UI Tutorial 1
React

Tutorial Outline

The purpose of this tutorial is to serve as an introduction to developing a web-based front-end components for a RESTful back-end using JavaScript and React.

- We don't have time to do a full introduction to JavaScript or React
- Instead, links to tutorials and documentation will be provided
- Purpose of this tutorial is to point you towards everything you need to understand the UI components implemented in the FaceSpace project and implement your own

What We Will Cover

- JavaScript and React basics
- Rendering React UI components to a web page in a Grails project
- Asynchronous communication between UI and back-end
- Design and Implementation of a new React component

What We Will Not Cover

- Configuring Grails/IntelliJ Workflow
  - Read through this tutorial from the official Grails team – it suggests a great way to configure IntelliJ
    - http://grailsblog.objectcomputing.com/posts/2016/05/28/using-react-with-grails.html
  - FaceSpace project on GitHub is already configured to work well with IntelliJ
- Integration of CSS and other JavaScript components
  - There are different ways of combining HTML/CSS into Grails projects
  - Many examples online, links will be posted

JavaScript and React Resources

JavaScript is a widely-used web programming language.
It can be a little weird at times.
- https://www.destroyallsoftware.com/talks/wat

For programmers new to web, JavaScript can be frustrating to learn.
- Debugging can be difficult
- Many, many libraries/versions to consider
- Different browsers, different implementations

JavaScript and React Resources

- Imperative constructs
  - Loops
  - Conditional statements
- Dynamic Typing
- Object-Oriented
- Functional
  - Functions are objects and may be passed as parameters
  - Anonymous functions (lambda expressions)

JavaScript and React Resources

There are MANY tutorials for JavaScript available online.

Before doing any coding, familiarize yourself with the basic language features: http://www.w3schools.com/js/default.asp

You should read, at least, everything up to and including JS Debugging. Keep this bookmarked as a reference.

JavaScript and React Resources

After familiarizing yourself with JavaScript, you will be ready to start with React.

React is a JavaScript-based framework for developing UI components developed by Facebook.

React is an **extension** of JavaScript.

There is a very good tutorial available here:
https://facebook.github.io/react/tutorial/tutorial.html

React

Before diving into any JavaScript or React code, let's consider what we actually want to do with React.

There is no single, correct way to develop a web-application.

Generally, a combination of HTML, CSS, and JavaScript are used.

React

CSS - Layout, visual parameters
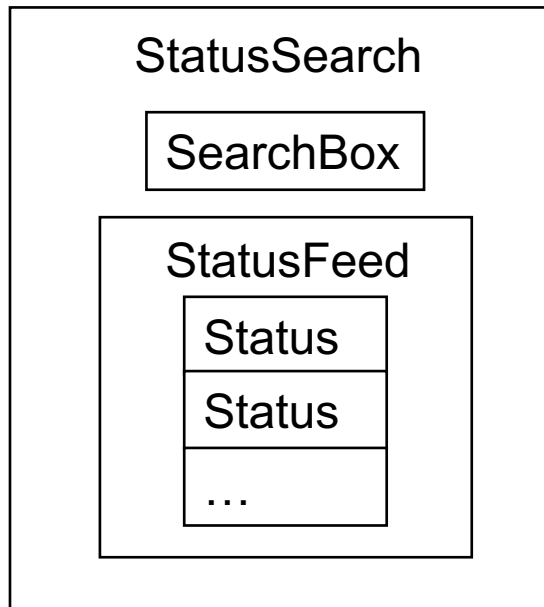HTML – Webpage content and structure
JavaScript – Interactive Functionality

React is a JavaScript library for developing **UI components**.

In our example project, we are only using React to define **self-encapsulated UI components** that can be embedded into an HTML webpage.

React

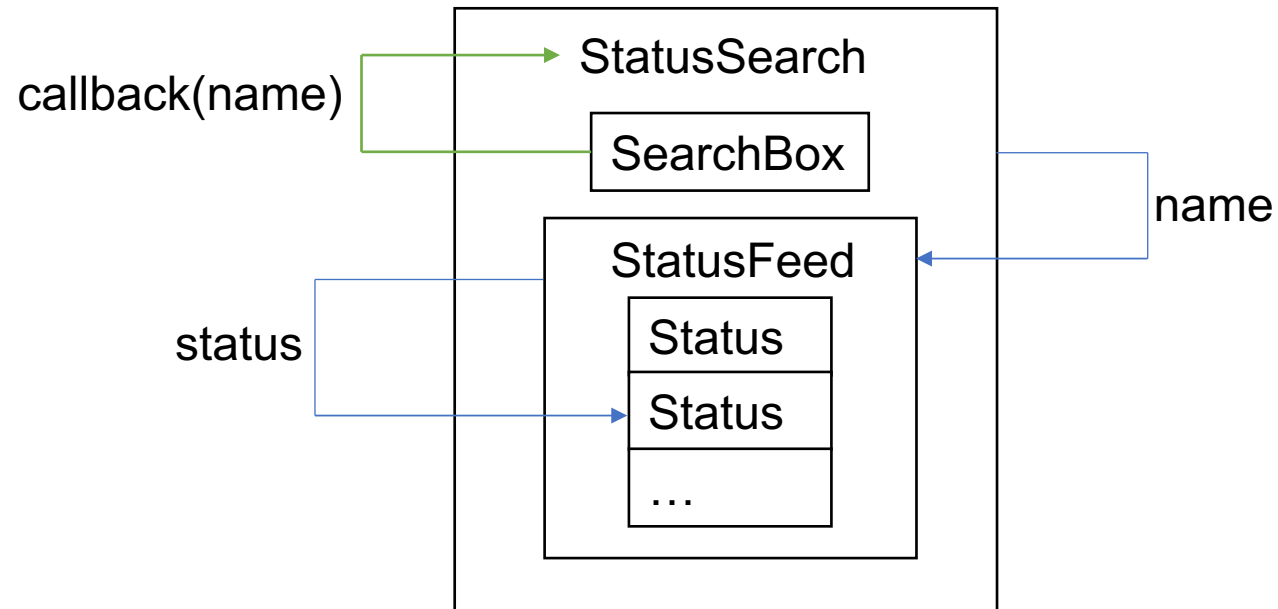React components should be designed hierarchically.

React

React components are expected to be developed using a **one-way data flow** approach.
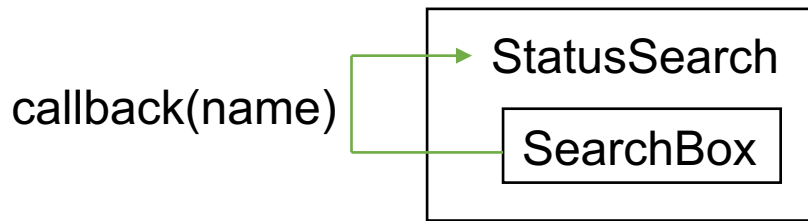
- Properties are passed down to subcomponents or child components
- Child component actions flow back up to the parent
  - Many struggle with this concept initially

React Data Flow Example

It is important to think about data flow before implementing a component.

React Data Flow Example

```
          ┌─────────────────────┐
          │    StatusSearch     │
  ────────────▶│                 │
callback(name)  │  ┌───────────┐  │
  └─────────────│  │ SearchBox │  │
                │  └───────────┘  │
                └─────────────────┘
```

Before statuses can be displayed, a user name needs to be read from the SearchBox component and then passed into the StatusFeed component.

Due to the asynchronous nature of web and UI programming, we achieve this via **bindings** on **state variables**.

Luckily, React makes this very easy!
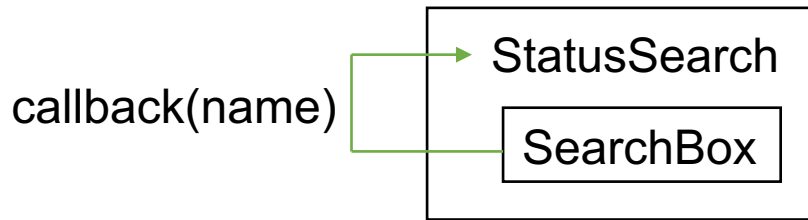
React Data Flow Example

In the StatusSearch class, we define a state variable called **name**.
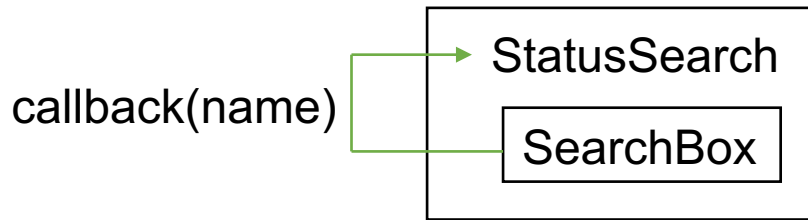
We need a mechanism for SearchBox to update the state of StatusSearch when a name has been entered.

We achieve this by implementing and passing a **callback function**.

React Data Flow Example

When we instantiate a SearchBox as a child component of StatusSearch, we can pass it a function as a parameter.

callback(name)

StatusSearch

SearchBox

We can write SearchBox to call that function when an action has occurred – like clicking a button or pressing enter.

React Data Flow Example

Class StatusSearch

```
getInitialState: function () {
    return {
        name : ""
    }
},
```

```
setNameState(n){
    this.setState({ name: n });
},
```

```
render(){
    return(
        <div>
            Enter a FaceSpace user's name here:<br/>
            <SearchBox callback={this.setNameState}/>
            <br/>
            <StatusFeed name={this.state.name}/>
        </div>
    );
}
```

React Data Flow Example

Class SearchBox

```
handleChange (e) {
    // Prevent following the link
    e.preventDefault();
    this.setState({ name : e.target.value });
},
```

```
handleSubmit(e) {
    // Prevents reinitialization
    e.preventDefault();
    this.props.callback(this.state.name);
},
```

```
render () {
    return (
        <form onSubmit={this.handleSubmit}>
            <label>
                <input type="text" defaultValue={this.state.name} onChange={this.handleChange}/>
            </label>
            <input type="submit" value="Get Statuses!" />
        </form>
    );
}
```

React Data Flow Example

callback(name)

StatusSearch

status

```
render(){
    return(
        <div>
            Enter a FaceSpace user's name here:<br/>
            <SearchBox callback={this.setNameState}/>
            <br/>
            <StatusFeed name={this.state.name}/>
        </div>
    );
}
```
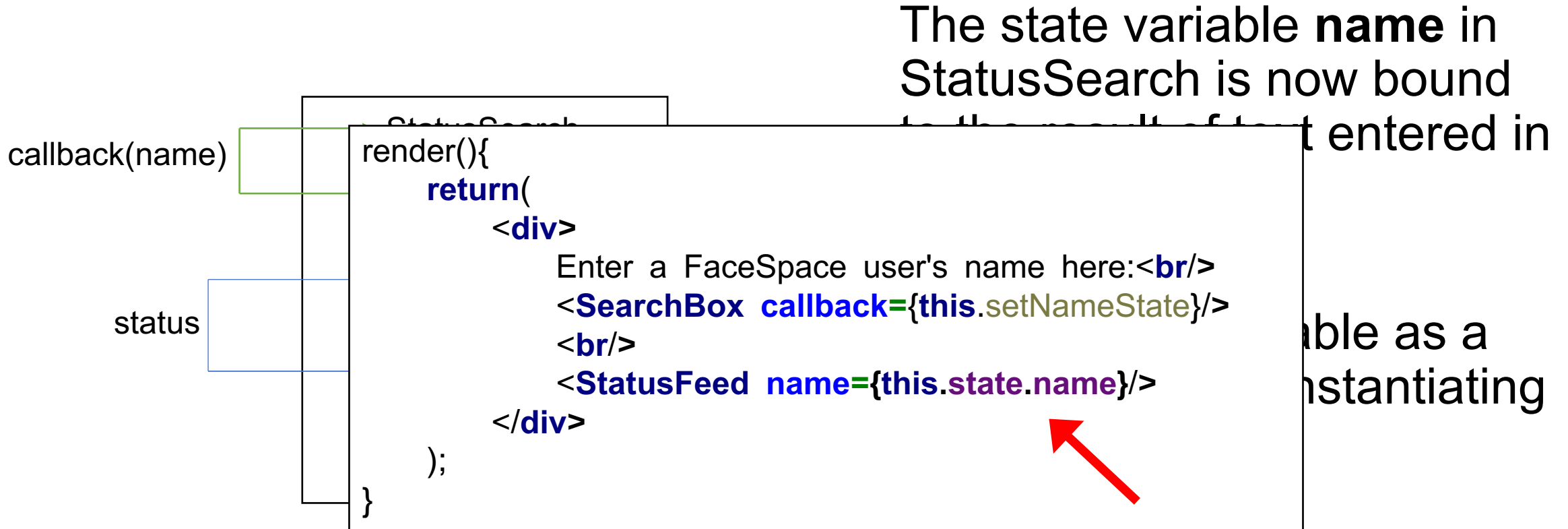
React Data Flow Example

React is doing a lot of work in the background:
- Whenever text is changed in the SearchBox, the state of SearchBox is being updated.
- Whenever the button is clicked or enter is pressed, the state of StatusSearch is being updated with the new text.
- Whenever this happens, the StatusFeed class is being updated with the new text.

The next step is to tell StatusFeed what to do when it receives new data – so let's now look at StatusFeed and Status.
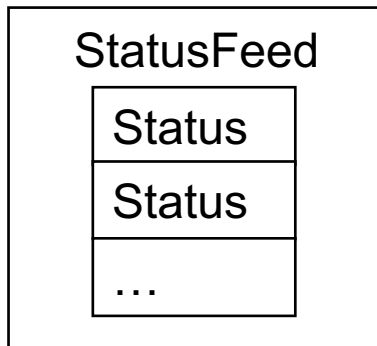
React Data Flow Example

Design UI components to be as self-encapsulated as possible.

Design Questions:
• Does a single status need to know anything about other statuses?
• Does a single status need to know how to retrieve data from the backend?
• Does a single status need to send any data up to the status feed?

React Data Flow Example

Let's design a single Status to be very simple. It does not need to communicate with the backend or send any information back to its parent StatusFeed.

```
var Status = React.createClass({
    getInitialState () {
        return {
        }
    },
    render () {
        return (
            <div>
                <b>{this.props.name}</b>
                <br/>
                {this.props.status}
            </div>
        );
    }
});
```

StatusFeed
Status
Status
…

React Data Flow Example

With Status now implemented, we need to implement the StatusFeed class.

We have already bound one of its properties to the **name** state variable of its parent StatusSearch.

We need to make sure that StatusFeed gets new statuses from the Grails backend whenever this variable is modified.

React Data Flow Example

First, let's look at how to cause a React component to respond to two different events.

React Data Flow Example

Class StatusFeed

```
componentDidMount() {
    let name = this.props.name;
    this.fetchFromAPI(name);
}
```

componentDidMount is called when the component instance has successfully been created.

```
componentWillReceiveProps(nextProps){
    let name = nextProps.name;
    this.fetchFromAPI(name);
}
```

componentWillReceiveProps is called when the component instance's properties are being changed.

A summary of Component methods can be found here: https://facebook.github.io/react/docs/react-component.html

React Data Flow Example

Class StatusFeed

```
componentWillReceiveProps(nextProps){
    let name = nextProps.name;
    this.fetchFromAPI(name);
}
```

Class StatusSearch

```
render(){
    return(
        <div>
            Enter a FaceSpace user's name here:<br/>
            <SearchBox callback={this.setNameState}/>
            <br/>
            <StatusFeed name={this.state.name} />
        </div>
    );
}
```

Because the **property name** of StatusFeed is bound to the **state name** of StatusSearch, componentWillReceiveProps is automatically called whenever the state name of StatusSearch is modified.

These automatically managed state/property bindings are one of React's best features. It makes code much easier to read and write.

React Data Flow Example

Now that StatusFeed will automatically receive the new name property, we can use it to make an API call to retrieve statuses.

To make REST API calls, we can use a JavaScript's Fetch API (available in recent JavaScript implementations or as an import).

By default, the Fetch API uses **asynchronous methods** to communicate with the backend. This is necessary to avoid undesirable latency or unresponsiveness.

React Data Flow Example

An asynchronous methods are used to access resources that may not be immediately available.

Typically, an asynchronous method takes a **callback function** as a parameter that will execute after some conditions have been satisfied (usually a resource access like an API call).

This allows the main program to continue execution even if the resource is taking some time to respond.

React Data Flow Example

Fortunately, the combination of Fetch's clean syntax and React's state variable bindings makes asynchronous coding very easy.

```
fetchExample( ){
    fetch('http://myURL/resource').then(response => {
        if(response.ok) {
            // update  state  variables
        }
        else{
            //update  state  variables
        }
    });
}
```

Instead of passing a callback function as a parameter to **fetch**, we can provide it as a parameter to the **then** method.

In this case we are implementing the callback function body as a **lambda expression**.

i.e. instead of passing in a function as a parameter, we can write an expression that denotes a fully-formed function.

React Data Flow Example

Fortunately, the combination of Fetch's clean syntax and React's state variable bindings makes asynchronous coding very easy.

```
fetchExample( ){
    fetch('http://myURL/resource').then(response => {
        if(response.ok) {
            // update state variables
        }
        else{
            //update state variables
        }
    });
}
```

To avoid adverse side effects, I advise you to only update state variables in response to fetching.

Updating state variables will cause the component to re-render automatically!

React Data Flow Example

Class StatusFeed

```
fetchFromAPI(name){
    fetch('http://localhost:8080/profileDisplay/getUserPosts?userName=' + name).then(response => {
        if(response.ok) {
            response.json().then(json => {
                let results = [];
                for (let i = 0; i < json.length; i++) {
                    results.push(<div><Status name={name} status={json[i].statusText}/><br/></div>);
                }
                this.setState({statuses: results});
            });
        }
        else{
            // If response is NOT OKAY (e.g. 404), clear the statuses.
            this.setState({statuses: []});
        }
    });
}
```

React Data Flow Example
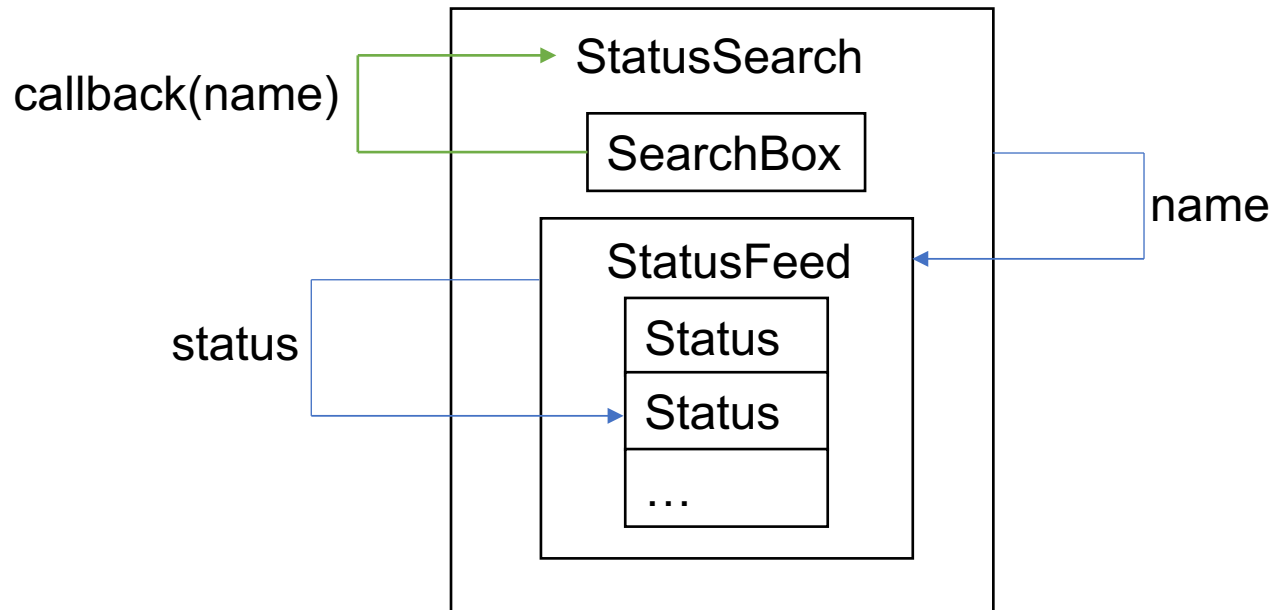
- Username entered into SearchBox
- State variable 'name' is updated
- SearchBox button clicked or enter key
- Callback function from StatusSearch executed
- StatusSearch state variable 'name' is updated
- StatusFeed property variable 'name' is changed
- Property variable 'name' is used to asynchronously fetch Status JSON objects from Grails controller StatusPostController
- On successful response, JSON is used to create array of Status components and update state variable 'statuses'
- State variable update triggers StatusFeed component re-render

React Data Flow Example

Important

- Asynchronous fetch calls should only update state variables
  - Components will re-render only when data has actually been retrieved and state variables have been changed
  - Prevents UI components from becoming unresponsive
- Components pass data down to children via **props**
- Components communicate with parents via **callback functions** that were passed in as instance parameters

Rendering React Components to HTML

We have now developed StatusSearch as a single React component that is composed of other React components.

To use it in an application, we need to render it to a view.

Rendering React Components to HTML

Recall that the purpose of Javascript/React is to define interactive UI components and functions.

We need to use HTML/CSS to tell Grails where and how to render components.

To help us manage our growing collection of JavaScripts and Views, we will use a tool called **webpack**.

Rendering React Components to HTML

Webpack can be used to create bundles of JavaScript files (including React) so that each view can have its own JavaScript resources.

Let's start with a simple example – we just want to render our StatusSearch component to a default view when Grails starts.

Rendering React Components to HTML

First we need to write a JavaScript file that will load components into a view.

Assuming we want to load our StatusSearch into ./grails-app/views/index.gsp – create a file **./src/main/js/index.js**

Webpack is configured to bundle index.js into index.bundle.js and place it into **./assets/javascripts/** when Grails starts. This is a necessary step to make the JavaScripts available to our views.

Rendering React Components to HTML

./src/main/js/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import { StatusSearch } from './statussearch';

ReactDOM.render(
   <div>
      <StatusSearch/>
   </div>, document.getElementById('statusFeed'));
```

Import statements
- Import React resources
- ReactDOM = React Document Object Model
  - Provides functions for rendering to HTML docs
- Import StatusSearch component

ReactDOM.render is responsible for rendering content to the parent document.

Assuming index.bundle.js has been imported by index.gsp, this function will look for an element with id 'statusFeed' and render our StatusSearch component there.

Rendering React Components to HTML

./grails-app/views/index.gsp

```html
<!doctype html>
<html>
   <head>
      <title>FaceSpace</title>
   </head>
   <body>
      <div id="statusFeed" align="left"></div>
      <asset:javascript src="index.bundle.js"/>
   </body>
</html>
```

Putting everything together, we now have a container called 'statusFeed' in which React can render our StatusSearch.

Don't forget to import the JavaScript code – otherwise your components will not render!

Configuring webpack.config.js

More information can be found here:
https://webpack.github.io/docs/configuration.html

Important note:
You need to modify webpack.config.js any time you create a new, main Javascript for a view.
- For example, if you create a new view called friendlist.gsp, you should also create a JavaScript file (e.g. friendlist.js) that is only responsible for rendering UI components. Add this .js file under entries in webpack.config.js

Configuring webpack.config.js

./webpack.config.js

```javascript
var path = require('path');

module.exports = {
    entry: {
        index: './src/main/js/index.js',
        page2: './src/main/js/page2.js'
    },
    output: {
        path: './grails-app/assets/javascripts',
        publicPath: '/assets/',
        filename: '[name].bundle.js'
    },
    …
```

Specify one main JavaScript file per view.

Makes sure multiple bundles can be created and saved in the Grails assets directory.
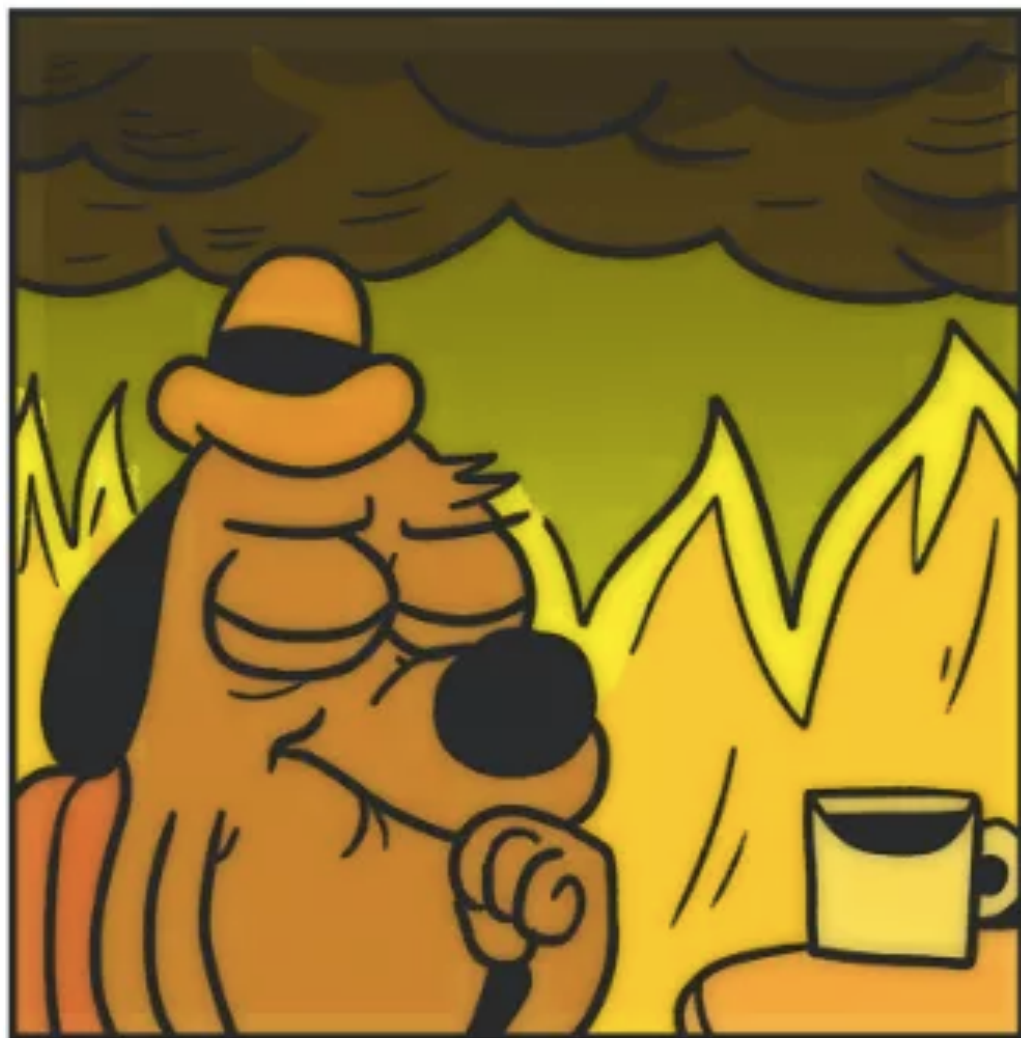
Dependencies

JDK, Groovy, Grails, Gradle, npm, webpack, babel, React…

"Are you kidding me?"

This is actually a **very** easy to use web development stack. If you are afraid of managing all of these dependencies – just clone my project and hit the ground running.

Recommended:
Go through the configuration tutorial suggested on Slide 4. Just focus on the workflow elements (don't worry about the Grails project or the database stuff, just install the dependencies in your own project).
- There will be errors, but if you read the error messages carefully, you should be able to resolve them.
- In a pinch, that's what I'm here for.

Your Tutorial Task

First – read through the [basic documentation on JavaScript](#).
• Try to relate JavaScript to a programming language you know.

Second – at least **read** the [React tutorial](#). Even better to play with the code in CodePen. Really try to embrace its data flow model.

Third – implement a React UI component to create a new UserAccount in FaceSpace.

Your Tutorial Task

Implement a React UI component to create a new UserAccount in FaceSpace.

- Calls AccountCreationController REST API
- Based on response
  - Show message saying account creation was successful, or
  - Show message saying name was already taken for account
- After creating an account, post some status updates to it.
- Then search for the new account's statuses.

Your Tutorial Task

Tips:
- This UI component will be very similar to StatusUpdate.
- Read StatusUpdate thoroughly before starting.
  - Understand how the component is defined and how information flows.
  - Understand the POST API call and response.
- Make sure you are using the up-to-date version of FaceSpace that includes the AccountCreationController class.

Your Tutorial Task - Example

Ethan [Create Account!]

Name: Ethan
Message: Account: Ethan was already taken...

Enter a FaceSpace user's name here:
[   ] [Get Statuses!]

[   ] [   ] [Post Status!]
Name:
Status:
Success:

Your Tutorial Task - Example

Your Tutorial Task

Submission on OWL Due Wed March 1st at 11:55pm

- Submit a single JavaScript file called **accountcreation.js** (2.5%)
- Work on the empty file provided in /src/main/js/
- Must be compatible with example project from GitHub

Your Tutorial Task

Notice that in /src/main/js/index.js, there is already an AccountCreation import statement and element.

Note that your code will not render if there is an error in accountcreation.js – try removing the element to ensure the rest of the code is working.

Your implementation will be tested by rendering a single <AccountCreation/> element from index.