SEBA Master: Web Application Engineering
5. Developing Single-Page Applications with a Specific Web Application Framework (React)

Prof. Dr. Florian Matthes, SS21

Chair of Software Engineering for Business Information Systems (sebis)
Faculty of Informatics
Technische Universität München
wwwmatthes.in.tum.de

Outline

5. Developing Single-Page Applications with a Specific Web Application Framework (React)

- Web Application Framework Comparison
- Basic Concepts of the Chosen Framework
- State Management
- Navigation & Routing
- Client-Server Communication
- Consuming RESTful Service

Motivation

Client-side web application frameworks help developers

- **Solve recurring problems**:
  - Rendering of content and layout
  - User interaction and re-rendering
  - Validation of user input
  - Error message handling
  - Localization and internationalization
  - Personalization, user & session management
  - Navigation
  - Support of different devices
- **Structure code in a scalable, extensible and reusable way**
- **Facilitate documentation**
- **Improve security**

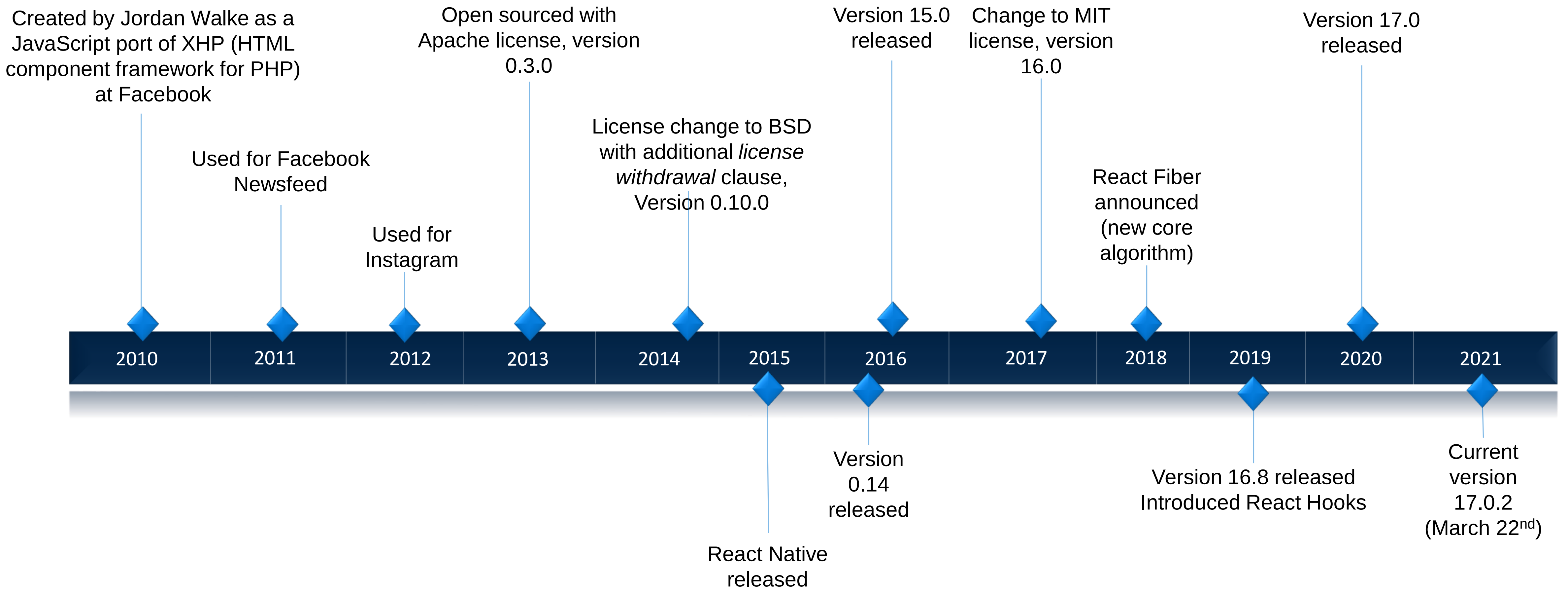
Some web application frameworks also support the implementation of recurring business requirements, for example search engine optimization (SEO) or user tracking - often through community extensions.

Early web application frameworks have a focus on structuring content and layout (HTML, CSS, JS).
Current frameworks put more emphasis on structuring behavior (state management, updates & re-rendering)

Web Application Framework Comparison for the Client-side (JavaScript)

	Angular 11	React 17	Vue 3
Application architecture	Components	**Components**	MVVM
Stability	Disruptive API changes from AngularJS to Angular	**Robust API**	Robust API
Community	Divided between AngularJS and Angular	**Active**	Active
Backed by company	Google	**Facebook**	-
Support for native applications (iOS, Android)	NativeScript	**React Native**	NativeScript
Server-side rendering	supported	**supported**	supported

History of React


Created by Jordan Walke as a JavaScript port of XHP (HTML component framework for PHP) at Facebook
Used for Facebook Newsfeed
Used for Instagram
Open sourced with Apache license, version 0.3.0
License change to BSD with additional *license withdrawal* clause, Version 0.10.0
React Native released
Version 0.14 released
Version 15.0 released
Change to MIT license, version 16.0
React Fiber announced (new core algorithm)
Version 16.8 released Introduced React Hooks
Version 17.0 released
Current version 17.0.2 (March 22nd)
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021

Introduction to React

- Open-source client-side JavaScript library
- First stable release in July 2013
- Current stable version: 17.0.2 (17+ should be used in assignments)
- Currently, one of the most popular JavaScript frameworks besides Angular and VueJS
- Used by many companies, e.g. Facebook, Netflix, AirBnB, Twitter, PayPal, Reddit, …
- Note: The **concept of a component-based architecture is independent of a concrete framework, e.g. React or Angular**.

Motivation for the development of React

- Support maintainability and scalability of large-scale web applications
- Focus on user interface aspects only (no commitment to specific client/server communication)
- Strictly component-based frontend applications, state is managed in components
- VirtualDOM rendering mechanism and using functional programming concepts improves testability
- A vivid developer ecosystem (e.g. providing the powerful create-react-app node module that allows live-editing and benefits from the NodeJS ecosystem)

React Basic Concepts and Terminology

Concept	Description
Component	A component is a logical encapsulation of (potentially complex) user interface elements that can be reused and nested.
VirtualDOM	The VirtualDOM is an abstraction layer of the the DOM used by React to minimize the (costly) operations on the document object model (DOM) using DOM Diffing.
State	State describes data objects that are managed within a component. The data objects are mostly altered by user actions.
Props	Props are immutable state objects received from a parent component. A component composes and sends props to its child components.
JSX	An extension to ECMAScript that enables the definition of XML/HTML in ECMAScript. It requires a preprocessor (transpiler) to transform JSX to JavaScript, which also allows to evaluate JavaScript expressions during the transformation process.
Router	Functionality provided by libraries/frameworks to facilitate navigation on a website, here in particular among pages of a single-page web application.

Simple "Hello World" Example

```
<html>
<head>
    <script src="../js/react.min.js"></script>
    <script src="../js/react-dom.min.js"></script>
    <script src="../js/babel.min.js"></script>
</head>
<body>
    <div id="root"></div>
    <script type="text/babel">
        class Hello extends React.Component {
            constructor() {
                super()
            }

            render() {
                return (
                    <h1 className={'title'}>
                        Hello {this.props.name}
                    </h1>
            )}
        }

        ReactDOM.render(
                        <Hello name='World' />,
              document.getElementById('root')
        )
    </script>
</body>
</html>
```

https://codepen.io/sebischair/pen/aYrGoq

Simple "Hello World" Example

1. The React library must be loaded

```html
<html>
<head>
    <script src="../js/react.min.js"></script>
    <script src="../js/react-dom.min.js"></script>
    <script src="../js/babel.min.js"></script>
</head>
<body>
    <div id="root"></div>
    <script type="text/babel">
        class Hello extends React.Component {
            constructor() {
                super()
            }

            render() {
                return (
                    <h1 className={'title'}>
                        Hello {this.props.name}
                    </h1>
            )}
        }

        ReactDOM.render(
                        <Hello name='World' />,
              document.getElementById('root')
        )
    </script>
</body>
</html>
```

https://codepen.io/sebischair/pen/aYrGoq

Simple "Hello World" Example

1. The React library must be loaded
2. The React DOM renderer must be loaded

```
<html>
<head>
    <script src="../js/react.min.js"></script>
    <script src="../js/react-dom.min.js"></script>
    <script src="../js/babel.min.js"></script>
</head>
<body>
    <div id="root"></div>
    <script type="text/babel">
        class Hello extends React.Component {
            constructor() {
                super()
            }

            render() {
                return (
                    <h1 className={'title'}>
                        Hello {this.props.name}
                    </h1>
            )}
        }

        ReactDOM.render(
                        <Hello name='World' />,
              document.getElementById('root')
        )
    </script>
</body>
</html>
```

https://codepen.io/sebischair/pen/aYrGoq

Simple "Hello World" Example

1. The React library must be loaded
2. The React DOM renderer must be loaded
3. The Babel transpiler must be loaded

```html
<html>
<head>
    <script src="../js/react.min.js"></script>
    <script src="../js/react-dom.min.js"></script>
    <script src="../js/babel.min.js"></script>
</head>
<body>
    <div id="root"></div>
    <script type="text/babel">
        class Hello extends React.Component {
            constructor() {
                super()
            }

            render() {
                return (
                    <h1 className={'title'}>
                        Hello {this.props.name}
                    </h1>
            )}
        }

        ReactDOM.render(
                        <Hello name='World' />,
                document.getElementById('root')
        )
    </script>
</body>
</html>
```

https://codepen.io/sebischair/pen/aYrGog

Simple "Hello World" Example

1. The React library must be loaded
2. The React DOM renderer must be loaded
3. The Babel transpiler must be loaded
4. Define the application container React will render to

```
<html>
<head>
    <script src="../js/react.min.js"></script>
    <script src="../js/react-dom.min.js"></script>
    <script src="../js/babel.min.js"></script>
</head>
<body>
    <div id="root"></div>
    <script type="text/babel">
        class Hello extends React.Component {
            constructor() {
                super()
            }

            render() {
                return (
                    <h1 className={'title'}>
                        Hello {this.props.name}
                    </h1>
            )}
        }

        ReactDOM.render(
                        <Hello name='World' />,
              document.getElementById('root')
        )
    </script>
</body>
</html>
```

https://codepen.io/sebischair/pen/aYrGoq

Simple "Hello World" Example

1. The React library must be loaded
2. The React DOM renderer must be loaded
3. The Babel transpiler must be loaded
4. Define the application container React will render to
5. Invoke the Babel transpiler

```html
<html>
<head>
    <script src="../js/react.min.js"></script>
    <script src="../js/react-dom.min.js"></script>
    <script src="../js/babel.min.js"></script>
</head>
<body>
    <div id="root"></div>
    <script type="text/babel">
        class Hello extends React.Component {
            constructor() {
                super()
            }

            render() {
                return (
                    <h1 className={'title'}>
                        Hello {this.props.name}
                    </h1>
            )}
        }

        ReactDOM.render(
                        <Hello name='World' />,
              document.getElementById('root')
        )
    </script>
</body>
</html>
```

https://codepen.io/sebischair/pen/aYrGoq

Simple "Hello World" Example

1. The React library must be loaded
2. The React DOM renderer must be loaded
3. The Babel transpiler must be loaded
4. Define the application container React will render to
5. Invoke the Babel transpiler
6. Create a new React component (ES 6 classes should have a constructor method)

```
<html>
<head>
    <script src="../js/react.min.js"></script>
    <script src="../js/react-dom.min.js"></script>
    <script src="../js/babel.min.js"></script>
</head>
<body>
    <div id="root"></div>
    <script type="text/babel">
        class Hello extends React.Component {
            constructor() {
                super()
            }

            render() {
                return (
                    <h1 className={'title'}>
                        Hello {this.props.name}
                    </h1>
            )}
        }

        ReactDOM.render(
                        <Hello name='World' />,
              document.getElementById('root')
        )
    </script>
</body>
</html>
```

https://codepen.io/sebischair/pen/aYrGoq

Simple "Hello World" Example

1. The React library must be loaded
2. The React DOM renderer must be loaded
3. The Babel transpiler must be loaded
4. Define the application container React will render to
5. Invoke the Babel transpiler
6. Create a new React component (ES 6 classes should have a constructor method)
7. Overwrite the default render method that returns the component‘s content to display

```html
<html>
<head>
    <script src="../js/react.min.js"></script>
    <script src="../js/react-dom.min.js"></script>
    <script src="../js/babel.min.js"></script>
</head>
<body>
    <div id="root"></div>
    <script type="text/babel">
        class Hello extends React.Component {
            constructor() {
                super()
            }

            render() {
                return (
                    <h1 className={'title'}>
                        Hello {this.props.name}
                    </h1>
            )}
        }

        ReactDOM.render(
                        <Hello name='World' />,
              document.getElementById('root')
        )
    </script>
</body>
</html>
```

https://codepen.io/sebischair/pen/aYrGoq

Simple "Hello World" Example

1. The React library must be loaded
2. The React DOM renderer must be loaded
3. The Babel transpiler must be loaded
4. Define the application container React will render to
5. Invoke the Babel transpiler
6. Create a new React component (ES 6 classes should have a constructor method)
7. Overwrite the default render method that returns the component's content to display
8. Define the content in JSX, note the “className“ syntax

```
<html>
<head>
    <script src="../js/react.min.js"></script>
    <script src="../js/react-dom.min.js"></script>
    <script src="../js/babel.min.js"></script>
</head>
<body>
    <div id="root"></div>
    <script type="text/babel">
        class Hello extends React.Component {
            constructor() {
                super()
            }

            render() {
                return (
                    <h1 className={'title'}>
                        Hello {this.props.name}
                    </h1>
            )}
        }

        ReactDOM.render(
                        <Hello name='World' />,
              document.getElementById('root')
        )
    </script>
</body>
</html>
```

https://codepen.io/sebischair/pen/aYrGoq

Simple "Hello World" Example

1. The React library must be loaded
2. The React DOM renderer must be loaded
3. The Babel transpiler must be loaded
4. Define the application container React will render to
5. Invoke the Babel transpiler
6. Create a new React component (ES 6 classes should have a constructor method)
7. Overwrite the default render method that returns the component‘s content to display
8. Define the content in JSX, note the “className“ syntax
9. Access the components state, in particular a variable name using JavaScript expressions

```
<html>
<head>
    <script src="../js/react.min.js"></script>
    <script src="../js/react-dom.min.js"></script>
    <script src="../js/babel.min.js"></script>
</head>
<body>
    <div id="root"></div>
    <script type="text/babel">
        class Hello extends React.Component {
            constructor() {
                super()
            }

            render() {
                return (
                    <h1 className={'title'}>
                        Hello {this.props.name}
                    </h1>
            )}
        }

        ReactDOM.render(
                        <Hello name='World' />,
              document.getElementById('root')
        )
    </script>
</body>
</html>
```

https://codepen.io/sebischair/pen/aYrGoq

Simple "Hello World" Example

1. The React library must be loaded
2. The React DOM renderer must be loaded
3. The Babel transpiler must be loaded
4. Define the application container React will render to
5. Invoke the Babel transpiler
6. Create a new React component (ES 6 classes should have a constructor method)
7. Overwrite the default render method that returns the component‘s content to display
8. Define the content in JSX, note the “className“ syntax
9. Access the components state, in particular a variable name using JavaScript expressions
10. Invoke React to render the component „Hello“ to the div with id=‘root’

```
<html>
<head>
    <script src="../js/react.min.js"></script>
    <script src="../js/react-dom.min.js"></script>
    <script src="../js/babel.min.js"></script>
</head>
<body>
    <div id="root"></div>
    <script type="text/babel">
        class Hello extends React.Component {
            constructor() {
                super()
            }

            render() {
                return (
                    <h1 className={'title'}>
                        Hello {this.props.name}
                    </h1>
            )}
        }

        ReactDOM.render(
                        <Hello name='World' />,
              document.getElementById('root')
        )
    </script>
</body>
</html>
```

https://codepen.io/sebischair/pen/aYrGoq

Simple "Hello World" Example

1. The React library must be loaded
2. The React DOM renderer must be loaded
3. The Babel transpiler must be loaded
4. Define the application container React will render to
5. Invoke the Babel transpiler
6. Create a new React component (ES 6 classes should have a constructor method)
7. Overwrite the default render method that returns the component‘s content to display
8. Define the content in JSX, note the “className“ syntax
9. Access the components state, in particular a variable name using JavaScript expressions
10. Invoke React to render the component „Hello“ to the div with id=‘root’
11. Once defined, components can be reused multiple times.

```
<html>
<head>
    <script src="../js/react.min.js"></script>
    <script src="../js/react-dom.min.js"></script>
    <script src="../js/babel.min.js"></script>
</head>
<body>
    <div id="root"></div>
    <script type="text/babel">
        class Hello extends React.Component {
            constructor() {
                super()
            }

            render() {
                return (
                    <h1 className={'title'}>
                        Hello {this.props.name}
                    </h1>
            )}
        }

        ReactDOM.render(<div><Hello name='World' />
                             <Hello name='SEBA' /></div>,
              document.getElementById('root')
        )
    </script>
</body>
</html>
```

JavaScript XML (JSX)

JSX

- is an XML-like extension to ECMAScript without any defined semantics
- requires a transpiler to convert JSX to JavaScript
- allows to embed HTML code in Javascript:

```
return (<h1>Hello World</h1>)
```

- requires a single root element, i.e. the following example does not work:

```
return (<h1>Hello World</h1><h1>Hello universe</h1>)
```

- requires developers to declare standard HTML attributes in a JSX-specific syntax:

```
return (<h1 className='title'>Hello World</h1>)
```

- With React, ES6 template literals are often used in JSX to access state and props variables:

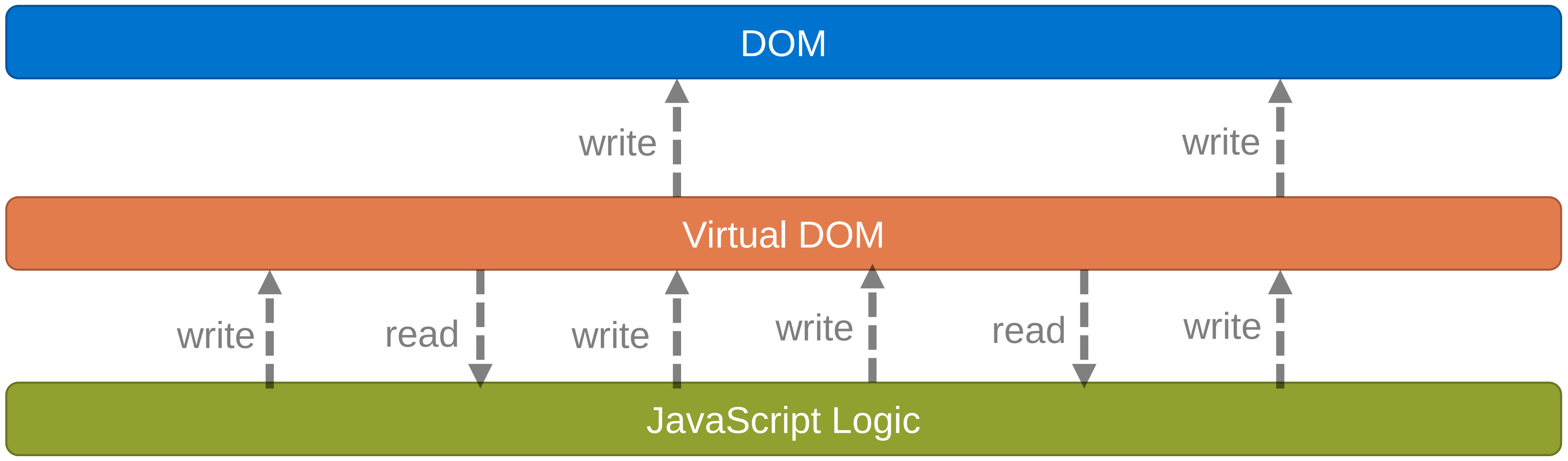
```
return (<h1>`Hello ${this.props.name} `</h1>)
```

- The JSX spread operator addresses all properties of an object with arbitrary many properties:

```
let props = {foo: 'x', bar: 'y'}; let c = <Component {...props} />;
console.log(c.foo); /* 'x' */ console.log(c.bar); /* 'y' */
```

http://facebook.github.io/jsx/
https://reactjs.org/docs/jsx-in-depth.html

Virtual DOM



- A general rule of thumb is that DOM operations are costly, especially in large applications
- React creates a lightweight in-memory data structure (Virtual DOM) that represents the DOM that is used to minimize the number of DOM operations:
  1. Eliminate read operations on the DOM
  2. Update only elements that actually changed (DOM Diffing)
- An element (for example a component) can be added to the Virtual DOM with the React.createElement(type, [props], …) method
- If ES6 classes are used to create components, createElement methods are called automatically

DOM Diffing

- The Virtual DOM is, like the DOM, a tree structure
- React minimizes re-render/write operations to the DOM by identifying changed elements in the VirtualDOM
- Therefore, an old Virtual DOM tree is compared to a Virtual DOM tree where all updates have been applied
- Comparing two trees has a complexity of $O(N^3)$, where N is the number of nodes of the larger of the two trees.

- React‘s diffing algorithm implements a heuristic that recursively identifies changed elements in O(N)
- The diffing algorithm recursively walks over the trees top-down and identifies changed components.
- For a particular component, React differentiates between changes of component type, attributes, style and state and identifies the minimal necessary updates to the DOM
- The concept of immutable state objects helps the Diffing algorithm to efficiently detect changed states
- Developers can optimize the heuristic by manually specifying if an element has or has not to be updated (e.g. defining shouldComponentUpdate())

https://reactjs.org/docs/reconciliation.html

React Components

What is a React Component?

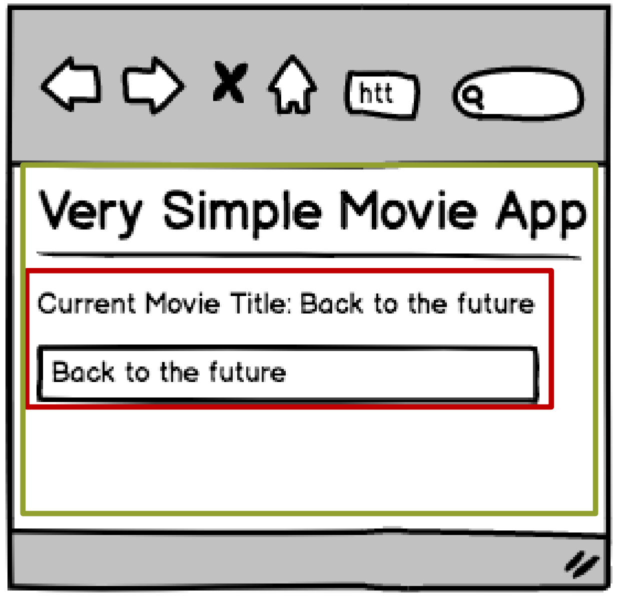
- React components are used to split the UI into reusable pieces.
- Components can be nested, i.e. they are organized in a hierarchy
- Components encapsulate state
- Components are the core concept of a component-based web application architecture
- See the official documentation: https://reactjs.org/docs/react-component.html

React Components & State Management Problems (1)

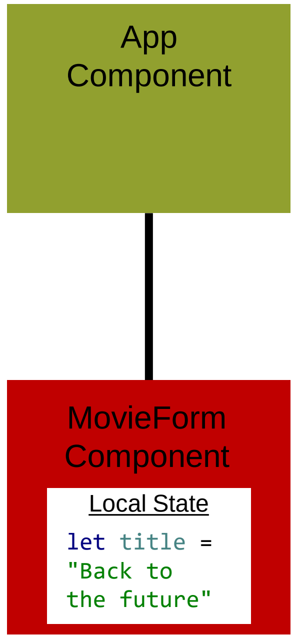
- The "Very Simple Movie App" contains a simple input field where a user can change the title of a single movie. The Very Simple Movie App displays the title as currently entered in the input field.

- From the mockups we identify two components: The App Component that represents the full application. It contains a MovieForm Component that handles the form.

View:



Component Hierarchy:



```
let title =
"Back to
the future"
```

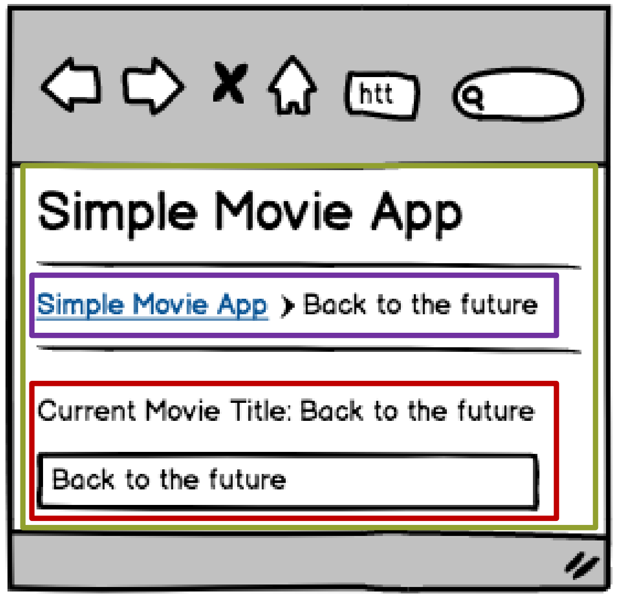
State:
In this simple case, the MovieForm component encapsulates its own state and does not depend on other components.

https://codepen.io/sebischair/pen/dmEePg
https://reactjs.org/docs/thinking-in-react.html

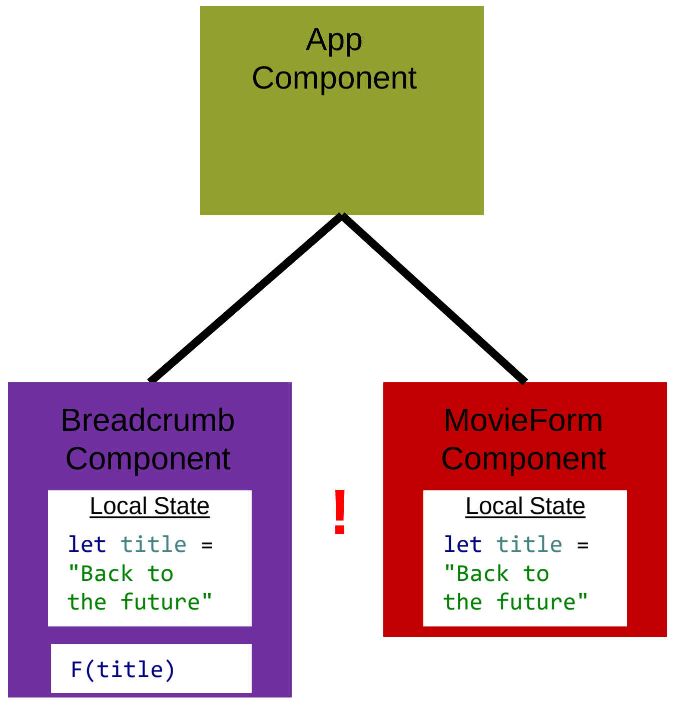
React Components & State Management Problems (2)

- In contrast to the previous "Very Simple Movie App", the "Simple Movie App" additionally contains a breadcrumb element that instantly changes with the user input in the MovieForm component.
- We identify the visual breadcrumb element as another component: the Breadcrumb Component.

View:


htt
Simple Movie App
Simple Movie App › Back to the future
Current Movie Title: Back to the future
Back to the future

Component Hierarchy:



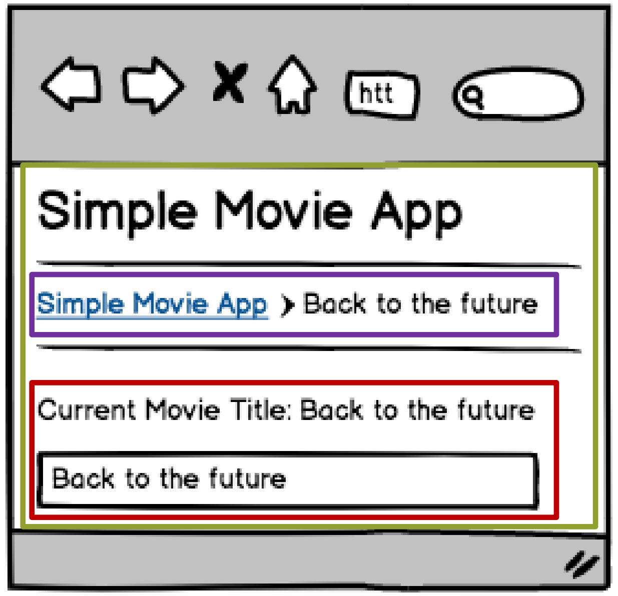
State:
In this more complex case, a component (or DOM element) depends on the state of another component: Whenever the Breadcrumb Component performs an operation F(title) on the title variable (e.g. to display the variable's content), it requires the current value of the title variable in the MovieForm Component.

https://codepen.io/sebischair/pen/PRvePY

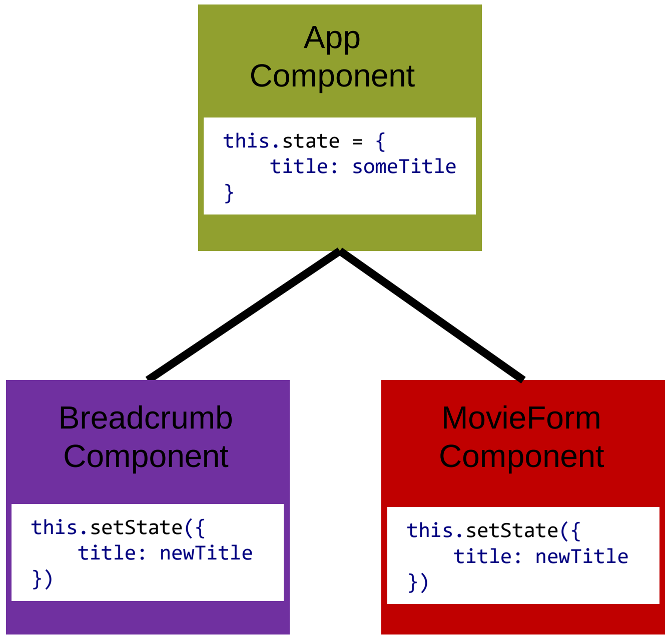
React Solution to State Management Problems (1)

- Components encapsulate state
- The state is stored in immutable JavaScript objects
- Similar to the functional programming paradigm, only new/copies of state objects are created
- A component’s state is assigned once in the constructor of a component and state updates are carried out via the setState(newStateObject) method of React components that gets passed a new state object as a parameter rather than manipulating the state object directly
- State is strictly passed down in the component hierarchy

View:



Component Hierarchy:



```
this.state = {
    title: someTitle
}
```

```
this.setState({
    title: newTitle
})
```

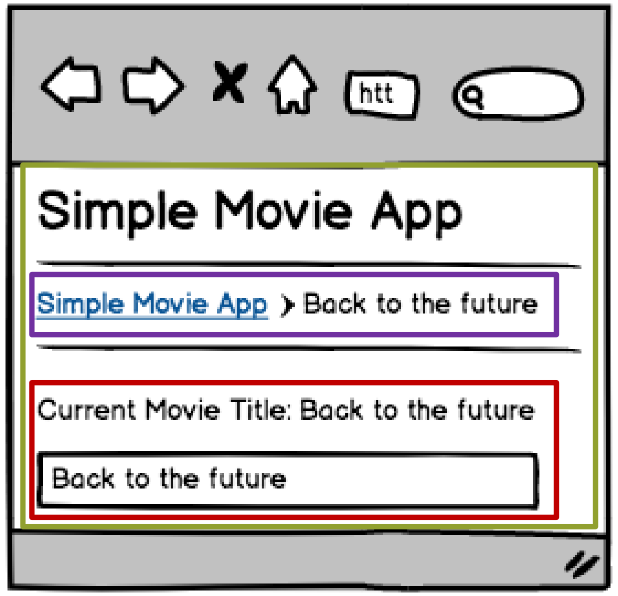
```
this.setState({
    title: newTitle
})
```

https://codepen.io/sebischair/pen/PRvePY

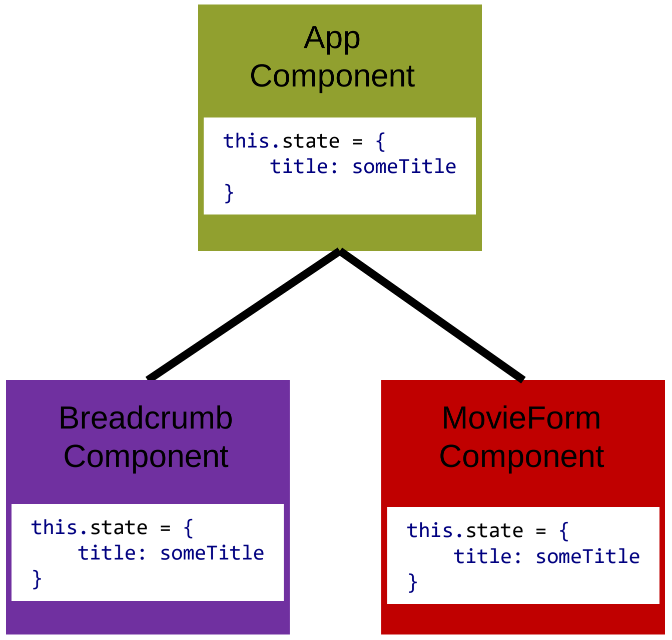
React Solution to State Management Problems (2)

- A core concept of React is to **handle state in the closest common ancestor component**, when components depend on the state of other components.
- When the Breadcrumb component is added to the ”Very Simple Movie App” example, the App Component can be identified as the closest common ancestor component of the MovieForm and the Breadcrumb Components.
- Therefore, the title state should be handled in the App Component.
- In React terminology this process is called “lifting state up”.
- This requires a few changes in all three components that we will go through step by step in the following slides together with a selection of React concepts.

View:


htt
Simple Movie App
Simple Movie App › Back to the future
Current Movie Title: Back to the future
Back to the future

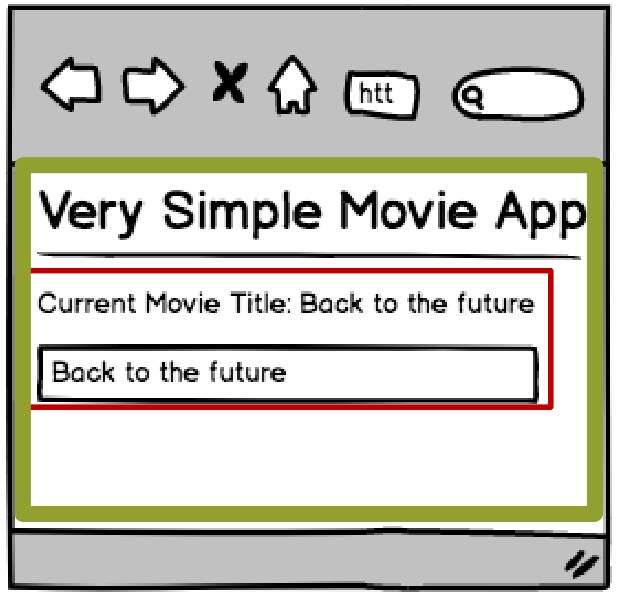
Component Hierarchy:



```
this.state = {
    title: someTitle
}
```

https://codepen.io/sebischair/pen/PRvePY
https://reactjs.org/docs/lifting-state-up.html

The Source Code for the Very Simple Movie App (1)



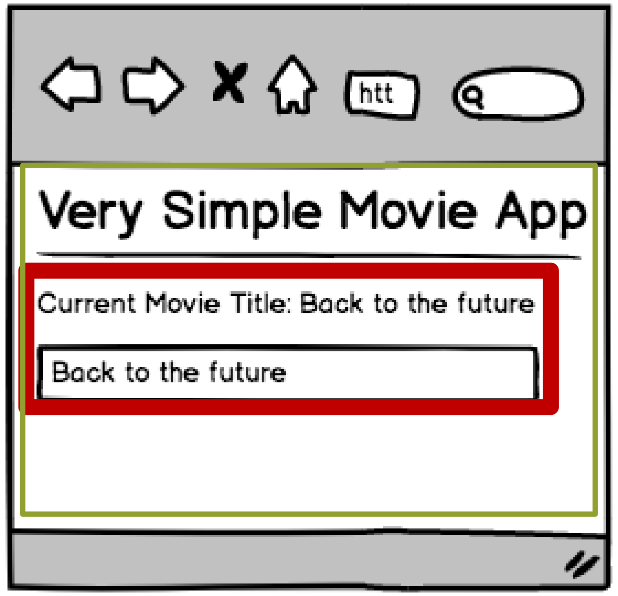
```
class App extends React.Component {
    constructor() {
        super();
    }

    render() {
        return (
            <div>
                <h1>Very Simple Movie App</h1>
                <hr/>
                <MovieForm title="Back to the Future" />
            </div>
        );
    }
}
```

- The App Component is defined as a standard ES6 class. In React this is called a **class component**.
- Information is propagated down the component hierarchy: Here, a string with a default movie title is a **parameter** to the MovieForm component.

https://codepen.io/sebischair/pen/dmEePg

The Source Code for the Very Simple Movie App (2)

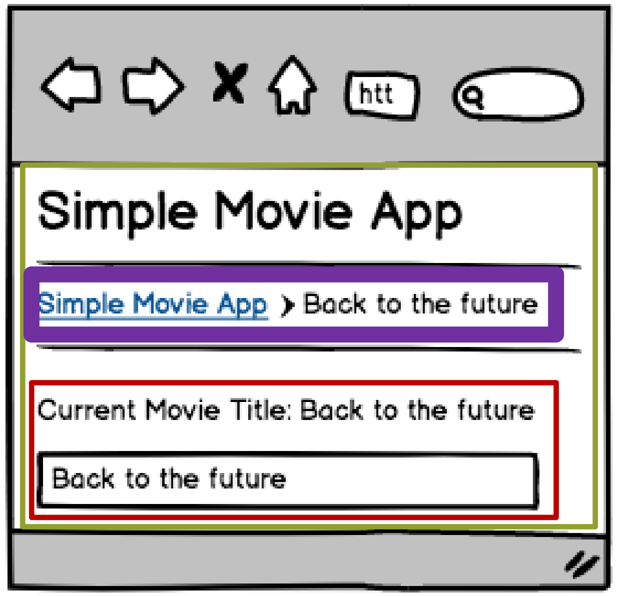


- The MovieForm Component receives information from the parent App Component in a variable typically called **props** and the local state is initialized.
- The MovieForm is a so-called **controlled component**, i.e. the form input is synchronized with the component's state. Note that the event handler function needs to be bound in the constructor

```
class MovieForm extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
            title: props.title
        };
        this.handleChange = this.handleChange.bind(this);
    }
    handleChange(event) {
        this.setState({
            title: event.target.value
        });
    }
    render() {
        return (
            <form>
                <p>
                    <b>Current movie title:</b>
                        {this.state.title}
                </p>
                <fieldset>
                    <legend>Enter movie title:</legend>
                    <input type="text"
  value={this.state.title} onChange={this.handleChange} />
                </fieldset>
            </form>
        );
}}
```

https://codepen.io/sebischair/pen/dmEePg
https://reactjs.org/docs/lifting-state-up.html

Adding a Breadcrumb Component – The Simple Movie App (1)


htt
Simple Movie App
Simple Movie App › Back to the future
Current Movie Title: Back to the future
Back to the future

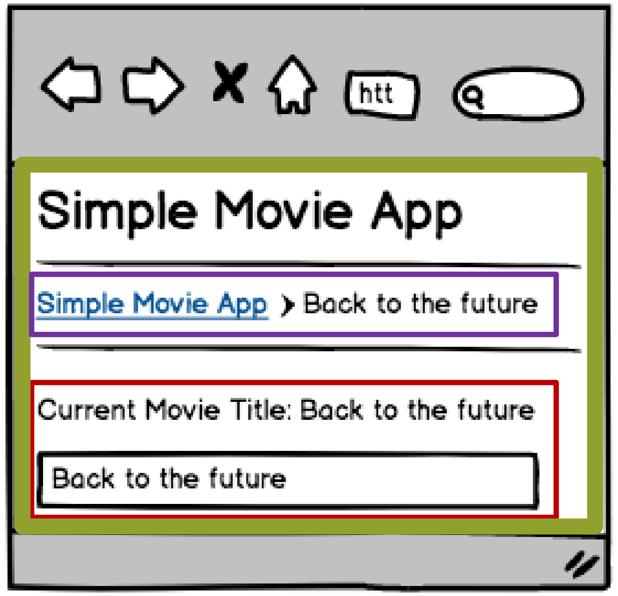
```
ReactDOM.render(
    <App title="Back to the Future" />,
    document.getElementById('root')
);
```

- First, we move the default title configuration so that the App Component receives the default title as a parameter. While not required for lifting the state up, this simplifies a structured explanation.
- We also add a Breadcrumb Component that displays the current title. It is a so-called **functional component** defined as a function.

```
function Breadcrumb(props) {
    return <span>
                <b>Simple Movie App</b>
                > {props.title}
            </span>;
}
```

https://codepen.io/sebischair/pen/PRvePY
https://reactjs.org/docs/lifting-state-up.html

Adding a Breadcrumb Component – The Simple Movie App (2)


htt
Simple Movie App
Simple Movie App › Back to the future
Current Movie Title: Back to the future
Back to the future

- The state handling of the title is moved from the MovieForm Component now to the parental App Component. It receives the default title from above.
- The handler function for state updates is moved also up to the App Component
- The Breadcrumb Component is added to the App Component and gets passed down the title
- The title now gets passed down to the MovieForm Component together with the callback handler

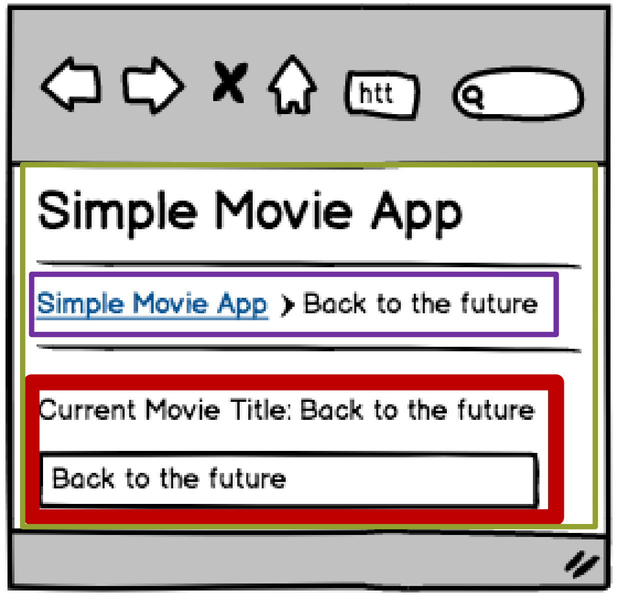
```
class App extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
            title: props.title
        };
        this.handleTitleChange =
            this.handleTitleChange.bind(this)
    }

    handleTitleChange(event) {
        this.setState({title: event.target.value});
    }

    render() {
        return (
            <div>
                <h1>Simple Movie App</h1>
                <hr/>
                <Breadcrumb title={this.state.title}/>
                <hr/>
                <MovieForm title={this.state.title}
                    onTitleChange={this.handleTitleChange}/>
            </div>
        );
    }
}
```

https://codepen.io/sebischair/pen/PRvePY
https://reactjs.org/docs/lifting-state-up.html

Adding a Breadcrumb Component – The Simple Movie App (3)


htt
Simple Movie App
Simple Movie App › Back to the future
Current Movie Title: Back to the future
Back to the future

- The MovieForm Component now receives the title from the parental App Component as props.
- Title changes are handled by the **callback handler** function received from the parental App Component.
- The title variable is now accessed via the props member variable rather than the local state member variable!

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

    handleChange(e) {
        this.props.onTitleChange(e)
    }

    render() {
        return (
            <form>
                <p>
                    <b>Current movie title:</b>
                    {this.props.title}
                </p>
                <fieldset>
                    <legend>Enter movie title:</legend>
                    <input type="text"
                        value={this.props.title}
                        onChange={this.handleChange} />
                </fieldset>
            </form>
        );
    }
}
```

https://codepen.io/sebischair/pen/PRvePY
https://reactjs.org/docs/lifting-state-up.html

Typechecking: PropTypes

- Type-checking can help to prevent bugs in larger applications
- Type-checking can be achieved by using TypeScript or React’s built-in **PropTypes**.

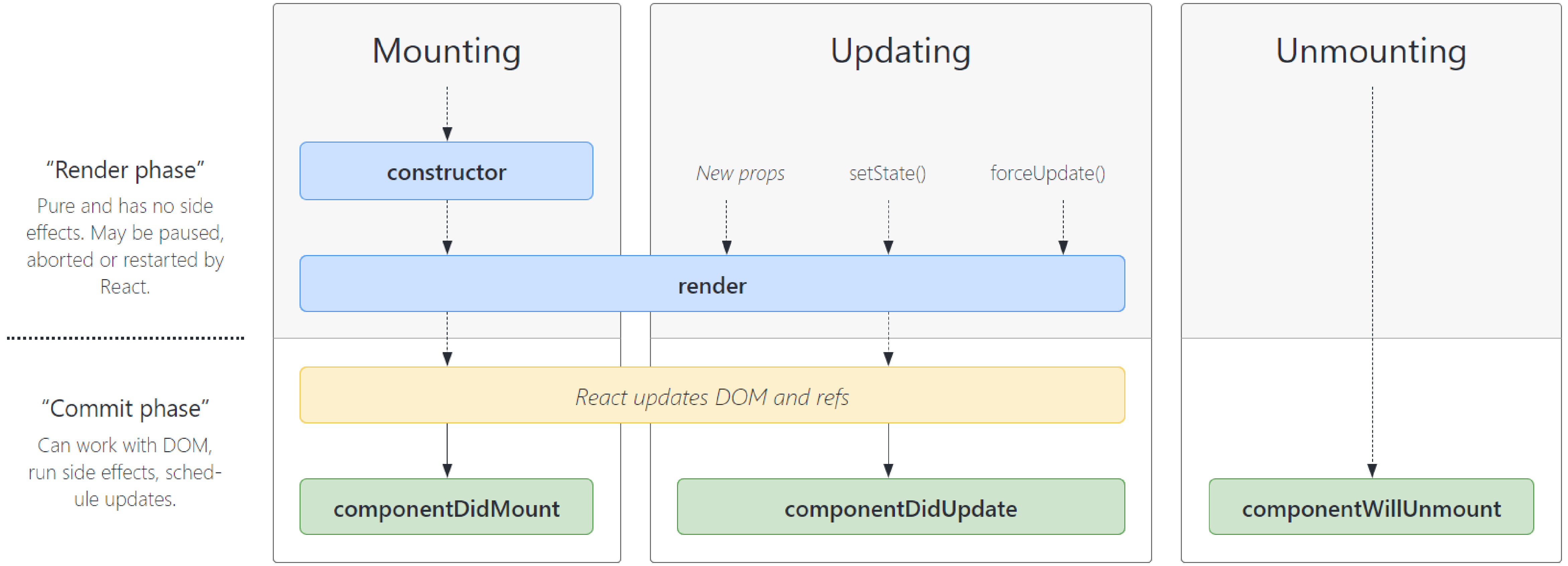
```
import PropTypes from 'prop-types';


class MovieForm extends React.Component {
    constructor(props) {
        super(props);
        …
    }
    …
    render() {
        return (
            <form>
                <p>
                    <b>Current movie title:</b>
{this.props.title}
                </p>
                …
            </form>
        );
    }
}

MovieForm.propTypes = {
    title: PropTypes.string.isRequired
};
```

https://reactjs.org/docs/typechecking-with-proptypes.html

React Component Lifecycle



Source (interactive): https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/

React Hooks

- Hooks were introduced to React with version 16.8. Naming Convention: "use" + "hook name".
- Main motivation: Use state in function components.

Example: useState hook

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

function SEBAHooksExample() {

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

  return (
    <div>
      <p>You clicked on this button {count} times.</p>
      <button onClick={() => setCount(count + 1)}> Click! </button>
    </div>
  )
}
```

always returns an array of two elements

initial value

current state

function to change the state

- Never change state directly by (re-)assigning a value to it. Always call the "setter" function.
- This will also cause React to reevaluate the content and trigger an update / rerender (if necessary).
- There are many more hooks for different use cases.

https://reactjs.org/docs/hooks-intro.html

React Navigation & Routing (1)

Classic navigation behavior of a browser

- The URL entered in the browser’s address bar reflects the current address of a website on the Internet
- The URL can be seen as the current state of the browser
- Usually, users navigate from one page to another by clicking a hyperlink
- Each subpage has its own unique link
- The browser offers controls to navigate back and forth in the history

Challenges for Single-Page Applications

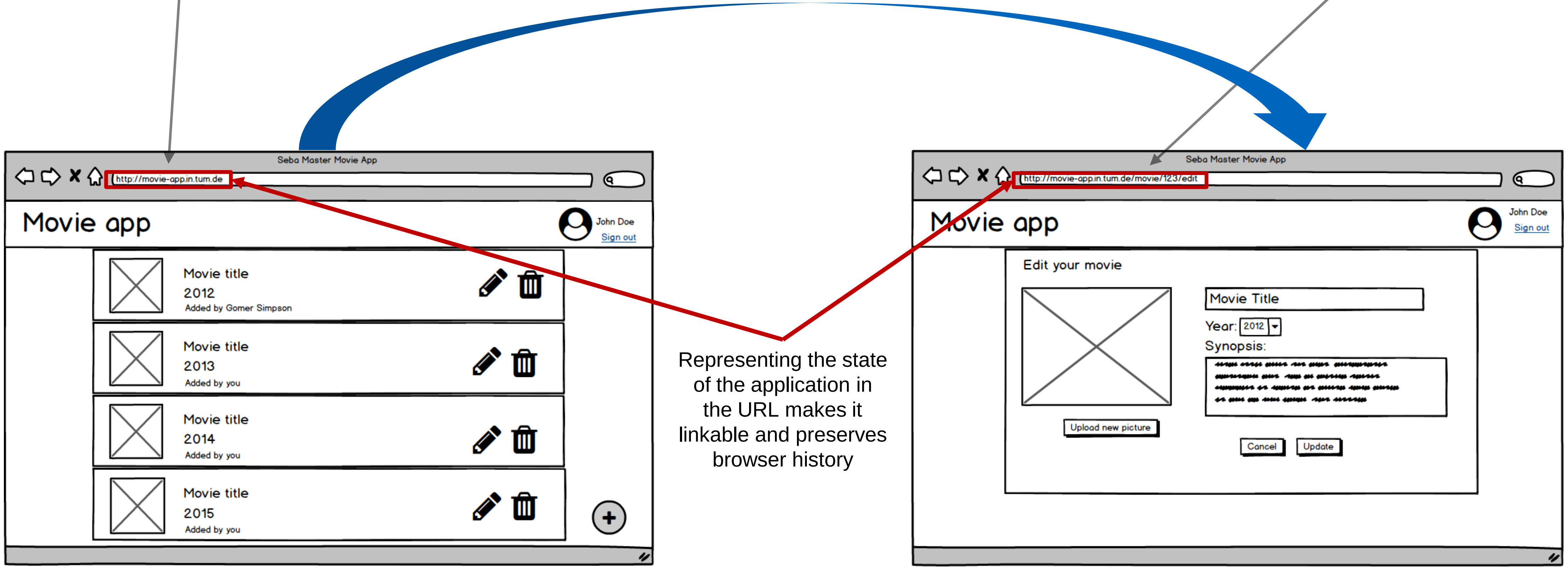
- New content is loaded via asynchronous API calls and a state change by URL is not necessary
- How to control and preserve the browser history for AJAX applications?
- How to manage states?
- How to make SPAs "linkable"?

React Navigation & Routing (2)

Root URL represents initial state of the application

User clicks on a movie item in the list and invokes a new state in the application

URL changes and represents new state of the application



Representing the state of the application in the URL makes it linkable and preserves browser history

Movie App Routing Example (Via react-router Package)

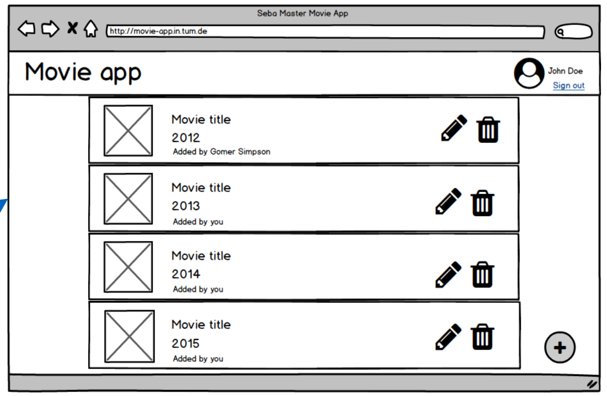
```
class App extends React.Component {
    constructor(props) {
        …
        this.state = {
            title: 'Movie Example App',
            routes: [
                {component: MovieListView, path: '/', exact: true},
                {component: MovieDetailView, path: '/show/:id‘},
                …
    ]}}
    render() {
        return (
            <div>
                <Router>
                    <Switch>
                        {this.state.routes.map((route, i) =>
                            (<Route key={i} {...route}/>))}
                    </Switch>
                </Router>
            </div>
);}}
```

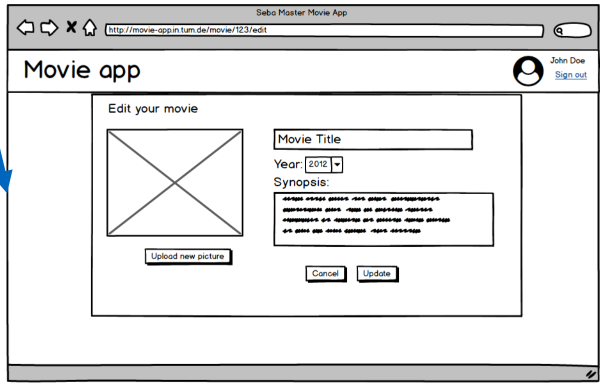
Routes are configured as a state object in the App Component

Route for MovieList View, default route

Route for Movie Detail View with parameter movie id

The render function for this App Component performs the loading of the configured components for the Routes



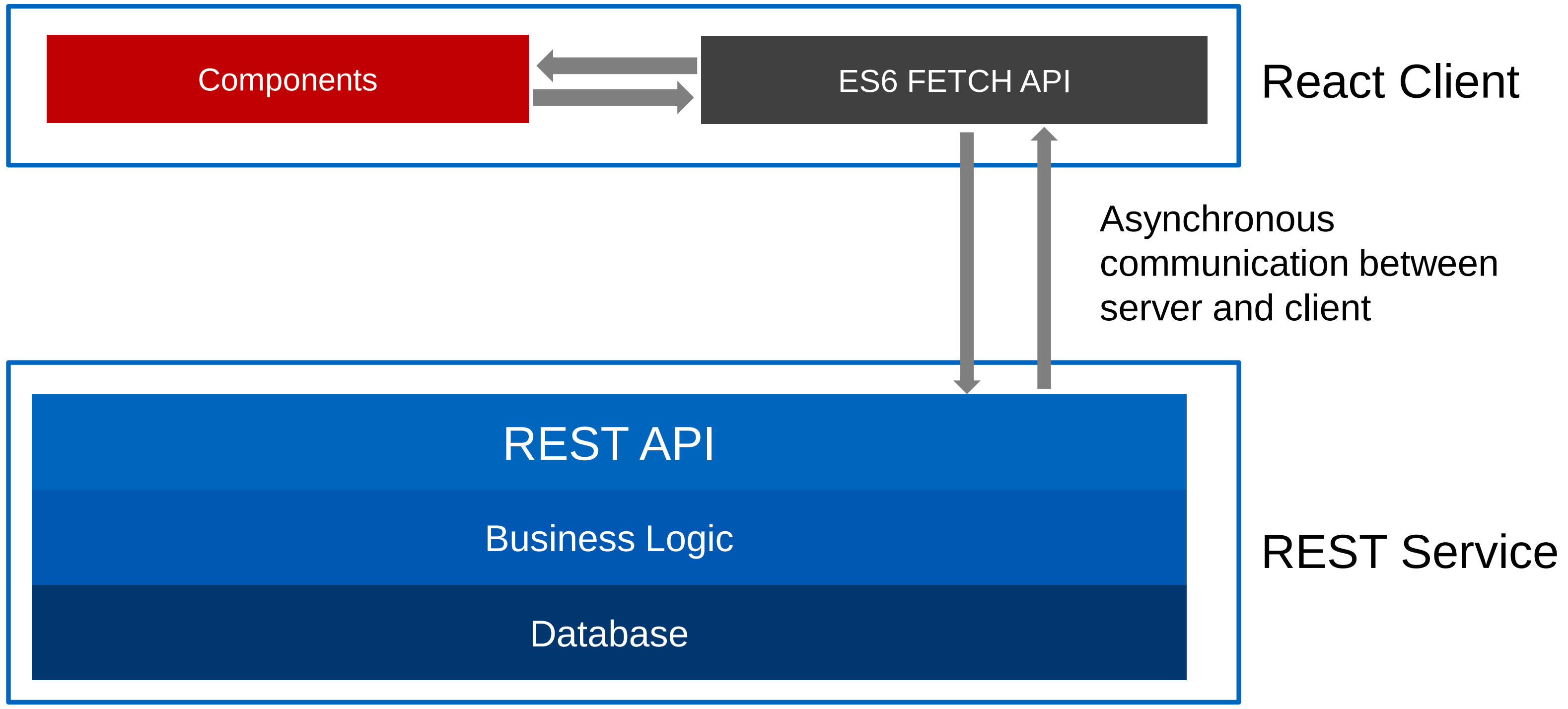


Adding links in (sub-) components

```
<Link to={'/'}>Home</Link>
<Link to= {`/show/${this.props.movie._id}`}>Link to movie {this.props.movie.title}</Link>
```

https://github.com/sebischair/sebamaster-movie-frontend

Consuming RESTful Services in React Applications and ES6



The ES6 standard provides a FETCH API that provides a global fetch() method to consume REST services.

Client-Server Communication
The ES6 FETCH API

- The FETCH API provides a standardized API to perform network requests.
- The FETCH API is supported by all modern browsers (except Internet Explorer).
- In contrast to former XMLHttpRequest calls, the fetch API uses promises (see Chapter 04).
- In contrast to the jQuery implementations of promises, server status codes need to be checked manually.

Via the fetch API requests can be configured regarding type, headers, cookies etc.

success_function:
Note that HTTP Status codes need to be checked

error_function, e.g. socket error, timeout, …

```
fetch(url, {
    method: 'GET',
    headers: header
}).then((response) => {
    if(response.ok) {
        console.log(response.json());
        /* work with data */
    }
    else {
        console.log(response.status);
        /* an error occurred */
    }
}).catch((e) => {
    console.log(e);
});
```

https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

Using Services to Consume REST Data

It is not required by React, but we recommend to encapsulate API **Services** in React to **share code and functionality** across an application.

A service is used to separate the actual data from the components.

We recommend to use the following guidelines:

- Services are only instantiated when a component depends on it (**lazy loading**)
- Services are instantiated as **singletons**

Separating Generic Services & API-specific Services

- It is a good practice to factor out code that frequently changes and to separate it from code that is stable for a long time

```
class HttpService {
    static get('http://localhost:3000/movies',
                                onSuccess, onError) {
    let header = new Headers();
    fetch(url, {
        method: 'GET',
        headers: header
    }).then((resp) => {
        if(resp.ok) {
            return resp.json();
        }
        else {
            resp.json().then((json) => {
                onError(json.error);
            });
        }
    }).then((resp) => {
            onSuccess(resp);
    }).catch((e) => {
            onError(e.message);
    });
}
```

```
import HttpService from './HttpService';

class MovieService {

    static getMovie(id) {
    return new Promise((resolve, reject) => {
        HttpService.get('http://localhost:3000/movies'/${id}`, function(data) {
            if(data != undefined || Object.keys(data).length !== 0) {
                resolve(data);
            }
            else {
                reject('Error while retrieving movie');
            }
        }, function(textStatus) {
            reject(textStatus);
        });
    });
}
```

https://github.com/sebischair/sebamaster-movie-backend