Comp 324/424 - Client-side Web Design

Spring Semester 2017 - Week 13

Dr Nick Hayward

Contents

- Final presentation & report
- Final assessment outline
- Project status report
- Data visualisation examples &c.
- Extras
  - *React*

Final Presentation & Report

- team presentation on Monday 24th April @ 4.15pm
- team report due on Monday 1st May by 4.15pm

Final Assessment Outline

- continue to develop your app concept and prototypes
- working app (as close as possible…)
  - *NO PHP, Python, Ruby, Go, XML, SQL, Bootstrap…*
- explain design decisions
  - *describe patterns used in design of UI and interaction*
  - *layout choices…*
- show and explain implemented differences from DEV week
  - *where and why did you update the app?*
  - *perceived benefits of the updates?*
- how did you respond to peer review?
- any other useful or pertinent information
- coursework outline

n.b. 10 minutes maximum per group

Final Assessment Report

- report outline - demo and report

Project Status Report

- what is currently working?
- which data store?
- any APIs? Internal or remote...
- what is left to add or fix?
  - *features, UI elements, interactions...*
- who is working on what?
  - *logic, design, testing, research...*

Teams

- Group 1 - Team Chicago
- Group 2 - Team FSSA
- Group 3 - Team Health
- Group 4 - Team JOKE
- Group 5 - Team JRE
- Group 6 - Team M2NS
- Group 7 - Team MDS
- Group 8 - Team Purple
- Group 9 - Team Sustainability
- Group 10 - Team VHR
- Group 11 - Team Weather.Mix

Data Visualisation

general examples

Sample dashboards and visualisations

- gaming dashboard
- schools and education
- students and grades
- D3 examples

Example datasets

- Chicago data portal

Article example

- dashboard designs
- replace jQuery with D3

Data Visualisation

projects examples

A few examples from recent projects,

- GitHub API tests
- check JSON return
- early test examples
- metrics test examples

React JavaScript Library

overview

- **React** began life as a port of a custom PHP framework called XHP
  - *developed internally at Facebook*
- XHP, as a PHP framework, was designed to render the full page for each request
- **React** developed from this concept
  - *creating a client-side implementation of loading the full page*
- **React** can, therefore, be perceived as a type of *state machine*
  - *control and manage inherent complexity of state as it changes over time*
- able to achieve this by concentrating on a narrow scope for development,
  - *maintaining and updating the DOM*
  - *responding to events*
- **React** is best perceived as a *view* library
  - *no definite requirements or restrictions on storage, data structure, routing...*
- allows developers freedom
  - *incorporate **React** code into a broad scope of applications and frameworks*

React JavaScript Library

why use React?

- React is often considered the *V* in the traditional *MVC*
- [React(http://facebook.github.io/react/docs/why-react.html) was designed to solve one problem

> *building large applications with data that changes over time*

- React can best be considered as addressing the core concerns
  - *simple, declarative, components*
- simple - define how your app should look at any given point in time
  - *React handles all UI changes and updates in response to data changes*
- declarative - as data changes, React effectively refreshes your app
  - *sufficiently aware to only update those parts that have changed*
- components - fundamental principle of React is building re-usable components
  - *components are encapsulated in their design and concepts*
  - *they make it simple for code re-use, testing...*
  - *in particular, the separation of design and app concerns in general*
- React leverages its built-in, powerful rendering system to produce
  - *quick, responsive rendering of DOM in response to received state changes*
- uses a virtual DOM
  - *enables React to maintain and update the DOM without the lag of reading it as well*

React JavaScript Library

state changes

- as **React** is informed of a state change, it re-runs render functions
- enables it to determine a new representation of the page in its virtual DOM
- then automatically translated into the necessary changes for the new DOM
  - *reflected in the new rendering of the view*
- may, at first glance, appear inherently slow
  - *React uses an efficient algorithm*
  - *checks and determines differences*
  - *differences between current page in the virtual DOM and the new virtual one*
- from these differences it makes the minimal set of necessary updates to the rendered DOM
- creates speed benefits and gains
  - *minimises usual reflows and DOM manipulations*
- also minimises effect of cascading updates caused by frequent DOM changes and updates

React JavaScript Library

component lifecycle

- in the lifecycle of a component
  - *its props or state might change along with any accompanying DOM representation*
- in effect, a component is a known state machine
  - *it will always return the same output for a given input*
- following this logic, React provides components with certain *lifecycle* hooks
  - *instantiation - mounting*
  - *lifetime - updating*
  - *teardown - unmounting*
- we may consider these hooks
  - *first through the instantiation of the component*
  - *then its active lifetime*
  - *finally its teardown*

React JavaScript Library

component lifecycle - intro

- React components include a minimal lifecycle API
- provides the developer with enough without being overwhelming
  - *at least in theory*
- React provides what are known as *will* and *did* methods
  - *will - called right before something happens*
  - *did - called right after something happens*
- relative to the lifecycle, we can consider the following groupings of methods
  - *Instantiation (mounting)*
  - *Lifetime (updating)*
  - *Teardown (unmounting)*
  - *Anti-pattern (calculated values)*

React JavaScript Library

component lifecycle - method groupings - Instantiation (mounting)

- includes methods called upon instantiation for the selected component class
- eg: `getDefaultProps` or `getInitialState`
  - *use such methods to set default values for new instances*
  - *initialise a custom state of each instance...*
- also have the important `render` method
  - *builds our application's virtual DOM*
  - *the only required method for a component*
- `render` method has rules it needs to follow
  - *such as accessible data*
  - *return values*
- `render` method must also remain *pure*
  - *cannot change the state or modify the DOM output*
  - *returned result is the virtual DOM*
  - *compared against actual DOM*
  - *helps determine if changes are required for the application*

React JavaScript Library

component lifecycle - method groupings - Lifetime (updating)

- component has now been rendered to the user for viewing and interaction
- as a user interacts with the component
  - *they are changing the state of that component or application*
  - *allows us as developers to act on the relevant points in the component tree*
- State changes for the application
  - *those affecting the component*
  - *may result in update methods being called*
- we're telling the component how and when to update

React JavaScript Library

component lifecycle - method groupings - Teardown (unmounting)

- as React is finished with a component
  - *it must be unmounted from the DOM and destroyed*
- there is a single hook for this moment
  - *provides opportunity to perform necessary cleanup and teardown*
- `componentWillUnmount`
  - *removes component from component hierarchy*
  - *this method cleans up the application before component removal*
  - *undo custom work performed during component's instantiation*

React JavaScript Library

component lifecycle - method groupings - Anti-pattern (calculated values)

- React is particularly concerned with maintaining a single source of truth
- one point where `props` and `state` are derived, set...
- consider calculated values derived from `props`
  - *considered an anti-pattern to store these calculated values as state*
- if we needed to convert a `props` date to a string for rendering
  - *this is not state*
  - *it should simply be calculated at the time of `render`*

React JavaScript Library

a few benefits

- one of the main benefits of this virtual approach
  - *avoidance of micro-managing any updates to the DOM*
- a developer simply informs React of any changes
  - *such as user input*
- React is able to process those passed changes and updates
- React has inherent benefit of delegating all events to a single event handler
  - *naturally gives React an associated performance boost*

React JavaScript Library

getting started - part 1

- many different options for using React
- create a new app using React
  - *e.g. Create React App - GitHub*
- add React to an existing app
  - *e.g. using NPM to install React and depencies*

```
npm init
npm install --save react react-dom
```

- import React into a project using the standard Node `import` options, e.g.

```
import React from 'react';
import ReactDOM from 'react-dom';
```

React JavaScript Library

getting started - part 2

- for earlier versions of React and JSX
  - *pre-compile JSX into JavaScript before deploying our application*
  - *used React's JSXTransformer option to compile and monitor JSX for dev projects*
- as React has evolved over the last year
  - *still use this underlying concept*
  - *Babel in-browser JSX transformer for explicit ES6 support (if required...)*
- Babel will add a check to our app to allow us to use JSX syntax
  - *React code then understood by the browser*
- dynamic transformation works well for most test scenarios
  - *preferable to pre-compile for production apps*
  - *should help to make an app faster for production usage*

React JavaScript Library

JSX - intro

- JSX stands for **JavaScript XML**
  - *follows an XML familiar syntax for developing markup within React components*
- JSX is not compulsory within React
  - *might be omitted due to compile requirements for an app*
- JSX may be useful for an app
  - *it makes components easier to read and understand*
  - *its structure is more succinct and less verbose*
- A few defining characteristics of JSX
  - *each JSX node maps to a function in JavaScript*
  - *JSX does not require a runtime library*
  - *JSX does not supplement or modify the underlying semantics of JavaScript*

React JavaScript Library

JSX - benefits

- why use JSX, in particular when it simply maps to JavaScript functions?
- many of the inherent benefits of JSX become more apparent
  - *as an application, and its code base, grows and becomes more complex*
- benefits can include
  - *a sense of familiarity - easier with experience of XML and DOM manipulation*
  - *eg: React components capture all possible representations of the DOM*
  - *JSX transforms an application's JavaScript code into semantic, meaningful markup*
  - *permits declaration of component structure and information flow using a similar syntax to HTML*
  - *permits use of pre-defined HTML5 tag names and custom components*
  - *easy to visualise code and components*
  - *considered easier to understand and debug*
  - *ease of abstraction due to JSX transpiler*
  - *abstracts process of converting markup to JavaScript*
  - *unity of concerns*
  - *no need for separation of view and templates*
  - *React encourages discrete component for each concern within an application*
  - *encapsulates the logic and markup in one definition*

React JavaScript Library

JSX - composite components

- example React component might allow us to output a HTML heading

```
var heading = React.createClass({
  render: function() {
    return (
      <div className="heading">
      <h2>Welcome</h2>
      </div>
    );
  }
});
```

- currently fixed to `Welcome` heading
- now update this example to work with dynamic values
- JSX considers values dynamic if they are placed between curly brackets `{..}`
  - *treated as JavaScript context*

```
var heading = 'Welcome';
<h2>{heading}</h2>
```

React JavaScript Library

JSX - more dynamic values

- also call functions
  - *move some logic for a component to a standard JavaScript function*
- then call this function, plus any supplied parameters
- React can also evaluate arrays, and then output each value

```
var heading = React.createClass({
  render: function() {
    var text = ['welcome', 'home'];
      return (
        <h3>{text}</h3>
      );
    }
});

React.render(<heading />, document.getElementById('example'));
```

React JavaScript Library

JSX - conditionals

- a component's markup and its logic are inherently linked in React
- this naturally includes *conditionals*, *loops*...
- adding `if` statements directly to JSX will create invalid JavaScript

1. ternary operator

```
render: function() {
  return <div className={
    this.state.isComplete ? 'is-complete' : ''
  }>...</div>
}
```

2. variable

```
getIsComplete: function() {
  return this.state.isComplete ? 'is-complete' : '';
},
render: function() {
  var isComplete = this.getIsComplete();
  return <div className={isComplete}>...</div>
}
```

3. function call

```
getIsComplete: function() {
  return this.state.isComplete ? 'is-complete' : '';
},
render: function() {
  return <div className={this.getIsComplete()}>...</div>;
}
```

- to handle React's lack of output for *null* or *false* values
  - *use a boolean value and follow it with the desired output*

React JavaScript Library

JSX - non-DOM attributes - part 1

- in JSX, there are three special considerations for attribute
  - `key`
  - `ref`
  - `dangerouslySetInnerHTML`

1. `key`

- an optional unique identifier that remains consistent throughout render passes
- informs React so it can more efficiently select when to reuse or destroy a component
- helps improve the rendering performance of the application.
- eg: if two elements already in the DOM need to switch position
  - *React is able to match the keys and move them*
  - *does not require unnecessary re-rendering of the complete DOM*

React JavaScript Library

JSX - non-DOM attributes - part 2

2. `ref`

- `ref` permits parent components to easily maintain a reference to child components
  - *available outside of the render function*
- to use `ref`, simply set the attribute to the desired reference name

```
render: function() {
  return <div>
    <input ref="myInput" ... />
    </div>;
}
```

- able to access this `ref` using the defined `this.refs.myInput`
  - *access anywhere in the component*
  - *object accessed through this `ref` known as a backing instance*
- **NB:** not the actual DOM
  - *a description of the component React uses to create the DOM when necessary*
- access DOM itself for this `ref`
  - *use `this.refs.myInput.getDOMNode()`, where myInput is name of previously defined ref*

React JavaScript Library

JSX - non-DOM attributes - part 3

3. `dangerouslySetInnerHTML`

- When absolutely necessary, React can set HTML content as a string using this attribute
- to correctly use this property set it as an object with key `__html`

```
render: function() {
  var htmlString = {
    __html: "<span>...your html string...</span>"
  };
  return <div dangerouslySetInnerHTML={htmlString}></div>;
}
```

React JavaScript Library

JSX - reserved words

- JSX transforms to plain JavaScript functions
  - *means there are some reserved or special attributes*
  - *eg: we can't use `class` or `for`*
- to create a form label with the `for` attribute we can use `htmlFor` instead

```
<label htmlFor="text...">
```

- create a custom class we can use `className`

```
<div className={class...}>
```

React JavaScript Library

data flow

- data flows in one direction in React
  - *namely from **parent to child***
- helps to make components nice and simple, and predictable as well
- components take *props* from the parent, and then render
- if a *prop* has been changed, for whatever reason
  - *React will update the component tree for that change*
  - *then re-render any components that used that property*
- Internal state also exists for each component
  - *state should only be updated within the component itself*
- we can think of data flow in React
  - *in terms of `props` and `state`*

React JavaScript Library

data flow - `props` - part I

- `props` can hold any data and are passed to a component for usage
- set `props` on a component during instantiation

```
var classics = [{ title: 'Greek'}];
<ListClassics classics={classics}/>
```

- also use the `setProps` method on a given instance of a component

```
var ListClassics = React.createClass({
  render: function() {
    return (
      <li className="classic">{this.props.classics}</li>
    );
  }
});

var classics = [{ title: 'Greek'}];
var listClassics = React.render (
  <ListClassics/>,
    document.getElementById('example')
);

listClassics.setProps({ classics: classics });
```

React JavaScript Library

data flow - `props` with JSX

- set `props` using { } syntax
  - *allows us to pass variables of any type via JavaScript injection*

```
<a href={'/classics/' + classic.id}>{classic.title}</a>
```

- also pass event handlers as `props`

```
var EditButton = React.createClass({
  render: function() {
    return (
      <a className='button edit' onClick={this.handleClick}>Edit</a>
      );
  },
  handleClick: function() {
  //handle click...
    alert('edit button clicked...');
  }
});
```

React JavaScript Library

state - intro - part 1

- a component in React is able to house *state*
- *State* is inherently different from `props` because it is internal to the component
- it is particularly useful for deciding a view state on an element
  - *eg: we could use state to track options within a hidden list or menu*
  - *track the current state*
  - *change it relative to component requirements*
  - *then show options based upon this amended state*
- **NB:** considered bad practice to update state directly using `this.state`
  - *use the method `this.setState`*
- try to avoid storing computed values or components directly in *state*
- focus upon using simple data
  - *directly required for given component to function correctly*
- considered good practice to perform required calculations in the `render` function
- try to avoid duplicating `prop` data into `state`
  - *use the `props` data instead*

React JavaScript Library

state - intro - part 2

```
var EditButton = React.createClass({
  getInitialState: function() {
    return {
      editShow: true
    };
  },
  render: function() {
    if (this.state.editShow == false) {
      alert('edit button will be turned off...');
    }
    return (
      <button className="button edit" onClick={this.handleClick}>Edit</button>
    );
  },
  handleClick: function() {
  //handle click...
  alert('edit button clicked');
  //set state after button click
  this.setState({ editShow: false });
  }
});
```

React JavaScript Library

state - intro - part 3

- when designing React apps, we often think about
  - ***stateless children*** *and a* ***stateful parent***

> *A common pattern is to create several stateless components that just render data, and have a stateful component above them in the hierarchy that passes its state to its children via props.*
>
> *React documentation*

- need to carefully consider how to identify and implement this type of component hierarchy

1. Stateless child components
   - *components should be passed data via* `props` *from the parent*
   - *to remain stateless they should not manipulate their* `state`
   - *they should send a callback to the parent informing it of a change, update etc*
   - *arent will then decide whether it should result in a* `state` *change, and a re-rendering of the DOM*
2. Stateful parent component
   - *can exist at any level of the hierarchy*
   - *does not have to be the root component for the app*
   - *instead can exist as a child to other parents*
   - *use parent component to pass* `props` *to its children*
   - *maintain and update state for the applicable components*

React JavaScript Library

state - intro - part 4

1. props vs state
   - *in React, we can often consider two types of model data*
   - *includes `props` and `state`*
   - *most components normally take their data from `props`*
   - *allows them to render the required data*
   - *as we work with users, add interactivity, and query and respond to servers*
   - *we also need to consider the `state` of the application*
   - *`state` is very useful and important in React*
   - *also important to try and keep many of our components stateless*
2. state
   - *React considers user interfaces, UIs, as simple state machines*
   - *acting in various states and then rendering as required*
   - *in React, we simply update a component's state*
   - *then render the new corresponding UI*

React JavaScript Library

state - intro - part 5

1. How state works

- if there is a change in data in the application
  - *perhaps due to a server update or user interaction*
  - *quickly and easily inform React by calling* `setState(data, callback)`
- this method allows us to easily merge `data` into `this.state`
  - *re-renders the component*
- as re-rendering is finished
  - *optional* `callback` *is available and is called by React*
- this `callback` will often be unnecessary
  - *it's still useful to know it is available*

React JavaScript Library

state - intro - part 6

2. In state

- try to keep data in `state` to a minimum
  - *consider minimal possible representation of an application's state*
  - *helps build a stateful component*
- `state` should try to just contain minimal data
  - *data required by a component's event handlers to help trigger a UI update*
  - *if and when they are modified*
- such properties should also normally only be stored in `this.state`
- as we render the updated UI
  - *simply compute required information in the `render()` method based on this `state`*
  - *avoids need to keep computed values in sync in state*
  - *instead relying on React to compute them for us*

3. out of state

- in React, `this.state` should only contain minimal data
- minimum necessary to represent an application's UI state
- should contain
  - *computed value/values*
  - *React components*
  - *duplicated data from `props`*

React JavaScript Library

state - an example app - part 1

- a simple app to allow us to test the concept of stateful parent and stateless child components
- resultant app outputs two parallel `div` elements
- allow a user to select one of the available categories
- then view all of the available *authors*

```
//static test data...
var AUTHORS = [
  {id:1, category: 'greek', categoryId:1, author: 'Plato'},
  {id:2, category: 'greek', categoryId:1, author: 'Aristotle'},
  {id:3, category: 'greek', categoryId:1, author: 'Aeschylus'},
  {id:4, category: 'roman', categoryId:2, author: 'Livy'},
  {id:5, category: 'greek', categoryId:1, author: 'Euripides'},
  {id:6, category: 'roman', categoryId:2, author: 'Ptolemy'},
  {id:7, category: 'greek', categoryId:1, author: 'Sophocles'},
  {id:8, category: 'roman', categoryId:2, author: 'Virgil'},
  {id:9, category: 'roman', categoryId:2, author: 'Juvenal'}
];
```

- start with some static data to help populate our app
- `categoryId` used to filter unique categories
  - *again to help get all of our authors per category*

React JavaScript Library

state - an example app - part 2

- for `stateless` child components
  - *need to output a list of filtered, unique categories*
  - *then a list of authors for each selected category*
- first child component is the `CategoryList`
  - *filters and renders our list of unique categories*
  - *`onClick` attribute is included*
  - *state is therefore passed via callback to the `stateful` parent*

React JavaScript Library

state - an example app - part 3

```
//output unique categories from passed data...
var CategoryList = React.createClass({
 render: function() {
  var category = [];
   return (
    <div id="left-titles" className="col-6">
     <ul>
      {this.props.data.map(function(item) {
        if (category.indexOf(item.category) > -1) {
        } else {
         category.push(item.category);
          return (
           <li key={item.id} onClick={this.props.onCategorySelected.bind(null, item.categoryId)}>
            {item.category}
           </li>);
          }}, this)}
      </ul>
     </div>
   );
  }
});
```

- the component is accepting `props` from the parent component
  - *then informing this parent of a required change in state*
  - *change reported via a callback to the `onCategorySelected` method*
  - *does not change `state` itself*
  - *it simply handles the passed data as required for a React app*

React JavaScript Library

state - an example app - part 4

- need to consider our second `stateless` child component
  - *renders the user's chosen authors per category*
  - *user clicks on their chosen category*
  - *a list of applicable authors is output to the right side div*

```
var AuthorList = React.createClass({
 render: function() {
  return (
   <div id="right-titles" className="col-md-6 col-sm-6 col-xs-6">
    <ul>
     {this.props.authors.map(function(item) {
      return (
       <li key={item.id}>{item.author}</li>
       );
      })
     }
    </ul>
   </div>
  );
 }
});
```

- this component does not set any state
- simply rendering the passed `props` data for viewing

React JavaScript Library

state - an example app - part 5

- to handle updates to the DOM, we need to consider our `stateful` parent
- this component passes the app's data as `props` to the children
- handles the setting and updating of the `state` for app as well
- as noted in the React documentation,

> *State should contain data that a component's event handler may change to trigger a UI update.*

- for this example app
  - *only need to store the `selectedCategoryAuthors` in `state`*
  - *enables us to update the UI for our app*

React JavaScript Library

state - an example app - part 6

```
var Container = React.createClass({
    getInitialState: function() {
        return {
        selectedCategoryAuthors: this.getCategoryAuthors(this.props.defaultCategoryId)
        };
    },
  getCategoryAuthors: function(categoryId) {
        var data = this.props.data;
        return data.filter(function(item) {
            return item.categoryId === categoryId;
        });
    },
  render: function() {
    return (
      <div className="container col-md-12 col-sm-12 col-xs-12">
      <CategoryList data={this.props.data} onCategorySelected={this.onCategorySelected} />
      <AuthorList authors={this.state.selectedCategoryAuthors} />
      </div>
    );
  },
  onCategorySelected: function(categoryId) {
    this.setState({
      selectedCategoryAuthors: this.getCategoryAuthors(categoryId)
    });
  }
});
```

React JavaScript Library

state - an example app - part 7

- our `stateful` parent component sets its initial state
  - *including passed data and app's selected category for authors*
- helps set a default state for the app
  - *we can then modify as a user selects their chosen category*
- callback for this user selected category is handled in the `onCategorySelected` method
  - *updates the app's state for the chosen `categoryId`*
  - *then leads to the app re-rendering the DOM for any changes*
- we still have computed data in the app's `state`
  - *as noted in the React documentation,*

> *`this.state` should only contain the minimal amount of data needed to represent your UIs state...*

- we should now move our computations to the `render` method of the parent component
  - *then update `state` accordingly*

React JavaScript Library

state - an example app - part 8

```
var Container = React.createClass({
    getInitialState: function() {
    return {
      selectedCategoryId: this.props.defaultCategoryId
    };
  },
  render: function() {
    var data = this.props.data;
    var selectedCategoryAuthors = data.filter(function(item){
      return item.categoryId === this.state.selectedCategoryId;
    }, this);
    return (
        <div className="container col-md-12 col-sm-12 col-xs-12">
        <CategoryList data={this.props.data} onCategorySelected={this.onCategorySelected} />
        <AuthorList authors={selectedCategoryAuthors} />
        </div>
    );
  },
    onCategorySelected: function(categoryId) {
    this.setState({selectedCategoryId: categoryId});
  }
});
```

- `state` is now solely storing the `categoryId` for our app
- can be modified and the DOM re-rendered correctly

React JavaScript Library

state - an example app - part 9

- we can then load this application
  - *passing data as props to the `Container`*
  - *data from JSON Authors*

```
var buildLibrary = React.render (
 <Container data={AUTHORS} defaultCategoryId='1' />,
 document.getElementById('library')
);
```

- DEMO - state example

React JavaScript Library

state - minimal state - part 1

- to help make our UI interactive
  - *use React's `state` to trigger changes to the underlying data model of an application*
  - *need to keep a minimal set of mutable state*
- **DRY**, or *don't repeat yourself*
  - *often cited as a good rule of thumb for this minimal set*
- need to decide upon an absolute minimal representation of the `state` of the application
  - *then compute everything else as required*
  - *eg: if we maintain an array of items*
  - *common practice to calculate array length as needed instead of maintaining a counter*

React JavaScript Library

state - minimal state - part 2

- as we develop an application with React
  - *start dividing our data into logical pieces*
  - *then start to consider which is state*
- for example,
  - *is it from `props`?*
  - *if yes, this is probably not `state` in React*
  - *does it update or change over time? (eg: due to API updates etc)*
  - *if yes, this is probably not `state`*
  - *can you compute the data based upon other `state` or `props` in a component?*
  - *if yes, it is not state*
- need to decide upon our minimal set of components that mutate, or *own* state
  - *React is based on the premise of one-way data flow down the hierarchy of components*
  - *can often be quite tricky to determine*
- initially, we can check the following
  - *each component that renders something based on state*
  - *determine the parent component that needs the state in the hierarchy*
  - *a common or parent component should own the state*
    - *NB:* if this can't be determined
    - simply create a basic component to hold this state
    - add component at the top of the state hierarchy

React JavaScript Library

Additional reading, material, and samples

- design thoughts
- event handling
- more composing components
- DOM manipulation
- forms
- intro to flux
- animations
- lots of samples...

Demos

- state example

References

- React
  - *React*
  - *React - API Reference*
  - *React - Create React App - GitHub*