

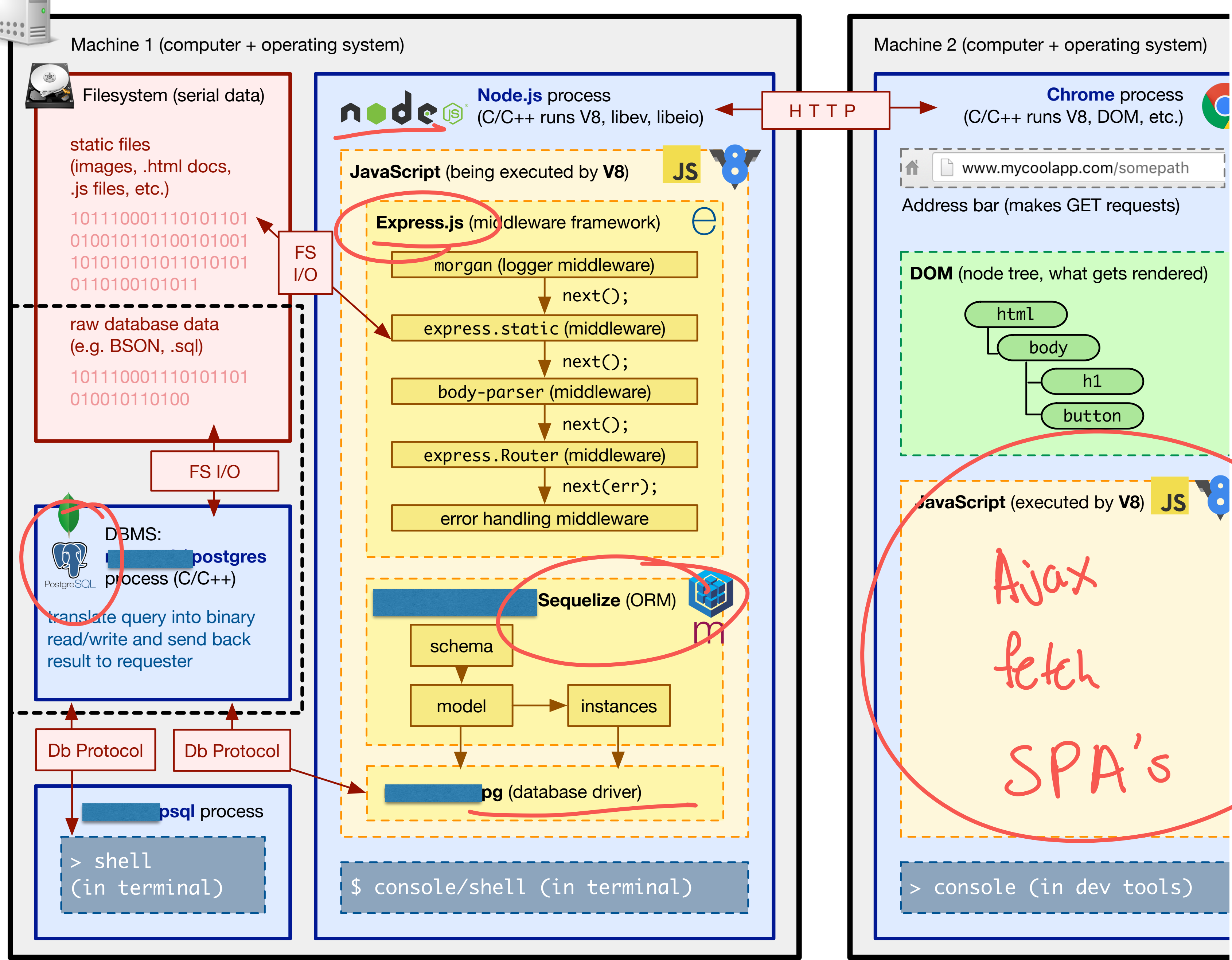


**FULLSTACK**  
ACADEMY

# Front-end with React.js

Server  
Side

Client  
Side



# Wikistack Front-End

module.exports = (page) => layout(html`

*HTML*

```
<h3>${page.title}
  <small> (<a href="/wiki/${page.slug}/similar">Similar</a>)</small>
</h3>
<h4>by <a href="/users/${page.author.id}">${page.author.name}</a></h4>
<ul>
  ${page.tags.map(tag => html`<li>${tag}</li>`)}
</ul>
<hr/>
<div class="page-body">${marked(page.content)}</div>
<hr/>
<a href="/wiki/${page.slug}/edit" class="btn btn-primary">edit this page</a>
<a href="/wiki/${page.slug}/delete" class="btn btn-danger">delete this page</a>
`);
```

## Trip-Planner

```
export function addItineraryItem (attraction) {
```

```
  //Make a dom element and append
```

```
  const itineraryItem = document.createElement('li');  
  itineraryItem.className = 'itinerary-item';  
  itineraryItem.append(attraction.name);
```

] DOM manipulation

```
  //Some other function for making my remove button
```

```
  const removeButton = makeRemoveBtn(itineraryItem, attractio
```

n)

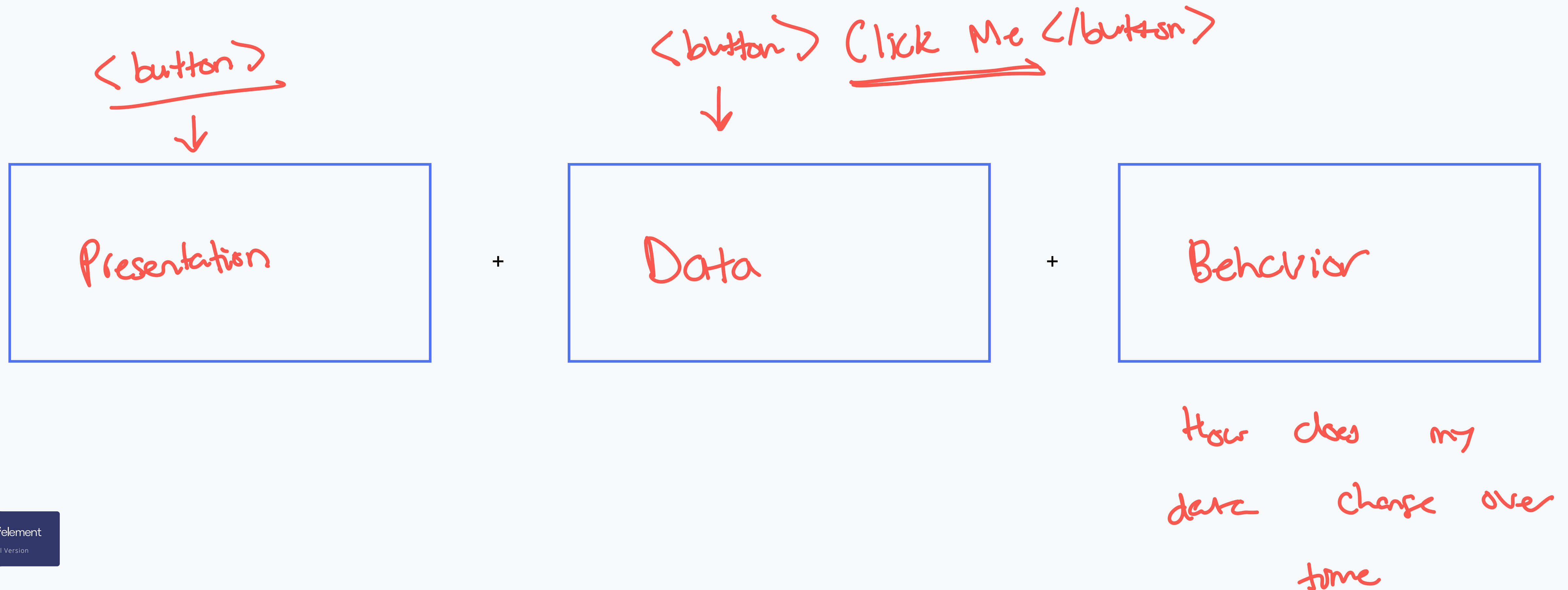
```
  itineraryItem.append(removeButton)
```

```
  //more DOM append logic, etc. etc.
```

```
}
```

# Making a user-interface is hard

Historically, managing all the code for our front-end has been a nightmare for developers....



# What if....

Front-end engineers brainstormed  
how can we make this better?  
**How can we make it more  
manageable?**

Presentation - "The Structure"

```
<button> </button>
```

BUTTON

Behavior - Data changes over time

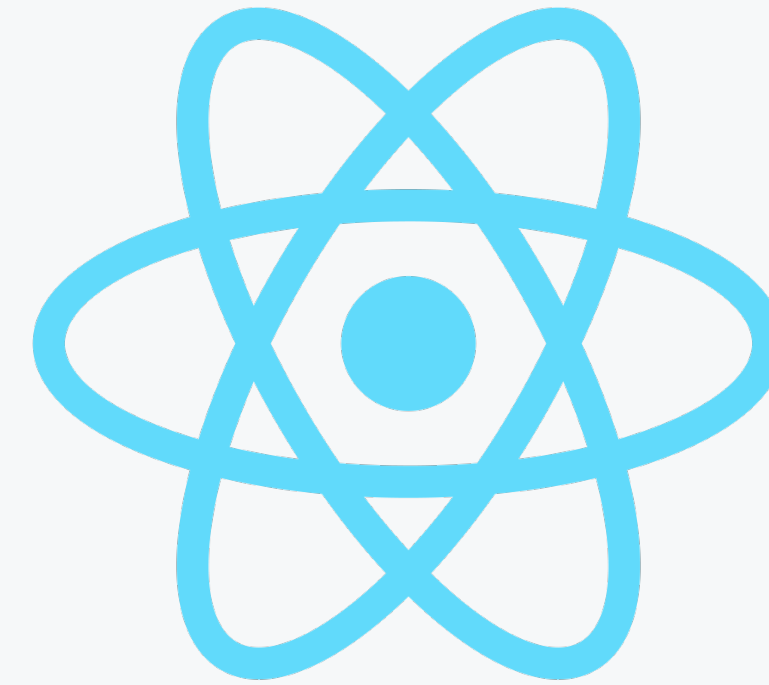
Data

```
<button> This is my Button </button>
```

```
<button> This is my Button </button>  
<button> Button changed yesterday</button>  
<button id="button" onclick="someFunction();">
```

# Introducing React.js

Our Front-End library of choice.



- Declarative (vs Imperative)
- Components (& props)
- JSX
- Component State Management (& Immutability)



# A React.js preview

Note all the things that look familiar

```
class App extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {  
      tasks: [],  
      input: ""  
    };  
  }  
  
  render() {  
    return (  
      <div>  
        <h1>Tasks</h1>  
        <ul>  
          {this.state.tasks.map((task, i) =>  
            <li key={i}>  
              {task}</li>  
          )}  
        </ul>  
        <div>  
          <input onChange={this.handleChange} value={this.state.input} />  
          <button onClick={this.addTask}>Add Task</button>  
        </div>  
      </div>  
    );  
  }  
  
  handleChange = (event) => {  
    this.setState({  
      input: event.target.value  
    });  
  }  
  
  addTask = () => {  
    this.setState(state => ({  
      tasks: [...state.tasks, state.input]  
    }));  
  }  
}
```

*Handwritten annotations:*

- JS Class** (circled) → **Subclass**
- JS** (next to constructor)
- function** (with arrow pointing to render)
- HTML** (next to render return)
- no quotes** (under <h1>Tasks</h1>)
- event listener** (under onChange and onClick)



# Live Code - React.js

# Component Types

In React.js there are two ways to write a component

## Class

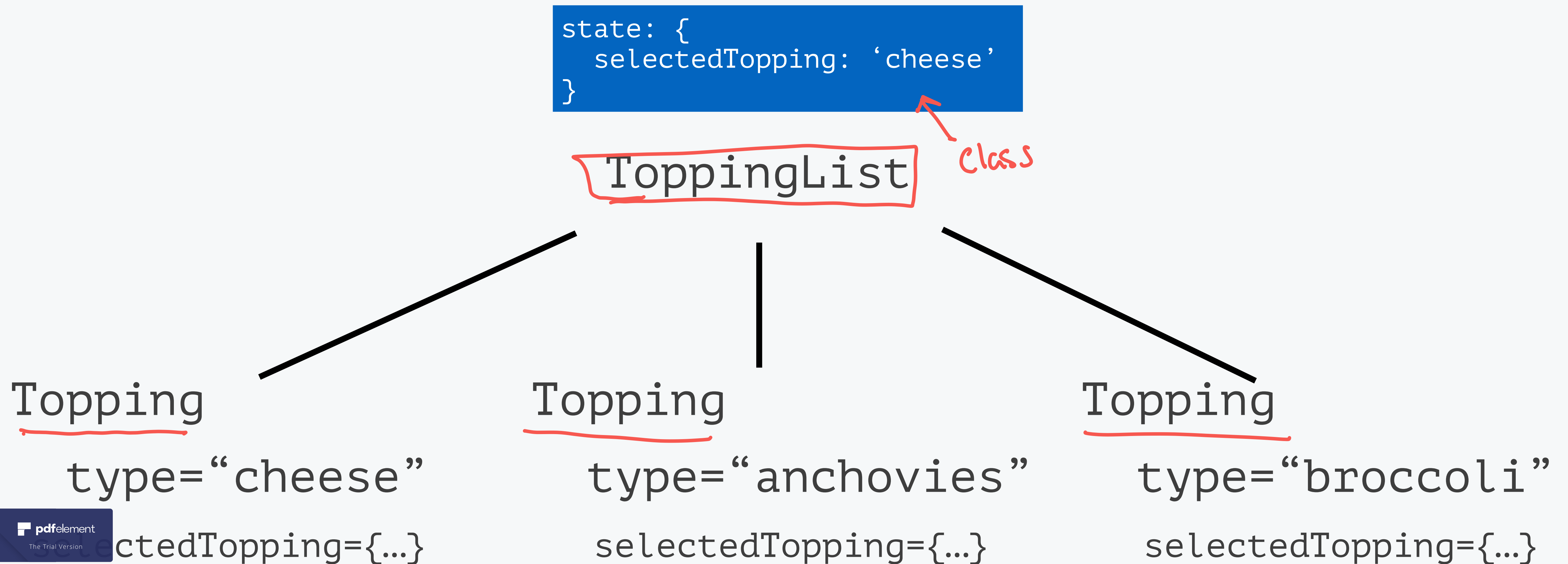
```
class Pizza extends React.Component {  
  render () {  
    return <div>Pizza Pie!</div>  
  }  
}
```

## Functional

```
const Pizza = () => {  
  return <div>Pizza Pie!</div>  
}
```

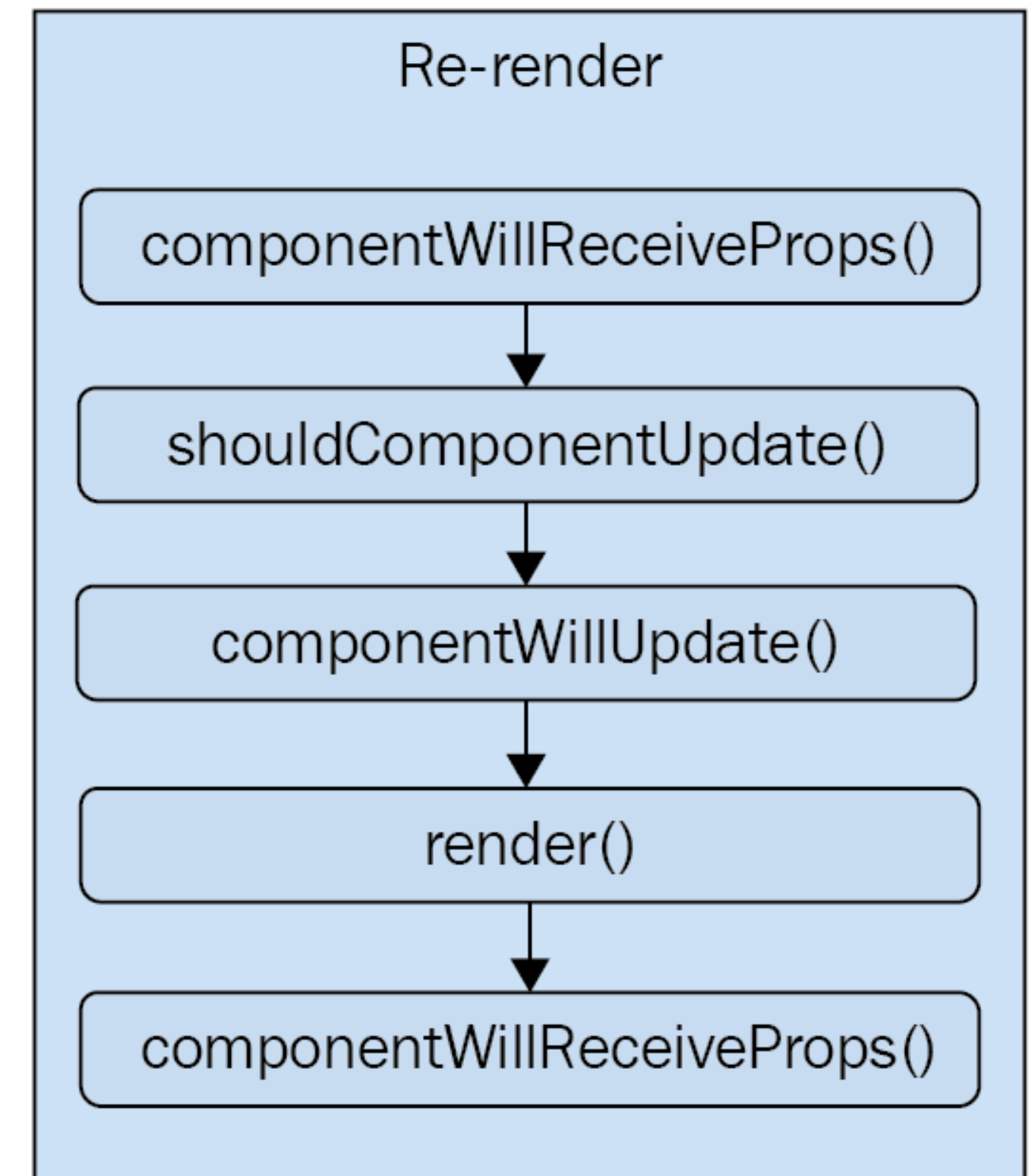
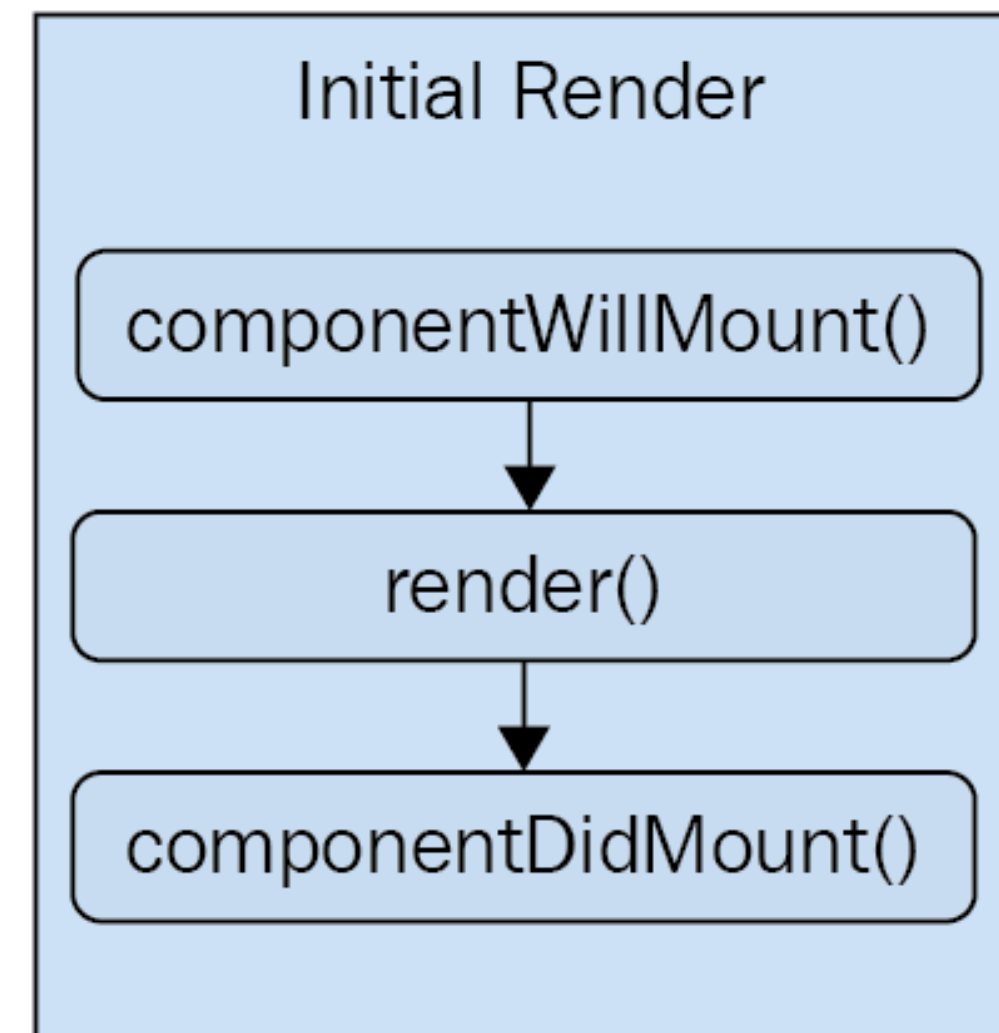
# State & Props

State and props of components always take a **uni-directional data flow**. They \*almost\* always flow down



# Component LifeCycle

When we render a component, React components go through several stages in addition to the “render” stage



Note: `setState` causes a re-render of the component

# Live Code - Pizza App