it too easy to introduce bugs and inconsistencies. In many cases it's not possible to break these components into smaller ones because the stateful logic is all over the place. It's also difficult to test them. This is one of the reasons many people prefer to combine React with a separate state management library. However, that often introduces too much abstraction, requires you to jump between different files, and makes reusing components more difficult.

To solve this, Hooks let you split one component into smaller functions based on what pieces are related (such as setting up a subscription or fetching data), rather than forcing a split based on lifecycle methods.

Classes confuse both people and machines

Some of the problems: understand how this works in javascript. When to use function and when class based components. Ahead of time compilation of components is more difficult with classes? More difficult to minimize. More difficult to stay in the optimized way.

Hooks let you use more of React's features without classes.

Hooks

useState

```
const [ loading, setLoading ] = React.useState(true)
```

It gets a single argument which is the initial state value. It outputs an array of two values, the first one is the state value and the second one a function used to modify the state.

React intentionally "waits" until all components call setState() in their event handlers before starting to re-render

Calls to setState are asynchronous (when is called inside event handlers - functions called on browser events for example onClick functions) - don't rely on this.state to reflect the new value immediately after calling setState. React "flushes" the state updates at the end of the browser event. This ensures, for example, that if both Parent and Child call setState during a click event, Child isn't re-rendered twice. So, React intentionally "waits" until all components call setState() in their event handlers before starting to re-render.

useEffect

The problem with the lifecycle way of thinking is that it forces us to sprinkle related code into different places (into different lifecycle methods). instead of lifecycle methods, we now have to think in terms of synchronization, which is actually what we were trying to achieve using the lifecycle methods. Synchronization between things that lie outside of react, for example fetch Api calls, with react things like state of components and UI.

```
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  // Similar to componentDidMount and componentDidUpdate:
  useEffect(() => {
    // Update the document title using the browser API
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

```
import React, { useState, useEffect } from 'react';

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    // Specify how to clean up after this effect:
    return function cleanup() {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

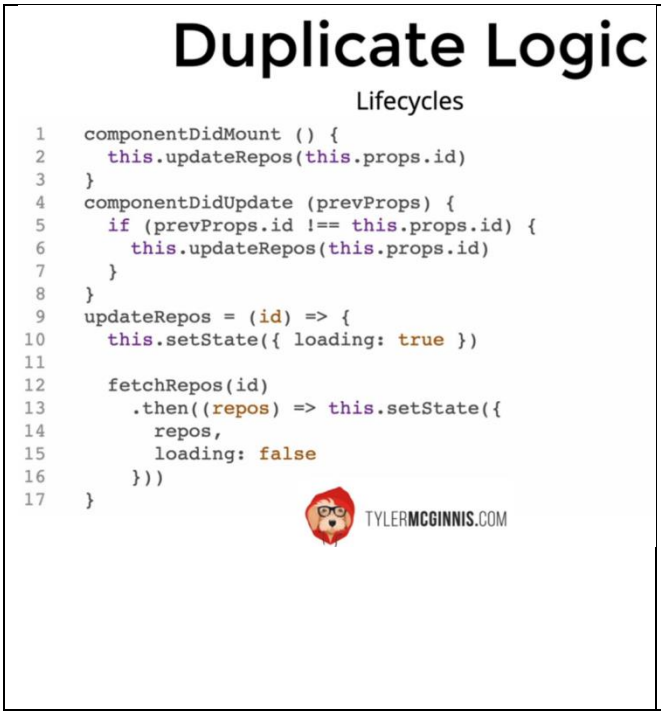
useEffect let you use do this. It let's you use side effects in function react components. It takes two arguments. The first is the function to call actually what side effect to call, and the second one is an array (called dependencies) defining when to run this function. After a rerender of the component, the effect will run only if a dependency has changed. Notice that the effect will not run if a dependency change. It only runs after a rerender. You are just telling React to skip applying an effect if certain values haven't changed between re-renders.

By default (without second argument), React runs the effects after every render — including the first render so its similar to DidMount and DidUpdate. With values as second argument, the effect doesn't run on every render, but only when these values change.
You can also write clean up code for an effect function. The cleanup code is the code that is returned from the effect function. This function will be executed **when the component unmounts** and **before re-running the effect**.

When you call useEffect, you're telling React to run your "effect" function after flushing changes to the DOM.

So in total, useEffect serves the same purpose as componentDidMount, componentDidUpdate, componentWillUnmout in React classes, but unified into a single API.

So we actually say, whenever the id changes make a fetch call to get the repos. Using lifecycle methods we had to fetch repos in didMount and didUpdate.



The array argument is used to avoid rerunning the effect on every re-render. It will only run if the re-render is done with values (defined in the array) different from the previous values.

Note: If you want to run an effect and clean it up only once (on mount and unmount), you can pass an empty array ([]) as a second argument. This tells React that your effect doesn't depend on any values from props or state, so it never needs to re-run. It will only run on mount and the clean up function on the unmount. This isn't handled as a special case — it follows directly from how the dependencies array always works.

Use effect Tips

`useEffect(() => {` `  // Your code here` `}, []);`	Runs only after the component has been mounted (initial render). It is the equivalent of componentDidMount
`useEffect(() => {` `  // Your code here` `});`	Runs after every render.
`React.useEffect(() => {` `  // this side effect wi` `}, [value1]);`	Be careful with your dependencies array. If you use the dependencies array, make sure it includes all values from the component scope (such as props and state) that change over time and that **are used by the effect**. (this is important. The dependencies must be variables that are used by the useEffect function). It's very common to forget a value or to think that you don't need it in the array. If you do that, you will produce bugs, because your code will reference stale values from previous renders. NOTICE An effect with dependencies will be executed too, after the component is mounted (since the previously stored values for the dependencies are undefined or something like that)

Equivalents

```
componentDidMount() {
  window.addEventListener('mousemove', () => {})
}

componentWillUnmount() {
  window.removeEventListener('mousemove', () => {})
}
```

```
useEffect(() => {
  window.addEventListener('mousemove', () => {});

  // returned function will be called on component unmount
  return () => {
    window.removeEventListener('mousemove', () => {})
  }
}, [])
```

Run something before render

```
const Component = () => {
   useMemo(() => {
     // componentWillMount events
   },[]);
   useEffect(() => {
     // componentDidMount events
     return () => {
       // componentWillUnmount events
     }
   }, []);
};
```

This works because useMemo doesn't require to return a value and you don't have to actually use it as anything, but since it memorizes a value based on dependencies which will only run once ("[]") and its on top of our component it runs once when the component mounts before anything else.

Notice

```
window.data = {}

useEffect(() => {
// do something
}, [window.data])
```

No, that won't work. Effect could only be triggered when component is rerendered and dependencies change (dependencies are variables that the useEffect function uses). Changing global variables won't cause a rerender, so the effect won't run.

More than one side effects

If you use more than one side effects (many useEffect calls) React will apply every effect used by the component, in the order they were specified

Custom hooks

Custom hooks are like components, but not tied to the UI. Like components, they have state and can respond to react lifecycle methods, but they don't render anything. They just return a value that can be used in other parts of the code (in other hooks or components). The returned value is updated whenever the custom hook is updated (it's state changes).

A custom hook can be thought of as a function which is executed from within the functional component and effectively the hooks that are present in the custom hook are transferred on to the component. So any change that would normally cause the component to re-render if the code within the custom hook was directly written within functional component will cause a re-render. So if the state of a custom hook changes, the host component will be re-rendered.

it doesn't really matter, whether the state change in hooks is "internal" or not. Every state change in a hook, whether it affects its return value or not, will cause the "host" component to re-render. And of course exactly the same story with chaining hooks: if a hook's state changes, it will cause its "host" hook change as well, which will propagate up through the whole chain of hooks until it reaches the "host" component and re-renders it. This might have a performance hit on your app.

Sharing stateful non visual logic. In react is kind of difficult to share stateful non visual logic (you need higher order components or render props) because react by design, ties its components to visual logic ie to UI.

Custom Hooks are more of a convention than a feature. If a function's name starts with "use" and it calls other Hooks, we say it is a custom Hook. The useSomething naming convention is how our linter plugin is able to find bugs in the code using Hooks. Just like in a component, make sure to only call other Hooks unconditionally at the top level of your custom Hook.

Custom Hooks offer the flexibility of sharing logic that wasn't possible in React components before. You can write custom Hooks that cover a wide range of use cases like form handling, animation, declarative subscriptions, timers, and probably many more we haven't considered.

This is the legacy way of sharing non visual logic, using higher order components.

```
function withRepos (Component) {
  return class WithRepos extends React.Component {
    state = {
      repos: [],
      loading: true
    }
    componentDidMount () {
      this.updateRepos(this.props.id)
    }
    componentDidUpdate (prevProps) {
      if (prevProps.id !== this.props.id) {
        this.updateRepos(this.props.id)
      }
    }
    updateRepos = (id) => {
      this.setState({ loading: true })

      fetchRepos(id)
        .then((repos) => this.setState({
          repos,
          loading: false
        }))
    }
    render () {
      return (
        <Component
          {...this.props}
          {...this.state}
        />
      )
    }
  }
}
```

Then:

```
// ReposGrid.js
function ReposGrid ({ loading, repos }) {
  ...
}
```

There is no special hook for that. Instead you can create your own hook that is decoupled from any UI.

```
function useRepos (id) {
  const [ repos, setRepos ] = React.useState([])
  const [ loading, setLoading ] = React.useState(true)

  React.useEffect(() => {
    setLoading(true)

    fetchRepos(id)
      .then((repos) => {
        setRepos(repos)
        setLoading(false)
      })
  }, [id])

  return [ loading, repos ]
}
```

Then:

```
function ReposGrid ({ id }) {
  const [ loading, repos ] = useRepos(id)

  ...
}
```

Any logic that is related to fetching repos is isolated inside this custom hook. So whenever we need repos inside any component we can consume our custom useRepos hook.

So why not just use a regular function that fetches the repos, import it in your react components and use it even in the class based components and avoid using HOCs? What makes the custom hook different? The difference is the fact that **the custom hook is a react thing, it hooks into react lifecycle features** (For example it is automatically updated when the id changes in this example). It has a state, it uses other hooks which make the custom hook to be automatically updated whenever the id changes. The modifications are then rippled to the components that consume the data output from the custom hook. **It is like a component but not tied with UI**. It doesn't render anything. It just holds non-UI stateful logic that needs to be shared with many components.

useSelector hook

Can use it to manage local state in a redux like way (with actions and reducers). instead of changing the local state with the setState you change it by dispatching actions.

Other useful hooks

useInterval

`` useInterval(() => { const newDataset = generateDataset() setDataset(newDataset) }, 2000) ```	

Suspense

Suspense lets your components "wait" for something before they can render. It's a mechanism for data fetching libraries to communicate to React that the data a component is reading is not ready yet. So until the data is fetched you can show something else.

It is also good to note that Suspense is not a data fetching mechanism but rather a way to delay the rendering of components while you wait for unavailable data. Dan Abramov gave a great demo on how Suspense works check it out https://codesandbox.io/s/frosty-hermann-bztrp?file=/src/index.js .

Notice: Suspense is an experimental feature as of late 2020

A data fetching library (like axios) must support Suspense. (probably suspense needs some specific values returned when you make a call for data and the data isn't available yet)

Version 18 notes

In React 18, you can start using Suspense for data fetching in opinionated frameworks like Relay, Next.js, Hydrogen, or Remix. Ad hoc data fetching with Suspense is technically possible, but still not recommended as a general strategy.

In the future, we may expose additional primitives that could make it easier to access your data with Suspense, perhaps without the use of an opinionated framework. However, Suspense works best when it's deeply integrated into your application's architecture: your router, your data layer, and your server rendering environment. So even long term, we expect that libraries and frameworks will play a crucial role in the React ecosystem.

As in previous versions of React, you can also use Suspense for code splitting on the client with React.lazy. But our vision for Suspense has always been about much more than loading code — the goal is to extend support for Suspense so that eventually, the same declarative Suspense fallback can handle any asynchronous operation (loading code, data, images, etc).

Currently the only built-in use of Suspense is to lazy load components (code splitting)

`` const ProfilePage = React.lazy(() => import('./ProfilePage')); // Lazy-loaded // Show a spinner while the profile is loading <Suspense fallback={<Spinner />}> <ProfilePage /> </Suspense> ```	React 16.6 added a <Suspense> component that lets you "wait" for some code to load (here lazy loading a component) and declaratively specify a loading state (like a spinner) while we're waiting.

```jsx
const resource = fetchProfileData();

function ProfilePage() {
  return (
    <Suspense fallback={<h1>Loading profile...</h1>}>
      <ProfileDetails />
      <Suspense fallback={<h1>Loading posts...</h1>}>
        <ProfileTimeline />
      </Suspense>
    </Suspense>
  );
}

function ProfileDetails() {
  // Try to read user info, although it might not have loaded yet
  const user = resource.user.read();
  return <h1>{user.name}</h1>;
}

function ProfileTimeline() {
  // Try to read posts, although they might not have loaded yet
  const posts = resource.posts.read();
  return (
    <ul>
      {posts.map(post => (
        <li key={post.id}>{post.text}</li>
      ))}
    </ul>
  );
}
```

Suspense for Data Fetching is a new feature that lets you also use <Suspense> to declaratively "wait" for anything else, including data. Suspense lets your components "wait" for something before they can render. In this example, two components wait for an asynchronous API call to fetch some data

Suspense is not a data fetching library. It's a mechanism for data fetching libraries to communicate to React that the data a component is reading is not ready yet. React can then wait for it to be ready and update the UI.
(At Facebook, we use Relay and its new Suspense integration. We expect that other libraries like Apollo can provide similar integrations.)

In the long term, we intend Suspense to become the primary way to read asynchronous data from components — no matter where that data is coming from.

Code splitting

The best way to introduce code-splitting into your app is through the ***dynamic import()* syntax**.

Before:

```jsx
import { add } from './math';

console.log(add(16, 26));
```

After:

```jsx
import("./math").then(math => {
  console.log(math.add(16, 26));
});
```

When Webpack comes across this syntax, it automatically starts code-splitting your app. If you're using Create React App, this is already configured for you and you can start using it immediately. It's also supported out of the box in Next.js. otherwise you have to properly configure webpack.

react.lazy()

The React.lazy function lets you render a dynamic import as a regular component. Does this mean that there will be a distinct chunk for each lazy loaded component?

Before:

```jsx
import OtherComponent from './OtherComponent';
```

After:

```jsx
const OtherComponent = React.lazy(() => import('./OtherComponent'));
```

This will automatically load the bundle containing the OtherComponent when this component (the one that contains this code) is first rendered.

```jsx
import React, { Suspense } from 'react';

const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <OtherComponent />
      </Suspense>
    </div>
  );
}
```

React lazy works for specific cases:
React.lazy takes a function that must call a dynamic import(). This import must return a Promise which resolves to a module with a default export containing a React component. The lazy component should then be rendered inside a **Suspense** component, which allows us to show some fallback content (such as a loading indicator) while we're waiting for the lazy component to load.

Error boundaries around suspense

If the other module fails to load (for example, due to network failure), it will trigger an error. You can handle these errors to show a nice user experience and manage recovery with Error Boundaries. Once you've created your Error Boundary, you can use it anywhere above your lazy components to display an error state when there's a network error.

Named exports

```jsx
// ManyComponents.js
export const MyComponent = /* ... */;
export const MyUnusedComponent = /* ... */;
```

```jsx
// MyComponent.js
export { MyComponent as default } from "./ManyComponents.js";
```

```jsx
// MyApp.js
import React, { lazy } from 'react';
const MyComponent = lazy(() => import("./MyComponent.js"));
```

React.lazy currently only supports default exports. If the module you want to import uses named exports, you can create an intermediate module that reexports it as the default. This ensures that tree shaking keeps working and that you don't pull in unused components.

Route based code splitting

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

const Home = lazy(() => import('./routes/Home'));
const About = lazy(() => import('./routes/About'));

const App = () => (
  <Router>
    <Suspense fallback={<div>Loading...</div>}>
      <Switch>
        <Route exact path="/" component={Home}/>
        <Route path="/about" component={About}/>
      </Switch>
    </Suspense>
  </Router>
);
```

Deciding where in your app to introduce code splitting can be a bit tricky. You want to make sure you choose places that will split bundles evenly, but won't disrupt the user experience. A good place to start is with routes.

Here's an example of how to setup route-based code splitting into your app using libraries like React Router with React.lazy.

Caching in react

Misc

Store and retrieve from local storage

```
const defaultSettings = {
  direction: 'ltr',
  responsiveFontSizes: true,
  theme: THEMES.ONE_DARK
};

export const restoreSettings = () => {
  let settings = null;

  try {
    const storedData = window.localStorage.getItem( key: 'settings');

    if (storedData) {
      settings = JSON.parse(storedData);
    }
  } catch (err) {
    console.error(err);
    // If stored data is not a strigified JSON this will fail,
    // that's why we catch the error
  }

  return settings;
};

export const storeSettings = (settings) => {
  window.localStorage.setItem('settings', JSON.stringify(settings));
};
```

A typical process

Store and retrieve from a cookie

```
import Cookies from 'js-cookie';

const useStyles = makeStyles( style: (theme : Theme ) => ({...}));

const SettingsNotification = () => {
  const classes = useStyles();
  const [isOpen, setOpen] = useState( initialState: false);
  const { saveSettings } = useSettings();

  const handleSwitch = () => {
    saveSettings({ theme: THEMES.LIGHT });
    Cookies.set('settingsUpdated', 'true');
    setOpen( value: false);
  };

  const handleClose = () => {
    Cookies.set('settingsUpdated', 'true');
    setOpen( value: false);
  };
```

Using the js-cookie library

Cookies.get

Styling components

CSS in JS

"CSS-in-JS" refers to a pattern where CSS is composed using JavaScript instead of defined in external files. Note that this functionality is not a part of React, but provided by third-party libraries.

CSS uses a global namespace for CSS Selectors that can easily result in style conflicts throughout your application when building an application using modern web components. You can avoid this problem by nesting CSS selectors or use a styling convention like BEM but this becomes complicated quickly and won't scale. **CSS-in-JS avoids these problems entirely by generating unique class names when styles are converted to CSS**. This allows you to think about styles on a component level without worrying about styles defined elsewhere. (A typical example of an automatically created class name is "makestyles-root-13")

- **JSS**

JSS is a set of libraries for writing CSS in JavaScript. (JSS is one CSS in JS solution). They address a wide spectrum of issues. The most significant features are class names scoping, critical CSS extraction, significantly improved maintenance, code reuse and sharing, theming, co-location and state-driven styles.

The general process goes like this:

1. Declaration: Styles are described by the user in JavaScript. By default we use JSON Syntax.
2. Processing: Styles are processed by JSS plugins. Plugins do vendor prefixing, implement syntactic sugar for user styles and can be made to do any other transformations, similar to PostCSS.
3. Injection: Once you call the .attach method, styles are compiled to a CSS string and injected into the DOM using a style element (this step is done by the core jss library)

```
import jss from "jss";
import preset from "jss-preset-default";

// One-time setup.
jss.setup(preset());

const styles = {
  button: {
    color: "red",
  },
};

// Compile and render the styles.
const { classes } = jss.createStyleSheet(styles).attach();

document.body.innerHTML = `
  <button class="${classes.button}">
    My Button
  </button>
`;
```

The essential libraries in JSS are core, React-JSS, and Styled-JSS. Low level and library-agnostic, the core is responsible for compilation and rendering (injection) of a stylesheet.

Jss library
Generates the CSS in runtime, but you can generate it in the server if you want (server side rendering). notice that in the server side rendering case, only the styles used by the current page are generated and send
The inputs are objects (or simple template strings) with keys as the class names and the outputs are the generated CSS and objects defining the generated scoped class names.

React-jss library
React-jss library injects the component styles in the DOM once the component is mounted and removes the injected styles once the component is unmounted.

- **Styled components**

Another CSS in JS solution.

Styled primitive or styled component is a component which has initial styles applied when created. There is no need to provide class names when you use it. It has been very actively promoted by the Styled Components library and is worth looking into as an alternative to other interfaces.

Styled components is a library that enables the CSS in JS concept. There are alternative libraries for achieving the same things.

styled-components utilizes *tagged template literals* to style your components. You don't have to write css in separate .css files. You write css directly inside the components code.

`` // Create a Title component that'll render an <h1> tag with some styles const Title = styled.h1`   font-size: 1.5em;   text-align: center;   color: palevioletred; `; // Create a Wrapper component that'll render a <section> tag with some styles const Wrapper = styled.section`   padding: 4em;   background: papayawhip; `; // Use Title and Wrapper like any other React component – except they're styled! render(<Wrapper> <Title> Hello World! </Title> </Wrapper>); ```	Hello World!

Misc

classnames

render() { let className = 'menu'; if (this.props.isActive) { className += ' menu-active'; } return Menu }	If you often find yourself writing code like this, ***classnames package*** can simplify it.

You can use Material-UI's styling solution in your app, whether or not you are using Material-UI components. It is a CSS in JS solution. It uses JSS at its core.

Can I use inline styles? Yes, see the docs on styling here.

Are inline styles bad? CSS classes are generally better for performance than inline styles.

React does not have an opinion about how styles are defined; if in doubt, a good starting point is to define your styles in a separate *.css file as usual and refer to them using className.

Inline styling: style attribute is most often used in React applications to add dynamically-computed styles at render time. In most cases, className should be used to reference classes defined in an external CSS stylesheet.

Change css properties based on screen size

```jsx
const useStyles = makeStyles( style: (theme : Theme ) => ({
  root: {
    backgroundColor: theme.palette.background.dark,
    minHeight: '100%',
    paddingTop: theme.spacing(3),
    paddingBottom: theme.spacing(3)
  },
  action: {
    marginBottom: theme.spacing(1),
    '& + &': {
      marginLeft: theme.spacing(1)
    },
    minWidth: 100,

    '@media (max-width:420px)': {
      minWidth: 70,
      paddingLeft: theme.spacing(0.5),
      paddingRight: theme.spacing(0.5)
    },
    // another way to control css properties based on size, is by using the built in theme's breakpoints
    // material's UI xs breakpoint is 0-600 px so it doesn't fit for this case.
    // [theme.breakpoints.down('xs')]: {
    //   minWidth: 70,
    // },

    // from the element with class action, select the children elements with class SvgIcon
    '& .SvgIcon': {
      fontSize: '1.2rem',
      '@media (max-width:420px)': {
        fontSize: '0.8rem',
      }
    }
  },
}));
```

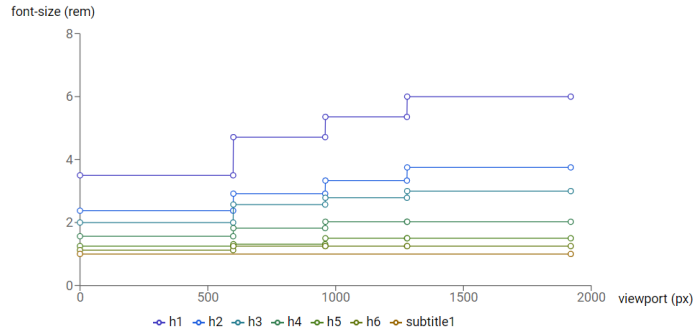
You can change the css properties assigned to a selection either using the build in material's UI breakpoints (xs, sm, md, lg) or by using custom media queries if you want to use specific screen sizes.

Notice also how you select children of a specific class. Using the
"& selector": {

}

& + & is not CSS but some file meant to be compiled to CSS. In SCSS and Less, the & is just a repetition of the enclosing selector. So here it means action + action (the first action element placed immediately after an action element)

Responsivefontsizes utility



It automatically creates responsive font sizes for the various typography variants (h1 to h6, subtitle1) so that you don't have to manually write something like this for every variant.

```jsx
const theme = createTheme();

theme.typography.h3 = {
  fontSize: '1.2rem',
  '@media (min-width:600px)': {
    fontSize: '1.5rem',
  },
  [theme.breakpoints.up('md')]: {
    fontSize: '2.4rem',
  },
};
```

You just use the h1 for example, and this h1 will be responsive.

```
let theme = createMuiTheme();
theme = responsiveFontSizes(theme);

const heading = "OUR WORK";
const text =
  "We work with farmers on paddy fields to understand

function App() {
  return (
    <div className="App">
      <MuiThemeProvider theme={theme}>
        <Typography variant="h1" gutterBottom>
          {heading}
        </Typography>
        <Typography variant="subtitle1" gutterBottom>
          {text}
        </Typography>
      </MuiThemeProvider>
    </div>
  );
}
```

Define a custom typography object to use in theme creation

typography.js | MinHeapAnimation.js | inde

```
export default {
  h1: {
    fontWeight: 500,
    fontSize: 35,
    letterSpacing: '-0.24px'
  },
  h3: {
    fontSize: '0.85rem',
    '@media (max-width:420px)': {
      fontSize: '0.6rem',
    },
  },

```

Notice that if you have modified the default typography and used this in the createMuiTheme function to create your custom theme then you have to define media queries (or use the built in breakpoints)

Using css

```
import styles from './Counter.module.css';

return (
  <div>
    <div className={styles.row}>
      <button
        className={styles.button}
```

Tips

From create-react-app docs: Generally, we recommend that you don't reuse the same CSS classes across different components. For example, instead of using a .Button CSS class in <AcceptButton> and <RejectButton> components, we recommend creating a <Button> component with its own .Button styles, that both <AcceptButton> and <RejectButton> can render (but not inherit). Following this rule often makes CSS preprocessors less useful, as features like mixins and nesting are replaced by component composition. You can, however, integrate a CSS preprocessor if you find it valuable.

Notice:

Create-React-app documentation mentions that they normally recommend importing stylesheets, images, and fonts from JavaScript. Not just using a reference to an image in the public folder, but importing the image in javascript and using it. This image doesn't need to be in the public folder.

Adding images fonts and files

This is the recommended approach to load images and assets in general. Using javascript imports.

`` import React from 'react'; import logo from './logo.png'; // Tell webpack this JS file uses this image console.log(logo); // /logo.84287d09.png function Header() { // Import result is the URL of your image return ; } export default Header; ```	This ensures that when the project is built, webpack will correctly move the images into the build folder, and provide the correct paths (the value returned for an image is the path to the image)

The advantages of importing assets in javascript with webpack are:

1. Scripts and stylesheets get minified and bundled together to avoid extra network requests.
2. Missing files cause compilation errors instead of 404 errors for your users.
3. Result filenames include content hashes so you don't need to worry about browsers caching their old versions.

The alternative is to manually add them to the build using the public folder (and referencing them in javascript with the env variable PUBLIC_URL (or directly as I saw).

Testing

There are a few ways to test React components. Broadly, they divide into two categories:

1. Rendering component trees in a simplified test environment and asserting on their output.
2. Running a complete app in a realistic browser environment also known as "end-to-end" tests. These systems automate browser actions. (Cypress, pupeteer, selenium)

Testing in Test environments

The second category isn't specific to react, so for testing react components you fall into the first category. Some tools to use for such kind of testing is

- Jest

If you use Create React App, Jest is already included out of the box with useful defaults. Jest is widely compatible with React projects, supporting features like mocked modules and timers, and jsdom support (a lightweight browser implementation that runs inside Node.js).

- React Testing Library
- Mocha

If you're writing a library that tests mostly browser-specific behavior, and requires native browser behavior like layout or real inputs (instead of the simulated browser environment offered by jsdom node package used by Jest), you could use a framework like mocha.

React component test frameworks often offer you the possibility to

- **Mock a rendering surface**: Tests often run in an environment without access to a real rendering surface like a browser. For these environments, we recommend simulating a browser with jsdom, a lightweight browser implementation that runs inside Node.js.
- **Mock functions**: This is especially useful for data fetching. It is usually preferable to use "fake" data for tests to avoid the slowness and flakiness due to fetching from real API endpoints
- **Mock modules**: Some components have dependencies for modules that may not work well in test environments, or aren't essential to our tests. It can be useful to selectively mock these modules out with suitable replacements
- **Mock timers**: In testing environments, it can be helpful to mock these functions out with replacements that let you manually "advance" time. This is great for making sure your tests run fast!

End to end tests

End-to-end tests are useful for testing longer workflows, especially when they're critical to your business (such as payments or signups)

Misc

Fetching data

Suspense will be the proper solution for this, but is not yet ready and not yet implemented by most data fetching libraries (late 2020). In general there are three basic approaches for fetching data:

- **Fetch-on-render** (for example, fetch in useEffect): Start rendering components. Each of these components may trigger data fetching in their effects and lifecycle methods. This approach often leads to "waterfalls".
- **Fetch-then-render** (for example, Relay without Suspense): Start fetching all the data for the next screen as early as possible. When the data is ready, render the new screen. We can't do anything until the data arrives.
- **Render-as-you-fetch** (for example, Relay with Suspense): Start fetching all the required data for the next screen as early as possible, and start rendering the new screen immediately — before we get a network response. As data streams in, React retries rendering components that still need data until they're all ready.

Fetch on render

A common way to fetch data in React apps today is to use an effect. This means that you have to make the component capable of rendering something when the data has not been fetched yet. Usually you do this by rendering loading indicators. Notice that you can use individual loading indicators for each component that waits for its data or one loading screen for the whole page that might contain more than one components that wait for data.

Waterfall

This approach leads to the problem of waterfall. If you have one component that renders another one where both the parent and the child fetch data as a side effect after they are rendered, then the child has to wait for the parent to fetch the data in order to start fetching data itself.

Notice that these workflows can be simplified with the use of Suspense, essentially by not having to write the "loading…" logic.	The same example using Suspense.

```jsx
function ProfilePage() {
  const [user, setUser] = useState(null);

  useEffect(() => {
    fetchUser().then(u => setUser(u));
  }, []);

  if (user === null) {
    return <p>Loading profile...</p>;
  }
  return (
    <>
      <h1>{user.name}</h1>
      <ProfileTimeline />
    </>
  );
}

function ProfileTimeline() {
  const [posts, setPosts] = useState(null);

  useEffect(() => {
    fetchPosts().then(p => setPosts(p));
  }, []);

  if (posts === null) {
    return <h2>Loading posts...</h2>;
  }
  return (
    <ul>
      {posts.map(post => (
        <li key={post.id}>{post.text}</li>
      ))}
    </ul>
  );
}
```

```jsx
const ProfilePage = React.lazy(() => import('./ProfilePage')); // Lazy-loaded

// Show a spinner while the profile is loading
<Suspense fallback={<Spinner />}>
  <ProfilePage />
</Suspense>
```

```jsx
const resource = fetchProfileData();

function ProfilePage() {
  return (
    <Suspense fallback={<h1>Loading profile...</h1>}>
      <ProfileDetails />
      <Suspense fallback={<h1>Loading posts...</h1>}>
        <ProfileTimeline />
      </Suspense>
    </Suspense>
  );
}

function ProfileDetails() {
  // Try to read user info, although it might not have loaded yet
  const user = resource.user.read();
  return <h1>{user.name}</h1>;
}

function ProfileTimeline() {
  // Try to read posts, although they might not have loaded yet
  const posts = resource.posts.read();
  return (
    <ul>
      {posts.map(post => (
        <li key={post.id}>{post.text}</li>
      ))}
    </ul>
  );
}
```

Misc

Env vars

For reading env vars from files react uses the dotenv package.

Note: (in the .env file) You must create custom environment variables beginning with **REACT_APP_**. Any other variables except NODE_ENV will be ignored to avoid accidentally exposing a private key on the machine that could have the same name. Changing any environment variables will require you to restart the development server if it is running.

Files on the left have more priority than files on the right:

- npm start: .env.development.local, .env.local, .env.development, .env
- npm run build: .env.production.local, .env.local, .env.production, .env
- npm test: .env.test.local, .env.test, .env (note .env.local is missing)

Mount, Unmount and Render

Have in mind that a component will be unmounted if you change a route that doesn't render this component, so the next time you will visit the route that does use it it will be mounted again. Have also in mind that while the component is mounted it can be re-rendered (if its props or state change).

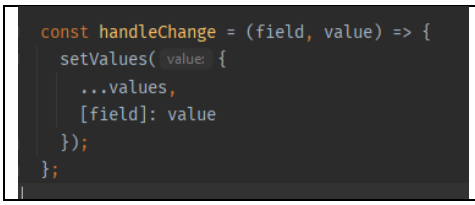
React only renders primitive data types

You can't do {algorithms_list}	A React child should be either a JSX tag, or of primitive data type (not an object).

If algorithms_list is an object. You either have to convert it to string or use one of its properties if it is of primitive data type.	

In JavaScript, a primitive (primitive value, primitive data type) is data that is not an object and has no methods. There are 7 primitive data types: string, number, bigint, boolean, undefined, symbol, and null.

Update object in state

	Values is an object like {'a':1, 'b':2, 'c':3} The purpose of the function is to update this object. You give it a key value pair like handleChange('b', 20) So it updatest he values object. first it spreads the values members and then it overrides one member with the given arguments.

FIRST

As Addy Osmani would say, good software components should be **f**ocused, **i**ndependent, **r**eusable, **s**mall and **t**estable (FIRST)

Environment variables

WARNING: Do not store any secrets (such as private API keys) in your React app! Environment variables are embedded into the build, meaning anyone can view them by inspecting your app's files.

Return null

To not render anything, just return null from the component

Curly braces

var css = { color: red } <h1 style={css}>Hello world</h1>	Curly braces in JSX: The curly braces are a special syntax to let the JSX parser know that it needs to interpret the contents in between them as JavaScript instead of a string. You need them when you want to use a JavaScript expression like a variable or a reference inside JSX. This process is generally referred to as "interpolation".

npx vs npm for avoiding global installations

Vue	React
● Css inside the component file ● State is kept in a data object which you can freely update whenever you want. ● Two way binding (an input field updates the data object (state) and the data object updates the input)	● Separate css file for each component ● State should be changed with setState so that various hooks can be triggered by it (componentWillReceiveProps, shouldComponentUpdate, componentWillUpdate, render, componentDidUpdate) ● One way biding (the state updates the input field)

create-react-app

```
npx create-react-app my-app --template [template-name]
```

You can use a template other than the default one.

npx create-react-app my-app --template redux

We recommend using the Redux templates for Create-React-App as the fastest way to create a new Redux + React project. It comes with Redux Toolkit and React-Redux already configured, using the same "counter" app example you saw in Part 1. This lets you jump right into writing your actual application code without having to add the Redux packages and set up the store.

- You can also install redux toolkit in an existing app with *npm install @reduxjs/toolkit*

```
npx create-react-app my-app --use-npm
```

When you create a new app, the CLI will use Yarn to install dependencies (when available). If you have Yarn installed, but would prefer to use npm, you can append --use-npm to the creation command.

Running the create-react-app my-app will create the following structure.

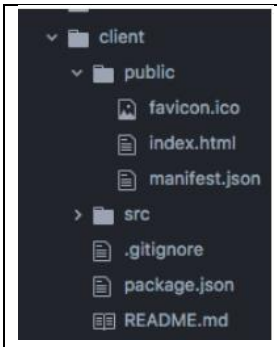
Inside the newly created project, you can run some built-in commands
cd my-app
npm start
npm test

Npm run build
Builds the app for production to the build folder. It correctly bundles React in production mode and optimizes the build for the best performance. The build is minified and the filenames include the hashes. Your app is ready to be deployed.

Public folder in create-react-app

In general the public folder is used to add an asset to your project, outside of the module system (handled by webpack). webpack does not read the public folder. If you put a file into the public folder, it will not be processed by webpack. Instead it will be copied into the build folder untouched. To reference assets in the public folder, you need to use an environment variable called PUBLIC_URL

When you run create-react-app the following files are generated

	The public folder is special for a few reasons. Perhaps the most important thing to know is that Webpack does not read the public folder; it will only read the files inside the src folder (you can overcome this with the %public_url% tag). This means that you CANNOT put JS, CSS or any other assets in the public folder, since it will not be compiled during the build process. Notice that the script loading elements <script src=".. /> will be created by the build process and added to the pages. Index.html This file holds the HTML template of our app. React will simply inject code into the element

Manifest.json

This file exists to provide app and extension info, such as name, icon and description, in json format. This comes in handy when your website behaves like an app, for instance when a user adds your website to their mobile homescreen. The manifest.json file will be compiled and added as a link tag in the head of the HTML template in index.html: <link rel="manifest" href="%PUBLIC_URL%/manifest.json">

%PUBLIC_URL% tag

During the build, assets prefixed with the %PULIC_URL% tag will be recognized and compiled. However this is NOT best practice and should be used only when necessary.

Favicon.ico

An ico file is just a file that browsers recognize is an icon, and the file contains multiple sizes and colors depths that the computer can scale accordingly.

To reference images in public folder

1. `<img src="/image.jpg" alt="image" />` You can also do 2. `<img src={process.env.PUBLIC_URL + '/yourPathHere.jpg'} />`	1. By default react will know its in public directory.
`const methodIcons = {` `'Auth0': '/static/images/auth0.svg',` `'FirebaseAuth': '/static/images/firebase.svg',` `'JWT': '/static/images/jwt.svg'` `};`	I also saw this in devias kit pro. Static is a folder inside the public folder. Actually this is the first approach where you reference something directly. Have in mind that you define the public/static folder as static files folder (a staticfiles dir in django) so that its contents are collected along with the other static files and placed in your static files service location. You define the url of this location as your-domain/static/ so any file that is referenced directly to /static/ instead of being imported in javascript, will be served by the static server.

Concurrent rendering

New in react 18

Micro frontends

Have in mind this pattern. The front end is composed of many mini frontends. Decomposing the front end monolith to individual components generated by individual back end services.

Pros:

- A micro frontend is more modular and reusable.
- A micro frontend is more scalable.
- The micro frontend is more maintainable.
- Independent and Faster development.
- Testing separate applications are easy.
- Different front-end technologies can be used for different projects(like React, Angular, Vue.js, etc).

Cons:

- Testing the entire application is not easy.
- Sharing code, state(data), etc is not easy.

`` import React from "react"; import ReactDOM from "react-dom"; import { Counter } from 'counter/Counter'; import "./index.css"; const App = () => (<div className="container"> <h1>Container App</h1> <Counter /> // Micro frontend app </div>); ReactDOM.render(<App />, document.getElementById("app")); ```	Module federation (Module Federation | webpack) Module Federation allows a JavaScript application to dynamically load code from another application and in the process, share dependencies. If an application consuming a federated module does not have a dependency needed by the federated code, Webpack will download the missing dependency from that federated build origin.

NextJS

It can render the react app in the server and serve the rendered html page. After that initial page is served, client side rendering takes over as a traditional react app. So you have server rendered content for bots and highly interactive content for users.

What Next. js does is to provide structure and better rendering features to React. As stated before, it works on top of React since it names itself as "The React Framework for Production". So, it works as an engine for React's capabilities, using many tools and resources already used by React like Redux or Hooks.

Problems with client side rendered content

- Not reliably indexed by all search engines or read by social media link bots
- Slower to first contentful paint

React + django

Architecture options

There are three main ways in which you can set up django with a front end framework.

1. Django based front end approach: React in its own "frontend" Django app: load a single HTML template and let React manage the front end (difficulty: medium)

2. The decoupled approach: Django REST as a standalone API + React as a standalone SPA (difficulty: hard, it involves JWT for authentication)
3. Mix and match: mini React apps inside Django templates (difficulty: simple, but not so maintainable in the long run)

1st vs 2nd approach

The difference between the 1st and 2nd approach is that in the 2nd one you just have a django project that doesn't serve nor a single django template neither static files. It just serves the json responses through django rest framework. The javascript code that renders the front end (which might be an html page or a mobile app or whatever else) is served by a separate service which just serves static files (the front end code). When a user visits your url the dns lookup ends up in a request to this front end service which sends the javascript code that creates the front end. This javascript code fetches data from the separate django service. One main side effect of this approach is that you can't use session based authentication. Instead you have to use token based authentication, for example JWT.

Mixed approach

Use the mixed approach when:

- the website doesn't need much Javascript
- SEO is a big concern and you can't use Node.js for Server Side Rendering

Decoupled approach

An ideal way to host React app is to serve it over a CDN like CloudFront and have it make API calls to the backend API, possibly on a different subdomain. For small apps (in terms of scope and usage), this is an overkill.

Pros:

A decoupled approach offers flexibility but imposes challenges:

- search engine optimization (SSR or die)
- authentication
- logic duplication (errors and form validation)
- more testing
- more developers

These are good signs that you will benefit from decoupling:	Use the django based approach when:
■ lot of JS-driven interactions ■ you're building a mobile app ■ you're building a dashboard for internal use	■ you're building an app-like website ■ you're ok with Session based authentication ■ there are no SEO concerns ■ you're fine with React Router

In fact keeping React closer to Django makes easier to reason about authentication. since the JavaScript bundle continues to live inside a Django template you can use Django's built-in authentication. You can exploit the Django builtin authentication for registering and logging in users. Use the good ol' Session authentication and do not worry too much about tokens and JWT.

Serve your react app from S3 (optionally with AWS Amplify)

In decoupled approach you need a web server for hosting and serving the static files of your react app. You can host and serve your app through S3. Notice that there is a managed service called AWS Amplify, one main feature of which is to do just that setting up also ci/cd automatically and it's very easy to set up. You just connect your repo through the interface and you are ready. It serves the app through a CDN.

Project Setup

Backend

- mkdir my-project-dir (rmdir /S a_folder to delete folder and contents)
- cd my-project-dir
- virtualenv venv
- venv/Scripts/activate (activate the virtual env in the my-project-dir folder)
- python --version
- pip install django
- mkdir project_src
- cd project_src
- **django-admin startproject project_name_project ./**

You could simply do django-admin startproject project_name without creating the "project_src" folder at all. I do this because you have the django project inside the "project_src" folder and the folder containing the settings file is named "project_name" instead of "project_src". Not a big difference, just for clarity.

- **python manage.py startapp my-app**
- python manage.py migrate
- python manage.py runserver
- **python manage.py createsuperuser** (to be able to login to admin)
- pip install djangorestframework django-cors-headers
- pip freeze > requirements.txt
- git init
- Create .gitignore file from gitignore.io
- git add .
- git commit -m "backend initial commit"
- Open project_src in pycharm and set up the interpeter to the virtual environment's python.exe

Frontend django app

1. Use a react template (like devias kit)

- cd project_src
- Create manually a folder inside your django project, named "web_client"
- Copy the template code inside this folder. Ideally the template source code is created with create-react-app and redux-toolkit and has the proper structure.
- cd to "web_client" and run npm install (to install template's dependencies)

- Add "proxy": "http://localhost:8000" to your package.json of the web_client folder (or use an env var instead)
- Add CORS headers for development (this is a backend thing)
  You do this either creating a custom django middleware or using corsheaders django app
- Get the csrf token from the cookie (or from the server) when your react app mounts, store it in redux state and add it to the post requests made by your react app.
- [production] Configure Django's staticfiles to serve the JS and CSS from create-react-app's build folder
- [production] create a django view to serve the index.html entry point

2. Create a react-app from scratch with create-react-app

- npx create-react-app web_client --template redux
- Delete the .git folder in the newly created web_client folder
  if you don't want to have a separate git repository inside the already existing django project git repository. The web client code would be part of the django project repository.

Decoupled Frontend with create-react-app

- npm install -g create-react-app
- cd my-project-dir_env
- create-react-app frontend (in the same dir with backend for uniformity)
- (A git repo is initialized automatically by create-react-app)
- npm start

Decoupled frontend with manual react setup

https://www.youtube.com/watch?v=GieYIzvdt2U&list=PLillGF-RfqbbRA-CIUxlxkUpbq0IFkX60&index=2

https://medium.com/better-programming/build-a-hello-world-react-app-with-a-django-api-backend-8ba814d89115

Tips

If there is a problem with immer which is defined as dependency to multiple packages causing problems apply this fix: https://github.com/rogeriochaves/npm-force-resolutions and use a new immer version

Update to the latest version of immer.

Development and Production Frontend django app

Development Workflow	Production Features
● python manage.py runserver -> Runs Django ● yarn start on frontend/ folder -> Runs Create React App 's server locally	● React builds are done on Heroku, no need to track build artifacts ● The app is available on the root URL ● The app generated by CRA (create-react-app) is untouched. Even Git Submodules can be used to link the project ● Static assets are served with GZIP compression ● Both pip packages and node_modules are cached in Heroku, thereby speeding up subsequent deploys. ● Even Session authentication can be used because it is being served from the same domain and no CORS issues

https://www.fusionbox.com/blog/detail/create-react-app-and-django/624/

In development you run the two servers manually. In production the react app and api would be served from the same domain. The react app (the static files that make it) would be served by django as static files. Django will also serve the entry point of the react app (index.html) with a django view in a catch all url.

Development workflow

1. Add "proxy": "http://localhost:8000" to your package.json of the frontend app. By default any request like fetch('/api/todos') will be made to the url from which the react app was served.

Run the two dev servers. Access the frontend like you do in a regular create-react-app project, at http://localhost:3000. Any API requests the frontend makes to http://localhost:3000 will get proxied to Django. So, on our code, we could do something like:

axios.get('/api/datasets/') and it will proxy the request to http://localhost:8000/api/datasets.

Keep in mind that proxy only has effect in development (with npm start), and it is up to you to ensure that URLs like /api/todos point to the right thing in production.

Notice: This works as long as the frontend uses only relative URLS, and doesn't follow links provided by the backend. If the frontend code does follow API links, they will be directly requested from runserver (http://localhost:8000), making them cross-origin requests. To make this work we'll need a way to add CORS headers only in local development.

2. Add CORS headers for development

```python
def dev_cors_middleware(get_response):
    """
    Adds CORS headers for local testing only to allow the frontend, which is served on
    localhost:3000, to access the API, which is served on localhost:8000.
    """
    def middleware(request):
        response = get_response(request)

        response['Access-Control-Allow-Origin'] = 'http://localhost:3000'
        response['Access-Control-Allow-Methods'] = 'GET, POST, PUT, PATCH, OPTIONS, DELETE, HE
        response['Access-Control-Allow-Headers'] = 'Content-Type, X-CSRFToken'
        response['Access-Control-Allow-Credentials'] = 'true'
        return response
    return middleware
```

```python
MIDDLEWARE.append('myapp.middleware.dev_cors_middleware')
```

Use it only in the development settings. In production react app and api will be served from the same domain.

Django production settings

(Django serves the built index.html page and the static files)

1. Build the react app
2. Configure Django's staticfiles to serve the JS and CSS from create-react-app's build

```python
REACT_APP_DIR = os.path.join(BASE_DIR, 'frontend')

STATICFILES_DIRS = [
    os.path.join(REACT_APP_DIR, 'build', 'static'),
]
```

In production settings configure the static file settings.

Now collectstatic will automatically find the static build artifacts, and deploy them however you have configured staticfiles. The only thing left is to serve the entry point to the React app, index.html.

3. Serve the built by react index.html file on all paths

First add the build folder as a template dir

```python
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [
            os.path.join(BASE_DIR, 'frontend/build')
        ],
```

Create a template view from the built index.html file for all urls

```python
    re_path(r'^(?P<path>.*)/$', TemplateView.as_view(template_name='index.html')),
    path('', TemplateView.as_view(template_name='index.html'))  # This matches the '
    # path('api-auth/', include('rest_framework.urls', namespace='rest_framework'))
]
```

The re_path matches all except '/'
The last path matches the '/'

Notice:

The entry point of the react app is the index.html file. This is also a static file but it can't be served as the rest of the static files. The static files are served from the static_url (for example *my_domain.com/static/*). but the index html has to be served at the root (*my_domain.com*). So we have to serve that file through a django view.

Notice:

This view must be installed with a catch-all urlpattern in order for pushState routing to work. Why? You visit example.com. The server serves the index.html and react fills the page. You visit example.com/users/list by pressing a link. The react app fetches data from server and renders a page normally. But if you reload the page or paste it in a new tab, then you will get the raw json data or an html with the data (using django rest content negotiation). You will not see the page you should see that is created by javascript using react. So you serve the index.html in all pages, and JavaScript has an access to the current page's URL so we may resolve the path and map it to some logic in our app, allowing us to render the correct page with react.

```python
from django.views.generic import TemplateView
from django.views.decorators.cache import never_cache

# Serve Single Page Application
index = never_cache(TemplateView.as_view(template_name='index.html'))
```

Notice:
Another article uses the never_cache decorator to add headers to a response so that it will never be cached. But index.html doesn't change. The bundle that generates the actual index view might change, but not the index.html. I don't think its necessary to use never_cache.

This is an alternative technique for step 5.

The view in myapp/views.py that serves the index.html

```
import logging

from django.views.generic import View
from django.http import HttpResponse
from django.conf import settings

class FrontendAppView(View):
    """
    Serves the compiled frontend entry point (only works if you have run `yarn
    run build`).
    """

    def get(self, request):
        try:
            with open(os.path.join(settings.REACT_APP_DIR, 'build', 'index.html')) as f:
                return HttpResponse(f.read())
        except FileNotFoundError:
            logging.exception('Production build of app not found')
            return HttpResponse(
                """
                This URL is only used when you have built the production
                version of the app. Visit http://localhost:3000/ instead, or
                run `yarn run build` to test the production version.
                """,
                status=501,
            )
```

Now once we configure out deployment process to run the create-react-app build script, users who visit our site will be served the React app. Other Django URLs like the API and admin will still continue to work, as long as their urlpatterns come before the catch all.

myapp/urls/py

```
from django.conf.urls import url
from . import views

urlpatterns = [
    # ... the rest of the urlpatterns ...
    # must be catch-all for pushState to work
    url(r'^', views.FrontendAppView.as_view()),
]
```

In a django template you define {% load static %} in the template that you require the static files. How do you define that the static files must be served in the case of a react app django app? When the react app is build, the index.html that is created loads the main js file among others. It does the job of loading the necessary static files.

(notice that when the static files of the react app are served by django you must find a way to make the build so that the index.html load the static files from the django static url, like src=/static/js/main.js. Do you?) no. The build process builds a static folder inside the build folder. In your django settings you just have to add that static folder to your statifiles_dir. This way these static files will be served by your static url which usually is your-domain/static/. the build process creates the index.html and the static folder in the same directory. So the index.html file loads the scripts with a relative path like <script src=”/static/…”. this is translated by the browser to your-domain/static/ (which should be defined as your static url)

Deployment

Notice that you either build locally and upload to heroku the already build files or you upload to heroku the react app source files and build in the heroku server (automatically). this requires additional configurations since you must have node installed in your heroku servers so that it builds the source code.

Heroku multiple buildpacks (django + react)

Heroku uses buildpacks to transform deployed code into slugs which can be executed by Dynos (server instances on Heroku). We’ll be needing two buildpacks. One for Node and another for Python. Our app would run on a Python server, even though we’ll use Node/NPM to build/bundle the React frontend. So the Python buildpack will be the main one in our config. The main buildpack determines the process type of the Heroku app.

The buildpack for the primary language of your app should always be the last buildpack in the list. This ensures that defaults for that primary language are applied instead of those for another language, and allows Heroku to correctly detect the primary language of your app. In case of react + django nodejs buildpack should be first and python buildpack second. Another reason that the node buildpack must be first, is that it needs to build the source code so that the python buildpack will use the built files. (We need to tell the Node.js buildpack to build the React app after it has installed Node and NPM. We can do this by adding the build command npm run build in the postinstall hook in the package.json)

Have also in mind the *django_heroku* library (built by heroku) that automatically configures your django settings for heroku serving static files with whitenoise.

Deployment sequence (commit source code and build in heroku)

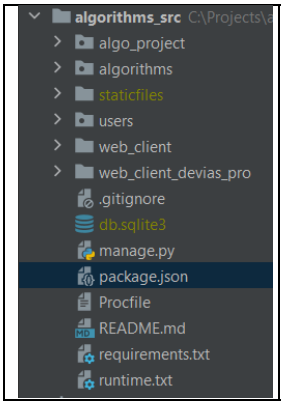
1. Configure django production settings as described above (static files, index template)
   Have in mind that "When a Django application is deployed to Heroku, $ python manage.py collectstatic --noinput is run automatically during the build. "

2. Configure package.json (node version, npm run build after commit)

```
"scripts": {
  "lint": "eslint ./src",
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "eject": "react-scripts eject",
  "postinstall": "npm run build"
},
"engines": {
  "node": "12.x",
  "npm": "6.x"
},
```

`"postinstall": "npm run build"` or (`"heroku-postbuild": "npm run build`)
The postinstall runs after the dependencies installation. This way when heroku installs the package dependencies, it will also build the package.

3. Create a root package.json file

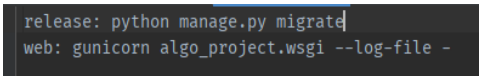


```
{
  "name": "algorithms-project",
  "version": "0.1.0",
  "scripts": {
    "postinstall": "npm install --prefix web_client_devias_pro"
  },
  "engines": {
    "node": "12.x",
    "npm": "6.x"
  }
}
```

You have to create a root level package.json file for your project. The reason is that heroku doesn't currently support installing a node package when the package is a subfolder of the project root. In this case the subfolder is the "*web_client_devias_pro*". one solution would be to move its contents under the root directory but then you wouldn't have the client code cleanly put inside its own folder.

So you create a root level package.json making the root itself a node package. This is a simple node package the job of which is to install another node package (the subfolder package). This way the heroku nodejs buildpack installs the root package and then the subfolder one.

4. Define the Procfile



```
release: python manage.py migrate
web: gunicorn algo_project.wsgi --log-file -
```

5. Define runtime.txt for python version on heroku (for example python-3.8.2)
6. Create new heroku app
7. Set the buildpacks for your heroku app
   `heroku buildpacks:set heroku/python -a my-app`
   `heroku buildpacks:add --index 1 heroku/nodejs -a my-app`
8. Provision an add on for example a db (`heroku addons:create heroku-postgresql:hobby-dev -a my-app`)
9. Set up env variables for your heroku app
10. Add a remote tracking branch for heroku (cd to your git repo and `heroku git:remote -a algorithms-project`)

Have in mind that when you have a heroku remote, you can type heroku cli commands without specifying the app. It extracts it from the current repo's heroku remote.

11. Push to heroku main (`git push heroku master`)

https://librenepal.com/article/django-and-create-react-app-together-on-heroku/ in this deployment static files are served by whitenoise and the react app is build by heroku. This is achieved by the nodejs buildpack that you define as the first one. The second one is the python buildpack.

Deploying with Kubernetes

https://datagraphi.com/blog/post/2021/2/10/kubernetes-guide-deploying-a-machine-learning-app-built-with-django-react-and-postgresql-using-kubernetes this author has some useful posts in general

serving static files

aws s3 behind cloudfront cdn

https://www.sebastian-lechner.info/serving-your-django-apps-static-and-media-files-with-s3-and-cloudfront/

whitenoise

With a couple of lines of config WhiteNoise allows your web app to serve its own static files, making it a self-contained unit that can be deployed anywhere without relying on nginx, Amazon S3 or any other external service. (Especially useful on Heroku, OpenShift and other PaaS providers.). WhiteNoise works with any WSGI-compatible app but has some special auto-configuration features for Django.

WhiteNoise will only work with storage backends that store their files on the local filesystem in STATIC_ROOT (so the static files are stored in the local file system of the django server. What happens if you scale out the web process? I guess that every instance would be able to serve the static files from its own local filesystem). It will not work with backends that store files remotely, for instance on Amazon S3.

Notice that if you care about performance and efficiency then you should be using WhiteNoise behind a CDN like CloudFront. If you're doing that then, because of the caching headers WhiteNoise sends, the vast majority of static requests will be served directly by the CDN without touching your application, so it really doesn't make much difference how efficient WhiteNoise is.

Notice that whitenoise is not suitable for serving user uploaded files (media files). it only checks for static files at startup and so files added after the app starts won't be seen. For that you have to use django-storages to store and serve them from elsewhere.

In a nutshell: Well, if adding a CDN is not on your agenda, then I would recommend using AWS S3 + Django-Storages instead of Whitenoise and Django-Compressor. One of the major benefits of using Whitenoise is the ability to integrate with a CDN. Without this benefit, the much simpler approach with AWS S3 serving both static and media files would make more sense.

Tips

Show django rest errors in react

Show the server generated error message instead of the generic bad request 400

https://redux-toolkit.js.org/api/createAsyncThunk#handling-thunk-results

https://github.com/reduxjs/redux-toolkit/issues/390

Google Search error.response.data unwrapresult react

The thunks generated by createAsyncThunk will always return a resolved promise (never a rejected one) with either the fulfilled action object or rejected action object inside, as appropriate. Redux Toolkit exports an unwrapResult function that can be used to extract the payload (in case of a fulfilled action object) or error (in case of rejected) from the action, and return or throw the result appropriately.

We said that the thunk returns a resolved promise, with either a fulfilled or a rejected action object. The fulfilled or rejected action object is created by the payload creator. Notice that the payload creator can return a rejected promise (the thunk can't). When it does, the thunk returns a rejected action object with a error attribute (action.error) that contains an automatically-serialized version of the error.

However, to ensure serializability, everything that does not match the SerializedError interface will have been removed from it (only name, message, stack and code attributes of the error are used). If you need to customize the contents of the rejected action, you should catch any errors yourself, and then return a new value using the thunkAPI.rejectWithValue utility. Doing return rejectWithValue(errorPayload) will cause the rejected action to use that value as action.payload

corsheaders

If you have two different servers for frontend and backend you need to add corsheaders package to django.

Csrf token

Csrf attack

let's say the vulnerable website is the vulnerable.com and the user has visited a website called malicious.com where he submitted a form. this form sent a post request to vulnerable.com (instead of to malicious.com). Since the request was made with the victim's browser, it sent any cookies the browser had for vulnerable.com. One of those cookies contained the authentication details for the victim. So the post request in vulnerable.com server was executed as if it came from the victim.

Adding a csrf token created for the victim by vulnerable.com to all unsafe requests to vulnerable.com, will protect the victim in the above case, because the malicious website will not have access to that csrf token and the request will not contain it.

- a cookie-less REST endpoint is completely immune from CSRF attacks

The jwt can be stored either in local storage or in a cookie. An http only cookie is the more secure option since this way you are less vulnerable to XSS attacks. But if you store it in a cookie, then this cookie will be sent along with every request which means that you need csrf protection. In this case, the client should get the csrf from an endpoint and then send it along with any request (either as a cookie, or as part of the request data). The easiest way to implement this is to read the csrf and store it in a cookie that is automatically sent along with any request to the auth server's url and is read by Django's middleware.

Why CSRF?

It really boils down to the browsers ability to automatically present login credentials for any request **by sending along cookies**. If a session id is stored in a cookie the browser will automatically send it along with all requests that go back to the original website. This means that an attacker doesn't actually have to know authentication details to take an action as the victim user. Rather, the attacker just has to trick the victims browser into making a request, and the credentials to authenticate the request will ride along for free.

An authentication system based on tokens (JWT or random) stored in cookies is vulnerable to CSRF attacks, because cookies are sent automatically to server in each request and an attacker could build a harmful url link to your site.

It would be worthy to note that a script from www.cute-cat-pictures.org normally does not have access to your CSRF token from www.mybank.com because of HTTP access control. But as I understood if the mybank.com sets the header Access-Control-Allow-Origin: * to it's responses then any origin could have access to the csrf token.

Django protects against CSRF attacks by generating a CSRF token in the server, send it to the client side, and mandating the client to send the token back in the request header. The server will then verify if the token from client is the same as the one generated previously; if not it will not authorize the request.Attackers won't be able to access this token due to protection by the Same-Origin Policy.

Csrf protection is not necessary for securing API endpoints

Rest API endpoints have a very important difference from other requests: they are specifically stateless and should never accept/use data from either a cookie or session. As a result, a REST API that sticks to the standard is automatically immune to such an attack. Even if a cookie was sent up by the browser, any credentials associated with the cookie would be completely ignored. Authentication of calls to a REST API are done in a completely different fashion. The most common solution is to have some sort of authentication key (an OAuth Token or the like) which is sent along in the header somewhere or possibly in the request body itself.

Since authentication is application-specific, and since the browser itself doesn't know what the authentication token is (since it is set in the header by javascript), there is no way for a browser to automatically provide authentication credentials even if it is somehow tricked into visiting the API endpoint. As a result, a cookie-less REST endpoint is completely immune from CSRF attacks.

Notice: If you use session authentication (which is the default authentication backend) then only authenticated requests require CSRF tokens and anonymous requests may be sent without CSRF tokens.

Django sets the csrf token to a cookie (and reads it from it), for all views from the csrf middleware.

Notice csrf protection is only required for unsafe methods which means that a GET request on a protected view, will not create a csrf token and cookie.

Apart from that, if you want to use the csrf token in the template (for example to have a hidden csrf field in a form) you have to use the {% csrf_token %} template tag. To use the value outside of django templates, you have to use the built in method for reading (or creating if none exists) the csrf token:

```
from django.middleware.csrf import get_token
csrf_token = get_token(request)
```

This function generates a new csrf token if it doesn't already exist in the request, and adds it to the request.META dictionary. Another middleware function sets this value to a cookie.

When you enable csrf protection for a view then the csrf cookie will be set by that protected view.

During development where the main index page is served by the node server you have no csrf cookie if no django view is called.

https://stackoverflow.com/questions/5207160/what-is-a-csrf-token-what-is-its-importance-and-how-does-it-work?rq=1

https://security.stackexchange.com/questions/166724/should-i-use-csrf-protection-on-rest-api-endpoints

https://stackoverflow.com/questions/45945951/jwt-and-csrf-differences

Webpack bundle for big projects

When you use the django based front end approach and your project is big containing a lot of js libraries and a lot of js code then the webpack bundle might end up bigger than 200KB which is considered a fair size limit. In cases like this you need to set up webpack to split the bundle to chunks (using splitChunks). This creates though an issue on how to load these chunks in your

django templates (if you use the django based front end approach). why exactly is not yet clear to me. A 200kb of compressed javascript code might correspond to 700kb of uncompressed code.

If you are using new webpack 4 split chunks API, then consider using `javascript_packs_with_chunks_tag` helper, which creates html tags for a pack and all the dependent chunks. `<%= javascript_packs_with_chunks_tag 'calendar', 'map', 'data-turbolinks-track': 'reload' %>` `<script src="/packs/vendor-16838bab065ae1e314.js" data-turbolinks-track="reload"></script>` `<script src="/packs/calendar~runtime-16838bab065ae1e314.js" data-turbolinks-track="reload"></script>` `<script src="/packs/calendar-1016838bab065ae1e314.js" data-turbolinks-track="reload"></script>` `<script src="/packs/map~runtime-16838bab065ae1e314.js" data-turbolinks-track="reload"></script>` `<script src="/packs/map-16838bab065ae1e314.js" data-turbolinks-track="reload"></script>`	This helper might be a solution

Have in mind that you need the trailing / in the urls for example example.com/api/leads/4/

To use variables in JSX in a string you need to use `string_${id}` instead of " "

When you copy generic html code to a react component replace the class html attribute with its corresponding className JSX attribute

`"scripts": {` `  "dev": "webpack --mode development --watch ./leadmanager/frontend/src/index.js --output ./leadmanager/frontend/static/frontend/main.js",` `  "build": "webpack --mode production ./leadmanager/frontend/src/index.js --output ./leadmanager/frontend/static/frontend/main.js"` `},`	In webpack.config.js there are the commands that we run (npm dev and npm build) to compile the js files into the defined output. Notice that in dev we can add watch so that it watches for changes in the files and re compiles on every change. (this is not hot reloading)

- **Gitignore.io**

Go to gitignore.io type django and will create a gitignore file specifically for django.

Redux

Tips

Data flow

Initial setup:

- A Redux store is created using a root reducer function
- The store calls the root reducer once, and saves the return value as its initial state
- When the UI is first rendered, UI components access the current state of the Redux store, and use that data to decide what to render. They also subscribe to any future store updates so they can know if the state has changed.

Updates:

- Something happens in the app, such as a user clicking a button
- The app code dispatches an action to the Redux store, like dispatch({type: 'counter/increment'})
- The store runs the reducer function again with the previous state and the current action, and saves the return value as the new state
- The store notifies all parts of the UI that are subscribed that the store has been updated

- Each UI component that needs data from the store checks to see if the parts of the state they need have changed.
- Each component that sees its data has changed forces a re-render with the new data, so it can update what's shown on the screen

Generic tips

- An action is a js object with a type property. An action creator is a function that returns an action object
- An action is dispatched with the redux dispatch function which gets an action creator.
- A reducer takes the current state and an action and returns the resulting state
- The reducers must be pure functions that don't mutate the state directly. So instead of pushing a new element to the array we create a brand new array containing all the previous state in addition to the new element
- Any action dispatched from any view will trigger all the reducers (combined reducers) being invoked to check the action type.
- When we pass an object like {counter: counterReducer} to combinereducers, we add a state.counter section to the store
- we get a store value using the useSelector hook offered from react-redux
- Redux actions and state should only contain plain JS values like objects, arrays, and primitives. Don't put class instances, functions, or other non-serializable values into Redux

- React-redux It is binding library that connects redux with react. It handles the creation of the store, subscribing to the store, checking for updated data, and triggering a re-render for subscribed components. Components are subscribed to the tsore with the use of the react-redux hooks useSelector, useDispatch, useStore. useSelector is used to read a specific state (a member from the store object)
- useDispatch() is used to dispatch an action.
- useSelector reads values from the store object. the useSelector hook will run
    - whenever the function component renders
    - whenever an action is dispatched and the redux store is updated

  If the selector returns a different value than last time, useSelector will make sure our component re-renders with the new value.
- You may call useSelector() multiple times within a single function component. Each call to useSelector() creates an individual subscription to the Redux Store. Because of the React update batching behavior used in React Redux v7, a dispatched action that causes multiple useSelector()s in the same component to return new values should only result in a single re-render.
- We can create selector functions to be invoked by useSelect to read the store. Write them in the slice file. If store format changes, you only change the selectors code. you can also use memoized selectors. meoizing selectors is achieved usually through the reselect package
- Redux provides a third-party extension point between dispatching an action, and the moment it reaches the reducer. People use Redux middleware for logging, crash reporting, talking to an asynchronous API, routing, and more

- Async-thunk is a middleware that lets you write async logic that interacts with the store. you dispatch thunk function creators. They are written in the slice file.
- Notice that the typical logic of async data fetching is a start action that triggers isloading, the request is made and depending on the request result we get a success or failure action. You can have only a success action if you don't care about isloading and failure.
- This pattern is provided by default by the createAsyncThunk of the redux-toolkit. It uses 3 actions pending, fulfilled, and rejected (posts/fetchPosts/pending, posts/fetchPosts/fulfilled, posts/fetchPosts/rejected).
- You can keep a status in your state and modify it based on these actions, to control your UI. The typical status values are idle, loading, succeeded, failed. In createAsyncThunk the status is idle only before the first time the component makes a call. After the call the status would be "succeeded" or "failed". If the call is made again (without reload the page) the status would switch directly to "loading".

- Redux Toolkit has a function called ***createSlice***, which takes care of the work of generating action type strings, action creator functions, and action objects (and the slice reducer function, the function that checks action types and modifies the state accordingly).

- You can only write "mutating" logic in Redux Toolkit's *createSlice* and *createReducer* because they use Immer inside! If you write mutating logic in reducers without Immer, it will mutate the state and cause bugs!

Call thunks asynchronously (todo..)

```
// you only need to make the onCreatePressed async if you want to await the thunk.
// you would want to await the thunk in order to catch any asyncThunk internal errors with the unwrapResult function
const onCreatePressed = async (values) => {
    // in case of no error the thunk returns a resolved promise with a fulfilled action object
    // in case of error, the thunk returns a resolved promise with a rejected action object
    // console.log('values', values, 'csrf_token', csrfToken)
    const createResult = await dispatch(createAlgorithmThunk( arg: {'name': values.name, 'csrfToken': csrfToken}))
    // console.log("result of dispatching the create thunk:", createResult)
    // we don't use the try catch here. We use it inside the createAlgorithmThunk's payload creator so that we get the
    // server generated message
    // try{
    //     const create_result = await dispatch(createAlgorithmThunk({'name': values.name, 'csrfToken': csrfToken}))
    //     unwrapResult(create_result)
    //     setName('')
    // }catch (error){
    //     // This error is the action.error that can be inspected in the redux dev tools window
    //     console.log('algorithm created caught error', error)
    // }
}
```

I don't use unwrap result. I use the rejectWithValue inside the asyncThunk. So could I use a typical synchronous function and call?

axios. Show server errors instead of javascript errors

```
axios.get('/api/xyz/abcd')
  .catch(function (error) {
    if (error.response) {
      // Request made and server responded
      console.log(error.response.data);
      console.log(error.response.status);
      console.log(error.response.headers);
    } else if (error.request) {
      // The request was made but no response was received
      console.log(error.request);
    } else {
      // Something happened in setting up the request that triggered an Error
      console.log('Error', error.message);
    }

  });
```

https://github.com/axios/axios/issues/960
The **error.response** is particularly useful since you can use the server error data.

The error.message is present in all cases as I saw.

Intro

Have in mind that you can avoid the extra boilerplate of redux and its additional complexity in small projects just by using context or passing down props (or using hooks).

The way Redux works is that it essentially stores all the dynamic information of an app in a single JavaScript object. Whenever a part of the app needed to show some data, it would request the information from the server, update the single JavaScript object, and then show that data to the users.

Redux was great for solving the problem of keeping your front-end application in sync, however it brought it's own problems: Extra code, stale data (keeping all data in one place you may have unwanted data appearing in your app from a previous state), steep learning curve

Redux is a very simple library that enables predictable atomic state changes. While the API is tiny and easy to learn, it gives you a lot of power and solves many of the problems when handling the application state and sharing data between components.

Redux is a state container for JavaScript apps, often called a Redux store. It stores the whole state of the app in an immutable object tree.

One of the important concepts of redux:

The state can only be changed by dispatching an action. The state tree is never mutated directly instead you use **pure functions called reducers**. A reducer takes the current state and an action and returns the resulting state. (The fact that the Redux state changes predictably opens up a lot of debugging possibilities. For example, using time travel makes it possible to travel back and forth between different states).

The reducers must be pure functions that don't mutate the state directly. So instead of pushing a new element to the array we create a brand new array containing all the previous state in addition to the new element. Redux is deterministic (the current state is only dependent on the previous actions) and that makes the behavior of the app very predictable and less prone to bugs.

Redux is in itself not written specifically for React. It can be used with other frameworks such as Angular, Ember or Vue.js. The easiest way to use Redux with React is the official React Redux binding library. With this library it's easy to bind the Redux state and actions to props.

In apps using React Redux the components are often separated into two categories, components and containers. This is to distinguish components that use the state tree and dispatch actions from dumb components that are only used for presentation. This is one of the strengths of Redux, by separating the state from the graphical presentation we can make tiny reusable components.

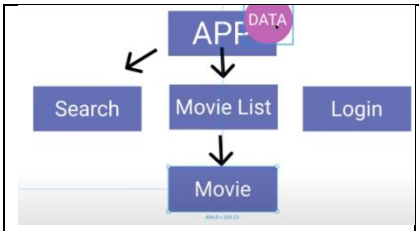
Redux is a great library for managing the state of React apps. Storing the whole state in a single tree is very useful since it avoids having multiple sources of truth.

Getting all users with Redux

1. Create the component to show a list of users (no problems here).
2. Create the fetch call to the API.
3. Add a new field in the state.
4. Add a new action which updates the state with data.
5. Add a new thunk method which executes the fetch call and then updates the state using our new actions.
6. Add the thunk function to our component using connect() which would wrap it in a dispatch function.
7. Extract the data from the Redux state, again by using connect() .
8. Declare the thunk function and the extracted data field in my component's prop types.
9. Call the thunk method in the componentDidMount() function.
10. Finally render the data in the DOM.

A lot of work!

Overview



Lets say you have a movies app with 5 components. One of them is the MovieList which renders a list of all movies. So it must have access to the actual movies data. So the data must reside in this component. If the Movie component needs the data too, then we can pass data to it from MovieList via props. But if the Search component needs the data, we have no way to pass it from MovieList since data can only flow in one direction. The solution would be to lift data up to the App component and pass it down as props to the lower components. But it would be more appropriate for the movies data to live in the Movie component. Apart from that if we want to have some actions to the data too, like deleting a movie in the Movie component for example, we couldn't do it directly in the Movie component since a component must never modify its props. We should do it from a function of the App component (which contains the data). So we have to have a delete function in the App component and pass it down to the Movie component along with the data.
So as a project starts to become big things start to get really complex really fast. That's what redux tries to solve by storing the data separately in a distinct object which all components can access and modify, obeying though to specific rules.

Actions

```
//ACTION INCREMENT
const increment = () => {
    return {
        type: 'INCREMENT'
    }
}
const decrement = () => {
    return {
        type: 'DECREMENT'
    }
}
```

An action is a function that returns an object. Actions are a declaration of what you want to do.

An action creator is a function that creates and returns an action object. We typically use these so we don't have to write the action object by hand every time. Increment and decrement are action creators.

We usually write the action's type string like "domain/eventName", where the first part is the feature or category that this action belongs to, and the second part is the specific thing that happened.

Reducers

A reducer is a function that receives the current state and an action object, decides how to update the state if necessary, and returns the new state: (state, action) => newState.

```
const counterReducer = (state = 0, action) => {
    switch (action.type) {
        case 'INCREMENT':
            return state + 1;
        case 'DECREMENT':
            return state - 1;
        default:
            return state;
    }
};

export default counterReducer;
```

```
const loggedReducer = (state = false, action) => {
    switch (action.type) {
        case 'SIGN_IN':
            return !state;
        default:
            return state;
    }
};

export default loggedReducer;
```

You create one reducer for each state object that you want to store in the global store object. For example in this case we have a counterReducer that modifies a state variable which is used as a counter and a loggedReducer that modifies its state which is used as a loggedIn variable (true/false).

Pass a value to the reducers

```
<button onClick={() => dispatch(increment(5))}>+</button>
```

```
export const increment = nr => {
    return {
        type: 'INCREMENT',
        payload: nr
    };
};
```

```
const counterReducer = (state = 0, action) => {
    switch (action.type) {
        case 'INCREMENT':
            return state + action.payload;
        case 'DECREMENT':
            return state - 1;
        default:
            return state;
    }
};
```

You modify the action so that it accepts a variable and add an attribute usually called payload to the action object. Then you can access that value in your reducer.

Rules of reducers

- They should only calculate the new state value based on the state and action arguments
- They are not allowed to modify the existing state. Instead, they must make immutable updates, by copying the existing state and making changes to the copied values.
- They must not do any asynchronous logic or other "side effects"

Store

```
import { createStore } from 'redux';

let store = createStore(counter);
```

The store is the global state object which you create with the *createStore()* function that gets a reducer as an argument.

```
import counterReducer from './counter';
import loggedReducer from './isLogged';
import { combineReducers } from 'redux';

const allReducers = combineReducers({
    counter: counterReducer,
    isLogged: loggedReducer
});
import allReducer from './reducers';

const store = createStore(allReducer);
```

The global store object is not only modified but also created by the reducers. If you have many reducers you have to combine them with the combineReducers() function and pass the combined result to the createStore() function. This will add two attributes to the global store object, counter and isLogged as they are defined in the combineReducers().

When we pass in an object like {counter: counterReducer}, that says that we want to have a state.counter section of our Redux state object, and that we want the counterReducer function to be in charge of deciding if and how to update the state.counter section whenever an action is dispatched.

Notice

Components can't talk to the Redux store directly, because we're not allowed to import it into component files. so we get a store value using the useSelector hook offered from react-redux.

```
  postAdded: {
    reducer(state, action) {
      state.push(action.payload)
    },
    prepare(title, content, userId) {
      return {
        payload: {
          id: nanoid(),
          date: new Date().toISOString(),
          title,
          content,
          user: userId,
        },
      }
    },
  },
```

Notice:
Redux actions and state should only contain plain JS values like objects, arrays, and primitives. Don't put class instances, functions, or other non-serializable values into Redux!

So we store the date as a string

❖ Setting up store with redux-toolkit

We use the configureStore function. Redux allows store setup to be customized with different kinds of plugins ("middleware" and "enhancers"). configureStore automatically adds several middleware to the store setup by default to provide a good developer experience, and also sets up the store so that the Redux DevTools Extension can inspect its contents.

```
import { configureStore } from '@reduxjs/toolkit'
import usersReducer from '../features/users/usersSlice'
import postsReducer from '../features/posts/postsSlice'
import commentsReducer from '../features/comments/commentsSlice'

export default configureStore({
  reducer: {
    users: usersReducer,
    posts: postsReducer,
    comments: commentsReducer
  }
})
```

```
import { configureStore } from '@reduxjs/toolkit'

const store = configureStore({ reducer: counterReducer })

console.log(store.getState())
// {value: 0}
```

The current Redux application state lives in an object called the store . The store is created by passing in a reducer, and has a method called getState that returns the current state value

Dispatching

The Redux store has a method called *dispatch*. The only way to update the state is to call store.dispatch() and pass in an action object. The store will run its reducer function and save the new state value inside, and we can call getState() to retrieve the updated value. You can think of dispatching actions as "triggering an event" in the application. Something happened, and we want the store to know about it. Reducers act like event listeners, and when they hear an action they are interested in, they update the state in response.

```jsx
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { increment, decrement } from './actions';

function App() {
    const counter = useSelector(state => state.counter);
    const isLogged = useSelector(state => state.isLogged);
    const dispatch = useDispatch();

    return (
        <div className="App">
            <h1>Counter {counter}</h1>
            <button onClick={() => dispatch(increment())}>+</button>
            <button onClick={() => dispatch(decrement())}>+</button>
            {isLogged ? <h3>Valuable Information I shouldn't see</h3> : ''}
        </div>
    );
}

export default App;
```

We typically call action creators to dispatch the right action:

Selectors

It would be nice if we didn't have to keep rewriting our components every time we made a change to the data format in our reducers. One way to avoid this is to define reusable selector functions in the slice files, and have the components use those selectors to extract the data they need instead of repeating the selector logic in each component. That way, if we do change our state structure again, we only need to update the code in the slice file.

Selectors are functions that know how to extract specific pieces of information from a store state value. As an application grows bigger, this can help avoid repeating logic as different parts of the app need to read the same data:

features/posts/postsSlice.js

```jsx
const postsSlice = createSlice(/* omit slice code*/)

export const { postAdded, postUpdated, reactionAdded } = postsSlice.actions

export default postsSlice.reducer

export const selectAllPosts = state => state.posts

export const selectPostById = (state, postId) =>
  state.posts.find(post => post.id === postId)
```

features/posts/PostsList.js

```jsx
// omit imports
import { selectAllPosts } from './postsSlice'

export const PostsList = () => {
  const posts = useSelector(selectAllPosts)
  // omit component contents
}
```

Then use them in the components

features/posts/SinglePostPage.js

```jsx
// omit imports
import { selectPostById } from './postsSlice'

export const SinglePostPage = ({ match }) => {
  const { postId } = match.params

  const post = useSelector(state => selectPostById(state, postId))
  // omit component logic
}
```

Notice the difference in using selectAllPosts and selectPostById that gets one additional argument apart from the state.

Notice that the selector should return the whole path to the state value, including the name defined for it in the store, so that it can be used from everywhere.

You can also create "memoized" selectors that can help improve performance, which we'll look at in a later part of this tutorial.

react-redux module

It is binding library that connects redux with react.

Integrating Redux with a UI. Using Redux with any UI layer requires the same consistent set of steps:

1. Create a Redux store
2. Subscribe to updates
3. Inside the subscription callback:
   a) Get the current store state
   b) Extract the data needed by this piece of UI
   c) Update the UI with the data
4. If necessary, render the UI with initial state
5. Respond to UI inputs by dispatching Redux actions

While it is possible to write this logic by hand, doing so would become very repetitive. In addition, optimizing UI performance would require complicated logic. The process of subscribing to the store, checking for updated data, and triggering a re-render can be made more generic and reusable. A UI binding library like React Redux handles the store interaction logic, so you don't have to write that code yourself.

Two main things we have to do is to provide the redux store to the app and then connect the store with the react components.

1. Providing the store object to the app

Index.js `import { Provider } from 'react-redux';` `const store = createStore(` `  allReducer,` `  window.__REDUX_DEVTOOLS_EXTENSION__ && window.__REDUX_DEVTOOLS_EXTENSION__()` `);` `ReactDOM.render(` `  <Provider store={store}>` `  <App />` `  </Provider>,` `  document.getElementById('root')` `);`	It provides a Provider component that wraps your react App component and accepts the store object as an argument. This way we make the store available to our app. Any React components that call useSelector or useDispatch will be talking to the Redux store we gave to the <Provider>.

2. Connecting to components

There are two ways to do this:

- The legacy way to do this is using the connect() function of react-redux. Connect() is a higher order component. Check the react-redux tutorial.
- React Redux now offers a set of hook APIs as an alternative to the existing connect() Higher Order Component. These APIs allow you to subscribe to the Redux store and dispatch actions, without having to wrap your components in connect(). these hooks are the useSelector, useDispatch, useStore

Using hooks

While React includes several built-in hooks like useState and useEffect, other libraries can create their own custom hooks that use React's hooks to build custom logic.

React components can read data from the Redux store using the useSelector hook from the React-Redux library.

<table>
<tr>
<td>

```jsx
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { increment, decrement } from './actions';

function App() {
    const counter = useSelector(state => state.counter);
    const isLogged = useSelector(state => state.isLogged);
    const dispatch = useDispatch();

    return (
        <div className="App">
            <h1>Counter {counter}</h1>
            <button onClick={() => dispatch(increment())}>+</button>
            <button onClick={() => dispatch(decrement())}>+</button>
            {isLogged ? <h3>Valuable Information I shouldn't see</h3> : ''}
        </div>
    );
}

export default App;
```

In case that you use a separate selectors.js file to store all of your selectors:

```jsx
// selectors.js
export const getTodoList = store =>
  store && store.todos ? store.todos.allIds : [];
```

```jsx
import {getTodoList} from ‘./selectors’;
const todo_list = useSelector(getTodosList}
```

</td>
<td>

(This is an example using hook apis).
We can select and modify the global store from any react component. To do so we use the useSelector() and useDispatch() functions of react-redux.

useSelector is used to select an attribute from the store, which we assign to a variable that we can then use in our component.

useDispatch() is used to dispatch an action.

The selector getTodoList is a function that gets the state as argument and returns an object from it.

</td>
</tr>
</table>

Local component state is kept in useState hooks

useSelector

Our components can't talk to the Redux store directly, because we're not allowed to import it into component files. But, useSelector takes care of talking to the Redux store behind the scenes for us. If we pass in a selector function, it calls `someSelector(store.getState())` for us, and returns the result.

`const count = selectCount(store.getState())` `console.log(count)` `// 0`	If we had access to a Redux store in a component, we could retrieve the current counter value as:
`const count = useSelector(selectCount)`	Since we haven’t access to store inside the component we use the useSelector

Any time an action has been dispatched and the Redux store has been updated, useSelector will re-run our selector function. If the selector returns a different value than last time, useSelector will make sure our component re-renders with the new value.

https://react-redux.js.org/api/hooks#equality-comparisons-and-updates

```
const result: any = useSelector(selector: Function, equalityFn?: Function)
```

The selector function which is the first argument of the useSelector hook, is approximately equivalent to the mapStateToProps argument to connect conceptually. Note: The selector function should be pure since it is potentially executed multiple times and at arbitrary points in time. The selector will be called with the entire Redux store state as its only argument.

The selector will be run

- whenever the function component renders (A cached result may be returned by the hook without re-running the selector if it's the same function reference as on a previous render of the component.)
- useSelector() will also subscribe to the Redux store, and run your selector whenever an action is dispatched. If it returns a new reference value the component will rerender.

It has some differences with the mapStateToProps though, see docs.

- the default comparison is a strict === reference comparison, not a ==

- others

Note that some extra care should be taken if we want to use memoized selectors (meoizing selectors is achieved usually through the reselect package). See the notes.

useDispatch

Similarly, we know that if we had access to a Redux store, we could dispatch actions using action creators, like store.dispatch(increment()). Since we don't have access to the store itself, we need some way to have access to just the dispatch method. The useDispatch hook does that for us, and gives us the actual dispatch method from the Redux store:

This hook returns a reference to the dispatch function from the Redux store. You may use it to dispatch actions as needed.

Note

When passing a callback using dispatch to a child component, it is recommended to memoize it with useCallback (Returns a memoized callback), since otherwise child components may render unnecessarily due to the changed reference.

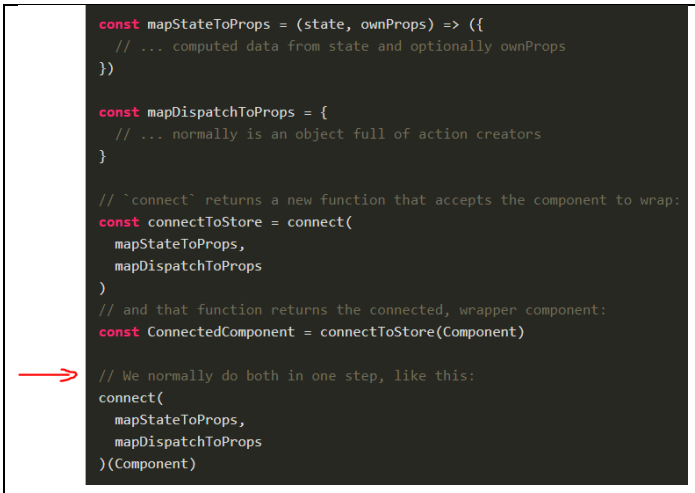
useStore

This hook returns a reference to the same Redux store that was passed in to the <Provider> component. This hook should probably not be used frequently. Prefer useSelector() as your primary choice.

Custom context

The <Provider> component allows you to specify an alternate context via the context prop. This is useful if you're building a complex reusable component, and you don't want your store to collide with any Redux store your consumers' applications might use.

Using connect()



```
const mapStateToProps = (state, ownProps) => ({
  // ... computed data from state and optionally ownProps
})

const mapDispatchToProps = {
  // ... normally is an object full of action creators
}

// `connect` returns a new function that accepts the component to wrap:
const connectToStore = connect(
  mapStateToProps,
  mapDispatchToProps
)
// and that function returns the connected, wrapper component:
const ConnectedComponent = connectToStore(Component)

// We normally do both in one step, like this:
connect(
  mapStateToProps,
  mapDispatchToProps
)(Component)
```

React Redux provides a connect function for you to read values from the Redux store (and re-read the values when the store updates). The connect function takes two arguments, both optional:

mapStateToProps: called every time the store state changes. It receives the entire store state, and should return an object of data this component needs.
mapDispatchToProps: this parameter can either be a function, or an object.

- If it's a function, it will be called once on component creation. It will receive dispatch as an argument, and should return an object full of functions that use dispatch to dispatch actions.
- If it's an object full of action creators, each action creator will be turned into a prop function that automatically dispatches its action when called. Note: We recommend using this "object shorthand" form.

- mapDispatchToProps implementation

```
// components/AddTodo.js

import React from 'react'
import { connect } from 'react-redux'
import { addTodo } from '../redux/actions'

class AddTodo extends React.Component {
  // ...

  handleAddTodo = () => {
    // dispatches actions to add todo
    this.props.addTodo(this.state.input)

    // sets state back to empty string
    this.setState({ input: '' })
  }

  render() {
    return (
      <div>
        <input
          onChange={e => this.updateInput(e.target.value)}
          value={this.state.input}
        />
        <button className="add-todo" onClick={this.handleAddTodo}>
          Add Todo
        </button>
      </div>
    )
  }
}

export default connect(
  null,
  { addTodo }
)(AddTodo)
```

Let's work on <AddTodo /> first. It needs to trigger changes to the store to add new todos. Therefore, it needs to be able to dispatch actions to the store.
By passing the action to connect, our component receives it as a prop, and it will automatically dispatch the action when it's called. Notice now that <AddTodo /> is wrapped with a parent component called <Connect(AddTodo) />. Meanwhile, <AddTodo /> now gains one prop: the addTodo action.

We also need to implement the handleAddTodo function to let it dispatch the addTodo action and reset the input. Now our <AddTodo /> is connected to the store. When we add a todo it would dispatch an action to change the store.

- mapStateToProps implementation

The <TodoList /> component is responsible for rendering the list of todos. Therefore, it needs to read data from the store. We enable it by calling connect with the mapStateToProps parameter, a function describing which part of the data we need from the store.

```
// components/TodoList.js

// ...other imports
import { connect } from "react-redux";
import { getTodos } from "../redux/selectors";

const TodoList = // ... UI component implementation

export default connect(state => ({ todos: getTodos(state) }))(TodoList);
```

index.js ● TodoList.js ×

```
1  import React from "react";
2  import Todo from "./Todo";
3
4  const TodoList = ({ todos }) => (
5    <ul className="todo-list">
6      {todos && todos.length
7        ? todos.map((todo, index) => {
8            return <Todo key={`todo-${todo.id}`} todo={todo} />;
9          })
10       : "No todos, yay!"}
11   </ul>
12 );
13
14 export default TodoList;
```

The mapStateToProps returns an object in the form `{ "todos": [list of todo items]}`
What the todo item is is not important in this context. In this case this list is returned by the selector function `getTodos()` that lives in the selectors.js

The TodoList react component takes an argument which is an object, from which it reads the object's "todos" argument (it does so with object destructuring).

This way every time the store is modified, the mapStateToProps function of all connected components is executed and in case of TodoList component, the component is re-rendered if there are changes in the "todos" list.

Common ways of calling `connect`

Depending on what kind of components you are working with, there are different ways of calling `connect`, with the most common ones summarized as below:

	Do Not Subscribe to the Store	Subscribe to the Store
Do Not Inject Action Creators	`connect()(Component)`	`connect(mapStateToProps)(Component)`
Inject Action Creators	`connect(null, mapDispatchToProps)(Component)`	`connect(mapStateToProps, mapDispatchToProps)(Component)`

Notice that even in cases in which you use connect() without the mapDispatchToProps function you receive `props.dispatch` that you may use to manually dispatch actions.

Middleware

Redux provides a third-party extension point **between dispatching an action, and the moment it reaches the reducer**. People use Redux middleware for logging, crash reporting, talking to an asynchronous API, routing, and more. Middleware extend the store and allow you to:

- Execute extra logic when any action is dispatched (such as logging the action and state)
- Pause, modify, delay, replace, or halt dispatched actions
- Write extra code that has access to dispatch and getState
- Teach dispatch how to accept other values besides plain action objects, such as functions and promises, by intercepting them and dispatching real action objects instead

Suppose that we want to log to the console every action and its corresponding resulting state. We also want to send any error in dispatching an action to a third party service like sentry. These things are done after we dispatch an action. We could just repeat them any time we call dispatch but this is not a very good approach. One approach would be to monkeypatch the dispatch method of the store object. But the best way is by using a middleware mechanism.

Monkey patching chain

Here is a nice pattern for easily monkey patching an object (replacing one of its methods)

```
function patchStoreToAddLogging(store) {
  const next = store.dispatch
  store.dispatch = function dispatchAndLog(action) {
    console.log('dispatching', action)
    let result = next(action)
    console.log('next state', store.getState())
    return result
  }
}

function patchStoreToAddCrashReporting(store) {
  const next = store.dispatch
  store.dispatch = function dispatchAndReportErrors(action) {
    try {
      return next(action)
    } catch (err) {
      console.error('Caught an exception!', err)
      Raven.captureException(err, {
        extra: {
          action,
          state: store.getState()
        }
      })
      throw err
    }
  }
}
```

If these functions are published as separate modules, we can later use them to patch our store:

```
patchStoreToAddLogging(store)
patchStoreToAddCrashReporting(store)
```

Using middleware mechanism

The middleware function in ES5	ES6 equivalent

```
function logger(store) {
  return function wrapDispatchToAddLogging(next) {
    return function dispatchAndLog(action) {
      console.log('dispatching', action)
      let result = next(action)
      console.log('next state', store.getState())
      return result
    }
  }
}
```

```
const logger = store => next => action => {
  console.log('dispatching', action)
  let result = next(action)
  console.log('next state', store.getState())
  return result
}
```

```
import { createStore, combineReducers, applyMiddleware } from 'redux'

const todoApp = combineReducers(reducers)
const store = createStore(
  todoApp,
  // applyMiddleware() tells createStore() how to handle middleware
  applyMiddleware(logger, crashReporter)
)
```

Using the logger middleware (and the crashReporter)

```
// Warning: Naïve implementation!
// That's *not* Redux API.
function applyMiddleware(store, middlewares) {
  middlewares = middlewares.slice()
  middlewares.reverse()
  let dispatch = store.dispatch
  middlewares.forEach(middleware => (dispatch = middleware(store)(dispatch)))
  return Object.assign({}, store, { dispatch })
}
```

This is the applyMiddleware implementation (with some differences).

It first obtains the final, fully wrapped dispatch() function, and returns a copy of the store using it.

Now, whenever an action is dispatched, it will pass from all the applied middleware where each one of them will do its stuff.

Redux toolkit

Redux toolkit

In the same spirit with create-react-app, sets up some things by default.

- Provides the ***createSlice*** function that gives the possibility to write redux slices (using the Immer package among others)

Apart from the slices, you put in the slice file the thunks and the selectors.

- Provides ***configureStore*** function to set up the store. It adds certain middleware by default (thunk and others)
- Provides ***createAsyncThunk*** API that generates thunks that automatically dispatch those "start/success/failure" actions for you.
- Provides the ***createSelector*** function of reselect for creating memoized selectors
- Provides the ***createEntityAdapter*** function for normalizing the way items are stored in the global state
  { ids: [], entities: {} }

Redux Slices

A "slice" is a collection of Redux reducer logic and actions for a single feature in your app, typically defined together in a single file. The name comes from splitting up the root Redux state object into multiple "slices" of state.

In the previous example, state.users, state.posts, and state.comments are each a separate "slice" of the Redux state. Since *usersReducer* is responsible for updating the *state.users* slice, we refer to it as a **"slice reducer" function**.

features/counter/counterSlice.js

```
import { createSlice } from '@reduxjs/toolkit'

export const counterSlice = createSlice({
  name: 'counter',
  initialState: {
    value: 0
  },
  reducers: {
    increment: state => {
      // Redux Toolkit allows us to write "mutating" logic in reducers. It
      // doesn't actually mutate the state because it uses the immer library,
      // which detects changes to a "draft state" and produces a brand new
      // immutable state based off those changes
      state.value += 1
    },
    decrement: state => {
      state.value -= 1
    },
    incrementByAmount: (state, action) => {
      state.value += action.payload
    }
  }
})

export const { increment, decrement, incrementByAmount } = counterSlice.actions

export default counterSlice.reducer
```

Every time we create a new slice, we need to add its reducer function to our Redux store.

Redux Toolkit has a function called ***createSlice***, which takes care of the work of generating action type strings, action creator functions, and action objects (and the slice reducer function, the function that checks action types and modifies the state accordingly).
All you have to do is define a name for this slice, write an object that has some reducer functions in it, and it generates the corresponding action code automatically.
The **string** from the name option is used as the first part of each action type, and the **key name of each reducer** function is used as the second part. So, the "counter" name + the "increment" reducer function generated an action type of {type: "**counter/increment**"}.
In addition to the name field, createSlice needs us to pass in the initial state value for the reducers, so that there is a state the first time it gets called. Here the state will be **counter.value**. {counter: {value: 0}}

The *slice.actions* return action creators
The *slice.reducer* returns the slice reducer function. It is imported to the file that creates the store, to be added to the store.

```
console.log(counterSlice.actions.increment())
// {type: "counter/increment"}
```

createSlice automatically generates action creators with the same names as the reducer functions we wrote (functions that return action objects). You can run them at any point in time and the action will be returned. I guess that if you don't dispatch it the action will not be dispatched and will not have any effect on the UI.

```
const newState = counterSlice.reducer(
  { value: 10 },
  counterSlice.actions.increment()
)
console.log(newState)
// {value: 11}
```

It also generates the slice reducer function that knows how to respond to all these action types:
Here we manually run a reducer function with a given state and action object.

```
return (
  <div>
    <div className={styles.row}>
      <button
        className={styles.button}
        aria-label="Increment value"
        onClick={() => dispatch(increment())}
      >
        +
      </button>
```

We dispatch the automatically created action creators (that have the same name with the defined reducer objects)

```
<button
  className={styles.button}
  onClick={() =>
    dispatch(incrementByAmount(Number(incrementAmount) || 0))
  }
>
  Add Amount
</button>
```

Dispatching an automatically created action with a payload.

You can only write "mutating" logic in Redux Toolkit's *createSlice* and *createReducer* because they use Immer inside! If you write mutating logic in reducers without Immer, it will mutate the state and cause bugs!

extraReducers

However, there are times when a slice reducer needs to respond to other actions that weren't defined as part of this slice's reducers field. (for example the actions dispatched automatically by the createAsyncThunk function). We can do that using the slice extraReducers field instead. More details on the *making async calls* notes…

Prepare callback

features/posts/postsSlice.js

```
const postsSlice = createSlice({
  name: 'posts',
  initialState,
  reducers: {
    postAdded: {
      reducer(state, action) {
        state.push(action.payload)
      },
      prepare(title, content) {
        return {
          payload: {
            id: nanoid(),
            title,
            content
          }
        }
      }
    }
    // other reducers here
  }
})
```

features/posts/AddPostForm.js

```
const onSavePostClicked = () => {
  if (title && content) {
    dispatch(postAdded(title, content))
    setTitle('')
    setContent('')
  }
}
```

Previously in the onSavePostClicked function of the appPost component we had the logic that creates the action's payload object. Here the logic was simple enough (just calculating the id of the new post with nanoid) but it could be more complex. This is a problem. If we want to dispatch this action from another component, we would have to copy that logic there too.
The solution is to put this logic inside the action creator function. But with createSlice we don't explicitly write the action creator, it is created automatically.
The solution to this problem is the prepare callback defined in the reducer of the createSlice. This function receives the initial payload from the component (here title, content) and returns the actual payload that will be used by the reducer. So inside it we can put any logic that modifies the action's payload before it is passed to the reducer.

redux-thunk

Typically written in the slice files.

With a plain basic Redux store, you can only do simple synchronous updates by dispatching an action. Middleware extends the store's abilities, and **lets you write async logic that interacts with the store**. As I understand, redux-thunk middleware purpose is to give you the possibility to dispatch a js function instead of an action object. the function will be processed by the middleware, will do the async call and dispatch the necessary action objects.

By itself, a Redux store doesn't know anything about async logic. It only knows how to synchronously dispatch actions, update the state by calling the root reducer function, and notify the UI that something has changed. Any asynchronicity has to happen outside the store.

Thunks

A thunk is a specific kind of Redux function that can contain asynchronous logic (setTimout, Promises and async/await). Thunks are written using two functions:

- An inside thunk function, which gets dispatch and getState as arguments and dispatches an action creator
- The outside creator function, which creates and returns the thunk function

We can use thunks the same way we use a typical Redux action creator

```
const exampleThunkFunction = (dispatch, getState) => {
  const stateBefore = getState()
  console.log(`Counter before: ${stateBefore.counter}`)
  dispatch(increment())
  const stateAfter = getState()
  console.log(`Counter after: ${stateAfter.counter}`)
}
```

Once the thunk middleware has been added to the Redux store, it allows you to pass thunk functions directly to store.dispatch. (so you dispatch a thunk creator instead of an action or an action creator)

```
const logAndAdd = amount => {
  return (dispatch, getState) => {
    const stateBefore = getState()
    console.log(`Counter before: ${stateBefore.counter}`)
    dispatch(incrementByAmount(amount))
    const stateAfter = getState()
    console.log(`Counter after: ${stateAfter.counter}`)
  }
}

store.dispatch(logAndAdd(5))
```

Notice that the same action creator could be written as:

```
const logAndAdd = amount => (dispatch, getState) => {…}
```

A thunk action creator is a function that returns a thunk function. To run the creator you have to dispatch it. A creator like this, is called a thunk and allows us to perform async logic. It can be dispatched like a regular action.

Thunks typically dispatch plain actions using action creators. For consistency with dispatching normal action objects, we typically write thunk functions as thunk action creators, which return the thunk function. Notice that they can take arguments that can be used inside the thunk.

Thunks are typically written in "slice" files. createSlice itself does not have any special support for defining thunks, so you should write them as separate functions in the same slice file. That way, they have access to the plain action creators for that slice, and it's easy to find where the thunk lives.

Notice that the typical logic of async data fetching is a start action that triggers isloading, the request is made and depending on the request result we get a success or failure action. You can have only a success action if you don't care about isloading and failure. But this pattern is provided by default by the createAsyncThunk of the redux-toolkit

Examples

```
// the outside "thunk creator" function
const fetchUserById = userId => {
  // the inside "thunk function"
  return async (dispatch, getState) => {
    try {
      // make an async call in the thunk
      const user = await userAPI.fetchById(userId)
      // dispatch an action when we get the response back
      dispatch(userLoaded(user))
    } catch (err) {
      // If something went wrong, handle it here
    }
  }
}
```

I guess that userAPI.fetchById uses axios or something like it, to do the call.

On a form submit which on submit makes an async call, you can't just simply dispatch a success action object on the submit event. You must dispatch a thunk function that makes the async call and dispatch the actual action on success (when the promise is resolved with success). This is done with thunk.

Dispatch an action when a promise is resolved

Although it is not explicitly visible, this is possible using redux-thunk.

```
import axios from "axios";

import { GET_LEADS } from "./types";

// GET LEADS
export const getLeads = () => dispatch => {
  axios
    .get("/api/leads/")
    .then(res => {
      dispatch({
        type: GET_LEADS,
        payload: res.data
      });
    })
    .catch(err => console.log(err));
};
```

This arrow syntax means

```
var getLeads = function getLeads (){
   return function(dispatch){
      axios.get(…).then(…).catch()
   }
}
```

You should dispatch the thunk function creator.

```
<button onClick={ () => dispatch(getLeads()) }> get leads </button>
```

The previous format is what is used. Why not the following? I think the reason is that an arrow function preserves "this".

```
<button onClick={ dispatch(getLeads() }> get leads </button>
```

In the App component

```
class App extends Component {
  componentDidMount() {
    store.dispatch(loadUser());
  }

  render() {
    return (
      <Provider store={store}>
        <AlertProvider template={AlertTemplate}
        {...alertOptions}>
          <Router>
            <Fragment>
```

We have an action file that makes an axios call and dispatches an action type on success. This function is a thunk. We call it from the main App component's didMount lifecycle method.

Action file

```
export const loadUser = () => (dispatch, getState) => {
   axios
      .get("/api/auth/user", config)
      .then(res => { dispatch({ type:USER_LOADED, payload: res.data}); })
      .catch(err => { console.log(err.data) })
```

createAsyncThunk

In these examples they use a custom client.js app to make the calls. We could replace that with **axios** (Probably with no additional changes).

When we make an API call, we can view its progress as a small state machine that can be in one of four possible states:

- The request hasn't started yet
- The request is in progress
- The request succeeded, and we now have the data we need
- The request failed, and there's probably an error message

We could track that information using some booleans, like isLoading: true, but it's better to track these states as a single enum value. A good pattern for this is to have a state section that looks like:

```
{
  status: 'idle' | 'loading' | 'succeeded' | 'failed',
  error: string | null
}
```

We can use this information to decide what to show in our UI as the request progresses, and also add logic in our reducers to prevent cases like loading data twice.

Notice

In createAsyncThunk the status is idle only before the first time the component makes a call. After the call the status would be "succeeded" or "failed". If the call is made again (without reload the page) the status would switch directly to "loading".

1. Creating the thunk that makes the call

Redux Toolkit's createAsyncThunk API generates thunks that automatically dispatch those "start/success/failure" actions for you.

features/posts/postsSlice

```
import { createSlice, nanoid, createAsyncThunk } from '@reduxjs/toolkit'
import { client } from '../../api/client'

const initialState = {
  posts: [],
  status: 'idle',
  error: null
}

export const fetchPosts = createAsyncThunk('posts/fetchPosts', async () => {
  const response = await client.get('/fakeApi/posts')
  return response.posts
})
```

We create the thunk in the slice file.

createAsyncThunk accepts two arguments:
A string that will be used as the prefix for the generated action types
A "payload creator" callback function that should return a Promise containing some data, or a rejected Promise with an error

The payload creator will usually make an AJAX call of some kind, and can either return the Promise from the AJAX call directly, or extract some data from the API response and return that. We typically write this using the JS async/await syntax, which lets us write functions that use Promises while using standard try/catch logic instead of somePromise.then() chains.

When dispatched, the thunk will:

- dispatch the pending action
- call the payloadCreator callback and wait for the returned promise to settle
- when the promise settles:
  - if the promise resolved successfully, dispatch the fulfilled action with the promise value as action.payload
  - if the promise resolved with a rejectWithValue(*value*) return value, dispatch the rejected action with the *value* passed into action.payload and 'Rejected' as action.error.message
  - if the promise failed and was not handled with rejectWithValue, dispatch the rejected action with a serialized version of the error value as action.error
- Return a fulfilled promise containing the final dispatched action (either the fulfilled or rejected action object)

The thunk's actions:

- posts/fetchPosts/pending

calling dispatch(fetchPosts()), the fetchPosts thunk will first dispatch the pending action type

- posts/fetchPosts/fulfilled

Once the Promise resolves, the fetchPosts thunk takes the response.posts array we returned from the callback, and dispatches this action type

- Posts/fetchPosts/rejected

2. Handling the built in thunk actions

```
export const fetchPosts = createAsyncThunk('posts/fetchPosts', async () => {
  const response = await client.get('/fakeApi/posts')
  return response.posts
})

const postsSlice = createSlice({
  name: 'posts',
  initialState,
  reducers: {
    // omit existing reducers here
  },
  extraReducers: {
    [fetchPosts.pending]: (state, action) => {
      state.status = 'loading'
    },
    [fetchPosts.fulfilled]: (state, action) => {
      state.status = 'succeeded'
      // Add any fetched posts to the array
      state.posts = state.posts.concat(action.payload)
    },
    [fetchPosts.rejected]: (state, action) => {
      state.status = 'failed'
      state.error = action.error.message
    }
  }
})
```

We use the extraReducers field of the createSlice function to handle actions not created by the defined reducers. In this case to handle the actions created by the createAsyncThunk.

I guess that fetchPosts.pending returns "fetchPosts/pending" and combined with the "posts" slice name you get the full action type.

3. Make the async call from a component and show content based on loading state

features/posts/PostsList.js

```
export const PostsList = () => {
  const dispatch = useDispatch()
  const posts = useSelector(selectAllPosts)

  const postStatus = useSelector(state => state.posts.status)
  const error = useSelector(state => state.posts.error)

  useEffect(() => {
    if (postStatus === 'idle') {
      dispatch(fetchPosts())
    }
  }, [postStatus, dispatch])

  let content

  if (postStatus === 'loading') {
    content = <div className="loader">Loading...</div>
  } else if (postStatus === 'succeeded') {
    // Sort posts in reverse chronological order by datetime string
    const orderedPosts = posts
      .slice()
      .sort((a, b) => b.date.localeCompare(a.date))

    content = orderedPosts.map(post => (
      <PostExcerpt key={post.id} post={post} />
    ))
  } else if (postStatus === 'failed') {
    content = <div>{error}</div>
  }

  return (
    <section className="posts-list">
      <h2>Posts</h2>
      {content}
    </section>
  )
}
```

Since we want to fetch this data when <PostsList> mounts, we need to import the React useEffect hook.

It's important that we only try to fetch the list of posts once. If we do it every time the <PostsList> component renders, or is re-created because we've switched between views, we might end up fetching the posts several times. We can use the posts.status enum to help decide if we need to actually start fetching, by selecting that into the component and only starting the fetch if the status is 'idle'

1. Component mounts by the initial page load
2. State posts Status is idle, so fecthPosts async thunk is dispatched.
3. This first dispatches the posts/fetchPosts/pending action and the posts.status becomes loading
4. The component is rendered, showing content based on loading status.
5. When the data is fetched from the server the posts/fetchPosts/fulfilled action is dispatched.
6. State posts Status changes to succeeded and posts state is updated
7. The component is rerendered since both posts and status coming from useSelector hooks, change.
8. It shows content based on succeeded status value.

Tips

axios proposes a specific handling of errors that allows you to easily handle and show the server generated error message. See the axios chapter

```
export const getAlgorithms = createAsyncThunk( typePrefix: 'algorithm/getAlgorithms', payloadCreator: async () => {
  const response = await axios.get( url: 'algorithms')
  console.log('response:', response)
  return response.data
})
```

In this case if there is an error with the axios request (for example CORS related) the async thunk (created by createAsyncThunk) will dispatch a "rejected" action which you can handle on the reducer (for example by filling an error state variable). the error is attached to the ***action.payload.message*** variable returned by the async thunk. If there is a success you access what is returned by the payload creation function of createAsyncThunk with ***action.payload***.

Whatever you return, response.data in this case (the json data returned by the server), it is accessed with action.payload.

Don't do this, use the first approach

```
export const getAlgorithms = createAsyncThunk( typePrefix: 'algorithm/getAlgorithms', payloadCreator: async () => {
  try {
    const response = await axios.get( url: 'algorithmsa')
    console.log('response:', response)
    return response.data
  } catch (error) {
    console.error(error)
  }
})
```

If you use a try catch block though, then if there is an error with the axios request the error will be caught and no "rejected" action will be dispatched. A "succeeded" action will be dispatched instead. So this means that you need to write your logic with this in mind. Don't do this, use the first approach and let createAsyncThunk handle the process.

Sending data asynchronously to the backend

Create the thunk that makes the call and then handle the fulfilled action

features/posts/postsSlice.js

```
export const addNewPost = createAsyncThunk(
  'posts/addNewPost',
  // The payload creator receives the partial `{title, content, user}` object
  async initialPost => {
    // We send the initial data to the fake API server
    const response = await client.post('/fakeApi/posts', { post: initialPost })
    // The response includes the complete post object, including unique ID
    return response.post
  }
)

const postsSlice = createSlice({
  name: 'posts',
  initialState,
  reducers: {
    // The existing `postAdded` reducer and prepare callback were deleted
    reactionAdded(state, action) {}, // omit logic
    postUpdated(state, action) {} // omit logic
  },
  extraReducers: {
    // omit posts loading reducers
    [addNewPost.fulfilled]: (state, action) => {
      // We can directly add the new post object to our posts array
      state.posts.push(action.payload)
    }
  }
})
```

Notice that we use the returned post, which contains any backend generated fields like the id.

Making the call from the component (and show content based on loading state)

features/posts/AddPostForm.js

```
import React, { useState } from 'react'
import { useDispatch, useSelector } from 'react-redux'
import { unwrapResult } from '@reduxjs/toolkit'

import { addNewPost } from './postsSlice'

export const AddPostForm = () => {
  const [title, setTitle] = useState('')
  const [content, setContent] = useState('')
  const [userId, setUserId] = useState('')
  const [addRequestStatus, setAddRequestStatus] = useState('idle')

  // omit useSelectors and change handlers

  const canSave =
    [title, content, userId].every(Boolean) && addRequestStatus === 'idle'

  const onSavePostClicked = async () => {
    if (canSave) {
      try {
        setAddRequestStatus('pending')
        const resultAction = await dispatch(
          addNewPost({ title, content, user: userId })
        )
        unwrapResult(resultAction)
        setTitle('')
        setContent('')
        setUserId('')
      } catch (err) {
        console.error('Failed to save the post: ', err)
      } finally {
        setAddRequestStatus('idle')
      }
    }
  }

  // omit rendering logic
}
```

In this example they don't use the global loading enum in the redux store (just inside this component as a local state).

In the case of using a global loading state, we don't use neither a try - catch block nor a .then chain for the call thunk. We just dispatch the (created with createAsyncThunk) thunk. This will dispatch its complete or fail actions to which we listen to with our reducers and handle them there by modifying the global "loading" state accordingly. Then in the component we show content based on the loading state.

In the second case we have no global loading state (You should decide if it would be in the global state or not based on the same criteria as with any other variable). We have only one reducer that listens to the thunk's fulfilled action and modifies the items state. Since there is no global loading state we need another way to show content based on success or failure of the call. This is achieved with a try-catch block using the unwrapResult function provided by the redux toolkit. In this case the status has 2 states: idle and pending. We could add failed and succeeded in the try catch block if we prefer.

Notice: The thunk created by createAsyncThunk only accepts one argument. This argument becomes the first argument of the payload creation callback. In this example the argument is an object *{title, content, user:userId}* which becomes the *InitialPost* object in the addNewPost payload creation callback.

(Notice the canSave pattern, every(Boolean))

When we call dispatch(addNewPost()), the async thunk returns a Promise from dispatch. We can await that promise here to know when the thunk has finished its request. But, we don't yet know if that request succeeded or failed.

unwrapResult()

createAsyncThunk handles any errors internally, so that we don't see any messages about "rejected Promises" in our logs. It then returns the final action it dispatched: either the fulfilled action if it succeeded, or the rejected action if it failed. Redux Toolkit has a utility function called unwrapResult that will return either the actual action.payload value from a fulfilled action, or throw an error if it's the rejected action. This lets us handle success and failure in the component using normal try/catch logic (without the need to use a global loading enum state).

payload_creation_callback(*whatever passed when calling the thunk* , *the special thunkAPI object*)

```
export const fetchNotifications = createAsyncThunk(
  'notifications/fetchNotifications',
  async (_, { getState }) => {
    const allNotifications = selectAllNotifications(getState())
    const [latestNotification] = allNotifications
    const latestTimestamp = latestNotification ? latestNotification.date : ''
    const response = await client.get(
      `/fakeApi/notifications?since=${latestTimestamp}`
    )
    return response.notifications
  }
)
```

For createAsyncThunk thunks, you can only pass in one argument, and whatever we pass in becomes the first argument of the payload creation callback. The second argument to our payload creator is a thunkAPI object containing several useful functions and pieces of information: dispatch and getState, extra, requestId, signal, rejectWithValue.

Here we get the getState object form the thunkAPI object using object destructuring

Notice: If you're writing a thunk by hand instead of using createAsyncThunk, the thunk function will get (dispatch, getState) as separate arguments, instead of putting them together in one object.

Data fetching without Redux

```
function Example() {
  const [data, dataSet] = useState<any>(null)

  useEffect(() => {
    async function fetchMyAPI() {
      let response = await fetch('api/data')
      response = await response.json()
      dataSet(response)
    }

    fetchMyAPI()
  }, [])

  return <div>{JSON.stringify(data)}</div>
}
```

Notice how much simpler it is…

Have in mind though that data fetching will be performed with Suspense in the future.

“Longer term we'll discourage this pattern because it encourages race conditions. Such as — anything could happen between your call starts and ends, and you could have gotten new props. Instead, we'll recommend Suspense for data fetching which will look more like”

```
const response = MyAPIResource.read();
```

Performance optimization

Solutions to performance problems can be

1. Memoization (reselect from redux-toolkit)
2. Normalization (createEntityAdapter from redux-toolkit)

Some issues

- An action is dispatched two times

features/notifications/NotificationsList.js

```
import React, { useEffect } from 'react'
import { useSelector, useDispatch } from 'react-redux'
import { formatDistanceToNow, parseISO } from 'date-fns'
import classnames from 'classnames'

import { selectAllUsers } from '../users/usersSlice'

import {
  selectAllNotifications,
  allNotificationsRead
} from './notificationsSlice'

export const NotificationsList = () => {
  const dispatch = useDispatch()
  const notifications = useSelector(selectAllNotifications)
  const users = useSelector(selectAllUsers)

  useEffect(() => {
    dispatch(allNotificationsRead())
  })

  const renderedNotifications = notifications.map(notification => {
    const date = parseISO(notification.date)
    const timeAgo = formatDistanceToNow(date)
    const user = users.find(user => user.id === notification.user) || {
      name: 'Unknown User'
    }

    const notificationClassname = classnames('notification', {
      new: notification.isNew
    })

    return (
      <div key={notification.id} className={notificationClassname}>
        <div>
          <b>{user.name}</b> {notification.message}
        </div>
        <div title={notification.date}>
          <i>{timeAgo} ago</i>
        </div>
      </div>
    )
  })

  return (
    <section className="notificationsList">
      <h2>Notifications</h2>
      {renderedNotifications}
    </section>
  )
}
```

Notice: have this structure of rendering items in mind

Notice: Our fake API is already sending back the notification entries with isNew and read fields, so we can use those in our code.

This works but:

This works, but actually has a slightly surprising bit of behavior. Any time there are new notifications (either because we've just switched to this tab, or we've fetched some new notifications from the API), you'll actually see two "notifications/allNotificationsRead" actions dispatched. Why is that?

Let's say we have fetched some notifications while looking at the <PostsList>, and then click the "Notifications" tab. The <NotificationsList> component will mount, and the useEffect callback will run after that first render and dispatch allNotificationsRead. Our notificationsSlice will handle that by updating the notification entries in the store. This creates a new state.notifications array containing the immutably-updated entries, which forces our component to render again because it sees a new array returned from the useSelector, and the useEffect hook runs again and dispatches allNotificationsRead a second time. The reducer runs again, but this time no data changes, so the component doesn't re-render.

- A component is re-rendered without data change

If we press the get-notifications button, the navbar will be re-rendered to show the number of the newly fetched notifications but also the UserPage component is rerendered which is something we didn't expect. (You can see what components are rendered after an action by using the react devTools profiler).

```
export const UserPage = ({ match }) => {
  const { userId } = match.params

  const user = useSelector(state => selectUserById(state, userId))

  const postsForUser = useSelector(state => {
    const allPosts = selectAllPosts(state)
    return allPosts.filter(post => post.user === userId)
  })

  // omit rendering logic
}
```

We know that useSelector will re-run every time an action is dispatched, and that it forces the component to re-render if we return a new reference value.
We're calling filter() inside of our useSelector hook, so that we only return the list of posts that belong to this user. **Unfortunately, this means that useSelector always returns a new array reference, and so our component will re-render after every action even if the posts data hasn't changed**!.

The solution is memoization

Memoization

What we really need is a way to only calculate the new filtered array if either state.posts or userId have changed. If they haven't changed, we want to return the same filtered array reference as the last time. This idea is called "memoization". We want to save a previous set of inputs and the calculated result, and if the inputs are the same, return the previous result instead of recalculating it again.

Memoized selectors are a valuable tool for improving performance in a React+Redux application, because they can help us avoid unnecessary re-renders, and also avoid doing potentially complex or expensive calculations if the input data hasn't changed.

Reselect package

Reselect is a library for creating memoized selector functions, and was specifically designed to be used with Redux. It has a *createSelector* function that generates memoized selectors that will only recalculate results when the inputs change. Redux Toolkit exports the createSelector function, so we already have it available.

features/posts/postsSlice.js

```
import { createSlice, createAsyncThunk, createSelector } from '@reduxjs/toolkit'

// omit slice logic

export const selectAllPosts = state => state.posts.posts

export const selectPostById = (state, postId) =>
  state.posts.posts.find(post => post.id === postId)

export const selectPostsByUser = createSelector(
  [selectAllPosts, (state, userId) => userId],
  (posts, userId) => posts.filter(post => post.user === userId)
)
```

createSelector takes one or more "input selector" functions as argument, plus an "output selector" function.
When we call selectPostsByUser(state, userId), createSelector will pass all of the arguments into each of our input selectors. Whatever those input selectors return becomes the arguments for the output selector.

In this case the first input selector returns posts and the second userId. These two arguments are passed to the output selector.

```
export const UserPage = ({ match }) => {
  const { userId } = match.params

  const user = useSelector(state => selectUserById(state, userId))

  const postsForUser = useSelector(state => selectPostsByUser(state, userId))

  // omit rendering logic
}
```

Calling the selector from the component

- Another problem: A component with many children is rerendered without data change as before. Children are rerendered too.

Data returned by a useSelector is not actually changed, but a new reference is returned. This is the origin of the problem. React's default behavior is that when a parent component renders, React will recursively render all child components inside of it!.

Possible solutions:

- Use React.memo()

```
let PostExcerpt = ({ post }) => {
  // omit logic
}

PostExcerpt = React.memo(PostExcerpt)
```

We could wrap the <PostExcerpt> component in React.memo(), which will ensure that the component inside of it only re-renders if the props have actually changed. This will actually work quite well

- Another option is to rewrite <PostsList> so that it only selects a list of post IDs from the store instead of the entire posts array, and rewrite <PostExcerpt> so that it receives a postId prop and calls useSelector to read the post object it needs. Unfortunately, this gets tricky because we also need to have all our posts sorted by date and rendered in the right order.
- The last option is to find some way to have our reducer keep a separate array of IDs for all the posts, and only modify that array when posts are added or removed, and do the same rewrite of <PostsList> and <PostExcerpt>. This way, <PostsList> only needs to re-render when that IDs array changes. Conveniently, Redux Toolkit has a createEntityAdapter function that will help us do just that.

Normalization

- You've seen that a lot of our logic has been looking up items by their ID field. Since we've been storing our data in arrays, that means we have to loop over all the items in the array using array.find() until we find the item with the ID we're looking for. Realistically, this doesn't take very long, but if we had arrays with hundreds or thousands of items inside, looking through the entire array to find one item becomes wasted effort. What we need is a way to look up a single item based on its ID, directly, without having to check all the other items. This can be achieved with data "normalization". **"Normalization" means no duplication of data, and keeping items stored in a lookup table by item ID**.

<pre>{ users: { ids: ["user1", "user2", "user3"], entities: { "user1": {id: "user1", firstName, lastName}, "user2": {id: "user2", firstName, lastName}, "user3": {id: "user3", firstName, lastName}, } } }</pre>	"Normalized state" means that: ● We only have one copy of each particular piece of data in our state, so there's no duplication ● Data that has been normalized is kept in a lookup table, where the item IDs are the keys, and the items themselves are the values. ● There may also be an array of all of the IDs for a particular item type (This is the shape of state that **createEntityAdapter** creates)
<pre>const userId = 'user2' const userObject = state.users.entities[userId]</pre>	This makes it easy to find a particular user object by its ID, without having to loop through all the other user objects in an array:

Redux Toolkit's **createEntityAdapter** API provides a standardized way to store your data in a slice by taking a collection of items and putting them into the shape of { ids: [], entities: {} }. Along with this predefined state shape, it generates a set of reducer functions and selectors that know how to work with that data.

This has several benefits:

- We don't have to write the code to manage the normalization ourselves
- createEntityAdapter's pre-built reducer functions handle common cases like "add all these items", "update one item", or "remove multiple items". setAll, addMany, upsertOne, removeMany
- createEntityAdapter can keep the ID array in a sorted order based on the contents of the items, and will only update that array if items are added / removed or the sorting order changes.

Prebuilt reducers for common cases, like setAll, addMany, upsertOne, and removeMany

todo

createEntityAdapter with createAsyncThunk

Error and Message handling

Notistack is a good library for this purpose.

Error handling

It is very useful to have any error, for example axios call errors, to be presented in the UI through a dedicated component. This means that the errors must be handled as any other piece of data that comes from the server. They must be added to the global redux state, and passed to the components that need them.

react-alert

```
  axios
    .post("/api/leads/", lead)
    .then(res => {
      dispatch({
        type: ADD_LEAD,
        payload: res.data
      });
    })
    .catch(err => {
      const errors = {
        msg: err.response.data,
        status: err.response.status
      }
      dispatch({
        type: GET_ERRORS,
        payload: errors
      });
    });
};
```

It is a nice 3rd party npm module that provides a nice alert react component. It offers a Provider (as react-redux does) with which we wrap our main App component. Notice that the outer most Provider must be the react-redux one. React router would be inside the AlertProvider.

The err.response.data is an object created by the json response that contains the error messages.

After the error is received we dispatch the action

```
// RETURN ERRORS
export const returnErrors = (msg, status) => {
  return {
    type: GET_ERRORS,
    payload: { msg, status }
  };
};
```

```
axios
  .post("/api/leads/", lead)
  .then(res => {
    dispatch(createMessage({ addLead: "Lead Added" })
    );
    dispatch({
      type: ADD_LEAD,
      payload: res.data
    });
  })
  .catch(err => dispatch(returnErrors
  (err.response.data, err.response.status)));
};
```

Actually instead of externally dispatching an action in every axios call, its better to create an action that returns the GET_ERROR action type and dispatch it whenever we want to get errors.

Message handling

You can use the same alert component to show custom messages instead of errors. To do so, you can create a message reducer that handles one action, the CREATE_MSG and trigger the action on buttons pressed or whenever you want. This action is caught by the messages reducer which modifies the message state, which is passed as props to the alert component and on change the component is rerendered and shows the message in the alert window. (the GET_MESSAGES action is not needed in this implementation)

Messages reducer

Message actions

```
import { CREATE_MESSAGE } from "./types";

// CREATE MESSAGE
export const createMessage = msg => {
  return {
    type: CREATE_MESSAGE,
    payload: msg
  };
};
```

```
import { GET_MESSAGES, CREATE_MESSAGE } from
"../actions/types";

const initialState = {};

export default function(state = initialState, action) {
  switch (action.type) {
    case GET_MESSAGES:
      return action.payload;
    case CREATE_MESSAGE:
      return (state = action.payload);
    default:
      return state;
  }
}
```

Notice that here we just return the action object and we dispatch it manually after the response from the server is received (after pressing a button for example)

```
// DELETE LEAD
export const deleteLead = id => dispatch => {
  axios
    .delete(`/api/leads/${id}/`)
    .then(res => {
      dispatch(createMessage({ deleteLead: "Lead
      Deleted" }));
      dispatch({
        type: DELETE_LEAD,
        payload: id
      });
    })
    .catch(err => console.log(err));
};
```

```
componentDidUpdate(prevProps) {
  const { error, alert, message } = this.props;
  if (error !== prevProps.error) {
    if (error.msg.name) alert.error(`Name: $
    {error.msg.name.join()}`);
    if (error.msg.email) alert.error(`Email: $
    {error.msg.email.join()}`);
    if (error.msg.message)
      alert.error(`Message: ${error.msg.message.join
      ()}`);
  }

  if (message !== prevProps.message) {
    if (message.deleteLead) alert.success
    (message.deleteLead);
  }
}
```

The messages reducer updates the state. This state is passed as props to the alert component which will be rerendered and will show the message in the alert window.

Essential tips

TIP

- **Redux is a library for managing global application state**
  - Redux is typically used with the React-Redux library for integrating Redux and React together
  - Redux Toolkit is the recommended way to write Redux logic
- **Redux uses a "one-way data flow" app structure**
  - State describes the condition of the app at a point in time, and UI renders based on that state
  - When something happens in the app:
    - The UI dispatches an action
    - The store runs the reducers, and the state is updated based on what occurred
    - The store notifies the UI that the state has changed
  - The UI re-renders based on the new state
- **Redux uses several types of code**
  - *Actions* are plain objects with a `type` field, and describe "what happened" in the app
  - *Reducers* are functions that calculate a new state value based on previous state + an action
  - A Redux *store* runs the root reducer whenever an action is *dispatched*

TIP

- **We can create a Redux store using the Redux Toolkit `configureStore` API**
  - `configureStore` accepts a `reducer` function as a named argument
  - `configureStore` automatically sets up the store with good default settings
- **Redux logic is typically organized into files called "slices"**
  - A "slice" contains the reducer logic and actions related to a specific feature / section of the Redux state
  - Redux Toolkit's `createSlice` API generates action creators and action types for each individual reducer function you provide
- **Redux reducers must follow specific rules**
  - Should only calculate a new state value based on the `state` and `action` arguments
  - Must make *immutable updates* by copying the existing state
  - Cannot contain any asynchronous logic or other "side effects"
  - Redux Toolkit's `createSlice` API uses Immer to allow "mutating" immutable updates
- **Async logic is typically written in special functions called "thunks"**
  - Thunks receive `dispatch` and `getState` as arguments
  - Redux Toolkit enables the `redux-thunk` middleware by default
- **React-Redux allows React components to interact with a Redux store**
  - Wrapping the app with `<Provider store={store}>` enables all components to use the store
  - Global state should go in the Redux store, local state should stay in React components

TIP

- **Redux state is updated by "reducer functions"**:
  - Reducers always calculate a new state *immutably*, by copying existing state values and modifying the copies with the new data
  - The Redux Toolkit `createSlice` function generates "slice reducer" functions for you, and lets you write "mutating" code that is turned into safe immutable updates
  - Those slice reducer functions are added to the `reducer` field in `configureStore`, and that defines the data and state field names inside the Redux store
- **React components read data from the store with the `useSelector` hook**
  - Selector functions receive the whole `state` object, and should return a value
  - Selectors will re-run whenever the Redux store is updated, and if the data they return has changed, the component will re-render
- **React components dispatch actions to update the store using the `useDispatch` hook**
  - `createSlice` will generate action creator functions for each reducer we add to a slice
  - Call `dispatch(someActionCreator())` in a component to dispatch an action
  - Reducers will run, check to see if this action is relevant, and return new state if appropriate
  - Temporary data like form input values should be kept as React component state. Dispatch a Redux action to update the store when the user is done with the form.

TIP

- **Any React component can use data from the Redux store as needed**
  - Any component can read any data that is in the Redux store
  - Multiple components can read the same data, even at the same time
  - Components should extract the smallest amount of data they need to render themselves
  - Components can combine values from props, state, and the Redux store to determine what UI they need to render. They can read multiple pieces of data from the store, and reshape the data as needed for display.
  - Any component can dispatch actions to cause state updates
- **Redux action creators can prepare action objects with the right contents**
  - `createSlice` and `createAction` can accept a "prepare callback" that returns the action payload
  - Unique IDs and other random values should be put in the action, not calculated in the reducer
- **Reducers should contain the actual state update logic**
  - Reducers can contain whatever logic is needed to calculate the next state
  - Action objects should contain just enough info to describe what happened

TIP

- **You can write reusable "selector" functions to encapsulate reading values from the Redux state**
  - Selectors are functions that get the Redux `state` as an argument, and return some data
- **Redux uses plugins called "middleware" to enable async logic**
  - The standard async middleware is called `redux-thunk`, which is included in Redux Toolkit
  - Thunk functions receive `dispatch` and `getState` as arguments, and can use those as part of async logic
- **You can dispatch additional actions to help track the loading status of an API call**
  - The typical pattern is dispatching a "pending" action before the call, then either a "success" containing the data or a "failure" action containing the error
  - Loading state should usually be stored as an enum, like `'idle' | 'loading' | 'succeeded' | 'failed'`
- **Redux Toolkit has a `createAsyncThunk` API that dispatches these actions for you**
  - `createAsyncThunk` accepts a "payload creator" callback that should return a `Promise`, and generates `pending/fulfilled/rejected` action types automatically
  - Generated action creators like `fetchPosts` dispatch those actions based on the `Promise` you return
  - You can listen for these action types in `createSlice` using the `extraReducers` field, and update the state in reducers based on those actions.
  - Action creators can be used to automatically fill in the keys of the `extraReducers` object so the slice knows what actions to listen for.

TIP

- **Memoized selector functions can be used to optimize performance**
  - Redux Toolkit re-exports the `createSelector` function from Reselect, which generates memoized selectors
  - Memoized selectors will only recalculate the results if the input selectors return new values
  - Memoization can skip expensive calculations, and ensure the same result references are returned
- **There are multiple patterns you can use to optimize React component rendering with Redux**
  - Avoid creating new object/array references inside of `useSelector` - those will cause unnecessary re-renders
  - Memoized selector functions can be passed to `useSelector` to optimize rendering
  - `useSelector` can accept an alternate comparison function like `shallowEqual` instead of reference equality
  - Components can be wrapped in `React.memo()` to only re-render if their props change
  - List rendering can be optimized by having list parent components read just an array of item IDs, passing the IDs to list item children, and retrieving items by ID in the children
- **Normalized state structure is a recommended approach for storing items**
  - "Normalization" means no duplication of data, and keeping items stored in a lookup table by item ID
  - Normalized state shape usually looks like `{ids: [], entities: {}}`
- **Redux Toolkit's `createEntityAdapter` API helps manage normalized data in a slice**
  - Item IDs can be kept in sorted order by passing in a `sortComparer` option
  - The adapter object includes:
    - `adapter.getInitialState`, which can accept additional state fields like loading state
    - Prebuilt reducers for common cases, like `setAll`, `addMany`, `upsertOne`, and `removeMany`
    - `adapter.getSelectors`, which generates selectors like `selectAll` and `selectById`

Notes

Async useEffect

```
const GetAlgorithms = () => {
  const getAllStatus = useSelector(state => getAllStatusSelector(state))
  const dispatch = useDispatch()

  useEffect( effect: () => {
    // since useEffect can't be an async function you can declare one inside it and call it later
    async function getItems(){
      await dispatch(getAlgorithmsThunk())
    }
    // Get items only on page load, not every time the component mounts.
    // This way you avoid the calls on back/forward operations to the component
    if (getAllStatus === 'idle'){
      const promise = getItems()
    }
  }, deps: [getAllStatus, dispatch]) // I'm not sure if dispatch needs to be in the dependencies array

  return null
}
```

useEffect can't be an async function. But you can of course call an async function from within the useEffect function.

Providers Pattern

Index.js	App.js	Notice that the Router provider wraps the contents of the app component in the app file while the react-redux Provider wraps the App component in the index file.

```
ReactDOM.render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>,
  document.getElementById('root')
)
```

```
function App() {
  return (
    <Router>
      <Navbar />
      <div className="App">
        <Switch>
          <Route
            exact
```

nanoid

Notice: they use nanoid here because they don't really save data to a back end service. If we did then the id would be generated by the backend and we would get the newly created item back from which we will use its id.

We could write some code that would figure out what the next incrementing ID number should be, but it would be better if we generated a random unique ID instead. Redux Toolkit has a nanoid function we can use for that.

They use it on item-created action. I don't like this idea. I would prefer to use the real database id of the created item, so that the id of the item in the database and the id of the item in the redux store are the same. With naonid the ids would be different. Although the difference will be corrected when there is a got-items action (items were retrieved from the database).

```
import { nanoid } from '@reduxjs/toolkit'
```

```
const onSavePostClicked = () => {
  if (title && content) {
    dispatch(
      postAdded({
        id: nanoid(),
        title,
        content
      })
    )

    setTitle('')
    setContent('')
  }
}
```

Notice: If an action needs to contain a unique ID or some other random value, always generate that first and put it in the action object. Reducers should never calculate random values, because that makes the results unpredictable.

A select button

```
export const AddPostForm = () => {
  const [title, setTitle] = useState('')
  const [content, setContent] = useState('')
  const [userId, setUserId] = useState('')

  const dispatch = useDispatch()

  const users = useSelector(state => state.users)

  const onTitleChanged = e => setTitle(e.target.value)
  const onContentChanged = e => setContent(e.target.valu
  const onAuthorChanged = e => setUserId(e.target.value)

  const onSavePostClicked = () => {
    if (title && content) {
      dispatch(postAdded(title, content, userId))
      setTitle('')
      setContent('')
    }
  }
```

```
const usersOptions = users.map(user => (
  <option key={user.id} value={user.id}>
    {user.name}
  </option>
))
```

```
    <label htmlFor="postAuthor">Author:</label>
    <select id="postAuthor" value={userId} onChange={onAuthorChanged}>
      <option value=""></option>
      {usersOptions}
    </select>
```

Handling dates

features/posts/TimeAgo.js

```
import React from 'react'
import { parseISO, formatDistanceToNow } from 'date-fns'

export const TimeAgo = ({ timestamp }) => {
  let timeAgo = ''
  if (timestamp) {
    const date = parseISO(timestamp)
    const timePeriod = formatDistanceToNow(date)
    timeAgo = `${timePeriod} ago`
  }

  return (
    <span title={timestamp}>
        <i>{timeAgo}</i>
    </span>
  )
}
```

A TimeAgo component

Notice that the timestamp (passed as prop) is a string.
new Date().toISOString()

date-fns
`import { sub } from 'date-fns'`
`date: sub(new Date(), { minutes: 10 }).toISOString()`
subtract minutes from this Date object

```
if (postStatus === 'loading') {
  content = <div className="loader">Loading...</div>
} else if (postStatus === 'succeeded') {
  // Sort posts in reverse chronological order by datetime string
  const orderedPosts = posts
    .slice()
    .sort((a, b) => b.date.localeCompare(a.date))

  content = orderedPosts.map(post => (
    <PostExcerpt key={post.id} post={post} />
  ))
} else if (postStatus === 'failed') {
  content = <div>{error}</div>
}
```

Sort by chronological order

Emoji

```jsx
import { useDispatch } from 'react-redux'

import { reactionAdded } from './postsSlice'

const reactionEmoji = {
  thumbsUp: '👍',
  hooray: '🎉',
  heart: '❤️',
  rocket: '🚀',
  eyes: '👀'
}

export const ReactionButtons = ({ post }) => {
  const dispatch = useDispatch()

  const reactionButtons = Object.entries(reactionEmoji).map(([name, emoji]) => {
    return (
      <button
        key={name}
        type="button"
        className="muted-button reaction-button"
        onClick={() =>
          dispatch(reactionAdded({ postId: post.id, reaction: name }))
        }
      >
        {emoji} {post.reactions[name]}
      </button>
    )
  })

  return <div>{reactionButtons}</div>
}
```

Major actions

Getting items

See fetchPosts in Making async calls chapter

features/posts/PostsList.js

```jsx
import React from 'react'
import { useSelector } from 'react-redux'

export const PostsList = () => {
  const posts = useSelector(state => state.posts)

  const renderedPosts = posts.map(post => (
    <article className="post-excerpt" key={post.id}>
      <h3>{post.title}</h3>
      <p>{post.content.substring(0, 100)}</p>
    </article>
  ))

  return (
    <section>
      <h2>Posts</h2>
      {renderedPosts}
    </section>
  )
}
```

Items list example

Creating an item

A more generic term is “posting data asynchronously to the server”. see See addPost in Making async calls chapter

Editing an item

What about the actual post request to the server? Where is it?

You are not doing the item editing like it is described in this example. You do it with createAsyncThunk function (similarly with creating an item) where an updatePost() async thunk automatically dispatches the post/updatePost/pending, post/updatePost/fulfilled and post/updatePost/rejected actions. The updatePost async thunk is dispatched by the onclick handler. The useHistory.push action described in this example takes place in the state.posts.postAddedStatus = succeeded case (which is set by the post/updatePost/fulfilled action)

features/posts/EditPostForm.js

```
import React, { useState } from 'react'
import { useDispatch, useSelector } from 'react-redux'
import { useHistory } from 'react-router-dom'

import { postUpdated } from './postsSlice'

export const EditPostForm = ({ match }) => {
  const { postId } = match.params

  const post = useSelector(state =>
    state.posts.find(post => post.id === postId)
  )

  const [title, setTitle] = useState(post.title)
  const [content, setContent] = useState(post.content)

  const dispatch = useDispatch()
  const history = useHistory()

  const onTitleChanged = e => setTitle(e.target.value)
  const onContentChanged = e => setContent(e.target.value)

  const onSavePostClicked = () => {
    if (title && content) {
      dispatch(postUpdated({ id: postId, title, content }))
      history.push(`/posts/${postId}`)
    }
  }
```

Continues ...

...continuing

```
  return (
    <section>
      <h2>Edit Post</h2>
      <form>
        <label htmlFor="postTitle">Post Title:</label>
        <input
          type="text"
          id="postTitle"
          name="postTitle"
          placeholder="What's on your mind?"
          value={title}
          onChange={onTitleChanged}
        />
        <label htmlFor="postContent">Content:</label>
        <textarea
          id="postContent"
          name="postContent"
          value={content}
          onChange={onContentChanged}
        />
      </form>
      <button type="button" onClick={onSavePostClicked}>
        Save Post
      </button>
    </section>
  )
}
```

Notice the ***useHistory*** hook. They are using it to externally change the url after the item has been edited to redirect to another page.
Important: in react router v6 it is recommended to use the ***useNavigate*** hook instead. See the relative chapter.

Misc

Dispatch actions outside of react components

In order to dispatch and action from outside of the scope of React.Component you need to get the store instance and call dispatch on it

When to store something in the global store

In a React + Redux app, your global state should go in the Redux store, and your local state should stay in React components. If you're not sure where to put something, here are some common rules of thumb for determining what kind of data should be put into Redux:

- Do other parts of the application care about this data?
- Do you need to be able to create further derived data based on this original data?
- Is the same data being used to drive multiple components?
- Is there value to you in being able to restore this state to a given point in time (ie, time travel debugging)?
- Do you want to cache the data (ie, use what's in state if it's already there instead of re-requesting it)?
- Do you want to keep this data consistent while hot-reloading UI components (which may lose their internal state when swapped)?

Typical workflow

Get items

We call getLeads when the component mounts, the data is fetched with an axios call, the action is dispatched, the reducer catches it, it gets the fetched data attached to the action as payload, updates the redux state with it, passes the particular state attribute (that refers to the leads list) to the component as a prop (with the mapStateToProps function and propTypes). Since the prop is modified, the component is rerendered, modifying the UI that now reflects the new redux state that reflects the backend state.

(So the backend data changes first, and then this change is eventually reflected to the UI by dispatching an action)

Delete an item

We call deleteItem(id) when a button is pressed, the axios call deletes the item in the server, the action is dispatched, the items reducer catches it, it gets the deleted id from the action (was defined as payload), modifies the redux state with the remaining items (excluding the deleted one), passes the particular state attribute to the component as prop. Since the prop is modified, the component is rerendered modifying the UI that now reflects the new redux state that reflects the backend state.

action

```
// DELETE LEAD
export const deleteLead = id => dispatch => {
  axios
    .delete(`/api/leads/${id}/`)
    .then(res => {
      dispatch({
        type: DELETE_LEAD,
        payload: id
      });
    })
    .catch(err => console.log(err));
};
```

reducer

```
export default function(state = initialState, action) {
  switch (action.type) {
    case GET_LEADS:
      return {
        ...state,
        leads: action.payload
      };
    case DELETE_LEAD:
      return {
        ...state,
        leads: state.leads.filter(lead => lead.id !==
        action.payload)
      };
    default:
      return state;
  }
}
```

Button

```
<td>
  <button
    onClick={this.props.deleteLead.bind
    (this, lead.id)}
    className="btn btn-danger btn-sm"
  >
    {" "}
    Delete
  </button>
```

Notice that he doesn't simply do

```
onClick = { this.props.deletetLead(lead.id) }
```

He binds the call instead. I'm not sure why. Maybe it is because it is inside a table that renders the leads list which is fetched asynchronously.

```
// ADD LEAD
export const addLead = (lead) => dispatch => {
  axios
    .post("/api/leads/", lead)
    .then(res => {
      dispatch({
        type: ADD_LEAD,
        payload: res.data
      });
    })
    .catch(err => console.log(err));
};
```

```
export class Form extends Component {
  state = {
    name: "",
    email: "",
    message: ""
  };

  static propTypes = {
    addLead: PropTypes.func.isRequired
  }

  onChange = e => this.setState({ [e.target.name]:
  e.target.value });

  onSubmit = e => {
    e.preventDefault();
    const { name, email, message } = this.state;
    const lead = { name, email, message };
    this.props.addLead(lead);
  };

  render() {
```

Redux state and localstorage

The auth token is stored in local storage too. How is this in sync with redux state? We must do it externally?

Notice how the initial state gets its value from local storage. On logout the token is removed from local storage and the state is externally set to null, so we must explicitly enforce synchronization.

```
const initialState = {
  token: localStorage.getItem("token"),
  isAuthenticated: null,
  isLoading: false,
  user: null
};
```

```
case LOGIN_SUCCESS:
case REGISTER_SUCCESS:
  localStorage.setItem("token", action.payload.token);
  return {
    ...state,
    ...action.payload,
    isAuthenticated: true,
    isLoading: false
  };
case AUTH_ERROR:
case LOGIN_FAIL:
case LOGOUT_SUCCESS:
  localStorage.removeItem("token");
  return {
```

```
case AUTH_ERROR:
case LOGIN_FAIL:
case LOGOUT_SUCCESS:
  localStorage.removeItem("token");
  return {
    ...state,
    token: null,
    user: null,
    isAuthenticated: false,
    isLoading: false
  };
```

React hooks vs Redux

Advantages of storing the state in Redux:

1. You can access and modify it globally
2. It persists even after your component is unmounted

Advantages of storing the state in the component:

1. You can have multiple components with different values in the state, which may be something you want
2. ...Or you could even have multiple hooks of the same type in one component!
3. You don't need to switch between files. Depending on how your code is organized, Redux can be split into 3 files + 1 file for the component which uses it - while this can help keep your code well-structured for complex use cases, it can be an overkill for keeping track of a simple state. Having to switch between multiple files to work on one component can reduce your productivity (I don't like having to keep track of 4 tabs in my IDE for every feature I work on).
4. (Also, hooks are new and cool.)

React context vs redux

If you're only using Redux to avoid passing down props, context could replace Redux - but then you probably didn't need Redux in the first place. Context also doesn't give you anything like the Redux DevTools, the ability to trace your state updates, middleware to add centralized application logic, and other powerful capabilities that Redux enables.

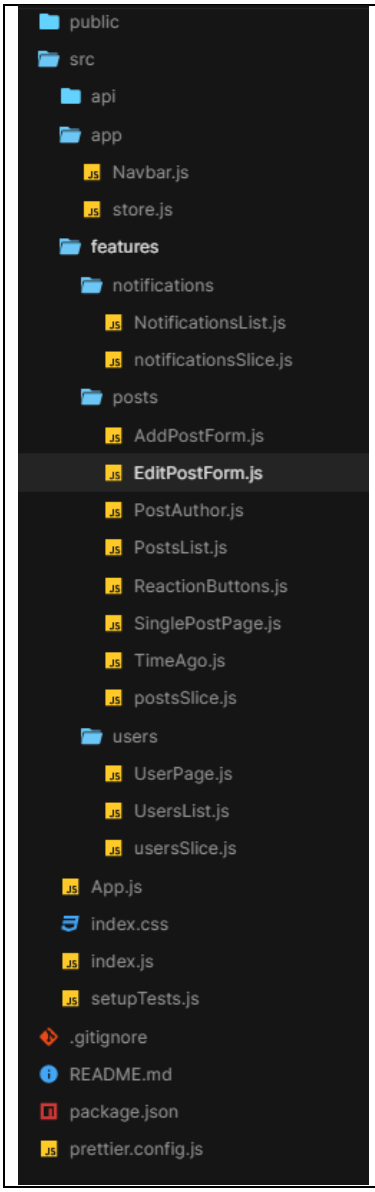
React Redux uses context internally but it doesn't expose this fact in the public API. So you should feel much safer using context via React Redux than directly because if it changes, the burden of updating the code will be on React Redux and not you.

redux-devtools-extension

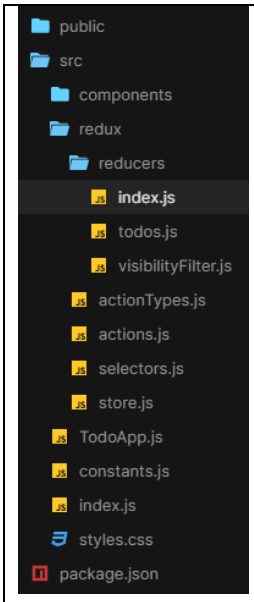
```
const store = createStore(
    allReducer,
    window.__REDUX_DEVTOOLS_EXTENSION__ && window.__REDUX_DEVTOOLS_EXTENSION__()
);
```

A very useful browser extension. To use it install it and add this line to your store.

Project structure



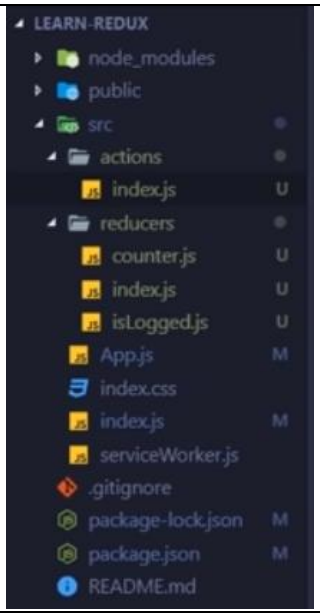
From the redux official website.
they organize the application by features (not by actions, components, reducers etc). Each feature folder contains the features components and the redux slice.



The left example from the react-redux tutorial, has a distinct redux folder. Notice that in this example there is a distinct selectors.js file that selects/extracts data from the store. It contains our functions for selecting todos by id, todos list etc. Also the store is a distinct file.

The right is another approach from a youtube video. The basic structure is created by the create-react-app. Then we can create folders for the actions and the reducers and have one reducer per file.

You might see the constants.js also as types.js



Layout components

any layout components, for example the header navbar, or a sidebar, or the footer etc could be placed in a layout folder within the components folder.

Selectors

selectors.js

```
export const getTodoList = store =>
  store && store.todos ? store.todos.allIds : [];

export const getTodoById = (store, id) =>
  store && store.todos && store.todos.byIds
    ? { ...store.todos.byIds[id], id }
    : {};

/**
 * example of a slightly more complex selector
 * select from store combining information from multiple reducers
 */
export const getTodos = store =>
  getTodoList(store).map(id => getTodoById(store, id));
```

We recommend encapsulating any complex lookups or computations of data in selector functions (in a distinct file). In addition, you can further optimize the performance by using Reselect to write "memoized" selectors that can skip unnecessary work.

Redux Middleware

store.js

```
import { createStore, applyMiddleware } from "redux";
import { composeWithDevTools } from
"redux-devtools-extension";
import thunk from "redux-thunk";
import rootReducer from "./reducers";

const initialState = {};

const middleware = [thunk];

const store = createStore(
  rootReducer,
  initialState,
  composeWithDevTools(applyMiddleware(...middleware))
);

export default store;
```

This is an example of setting up the redux store for your project.

The createStore function gets a rootreducer, then the initial state and then any middleware.

Since we use the redux devtools we define the middleware inside the composeWithDevtools function of the devtools module.

dispatch prop by default

If you don't specify the second argument to connect(), your component will receive dispatch by default.

Can I call store.dispatch?

It's an anti-pattern to interact with the store directly in a React component, whether it's an explicit import of the store or accessing it via context. Let React Redux's connect handle the access to the store, and use the dispatch it passes to the props to dispatch actions.

A todo app overview

The React UI Components

We have implemented our React UI components as follows:

TodoApp is the entry component for our app. It renders the header, the AddTodo, TodoList, and VisibilityFilters components.

AddTodo is the component that allows a user to input a todo item and add to the list upon clicking its “Add Todo” button:

It uses a controlled input that sets state upon onChange.

When the user clicks on the “Add Todo” button, it dispatches the action (that we will provide using React Redux) to add the todo to the store.

TodoList is the component that renders the list of todos:

It renders the filtered list of todos when one of the VisibilityFilters is selected.

Todo is the component that renders a single todo item:

It renders the todo content, and shows that a todo is completed by crossing it out.

It dispatches the action to toggle the todo's complete status upon onClick.

VisibilityFilters renders a simple set of filters: all, completed, and incomplete. Clicking on each one of them filters the todos:

It accepts an activeFilter prop from the parent that indicates which filter is currently selected by the user. An active filter is rendered with an underscore.

It dispatches the setFilter action to update the selected filter.

constants holds the constants data for our app.

And finally index renders our app to the DOM.

You would have a todosReducer (that handles certain action types like ADD_TODO, GET_TODOS, DELETE_TODO etc.), an authRecuder, an errorReducer to bringing errors to the componenets,

It is a good convention that the action types exist in a distinct file called types.js or constants.js.

Fetching data from the server is done in the actions and the fetched data are added as payload to the actions.

...state

```
import { GET_LEADS } from "../actions/types.js";

const initialState = {
  leads: []
};

export default function(state = initialState, action) {
  switch (action.type) {
    case GET_LEADS:
      return {
        ...state,
        leads: action.payload
      };
    default:
      return state;
  }
}
```

A reducer returns the whole state not only the state property that the action modifies. This is why we return ...state, and then the modified property (which will override the property in the ...state)

getState()

	Have in mind that you can access the global state within an action with the getState function

Packages and Templates

List

1. **axios** package for ajax requests
2. **styled-components** for writing actual CSS code to style your components
3. **Path-to-RegExp** package
4. **classnames** package: A simple JavaScript utility for conditionally joining classNames together.
5. **clsx** (an alternative to classnames, smaller and faster)
6. **redux-devtools-extension**
7. **reselect** package: A selector library (eg. create memoized selectors for redux)
8. **redux-devtools**
9. **redux-observable**
10. **redux-thunk** (handle API request in Action creators) a middleware used to make asynchronous requests from the actions.
11. **redux-persist** (allow you to save store data in localStorage and rehydrate on refresh)
12. **date-fns** (handle dates)
13. **Formik** (build forms without tears)
14. **react-helmet** (for writing and managing html head content)
15. **prop-types** (for type checking of props)
16. **yup** (The JavaScript Object Schema Validator and Object Parser to use with React)
17. **Lodash** (Lodash makes JavaScript easier by taking the hassle out of working with arrays, numbers, objects, string)
18. **Notistack** (display notifications on your web apps)
19. **Jss** (a library for generating stylesheets with javascript. use JavaScript to describe styles)
20. **React-jss** (provides some nice features for react like theme support, lazy evaluation etc. implemented using hooks) Material UI uses react-jss internally like in makestyles()(where the pattern is identical or the ThemeProvider.
21. **Nprogress** (for slim progress bars like youtube)
22. **React-feather** (a collection of simply beautiful open source icons for React.js)
23. **https://miragejs.com/** fake backend API
24. **nanoid** Nano ID is a unique string ID generator for JavaScript and other languages.
25. **React perfect scrollbar** (it is a react component for the generic library “perfect scrollbar” with which you can customize the scrollbar, when it appears, how it looks etc. Instead of using the default browser scrollbar)
26. **React-spring** (bring your components to life with simple spring animation primitives, for example useSpring hook)
27. **react-router-config**

[0, 10, 20, 30, 40, 50, 60, 70, 80, 90],

Collection of React Hooks

There is a collection of community created custom hooks that you can use in your applications https://react-hooks.org/

Generic tools

loadtest, an HTTP load generator. It can be used to measure load times for your web pages.

Templates

https://blog.logrocket.com/comparing-popular-react-component-libraries/

- https://arwes.dev/ have the this SciFi like template in mind
- https://github.com/siriwatknp/mui-treasury **Layout for material UI** https://blog.bitsrc.io/introducing-layout-for-material-ui-329043618cb3 This is a very light layout only components library. It just provides the basic layout with minimal requirements.

npm install @material-ui/core @material-ui/icons @mui-treasury/layout

(styled-components is needed too, @mui-treasury/mockup is a separate package that contains mockup content for header, footer etc.)

(The old package was just mui-layout)

- **Bootswatch.com**

They have customized bootstrap CSS files for various templates. You can select one template and use that as cdn for your bootstrap css files. They give you a link.

- **Ant design** https://ant.design/ (from Alibaba)

Component library that is styled already. You install it as an npm package and use the css files it provides. Then you can grap the JSX code that describes a component and return it from one of your components.

This video shows an example around minute 24, https://www.youtube.com/watch?v=uZgRbnIsgrA. Notice that the parent layout component that creates the top menu, contents area etc. (named CustomLayout in this example) uses {props.children} to show its contents. These children contents is the things it wraps, in this example the ArticleList.

- **Material UI** https://material-ui.com/ direct competitor of Ant design
- **Metronic** https://keenthemes.com/metronic/
- IDE plugin that creates the boilerplate code for class and function based components with a shortcut.

Using an html template

You can use an html template (for example bootstrap templates). these templates are made of html elements and use jquery. If you want to use such a template then you download the files and copy the css, js and img folders in your public folder of your react app. Then you reference them in your index.html file. You then have to create react components that return the html contents of the "html components" you want to use. For example https://blog.telexarsoftware.com/integrating-a-bootstrap-template-to-a-reactjs-application/

Using a react template

You download the files which usually are a create-react-app structure. You run npm install to install the dependencies. Then you start writing your logic and using the already made react components. You just import them in your code and use them.

axios

Similar with the fetch api (built in), it allows you to call an http endpoint and return a promise of the response (instead of the response itself).

Axios is a promise-based lightweight HTTP client for the browser and Node.js. It is similar to the Fetch API as it is also used in making network requests. Unlike fetch, it transforms all responses to JSON.

```
// Headers
const config = {
  headers: {
    "Content-Type": "application/json"
  }
};
// If token, add to headers config
if (token) {
  config.headers["Authorization"] = `Token ${token}`;
}
```

You can define custom http headers by creating an object and passing it during the call

```
axios
  .get("/api/auth/user", config)
  .then(res => {
    dispatch({
      type: USER_LOADED,
      payload: res.data
    });
  })
  .catch(err => {
    dispatch(returnErrors(err.response.data,
    err.response.status));
    dispatch({
      type: AUTH_ERROR
    });
  });
```

Axios automatically sets the `Content-Type` based on the 2nd argument to axios.post().

```
// Request Body
const body = JSON.stringify({ username, password });

axios
  .post("/api/auth/login", body, config)
  .then(res => {
    dispatch({
      type: USER_LOADED,
      payload: res.data
    });
  })
```

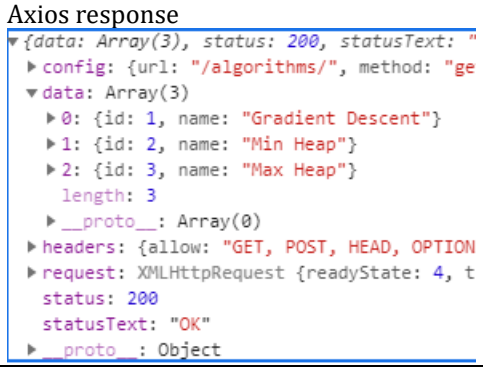
Pass a body with axios. Notice that we json stringify the data we want to send. This is not necessary. Axios automatically json stringifies javascript objects. If you happen to have a serialized JSON string that you want to send as JSON, be careful. If you pass a string to axios.post(), Axios treats that as a form-encoded request body so in this case make sure you explicitly set the content type to application json with a header.

Notice that you might have to pass **null** in cases of posting with an empty body.

Axios response to a get request for json data is an object with some attributes, one of which is the data attribute that contains the json response which has been parsed as a javascript object.

In this case django rest returns a list of objects by the default get request to a model viewset.

Axios response



Axios instance?

To intercept responses

Set default headers

```
axios.defaults.headers.common.Authorization = `Bearer ${accessToken}`;
delete axios.defaults.headers.common.Authorization;
```
to delete it

Show server errors instead of javascript errors

https://github.com/axios/axios/issues/960

```
axios.get('/api/xyz/abcd')
  .catch(function (error) {
    if (error.response) {
      // Request made and server responded
      console.log(error.response.data);
      console.log(error.response.status);
      console.log(error.response.headers);
    } else if (error.request) {
      // The request was made but no response was received
      console.log(error.request);
    } else {
      // Something happened in setting up the request that triggered an Error
      console.log('Error', error.message);
    }

  });
```

The **error.response** is particularly useful since you can use the server error data.

The error.message is present in all cases as I saw.

RTK Query

RTK Query is a powerful data fetching and caching tool.

It can eliminate the need to write any thunks or reducers to manage data fetching.

It is designed to simplify common cases for loading data in a web application, eliminating the need to hand-write data fetching & caching logic yourself.

RTK Query is an optional addon included in the Redux Toolkit package, and its functionality is built on top of the other APIs in Redux Toolkit.

Formik

It allows you to easily write the front end form validation logic among others, making the form creation a smoother task. You should still externally handle (show) any error messages coming from the server.

Integrating Formik for your forms makes the following things stress minimal:

- Managing form state — done automatically and locally. Packages like Redux Forms tie your form state to your state tree. This means that your top-level reducer is called on every keystroke. This is unnecessary overhead and bad design. Form state should be kept local.
- Validating a form — using Formik's validation handlers and (optionally) Yup. We are free to handle validation as we please with Formik, however, instead of reinventing the wheel, Formik also supports Yup: the most widely adopted object validation solution for React, directly into its handlers. (Read more about Yup here).
- Handling form submission — easy value parsing and error formatting, via handler functions passed into Formik.

Formik gives us multiple ways to build forms: a more traditional component based workflow for building forms (<Formik /> and <Field />components, amongst others), as well as a more abstract method using a "higher order component" with the withFormik class wrapper, for prop and handler management.

The Formik object also includes props for initial values, validation, and of course, onSubmit and render. Notice that in general a render prop can be replaced by a function as an Element's child.

When constructing forms, the last thing we want to worry about is how to juggle all our state, with errors, values and what not. With Formik we do not have to — it allows us to focus on our form components and the handling of its interaction.

Formik comes with a few extra components to make life easier and less verbose: <Form />, <Field />, and <ErrorMessage />. They use React context to hook into the parent <Formik /> state/methods.

Formik form structure

The form is constructed by a function passed to the render prop of the Formik element.

FormikProps is passed through the above render prop function, which gives us access to the state of our form (values, errors, touched, isSubmitting etc.)

Formik hooks up our input values to state using the Field's name prop. In the Textfield with name='email', the state value will be stored as *values.email*. Notice that the name must much the initialValues defined name.

```
...
import { Formik, FormikProps, Form, Field } from 'formik';
export class MyForm extends React.Component {


  handleSubmit = (values, {
     props = this.props,
     setSubmitting
   }) => {

   //process form submission here


   //done submitting, set submitting to false
   setSubmitting(false);


   return;
 }
render() {

 return(
   <Formik
     initialValues={{
         first_name: '',
         email: '',
         gender: ''
     }}
     validate={(values) => {
        let errors = {};

        //check if my values have errors
        return errors;
     }}
     onSubmit={handleSubmit}
     render={formProps: FormikProps => {


        return(
           <Form>
             <Field ... />
             <Field ... />
             <Field ... />


             <button
               type="submit"
               disabled={formProps.isSubmitting}>
                  Submit Form
             </button>
           </Form>
        );
    }
   />);
}
```

These arguments are returned in a formik object by the useFormik hook (that the Formik element calls internally). They are not generated by us but by Formik. They just need to be passed as arguments to the function of the Formik render prop, to be used by the form.

- **values**: contains the form's current input values. You can access each input value in formik.values.field-name
- **errors**: contains the errors generated by the validate Formik prop. you can access the errors for a field by *formik.errors.field-name*
- **touched**: Formik can keep track of which fields have been visited. It stores this information in an object called touched that also mirrors the shape of values/initialValues (*formik.touched.field-name*), but each key can only be a boolean true/false. This is useful to know which field was last touched so to render only that field's error.
- **handleBlur**: In JS the blur event fires when an element has lost focus. To take advantage of touched, we can pass formik.handleBlur to each input's onBlur prop. This function works similarly to formik.handleChange in that it uses the name attribute to figure out which field to update.
- **isSubmitting**: Formik will set this to true as soon as submission is attempted. Notice that you have to externally call setSubmitting(false) to your handleSubmit function..
- **handleChange**: a change handler to pass to each input. It does the standard process of controlled components.
- **handleSubmit**: the onSubmit Formik prop creates the handleSubmit function that is used in the form. Notice the arguments of the handleSubmit function. the first is values: this contains all my form values to work with. The second argument is an object, allowing us to pass our props and Formik methods into our submission handler
- handleReset

Validation

```
<Formik
  validate={(values) => {
     let errors = {};

      if(!values.email || is_valid_email(values.email))
         errors.email = 'Valid Email Address Required';

      return errors;
  }}
  ...
/>
```

```
// A custom validation function. This must return an object
// which keys are symmetrical to our values/initialValues
const validate = values => {
  const errors = {};
  if (!values.firstName) {
    errors.firstName = 'Required';
  } else if (values.firstName.length > 15) {
    errors.firstName = 'Must be 15 characters or less';
  }

  if (!values.lastName) {
    errors.lastName = 'Required';
  } else if (values.lastName.length > 20) {
    errors.lastName = 'Must be 20 characters or less';
  }

  if (!values.email) {
    errors.email = 'Required';
  } else if (!/^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}$/i.test(values.email)) {
    errors.email = 'Invalid email address';
  }

  return errors;
};
```

Validate method: The errors are generated by the Formik's validate prop. With the validate method we check the values fields and can returns any errors we like. These errors can then be accessed by the FormikProps.errors object. We can also adopt any third party library to use as our means of validation using this method.

By default, Formik will validate after each keystroke (change event), each input's blur event, as well as prior to submission. It will only proceed with executing the onSubmit function we passed to useFormik() if there are no errors (i.e. if our validation function returned {}

We could indeed carry out Yup validation within the validate() prop, however, Formik is now supporting a validationSchema prop to automatically validate your form based on a Yup object. validationSchema will automatically transform Yup's validation errors into a pretty object whose keys match values and touched.

```
const SignupSchema = Yup.object().shape({
  firstName: Yup.string()
    .min(2, 'Too Short!')
    .max(50, 'Too Long!')
    .required('Required'),
  lastName: Yup.string()
    .min(2, 'Too Short!')
    .max(50, 'Too Long!')
    .required('Required'),
  email: Yup.string().email('Invalid email').required('Required'),
});
```

```
//define my schema using Yup
const formSchema = yup.object().shape({
   ...
});

<Formik
  validationSchema={formSchema}
  ...
/>
```

However — if you indeed need further functionality, maybe an API request or websocket to validate an available username, we always have the validate prop at our disposal.

Whether or not you use Yup, it is highly recommended that you share commonly used validation methods across your application. This will ensure that common fields (e.g. email, street addresses, usernames, phone numbers, etc.) are validated consistently and result in a better user experience.

Error handling

Instead of this

```
{ formProps.touched.email &&
    formProps.errors.email &&
       <span className='error'>{formProps.errors.email}</span>
  }
```

We can use the Formik's ErrorMessage element

```
<ErrorMessage name="email" />
```

We only display this error if the email field has been touched, and if the error exists. then we access it via *formProps.errors.email*.

Input Field Types

The <Field> component by default will render an <input> component that given a name prop will implicitly grab the respective onChange, onBlur, value props and pass them to the element as well as any props you pass to it.

```
<Field
  name="gender"
  component="select"
  placeholder="Your Gender">
     <option value="male">Male</option>
     <option value="female">Female</option>
</Field>
```

We can use other input types with Fields. In fact, the *component* prop of <Field /> can accept input, select and textarea, as strings. We can also pass components into this prop, consequently allowing us to render any component we wish.

Another handy Formik object is the <FieldArray />. The <FieldArray /> object helps us manage forms that adopt an iterable list of inputs with a common subject. This method will allow us to loop through a range of values and construct Fields for each of them.

Devias Kit Formik example

```
<Container maxWidth="sm">
  <Formik
    initialValues={{
      email: 'demo@devias.io',
      password: 'Password123'
    }}
    validationSchema={Yup.object().shape({
      email: Yup.string().email('Must be a valid email').max(255).required('Email is requ
      password: Yup.string().max(255).required('Password is required')
    })}
    onSubmit={() => {
      navigate('/app/dashboard', { replace: true });
    }}
  >
    {({
      errors : FormikErrors<FormikValues> ,
      handleBlur : {(e: FocusEvent<any>): (fieldOrEvent: T)} ,
      handleChange : {(e: ChangeEvent<any>): (field: T)} ,
      handleSubmit,
      isSubmitting : boolean ,
      touched : FormikTouched<FormikValues> ,
      values : FormikValues
    }) => (
      <form onSubmit={handleSubmit}>
        <Box mb={3}...>
        <Grid
          container
          spacing={3}
        ...>
        <Box
          mt={3}
          mb={1}
        >
```

```
<TextField
  error={Boolean( value: touched.email && errors.email)}
  fullWidth
  helperText={touched.email && errors.email}
  label="Email Address"
  margin="normal"
  name="email"
  onBlur={handleBlur}
  onChange={handleChange}
  type="email"
  value={values.email}
  variant="outlined"
/>
```

The textField is a component returned by the useField hook provided by Formik. For example:

```
const MyTextInput = ({ label, ...props }) => {
  // useField() returns [formik.getFieldProps(), formik.getFieldMeta()]
  // which we can spread on <input> and also replace ErrorMessage entirely.
  const [field, meta] = useField(props);
  return (
    <>
      <label htmlFor={props.id || props.name}>{label}</label>
      <input className="text-input" {...field} {...props} />
      {meta.touched && meta.error ? (
        <div className="error">{meta.error}</div>
      ) : null}
    </>
  );
};
```

```
<Form>
  <MyTextInput
    label="First Name"
    name="firstName"
    type="text"
    placeholder="Jane"
  />
```

As you can see above, useField() gives us the ability to connect any kind input of React component to Formik as if it were a <Field> + <ErrorMessage>. We can use it to build a group of reusable inputs that fit our needs.

Other packages

Classnames

classnames library lets you join different classes based on different conditions in a simpler way.

Without classnames

```
var Button = React.createClass({
  // ...
  render () {
    var btnClass = 'btn';
    if (this.state.isPressed) btnClass += ' btn-pressed';
    else if (this.state.isHovered) btnClass += ' btn-over';
    return <button className={btnClass}>{this.props.label}</button>;
  }
});
```

With classnames

```
var classNames = require('classnames');

var Button = React.createClass({
  // ...
  render () {
    var btnClass = classNames({
      btn: true,
      'btn-pressed': this.state.isPressed,
      'btn-over': !this.state.isPressed && this.state.isHovered
    });
    return <button className={btnClass}>{this.props.label}</button>;
  }
});
```

Suppose you have 2 classes of which one is going to get used every time but the second one gets used based on some condition

Yup

With Yup, we create a Yup formatted object that resembles our intended schema for an object, and then use Yup utility functions to check if our data objects match this schema — hence validating them. https://medium.com/@rossbulat/introduction-to-yup-object-validation-in-react-9863af93dc0e . used by Formik.

API requests, form submissions, or custom objects to handle our state. We need to make sure we are delivering data that our components expect to work with

Yup can test whether a value is an email address with one method call. Likewise with our timestamp value — Yup can test whether this value is a date.

```
const checkoutAddressSchema = yup.object().shape({

  email_address: yup
      .string()
      .email()
      .required(),
  full_name: yup
      .string()
      .required(),
  house_no: yup
      .number()
      .required()
      .positive()
      .integer(),
  address_1: yup
      .string()
      .required(),
  address_2: yup
      .string(),
  post_code: yup
      .string()
      .required(),
  timestamp: yup
      .date()
      .default(() => (new Date()),
   }),

});
```

Each method is a validator. This granularity allows us as developers to combine as many validators as we see fit for our data validation.
Suppose we have the following data:

```
let addressFormData = {
   email_address: 'ross@jkrbinvestments.com',
   full_name: 'Ross Bulat',
   house_no: null,
   address_1: 'My Street Name',
   post_code: 'AB01 0AB',
}
```

- Check if it is valid

```
const valid = await checkoutAddressSchema.isValid(addressFormData);
```

Or

```
checkoutAddressSchema
  .isValid(addressFormData)
  .then(function(valid) {

     //valid - true or false
  });
```

- Validate

```
await checkoutAddressSchema.validate(addressFormData);
```

- Cast

```
checkoutAddressSchema.cast(addressFormData);
```

- is_valid() method returns true or false
- validate() method checking whether our data is valid, and we are running validation on it (for example adding missing fields with the default values)
- cast() method that checks and fixes its types (for example convert a string to integer or date object)
- concat() combine two schemas to one
- abortEarly() to stop validation execution as soon as the first error crops up
- trim(), uppercase(), lowercase(). min(), max(), round(), morethan(). lessthan(), truncate()

Notistack

Notistack is a Snackbar library which makes it extremely easy to display notifications on your web apps. It is highly customizable and enables you to stack snackbars/toasts on top of one another.

```
import { SnackbarProvider } from 'notistack';

<SnackbarProvider>
    <MyApp />
</SnackbarProvider>
```

```
import { useSnackbar } from 'notistack';

const MyButton = () => {
    const { enqueueSnackbar, closeSnackbar } = useSnackbar();

    const handleClick = () => {
        enqueueSnackbar('I love hooks');
    };

    return (
        <Button onClick={handleClick}>Show snackbar</Button>
    );
}
```

1. Wrap your app inside a SnackbarProvider component. Note: If you're using material-ui ThemeProvider, make sure SnackbarProvider is a child of it.

2. Use the useSnackbar hook (or the withSnackbar HOC)

```
const key = this.props.enqueueSnackbar(message, options)
// dismiss a specific snackbar
this.props.closeSnackbar(key)
```

Material UI

Layout

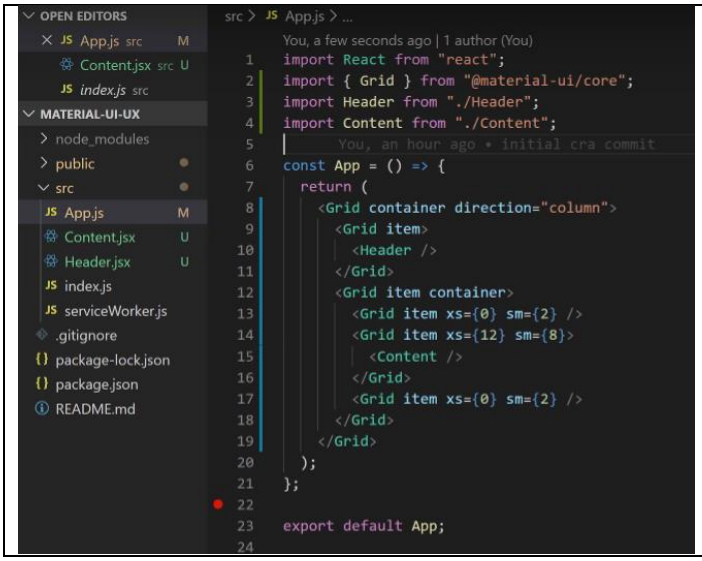
Container

The container centers your content horizontally. It's the most basic layout element.

Grid

The grid system is implemented with the Grid component:

- It uses CSS's Flexible Box module for high flexibility.
- There are two types of layout: containers and items.
- Item widths are set in percentages, so they're always fluid and sized relative to their parent element.
- Items have padding to create the spacing between individual items.
- There are five grid breakpoints: xs, sm, md, lg, and xl.



Material Design's responsive UI is based on a 12-column grid layout.
The layout is defined by the Grid components. The Grid components can be either containers or items of containers.

A grid container has a Direction property which can be either column or row. If it is row which is the default one then its items are arranged horizontally. For example if you define xs={6} for two items they will be next to each other. But if the direction is column they will be one on top of the other. This is very convenient for header and content items where you want one on top of the other.

There is also the spacing property of a container grid that defines the spacing between its items. Defining spacing={2} means two times the spacing value of your theme which by default is 8px.

Box

The Box component serves as a wrapper component for most of the CSS utility needs. It is an element wrapped around its content which by itself contains no styling rules nor has any default effects on the visual output. But it's a place to put styling rules as needed. The Box renders a div within which Box children elements are rendered.

The thing is that you can apply CSS styles to the BOX element directly via React props instead of using separate CSS files, CSS-in-JS or inline styling for the div. You could style the div either with inline styling, or with a separate css file or with css in js (using makeStyles and classname). instead you just define the css in the Box element (the difference with inline css is that the box element's props can be material UI's things)

```
import * as React from 'react'

import Box from '@material-ui/core/Box'

const Example = ({children}) => (
  <Box display="flex" flexDirection="column" alignItems="stretch" padding={1}>
    {children}
  </Box>
)
```

Notice also how padding={1} is a shorthand for theme.spacing(1). Box provides various conveniences for working with Material-UI themes like this.

Hidden

All elements are visible unless they are explicitly hidden. To ease integration with Material-UI's responsive breakpoints, this component can be used to hide any content, or you can use it in conjunction with the Grid component.

Misc

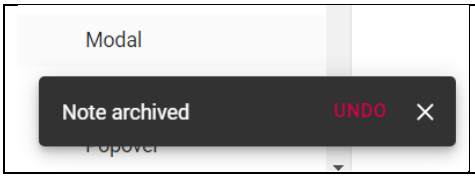
It uses the flexbox model. A flex container is the box generated by an element with a computed display of flex or inline-flex. In-flow children of a flex container are called flex items and are laid out using the flex layout model.

However, there is no instruction or topic about how to build layout based on them. That means you have to combine *Drawer, Header (AppBar), Content* and *Footer* by yourself.

The theme is an object. You can override its properties to modify it. You can customize the default theme by creating a theme object that overrides the properties of the default theme object you want to change.

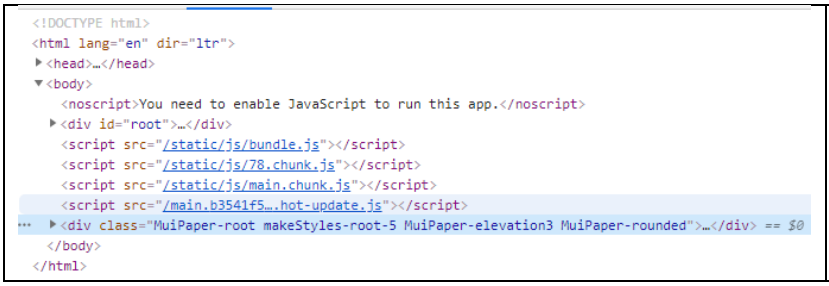
Other Components

Snackbar

	You can use them to display notifications

Portal

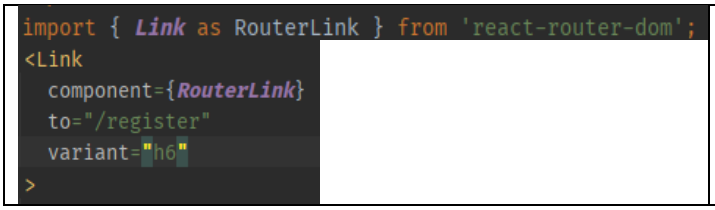
The portal component renders its children into a new "subtree" outside of current DOM hierarchy.

	Notice that it is outside the root div.

Portals provide a first-class way to render children into a DOM node that exists outside the DOM hierarchy of the parent component.

Link

React router has also a Link component. Deivas toolkit imports it as routerlink to separate it from the material ui's Link component.

	Why? It probably gives you some more props to pass to the material ui's link element, for example variant or color.

Variant

Is a prop of many components

variant	'h1' | 'h2' | 'h3' | 'h4' | 'h5' | 'h6' | 'subtitle1' | 'subtitle2' | 'body1' | 'body2' | 'caption' | 'button' | 'overline' | 'srOnly' | 'inherit'	'body1'	Applies the theme typography styles.

Theme and styles

ThemeProvider

The theme provider wraps the main App component and this way passes the theme variables to all the components of the app. The components that use these variables are the material UI components. The themeProvider takes a theme property that is the used theme object.

```jsx
import React from "react";
import { Grid, Typography, Button, Paper } from "@material-ui/core";
import { ThemeProvider, createMuiTheme } from "@material-ui/core/styles

function App() {
  const theme = createMuiTheme({
    palette: {
      type: "dark",
    },
  });
  return (
    <ThemeProvider theme={theme}>
      <Paper style={{ height: "100vh" }}>
        <Grid container direction="column">
          <Typography variant="h1">This is my App!</Typography>
          <Button variant="contained">This is a button</Button>
          <Button variant="contained" color="secondary">
            This is another button
          </Button>
        </Grid>
      </Paper>
    </ThemeProvider>
  );
}

export default App;
```

This is an example of implementing a dark theme.

The **Paper** element
Notice that in order for this to work you need to wrap your elements within a Paper element. The paper is the one that is affected by the palette, it becomes dark and the typography of wrapped elements is automatically adjusted to dark theme.

Have in mind that you can get the created theme in your function components with the useTheme hook.
const theme = useTheme()

Material ui styling

In general, the styles created by material ui (using whatever method like makestyles for example), are injected into the DOM by the JSS library.

You can use Material-UI's styling solution in your app, whether or not you are using Material-UI components. It is a CSS in JS solution. It uses JSS at its core. As a css in js solution, it offers many great features (theme nesting, dynamic styles, self-support, etc.). Material-UI's styling solution is inspired by many other styling libraries such as styled-components and emotion.

- the same advantages as styled-components.
- blazing fast.
- plugin API.
- It uses JSS at its core – a high performance JavaScript to CSS compiler which works at runtime and server-side.
- 15 KB gzipped; and no bundle size increase if used alongside Material-UI.

There are 3 possible APIs you can use to generate and apply styles, however they all share the same underlying logic.

Hook API, Styled components API and HOC API.

Hook API

```
import { makeStyles } from '@material-ui/core/styles';
const useStyles = makeStyles(() => ({
  header: {
    backgroundColor: '#fff',
    borderBottom: '1px solid hsl(210, 32%, 93%)',
  },
  collapseBtn: {
    color: '#fff',
    minWidth: 0,
    width: 40,
    borderRadius: '50%',
    border: 'none',
    backgroundColor: 'rgba(0,0,0,0.24)',
    margin: '0 auto 16px',
    '&:hover': {
      backgroundColor: 'rgba(0,0,0,0.38)',
    },
  },
  sidebar: {
    backgroundColor: '#4065E0',
    border: 'none',
  },
  content: {
    backgroundColor: '#f9f9f9',
  },
}));

const CustomStylesDemo = () => {
  const styles = useStyles();
  return (
    <Root scheme={fixedScheme}>
      <CssBaseline />
      <Header className={styles.header}>
        <Toolbar>
```

Styled components API

```
import React from 'react';
import { styled } from '@material-ui/core/styles';
import Button from '@material-ui/core/Button';

const MyButton = styled(Button)({
  background: 'linear-gradient(45deg, #FE6B8B 30%, #FF8E53 90%)',
  border: 0,
  borderRadius: 3,
  boxShadow: '0 3px 5px 2px rgba(255, 105, 135, .3)',
  color: 'white',
  height: 48,
  padding: '0 30px',
});

export default function StyledComponents() {
  return <MyButton>Styled Components</MyButton>;
}
```

HOC API

Makestyles classes are named like this: makeStyles-root-43. the auto generated class names is done by JSS.

makestyles function uses JSS in the background. For example the useStyles() hook is very similar with the jss.createStyleSheet function which compiles and renders the stylesheet (see jss notes). If you want to modify jss settings you have to use a StylesProvider component. If it is so, then the stylesheet is injected into the DOM by the useStyles() hook call inside the component's code.

```jsx
// A style sheet
const useStyles = makeStyles({
  root: {}, // a style rule
  label: {}, // a nested style rule
});

function Nested(props) {
  const classes = useStyles();
  return (
    <button className={classes.root}> // 'jss1'
      <span className={classes.label}> // 'jss2'
        nested
      </span>
    </button>
  );
}

function Parent() {
  return <Nested />
}
```

The makeStyles function is a Hook generation function, it returns a hook with which you can get the css classes inside the js file and assign them to your elements.

See in the docs how you can override classes of the nested elements with a parent prop.

JSS does not use any inline styles. Inline styles are slow if you overuse them. They are particularly slow in React.

```jsx
const useStyles = makeStyles( style: (theme : Theme ) => ({
  root: {
    backgroundColor: theme.palette.background.dark,
    minHeight: '100%',
    paddingTop: theme.spacing(3),
    paddingBottom: theme.spacing(3)
  },
```

The first argument (style) to the makestyles function is either an object or a function that generates an object containing the styles. This object will be linked to the component.
Notice that if you want access to your theme defined through the ThemeProvider, you have to use the function signature which takes the provided theme as its first argument.

StylesProvider (and JSS)

You only have to use it if you want to modify the default jss settings (as they are defined by material ui). If you want to change something regarding the JSS library settings, you have to use a StylesProvider component.

The stylesProvider is a component that you use to change how styles are applied to its children, for example it controls css specificity by controlling when and where is CSS injected into the page. (As I saw, the stylesProvider achieves this by being used along with a jss object created by the jss package).

CSS specificity: By default, the style tags are injected last in the <head> element of the page. They gain more specificity (because they are inserted last) than any other style tags on your page e.g. CSS modules, styled components. (you can change that with the injectFirst prop of the StylesProvider component)

For example, here we want to use the jss-rtl plugin.

```jsx
import { create } from 'jss';
import { StylesProvider, jssPreset } from '@material-ui/core/styles';
import rtl from 'jss-rtl'

const jss = create({
  plugins: [...jssPreset().plugins, rtl()],
});

export default function App() {
  return (
    <StylesProvider jss={jss}>
      ...
    </StylesProvider>
  );
}
```

JSS uses plugins to extend its core, allowing you to cherry-pick the features you need, and only pay the performance overhead for what you are using.

How is this related with the makestyles? It affects what styles, where and when are injected into the DOM. It doesn't define new styles. The styles definition is done by makestyles (or an equivalent api).

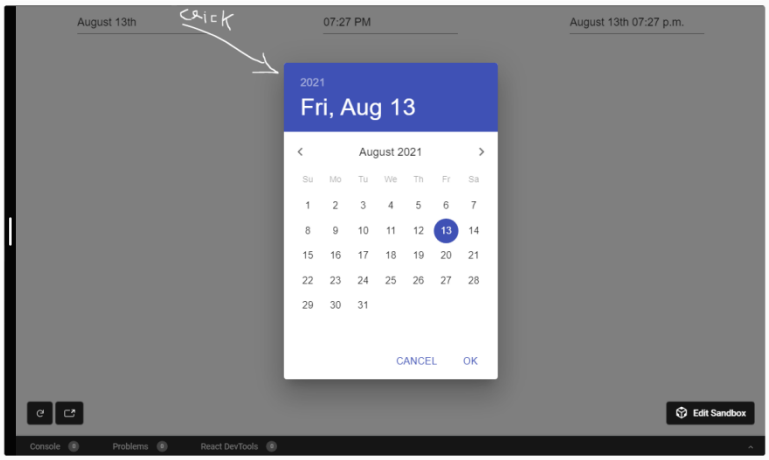
material-ui-pickers

Provides components for picking dates

```jsx
import React, { useState } from 'react';
import DateFnsUtils from '@date-io/date-fns'; // choose your lib
import {
  DatePicker,
  TimePicker,
  DateTimePicker,
  MuiPickersUtilsProvider,
} from '@material-ui/pickers';

function App() {
  const [selectedDate, handleDateChange] = useState(new Date());

  return (
    <MuiPickersUtilsProvider utils={DateFnsUtils}>
      <DatePicker value={selectedDate} onChange={handleDateChange} />
      <TimePicker value={selectedDate} onChange={handleDateChange} />
      <DateTimePicker value={selectedDate} onChange={handleDateChange} />
    </MuiPickersUtilsProvider>
  );
}
```



- Be ready to out-of-box localization, accessibility, timezone management, static typing and useful API
- Designed to be zero-effort compatible with moment, date-fns, luxon and dayjs (with the utils prop)
- Following material design guidelines and provide awesome ui both for desktop and mobile experience.

```jsx
import { MuiPickersUtilsProvider } from '@material-ui/pickers';

// pick a date util library
import MomentUtils from '@date-io/moment';
import DateFnsUtils from '@date-io/date-fns';
import LuxonUtils from '@date-io/luxon';

function App() {
  return (
    <MuiPickersUtilsProvider utils={DateFnsUtils}>
      <Root />
    </MuiPickersUtilsProvider>
  );
}

ReactDOM.render(<App />, document.querySelector('#app'));
```

Tell pickers which date management library it should use with MuiPickersUtilsProvider. This component takes a utils prop, and makes it available down the React tree with React Context. It should be used at the root of your component tree, or at the highest level you wish the pickers to be available.

Other

Typography

It automatically detects the background color and changes the main color so that it is visible. If you make a button color black the text will become white automatically.

CssBaseline

The docs say that its a collection of HTML element and attribute style-normalizations. Basically, it resets your CSS to a consistent baseline. That way, you can restyle your HTML doc so that you know you can expect all of the elements to look the same across all browsers. It's based on normalize.js,

Tips

Notice that the sidebar can be open, closed or collapsed. Collapsed is when it appears but it is very narrow.

Devias Kit

A pattern

Not all values are stored in redux. They use context instead. They build providers that provide some variables to their children through context. In addition to this, they also create some hooks with which you can access these same variables. The reason for these hooks is to be able to manipulate the values in the javascript code in another part of your code. You can read the data from the hook and manipulate it as needed. Examples of such providers are SettingsProvider, ThemeProvider, StylesProvider,

Updating the settings

When you press save settings, the currentsettings variable is updated. This variable is part of the SettingsContext Provider value. Since the value is updated all children of this provider that use that value are rerendered.

There is a TemeProvider child of SettingsProvider. The theme is created by considering the settings values. Whenever the settings change the theme change. Many components are children of the theme provider and use its theme value. So settings, change the theme value and the children component that use the theme value are re-rendered.

It uses route-based code splitting