React and JavaScript frontends

GÖTEBORGS UNIVERSITET

DAT076/DIT126 Web Applications

Adam Waldenberg

JavaScript frontends

Can be run on server-side or client side (as covered in the lecture on web application design)

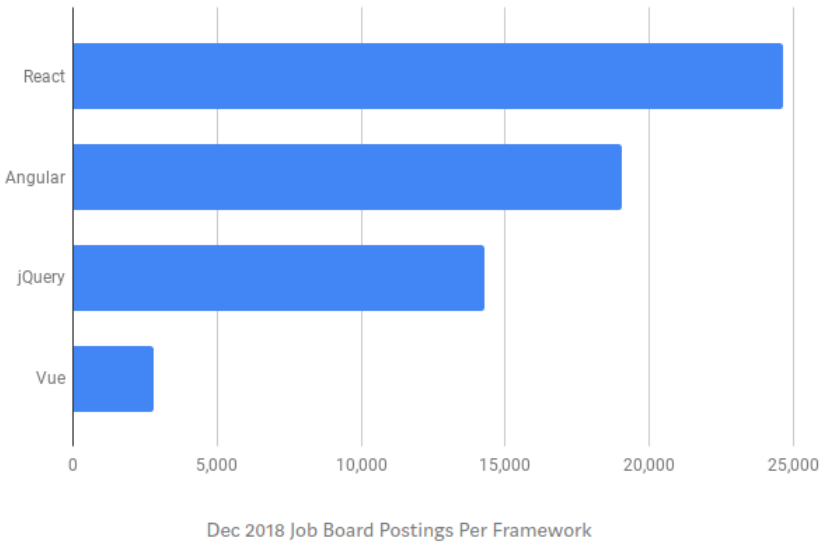
- Many JavaScript frameworks exist
- Some notable mentions include *React (Facebook)*, *Vue.js*, *Angular (Google)*, *Ember* and *Backbone.js* (with the exception of *Ember*, we have covered all of them in one way or another in past iterations of this course)
- We focus on *React* specifically because it has become very popular and has a big business backing it
- *React* is also very popular to combine with EE backends

JavaScript popularity

Compared to EE, choosing JavaScript (and often NodeJS for the backend) is especially popular with startups and smaller businesses

- Works especially well if security and stability is not critical
- The learning curve is less steep, allowing businesses to lower development costs by using cheaper developers
- Growing in popularity on a yearly basis
- Just like with Java, there is a huge infrastructure of open source community libraries, additional frameworks and extensive documentation available

JavaScript popularity continued



In the last few years, React has gained a dominant position among JavaScript frameworks

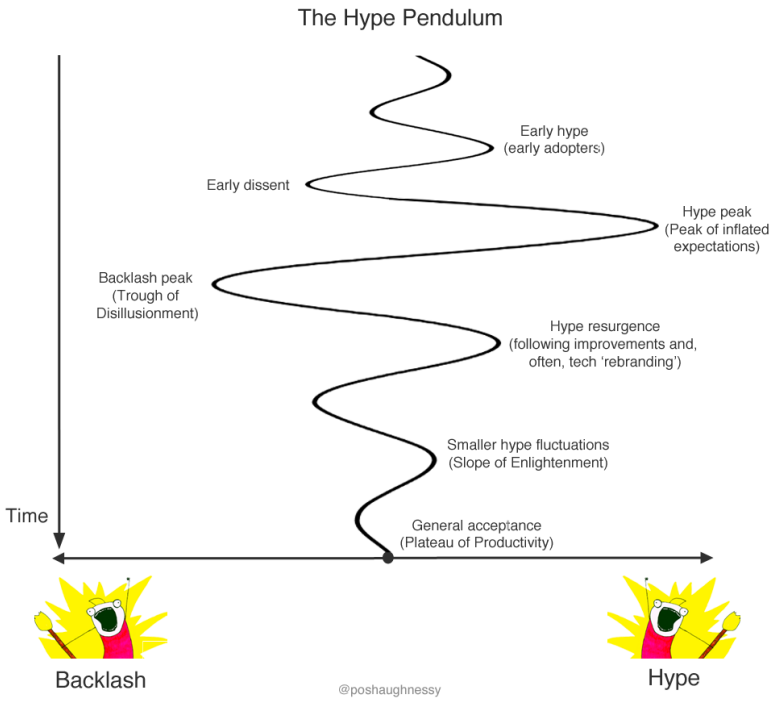
- Statistics taken from indeed.com
- At the time of making these slides - in Sweden, there are roughly 700-800 free job postings concerning React
- There are roughly 300-500 postings alone concerning web development and Java and/or EE in some way
- Many of them actually ask for both!

When to learn something new

Watch out for the constant hype in the web development sphere

- When learning something new, always consider the long-term return and if it’s even a good idea
- Previous hypes include (not in order of greatness) Ruby on rails, Perl, PHP OOP, Web 3.0, Angular 1.x, JQuery etc
- Enterprise frameworks generally allow you to take advantage of all your hard-earned learning during a longer period

The Hype Pendulum



hackernoon.com/the-hype-pendulum-of-web-development-33f500723f31

Just as relevant today as before

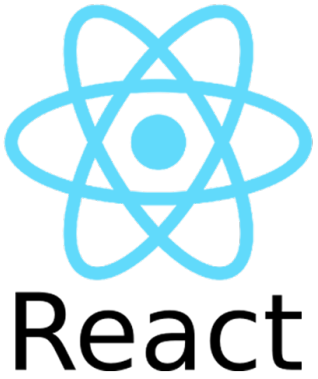
- A few years ago there was a lot of talk and hype around NodeJS
  - *"Why NodeJS is awesome and why you shouldn't even think about using it" - 2013*
    www.giantbomb.com/profile/rick/blog/why-nodejs-is-awesome-and-why-you-shouldn-t-even-t/102475/
  - *"NodeJS is cancer" - 2015*
    www.semitwist.com/mirror/node-js-is-cancer.html
- Leave the Hype Pendulum to swing and come to an informed long-term decision on your own
- This course is guilty of occasionally jumping on the hype bandwagon in previous years

The Hype Pendulum continued

JavaScript frontends, JavaScript tooling and JavaScript frameworks seem especially susceptible to hype

- There is an excessive amount of competing tooling, different forks and solutions to the same problem. Some hyped more than others, with many being utter garbage from a software engineering viewpoint
- It's the human condition. To take an analogy, Wikipedia lists over 26 types of hammers (***en.wikipedia.org/wiki/Hammer***). Do we *really* need all of them?
- Use common sense and let the hype pendulum find it's resting place
- It can take a while before you can make a informed long-term decision

Introducing React



A JavaScript library for building user interfaces

- Define views in ***.jsx*** or a ***.js*** files
- Build encapsulated components that manage their own state and compose them to make complex user interfaces

- Similar philosophy to Java Server Faces components or composite components with Facelets
- Supports client-side and server-side rendering
- Can power mobile apps using React Native
- Can be used with Electron (***www.electronjs.org***) to deploy desktop applications

A simple view definition in React

A view and component defined in pure JavaScript

```
class Hello extends React.Component {
  render() {
    return React.createElement('div', null, `Hello ${this.props.toWhat}`);
  }
}
```

```
ReactDOM.render(
  React.createElement(Hello, {toWhat: 'World'}, null),
  document.getElementById('root')
);
```

- A clean way to use it on the browser side, but not very pleasant code to read
- A React component holds properties and states
- A new or updated state causes the component to refresh

A simple view definition in React continued

A view and component defined in JSX

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

```
ReactDOM.render(
  <Hello toWhat="World" />,
  document.getElementById('root'));
```

- Much nicer code, but introduces an additional dependency on Babel (***babeljs.io***) – a transpiler to convert back to compatible JavaScript
- Browsers generally support ES6. For anything newer you need to transpile with Babel.

React and properties

React properties

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

```
ReactDOM.render(
  <Hello toWhat="World" />,
  document.getElementById('root'));
```

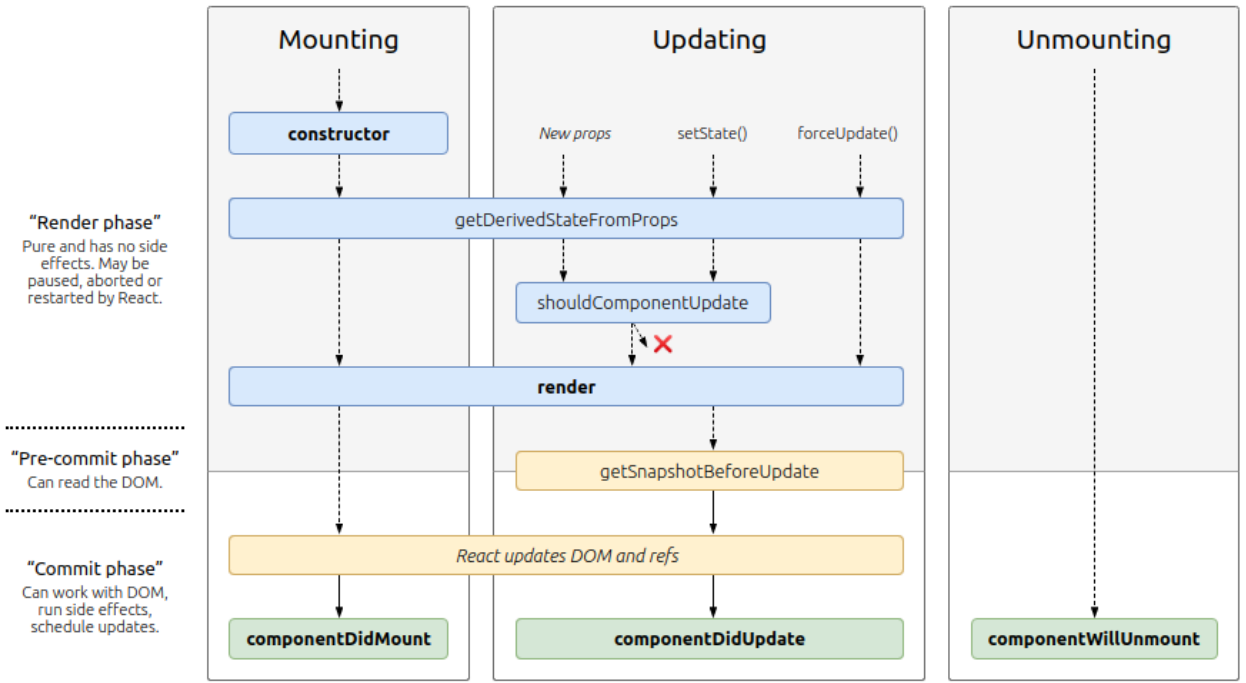
- Components in React take inputs in the form of properties (often referred to as *props*) and return elements describing what should appear on the screen
- Properties are read-only and should never be changed
- If we need to change the state of a component we use the concept of states

React states

- Each React component can hold a state
- The state represent the visual backing data that the component uses to render it’s view. The concept is similar to backing bean values in Java Server Faces
- To modify the state you call ***`this.setState({newstate})`*** on the object in question
- This triggers an update of the component, calls the life-cycle events and invokes the ***render()*** method of the component.

The life cycle of React

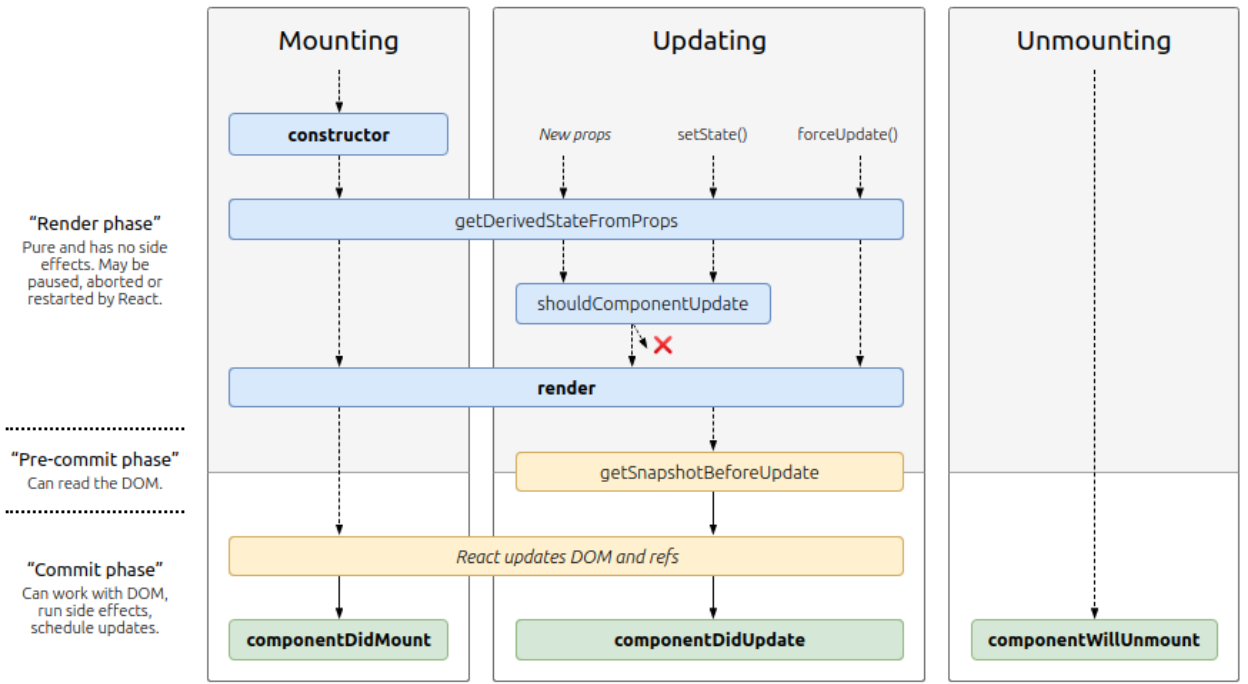
React components have a life cycle which you can exploit when you write components



getDerivedStatesFromProps()

- Invoked right before the render() method is called, both on the initial mount and on subsequent updates
- It should return an object to update the state, or null to update nothing
- A component can not update its own props - but it can update its children and their props

The life cycle of React continued



getSnapshotBeforeUpdate()

- Invoked right before the most recently rendered output is committed to the DOM. Allows you to capture some information from the DOM (e.g. scroll position) before it is potentially changed. Any value returned by this lifecycle will be passed as a parameter to ***componentDidUpdate()***

React and input events

In React, we can listen to input events and other types of events that happen - allowing us to add input handlers to handle those events

- The frameork offers so many events it’s easier to list the event categories rather than the events themselves:

Image Events

Animation Events

Transition Events

Other Events

Pointer Events

Clipboard Events

Composition Events

Keyboard Events

Focus Events

Form Events

Mouse Events

Selection Events

Touch Events

UI Events

Wheel Events

Media Events

React and input events – An example

```
onHandleResult = e => {
     this.setState({
          result: this.state.number1 + this.state.number2
     });
}
```

```
<button onClick={this.onHandleResult}>=</button>
```

- Connects to a handler that (in this case) updates the state of the component
- Will trigger a refresh and a call to ***render()***
- Not dissimilar to the way you connect method expressions to action listeners in Java Server Faces

We can easily combine React and Facelets – allowing us to use the templating engine together with JSX pages

```
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:ui="http://xmlns.jcp.org/jsf/facelets">
     <ui:composition template="WEB-INF/template/common.xhtml">
          <ui:define name="title">A Simple react page</ui:define>
          <ui:define name="content">
               const App = () => {
                    return (
                         <div>hej</div>
                    );
               };
               ReactDOM.render(<App />, document.querySelector("#root"));
          </ui:define>
     </ui:composition>
</html>
```

- In practice this is an XHTML document extended with Babel and JSX syntax
- Babel and React are included in the template

Using React client-side in an enterprise application - continued

```
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:ui="http://xmlns.jcp.org/jsf/facelets">
     <head>
          <title><ui:insert name="title"/></title>

          <script src="https://unpkg.com/react@16/umd/react.development.js"></script>
          <script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
          <script src="https://unpkg.com/babel-standalone@6/babel.min.js"></script>
          <ui:insert name="dependencies"/>
     </head>
     <body>
          <div id="root"/>
          <script data-plugins="transform-es2015-modules-umd" type="text/babel">
               <ui:insert name="content"/>
          </script>
     </body>
</html>
```

- We incude all the React and Babel dependencies
- *`data-plugins="transform-es2015-modules-umd"`* is used by Babel to handle imports on the client side without a NodeJS server

Creating a Book component

Lets create a Book component for our previously written web service

```
export default class Book extends React.Component {
     render() {
          return (
               <div>
                    <p>Title: {this.props.title}</p>
                    <p>Author: {this.props.author}</p>
               </div>
          );
     }
}
```

- We reference two properties which are defined with ***<Book title="a" author="b" />*** when the parent view and it's element are created
- We extend from the React.Component class using ordinary ES6 syntax

Using the Book component on a page

To use the component, we import it and use it as a usual React component or HTML element

```
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:ui="http://xmlns.jcp.org/jsf/facelets">
     <ui:composition template="WEB-INF/template/common.xhtml">
          <ui:define name="title">A Simple react page</ui:define>
          <ui:define name="dependencies">
               <script data-plugins="transform-es2015-modules-umd" type="text/babel" src="./components/Book.jsx"></script>
          </ui:define>
          <ui:define name="content">
               import Book from "./components/Book";

               const App = () => {
                    return (<Book author="Carl Sagan" title="Contact"/>);
               };
               ReactDOM.render(<App />, document.querySelector("#root"));
          </ui:define>
     </ui:composition>
</html>
```

- We define the Book dependency and import it in our code
- The rest works like any other component

Defining a Shelf component

Next, let’s define a Shelf component that uses our Book component

```
import Book from "./Book";

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

      ComponentDidMount() { /* We want to fetch the books here at some point */ }

      render() {
            return (
                  <div>
                        <Book author=”a” title=”b” />
                        <Book author=”c” title=”d” />
                  </div>
            );
      }
}
```

- We just hard code two books into the shelf for now

Defining a Shelf component - continued

Using the Shelf component on the main page is easy

```
<ui:define name="content">
     import Shelf from "./components/Shelf";

     const App = () => {
          return (
               <Shelf />
          );
     };
     ReactDOM.render(<App />, document.querySelector("#root"));
</ui:define>
```

- At this point we have a shelf and two books in that shelf
- Next, let's consume our web service and populate the shelf with actual data from our model!
- Where did we define our books?

Consuming our web service

To fetch data from our web service, we simply send a GET with the JavaScript fetch() function

```
componentDidMount() {
     fetch("http://localhost:8080/wsbooks-react/ws/shelf")
          .then(res => res.json())
          .then((data) => {
               this.setState({books: data})
          }).catch(console.log);
}
```

- We fetch JSON directly from the web service endpoint ***/shelf***
- This updates the state of the component and populates ***books***
- When we change the state we trigger a re-rendering of the component, causing React to update the view

Consuming our web service

We modify the *render()* method accordingly, fetching the books and populating a list of Book components with the data

```
render() {
     return (
          <div>
               {this.state.books.map((b, idx) => {
                    return <Book author={b.author} title={b.title} key={idx} />
               })}
          </div>
     );
}
```

- React calls the render() method
- We loop through the book array and populate the Shelf component with a series of Book components

Avoiding client-side transpiling

With babel as a dependency on the client-side, the client will always transpile the XHTML files

- Small performance hit incurred, which gets worse with more complex page structures
- You can use ***babel-maven-plugin*** (***github.com/jarslab/babel-maven-plugin***) to instead run this step when the application is being built
- Instead of constantly transpiling, the plugin will generate the files and write them to the target directory before the web archive is built and deployed to the server
- We won’t cover it in detail, as it only complicates the build slightly - but you may play with it if you find it interesting

An example of using React!

Let’s take a look at a test project based on the code we just covered that consumes our web services!