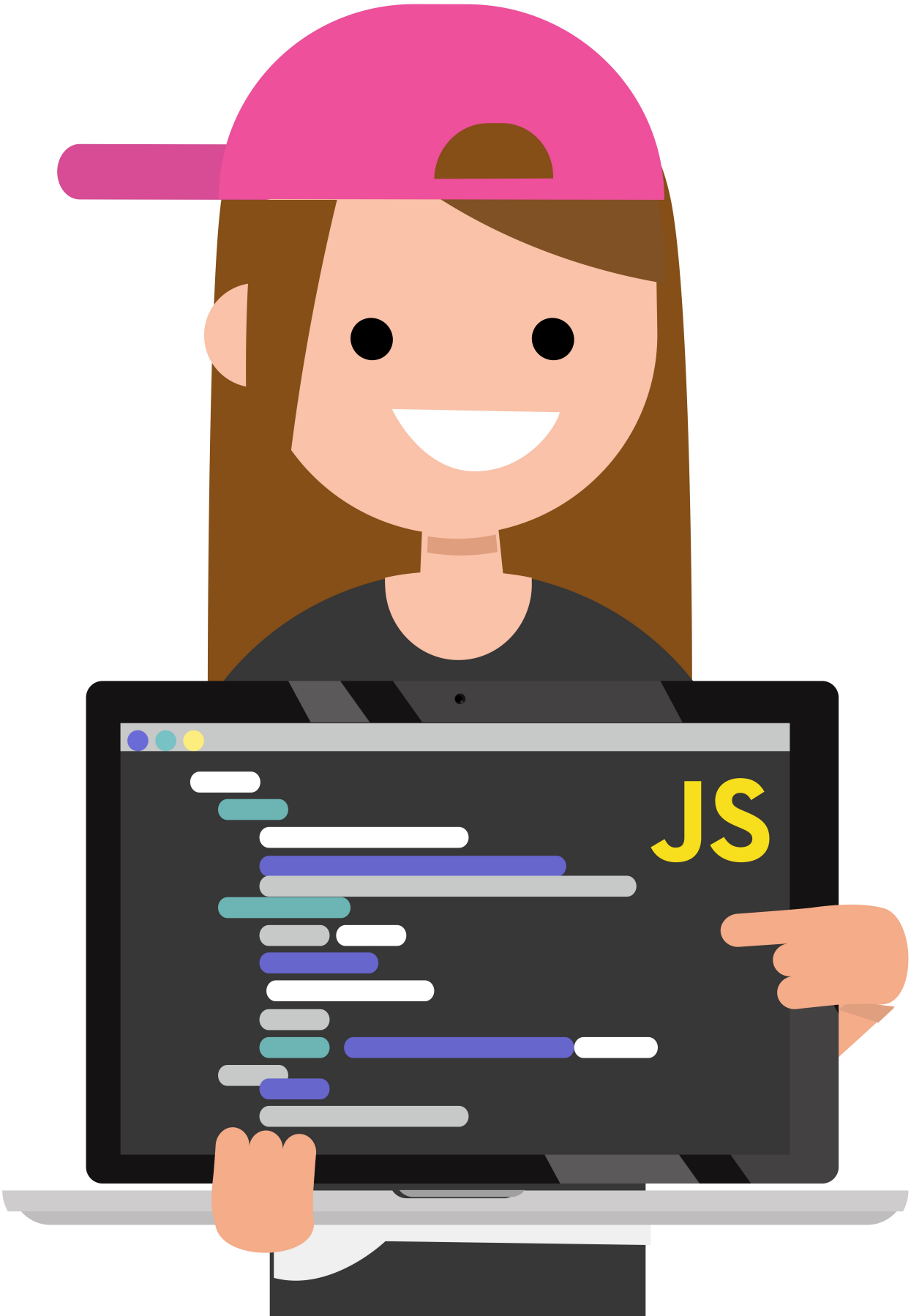
Learning

Patterns

By Lydia Hallie and Addy Osmani

ABOUT



We enable developers to build amazing things

Authors

Lydia and Addy started work on "Learning Patterns" to bring a modern perspective to JavaScript design, rendering and performance patterns.

Lydia Hallie

Lydia Hallie is a full-time software engineering consultant and educator that primarily works with JavaScript, React, Node, GraphQL, and serverless technologies. She also spends her time mentoring and doing in-person training sessions.

Addy Osmani

Addy Osmani is an engineering manager working on Google Chrome. He leads up teams focused on making the web fast. Some of the team's projects include Lighthouse, PageSpeed Insights, Aurora - working with React/Next.js/ Angular/Vue, contributions to Chrome DevTools and others.

The humans behind patterns.dev

Lydia Hallie

Co-creator & Writer

Twitter · GitHub · LinkedIn

Addy Osmani

Co-creator & Writer

Twitter · GitHub · LinkedIn

Josh W. Comeau

Whimsical UX

Twitter · GitHub · LinkedIn

Anton Karlovskiy

Software Engineer

Twitter · GitHub · LinkedIn

Leena Sohoni-Kasture

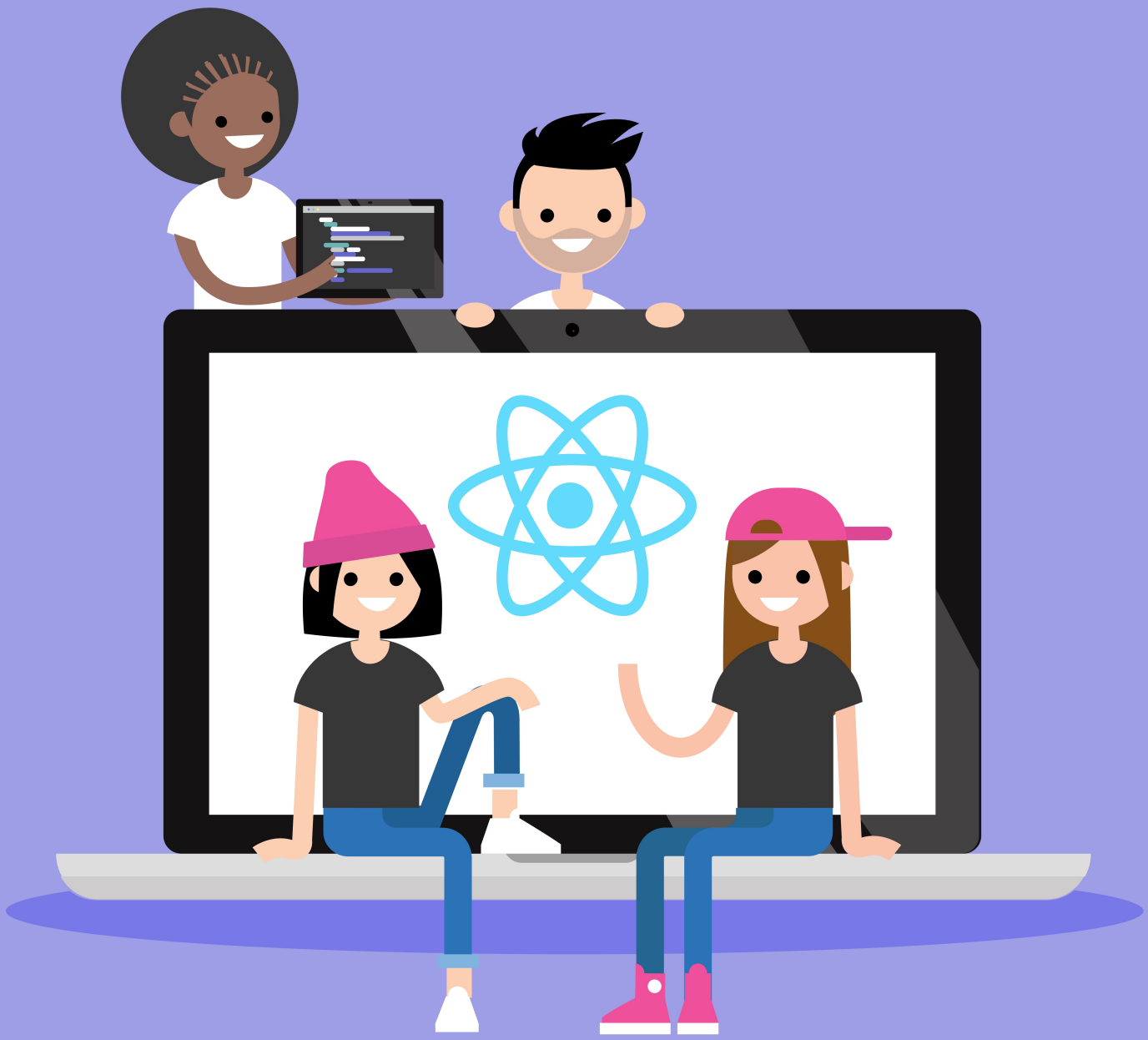
Writer & Editing

Nadia Snopek

Illustrator

Instagram · Behance

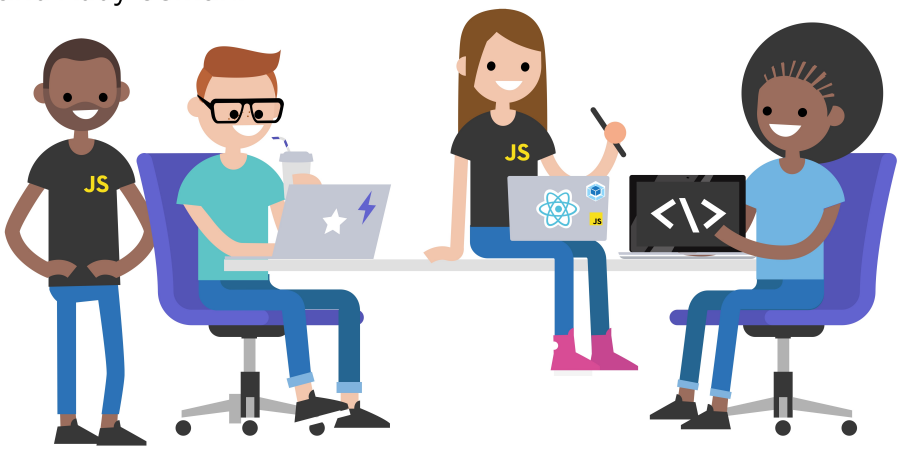
License



DESIGN PATTERNS

With Lydia Hallie

and Addy Osmani



INTRODUCTION

Design patterns are a fundamental part of software development, as they provide typical solutions to commonly recurring problems in software design. Rather than providing specific pieces of software, design patterns are merely concepts that can be used to handle recurring themes in an optimized way.

Over the past couple of years, the web development ecosystem has changed rapidly. Whereas some well-known design patterns may simply not be as valuable as they used to be, others have evolved to solve modern problems with the latest technologies.

Facebook's JavaScript library React has gained massive traction in the past 5 years, and is currently the most frequently downloaded framework on NPM compared to competing JavaScript libraries such as Angular, Vue, Ember and Svelte. Due to the popularity of React, design patterns have been modified, optimized, and new ones have been created in order to provide value in the current modern web development ecosystem. The latest version of React introduced a new feature called Hooks, which

plays a very important role in your application design and can replace many traditional design patterns.

Modern web development involves lots of different kinds of patterns. This project covers the implementation, benefits and pitfalls of common design patterns using ES2015+, React-specific design patterns and their possible modification and implementation using React Hooks, and many more patterns and optimizations that can help improve your modern web app!

Overview of React.js

A UI library for building reusable user interface components

Over the years, there has been an increased demand for straight-forward ways to **compose** user-interfaces using JavaScript. React, also referred to as React.js, is an open-source JavaScript library designed by Facebook, used for building user interfaces or UI components.

React is of course not the only UI library out there. Preact, Vue, Angular, Svelte, Lit and many others are also great for composing interfaces from reusable elements. Given React's popularity, it's worth walking through how it works given we will be using it to walk through some of the design, rendering and performance patterns in this guide.

When front-end developers talk about code, it's most often in the context of designing interfaces for the web. And the way we think of interface composition is in elements, like buttons, lists, navigation, and the likes. React provides an optimized and simplified way of expressing interfaces in these elements. It also helps build complex and tricky interfaces by organizing your interface into three key concepts— *components, props,* and *state.*

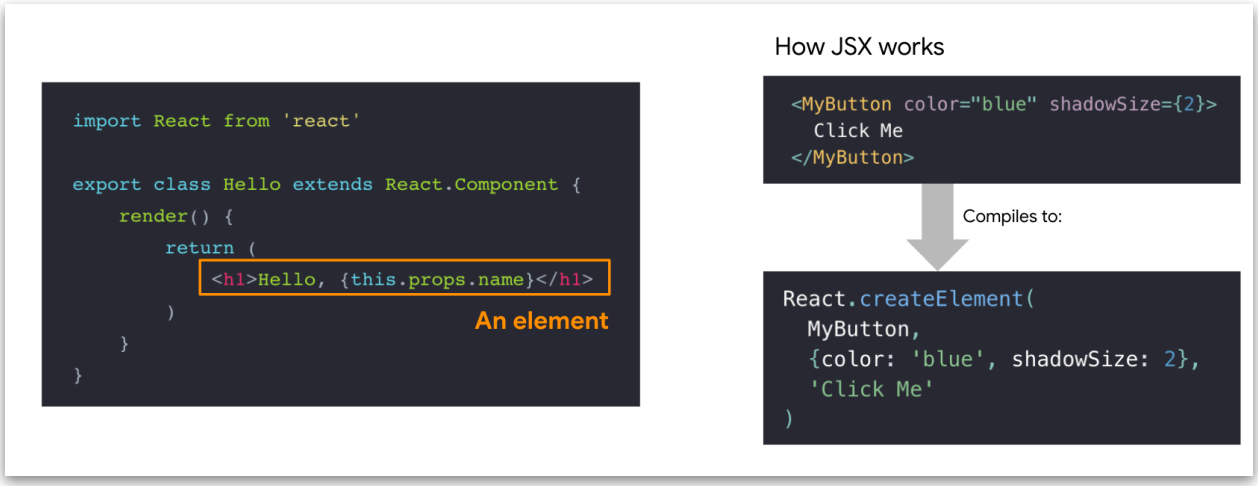
Because React is composition-focused, it can, perfectly map to the elements of your design system. So, in essence, designing for React actually rewards you for thinking in a modular way. It allows you to design individual components before putting together a page or view, so you fully understand each component's scope and purpose—a process referred to as *componentization*.

Terminology we will use:

- **React / React.js / ReactJS** - React library, created by Facebook in 2013
- **ReactDOM** - The package for DOM and server rendering
- **JSX** - Syntax extension to JavaScript
- **Redux** - Centralized state container
- **Hooks** - A new way to use state and other React features without writing a class
- **React Native** - The library to develop cross-platform native apps with Javascript
- **Webpack** - JavaScript module bundler, popular in React community.
- **CRA (Create React App)** - A CLI tool to create a scaffolding React app for bootstrapping a project.
- **Next.js** - A React framework with many best-in-class features including SSR, Code-splitting, optimized for performance, etc.

Rendering with JSX

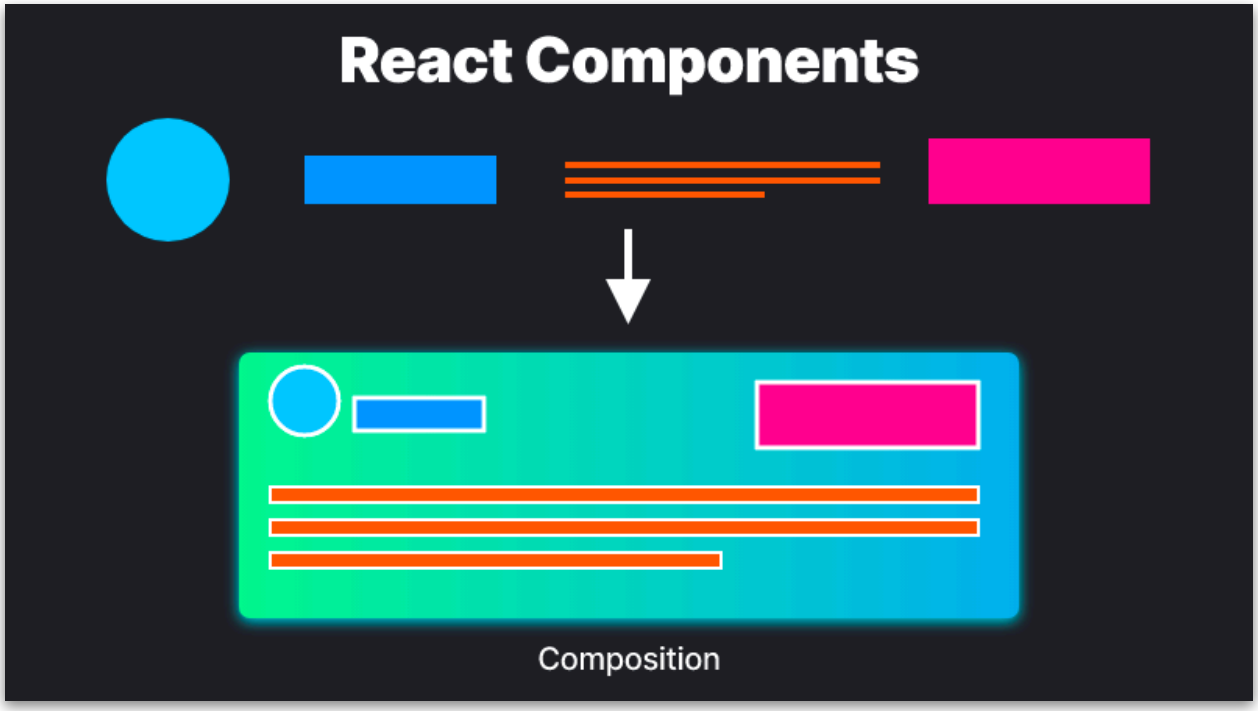
We will be using JSX in a number of our examples. JSX is an extension to JavaScript which embeds template HTML in JS using XML-like syntax. It is meant to be transformed into valid JavaScript, though the semantics of that transformation are implementation-specific. JSX rose to popularity with the React library, but has since seen other implementations as well.



Components, Props, and State

Components, props, and state are the three key concepts in React. Virtually everything you're going to see or do in React can be classified into at least one of these key concepts, and here's a quick look at these key concepts:

Components



Components are the building blocks of any React app. They are like JavaScript functions that accept arbitrary input (*Props*) and return React elements describing what should be displayed on the screen.

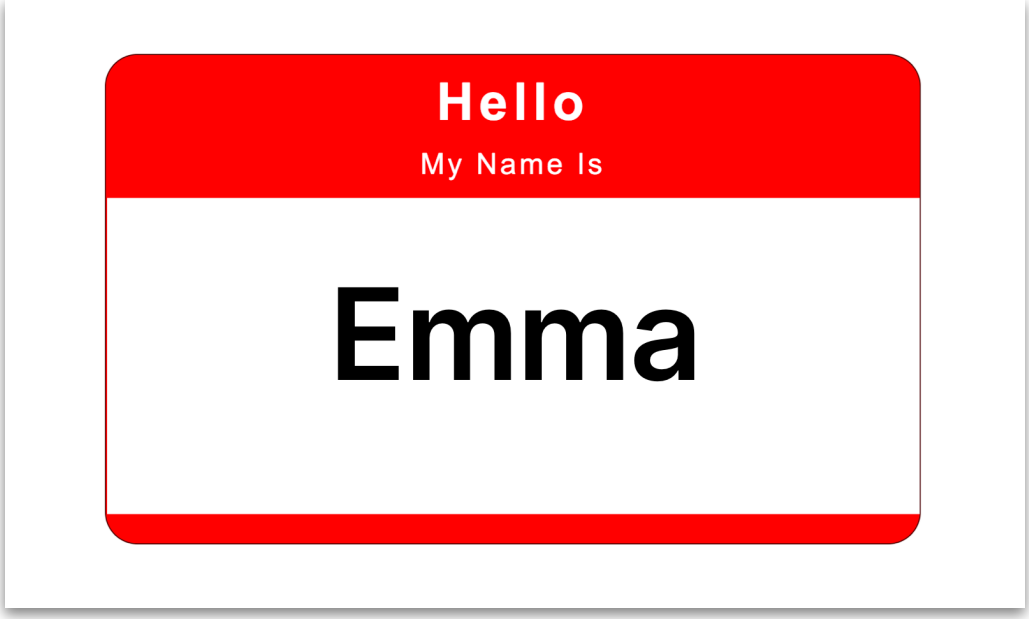
The first thing to understand is that everything on screen in a React app is part of a component. Essentially, a React app is just components within components within components. So developers don't build pages in React; they build components.

Components let you split your UI into independent, reusable pieces. If you're used to designing pages, thinking in this modular way might seem like a big change. But if you use a design system or style guide? Then this might not be as big of a paradigm shift as it seems.

The most direct way to define a component is to write a JavaScript function.

```
function Badge(props) {
  return <h1>Hello, my name is {props.name}</h1>;
}
```

This function is a valid React component because it accepts a single prop (*which stands for properties*) object argument with data and returns a React element. Such components are called "*function components*" because they are literally JavaScript functions.

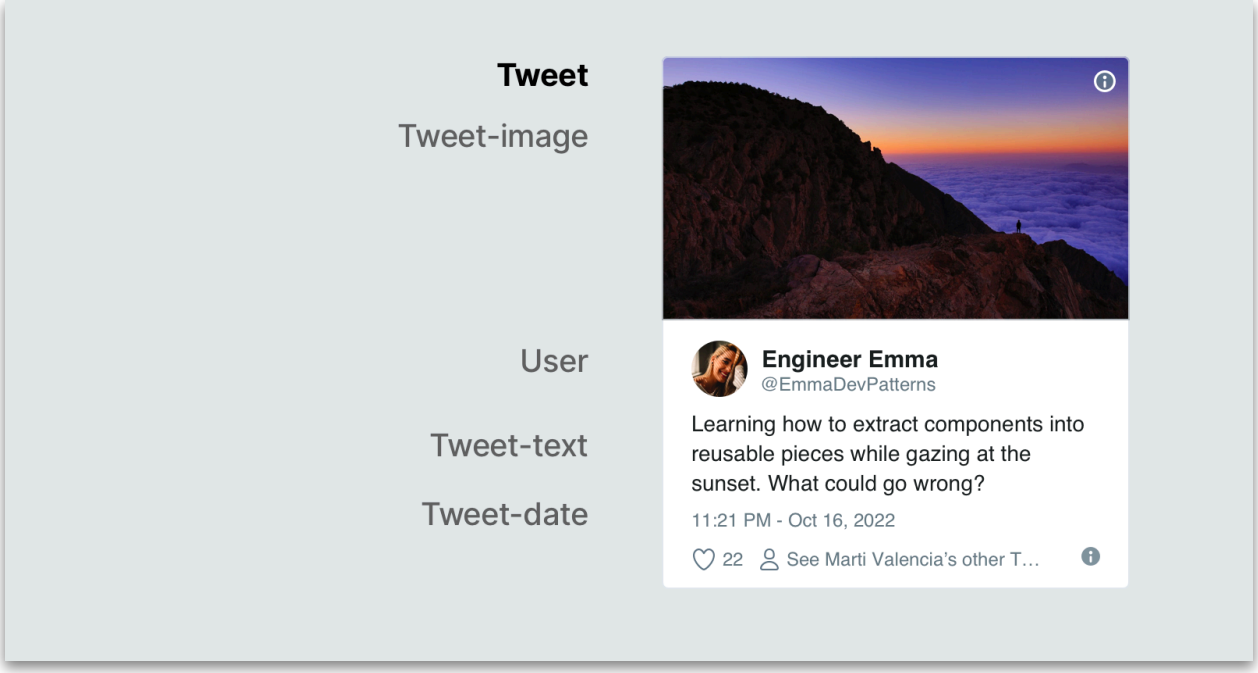


Aside from function components, another type of component are "*class components*." A class component is different from a function component in that it is defined by an ES6 class, as shown below:

```
class Badge extends React.Component {
  render() {
    return <h1>Hello, my name is {this.props.name}</h1>;
  }
}
```

Extracting components

To illustrate the facts that components can be split into smaller components, consider the following `Tweet` component:



Which can be implemented as follows:

```
function Tweet(props) {
  return (
    <div className="Tweet">
      <div className="User">
        <img className="Avatar"
          src={props.author.avatarUrl}
          alt={props.author.name}
        />
        <div className="User-name">
          {props.author.name}
        </div>
      </div>
      <div className="Tweet-text">
        {props.text}
      </div>
      <img className="Tweet-image"
          src={props.image.imageUrl}
          alt={props.image.description}
        />
      <div className="Tweet-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

This component can be a bit difficult to manipulate because of how clustered it is, and reusing individual parts of it would also prove difficult. But, we can still extract a few components from it.

The first thing we will do is extract *Avatar*:

```
function Avatar(props) {
  return (
    <img className="Avatar"
      src={props.user.avatarUrl}
      alt={props.user.name}
    />
  );
}
```

This component can be a bit difficult to manipulate because of how clustered it is, and reusing individual parts of it would also prove difficult. But, we can still extract a few components from it.

The first thing we will do is extract *Avatar*:

```
function Tweet(props) {
  return (
    <div className="Tweet">
      <div className="User">
        <Avatar user={props.author} />
        <div className="User-name">
          {props.author.name}
        </div>
      </div>
      <div className="Tweet-text">
        {props.text}
      </div>
      <img className="Tweet-image"
          src={props.image.imageUrl}
          alt={props.image.description}
        />
      <div className="Tweet-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

Avatar doesn't need to know that it is being rendered inside a `Comment`. This is why we have given its prop a more generic name: *user* rather than *author*.

```
function User(props) {
  return (
    <div className="User">
      <Avatar user={props.user} />
      <div className="User-name">
        {props.user.name}
      </div>
    </div>
  );
}
```

Now we will simplify the comment a little:

The next thing we will do is to a `User` component that renders an_ Avatar _next to the user's name:

```
function Tweet(props) {
  return (
    <div className="Tweet">
      <User user={props.author} />
      <div className="Tweet-text">
        {props.text}
      </div>
      <img className="Tweet-image"
          src={props.image.imageUrl}
          alt={props.image.description}
        />
      <div className="Tweet-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

Extracting components seems like a tedious job, but having reusable components makes things easier when coding for larger apps. A good criterion to consider when simplifying components is this: 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 extracted to a separate component.

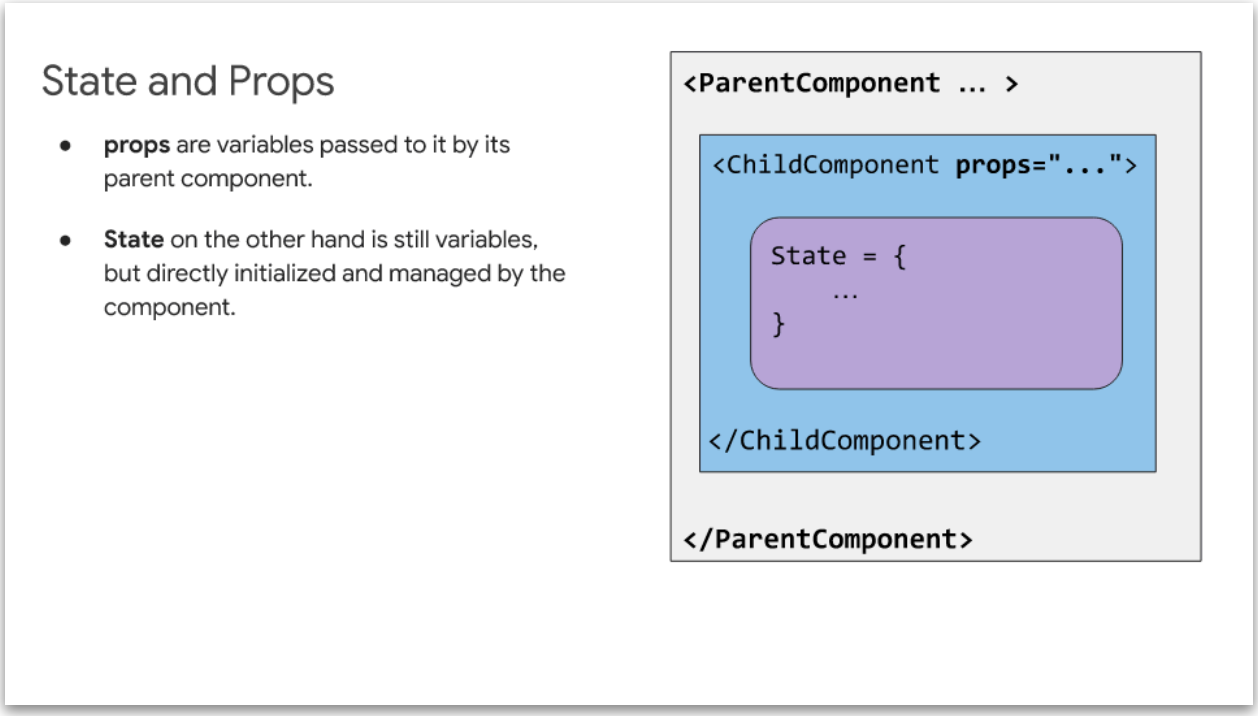
Props

Props are a short form for properties, and they simply refer to the internal data of a component in React. They are written inside component calls and are passed into components. They also use the same syntax as HTML attributes, e.g._ `prop="value"`. Two things that are worth remembering about props; Firstly, we determine the value of a prop and use it as part of the blueprint before the component is built. Secondly, the value of a prop will never change, i.e. props are read-only once they are passed into components.

The way you access a prop is by referencing it via the "this.props" property that every component has access to.

State

State is an object that holds some information that may change over the lifetime of the component. Meaning it is just the current snapshot of data stored in a component's Props. The data can change over time, so techniques to manage the way that data changes become necessary to ensure the component looks the way engineers want it to, at just the right time — this is called *State management.*



It's almost impossible to read one paragraph about React without coming across the idea of state-management. Developers love expounding upon this topic, but at its core, state management isn't really as complex as it sounds.

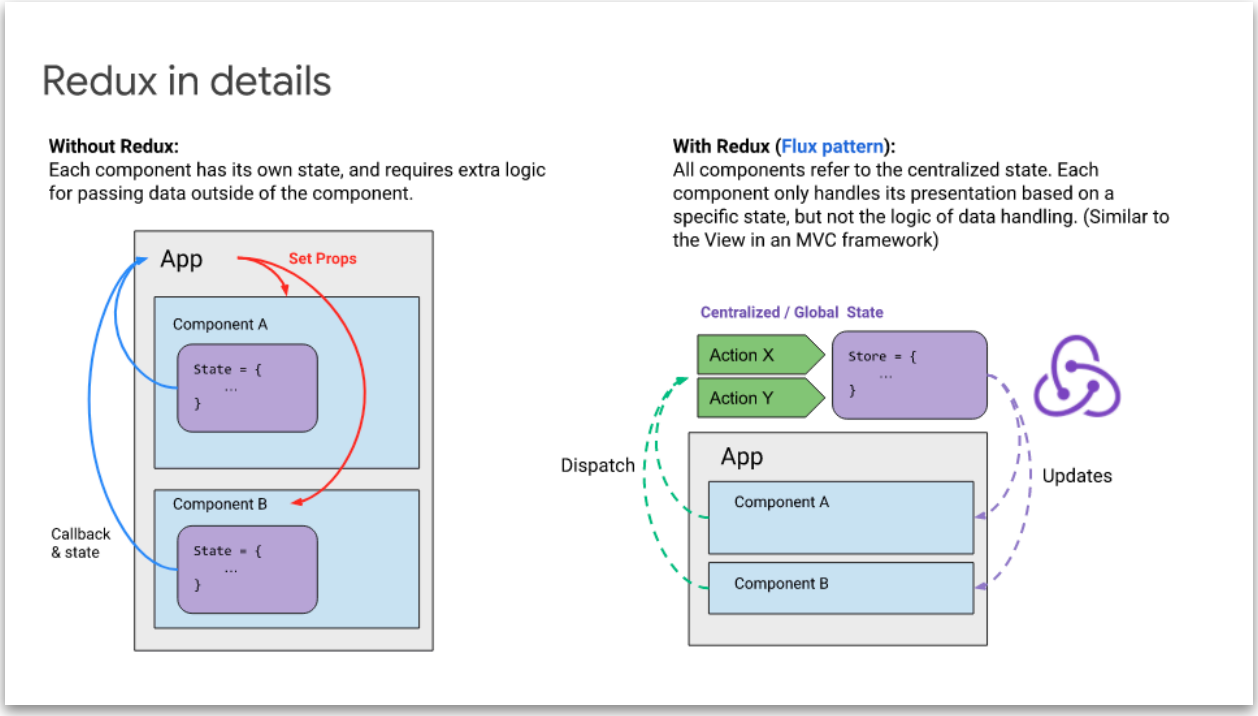
In React, state can also be tracked globally, and data can be shared between components as needed. Essentially, this means that in React apps, loading data in new places is not as expensive as it is with other technologies. React apps are smarter about which data they save and load, and when. This opens up opportunities to make interfaces that use data in new ways.

Think of React components like micro-applications with their own data, logic, and presentation. Each component should have a single purpose. As an engineer, you get to decide that purpose and have complete control over how each component behaves and what data is used. You're no longer limited by the data on the rest of the page. In your design, you can take advantage of

this in all kinds of ways. There are opportunities to present additional data that can improve the user experience or make areas within the design more contextual.

How to add State in React

When designing, Including state is a task that you should save for last. It is much better to design everything as stateless as possible, using props and events. This makes components easier to maintain, test, and understand. Adding states should be done through either state containers such as Redux and MobX, or a container/wrapper component. Redux is a popular state management system for other reactive frameworks. It implements a centralized state machine driven by actions.



In the example below, the place for the state could be *LoginContainer* itself. Let's use React Hooks (this will be discussed in the next section) for this:

```
const LoginContainer = () => {
  const [username, setUsername] = useState("");
  const [password, setPassword] = useState("");

  const login = async event => {
    event.preventDefault();
    const response = await fetch("/api", {
      method: "POST",
      body: JSON.stringify({
        username,
        password,
      }),
    });
    // Here we could check response.status to login or show error
  };

  return (
    <LoginForm onSubmit={login}>
      <FormInput
        name="username"
        title="Username"
        onChange={event => setUsername(event.currentTarget.value)}
        value={username}
      />
      <FormPasswordInput
        name="password"
        title="Password"
        onChange={event => setPassword(event.currentTarget.value)}
        value={password}
      />
      <SubmitButton>Login</SubmitButton>
    </LoginForm>
  );
};
```

For further examples such as the above, see Thinking in React 2020.

Props vs State

Props and state can sometimes be confused with each other because of how similar they are. Here are some key differences between them:

Props	**State**
The data remains unchanged from component to component.	Data is the current snapshot of data stored in a component's Props. It changes over the lifecycle of the component.
The data is read-only	The data can be asynchronous
The data in props cannot be modified	The data in state can be modified using *this.setState*
Props are what is passed on to the component	State is managed within the component

Other Concepts in React

Components, props, and state are the three key concepts for everything you'll be doing in react. But there are also other concepts to learn about:

Lifecycle

Every react component goes through three stages; mounting, rendering, and dismounting. The series of events that occur during these three stages can be referred to as the component's lifecycle. While these events are partially related to the component's state (its internal data), the lifecycle is a bit

different. React has internal code that loads and unloads components as needed, and a component can exist in several stages of use within that internal code.

There are a lot of lifecycle methods, but the most common ones are:

`render()` This method is the only required method within a class component in React and is the most used. As the name suggests, it handles the rendering of your component to the UI, and it happens during the mounting and rendering of your component.

When the component is created or removed:

- `componentDidMount()` runs after the component output has been rendered to the DOM.
- `componentWillUnmount()` is invoked immediately before a component is unmounted and destroyed

When the props or states get updated:

- `shouldComponentUpdate()` is invoked before rendering when new props or state are being received.
- `componentDidUpdate()` is invoked immediately after updating occurs. This method is not called for the initial render.

Higher-order component(HOC)

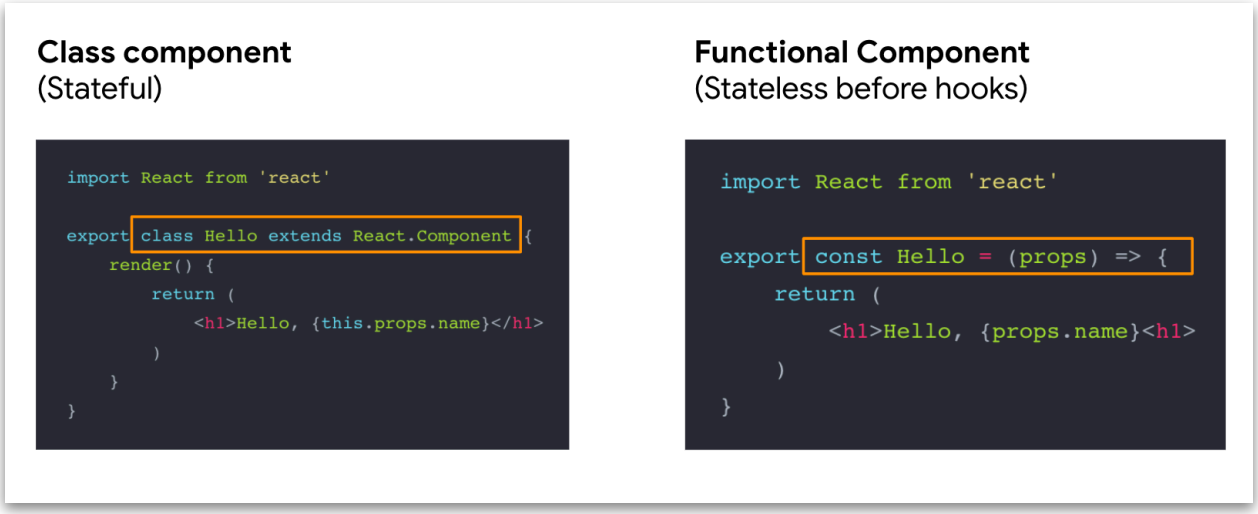
Higher-order components (HOC) are an advanced technique in React for reusing component logic. Meaning a higher-order component is a function that takes a component and returns a new component. They are patterns that emerge from React's compositional nature. While a component transforms props into UI, a higher-order component transforms a component into another component, and they tend to be popular in third-party libraries.

Context

In a typical React app, data is passed down via props, but this can be cumbersome for some types of props that are required by many components within an application. Context provides a way to share these types of data between components without having to explicitly pass a prop through every level of hierarchy. Meaning with context, we can avoid passing props through intermediate elements.

React Hooks

Hooks are functions that let you "hook into" React state and lifecycle features from functional components. They let you use state and other React features without writing a class. You can learn more about Hooks in our Hooks guide.



Thinking in React

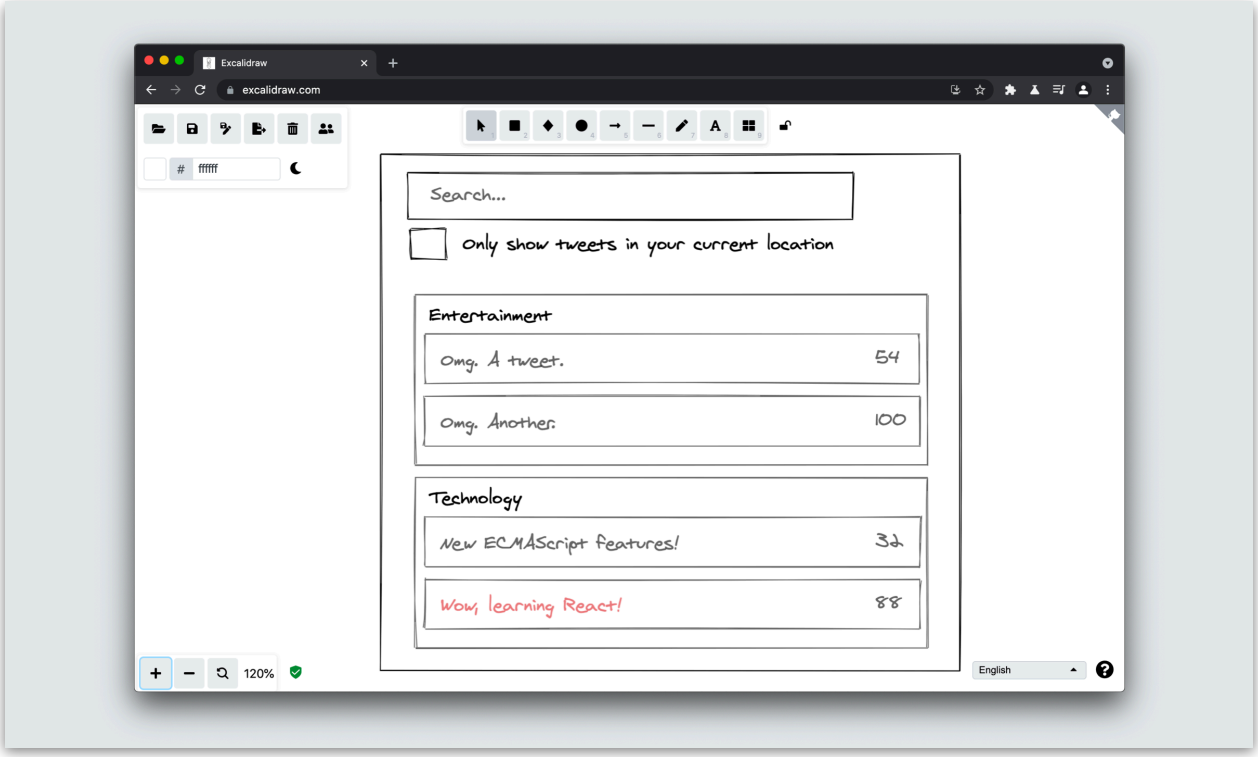
One thing that is really amazing about React is how it makes you think about apps as you build them. In this section, we'll walk you through the thought process of building a *Searchable product data table* using React Hooks.

Step 1: Start with a Mock

Imagine that we already have a JSON API and a mock of our interface:

Our JSON API returns some data that looks like this:

Tip: You may find free tools like Excalidraw useful for drawing out a high-level



mock of your UI and components.

```
[
  {category: "Entertainment", retweets: "54", isLocal: false, text: "Omg. A tweet."},
  {category: "Entertainment", retweets: "100", isLocal: false, text: "Omg. Another."},
  {category: "Technology", retweets: "32", isLocal: false, text: "New ECMAScript features!"},
  {category: "Technology", retweets: "88", isLocal: true, text: "Wow, learnin React!"}
];
```

Step 2: Break the UI into a Hierarchy Component

When you have your mock, the next thing to do is to draw boxes around every component (and subcomponent) in the mock and name all of them, as shown below.

Use the single responsibility principle: a component should ideally have a single function. If it ends up growing, it should be broken down into smaller subcomponents. Use this same technique for deciding if you should create a new function or object.

You'll see in the image above that we have five components in our app. We've listed the data each component represents.

- **TweetSearchResults (orange):** container for the full component
- **SearchBar (blue):** user input for what to search for
- **TweetList (green):** displays and filters tweets based on user input
- **TweetCategory (turquoise):** displays a heading for each category
- **TweetRow (red):** displays a row for each tweet

Now that the components in the mock have been identified, the next thing to do would be to sort them into a hierarchy. Components that are found within another component in the mock should appear as a child in the hierarchy. Like this:

- TweetSearchResults
    - SearchBar
    - TweetList
        - TweetCategory
        - TweetRow

Step 3: Implement the components in React

The next step after completing the component hierarchy is to implement your app. Before last year, the quickest way was to build a version that takes your data model and renders the UI but has zero interactivity, but since the introduction of React Hooks, an easier way to implement your app is to use the Hooks as seen below:

Filterable list of tweets

```
const TweetSearchResults = ({tweets}) => {
  const [filterText, setFilterText] = useState('');
  const [inThisLocation, setInThisLocation] = useState(false);
  return (
    <div>
      <SearchBar
        filterText={filterText}
        inThisLocation={inThisLocation}
        setFilterText={setFilterText}
        setInThisLocation={setInThisLocation}
      />
      <TweetList
        tweets={tweets}
        filterText={filterText}
        inThisLocation={inThisLocation}
      />
    </div>
  );
}
```

SearchBar

```
const SearchBar = ({
  filterText,
  inThisLocation,
  setFilterText,
  setInThisLocation
}) => (
  <form>
    <input
      type="text"
      placeholder="Search..."
      value={filterText}
      onChange={(e) => setFilterText(e.target.value)}
    />
    <p>
      <label>
        <input
          type="checkbox"
          checked={inThisLocation}
          onChange={(e) => setInThisLocation(e.target.checked)}
        />
        {' '}
        Only show tweets in your current location
      </label>
    </p>
  </form>
);
```

Tweet list (list of tweets)

```
const TweetList = ({tweets, filterText, inThisLocation}) => {
  const rows = [];
  let lastCategory = null;

  tweets.forEach((tweet) => {
    if (tweet.text.toLowerCase().indexOf(filterText.toLowerCase()) === -1) {
      return;
    }
    if (inThisLocation && !tweet.isLocal) {
      return;
    }
    if (tweet.category !== lastCategory) {
      rows.push(
        <TweetCategory
          category={tweet.category}
          key={tweet.category} />
      );
    }
    rows.push(
      <TweetRow
        tweet={tweet}
        key={tweet.text} />
    );
    lastCategory = tweet.category;
  });

  return (
    <table>
      <thead>
        <tr>
          <th>Tweet Text</th>
          <th>Retweets</th>
        </tr>
      </thead>
      <tbody>{rows}</tbody>
    </table>
  );
}
```

Tweet category row

```
const TweetCategory = ({category}) => (
  <tr>
    <th colSpan="2">
      {category}
    </th>
  </tr>
);
```

Tweet Row

```
const TweetRow = ({tweet}) =>  {
  const color = tweet.isLocal ? 'inherit' : 'red';

  return (
    <tr>
      <td><span style="">{tweet.text}</span></td>
      <td>{tweet.retweets}</td>
    </tr>
  );
}
```

The final implementation would be all the code written together in the previously stated hierarchy :

- TweetSearchResults
    - SearchBar
    - TweetList
        - TweetCategory
        - TweetRow

Getting Started

There are various ways to start using React.

Load directly on the web page: This is the simplest way to set up React. Add the React JavaScript to your page, either as an `npm` dependency or via a CDN.

Use create-react-app: create-react-app is a project aimed at getting you to use React as soon as possible, and any React app that needs to outgrow a single page will find that create-react-app meets that need quite easily. More serious production applications should consider using Next.js as it has stronger defaults (like code-splitting) baked in.

Code Sandbox: An easy way to have the create-react-app structure, without installing it, is to go to https://codesandbox.io/s and choose "React."

Codepen: If you are prototyping a React component and enjoy using Codepen, a number of React starting points are also available for use.

Conclusion

The React.js library was designed to make the process of building modular, reusable user interface components simple and intuitive. As you read through some of our other guides, we hope you found this brief introduction a helpful high-level overview.

This guide would not have been possible without the teaching styles shared in the official React components and props, thinking in React, thinking in React Hooks and the scriptverse docs

Singleton Pattern

Share a single global instance throughout our application

Singletons are classes which can be instantiated once, and can be accessed globally. This single instance can be shared throughout our application, which makes Singletons great for managing global state in an application.

First, let's take a look at what a singleton can look like using an ES2015 class. For this example, we're going to build a `Counter` class that has:

```
let counter = 0;

class Counter {
  getInstance() {
    return this;
  }

  getCount() {
    return counter;
  }

  increment() {
    return ++counter;
  }

  decrement() {
    return --counter;
  }
}
```

a `getInstance` method that returns the value of the instance

a `getCount` method that returns the current value of the `counter` variable

an `increment` method that increments the value of `counter` by one

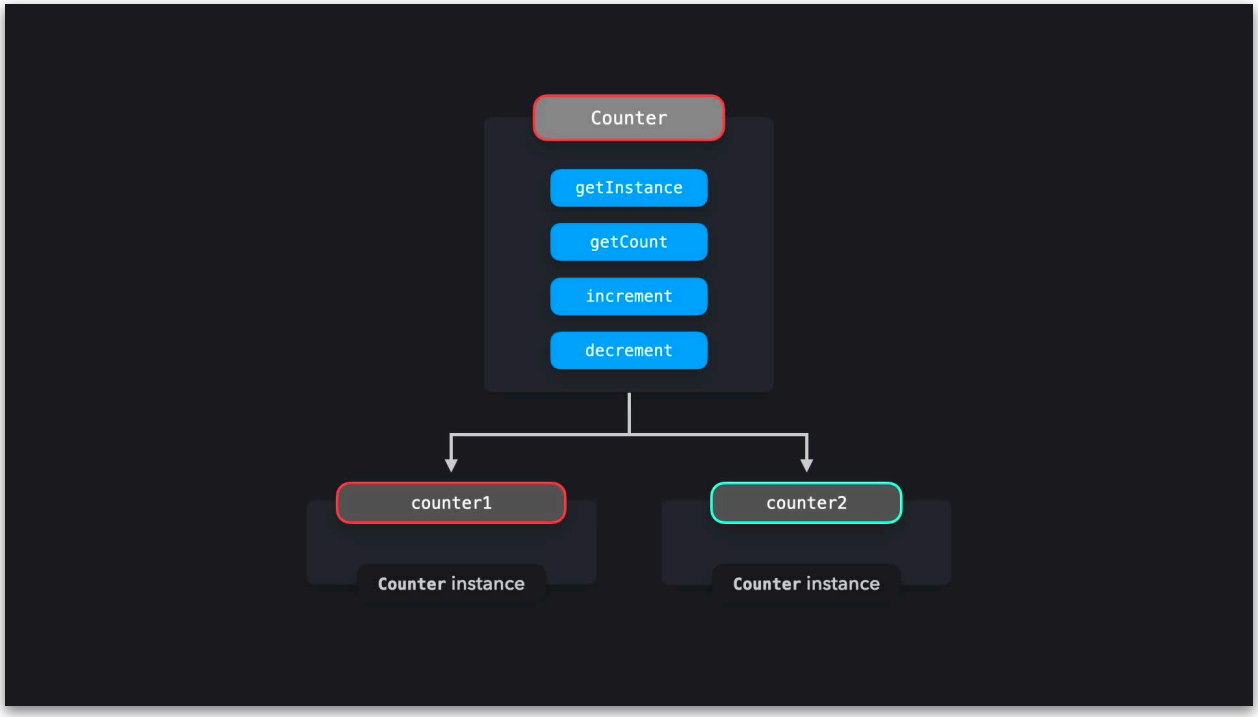
a `decrement` method that decrements the value of `counter` by one

However, this class doesn't meet the criteria for a Singleton! A Singleton should only be able to get instantiated once. Currently, we can create multiple instances of the `Counter` class.

```
let counter = 0;

class Counter {
  getInstance() {
    return this;
  }

  getCount() {
    return counter;
  }

  increment() {
    return ++counter;
  }

  decrement() {
    return --counter;
  }
}

const counter1 = new Counter();
const counter2 = new Counter();

console.log(counter1.getInstance() === counter2.getInstance()); // false
```

By calling the new method twice, we just set `counter1` and `counter2` equal to different instances. The values returned by the getInstance method on `counter1` and `counter2` effectively returned references to different instances: they aren't strictly equal!

Let's make sure that only one instance of the `Counter` class can be created.



One way to make sure that only one instance can be created, is by creating a variable called instance. In the constructor of `Counter`, we can set instance equal to a reference to the instance when a new instance is created. We can prevent new instantiations by checking if the instance variable already had a value. If that's the case, an instance already exists. This shouldn't happen: an error should get thrown.

```
let instance;
let counter = 0;

class Counter {
  constructor() {
    if (instance) {
      throw new Error("You can only create one instance!");
    }
    instance = this;
  }

  getInstance() {
    return this;
  }

  getCount() {
    return counter;
  }

  increment() {
    return ++counter;
  }

  decrement() {
    return --counter;
  }
}

const counter1 = new Counter();
const counter2 = new Counter();
// Error: You can only create one instance!
```

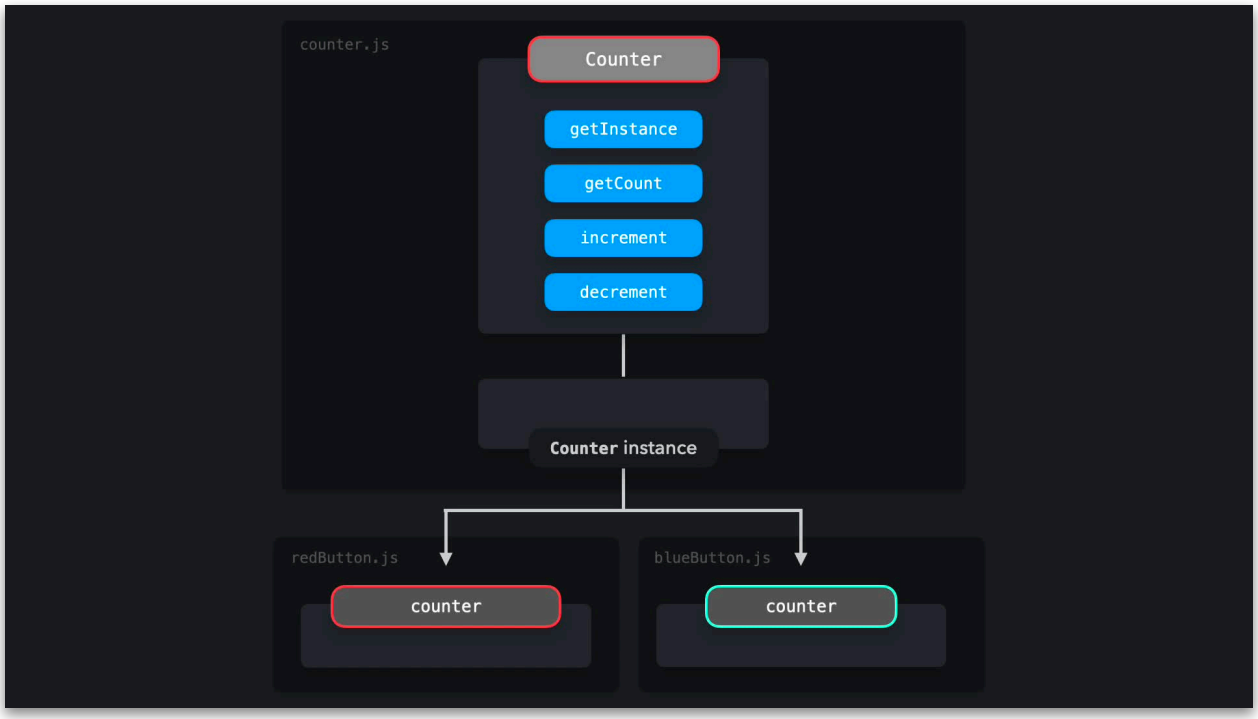
Perfect! We aren't able to create multiple instances anymore.

Let's export the `Counter` instance from the `counter.js` file. But before doing so, we should freeze the instance as well.
The `Object.freeze` method makes sure that consuming code cannot modify the Singleton. Properties on the frozen instance cannot be added or modified, which reduces the risk of accidentally overwriting the values on the Singleton.

```
let instance;
let counter = 0;

class Counter {
  constructor() {
    if (instance) {
      throw new Error("You can only create one instance!");
    }
    instance = this;
  }

  getInstance() {
    return this;
  }

  getCount() {
    return counter;
  }

  increment() {
    return ++counter;
  }

  decrement() {
    return --counter;
  }
}

const singletonCounter = Object.freeze(new Counter());
export default singletonCounter;
```

Let's take a look at an application that implements the Counter example. We have the following files:

- `counter.js`: contains the `Counter` class, and exports a `Counter` instance as its default export
- `index.js`: loads the `redButton.js` and `blueButton.js` modules
- `redButton.js`: imports `Counter`, and adds `Counter`'s `increment` method as an event listener to the red button, and logs the current value of counter by invoking the `getCount` method
- `blueButton.js`: imports Counter, and adds Counter's increment method as an event listener to the blue button, and logs the current value of counter by invoking the getCount method



Both `blueButton.js` and `redButton.js` import the same instance from `counter.js`. This instance is imported as `Counter` in both files.

When we invoke the increment method in either `redButton.js` or `blueButton.js`, the value of the counter property on the `Counter` instance updates in both files. It doesn't matter whether we click on the red or blue button: the same value is shared among all instances. This is why the counter keeps incrementing by one, even though we're invoking the method in different files.

(Dis)advantages

Restricting the instantiation to just one instance could potentially save a lot of memory space. Instead of having to set up memory for a new instance each time, we only have to set up memory for that one instance, which is referenced throughout the application. However, Singletons are actually considered an anti-pattern, and can (or.. should) be avoided in JavaScript.

In many programming languages, such as Java or C++, it's not possible to directly create objects the way we can in JavaScript. In those object-oriented programming languages, we need to create a class, which creates an object. That created object has the value of the instance of the class, just like the value of instance in the JavaScript example.

However, the class implementation shown in the examples above is actually overkill. Since we can directly create objects in JavaScript, we can simply use

a regular object to achieve the exact same result. Let's cover some of the disadvantages of using Singletons!

Using a regular object

Let's use the same example as we saw previously. However this time, the counter is simply an object containing:

- a `count` property
- an `increment` method that increments the value of `count` by one
- a `decrement` method that decrements the value of `count` by one

```
let count = 0;

const counter = {
  increment() {
    return ++count;
  },
  decrement() {
    return --count;
  }
};

Object.freeze(counter);
export { counter };
```

Since objects are passed by reference, both `redButton.js` and `blueButton.js` are importing a reference to the same singleton `Counter` object. Modifying the value of count in either of these files will modify the value on the singletonCounter, which is visible in both files.

Testing

Testing code that relies on a Singleton can get tricky. Since we can't create new instances each time, all tests rely on the modification to the global instance of the previous test. The order of the tests matter in this case, and one small modification can lead to an entire test suite failing. After testing, we need to reset the entire instance in order to reset the modifications made by the tests.

```
import Counter from "../src/counterTest";

test("incrementing 1 time should be 1", () => {
  Counter.increment();
  expect(Counter.getCount()).toBe(1);
});

test("incrementing 3 extra times should be 4", () => {
  Counter.increment();
  Counter.increment();
  Counter.increment();
  expect(Counter.getCount()).toBe(4);
});

test("decrementing 1  times should be 3", () => {
  Counter.decrement();
  expect(Counter.getCount()).toBe(3);
});
```

Dependency hiding

When importing another module, superCounter.js in this case, it may not be obvious that that module is importing a Singleton. In other files, such as index.js in this case, we may be importing that module and invoke its methods. This way, we accidentally modify the values in the Singleton. This can lead to unexpected behavior, since multiple instances of the Singleton can be shared throughout the application, which would all get modified as well.

index.js

```
import Counter from "./counter";
import SuperCounter from "./superCounter";
const counter = new SuperCounter();

counter.increment();
counter.increment();
counter.increment();

console.log("Counter in index.js", Counter.getCount())
```

Global behavior

A Singleton instance should be able to get referenced throughout the entire app. Global variables essentially show the same behavior: since global variables are available on the global scope, we can access those variables throughout the application.

Having global variables is generally considered as a bad design decision. Global scope pollution can end up in accidentally overwriting the value of a global variable, which can lead to a lot of unexpected behavior.

In ES2015, creating global variables is fairly uncommon. The new let and const keyword prevent developers from accidentally polluting the global scope, by keeping variables declared with these two keywords block-scoped. The new module system in JavaScript makes creating globally accessible values easier without polluting the global scope, by being able to export values from a module, and import those values in other files.

However, the common usecase for a Singleton is to have some sort of global state throughout your application. Having multiple parts of your codebase rely on the same mutable object can lead to unexpected behavior.

Usually, certain parts of the codebase modify the values within the global state, whereas others consume that data. The order of execution here is important: we don't want to accidentally consume data first, when there is no data to consume (yet)! Understanding the data flow when using a global state can get very tricky as your application grows, and dozens of components rely on each other.

State management in React

In React, we often rely on a global state through state management tools such as Redux or React Context instead of using Singletons. Although their global state behavior might seem similar to that of a Singleton, these tools provide a read-only state rather than the mutable state of the Singleton. When using

Redux, only pure function reducers can update the state, after a component has sent an action through a dispatcher.

Although the downsides to having a global state don't magically disappear by using these tools, we can at least make sure that the global state is mutated the way we intend it, since components cannot update the state directly.

Proxy Pattern

Share a single global instance throughout our application

With a **Proxy** object, we get more control over the interactions with certain objects. A proxy object can determine the behavior whenever we're interacting with the object, for example when we're getting a value, or setting a value.

Generally speaking, a proxy means a stand-in for someone else. Instead of speaking to that person directly, you'll speak to the proxy person who will represent the person you were trying to reach. The same happens in JavaScript: instead of interacting with the target object directly, we'll interact with the `Proxy` object.

Let's create a `person` object, that represents John Doe.

```
const person = {
  name: "John Doe",
  age: 42,
  nationality: "American"
};
```

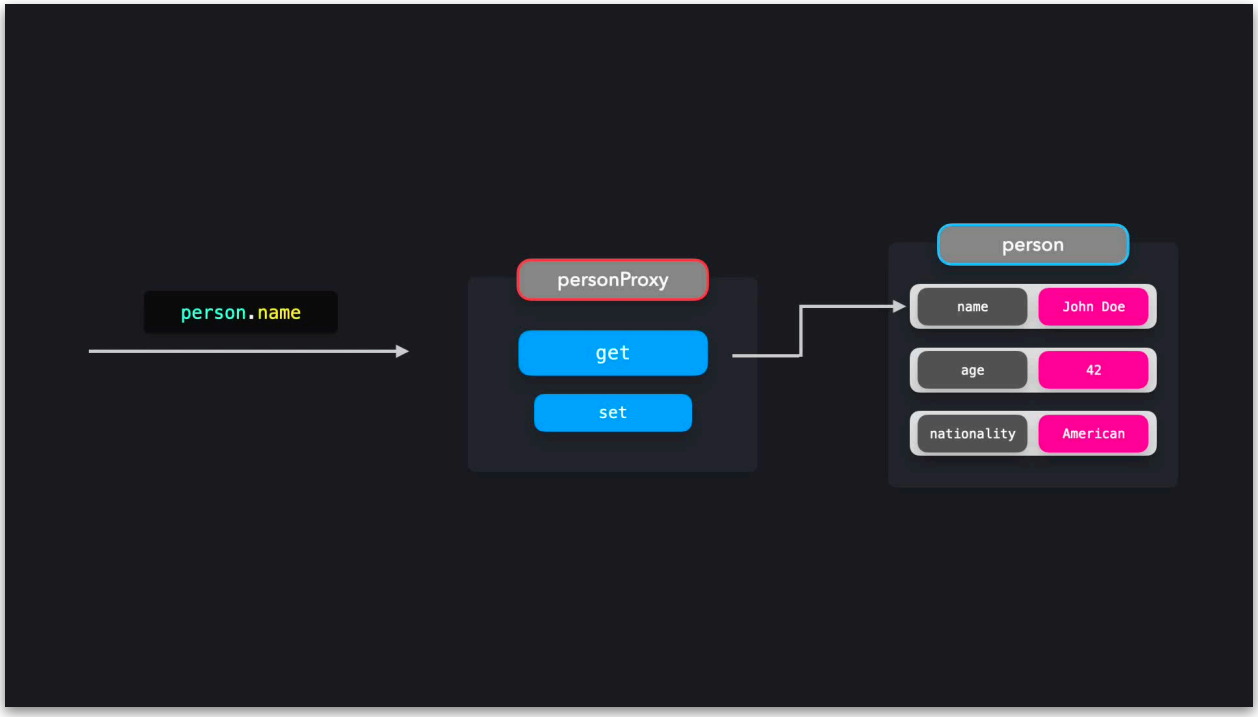
Instead of interacting with this object directly, we want to interact with a proxy object. In JavaScript, we can easily create a new proxy with by creating a new instance of `Proxy`.

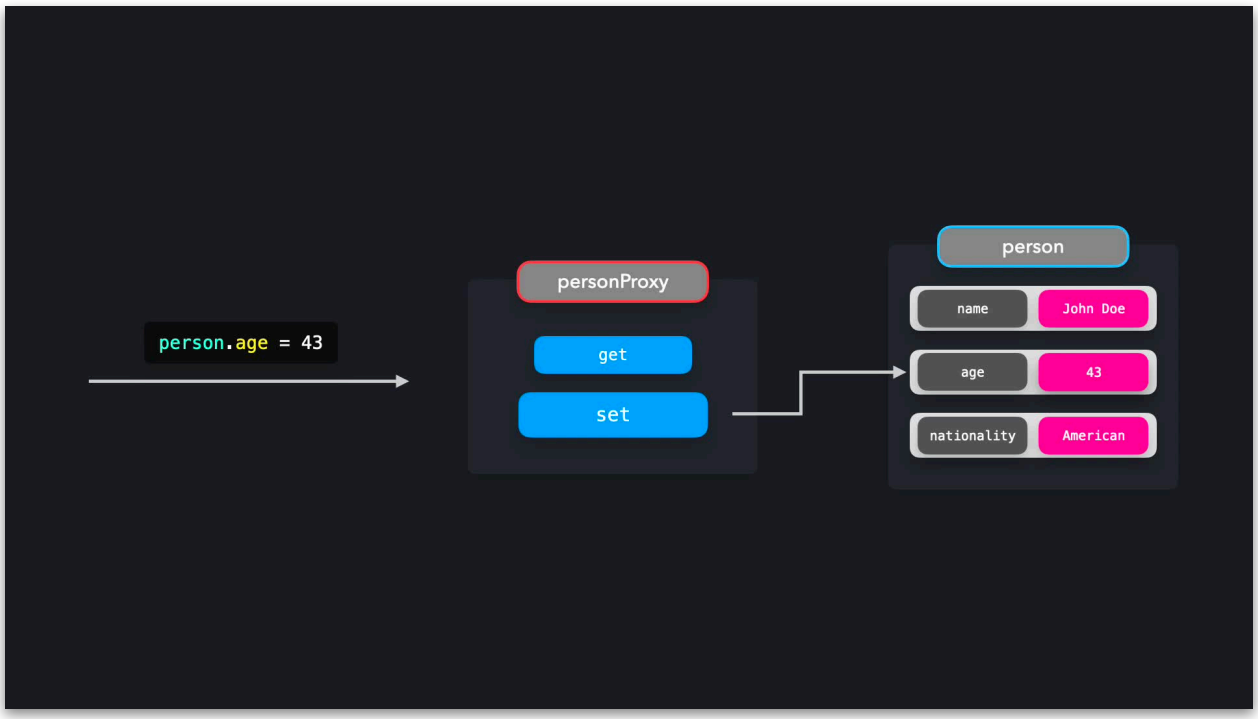
```
const person = {
  name: "John Doe",
  age: 42,
  nationality: "American"
};

const personProxy = new Proxy(person, {});
```

The second argument of `Proxy` is an object that represents the handler. In the handler object, we can define specific behavior based on the type of interaction. Although there are many methods that you can add to the `Proxy` handler, the two most common ones are `get` and `set`:

- `get`: Gets invoked when trying to access a property
- `set`: Gets invoked when trying to modify a property

Effectively, what will end up happening is the following:





Instead of interacting with the person object directly, we'll be interacting with the `personProxy`.

Let's add handlers to the `personProxy`. When trying to modify a property, thus invoking the set method on the proxy, we want it to log the previous value and the new value of the property. When trying to access a property, thus invoking the get a method on the Proxy, we want it to log a more readable sentence that contains the key any value of the property.

```
const personProxy = new Proxy(person, {
  get: (obj, prop) => {
    console.log(`The value of ${prop} is ${obj[prop]}`);
  },
  set: (obj, prop, value) => {
    console.log(`Changed ${prop} from ${obj[prop]} to ${value}`);
    obj[prop] = value;
  }
});
```

Perfect! Let's see what happens when we're trying to modify or retrieve a property.

```
const person = {
  name: "John Doe",
  age: 42,
  nationality: "American"
};

const personProxy = new Proxy(person, {
  get: (obj, prop) => {
    console.log(`The value of ${prop} is ${obj[prop]}`);
  },
  set: (obj, prop, value) => {
    console.log(`Changed ${prop} from ${obj[prop]} to ${value}`);
    obj[prop] = value;
    return true;
  }
});

personProxy.name;
personProxy.age = 43;
```

When accessing the name property, the `personProxy` returned a better sounding sentence: `The value of name is John Doe`. When modifying the `age` property, the Proxy returned the previous and new value of this property: `Changed age from 42 to 43`.

A proxy can be useful to add **validation**. A user shouldn't be able to change person's age to a string value, or give him an empty name. Or if the user is trying to access a property on the object that doesn't exist, we should let the user know.

```
const personProxy = new Proxy(person, {
  get: (obj, prop) => {
    if (!obj[prop]) {
      console.log(
        `Hmm.. this property doesn't seem to exist on the target object`
      );
    } else {
      console.log(`The value of ${prop} is ${obj[prop]}`);
    }
  },
  set: (obj, prop, value) => {
    if (prop === "age" && typeof value !== "number") {
      console.log(`Sorry, you can only pass numeric values for age.`);
    } else if (prop === "name" && value.length < 2) {
      console.log(`You need to provide a valid name.`);
    } else {
      console.log(`Changed ${prop} from ${obj[prop]} to ${value}.`);
      obj[prop] = value;
    }
  }
});
```

The proxy makes sure that we weren't modifying the person object with faulty values, which helps us keep our data pure!

Reflect

JavaScript provides a built-in object called `Reflect`, which makes it easier for us to manipulate the target object when working with proxies.

Previously, we tried to modify and access properties on the target object within the proxy through directly getting or setting the values with bracket notation. Instead, we can use the `Reflect` object. The methods on the `Reflect` object have the same name as the methods on the handler object.

Instead of accessing properties through `obj[prop]` or setting properties through `obj[prop] = value`, we can access or modify properties on the target object through `Reflect.get()` and `Reflect.set()`. The methods receive the same arguments as the methods on the handler object.

```
const personProxy = new Proxy(person, {
  get: (obj, prop) => {
    console.log(`The value of ${prop} is ${Reflect.get(obj, prop)}`);
  },
  set: (obj, prop, value) => {
    console.log(`Changed ${prop} from ${obj[prop]} to ${value}`);
    Reflect.set(obj, prop, value);
  }
});
```

Perfect! We can access and modify the properties on the target object easily with the `Reflect` object.

```
const person = {
  name: "John Doe",
  age: 42,
  nationality: "American"
};

const personProxy = new Proxy(person, {
  get: (obj, prop) => {
    console.log(`The value of ${prop} is ${Reflect.get(obj, prop)}`);
  },
  set: (obj, prop, value) => {
    console.log(`Changed ${prop} from ${obj[prop]} to ${value}`);
    return Reflect.set(obj, prop, value);
  }
});

personProxy.name;
personProxy.age = 43;
personProxy.name = "Jane Doe";
```

Proxies are a powerful way to add control over the behavior of an object. A proxy can have various use-cases: it can help with validation, formatting, notifications, or debugging.

Overusing the `Proxy` object or performing heavy operations on each handler method invocation can easily affect the performance of your application negatively. It's best to not use proxies for performance-critical code.

Provider Pattern

Make data available to multiple child components

In some cases, we want to make available data to many (if not all) components in an application. Although we can pass data to components using props, this can be difficult to do if almost all components in your application need access to the value of the props.

We often end up with something called prop drilling, which is the case when we pass props far down the component tree. Refactoring the code that relies on the props becomes almost impossible, and knowing where certain data comes from is difficult.

Let's say that we have one `App` component that contains certain data. Far down the component tree, we have a `ListItem`, `Header` and `Text` component that all need this data. In order to get this data to these components, we'd have to pass it through multiple layers of components.

In our codebase, that would look something like the following:

```
function App() {
  const data = { ... }

  return (
    <div>
      <SideBar data={data} />
      <Content data={data} />
    </div>
  )
}

const SideBar = ({ data }) => <List data={data} />
const List = ({ data }) => <ListItem data={data} />
const ListItem = ({ data }) => <span>{data.listItem}</span>

const Content = ({ data }) => (
  <div>
    <Header data={data} />
    <Block data={data} />
  </div>
)
const Header = ({ data }) => <div>{data.title}</div>
const Block = ({ data }) => <Text data={data} />
const Text = ({ data }) => <h1>{data.text}</h1>
```

Passing props down this way can get quite messy. If we want to rename the data prop in the future, we'd have to rename it in all components. The bigger your application gets, the trickier *prop drilling* can be.

It would be optimal of we could skip all the layers of components that don't need to use this data. We need to have something that gives the components that need access to the value of data direct access to it, without relying on *prop drilling*.

This is where the Provider Pattern can help us out! With the Provider Pattern, we can make data available to multiple components. Rather than passing that data down each layer through props, we can wrap all components in a `Provider`. A `Provider` is a higher order component provided to us by the a `Context` object. We can create a `Context` object, using the `createContext` method that React provides for us.

The `Provider` receives a value prop, which contains the data that we want to pass down. All components that are wrapped within this provider have access to the value of the value prop.

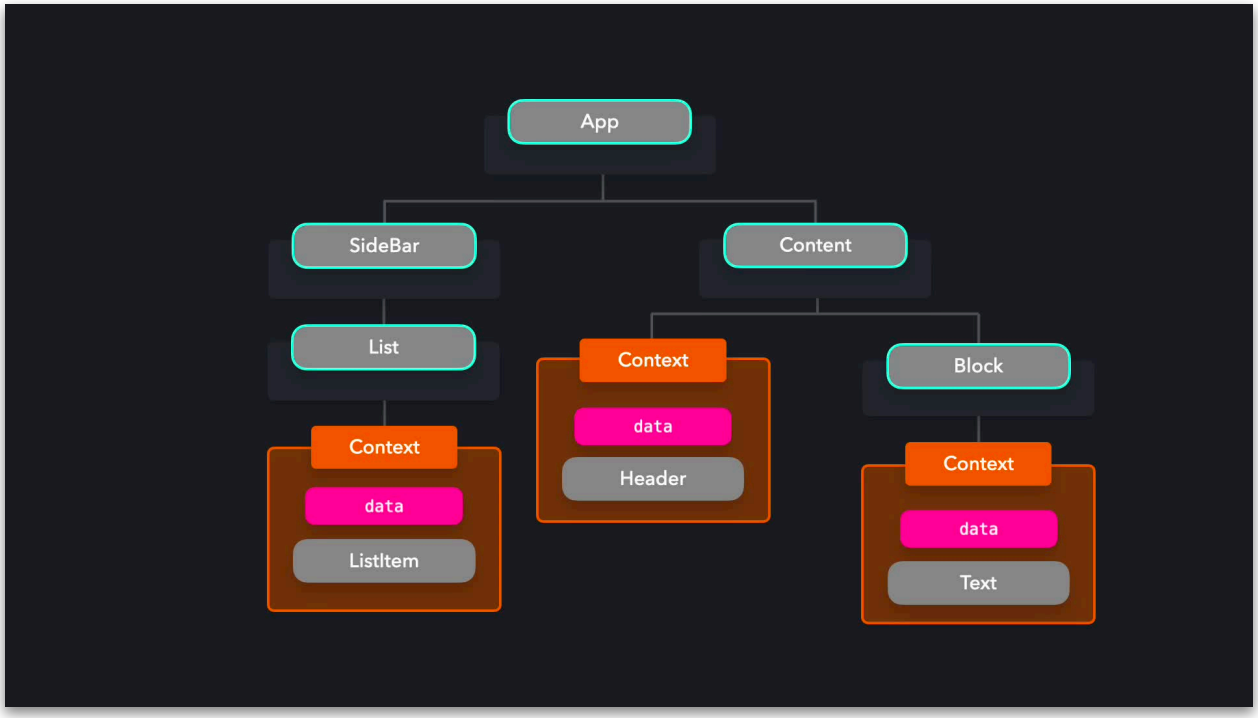
```
const DataContext = React.createContext()

function App() {
  const data = { ... }

  return (
    <div>
      <DataContext.Provider value={data}>
        <SideBar />
        <Content />
      </DataContext.Provider>
    </div>
  )
}
```

We no longer have to manually pass down the data prop to each component!

Each component can get access to the data, by using the `useContext` hook. This hook receives the context that data has a reference with, `DataContext` in this case. The `useContext` hook lets us read and write data to the context object.

```
const DataContext = React.createContext();

function App() {
  const data = { ... }

  return (
    <div>
      <SideBar />
      <Content />
    </div>
  )
}

const SideBar = () => <List />
const Content = () => <div><Header /><Text /></div>

function List() {
  const { data } = React.useContext(DataContext);
  return <ul>{data.list}</ul>;
}

function Text() {
  const { data } = React.useContext(DataContext);
  return <h1>{data.text}</h1>;
}

function Header() {
  const { data } = React.useContext(DataContext);
  return <div>{data.title}</div>;
}
```

The components that aren't using the data value won't have to deal with data at all. We no longer have to worry about passing props down several levels through components that don't need the value of the props, which makes refactoring a lot easier.



The Provider pattern is very useful for sharing global data. A common use case for the provider pattern is sharing a theme UI state with many components.

Say we have a simple app that shows a list.

App.js

```
import React from "react";
import "./styles.css";

import List from "./List";
import Toggle from "./Toggle";

export default function App() {
  return (
    <div className="App">
      <Toggle />
      <List />
    </div>
  );
}
```

List.js

```
import React from "react";
import ListItem from "./ListItem";

export default function Boxes() {
  return (
    <ul className="list">
      {new Array(10).fill(0).map((x, i) => (
        <ListItem key={i} />
      ))}
    </ul>
  );
}
```

We want the user to be able to switch between light mode and dark mode, by toggling the switch. When the user switches from dark- to light mode and vice versa, the background color and text color should change! Instead of passing the current theme value down to each component, we can wrap the components in a `ThemeProvider`, and pass the current theme colors to the provider.

```
export const ThemeContext = React.createContext();

const themes = {
  light: { background: "#fff",  color: "#000" },
  dark: { background: "#171717", color: "#fff" }
};

export default function App() {
  const [theme, setTheme] = useState("dark");

  function toggleTheme() {
    setTheme(theme === "light" ? "dark" : "light");
  }

  const providerValue = { theme: themes[theme], toggleTheme };

  return (
    <div className={`App theme-${theme}`}>
      <ThemeContext.Provider value={providerValue}>
        <Toggle />
        <List />
      </ThemeContext.Provider>
    </div>
  );
}
```

Since the `Toggle` and `List` components are both wrapped within the `ThemeContext` provider, we have access to the values `theme` and `toggleTheme` that are passed as a value to the provider.

Within the `Toggle` component, we can use the `toggleTheme` function to update the theme accordingly.

```
import React, { useContext } from "react";
import { ThemeContext } from "./App";

export default function Toggle() {
  const theme = useContext(ThemeContext);

  return (
    <label className="switch">
      <input type="checkbox" onClick={theme.toggleTheme} />
      <span className="slider round" />
    </label>
  );
}
```

The `List` component itself doesn't care about the current value of the theme. However, the `ListItem` components do! We can use the theme context directly within the `ListItem`.

```jsx
import React, { useContext } from "react";
import { ThemeContext } from "./App";

export default function ListItem() {
  const theme = useContext(ThemeContext);

  return <li style={theme.theme}>...</li>;
}
```

Perfect! We didn't have to pass down any data to components that didn't care about the current value of the theme.

Hooks

We can create a hook to provide context to components. Instead of having to import `useContext` and the `Context` in each component, we can use a hook that returns the context we need.

```
function useThemeContext() {
  const theme = useContext(ThemeContext);
  return theme;
}
```

To make sure that it's a valid theme, let's throw an error if `useContext(ThemeContext)` returns a falsy value.

```
function useThemeContext() {
  const theme = useContext(ThemeContext);
  if (!theme) {
    throw new Error("useThemeContext must be used within ThemeProvider");
  }
  return theme;
}
```

Instead of wrapping the components directly with the `ThemeContext.Provider` component, we can create a HOC that wraps this component to provide its values. This way, we can separate the context logic from the rendering components, which improves the reusability of the provider.

```
function ThemeProvider({children}) {
  const [theme, setTheme] = useState("dark");

  function toggleTheme() {
    setTheme(theme === "light" ? "dark" : "light");
  }

  const providerValue = { theme: themes[theme], toggleTheme };

  return (
    <ThemeContext.Provider value={providerValue}>
      {children}
    </ThemeContext.Provider>
  );
}

export default function App() {
  return (
    <div className={`App theme-${theme}`}>
      <ThemeProvider>
        <Toggle />
        <List />
      </ThemeProvider>
    </div>
  );
}
```

Each component that needs to have access to the ThemeContext, can now simply use the useThemeContext hook.

```jsx
export default function ListItem() {
  const theme = useThemeContext();

  return <li style={theme.theme}>...</li>;
}
```

Prototype Pattern

Share properties among many objects of the same type

The prototype pattern is a useful way to share properties among many objects of the same type. The prototype is an object that's native to JavaScript, and can be accessed by objects through the prototype chain.

In our applications, we often have to create many objects of the same type. A useful way of doing this is by creating multiple instances of an ES6 class.

Let's say we want to create many dogs! In our example, dogs can't do that much: they simply have a name, and they can bark!

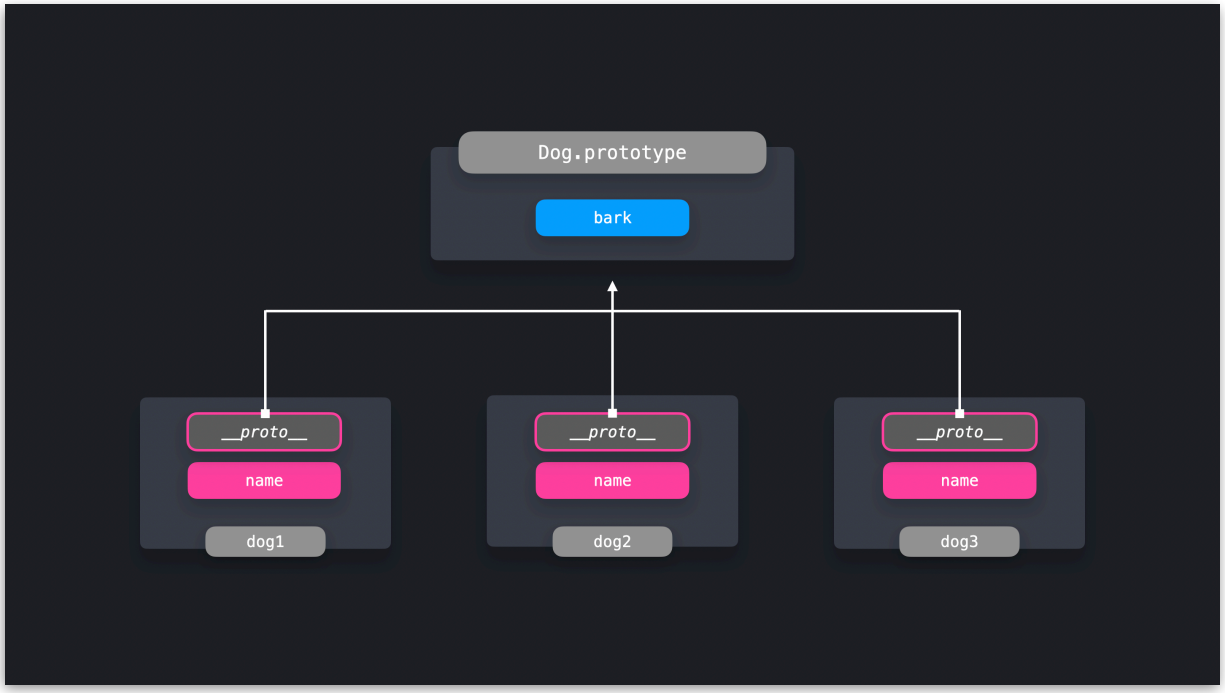
```
class Dog {
  constructor(name) {
    this.name = name;
  }

  bark() {
    return `Woof!`;
  }
}

const dog1 = new Dog("Daisy");
const dog2 = new Dog("Max");
const dog3 = new Dog("Spot");
```

Notice here how the constructor contains a `name` property, and the class itself contains a `bark` property. When using ES6 classes, all properties that are defined on the class itself, `bark` in this case, are automatically added to the prototype.

We can see the prototype directly through accessing the prototype property on a constructor, or through the `__proto__` property on any instance.

```
console.log(Dog.prototype);
// constructor: ƒ Dog(name, breed) bark: ƒ bark()

console.log(dog1.__proto__);
// constructor: ƒ Dog(name, breed) bark: ƒ bark()
```

The value of `__proto__` on any instance of the constructor, is a direct reference to the constructor's prototype! Whenever we try to access a property on an object that doesn't exist on the object directly, JavaScript will go down the prototype chain to see if the property is available within the prototype chain.



The prototype pattern is very powerful when working with objects that should have access to the same properties. Instead of creating a duplicate of the property each time, we can simply add the property to the prototype, since all instances have access to the prototype object.

Since all instances have access to the prototype, it's easy to add properties to the prototype even after creating the instances.

Say that our dogs shouldn't only be able to bark, but they should also be able to play! We can make this possible by adding a play property to the prototype.

```
class Dog {
  constructor(name) {
    this.name = name;
  }

  bark() {
    return `Woof!`;
  }
}

const dog1 = new Dog("Daisy");
const dog2 = new Dog("Max");
const dog3 = new Dog("Spot");

Dog.prototype.play = () => console.log("Playing now!");

dog1.play();
```

The term **prototype chain** indicates that there could be more than one step. Indeed! So far, we've only seen how we can access properties that are directly available on the first object that `__proto__` has a reference to. However, prototypes themselves also have a `__proto__` object!

Let's create another type of dog, a super dog! This dog should inherit everything from a normal `Dog`, but it should also be able to fly. We can create a super dog by extending the `Dog` class and adding a `fly` method.

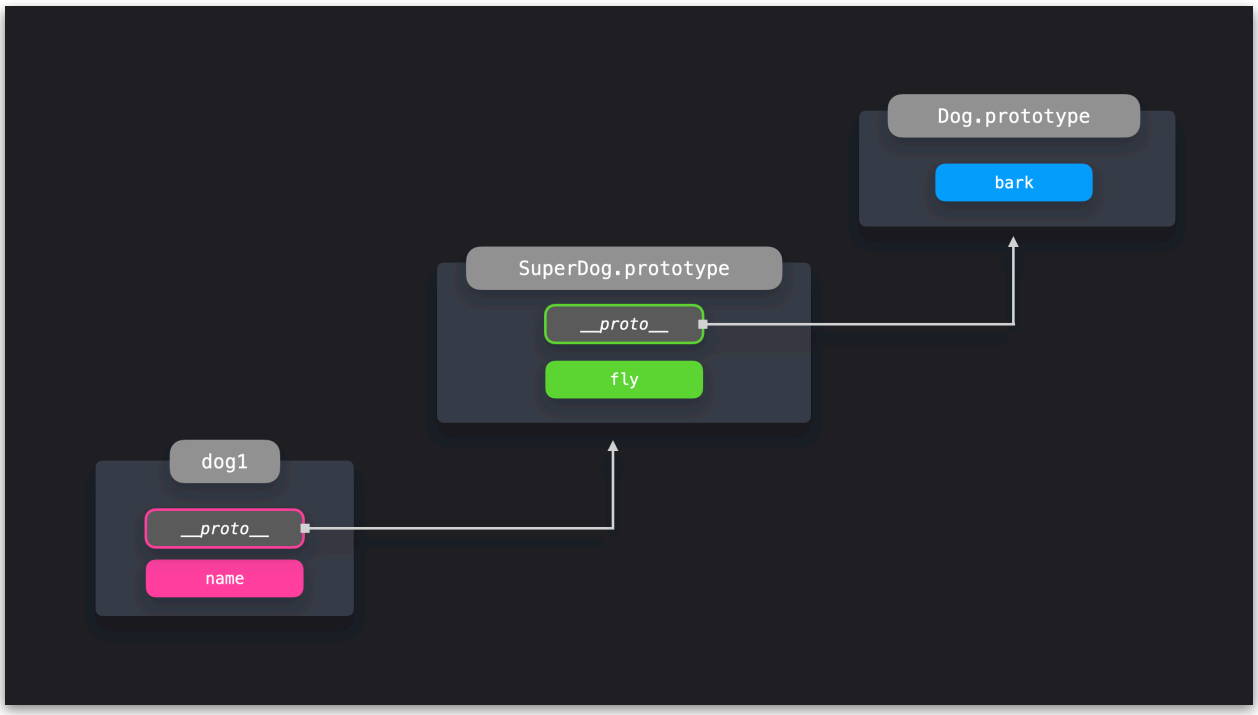
```
class SuperDog extends Dog {
  constructor(name) {
    super(name);
  }

  fly() {
    return "Flying!";
  }
}
```

Let's create a flying dog called Daisy, and let her bark and fly!

```
class Dog {
  constructor(name) {
    this.name = name;
  }

  bark() {
    console.log("Woof!");
  }
}

class SuperDog extends Dog {
  constructor(name) {
    super(name);
  }

  fly() {
    console.log(`Flying!`);
  }
}

const dog1 = new SuperDog("Daisy");
dog1.bark();
dog1.fly();
```

We have access to the `bark` method, as we extended the `Dog` class. The value of `__proto__` on the prototype of `SuperDog` points to the `Dog.prototype` object!

It gets clear why it's called a prototype chain: when we try to access a property that's not directly available on the object, JavaScript recursively walks down all the objects that `__proto__` points to, until it finds the property!



`Object.create`

The `Object.create` method lets us create a new object, to which we can explicitly pass the value of its prototype.

```
const dog = {
  bark() {
    return `Woof!`;
  }
};

const pet1 = Object.create(dog);
```

Although `pet1` itself doesn't have any properties, it does have access to properties on its prototype chain! Since we passed the `dog` object as `pet1's` prototype, we can access the `bark` property.

```
const dog = {
  bark() {
    console.log(`Woof!`);
  }
};

const pet1 = Object.create(dog);

pet1.bark(); // Woof!
console.log("Direct properties on pet1: ", Object.keys(pet1));
console.log("Properties on pet1's prototype: ",
Object.keys(pet1.__proto__));
```

Perfect! `Object.create` is a simple way to let objects directly inherit properties from other objects, by specifying the newly created object's prototype. The new object can access the new properties by walking down the prototype chain.

The prototype pattern allows us to easily let objects access and inherit properties from other objects. Since the prototype chain allows us to access properties that aren't directly defined on the object itself, we can avoid duplication of methods and properties, thus reducing the amount of memory used.

Container/ Presentational Pattern

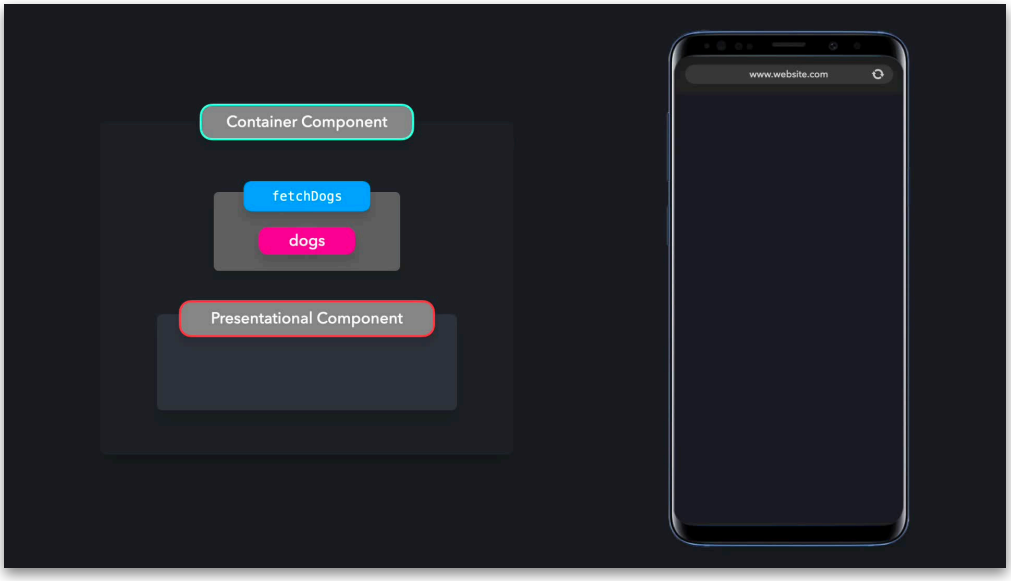
Enforce separation of concerns by separating the view from the application logic

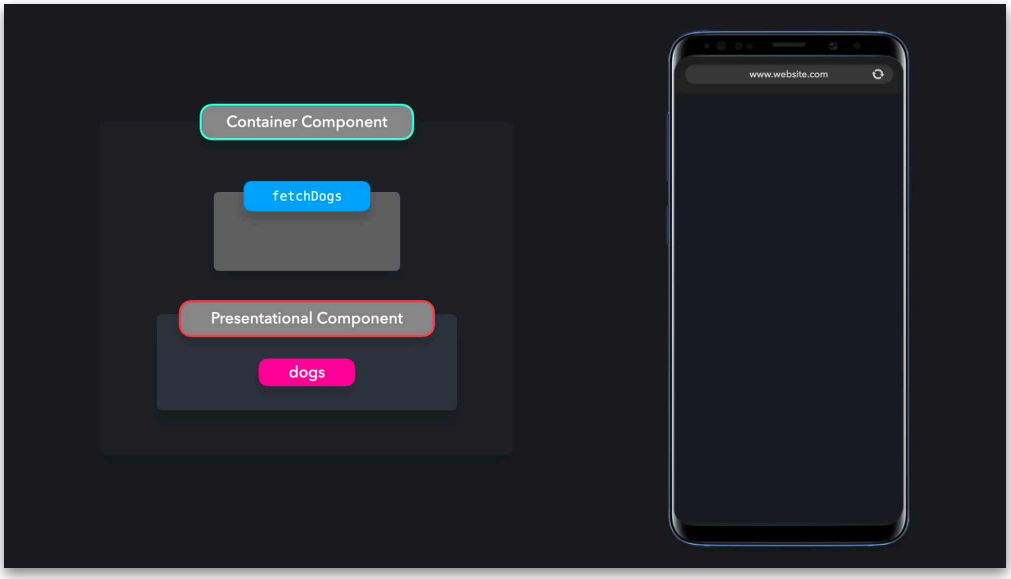
In React, one way to enforce separation of concerns is by using the Container/Presentational pattern. With this pattern, we can separate the view from the application logic.

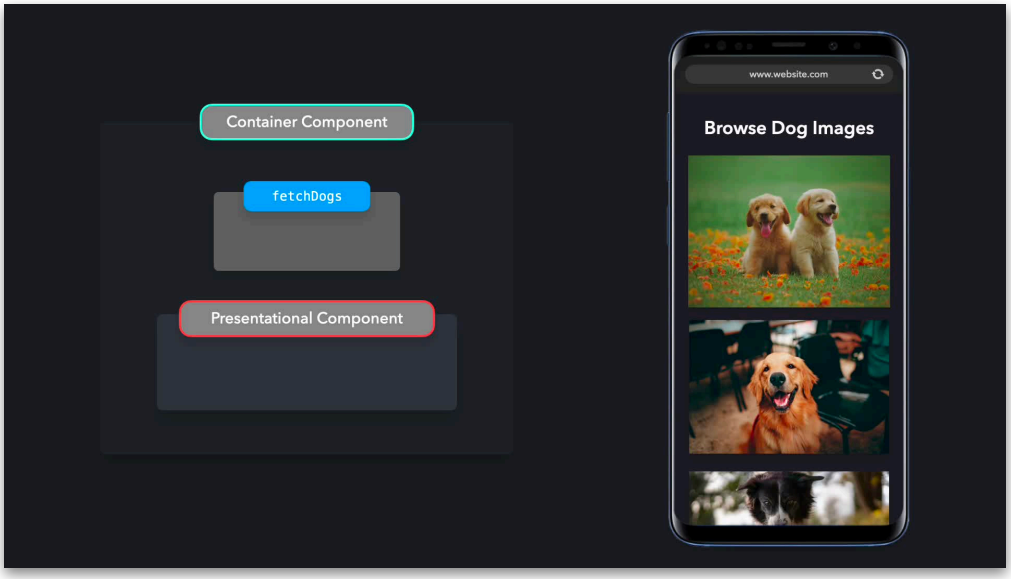
Let's say we want to create an application that fetches 6 dog images, and renders these images on the screen. Ideally, we want to enforce separation of concerns by separating this process into two parts:

1. **Presentational Components**: Components that care about how data is shown to the user. In this example, that's the rendering the list of dog images.

2. **Container Components**: Components that care about what data is shown to the user. In this example, that's fetching the dog images.







Fetching the dog images deals with application logic, whereas displaying the images only deals with the view.

Presentational Component

A presentational component receives its data through props. Its primary function is to simply display the data it receives the way we want them to, including styles, without modifying that data.

Let's take a look at the example that displays the dog images. When rendering the dog images, we simply want to map over each dog image that was fetched from the API, and render those images. In order to do so, we can create a functional component that receives the data through props, and renders the data it received.

DogImages.js

```
import React from "react";

export default function DogImages({ dogs }) {
  return dogs.map((dog, i) => <img src={dog} key={i} alt="Dog" />);
}
```

The `DogsImages` component is a presentational component. Presentational components are usually stateless: they do not contain their own React state, unless they need a state for UI purposes. The data they receive, is not altered by the presentational components themselves.

Presentational components receive their data from container components.

Container Components

The primary function of container components is to pass data to presentational components, which they contain. Container components themselves usually don't render any other components besides the presentational components that care about their data. Since they don't render anything themselves, they usually do not contain any styling either.

In our example, we want to pass dog images to the `DogsImages` presentational component. Before being able to do so, we need to fetch the images from an external API. We need to create a container component that fetches this data, and passes this data to the presentational component `DogsImages` in order to display it on the screen.

DogImagesContainer.js

```
import React from "react";
import DogImages from "./DogImages";

export default class DogImagesContainer extends React.Component {
  constructor() {
    super();
    this.state = {
      dogs: []
    };
  }

  componentDidMount() {
    fetch("https://dog.ceo/api/breed/labrador/images/random/6")
      .then(res => res.json())
      .then(({ message }) => this.setState({ dogs: message }));
  }

  render() {
    return <DogImages dogs={this.state.dogs} />;
  }
}
```

Combining these two components together makes it possible to separate handling application logic with the view.

Hooks

In many cases, the Container/Presentational pattern can be replaced with React Hooks. The introduction of Hooks made it easy for developers to add statefulness without needing a container component to provide that state.

Instead of having the data fetching logic in the `DogsImagesContainer` component, we can create a custom hook that fetches the images, and returns the array of dogs.

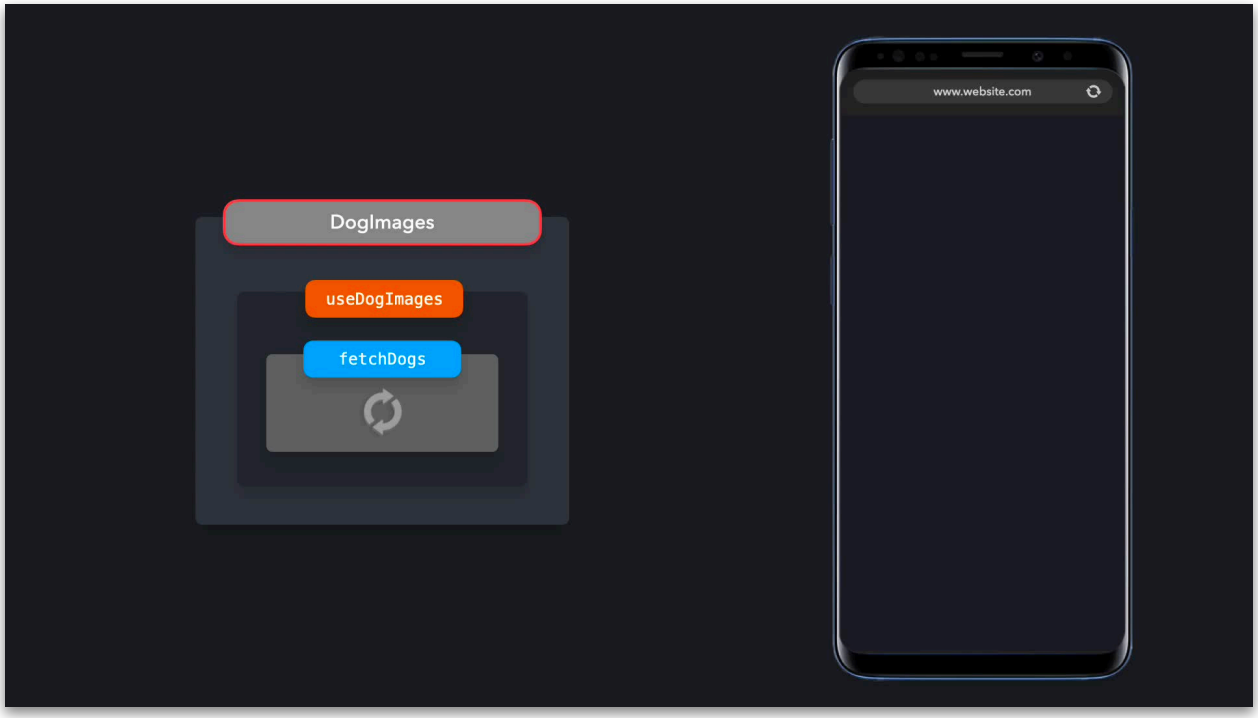
```
export default function useDogImages() {
  const [dogs, setDogs] = useState([]);

  useEffect(() => {
    fetch("https://dog.ceo/api/breed/labrador/images/random/6")
      .then(res => res.json())
      .then(({ message }) => setDogs(message));
  }, []);

  return dogs;
}
```

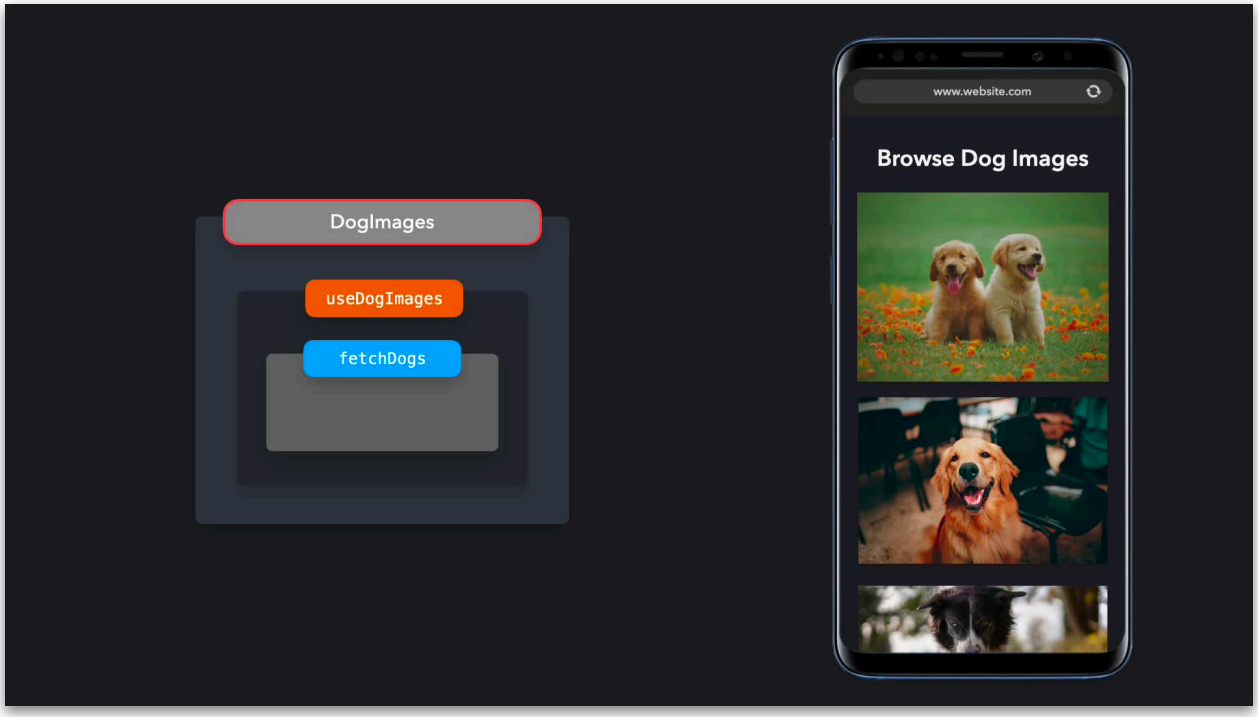
By using this hook, we no longer need the wrapping `DogsImagesContainer` container component to fetch the data, and send this to the presentational `DogsImages` component. Instead, we can use this hook directly in our presentational `DogsImages` component!

```jsx
import React from "react";
import useDogImages from "./useDogImages";

export default function DogImages() {
  const dogs = useDogImages();

  return dogs.map((dog, i) => <img src={dog} key={i} alt="Dog" />);
}
```

```jsx
import { useState, useEffect } from "react";

export default function useDogImages() {
  const [dogs, setDogs] = useState([]);

  useEffect(() => {
    async function fetchDogs() {
      const res = await fetch(
        "https://dog.ceo/api/breed/labrador/images/random/6"
      );
      const { message } = await res.json();
      setDogs(message);
    }

    fetchDogs();
  }, []);

  return dogs;
}
```

By using the `useDogImages` hook, we still separated the application logic from the view. We're simply using the returned data from the `useDogImages` hook, without modifying that data within the `DogImages` component.





Hooks make it easy to separate logic and view in a component, just like the Container/Presentational pattern. It saves us the extra layer that was necessary in order to wrap the presentational component within the container component.

Pros

There are many benefits to using the Container/Presentational pattern.

The Container/Presentational pattern encourages the separation of concerns. Presentational components can be pure functions which are responsible for the UI, whereas container components are responsible for the state and data of the application. This makes it easy to enforce the separation of concerns

Presentational components are easily made reusable, as they simply display data without altering this data. We can reuse the presentational components throughout our application for different purposes.

Since presentational components don't alter the application logic, the appearance of presentational components can easily be altered by someone without knowledge of the codebase, for example a designer. If the presentational component was reused in many parts of the application, the change can be consistent throughout the app.

Testing presentational components is easy, as they are usually pure functions. We know what the components will render based on which data we pass, without having to mock a data store.

Cons

The Container/Presentational pattern makes it easy to separate application logic from rendering logic. However, Hooks make it possible to achieve the same result without having to use the Container/Presentational pattern, and without having to rewrite a stateless functional component into a class component. Note that today, we don't need to create class components to use state anymore.

Although we can still use the Container/Presentational pattern, even with React Hooks, this pattern can easily be an overkill in smaller sized application.

Observer Pattern

Use observables to notify subscribers when an event occurs

With the observer pattern, we can subscribe certain objects, the observers, to another object, called the observable. Whenever an event occurs, the observable notifies all its observers!

An observable object usually contains 3 important parts:

- observers: an array of observers that will get notified whenever a specific event occurs
- `subscribe()`: a method in order to add observers to the observers list
- `unsubscribe()`: a method in order to remove observers from the observers list
- `notify()`: a method to notify all observers whenever a specific event occurs

Perfect, let's create an observable! An easy way of creating one, is by using an ES6 class.

```
class Observable {
  constructor() {
    this.observers = [];
  }

  subscribe(func) {
    this.observers.push(func);
  }

  unsubscribe(func) {
    this.observers = this.observers.filter(observer => observer !== func);
  }

  notify(data) {
    this.observers.forEach(observer => observer(data));
  }
}
```

Awesome! We can now add observers to the list of observers with the subscribe method, remove the observers with the unsubscribe method, and notify all subscribes with the notify method.

Let's build something with this observable. We have a very basic app that only consists of two components: a `Button`, and a `Switch`.

```
export default function App() {
  return (
    <div className="App">
      <Button>Click me!</Button>
      <FormControlLabel control={<Switch />} />
    </div>
  );
}
```

We want to keep track of the user interaction with the application. Whenever a user either clicks the button or toggles the switch, we want to log this event with the timestamp. Besides logging it, we also want to create a toast notification that shows up whenever an event occurs!

Whenever the user invokes the `handleClick` or `handleToggle` function, the functions invoke the notify method on the observer. The notify method notifies all subscribers with the data that was passed by the `handleClick` or `handleToggle` function!

First, let's create the `logger` and `tastily` functions. These functions will eventually receive data from the notify method.

```
import { ToastContainer, toast } from "react-toastify";

function logger(data) {
  console.log(`${Date.now()} ${data}`);
}

function toastify(data) {
  toast(data);
}

export default function App() {
  return (
    <div className="App">
      <Button>Click me!</Button>
      <FormControlLabel control={<Switch />} />
      <ToastContainer />
    </div>
  );
}
```

Currently, the `logger` and `toastify` functions are unaware of observable: the observable can't notify them yet! In order to make them observers, we'd have to subscribe them, using the subscribe method on the observable!

```
import { ToastContainer, toast } from "react-toastify";

function logger(data) {
  console.log(`${Date.now()} ${data}`);
}

function toastify(data) {
  toast(data);
}

observable.subscribe(logger);
observable.subscribe(toastify);

export default function App() {
  return (
    <div className="App">
      <Button>Click me!</Button>
      <FormControlLabel control={<Switch />} />
      <ToastContainer />
    </div>
  );
}
```

Whenever an event occurs, the `logger` and `toastify` functions will get notified. Now we just need to implement the functions that actually notify the observable: the `handleClick` and `handleToggle` functions! These functions should invoke the notify method on the observable, and pass the data that the observers should receive.

```
import { ToastContainer, toast } from "react-toastify";

function logger(data) {
  console.log(`${Date.now()} ${data}`);
}

function toastify(data) {
  toast(data);
}

observable.subscribe(logger);
observable.subscribe(toastify);

export default function App() {
  function handleClick() {
    observable.notify("User clicked button!");
  }

  function handleToggle() {
    observable.notify("User toggled switch!");
  }

  return (
    <div className="App">
      <Button>Click me!</Button>
      <FormControlLabel control={<Switch />} />
      <ToastContainer />
    </div>
  );
}
```

Awesome! We just finished the entire flow: `handleClick` and `handleToggle` invoke the notify method on the observer with the data, after which the observer notifies the subscribers: the `logger` and `toastify` functions in this case.

Whenever a user interacts with either of the components, both the logger and the `toastify` functions will get notified with the data that we passed to the notify method!

Observable.js

```
class Observable {
  constructor() {
    this.observers = [];
  }

  subscribe(f) {
    this.observers.push(f);
  }

  unsubscribe(f) {
    this.observers = this.observers.filter(subscriber => subscriber !==
f);
  }

  notify(data) {
    this.observers.forEach(observer => observer(data));
  }
}

export default new Observable();
```

App.js

```
import React from "react";
import { Button, Switch, FormControlLabel } from "@material-ui/core";
import { ToastContainer, toast } from "react-toastify";
import observable from "./Observable";

function handleClick() {
  observable.notify("User clicked button!");
}

function handleToggle() {
  observable.notify("User toggled switch!");
}

function logger(data) {
  console.log(`${Date.now()} ${data}`);
}

function toastify(data) {
  toast(data, {
    position: toast.POSITION.BOTTOM_RIGHT,
    closeButton: false,
    autoClose: 2000
  });
}

observable.subscribe(logger);
observable.subscribe(toastify);

export default function App() {
  return (
    <div className="App">
      <Button variant="contained" onClick={handleClick}>
        Click me!
      </Button>
      <FormControlLabel
        control={<Switch name="" onChange={handleToggle} />}
        label="Toggle me!"
      />
      <ToastContainer />
    </div>
  );
}
```

Although we can use the observer pattern in many ways, it can be very useful when working with asynchronous, event-based data. Maybe you want certain components to get notified whenever certain data has finished downloading, or whenever users sent new messages to a message board and all other members should get notified.

Pros

Using the observer pattern is a great way to enforce separation of concerns and the single-responsiblity principle. The observer objects aren't tightly coupled to the observable object, and can be (de)coupled at any time. The observable object is responsible for monitoring the events, while the observers simply handle the received data.

Cons

If an observer becomes too complex, it may cause performance issues when notifying all subscribers.

Case study

A popular library that uses the observable pattern is RxJS.

ReactiveX combines the Observer pattern with the Iterator pattern and functional programming with collections to fill the need for an ideal way of managing sequences of events. - RxJS

With RxJS, we can create observables and subscribe to certain events! Let's look at an example that's covered in their documentation, which logs whether a user was dragging in the document or not.

```
import React from "react";
import ReactDOM from "react-dom";
import { fromEvent, merge } from "rxjs";
import { sample, mapTo } from "rxjs/operators";

import "./styles.css";

merge(
  fromEvent(document, "mousedown").pipe(mapTo(false)),
  fromEvent(document, "mousemove").pipe(mapTo(true))
)
  .pipe(sample(fromEvent(document, "mouseup")))
  .subscribe(isDragging => {
    console.log("Were you dragging?", isDragging);
  });

ReactDOM.render(
  <div className="App">
    Click or drag anywhere and check the console!
  </div>,
  document.getElementById("root")
);
```

Module Pattern

Split up your code into smaller, reusable pieces

As your application and codebase grow, it becomes increasingly important to keep your code maintainable and separated. The module pattern allows you to split up your code into smaller, reusable pieces.

Besides being able to split your code into smaller reusable pieces, modules allow you to keep certain values within your file private. Declarations within a module are scoped (encapsulated) to that module , by default. If we don't explicitly export a certain value, that value is not available outside that module. This reduces the risk of name collisions for values declared in other parts of your codebase, since the values are not available on the global scope.

ES2015 Modules

ES2015 introduced built-in JavaScript modules. A module is a file containing JavaScript code, with some difference in behavior compared to a normal script.

Let's look at an example of a module called `math.js`, containing mathematical functions.

math.js

```
function add(x, y) {
  return x + y;
}
function multiply(x) {
  return x * 2;
}
function subtract(x, y) {
  return x - y;
}
function square(x) {
  return x * x;
}
```

We have a `math.js` file containing some simple mathematical logic. We have functions that allow users to add, multiply, subtract, and get the square of values that they pass.

However, we don't just want to use these functions in the `math.js` file, we want to be able to reference them in the `index.js` file!

In order to make the functions from `math.js` available to other files, we first have to export them. In order to export code from a module, we can use the `export` keyword.

One way of exporting the functions, is by using named exports: we can simply add the `export` keyword in front of the parts that we want to publicly expose. In this case, we'll want to add the export keyword in front of every function, since `index.js` should have access to all four functions.

math.js

```
export function add(x, y) {
  return x + y;
}

export function multiply(x) {
  return x * 2;
}

export function subtract(x, y) {
  return x - y;
}

export function square(x) {
  return x * x;
}
```

We just made the `add`, `multiply`, `subtract`, and `square` functions exportable! However, just exporting the values from a module is not enough to make them publicly available to all files. In order to be able to use the exported values from a module, you have to explicitly import them in the file that needs to reference them.

We have to import the values on top of the `index.js` file, by using the import keyword. To let javascript know from which module we want to import these functions, we need to add a from value and the relative path to the module.

index.js

```
import { add, multiply, subtract, square } from "./math.js";
```

We just imported the four functions from the `math.js` module in the `index.js` file! Let's try and see if we can use the functions now!

```
import { add, multiply, subtract, square } from "./math";

console.log(add(2, 3));
console.log(multiply(2));
console.log(subtract(2, 3));
console.log(square(2));
```

The reference error is gone, we can now use the exported values from the module!

A great benefit of having modules, is that we only have access to the values that we explicitly exported using the `export` keyword. Values that we didn't explicitly export using the `export` keyword, are only available within that module.

Let's create a value that should only be referenceable within the `math.js` file, called `privateValue`.

math.js

```
const privateValue = "This is a value private to the module!";

export function add(x, y) {
  return x + y;
}

export function multiply(x) {
  return x * 2;
}

export function subtract(x, y) {
  return x - y;
}

export function square(x) {
  return x * x;
}
```

Notice how we didn't add the export keyword in front of privateValue. Since we didn’t export the privateValue variable, we don’t have access to this value outside of the `math.js` module!

By keeping the value private to the module, there is a reduced risk of accidentally polluting the global scope. You don't have to fear that you will accidentally overwrite values created by developers using your module, that may have had the same name as your private value: it prevents naming collisions.

Sometimes, the names of the exports could collide with local values.

```
import { add, multiply, subtract, square } from "./math.js";

function add(...args) {
  return args.reduce((acc, cur) => cur + acc);
} /* Error: add has  already been declared */

function multiply(...args) {
  return args.reduce((acc, cur) => cur * acc);
}
/* Error: multiply has already been declared */
```

In this case, we have functions called add and multiply in `index.js`. If we would import values with the same name, it would end up in a naming collision: `add` and `multiply` have already been declared! Luckily, we can rename the imported values, by using the as keyword.

Let's rename the imported `add` and `multiply` functions to `addValues` and `multiplyValues`.

index.js

```
import {
  add as addValues,
  multiply as multiplyValues,
  subtract,
  square
} from "./math.js";

function add(...args) {
  return args.reduce((acc, cur) => cur + acc);
}

function multiply(...args) {
  return args.reduce((acc, cur) => cur * acc);
}

/* From math.js module */
addValues(7, 8);
multiplyValues(8, 9);
subtract(10, 3);
square(3);

/* From index.js file */
add(8, 9, 2, 10);
multiply(8, 9, 2, 10);
```

Besides named exports, which are exports defined with just the `export` keyword, you can also use a default export. You can only have one default export per module.

Let's make the add function our default export, and keep the other functions as named exports. We can export a default value, by adding `export default` in front of the value.

math.js

```
export default function add(x, y) {
  return x + y;
}

export function multiply(x) {
  return x * 2;
}

export function subtract(x, y) {
  return x - y;
}

export function square(x) {
  return x * x;
}
```

The difference between named exports and default exports, is the way the value is exported from the module, effectively changing the way we have to import the value.

Previously, we had to use the brackets for our named exports: `import { module } from 'module'`. With a default export, we can import the value without the brackets: import module from 'module'.

index.js

```
import * as math from "./math.js";

math.default(7, 8);
math.multiply(8, 9);
math.subtract(10, 3);
math.square(3);
```

The value that's been imported from a module without the brackets, is always the value of the default export, if there is a default export available.

Since JavaScript knows that this value is always the value that was exported by default, we can give the imported default value another name than the name we exported it with. Instead of importing the add function using the name add, we can call it `addValues`, for example.

index.js

```
import addValues, { multiply, subtract, square } from "./math.js";

addValues(7, 8);
multiply(8, 9);
subtract(10, 3);
square(3);
```

Even though we exported the function called `add`, we can import it calling it anything we like, since JavaScript knows you are importing the default export.

We can also import all exports from a module, meaning all named exports and the default export, by using an asterisk `*` and giving the name we want to import the module as. The value of the import is equal to an object containing all the imported values.

Say that you want to import the entire module as `math`.

index.js

```
import * as math from "./math.js";
```

The imported values are properties on the `math` object.

index.js

```
import * as math from "./math.js";

math.default(7, 8);
math.multiply(8, 9);
math.subtract(10, 3);
math.square(3);
```

In this case, we're importing all exports from a module. Be careful when doing this, since you may end up unnecessarily importing values. Using the ∗ only imports all exported values. Values private to the module are still not available in the file that imports the module, unless you explicitly exported them.

React

When building applications with React, you often have to deal with a large amount of components. Instead of writing all of these components in one file, we can separate the components in their own files, essentially creating a module for each component.

We have a basic todo-list, containing a list, list items, an input field, and a button.

App.js

```
import React from "react";
import { render } from "react-dom";

import { TodoList } from "./components/TodoList";
import "./styles.css";

render(
  <div className="App">
    <TodoList />
  </div>,
  document.getElementById("root")
);
```

Button.js

```
import React from "react";
import Button from "@material-ui/core/Button";

export default function CustomButton(props) {
  return (
    <Button {...props}>
      {props.children}
    </Button>
  );
}
```

Input.js

```
import React from "react";
import Input from "@material-ui/core/Input";

export default function CustomInput(props, { variant = "standard" }) {
  return (
    <Input
      {...props}
      variant={variant}
      placeholder="Type..."
    />
  );
}
```

We just split our components in their separate files:

- `TodoList.js` for the `List` component
- `Button.js` for the customized `Button` component
- `Input.js` for the customized `Input` component.

Throughout the app, we don't want to use the default `Button` and `Input` component, imported from the material-ui library. Instead, we want to use our custom version of the components, by adding custom styles to it defined in the styles object in their files.
Rather than importing the default `Button` and `Input` component each time in our application and adding custom styles to it over and over, we can now simply import the default `Button` and `Input` component once, add styles, and export our custom component.

Dynamic import

When importing all modules on the top of a file, all modules get loaded before the rest of the file. In some cases, we only need to import a module based on a certain condition. With a dynamic import, we can import modules on demand.

```
import("module").then(module => {
  module.default();
  module.namedExport();
});

// Or with async/await
(async () => {
  const module = await import("module");
  module.default();
  module.namedExport();
})();
```

Let's dynamically import the `math.js` example used in the previous paragraphs. The module only gets loaded, if the user clicks on the button.

```
const button = document.getElementById("btn");

button.addEventListener("click", () => {
  import("./math.js").then((module) => {
    console.log("Add: ", module.add(1, 2));
    console.log("Multiply: ", module.multiply(3, 2));

    const button = document.getElementById("btn");
    button.innerHTML = "Check the console";
  });
});
```

By dynamically importing modules, we can reduce the page load time. We only have to load, parse, and compile the code that the user really needs, when the user needs it.

With the module pattern, we can encapsulate parts of our code that should not be publicly exposed. This prevents accidental name collision and global scope pollution, which makes working with multiple dependencies and namespaces less risky. In order to be able to use ES2015 modules in all JavaScript runtimes, a transpiler such as Babel is needed.

Mixin Pattern

Add functionality to objects or classes without inheritance

A mixin is an object that we can use in order to add reusable functionality to another object or class, without using inheritance. We can't use mixins on their own: their sole purpose is to add functionality to objects or classes without inheritance.

Let's say that for our application, we need to create multiple dogs. However, the basic dog that we create doesn't have any properties but a name property.

```
class Dog {
  constructor(name) {
    this.name = name;
  }
}
```

A dog should be able to do more than just have a name. It should be able to bark, wag its tail, and play! Instead of adding this directly to the `Dog`, we can create a mixin that provides the `bark`, `wagTail` and `play` property for us.

```
const dogFunctionality = {
  bark: () => console.log("Woof!"),
  wagTail: () => console.log("Wagging my tail!"),
  play: () => console.log("Playing!")
};
```

We can add the `dogFunctionality` mixin to the `Dog` prototype with the Object.assign method. This method lets us add properties to the target object: `Dog.prototype` in this case. Each new instance of `Dog` will have access to the the properties of `dogFunctionality`, as they're added to the `Dog`'s prototype!

```
class Dog {
  constructor(name) {
    this.name = name;
  }
}

const dogFunctionality = {
  bark: () => console.log("Woof!"),
  wagTail: () => console.log("Wagging my tail!"),
  play: () => console.log("Playing!")
};

Object.assign(Dog.prototype, dogFunctionality);
```

Let's create our first pet, `pet1`, called Daisy. As we just added the `dogFunctionality` mixin to the `Dog`'s prototype, Daisy should be able to walk, wag her tail, and play!

```
const pet1 = new Dog("Daisy");

pet1.name; // Daisy
pet1.bark(); // Woof!
pet1.play(); // Playing!
```

Perfect! Mixins make it easy for us to add custom functionality to classes or objects without using inheritance.

Although we can add functionality with mixins without inheritance, mixins themselves can use inheritance!

Most mammals (besides dolphins.. and maybe some more) can walk and sleep as well. A dog is a mammal, and should be able to walk and sleep! Let's create a `animalFunctionality` mixin that adds the `walk` and `sleep` properties.

```
const animalFunctionality = {
  walk: () => console.log("Walking!"),
  sleep: () => console.log("Sleeping!")
};
```

We can add these properties to the `dogFunctionality` prototype, using `Object.assign`. In this case, the target object is `dogFunctionality`.

```
const animalFunctionality = {
  walk: () => console.log("Walking!"),
  sleep: () => console.log("Sleeping!")
};

const dogFunctionality = {
  bark: () => console.log("Woof!"),
  wagTail: () => console.log("Wagging my tail!"),
  play: () => console.log("Playing!"),
  walk() {
    super.walk();
  },
  sleep() {
    super.sleep();
  }
};

Object.assign(dogFunctionality, animalFunctionality);
Object.assign(Dog.prototype, dogFunctionality);
```

Perfect! Any new instance of `Dog` can now access the walk and sleep methods as well.

```
class Dog {
  constructor(name) {
    this.name = name;
  }
}

const animalFunctionality = {
  walk: () => console.log("Walking!"),
  sleep: () => console.log("Sleeping!")
};

const dogFunctionality = {
  __proto__: animalFunctionality,
  bark: () => console.log("Woof!"),
  wagTail: () => console.log("Wagging my tail!"),
  play: () => console.log("Playing!"),
  walk() {
    super.walk();
  },
  sleep() {
    super.sleep();
  }
};

Object.assign(Dog.prototype, dogFunctionality);

const pet1 = new Dog("Daisy");

console.log(pet1.name);
pet1.bark();
pet1.play();
pet1.walk();
pet1.sleep();
```

An example of a mixin in the real world is visible on the Window interface in a browser environment. The `Window` object implements many of its properties from the `WindowOrWorkerGlobalScope` and `WindowEventHandlers` mixins, which allow us to have access to properties such as `setTimeout` and `setInterval`, `indexedDB`, and `isSecureContext`.

Since it's a mixin, thus is only used to add functionality to objects, you won't be able to create objects of type `WindowOrWorkerGlobalScope`.

```
window.indexedDB.open("toDoList");

window.addEventListener("beforeunload", event => {
  event.preventDefault();
  event.returnValue = "";
});

window.onbeforeunload = () => console.log("Unloading!");

console.log("From WindowEventHandlers mixin: onbeforeunload",
  window.onbeforeunload
);

console.log("From WindowOrWorkerGlobalScope mixin: isSecureContext",
  window.isSecureContext
);

console.log("WindowEventHandlers itself is undefined",
  window.WindowEventHandlers
);

console.log("WindowOrWorkerGlobalScope itself is undefined",
  window.WindowOrWorkerGlobalScope
);
```

React (pre ES6)

Mixins were often used to add functionality to React components before the introduction of ES6 classes. The React team discourages the use of mixins as it easily adds unnecessary complexity to a component, making it hard to maintain and reuse. The React team encouraged the use of higher order components instead, which can now often be replaced by Hooks.

Mixins allow us to easily add functionality to objects without inheritance by injecting functionality into an object's prototype. Modifying an object's prototype is seen as bad practice, as it can lead to prototype pollution and a level of uncertainty regarding the origin of our functions.

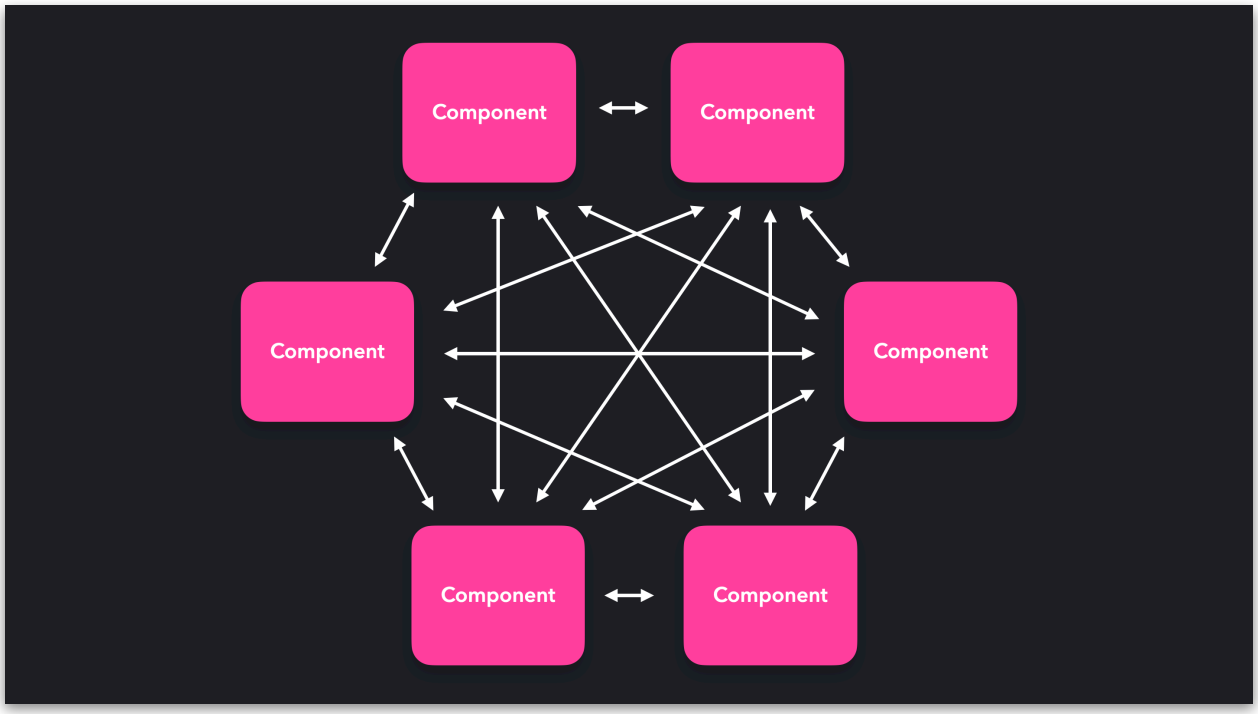
Mediator/ Middleware Pattern

Use a central mediator object to handle communication between components

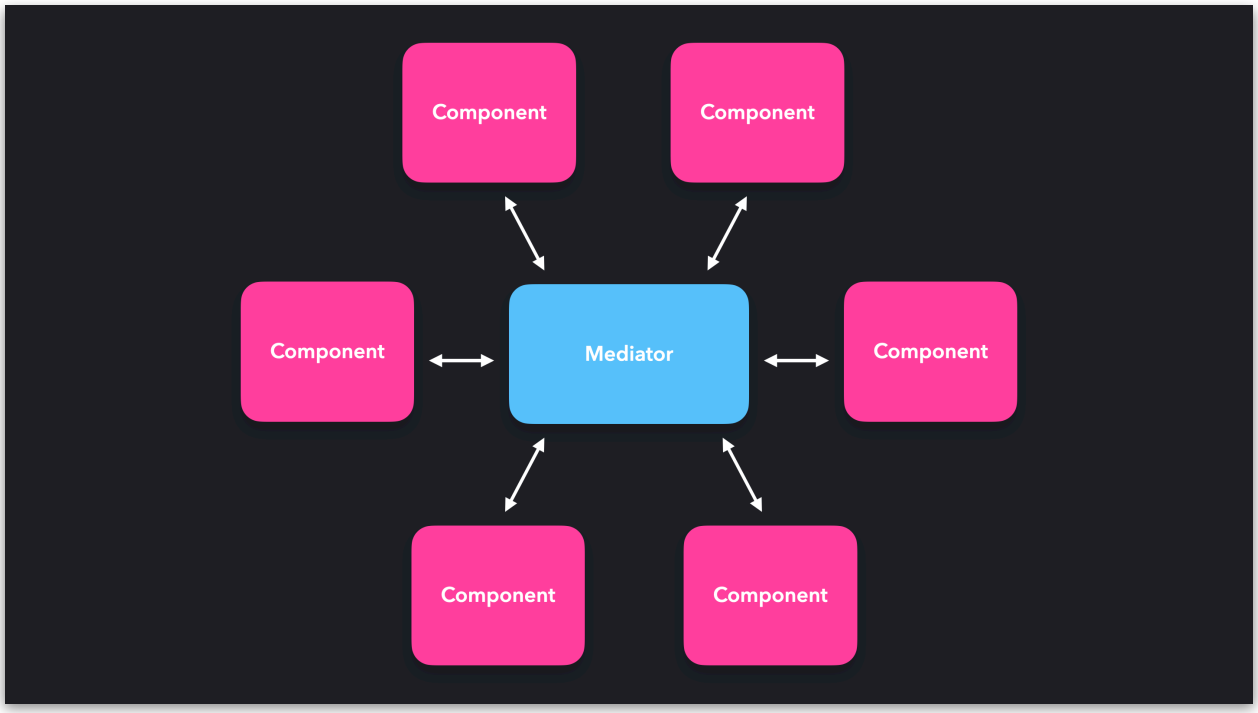
The mediator pattern makes it possible for components to interact with each other through a central point: the mediator. Instead of directly talking to each other, the mediator receives the requests, and sends them forward! In JavaScript, the mediator is often nothing more than an object literal or a function.

You can compare this pattern to the relationship between an air traffic controller and a pilot. Instead of having the pilots talk to each other directly, which would probably end up being quite chaotic, the pilots talk the air traffic controller. The air traffic controller makes sure that all planes receive the information they need in order to fly safely, without hitting the other airplanes.

Although we're hopefully not controlling airplanes in JavaScript, we often have to deal with multidirectional data between objects. The communication between the components can get rather confusing if there is a large number of components.



Instead of letting every objects talk directly to the other objects, resulting in a many-to-many relationship, the object's requests get handled by the mediator. The mediator processes this request, and sends it forward to where it needs to be.



A good use case for the mediator pattern is a chatroom! The users within the chatroom won't talk to each other directly. Instead, the chatroom serves as the mediator between the users.

```
class ChatRoom {
  logMessage(user, message) {
    const time = new Date();
    const sender = user.getName();

    console.log(`${time} [${sender}]: ${message}`);
  }
}

class User {
  constructor(name, chatroom) {
    this.name = name;
    this.chatroom = chatroom;
  }

  getName() {
    return this.name;
  }

  send(message) {
    this.chatroom.logMessage(this, message);
  }
}
```

We can create new users that are connected to the chat room. Each user instance has a send method which we can use in order to send messages.

```
class ChatRoom {
  logMessage(user, message) {
    const sender = user.getName();
    console.log(`${new Date().toLocaleString()} [${sender}]: ${message}`);
  }
}

class User {
  constructor(name, chatroom) {
    this.name = name;
    this.chatroom = chatroom;
  }

  getName() {
    return this.name;
  }

  send(message) {
    this.chatroom.logMessage(this, message);
  }
}

const chatroom = new ChatRoom();

const user1 = new User("John Doe", chatroom);
const user2 = new User("Jane Doe", chatroom);

user1.send("Hi there!");
user2.send("Hey!");
```

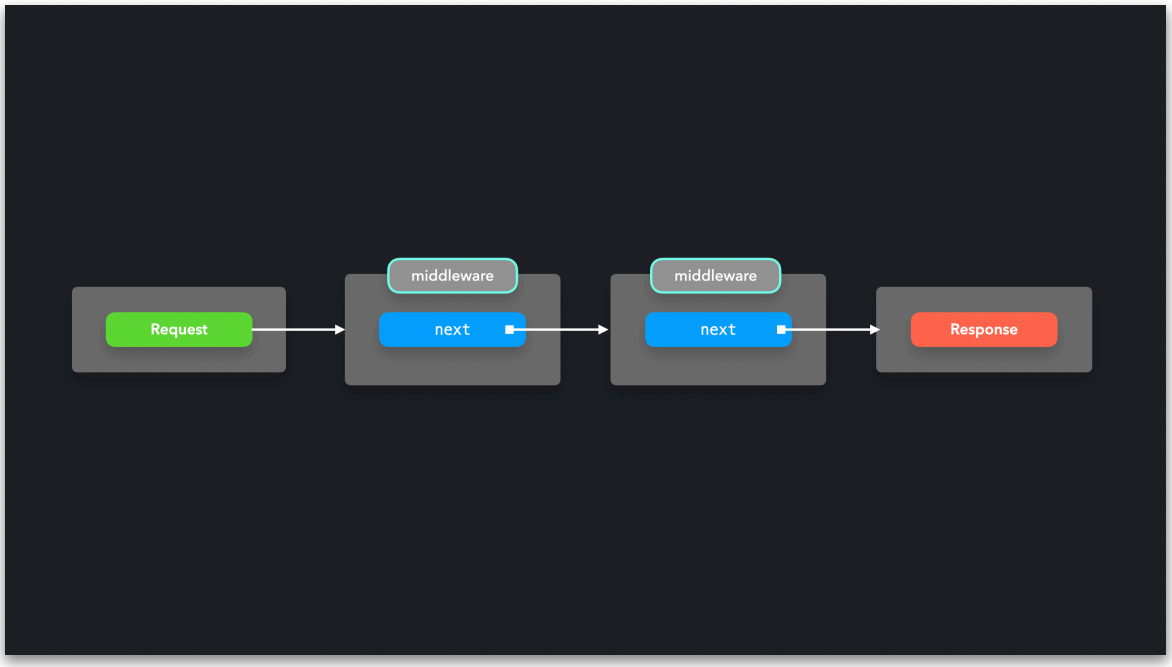
Case Study

Express.js is a popular web application server framework. We can add callbacks to certain routes that the user can access.

Say we want add a header to the request if the user hits the root `/`. We can add this header in a middleware callback.

```
const app = require("express")();

app.use("/", (req, res, next) => {
  req.headers["test-header"] = 1234;
  next();
});
```

The `next` method calls the next callback in the request-response cycle. We'd effectively be creating a chain of middleware functions that sit between the request and the response, or vice versa.



Let's add another middleware function that checks whether the test-header was added correctly. The change added by the previous middleware function will be visible throughout the chain.

```
const app = require("express")();

app.use(
  "/",
  (req, res, next) => {
    req.headers["test-header"] = 1234;
    next();
  },
  (req, res, next) => {
    console.log(`Request has test header: ${!!req.headers["test-header"]}`);
    next();
  }
);
```

Perfect! We can track and modify the request object all the way to the response through one or multiple middleware functions.

```
const app = require("express")();
const html = require("./data");

app.use(
  "/",
  (req, res, next) => {
    req.headers["test-header"] = 1234;
    next();
  },
  (req, res, next) => {
    console.log(`Request has test header: ${!!req.headers["test-header"]}`);
    next();
  }
);

app.get("/", (req, res) => {
  res.set("Content-Type", "text/html");
  res.send(Buffer.from(html));
});

app.listen(8080, function() {
  console.log("Server is running on 8080");
});
```

Every time the user hits a root endpoint '/', the two middleware callbacks will be invoked.

The middleware pattern makes it easy for us to simplify many-to-many relationships between objects, by letting all communication flow through one central point.

Render Props Pattern

Pass JSX elements to components through props

In the section on Higher Order Components, we saw that being able to reuse component logic can be very convenient if multiple components need access to the same data, or contain the same logic.

Another way of making components very reusable, is by using the render prop pattern. A render prop is a prop on a component, which value is a function that returns a JSX element. The component itself does not render anything besides the render prop. Instead, the component simply calls the render prop, instead of implementing its own rendering logic.

Imagine that we have a `Title` component. In this case, the `Title` component shouldn't do anything besides rendering the value that we pass. We can use a render prop for this! Let's pass the value that we want the `Title` component to render to the render prop.

```
<Title render={() => <h1>I am a render prop!</h1>} />
```

Within the `Title` component, we can render this data by returning the invoked render prop!

```
const Title = props => props.render();
```

To the `Title` element, we have to pass a prop called render, which is a function that returns a React element.

```
import "./styles.css";

const Title = (props) => props.render();

render(
  <div className="App">
    <Title
      render={() => (
        <h1>I am a render prop!</h1>
      )}
    />
  </div>,
  document.getElementById("root")
);
```

Perfect, works smoothly! The cool thing about render props, is that the component that receives the prop is very reusable. We can use it multiple times, passing different values to the render prop each time.

```
import React from "react";
import { render } from "react-dom";
import "./styles.css";

const Title = (props) => props.render();

render(
  <div className="App">
    <Title render={() => <h1>✨ First render prop! ✨</h1>} />
    <Title render={() => <h2>🔥 Second render prop! 🔥</h2>} />
    <Title render={() => <h3>🚀 Third render prop! 🚀</h3>} />
  </div>,
  document.getElementById("root")
);
```

Although they're called render props, a render prop doesn't have to be called render. Any prop that renders JSX is considered a render prop! Let's rename the render props that were used in the previous example, and give them specific names instead!

```
import React from "react";
import { render } from "react-dom";
import "./styles.css";

const Title = (props) => (
  <>
    {props.renderFirstComponent()}
    {props.renderSecondComponent()}
    {props.renderThirdComponent()}
  </>
);

render(
  <div className="App">
    <Title
      renderFirstComponent={() => <h1>✨ First render prop! ✨</h1>}
      renderSecondComponent={() => <h2>🔥 Second render prop! 🔥</h2>}
      renderThirdComponent={() => <h3>🚀 Third render prop! 🚀</h3>}
    />
  </div>,
  document.getElementById("root")
);
```

Great! We've just seen that we can use render props in order to make a component reusable, as we can pass different data to the render prop each time. But, why would you want to use this?

A component that takes a render prop usually does a lot more than simply invoking the render prop. Instead, we usually want to pass data from the component that takes the render prop, to the element that we pass as a render prop!

```
function Component(props) {
  const data = { ... }

  return props.render(data)
}
```

The render prop can now receive this value that we passed as its argument.

```
<Component render={data => <ChildComponent data={data} />}
```

Let's look at an example! We have a simple app, where a user can type a temperature in Celsius. The app shows the value of this temperature in Fahrenheit and Kelvin.

```jsx
import React, { useState } from "react";
import "./styles.css";

function Input() {
  const [value, setValue] = useState("");

  return (
    <input
      type="text"
      value={value}
      onChange={e => setValue(e.target.value)}
      placeholder="Temp in °C"
    />
  );
}

export default function App() {
  return (
    <div className="App">
      <h1>☃️ Temperature Converter 🌞</h1>
      <Input />
      <Kelvin />
      <Fahrenheit />
    </div>
  );
}

function Kelvin({ value = 0 }) {
  return <div className="temp">{value + 273.15}K</div>;
}

function Fahrenheit({ value = 0 }) {
  return <div className="temp">{(value * 9) / 5 + 32}°F</div>;
}
```

Hmm.. Currently there's a problem. The stateful Input component contains the value of the user's input, meaning that the Fahrenheit and Kelvin component don't have access to the user's input!

Lifting state

One way to make the users input available to both the `Fahrenheit` and `Kelvin` component in the above example, we'd have to lift the state

In this case, we have a stateful `Input` component. However, the sibling components `Fahrenheit` and `Kelvin` also need access to this data. Instead of having a stateful Input component, we can lift the state up to the first common ancestor component that has a connection to `Input`, `Fahrenheit` and `Kelvin`: the App component in this case!

```
function Input({ value, handleChange }) {
  return <input value={value} onChange={e => handleChange(e.target.value)} />;
}

export default function App() {
  const [value, setValue] = useState("");

  return (
    <div className="App">
      <h1>🤖 Temperature Converter 🌞</h1>
      <Input value={value} handleChange={setValue} />
      <Kelvin value={value} />
      <Fahrenheit value={value} />
    </div>
  );
}
```

Although this is a valid solution, it can be tricky to lift state in larger applications with components that handle many children. Each state change could cause a re-render of all the children, even the ones that don't handle the data, which could negatively affect the performance of your app.

Instead, we can use render props! Let's change the Input component in a way that it can receive render props

```
function Input(props) {
  const [value, setValue] = useState("");

  return (
    <>
      <input
        type="text"
        value={value}
        onChange={e => setValue(e.target.value)}
        placeholder="Temp in °C"
      />
      {props.render(value)}
    </>
  );
}

export default function App() {
  return (
    <div className="App">
      <h1>🚜 Temperature Converter 🌞</h1>
      <Input
        render={value => (
          <>
            <Kelvin value={value} />
            <Fahrenheit value={value} />
          </>
        )}
      />
    </div>
  );
}
```

Perfect, the `Kelvin` and `Fahrenheit` components now have access to the value of the user's input!

Besides regular JSX components, we can pass functions as children to React components. This function is available to us through the children prop, which is technically also a render prop.

Let's change the `Input` component. Instead of explicitly passing the render prop, we'll just pass a function as a child for the `Input` component.

```
export default function App() {
  return (
    <div className="App">
      <h1>☃️ Temperature Converter 🌞</h1>
      <Input>
        {value => (
          <>
            <Kelvin value={value} />
            <Fahrenheit value={value} />
          </>
        )}
      </Input>
    </div>
  );
}
```

.

We have access to this function, through the `props.children` prop that's available on the Input component. Instead of calling `props.render` with the value of the user input, we'll call props.children with the value of the user input.

```
function Input(props) {
  const [value, setValue] = useState("");

  return (
    <>
      <input
        type="text"
        value={value}
        onChange={e => setValue(e.target.value)}
        placeholder="Temp in °C"
      />
      {props.children(value)}
    </>
  );
}
```

Hooks

In some cases, we can replace render props with Hooks. A good example of this is Apollo Client.

One way to use Apollo Client is through the `Mutation` and `Query` components. Let's look at the same Input example that was covered in the Higher Order Components section. Instead of using a the `graphql()` higher order component, we'll now use the `Mutation` component that receives a render prop.

```
import React from "react";
import "./styles.css";

import { Mutation } from "react-apollo";
import { ADD_MESSAGE } from "./resolvers";

export default class Input extends React.Component {
  constructor() {
    super();
    this.state = { message: "" };
  }

  handleChange = (e) => {
    this.setState({ message: e.target.value });
  };

  render() {
    return (
      <Mutation
        mutation={ADD_MESSAGE}
        variables={{ message: this.state.message }}
        onCompleted={() =>
          console.log(`Added with render prop: ${this.state.message} `)
        }
      >
        {(addMessage) => (
          <div className="input-row">
            <input
              onChange={this.handleChange}
              type="text"
              placeholder="Type something..."
            />
            <button onClick={addMessage}>Add</button>
          </div>
        )}
      </Mutation>
    );
  }
}
```

In order to pass data down from the `Mutation` component to the elements that need the data, we pass a function as a child. The function receives the value of the data through its arguments.

```
<Mutation mutation={...} variables={...}>
  {addMessage => <div className="input-row">...</div>}
</Mutation>
```

Although we can still use the render prop pattern and is often preferred compared to the higher order component pattern, it has its downsides.

One of the downsides is deep component nesting. We can nest multiple `Mutation` or `Query` components, if a component needs access to multiple mutations or queries.

```
<Mutation mutation={...} variables={...}>
  {addMessage => <div cl<Mutation mutation={FIRST_MUTATION}>
  {firstMutation => (
    <Mutation mutation={SECOND_MUTATION}>
      {secondMutation => (
        <Mutation mutation={THIRD_MUTATION}>
          {thirdMutation => (
            <Element
              firstMutation={firstMutation}
              secondMutation={secondMutation}
              thirdMutation={thirdMutation}
            />
          )}
        </Mutation>
      )}
    </Mutation>
  )}
</Mutation>assName="input-row">...</div>}
</Mutation>
```

After the release of Hooks, Apollo added Hooks support to the Apollo Client library. Instead of using the `Mutation` and `Query` render props, developers can now directly access the data through the hooks that the library provides.

Let's look at an example that uses the exact same data as we previously saw in the example with the `Query` render prop. This time, we'll provide the data to the component by using the `useQuery` hook that Apollo Client provided for us.

```
import React, { useState } from "react";
import "./styles.css";

import { useMutation } from "@apollo/react-hooks";
import { ADD_MESSAGE } from "./resolvers";

export default function Input() {
  const [message, setMessage] = useState("");
  const [addMessage] = useMutation(ADD_MESSAGE, {
    variables: { message }
  });

  return (
    <div className="input-row">
      <input
        onChange={(e) => setMessage(e.target.value)}
        type="text"
        placeholder="Type something..."
      />
      <button onClick={addMessage}>Add</button>
    </div>
  );
}
```

By using the `useQuery` hook, we reduced the amount of code that was needed in order to provide the data to the component.

Pros

Sharing logic and data among several components is easy with the render props pattern. Components can be made very reusable, by using a render or children prop. Although the Higher Order Component pattern mainly solves the same issues, namely reusability and sharing data, the render props pattern solves some of the issues we could encounter by using the HOC pattern.

The issue of naming collisions that we can run into by using the HOC pattern no longer applies by using the render props pattern, since we don't automatically merge props. We explicitly pass the props down to the child components, with the value provided by the parent component.

Since we explicitly pass props, we solve the HOC's implicit props issue. The props that should get passed down to the element, are all visible in the render prop's arguments list. This way, we know exactly where certain props come from.

We can separate our app's logic from rendering components through render props. The stateful component that receives a render prop can pass the data onto stateless components, which merely render the data.

Cons

The issues that we tried to solve with render props, have largely been replaced by React Hooks. As Hooks changed the way we can add reusability and data sharing to components, they can replace the render props pattern in many cases.

Since we can't add lifecycle methods to a render prop, we can only use it on components that don't need to alter the data they receive.

Hooks Pattern

Use functions to reuse stateful logic among multiple components throughout the app

React 16.8 introduced a new feature called Hooks. Hooks make it possible to use React state and lifecycle methods, without having to use a ES2015 class component.

Although Hooks are not necessarily a design pattern, Hooks play a very important role in your application design. Many traditional design patterns can be replaced by Hooks.

Class components

Before Hooks were introduced in React, we had to use class components in order to add state and lifecycle methods to components. A typical class component in React can look something like:

```
class MyComponent extends React.Component {
  /* Adding state and binding custom methods */
  constructor() {
    super()
    this.state = { ... }

    this.customMethodOne = this.customMethodOne.bind(this)
    this.customMethodTwo = this.customMethodTwo.bind(this)
  }

  /* Lifecycle Methods */
  componentDidMount() { ...}
  componentWillUnmount() { ... }

  /* Custom methods */
  customMethodOne() { ... }
  customMethodTwo() { ... }

  render() { return { ... }}
}
```

A class component can contain a state in its constructor, lifecycle methods such as `componentDidMount` and `componentWillUnmount` to perform side effects based on a component's lifecycle, and custom methods to add extra logic to a class.

Although we can still use class components after the introduction of React Hooks, using class components can have some downsides! Let's look at some of the most common issues when using class components.

Understanding ES2015 classes

Since class components were the only component that could handle state and lifecycle methods before React Hooks, we often ended up having to refactor functional components into a class components, in order to add the extra functionality.

In this example, we have a simple div that functions as a button.

```
function Button() {
  return <div className="btn">disabled</div>;
}
```

Instead of always displaying disabled, we want to change it to enabled when the user clicks on the button, and add some extra CSS styling to the button when that happens.

In order to do that, we need to add state to the component in order to know whether the status is enabled or disabled. This means that we'd have to refactor the functional component entirely, and make it a class component that keeps track of the button's state.

```
export default class Button extends React.Component {
  constructor() {
    super();
    this.state = { enabled: false };
  }

  render() {
    const { enabled } = this.state;
    const btnText = enabled ? "enabled" : "disabled";

    return (
      <div
        className={`btn enabled-${enabled}`}
        onClick={() => this.setState({ enabled: !enabled })}
      >
        {btnText}
      </div>
    );
  }
}
```

Finally, our button works the way we want it to!

```
import React from "react";
import "./styles.css";

export default class Button extends React.Component {
  constructor() {
    super();
    this.state = { enabled: false };
  }

  render() {
    const { enabled } = this.state;
    const btnText = enabled ? "enabled" : "disabled";

    return (
      <div
        className={`btn enabled-${enabled}`}
        onClick={() => this.setState({ enabled: !enabled })}
      >
        {btnText}
      </div>
    );
  }
}
```

In this example, the component is very small and refactoring wasn't a such a great deal. However, your real-life components probably contain of many more lines of code, which makes refactoring the component a lot more difficult.

Besides having to make sure you don't accidentally change any behavior while refactoring the component, you also need to understand how ES2015

classes work. Why do we have to bind the custom methods? What does the constructor do? Where does the this keyword come from? It can be difficult to know how to refactor a component properly without accidentally changing the data flow.

Restructuring

The common way to share code among several components, is by using the Higher Order Component or Render Props pattern. Although both patterns are valid and a good practice, adding those patterns at a later point in time requires you to restructure your application.

Besides having to restructure your app, which is trickier the bigger your components are, having many wrapping components in order to share code among deeper nested components can lead to something that's best referred to as a wrapper hell. It's not uncommon to open your dev tools and seeing a structure similar to:

```
<WrapperOne>
  <WrapperTwo>
    <WrapperThree>
      <WrapperFour>
        <WrapperFive>
          <Component>
            <h1>Finally in the component!</h1>
          </Component>
        </WrapperFive>
      </WrapperFour>
    </WrapperThree>
  </WrapperTwo>
</WrapperOne>
```

The wrapper hell can make it difficult to understand how data is flowing through your application, which can make it harder to figure out why unexpected behavior is happening.

Complexity

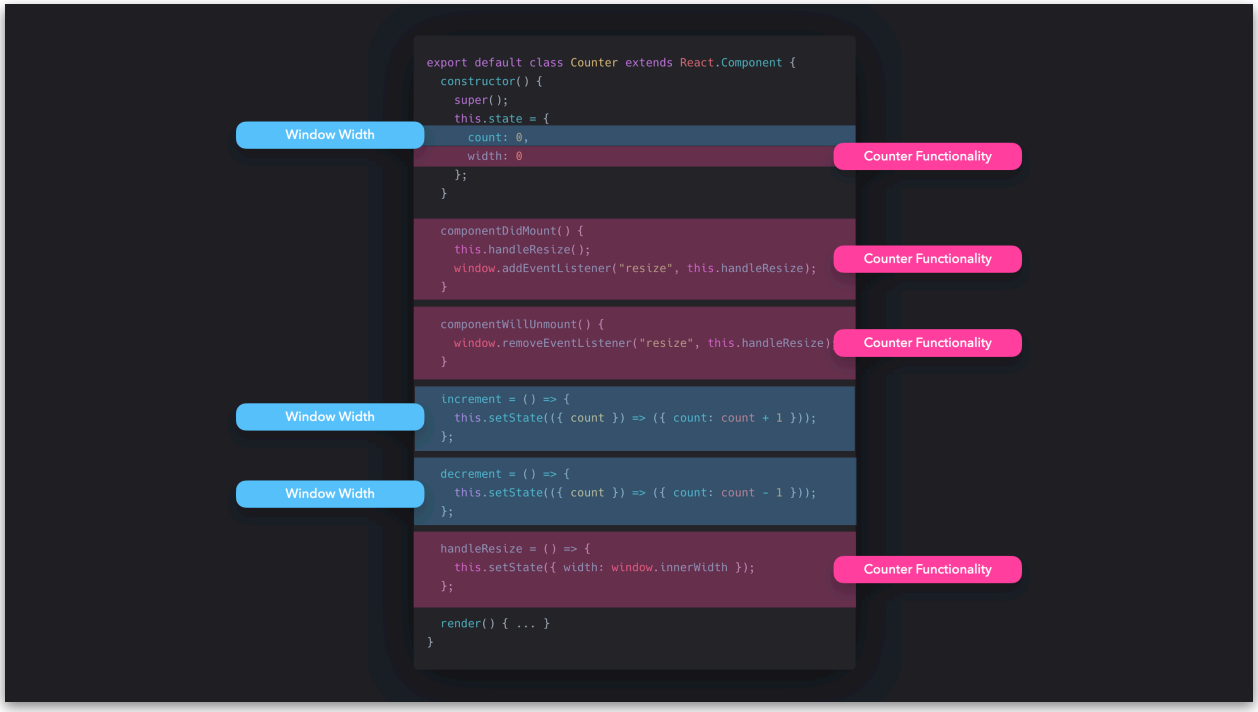
As we add more logic to class components, the size of the component increases fast. Logic within that component can get tangled and unstructured, which can make it difficult for developers to understand where certain logic is used in the class component. This can make debugging and optimizing performance more difficult.

Lifecycle methods also require quite a lot of duplication in the code. Let's take a look at an example, which uses a Counter component and a Width component.

```
import React from "react";
import "./styles.css";

import { Count } from "./Count";
import { Width } from "./Width";

export default class Counter extends React.Component {
  constructor() {
    super();
    this.state = {
      count: 0,
      width: 0
    };
  }
```

```
  componentDidMount() {
    this.handleResize();
    window.addEventListener("resize", this.handleResize);
  }

  componentWillUnmount() {
    window.removeEventListener("resize", this.handleResize);
  }

  increment = () => {
    this.setState(({ count }) => ({ count: count + 1 }));
  };

  decrement = () => {
    this.setState(({ count }) => ({ count: count - 1 }));
  };

  handleResize = () => {
    this.setState({ width: window.innerWidth });
  };

  render() {
    return (
      <div className="App">
        <Count
          count={this.state.count}
          increment={this.increment}
          decrement={this.decrement}
        />
        <div id="divider" />
        <Width width={this.state.width} />
      </div>
    );
  }
}
```

The way the App component is structured can be visualized as the following:



Although this is a small component, the logic within the component is already quite tangled. Whereas certain parts are specific for the counter logic, other parts are specific for the width logic. As your component grows, it can get increasingly difficult to structure logic within your component, find related logic within the component.

Besides tangled logic, we're also duplicating some logic within the lifecycle methods. In both `componentDidMount` and `componentWillUnmount`, we're customizing the behavior of the app based on the window's resize event.

Hooks

It's quite clear that class components aren't always a great feature in React. In order to solve the common issues that React developers can run into when using class components, React introduced React Hooks. React Hooks are functions that you can use to manage a components state and lifecycle methods. React Hooks make it possible to:

- add state to a functional component
- manage a component's lifecycle without having to use lifecycle methods such as `componentDidMount` and `componentWillUnmount`
- reuse the same stateful logic among multiple components throughout the app

First, let's take a look at how we can add state to a functional component, using React Hooks.

State Hook

React provides a hook that manages state within a functional component, called `useState.`

Let’s see how a class component can be restructured into a functional component, using the useState hook. We have a class component called Input, which simply renders an input field. The value of input in the state updates, whenever the user types anything in the input field.

```
class Input extends React.Component {
  constructor() {
    super();
    this.state = { input: "" };

    this.handleInput = this.handleInput.bind(this);
  }

  handleInput(e) {
    this.setState({ input: e.target.value });
  }

  render() {
    <input onChange={handleInput} value={this.state.input} />;
  }
}
```

In order to use the `useState` hook, we need to access the `useState` method that React provides for us. The `useState` method expects an argument: this is the initial value of the state, an empty string in this case.

We can destructure two values from the useState method:

1. The current value of the state.

2. The method with which we can update the state.

```
const [value, setValue] = React.useState(initialValue);
```

The first value can be compared to a class component's `this.state.[value]`. The second value can be compared to a class component's `this.setState` method.

Since we're dealing with the value of an input, let's call the current value of the state input, and the method in order to update the state `setInput`. The initial value should be an empty string.

```
const [input, setInput] = React.useState("");
```

We can now refactor the `Input` class component into a stateful functional component.

```
function Input() {
  const [input, setInput] = React.useState("");

  return <input onChange={(e) => setInput(e.target.value)} value={input} />;
}
```

The value of the input field is equal to the current value of the input state, just like in the class component example. When the user types in the input field, the value of the input state updates accordingly, using the setInput method.

```jsx
import React, { useState } from "react";

export default function Input() {
  const [input, setInput] = useState("");

  return (
    <input
      onChange={e => setInput(e.target.value)}
      value={input}
      placeholder="Type something..."
    />
  );
}
```

Effect Hook

We've seen we can use the `useState` component to handle state within a functional component, but another benefit of class components was the possibility to add lifecycle methods to a component.

With the `useEffect` hook, we can "hook into" a components lifecycle. The `useEffect` hook effectively combines the `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` lifecycle methods.

Let's use the input example we used in the State Hook section. Whenever the user is typing anything in the input field, we also want to log that value to the console.

```
componentDidMount() { ... }
useEffect(() => { ... }, [])

componentWillUnmount() { ... }
useEffect(() => { return () => { ... } }, [])

componentDidUpdate() { ... }
useEffect(() => { ... })
```

We need to use a `useEffect` hook that "listens" to the input value. We can do so, by adding input to the dependency array of the `useEffect` hook. The dependency array is the second argument that the `useEffect` hook receives.

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

export default function Input() {
  const [input, setInput] = useState("");

  useEffect(() => {
    console.log(`The user typed ${input}`);
  }, [input]);

  return (
    <input
      onChange={e => setInput(e.target.value)}
      value={input}
      placeholder="Type something..."
    />
  );
}
```

The value of the input now gets logged to the console whenever the user types a value.

Custom Hooks

Besides the built-in hooks that React provides (`useState`, `useEffect`, `useReducer`, `useRef`, `useContext`, `useMemo`, `useContext, useImperativeHandle`, `useLayoutEffect`, `useDebugValue`, `useCallback`), we can easily create our own custom hooks.

You may have noticed that all hooks start with use. It's important to start your hooks with use in order for React to check if it violates the rules of Hooks.

Let's say we want to keep track of certain keys the user may press when writing the input. Our custom hook should be able to receive the key we want to target as its argument.

```
function useKeyPress(targetKey) {}
```

We want to add a keydown and keyup event listener to the key that the user passed as an argument. If the user pressed that key, meaning the keydown event gets triggered, the state within the hook should toggle to true. Else, when the user stops pressing that button, the keyup event gets triggered and the state toggles to false.

```
function useKeyPress(targetKey) {
  const [keyPressed, setKeyPressed] = React.useState(false);

  function handleDown({ key }) {
    if (key === targetKey) {
      setKeyPressed(true);
    }
  }

  function handleUp({ key }) {
    if (key === targetKey) {
      setKeyPressed(false);
    }
  }

  React.useEffect(() => {
    window.addEventListener("keydown", handleDown);
    window.addEventListener("keyup", handleUp);

    return () => {
      window.removeEventListener("keydown", handleDown);
      window.removeEventListener("keyup", handleUp);
    };
  }, []);

  return keyPressed;
}
```

```
import React from "react";
import useKeyPress from "./useKeyPress";

export default function Input() {
  const [input, setInput] = React.useState("");
  const pressQ = useKeyPress("q");
  const pressW = useKeyPress("w");
  const pressL = useKeyPress("l");

  React.useEffect(() => {
    console.log(`The user pressed Q!`);
  }, [pressQ]);

  React.useEffect(() => {
    console.log(`The user pressed W!`);
  }, [pressW]);

  React.useEffect(() => {
    console.log(`The user pressed L!`);
  }, [pressL]);

  return (
    <input
      onChange={e => setInput(e.target.value)}
      value={input}
      placeholder="Type something..."
    />
  );
}
```

Perfect! We can use this custom hook in our input application. Let's log to the console whenever the user presses the q, l or w key.

Instead of keeping the key press logic local to the Input component, we can now reuse the `useKeyPress` hook throughout multiple components, without having to rewrite the same logic over and over.

Another great advantage of Hooks, is that the community can build and share hooks. We just wrote the `useKeyPress` hook ourselves, but that actually wasn't necessary at all! The hook was already built by someone else and ready to use in our application if we just installed it!

Let's rewrite the counter and width example shown in the previous section. Instead of using a class component, we'll rewrite the app using React Hooks.

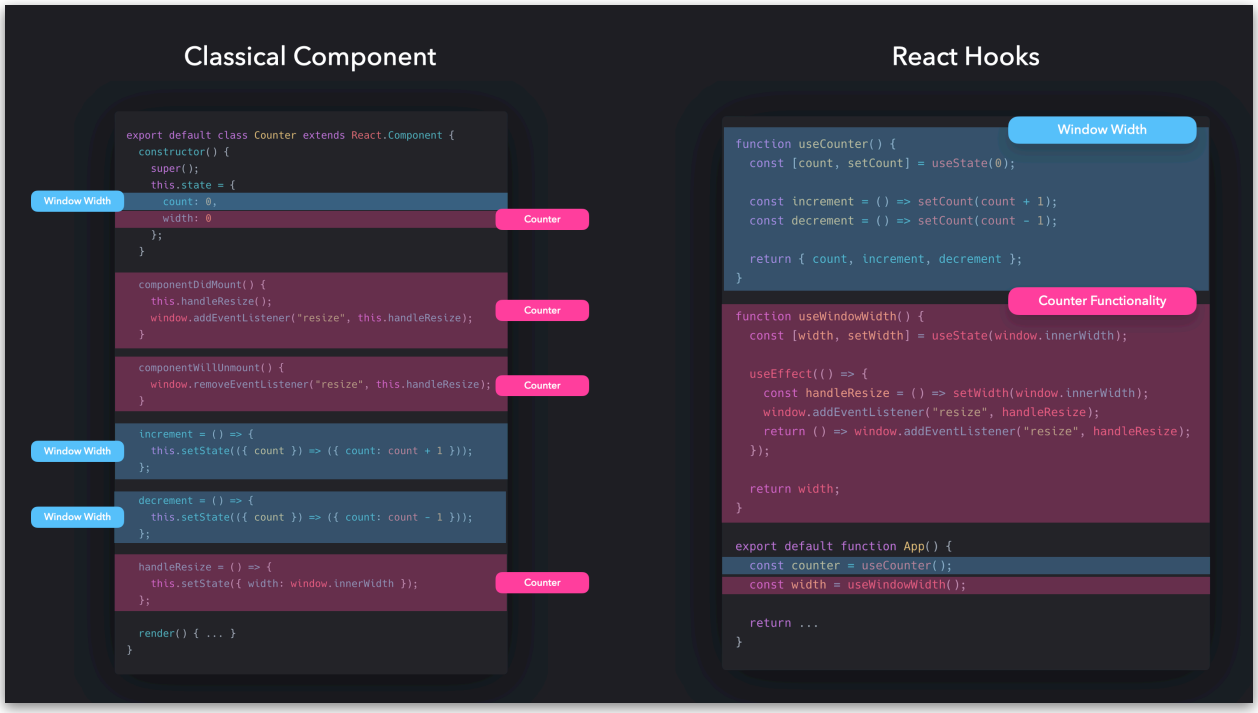
```jsx
import React, { useState, useEffect } from "react";
import "./styles.css";

import { Count } from "./Count";
import { Width } from "./Width";

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

  const increment = () => setCount(count + 1);
  const decrement = () => setCount(count - 1);

  return { count, increment, decrement };
}

function useWindowWidth() {
  const [width, setWidth] = useState(window.innerWidth);

  useEffect(() => {
    const handleResize = () => setWidth(window.innerWidth);
    window.addEventListener("resize", handleResize);
    return () => window.addEventListener("resize", handleResize);
  });

  return width;
}

export default function App() {
  const counter = useCounter();
  const width = useWindowWidth();

  return (
    <div className="App">
      <Count
        count={counter.count}
        increment={counter.increment}
        decrement={counter.decrement}
      />
      <div id="divider" />
      <Width width={width} />
    </div>
  );
}
```

We broke the logic of the App function into several pieces:

- `useCounter`: A custom hook that returns the current value of count, an increment method, and a decrement method.
- `useWindowWidth`: A custom hook that returns the window's current width.
- App: A functional, stateful component that returns the `Counter` and `Width` component.

By using React Hooks instead of a class component, we were able to break the logic down into smaller, reusable pieces that separated the logic.

Let's visualize the changes we just made, compared to the old App class component.



Using React Hooks just made it much clearer to separate the logic of our component into several smaller pieces. Reusing the same stateful logic just became much easier, and we no longer have to rewrite functional components into class components if we want to make the component stateful. A good knowledge of ES2015 classes is no longer required, and having reusable stateful logic increases the testability, flexibility and readability of components.

Adding Hooks

Like other components, there are special functions that are used when you want to add Hooks to the code you have written. Here's a brief overview of some common Hook functions:

`useState`

The `useState` Hook enables developers to update and manipulate state inside function components without needing to convert it to a class component. One advantage of this Hook is that it is simple and does not require as much complexity as other React Hooks.

`useEffect`

The `useEffect` Hook is used to run code during major lifecycle events in a function component. The main body of a function component does not allow mutations, subscriptions, timers, logging, and other side effects. If they are allowed, it could lead to confusing bugs and inconsistencies within the UI. The useEffect hook prevents all of these "side effects" and allows the UI to run

smoothly. It is a combination
of `componentDidMount` , `componentDidUpdate` ,
and `componentWillUnmount`, all in one place.

`useContext`

The `useContext` Hook accepts a context object, which is the value returned from `React.createContext`, and returns the current context value for that context. The useContext Hook also works with the React Context API in order to share data throughout the app without the need to pass your app props down through various levels.

It should be noted that the argument passed to the `useContext` hook must be the context object itself and any component calling
the `useContext` always re-render whenever the context value changes.

`useReducer`

The `useReducer` Hook gives an alternative to `useState` and is especially preferable to it when you have complex state logic that involves multiple sub-values or when the next state depends on the previous one. It takes on
a reducer function and an initial state input and returns the current state and a dispatch function as output by means of array
destructuring. `useReducer` also optimizes the performance of components that trigger deep updates.

Pros and Cons of using Hooks

Here are some benefits of making use of Hooks:

Fewer lines of code Hooks allows you group code by concern and functionality, and not by lifecycle. This makes the code not only cleaner and concise but also shorter. Below is a comparison of a simple stateless component of a searchable product data table using React, and how it looks in Hooks after using the `useState` keyword.

Stateless Component

```
class TweetSearchResults extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      filterText: '',
      inThisLocation: false
    };

    this.handleFilterTextChange = this.handleFilterTextChange.bind(this);
    this.handleInThisLocationChange = this.handleInThisLocationChange.bind(this);
  }

  handleFilterTextChange(filterText) {
    this.setState({
      filterText: filterText
    });
  }

  handleInThisLocationChange(inThisLocation) {
    this.setState({
      inThisLocation: inThisLocation
    })
  }

  render() {
    return (
      <div>
        <SearchBar
          filterText={this.state.filterText}
          inThisLocation={this.state.inThisLocation}
          onFilterTextChange={this.handleFilterTextChange}
          onInThisLocationChange={this.handleInThisLocationChange}
        />
        <TweetList
          tweets={this.props.tweets}
          filterText={this.state.filterText}
          inThisLocation={this.state.inThisLocation}
        />
      </div>
    );
  }
}
```

Same component with Hooks

```
const TweetSearchResults = ({tweets}) => {
  const [filterText, setFilterText] = useState('');
  const [inThisLocation, setInThisLocation] = useState(false);
  return (
    <div>
      <SearchBar
        filterText={filterText}
        inThisLocation={inThisLocation}
        setFilterText={setFilterText}
        setInThisLocation={setInThisLocation}
      />
      <TweetList
        tweets={tweets}
        filterText={filterText}
        inThisLocation={inThisLocation}
      />
    </div>
  );
}
```

Simplifies complex components

JavaScript classes can be difficult to manage, hard to use with hot reloading and may not minify as well. React Hooks solves these problems and ensures functional programming is made easy. With the implementation of Hooks, We don't need to have class components.

Reusing stateful logic Classes in JavaScript encourage multiple levels of inheritance that quickly increase overall complexity and potential for errors.

However, Hooks allow you to use state, and other React features without writing a class. With React, you can always reuse stateful logic without the need to rewrite the code over and over again. This reduces the chances of errors and allows for composition with plain functions.

Sharing non-visual logic

Until the implementation of Hooks, React had no way of extracting and sharing non-visual logic. This eventually led to more complexities, such as the HOC patterns and Render props, just to solve a common problem. But, the introduction of Hooks has solved this problem because it allows for the extraction of stateful logic to a simple JavaScript function.

There are of course some potential downsides to Hooks worth keeping in mind:

- Have to respect its rules, without a linter plugin, it is difficult to know which rule has been broken.
- Need a considerable time practicing to use properly (Exp: `useEffect`).
- Be aware of the wrong use (Exp: `useCallback`, `useMemo`).

React Hooks vs Classes

When Hooks were introduced to React, it created a new problem: how do we know when to use function components with Hooks and class components? With the help of Hooks, it is possible to get state and partial lifecycle Hooks even in function components. Hooks also allow you to use local state and other React features without writing a class.

Here are some differences between Hooks and Classes to help you decide:

React Hooks	**Classes**
It helps avoid multiple hierarchies and make code clearer	Generally, when you use HOC or *renderProps,* you have to restructure your App with multiple hierarchies when you try to see it in DevTools
It provides uniformity across React components.	Classes confuse both humans and machines due to the need to understand binding and the context in which functions are called.

HOC Pattern

Pass reusable logic down as props to components throughout your application

Within our application, we often want to use the same logic in multiple components. This logic can include applying a certain styling to components, requiring authorization, or adding a global state.

One way of being able to reuse the same logic in multiple components, is by using the higher order component pattern. This pattern allows us to reuse component logic throughout our application.

A Higher Order Component (HOC) is a component that receives another component. The HOC contains certain logic that we want to apply to the component that we pass as a parameter. After applying that logic, the HOC returns the element with the additional logic.

Say that we always wanted to add a certain styling to multiple components in our application. Instead of creating a style object locally each time, we can simply create a HOC that adds the style objects to the component that we pass to it

```
function withStyles(Component) {
  return props => {
    const style = { padding: '0.2rem', margin: '1rem' }
    return <Component style={style} {...props} />
  }
}

const Button = () = <button>Click me!</button>
const Text = () => <p>Hello World!</p>

const StyledButton = withStyles(Button)
const StyedText = withStyles(Text)
```

We just created a `StyledButton` and `StyledText` component, which are the modified versions of the `Button` and `Text` component. They now both contain the style that got added in the `withStyles` HOC!

Let's take a look at the same `DogImages` example that was previously used in the Container/Presentational pattern! The application does nothing more than rendering a list of dog images, fetched from an API.

Let's improve the user experience a little bit. When we're fetching the data, we want to show a *`Loading...`* screen to the user. Instead of adding data to the `DogImages` component directly, we can use a Higher Order Component that adds this logic for us.

Let's create a HOC called `withLoader`. A HOC should receive an component, and return that component. In this case, the withLoader HOC should receive the element which should display *`Loading...`* until the data is fetched.

Let's create the bare minimum version of the withLoader HOC that we want to use!

```
function withLoader(Element) {
  return props => <Element />;
}
```

However, we don't just want to return the element it received. Instead, we want this element to contain logic that tells us whether the data is still loading or not.

To make the `withLoader` HOC very reusable, we won't hardcode the Dog API url in that component. Instead, we can pass the URL as an argument to the withLoader HOC, so this loader can be used on any component that needs a loading indicator while fetching data from a different API endpoint.

```
function withLoader(Element, url) {
  return props => {};
}
```

A HOC returns an element, a functional component `props => {}` in this case, to which we want to add the logic that allows us to display a text with *`Loading…`* as the data is still being fetched. Once the data has been fetched, the component should pass the fetched data as a prop.

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

export default function withLoader(Element, url) {
  return (props) => {
    const [data, setData] = useState(null);

    useEffect(() => {
      async function getData() {
        const res = await fetch(url);
        const data = await res.json();
        setData(data);
      }

      getData();
    }, []);

    if (!data) {
      return <div>Loading...</div>;
    }

    return <Element {...props} data={data} />;
  };
}
```

Perfect! We just created a HOC that can receive any component and url.

In the `useEffect` hook, the `withLoader` HOC fetches the data from the API endpoint that we pass as the value of url. While the data hasn't returned yet, we return the element containing the *`Loading...`* text.

Once the data has been fetched, we set data equal to the data that has been fetched. Since data is no longer null, we can display the element that we passed to the HOC!

So, how can we add this behavior to our application, so it'll actually show the *Loading*... indicator on the `DogImages` list?

In `DogImages.js`, we no longer want to just export the plain `DogImages` component. Instead, we want to export the "wrapped" `withLoader` HOC around the `DogImages` component.

```
export default withLoader(DogImages);
```

The `withLoader` HOC also expects the url to know which endpoint to fetch the data from. In this case, we want to add the Dog API endpoint.

```
export default withLoader(
  DogImages,
  "https://dog.ceo/api/breed/labrador/images/random/6"
);
```

Since the `withLoader` HOC returned the element with an extra data prop, `DogImages` in this case, we can access the data prop in the `DogImages` component.

```
import React from "react";
import withLoader from "./withLoader";

function DogImages(props) {
  return props.data.message.map((dog, index) => (
    <img src={dog} alt="Dog" key={index} />
  ));
}

export default withLoader(
  DogImages,
  "https://dog.ceo/api/breed/labrador/images/random/6"
);
```

Perfect! We now see a `Loading...` screen while the data is being fetched.

The Higher Order Component pattern allows us to provide the same logic to multiple components, while keeping all the logic in one single place.
The `withLoader` HOC doesn't care about the component or url it receives: as long as it's a valid component and a valid API endpoint, it'll simply pass the data from that API endpoint to the component that we pass.

Composing

We can also compose multiple Higher Order Components. Let's say that we also want to add functionality that shows a Hovering! text box when the user hovers over the `DogImages` list.

We need to create a HOC that provides a hovering prop to the element that we pass. Based on that prop, we can conditionally render the text box based on whether the user is hovering over the DogImages list.

```
import React from "react";
import withLoader from "./withLoader";
import withHover from "./withHover";

function DogImages(props) {
  return (
    <div {...props}>
      {props.hovering && <div id="hover">Hovering!</div>}
      <div id="list">
        {props.data.message.map((dog, index) => (
          <img src={dog} alt="Dog" key={index} />
        ))}
      </div>
    </div>
  );
}

export default withHover(
  withLoader(DogImages, "https://dog.ceo/api/breed/labrador/images/random/6")
);
```

We can now wrap the `withHover` HOC around the `withLoader`

The `DogImages` element now contains all props that we passed fr both `withHover` and `withLoader`. We can now conditionally render

the `Hovering!` text box, based on whether the value of the hovering prop is true or false.

A well-known library used for composing HOCs is recompose. Since HOCs can largely be replaced by React Hooks, the recompose library is no longer maintained, thus won't be covered in this article.

Hooks

In some cases, we can replace the HOC pattern with React Hooks.

Let's replace the `withHover` HOC with a `useHover` hook. Instead of having a higher order component, we export a hook that adds a `mouseOver` and `mouseLeave` event listener to the element. We cannot pass the element anymore like we did with the HOC. Instead, we'll return a ref from the hook for that should get the `mouseOver` and `mouseLeave` events.

```
import { useState, useRef, useEffect } from "react";

export default function useHover() {
  const [hovering, setHover] = useState(false);
  const ref = useRef(null);

  const handleMouseOver = () => setHover(true);
  const handleMouseOut = () => setHover(false);

  useEffect(() => {
    const node = ref.current;
    if (node) {
      node.addEventListener("mouseover", handleMouseOver);
      node.addEventListener("mouseout", handleMouseOut);

      return () => {
        node.removeEventListener("mouseover", handleMouseOver);
        node.removeEventListener("mouseout", handleMouseOut);
      };
    }
  }, [ref.current]);

  return [ref, hovering];
}
```

The `useEffect` hook adds an event listener to the component, and sets the value hovering to true or false, depending on whether the user is currently hovering over the element. Both the `ref` and `hovering` values need to be returned from the hook: `ref` to add a ref to the component that should receive the `mouseOver` and `mouseLeave` events, and hovering in order to be able to conditionally render the Hovering! text box.

Instead of wrapping the DogImages component with the withHover HOC, we can use the useHover hook right inside the DogImages component.

```
import React from "react";
import withLoader from "./withLoader";
import useHover from "./useHover";

function DogImages(props) {
  const [hoverRef, hovering] = useHover();

  return (
    <div ref={hoverRef} {...props}>
      {hovering && <div id="hover">Hovering!</div>}
      <div id="list">
        {props.data.message.map((dog, index) => (
          <img src={dog} alt="Dog" key={index} />
        ))}
      </div>
    </div>
  );
}

export default withLoader(
  DogImages,
  "https://dog.ceo/api/breed/labrador/images/random/6"
);
```

Perfect! Instead of wrapping the DogImages component with the withHover component, we can simply use the useHover hook within the component directly.

Generally speaking, React Hooks don't replace the HOC pattern. As the React docs tell us, using Hooks can reduce the depth of the component tree. Using the HOC pattern, it's easy to end up with a deeply nested component tree.

```
<withAuth>
  <withLayout>
    <withLogging>
      <Component />
    </withLogging>
  </withLayout>
</withAuth>
```

By adding a Hook to the component directly, we no longer have to wrap components.

Using Higher Order Components makes it possible to provide the same logic to many components, while keeping that logic all in one single place. Hooks allow us to add custom behavior from within the component, which could potentially increase the risk of introducing bugs compared to the HOC pattern if multiple components rely on this behavior.

Best use-cases for a HOC:

- The same, uncustomized behavior needs to be used by many components throughout the application.
- The component can work standalone, without the added custom logic.

Best use-cases for Hooks:

- The behavior has to be customized for each component that uses it.
- The behavior is not spread throughout the application, only one or a few components use the behavior.
- The behavior adds many properties to the component

Case Study

Some libraries that relied on the HOC pattern added Hooks support after the release. A good example of this is Apollo Client.

One way to use Apollo Client is through the `graphql()` higher order component.

```
import React from "react";
import "./styles.css";

import { graphql } from "react-apollo";
import { ADD_MESSAGE } from "./resolvers";

class Input extends React.Component {
  constructor() {
    super();
    this.state = { message: "" };
  }

  handleChange = (e) => {
    this.setState({ message: e.target.value });
  };

  handleClick = () => {
    this.props.mutate({ variables: { message: this.state.message } });
  };

  render() {
    return (
      <div className="input-row">
        <input
          onChange={this.handleChange}
          type="text"
          placeholder="Type something..."
        />
        <button onClick={this.handleClick}>Add</button>
      </div>
    );
  }
}

export default graphql(ADD_MESSAGE)(Input);
```

```
import React, { useState } from "react";
import "./styles.css";

import { useMutation } from "@apollo/react-hooks";
import { ADD_MESSAGE } from "./resolvers";

export default function Input() {
  const [message, setMessage] = useState("");
  const [addMessage] = useMutation(ADD_MESSAGE, {
    variables: { message }
  });

  return (
    <div className="input-row">
      <input
        onChange={(e) => setMessage(e.target.value)}
        type="text"
        placeholder="Type something..."
      />
      <button onClick={addMessage}>Add</button>
    </div>
  );
}
```

With the `graphql()` HOC, we can make data from the client available to components that are wrapped by the higher order component! Although we can still use the `graphql()` HOC currently, there are some downsides to using it.

When a component needs access to multiple resolvers, we need to compose multiple `graphql()` higher order components in order to do so. Composing multiple HOCs can make it difficult to understand how the data is passed to your components. The order of the HOCs can matter in some cases, which can easily lead to bugs when refactoring the code.

After the release of Hooks, Apollo added Hooks support to the Apollo Client library. Instead of using the `graphql()` higher order component, developers can now directly access the data through the hooks that the library provides.

Pros

Using the Higher Order Component pattern allows us to keep logic that we want to re-use all in one place. This reduces the risk of accidentally spreading bugs throughout the application by duplicating code over and over, potentially introducing new bugs each time. By keeping the logic all in one place, we can keep our code DRY and easily enforce separation of concerns

Cons

The name of the prop that a HOC can pass to an element, can cause a naming collision.

```
function withStyles(Component) {
  return props => {
    const style = { padding: '0.2rem', margin: '1rem' }
    return <Component style={style} {...props} />
  }
}

const Button = () = <button style={{ color: 'red' }}>Click me!</button>
const StyledButton = withStyles(Button)
```

In this case, the `withStyles` HOC adds a prop called style to the element that we pass to it. However, the `Button` component already had a prop called style, which will be overwritten! Make sure that the HOC can handle accidental name collision, by either renaming the prop or merging the props.

```
function withStyles(Component) {
  return props => {
    const style = {
      padding: '0.2rem',
      margin: '1rem',
      ...props.style
    }

    return <Component style={style} {...props} />
  }
}

const Button = () = <button style={{ color: 'red' }}>Click me!</button>
const StyledButton = withStyles(Button)
```

When using multiple composed HOCs that all pass props to the element that's wrapped within them, it can be difficult to figure out which HOC is responsible for which prop. This can hinder debugging and scaling an application easily.

Flyweight Pattern

Reuse existing instances when working with identical objects

The flyweight pattern is a useful way to conserve memory when we're creating a large number of similar objects.

In our application, we want users to be able to add books. All books have a title, an author, and an isbn number! However, a library usually doesn't have just one copy of a book: it usually has multiple copies of the same book.

It wouldn't be very useful to create a new book instance each time if there are multiple copies of the exact same book. Instead, we want to create multiple instances of the `Book` constructor, that represent a single book.

```
class Book {
  constructor(title, author, isbn) {
    this.title = title;
    this.author = author;
    this.isbn = isbn;
  }
}
```

Let's create the functionality to add new books to the list. If a book has the same ISBN number, thus is the exact same book type, we don't want to create

an entirely new Book instance. Instead, we should first check whether this book already exists.

```
const books = new Map();

const createBook = (title, author, isbn) => {
  const existingBook = books.has(isbn);

  if (existingBook) {
    return books.get(isbn);
  }
};
```

If it doesn't contain the book's ISBN number yet, we'll create a new book and add its ISBN number to the isbnNumbers set.

```
const createBook = (title, author, isbn) => {
  const existingBook = books.has(isbn);

  if (existingBook) {
    return books.get(isbn);
  }

  const book = new Book(title, author, isbn);
  books.set(isbn, book);

  return book;
};
```

The `createBook` function helps us create new instances of one type of book. However, a library usually contains multiple copies of the same book! Let's create an `addBook` function, which allows us to add multiple copies of the same book. It should invoke the `createBook` function, which returns either a newly created `Book` instance, or returns the already existing instance.

In order to keep track of the total amount of copies, let's create a `bookList` array that contains the total amount of books in the library.

```
const bookList = [];

const addBook = (title, author, isbn, availability, sales) => {
  const book = {
    ...createBook(title, author, isbn),
    sales,
    availibility,
    isbn
  };

  bookList.push(book);
  return book;
};
```

Perfect! Instead of creating a new `Book` instance each time we add a copy, we can effectively use the already existing `Book` instance for that particular copy. Let's create 5 copies of 3 books: Harry Potter, To Kill a Mockingbird, and The Great Gatsby.

```
addBook("Harry Potter", "JK Rowling", "AB123", false, 100);
addBook("Harry Potter", "JK Rowling", "AB123", true, 50);
addBook("To Kill a Mockingbird", "Harper Lee", "CD345", true, 10);
addBook("To Kill a Mockingbird", "Harper Lee", "CD345", false, 20);
addBook("The Great Gatsby", "F. Scott Fitzgerald", "EF567", false, 20);
```

Although there are 5 copies, we only have 3 Book instances!

```
class Book {
  constructor(title, author, isbn) {
    this.title = title;
    this.author = author;
    this.isbn = isbn;
  }
}

const books = new Map();
const bookList = [];

const addBook = (title, author, isbn, availability, sales) => {
  const book = {
    ...createBook(title, author, isbn),
    sales,
    availability,
    isbn
  };

  bookList.push(book);
  return book;
};
```

```
const createBook = (title, author, isbn) => {
  const existingBook = books.has(isbn);

  if (existingBook) {
    return books.get(isbn);
  }

  const book = new Book(title, author, isbn);
  books.set(isbn, book);

  return book;
};

addBook("Harry Potter", "JK Rowling", "AB123", false, 100);
addBook("Harry Potter", "JK Rowling", "AB123", true, 50);
addBook("To Kill a Mockingbird", "Harper Lee", "CD345", true, 10);
addBook("To Kill a Mockingbird", "Harper Lee", "CD345", false, 20);
addBook("The Great Gatsby", "F. Scott Fitzgerald", "EF567", false, 20);

console.log("Total amount of copies: ", bookList.length);
console.log("Total amount of books: ", books.size);
```

The flyweight pattern is useful when you're creating a huge number of objects, which could potentially drain all available RAM. It allows us to minimize the amount of consumed memory.

In JavaScript, we can easily solve this problem through prototypal inheritance. Nowadays, hardware has GBs of RAM, which makes the flyweight pattern less important.

Factory Pattern

Use a factory function in order to create objects

With the factory pattern we can use factory functions in order to create new objects. A function is a factory function when it returns a new object without the use of the new keyword!

Say that we need many users for our application. We can create new users with a `firstName`, `lastName`, and `email` property. The factory function adds a `fullName` property to the newly created object as well, which returns the `firstName` and the `lastName`.

```
const createUser = ({ firstName, lastName, email }) => ({
  firstName,
  lastName,
  email,
  fullName() {
    return `${this.firstName} ${this.lastName}`;
  }
});
```

Perfect! We can now easily create multiple users by invoking the `createUser` function.

```
const createUser = ({ firstName, lastName, email }) => ({
  firstName,
  lastName,
  email,
  fullName() {
    return `${this.firstName} ${this.lastName}`;
  }
});

const user1 = createUser({
  firstName: "John",
  lastName: "Doe",
  email: "john@doe.com"
});

const user2 = createUser({
  firstName: "Jane",
  lastName: "Doe",
  email: "jane@doe.com"
});

console.log(user1);
console.log(user2);
```

The factory pattern can be useful if we're creating relatively complex and configurable objects. It could happen that the values of the keys and values are dependent on a certain environment or configuration. With the factory pattern, we can easily create new objects that contain the custom keys and values!

```
const createObjectFromArray = ([key, value]) => ({
  [key]: value
});

createObjectFromArray(["name", "John"]); // { name: "John" }
```

Pros

The factory pattern is useful when we have to create multiple smaller objects that share the same properties. A factory function can easily return a custom object depending on the current environment, or user-specific configuration.

Cons

In JavaScript, the factory pattern isn't much more than a function that returns an object without using the new keyword. ES6 arrow functions allow us to create small factory functions that implicitly return an object each time.

However, in many cases it may be more memory efficient to create new instances instead of new objects each time.

```
class User {
  constructor(firstName, lastName, email) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.email = email;
  }

  fullName() {
    return `${this.firstName} ${this.lastName}`;
  }
}

const user1 = new User({
  firstName: "John",
  lastName: "Doe",
  email: "john@doe.com"
});

const user2 = new User({
  firstName: "Jane",
  lastName: "Doe",
  email: "jane@doe.com"
});
```

Compound Pattern

Create multiple components that work together to perform a single task

In our application, we often have components that belong to each other. They're dependent on each other through the shared state, and share logic together. You often see this with components like select, dropdown components, or menu items. The compound component pattern allows you to create components that all work together to perform a task.

Context API

Let's look at an example: we have a list of squirrel images! Besides just showing squirrel images, we want to add a button that makes it possible for the user to edit or delete the image. We can implement a `FlyOut` component that shows a list when the user toggles the component.

Within a `FlyOut` component, we essentially have three things:

- The `FlyOut` wrapper, which contains the toggle button and the list
- The `Toggle` button, which toggles the List
- The `List` , which contains the list of menu items

Using the Compound component pattern with React's Context API is perfect for this example!

First, let's create the `FlyOut` component. This component keeps the state, and returns a `FlyOutProvider` with the value of the toggle to all the children it receives.

```
const FlyOutContext = createContext();

function FlyOut(props) {
  const [open, toggle] = useState(false);

  const providerValue = { open, toggle };

  return (
    <FlyOutContext.Provider value={providerValue}>
      {props.children}
    </FlyOutContext.Provider>
  );
}
```

We now have a stateful `FlyOut` component that can pass the value of open and toggle to its children!

Let's create the `Toggle` component. This component simply renders the component on which the user can click in order to toggle the menu.

```
function Toggle() {
  const { open, toggle } = useContext(FlyOutContext);

  return (
    <div onClick={() => toggle(!open)}>
      <Icon />
    </div>
  );
}
```

In order to actually give `Toggle` access to the `FlyOutContext` provider, we need to render it as a child component of `FlyOut`! We could just simply render this as a child component. However, we can also make the `Toggle` component a property of the `FlyOut` component!

```
const FlyOutContext = createContext();

function FlyOut(props) {
  const [open, toggle] = useState(false);

  return (
    <FlyOutContext.Provider value={{ open, toggle }}>
      {props.children}
    </FlyOutContext.Provider>
  );
}

function Toggle() {
  const { open, toggle } = useContext(FlyOutContext);

  return (
    <div onClick={() => toggle(!open)}>
      <Icon />
    </div>
  );
}

FlyOut.Toggle = Toggle;
```

This means that if we ever want to use the `FlyOut` component in any file, we only have to import `FlyOut`!

```
import React from "react";
import { FlyOut } from "./FlyOut";

export default function FlyoutMenu() {
  return (
    <FlyOut>
      <FlyOut.Toggle />
    </FlyOut>
  );
}
```

Just a toggle is not enough. We also need to have a `List` with list items, which open and close based on the value of open.

```
function List({ children }) {
  const { open } = React.useContext(FlyOutContext);
  return open && <ul>{children}</ul>;
}

function Item({ children }) {
  return <li>{children}</li>;
}
```

The `List` component renders its children based on whether the value of open is true or false. Let's make `List` and `Item` a property of the `FlyOut` component, just like we did with the `Toggle` component.

```
const FlyOutContext = createContext();

function FlyOut(props) {
  const [open, toggle] = useState(false);

  return (
    <FlyOutContext.Provider value={{ open, toggle }}>
      {props.children}
    </FlyOutContext.Provider>
  );
}

function Toggle() {
  const { open, toggle } = useContext(FlyOutContext);

  return (
    <div onClick={() => toggle(!open)}>
      <Icon />
    </div>
  );
}

function List({ children }) {
  const { open } = useContext(FlyOutContext);
  return open && <ul>{children}</ul>;
}

function Item({ children }) {
  return <li>{children}</li>;
}

FlyOut.Toggle = Toggle;
FlyOut.List = List;
FlyOut.Item = Item;
```

We can now use them as properties on the `FlyOut` component! In this case, we want to show two options to the user: Edit and Delete. Let's create a `FlyOut.List` that renders two `FlyOut.Item` components, one for the Edit option, and one for the Delete option.

```
import React from "react";
import { FlyOut } from "./FlyOut";

export default function FlyoutMenu() {
  return (
    <FlyOut>
      <FlyOut.Toggle />
      <FlyOut.List>
        <FlyOut.Item>Edit</FlyOut.Item>
        <FlyOut.Item>Delete</FlyOut.Item>
      </FlyOut.List>
    </FlyOut>
  );
}
```

Perfect! We just created an entire `FlyOut` component without adding any state in the `FlyOutMenu` itself!

The compound pattern is great when you're building a component library. You'll often see this pattern when using UI libraries like Semantic UI.

`React.Children.map`

We can also implement the Compound Component pattern by mapping over the children of the component. We can add the open and toggle properties to these elements, by cloning them with the additional props.

```
export function FlyOut(props) {
  const [open, toggle] = React.useState(false);

  return (
    <div>
      {React.Children.map(props.children, child =>
        React.cloneElement(child, { open, toggle })
      )}
    </div>
  );
}
```

All children components are cloned, and passed the value of open and toggle. Instead of having to use the Context API like in the previous example, we now have access to these two values through props.

```jsx
import React from "react";
import Icon from "./Icon";

const FlyOutContext = React.createContext();

export function FlyOut(props) {
  const [open, toggle] = React.useState(false);

  return (
    <div>
      {React.Children.map(props.children, child =>
        React.cloneElement(child, { open, toggle })
      )}
    </div>
  );
}

function Toggle() {
  const { open, toggle } = React.useContext(FlyOutContext);

  return (
    <div className="flyout-btn" onClick={() => toggle(!open)}>
      <Icon />
    </div>
  );
}

function List({ children }) {
  const { open } = React.useContext(FlyOutContext);
  return open && <ul className="flyout-list">{children}</ul>;
}

function Item({ children }) {
  return <li className="flyout-item">{children}</li>;
}

FlyOut.Toggle = Toggle;
FlyOut.List = List;
FlyOut.Item = Item;
```

Pros

Compound components manage their own internal state, which they share among the several child components. When implementing a compound component, we don't have to worry about managing the state ourselves.

When importing a compound component, we don't have to explicitly import the child components that are available on that component.

```
import { FlyOut } from "./FlyOut";

export default function FlyoutMenu() {
  return (
    <FlyOut>
      <FlyOut.Toggle />
      <FlyOut.List>
        <FlyOut.Item>Edit</FlyOut.Item>
        <FlyOut.Item>Delete</FlyOut.Item>
      </FlyOut.List>
    </FlyOut>
  );
}
```

Cons

When using the `React.children.map` to provide the values, the component nesting is limited. Only direct children of the parent component will have access to the open and toggle props, meaning we can't wrap any of these components in another component.

```
export default function FlyoutMenu() {
  return (
    <FlyOut>
      {/* This breaks */}
      <div>
        <FlyOut.Toggle />
        <FlyOut.List>
          <FlyOut.Item>Edit</FlyOut.Item>
          <FlyOut.Item>Delete</FlyOut.Item>
        </FlyOut.List>
      </div>
    </FlyOut>
  );
}
```

Cloning an element with `React.cloneElement` performs a shallow merge. Already existing props will be merged together with the new props that we pass. This could end up in a naming collision, if an already existing prop has the same name as the props we're passing to the `React.cloneElement` method. As the props are shallowly merged, the value of that prop will be overwritten with the latest value that we pass.

Command Pattern

Decouple methods that execute tasks by sending commands to a commander

With the Command Pattern, we can decouple objects that execute a certain task from the object that calls the method.

Let's say we have an online food delivery platform. Users can place, track, and cancel orders.

```
class OrderManager() {
  constructor() {
    this.orders = []
  }

  placeOrder(order, id) {
    this.orders.push(id)
    return `You have successfully ordered ${order} (${id})`;
  }

  trackOrder(id) {
    return `Your order ${id} will arrive in 20 minutes.`
  }

  cancelOrder(id) {
    this.orders = this.orders.filter(order => order.id !== id)
    return `You have canceled your order ${id}`
  }
}
```

On the `OrderManager` class, we have access to the `placeOrder`, `trackOrder` and `cancelOrder` methods. It would be totally valid JavaScript to just use these methods directly!

```
const manager = new OrderManager();

manager.placeOrder("Pad Thai", "1234");
manager.trackOrder("1234");
manager.cancelOrder("1234");
```

However, there are downsides to invoking the methods directly on the manager instance. It could happen that we decide to rename certain methods later on, or the functionality of the methods change.

Say that instead of calling it `placeOrder`, we now rename it to `addOrder`! This would mean that we would have to make sure that we don't call the `placeOrder` method anywhere in our codebase, which could be very tricky in larger applications.

Instead, we want to decouple the methods from the manager object, and create separate command functions for each command!

Let's refactor the `OrderManager` class: instead of having the `placeOrder`, `cancelOrder` and `trackOrder` methods, it will have one single method: `execute`. This method will execute any command it's given.

Each command should have access to the orders of the manager, which we'll pass as its first argument.

```
class OrderManager {
  constructor() {
    this.orders = [];
  }

  execute(command, ...args) {
    return command.execute(this.orders, ...args);
  }
}
```

We need to create three Commands for the order manager:

- `PlaceOrderCommand`
- `CancelOrderCommand`
- `TrackOrderCommand`

```
class Command {
  constructor(execute) {
    this.execute = execute;
  }
}

function PlaceOrderCommand(order, id) {
  return new Command(orders => {
    orders.push(id);
    return `You have successfully ordered ${order} (${id})`;
  });
}

function CancelOrderCommand(id) {
  return new Command(orders => {
    orders = orders.filter(order => order.id !== id);
    return `You have canceled your order ${id}`;
  });
}

function TrackOrderCommand(id) {
  return new Command(() => `Your order ${id} will arrive in 20 minutes.`);
}
```

Perfect! Instead of having the methods directly coupled to the `OrderManager` instance, they're now separate, decoupled functions that we can invoke through the execute method that's available on the `OrderManager`.

```
class OrderManager {
  constructor() {
    this.orders = [];
  }

  execute(command, ...args) {
    return command.execute(this.orders, ...args);
  }
}

class Command {
  constructor(execute) {
    this.execute = execute;
  }
}

function PlaceOrderCommand(order, id) {
  return new Command(orders => {
    orders.push(id);
    console.log(`You have successfully ordered ${order} (${id})`);
  });
}

function CancelOrderCommand(id) {
  return new Command(orders => {
    orders = orders.filter(order => order.id !== id);
    console.log(`You have canceled your order ${id}`);
  });
}

function TrackOrderCommand(id) {
  return new Command(() =>
    console.log(`Your order ${id} will arrive in 20 minutes.`)
  );
}

const manager = new OrderManager();

manager.execute(new PlaceOrderCommand("Pad Thai", "1234"));
manager.execute(new TrackOrderCommand("1234"));
manager.execute(new CancelOrderCommand("1234"));
```

Pros

The command pattern allows us to decouple methods from the object that executes the operation. It gives you more control if you're dealing with commands that have a certain lifespan, or commands that should be queued and executed at specific times.

Cons

The use cases for the command pattern are quite limited, and often adds unnecessary boilerplate to an application.