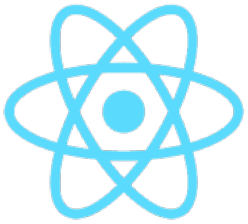
COMPSCI 326

Web Programming

Lecture 11: Static React



React

What is React?

React Components

React is divided into component objects

- These are implemented in JavaScript
- Well, JSX - JavaScript with Embedded "HTML"
- JSX files are compiled into pure JavaScript

Components are arranged in a tree-like relationship

- Top-level components create/communicate with child components
- Child components can also have child components
- Components are implement in JavaScript
- Components are instantiated in HTML-like snippets

Model-View-Controller (MVC)

Model-View-Controller (MVC) is a software architectural pattern for implementing user interfaces on computers. It divides a given software application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user.

Traditionally used for desktop graphical user interfaces (GUIs), this architecture has become popular for *designing web applications*.

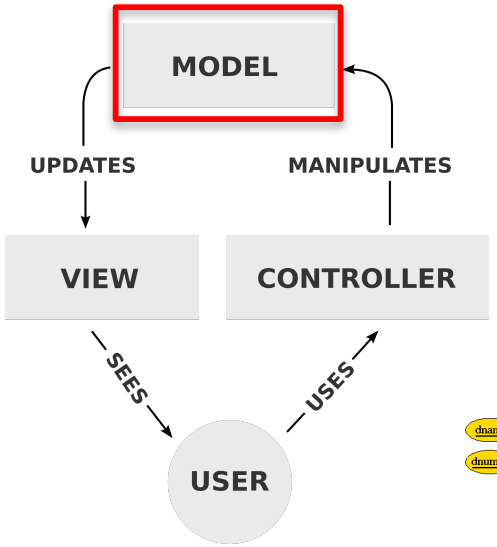
As with other software patterns, MVC expresses the "core of the solution" to a problem while allowing it to be adapted for each system. (wikipedia)

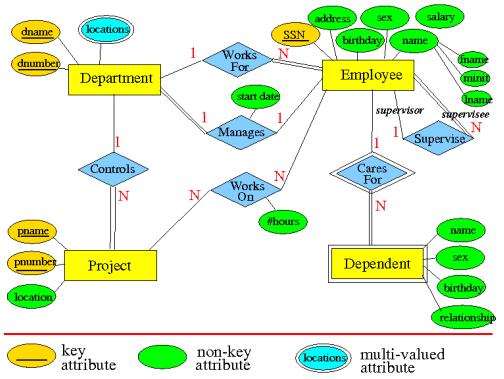
Model-View-Controller: The Model

The central component of MVC, **the model**, captures the behavior of the application in terms of its problem domain, independent of the user interface.

The *model* turns out to be of central importance not only to the design and implementation of user interfaces, but also to the application as a whole. We will turn our attention to *data modeling* in a couple of weeks.

The model directly manages the data, logic, and rules of the application.

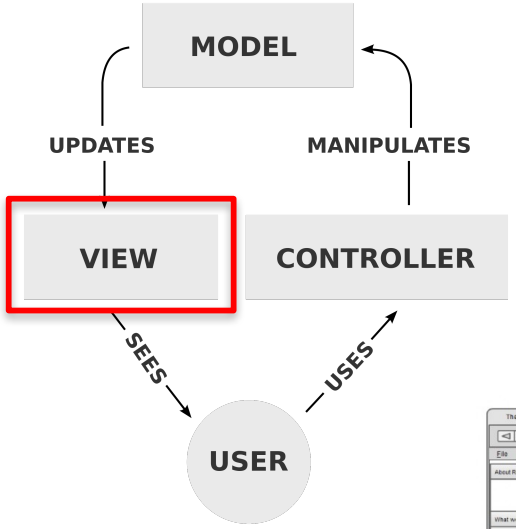


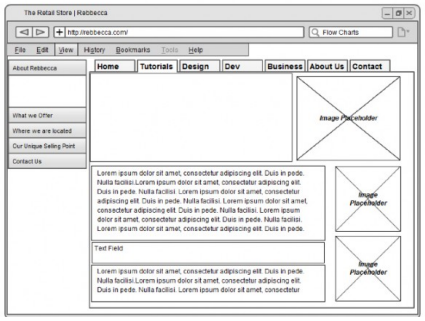


Model-View-Controller: The View

A **view** is a (visual) representation of its model. It would ordinarily highlight certain attributes of the model and suppress others. It is thus acting as a presentation filter.

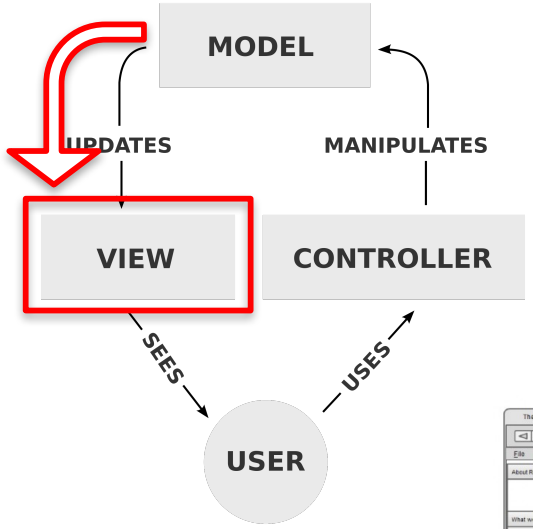
Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants.

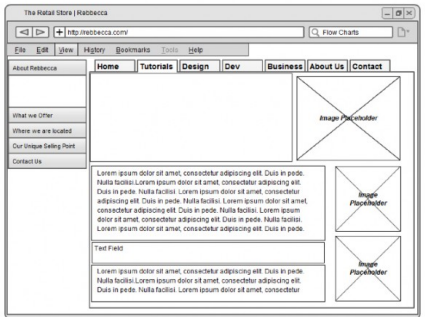




Model-View-Controller: The View + Model

A **view** is attached to its model (or model part) and gets the data necessary for the presentation from the model by asking questions.



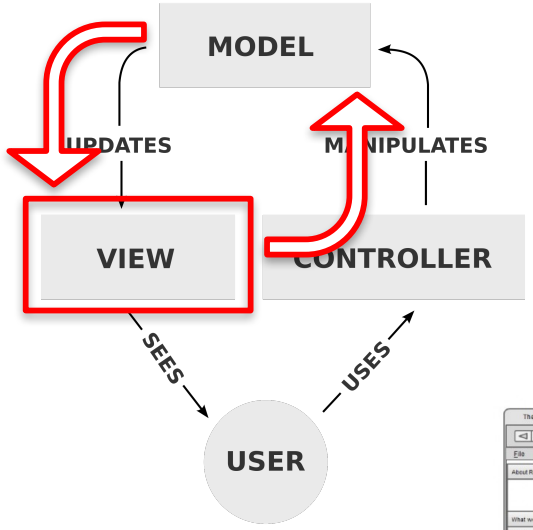


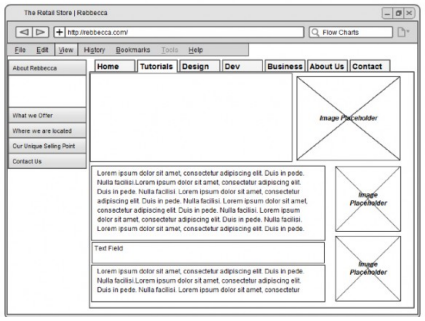
Model-View-Controller: The View + Model

A **view** is attached to its model (or model part) and gets the data necessary for the presentation from the model by asking questions.

It may also update the model by sending appropriate messages.

All these questions and messages have to be in the terminology of the model, the view will therefore have to know the semantics of the attributes of the model it represents.

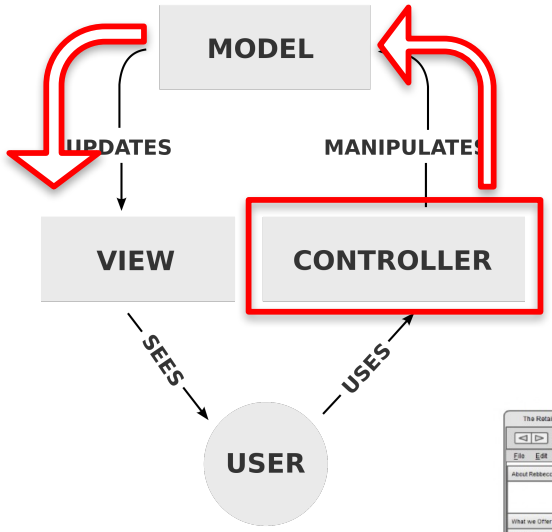


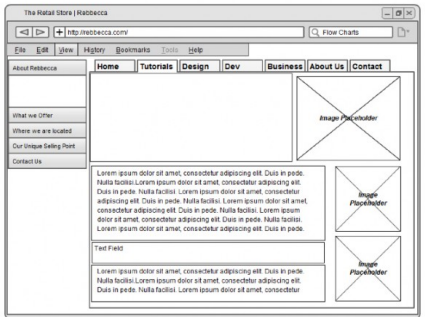


Model-View-Controller: The Controller

A **controller** is the link between a user and the system. It provides the user with input by arranging for relevant views to present themselves in appropriate places on the screen.

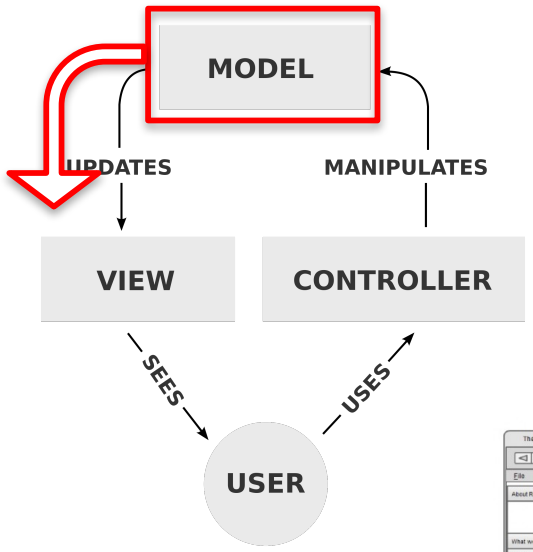
It provides means for user output by presenting the user with menus or other means of giving commands and data. The controller receives such user output, translates it into the appropriate messages and pass these messages on to one or more of the views.

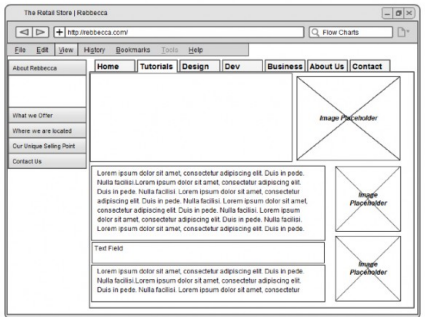




Model-View-Controller: MVC Interactions

A **model** stores data that is retrieved according to commands from the controller and displayed in the view.

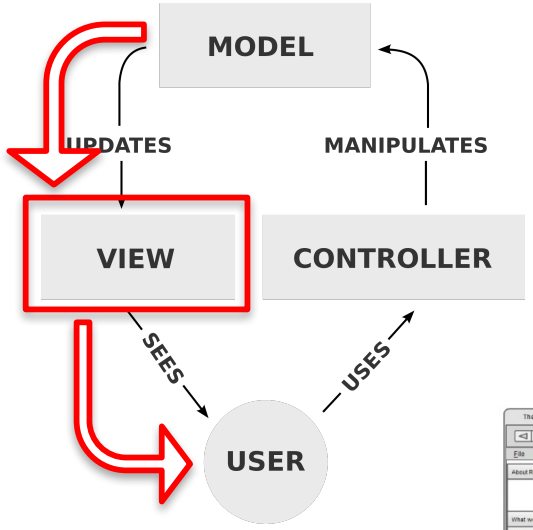


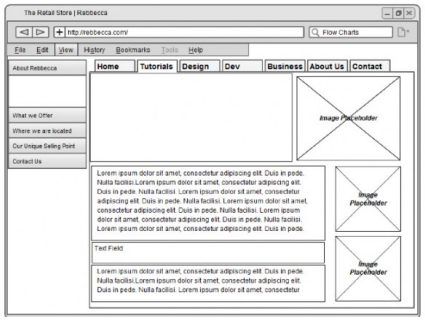


Model-View-Controller: MVC Interactions

A **model** stores data that is retrieved according to commands from the controller and displayed in the view.

A **view** generates new output to the user based on changes in the model.



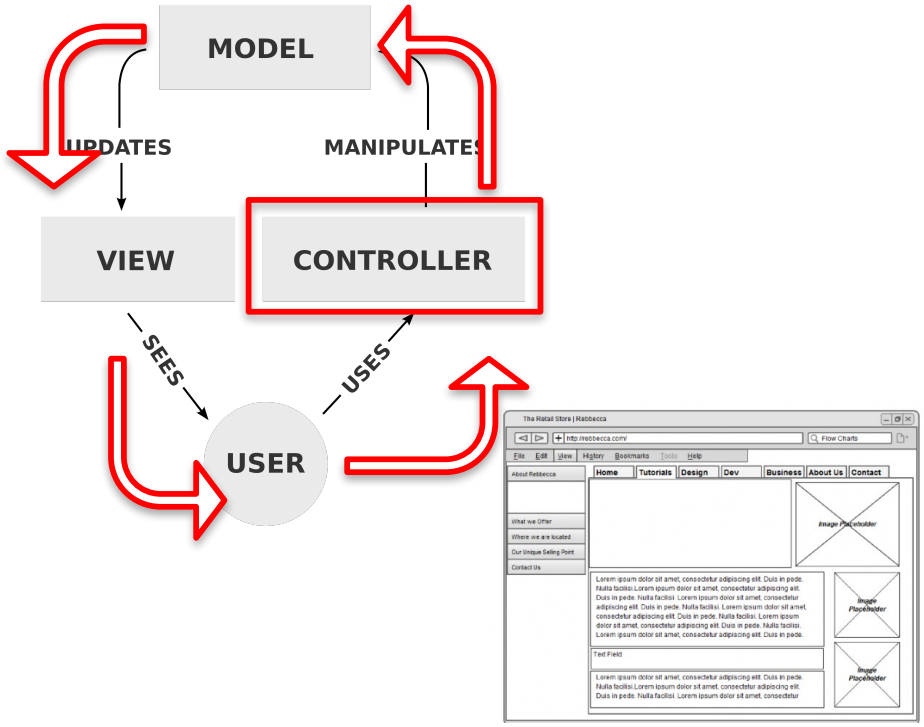


Model-View-Controller: MVC Interactions

A **model** stores data that is retrieved according to commands from the controller and displayed in the view.

A **view** generates new output to the user based on changes in the model.

A **controller** can send commands to the model to update the model's state (e.g. editing a document). It can also send commands to its associated view to change the view's presentation of the model (e.g. by scrolling through a document).



React: Just the UI

"Lots of people use React as the V in MVC. Since React makes no assumptions about the rest of your technology stack, it's easy to try out on a small feature in an existing project."

- https://facebook.github.io/react

React: Virtual DOM

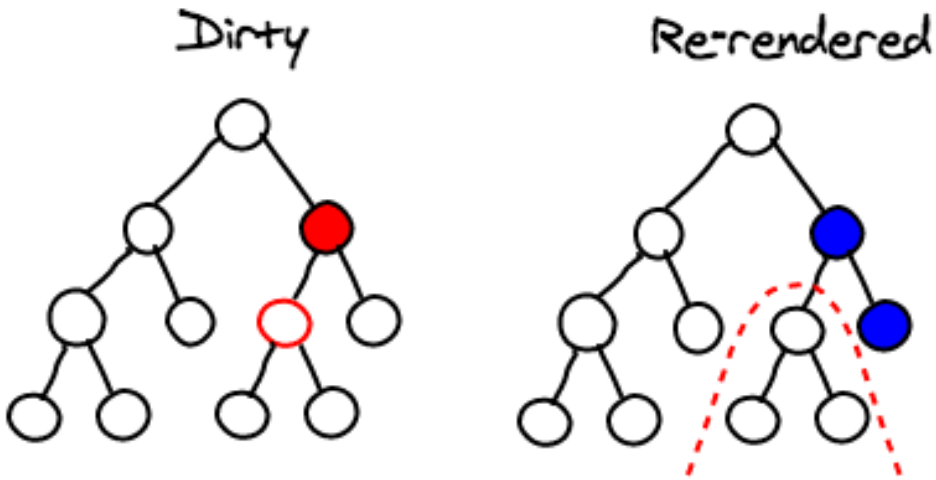
"React abstracts away the DOM from you, giving you a simpler programming model and better performance. React can also render on the server using Node, and it can power native apps using React Native."

- https://facebook.github.io/react

React: Virtual DOM

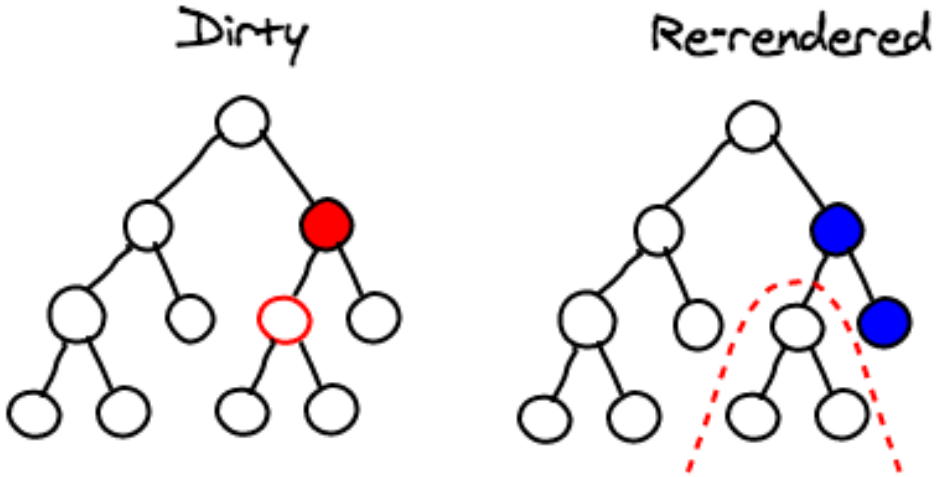
Since DOM manipulations are slow, React replicates the DOM virtually.

Your application talks to the virtual DOM that is very fast, and then React diffs the virtual DOM with the real DOM and applies all changes efficiently.



React: Virtual DOM & Changes

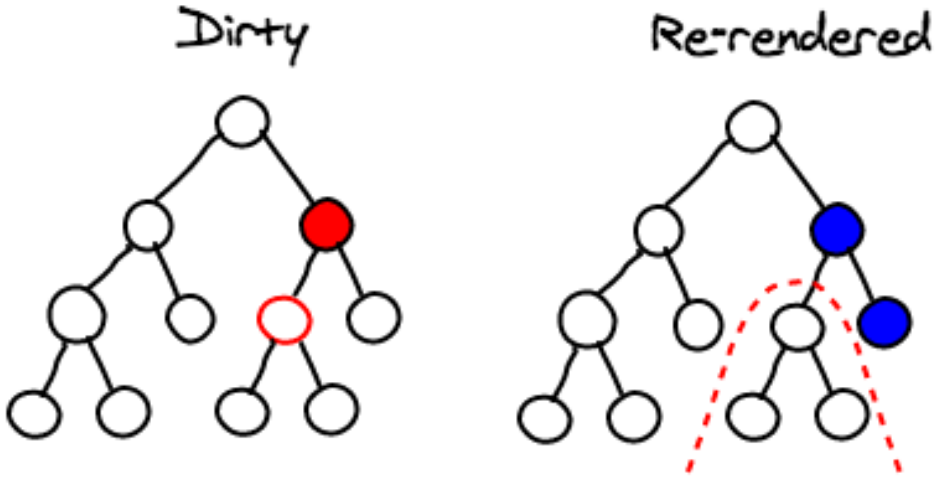
To track down model changes and apply them on the DOM (alias rendering) we have to be aware of two important things:



React: Virtual DOM & Changes

To track down model changes and apply them on the DOM (alias rendering) we have to be aware of two important things:

1. Identify when changes occur.
2. Identify which DOM element(s) need to be updated.

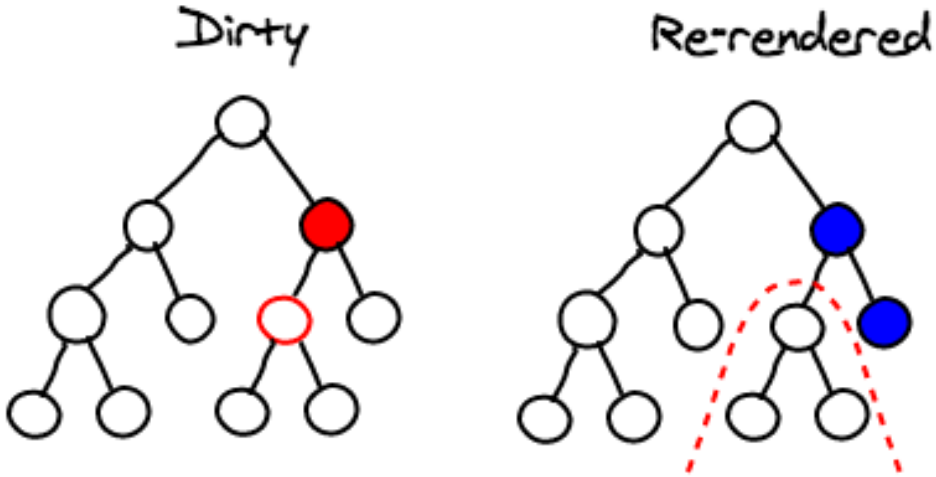


React: Virtual DOM & Changes

To track down model changes and apply them on the DOM (alias rendering) we have to be aware of two important things:

1. Identify when changes occur.
2. Identify which DOM element(s) need to be updated.

For the change detection, React uses an observer model instead of dirty checking (continuous model checking for changes). That’s why it doesn't have to calculate what is changed, it knows immediately. It reduces the calculations and make the app smoother.



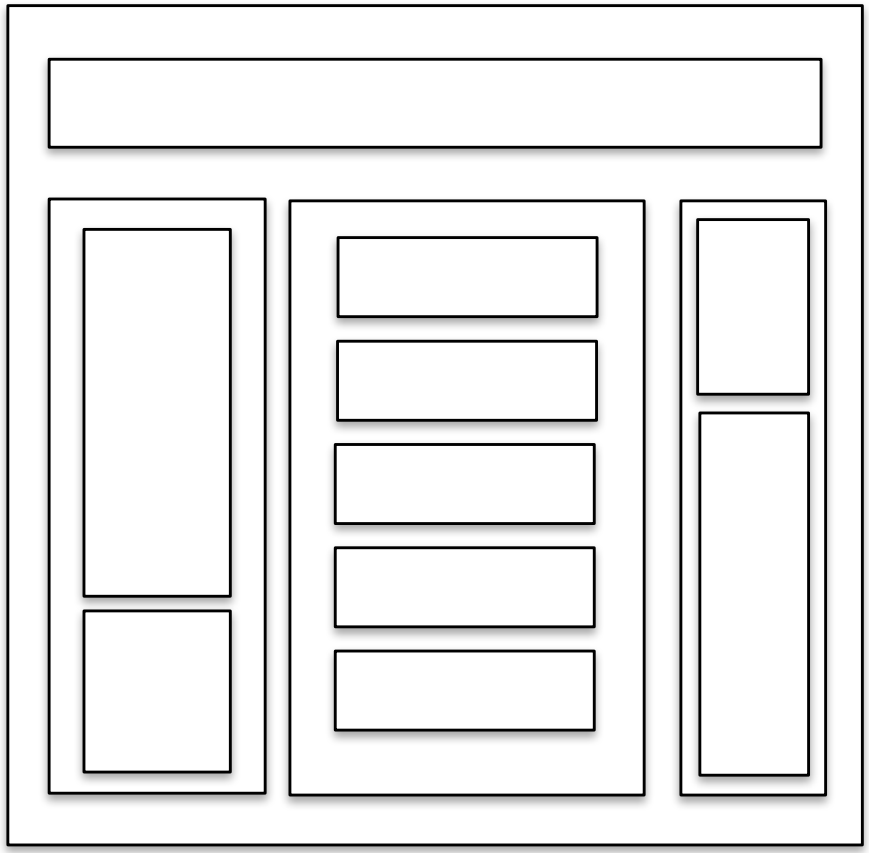
React: Data Flow

"React implements one-way reactive data flow which reduces boilerplate and is easier to reason about than traditional data binding"

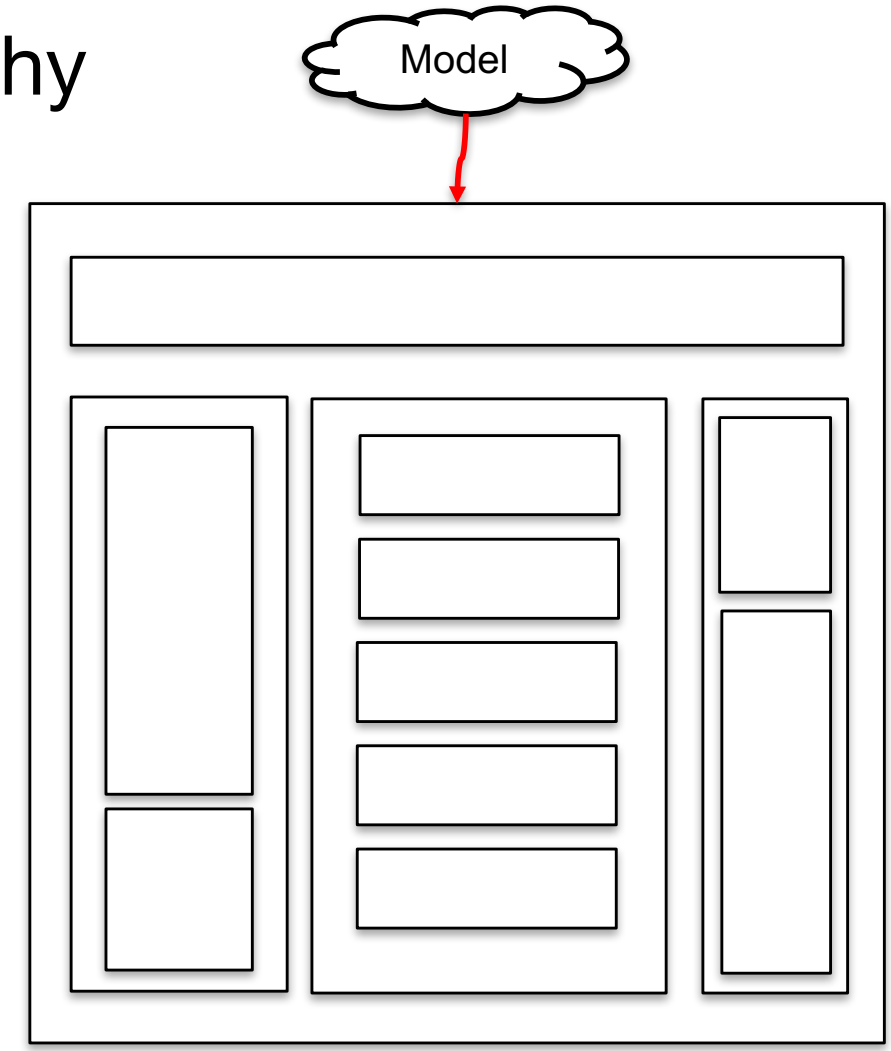
- https://facebook.github.io/react

React Component Hierarchy

React applications have a unidirectional data flow.



React Component Hierarchy



React applications have a unidirectional data flow.

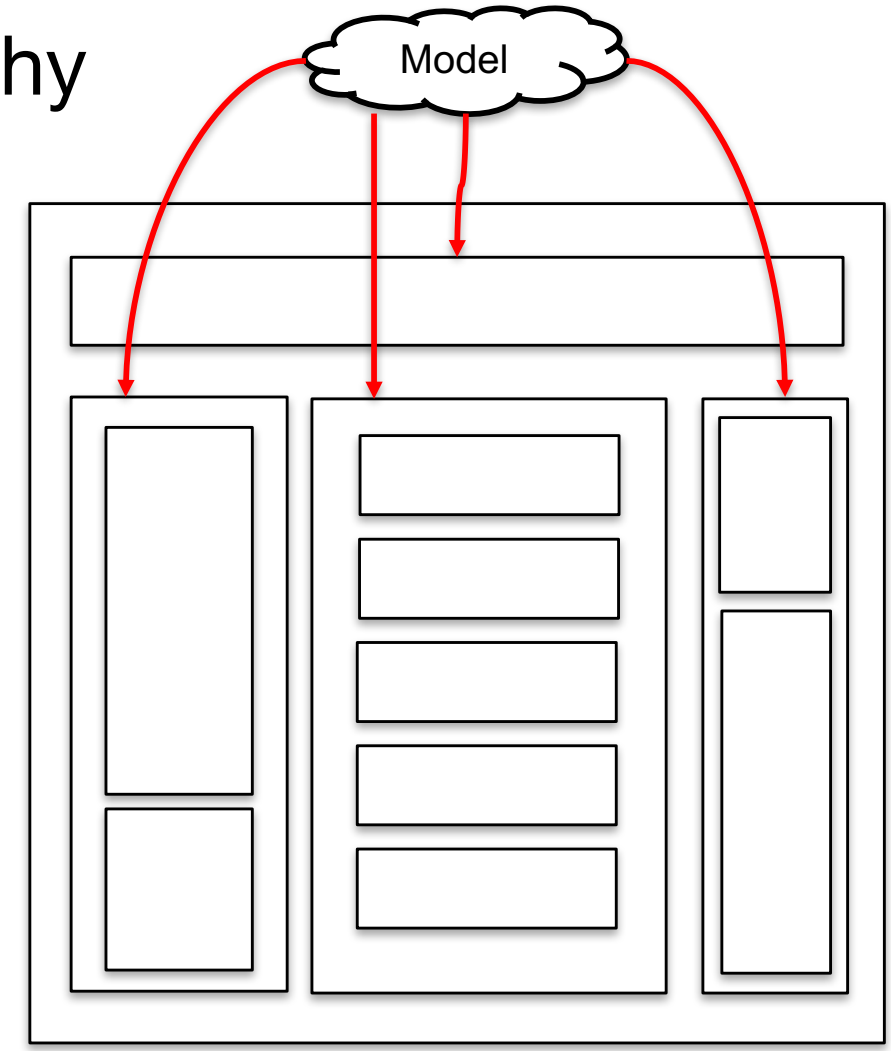
The user interface is constructed by feeding the model through the top-most component.

React Component Hierarchy

React applications have a unidirectional data flow.

The user interface is constructed by feeding the model through the top-most component.

The model data is then fed through contained UI components.



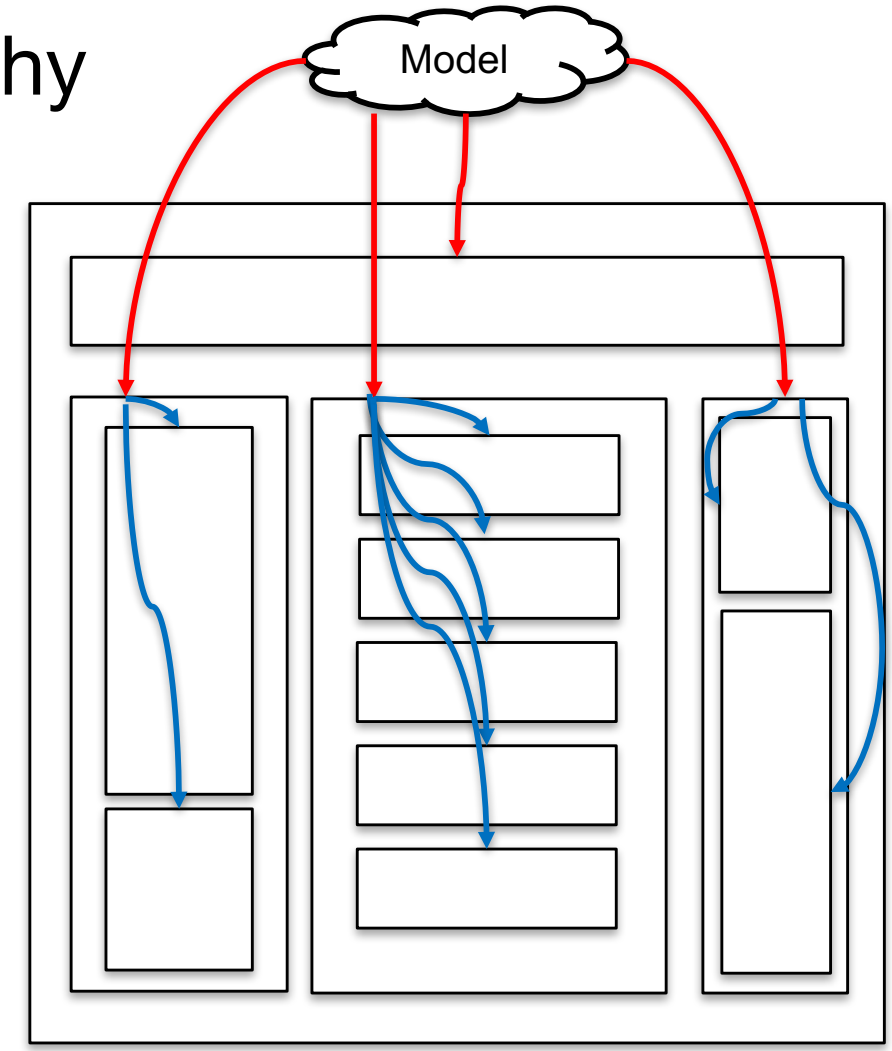
React Component Hierarchy

React applications have a unidirectional data flow.

The user interface is constructed by feeding the model through the top-most component.

The model data is then fed through contained UI components.

The contained components then feed the model data into their contained components.



React Component Hierarchy

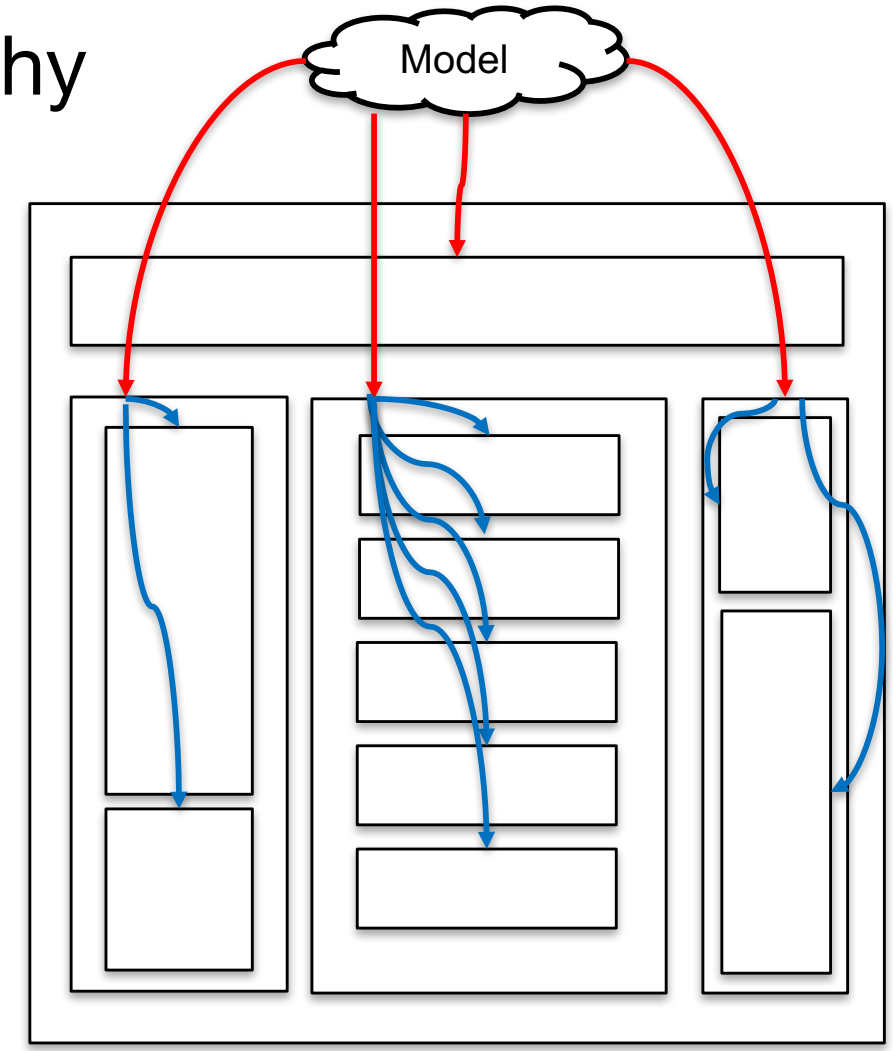
React applications have a unidirectional data flow.

The user interface is constructed by feeding the model through the top-most component.

The model data is then fed through contained UI components.

The contained components then feed the model data into their contained components.

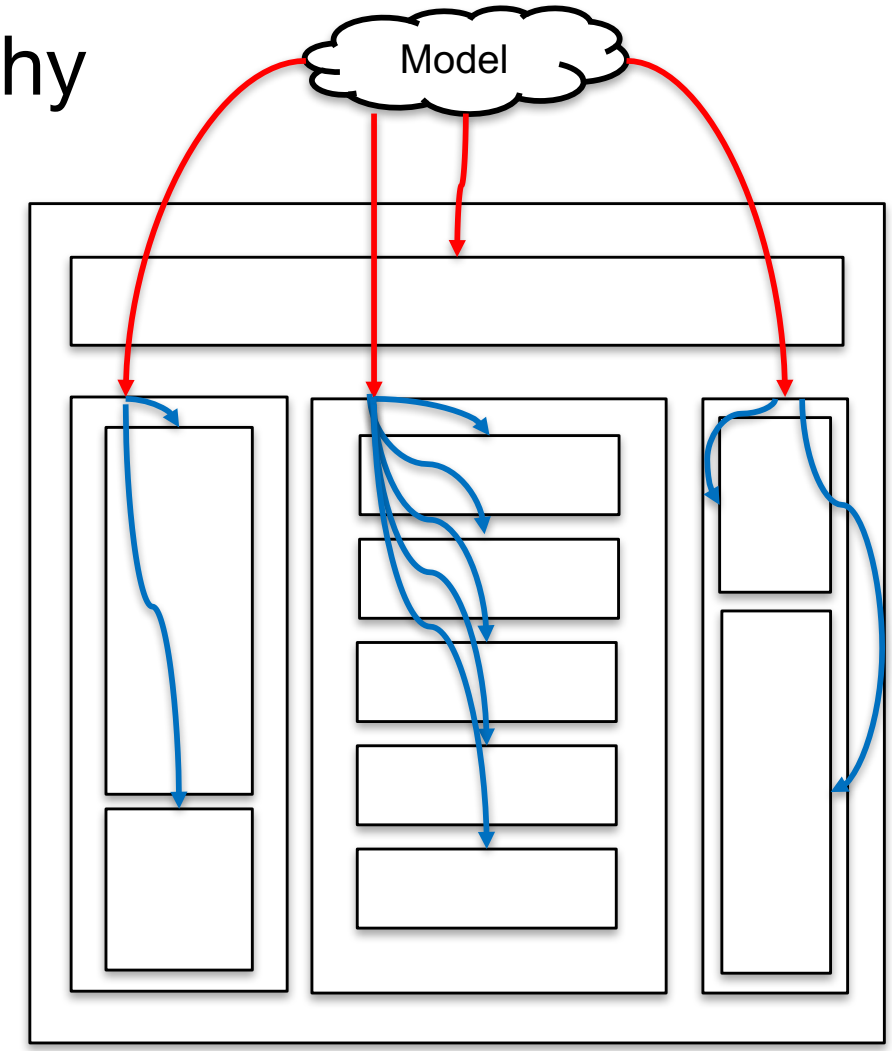
This process continues until all UI components have been completely constructed and rendered into the browser window.



React Component Hierarchy

React Component Properties

Each component is constructed by giving it **properties** (props for short).

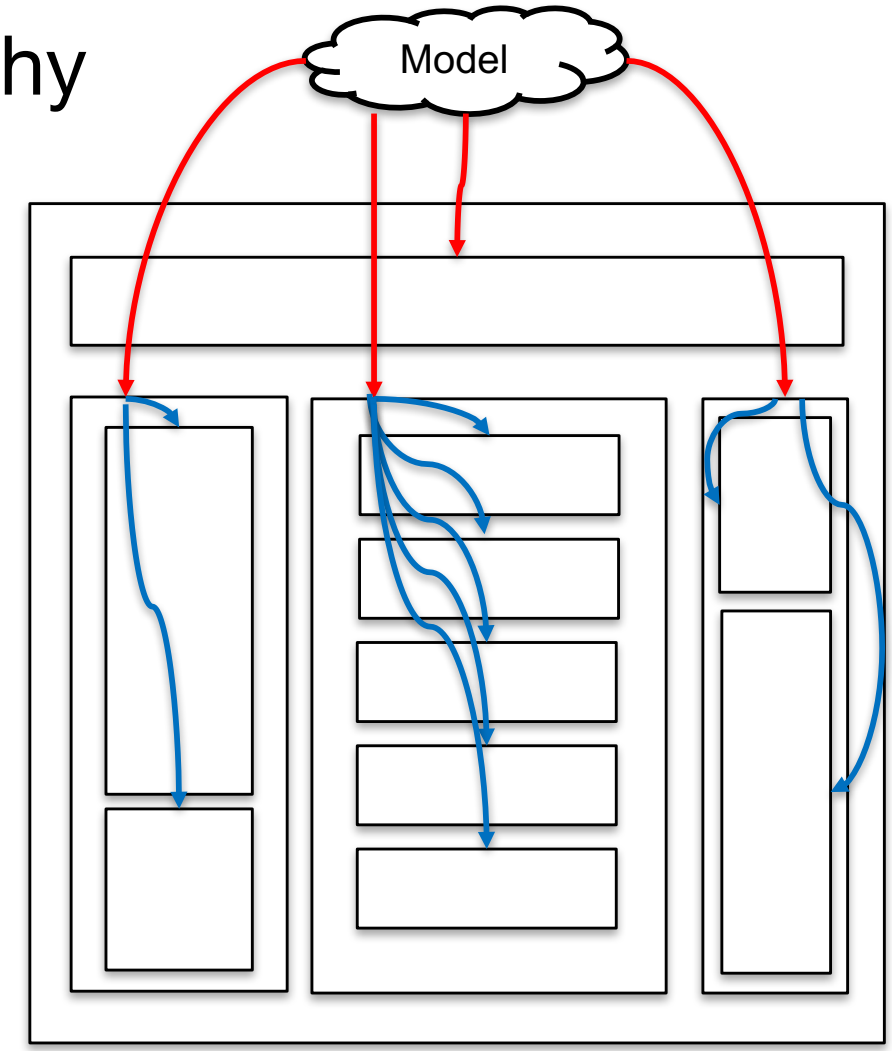


React Component Hierarchy

React Component Properties

Each component is constructed by giving it **properties** (props for short).

The *props* provide the data to each of the components as they are constructed.



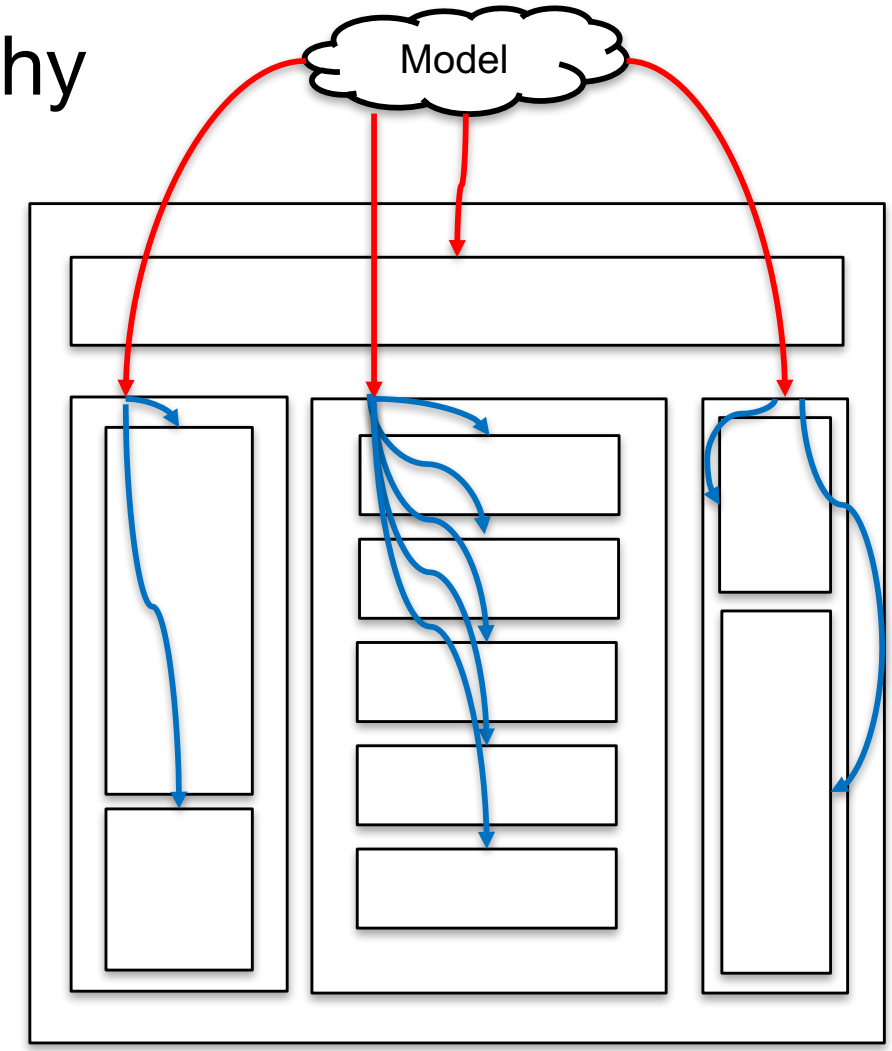
React Component Hierarchy

React Component Properties

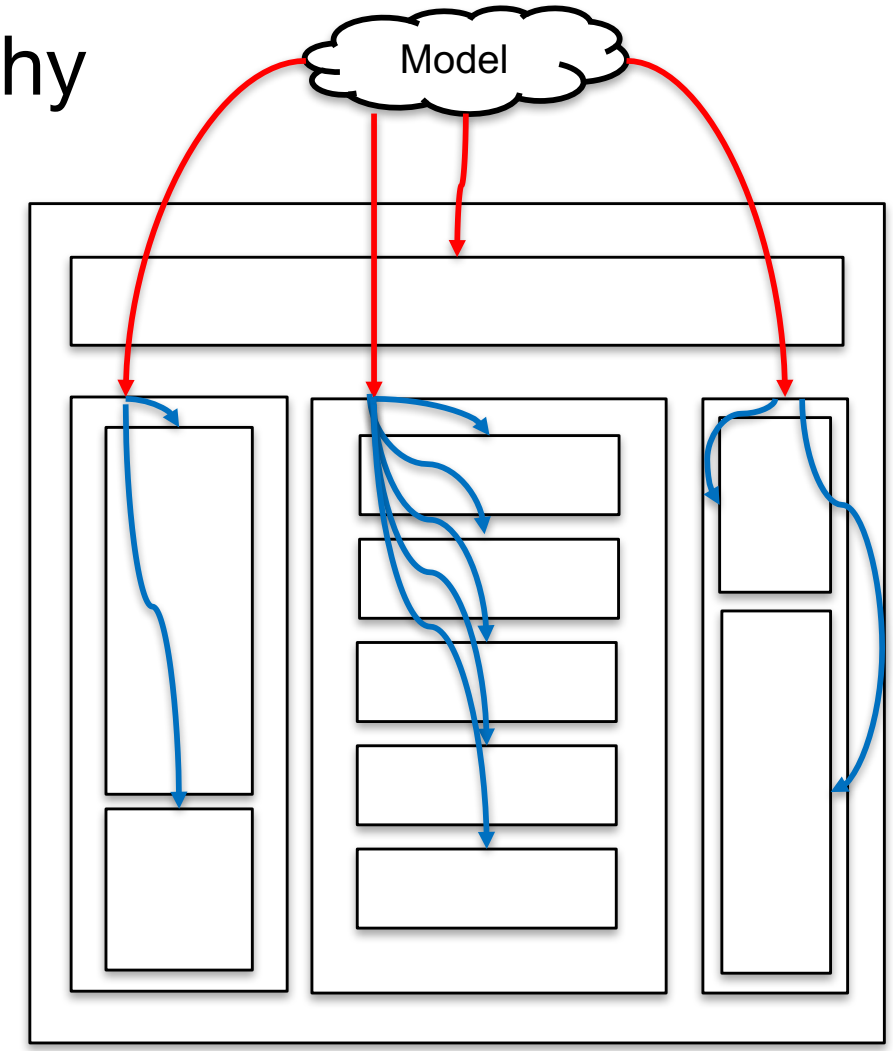
Each component is constructed by giving it **properties** (props for short).

The *props* provide the data to each of the components as they are constructed.

What those props are depends on the model and the particular UI components being constructed.



React Component Hierarchy



React Component Properties

Each component is constructed by giving it **properties** (props for short).

The *props* provide the data to each of the components as they are constructed.

What those props are depends on the model and the particular UI components being constructed.

Example: A UI component for displaying posts in a Facebook feed require data such as: the user who posted, the date of the post, the number of likes, and the content of the post.

Example React Component

```
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}

let mountNode = document.getElementById('app')

ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

This is a simple React application.

Example React Component

```
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}
```

```
let mountNode = document.getElementById('app')

ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

React components are defined as a JavaScript class that extends `React.Component`.

Example React Component

```
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}

let mountNode = document.getElementById('app')

ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

React components are defined as a JavaScript class that extends **`React.Component`**.

`React.Component` handles the machinery for rendering your component and expects your React component to implement the **`render()`** method.

Example React Component

```
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}

let mountNode = document.getElementById('app')

ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

React components are defined as a JavaScript class that extends **`React.Component`**.

`React.Component` handles the machinery for rendering your component and expects your React component to implement the **`render()`** method.

The **`render()`** method returns a React DOM element. React DOM elements can be easily created using JSX syntax.

Example React Component

```
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}

let mountNode = document.getElementById('app')

ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

React components are defined as a JavaScript class that extends **`React.Component`**.

`React.Component` handles the machinery for rendering your component and expects your React component to implement the **`render()`** method.

The **`render()`** method returns a React DOM element. React DOM elements can be easily created using JSX syntax.

JSX can reference JavaScript variables and render their value into the surrounding React DOM elements.

Example React Component

```
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}

let mountNode = document.getElementById('app')

ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

To render a React component into the browser window we must retrieve an element into which our component will be placed.

Here we assume that there exists an HTML element whose id is **app**.

Example React Component

```
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}

let mountNode = document.getElementById('app')

ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

To render a React component into the browser window we must retrieve an element into which our component will be placed.

Here we assume that there exists an HTML element whose id is **app**.

Lastly, we *instantiate* our React component, **`HelloMessage`**, using JSX and pass along a *prop* called **`name`** with the value "`John`".

Example React Component

```
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}

let mountNode = document.getElementById('app')

ReactDOM.render(<HelloMessage name="John" />, mountNode);
```


To render a React component into the browser window we must retrieve an element into which our component will be placed.

Here we assume that there exists an HTML element whose id is **app**.

Lastly, we *instantiate* our React component, **`HelloMessage`**, using JSX and pass along a *prop* called **`name`** with the value "`John`".

The **`name`** prop is added to the **`this.props`** object property of the instantiated **`HelloMessage`** class

Example React Component

```
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}

let mountNode = document.getElementById('app')

ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

Lastly, we render our React component using the special **`ReactDOM.render()`** method passing to it our newly constructed React component and the HTML DOM element we want our React component to be mounted in to (e.g., **`mountNode`**).

To render a React component into the browser window we must retrieve an element into which our component will be placed.

Here we assume that there exists an HTML element whose id is **`app`**.

Lastly, we *instantiate* our React component, **`HelloMessage`**, using JSX and pass along a *prop* called **`name`** with the value "`John`".

The **`name`** prop is added to the **`this.props`** object property of the instantiated **`HelloMessage`** class

Running a React Component

```
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}

let mountNode = document.getElementById('app')

ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

So, how do we actually “run” this react application?

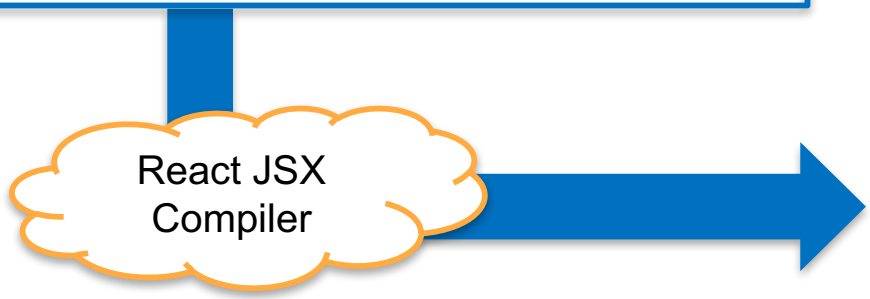
JSX is not really JavaScript, so we must compile it to plain JavaScript before it can be executed in the browser.

Running a React Component

```
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}

let mountNode = document.getElementById('app')

ReactDOM.render(<HelloMessage name="John" />, mountNode);
```



So, how do we actually “run” this react application?

JSX is not really JavaScript, so we must compile it to plain JavaScript before it can be executed in the browser.

```
class HelloMessage extends React.Component {
  render() {
    return React.createElement(
      "div",
      null,
      "Hello ",
      this.props.name
    );
  }
}

ReactDOM.render(React.createElement(
  HelloMessage, { name: "John" }),
  mountNode);
```

Running a React Component

```
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}

let mountNode = document.getElementById('app')

ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

How does this work?

So, how do we actually “run” this react application?

JSX is not really JavaScript, so we must compile it to plain JavaScript before it can be executed in the browser.

```
class HelloMessage extends React.Component {
  render() {
    return React.createElement(
      "div",
      null,
      "Hello ",
      this.props.name
    );
  }
}

ReactDOM.render(React.createElement(
  HelloMessage, { name: "John" }),
  mountNode);
```

Compiling React in the Browser

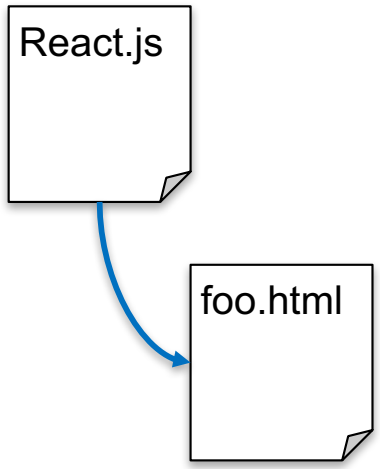
The simplest way to compile React/JSX into executable JavaScript is to do it in the browser.

Compiling React in the Browser

The simplest way to compile React/JSX into executable JavaScript is to do it in the browser.

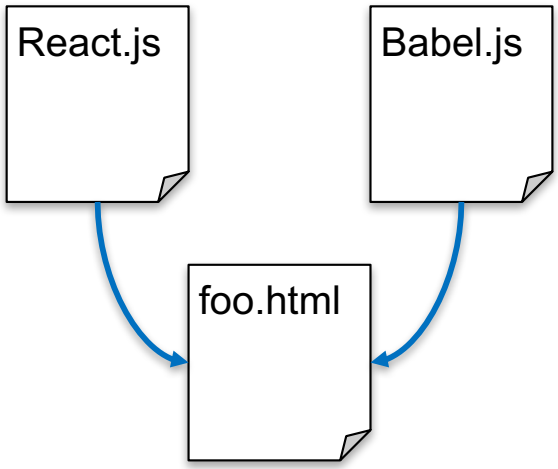
Compiling React in the Browser

The simplest way to compile React/JSX into executable JavaScript is to do it in the browser.



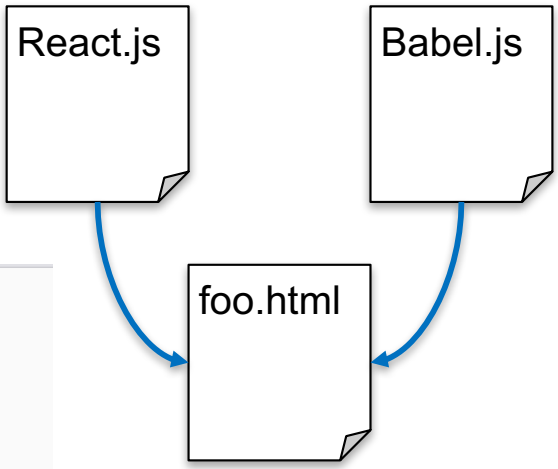
Compiling React in the Browser

The simplest way to compile React/JSX into executable JavaScript is to do it in the browser.



Compiling React in the Browser

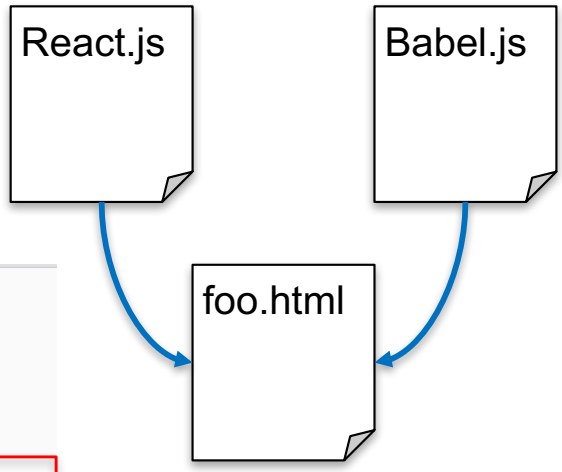
The simplest way to compile React/JSX into executable JavaScript is to do it in the browser.




```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>Hello React!</title>
  <script src="build/react.js"></script>
  <script src="build/react-dom.js"></script>
  <script src="https://unpkg.com/babel-core@5.8.38/browser.min.js"></script>
</head>
<body>
  <div id="example"></div>
  <script type="text/babel" src="01-react-hello-world.js"></script>
</body>
<div id="app"></div>
</html>
```

Compiling React in the Browser

The simplest way to compile React/JSX into executable JavaScript is to do it in the browser.

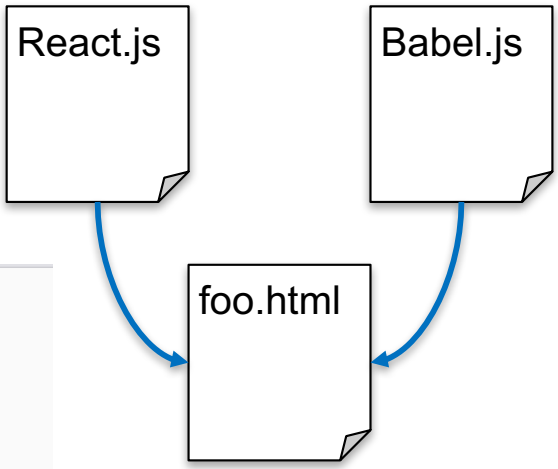



```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>Hello React!</title>
  <script src="build/react.js"></script>
  <script src="build/react-dom.js"></script>
  <script src="https://unpkg.com/babel-core@5.8.38/browser.min.js"></script>
</head>
<body>
  <div id="example"></div>
  <script type="text/babel" src="01-react-hello-world.js"></script>
</body>
<div id="app"></div>
</html>
```

First, we include the React libraries.

Compiling React in the Browser

The simplest way to compile React/JSX into executable JavaScript is to do it in the browser.



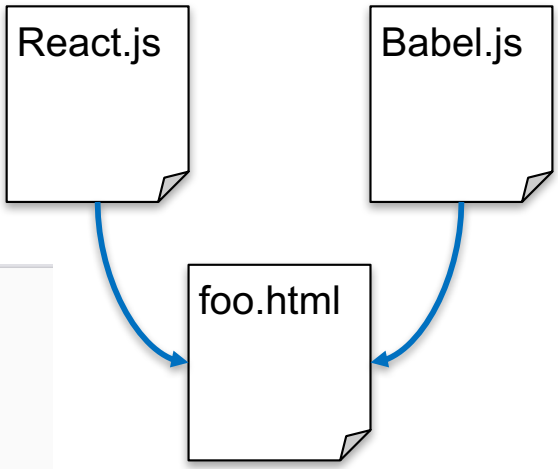

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>Hello React!</title>
  <script src="build/react.js"></script>
  <script src="build/react-dom.js"></script>
  <script src="https://unpkg.com/babel-core@5.8.38/browser.min.js"></script>
</head>
<body>
  <div id="example"></div>
  <script type="text/babel" src="01-react-hello-world.js"></script>
</body>
<div id="app"></div>
</html>
```

First, we include the React libraries.

Next, we include the Babel library.

Compiling React in the Browser

The simplest way to compile React/JSX into executable JavaScript is to do it in the browser.




```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>Hello React!</title>
  <script src="build/react.js"></script>
  <script src="build/react-dom.js"></script>
  <script src="https://unpkg.com/babel-core@5.8.38/browser.min.js"></script>
</head>
<body>
  <div id="example"></div>
  <script type="text/babel" src="01-react-hello-world.js"></script>
</body>
<div id="app"></div>
</html>
```

First, we include the React libraries.

Next, we include the Babel library.

Lastly, we include our JavaScript containing our React component. Notice the type given to this **`<script>`** tab.

Compiling React in the Browser

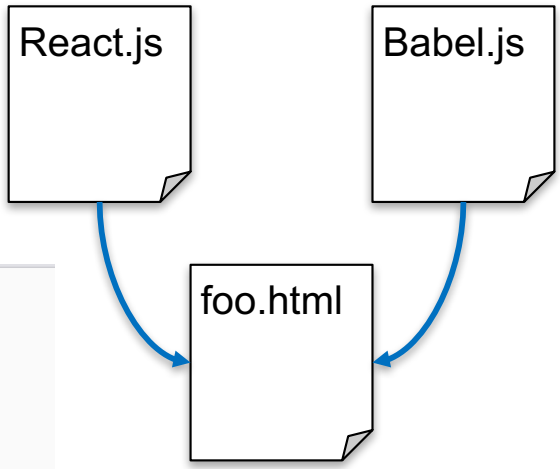
The simplest way to compile React/JSX into executable JavaScript is to do it in the browser.

```
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}

let mountNode = document.getElementById('app')

ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

```
    <script src="build/react-dom.js"></script>
    <script src="https://unpkg.com/babel-core@5.8.38/browser.min.js"></script>
  </head>
  <body>
    <div id="example"></div>
    <script type="text/babel" src="01-react-hello-world.js"></script>
  </body>
  <div id="app"></div>
  </html>
```



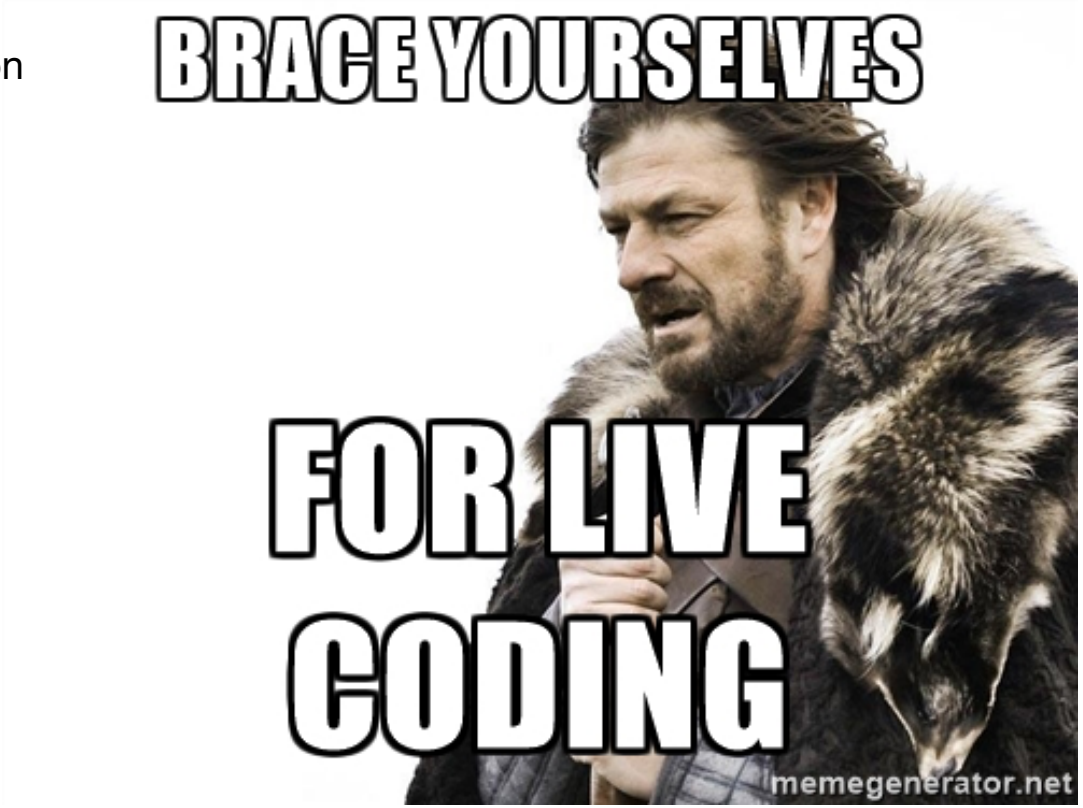
Also, notice the ** ** with the **id** app that we use in our React application to mount our React component.

React Example 01 – hello world

Let us take a look at the actual implementation of the previously described example.

The React component is defined in:
`01-react-hello-world.js`

The HTML file that sets the stage is:
`01-react-hello-world.html`



React Example 02 – Static Todo List

Let us take a look at the actual implementation of the previously described example.

The React component is defined in:
`02-react-static-todo.js`

The HTML file that sets the stage is:
`02-react-static-todo.html`