Extra Notes - React - Getting Started

- Dr Nick Hayward

React JavaScript Library

A quick introduction to React JavaScript library. Further details are available at the React website.

Contents

Intro

React began life as a port of a custom PHP framework called XHP, which was developed internally at Facebook. XHP, as a PHP framework, was designed to render the full page for each request. **React** developed from this concept, thereby creating a client-side implementation of loading the full page.

Overview

React can, therefore, be perceived as a type of *state machine*, thereby allowing a developer to control and manage the inherent complexity of state as it changes over time. It is 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, and has no definite requirements or restrictions on storage, data structure, routing, and so on. This allows developers the freedom to incorporate **React** code into a broad scope of applications and frameworks.

why use React?

React is often considered the *V* in the traditional *MVC*. As defined in the React docs, it was designed to solve one problem,

> building large applications with data that changes over time.

React, therefore, can best be considered as addressing the following

- simple - define how your app should look at any given point in time, and React handles all UI changes and updates in response to data changes
- declarative - as this data changes, React effectively refreshes your app and is sufficiently aware to only update those parts that have changed
- components - a fundamental principle of React is building re-usable components. These components are so encapsulated in their design and concepts, they make it simple for code *re-use*, *testing*, and the separation of design and app concerns in general.

React leverages its built-in, powerful rendering system to produce quick, responsive rendering of the DOM in response to received state changes. It uses a virtual DOM, which enables **React** to maintain and update the DOM without the lag of reading it as well.

state changes

As **React** is informed of a state change, it re-runs render functions. This enables it to determine a new representation of the page in its virtual DOM. This is then automatically translated into the necessary changes for the new DOM, which is reflected in the new rendering of the view.

This may, at first glance, appear inherently slow. However, **React** uses an efficient algorithm to check and determine the differences between the 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.

This creates speed benefits and gains as it minimises the usual reflows and DOM manipulations. It also minimises the effect of cascading updates caused by frequent DOM changes and updates.

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, and it will always return the same output for a given input.

Following this logic, React provides components with certain *lifecycle* hooks. For example,

- instantiation - mounting
- lifetime - updating
- teardown - unmounting

So, we may consider these hooks first through the instantiation of the component, then its active lifetime, and finally its teardown.

a few benefits

One of the main benefits of this virtual approach is the avoidance of micro-managing any updates to the DOM. Instead, a developer simply informs **React** of any changes, such as user input, and it is able to process those passed changes and updates.

React has the inherent benefit of delegating all events to a single event handler, which naturally gives **React** an associated performance boost.

Getting started

How we start to use React is often dependent upon the application itself. For example, we may choose to start with a new app from scratch. So, we can use the *Create React App* boilerplate to get started,

- https://github.com/facebookincubator/create-react-app

We may also choose to add React to an existing project. The application doesn't need to be rewritten from scratch to work with React. Instead, it may simply integrate widgets or components built using React.

part 1

As with most JS based apps today, there are many different ways to setup and manage a React-based app. We might choose to use NPM or Browserify, for example, to install and maintain dependencies and updates, and then later publish any useful components as well.

For example, to add React to a project with NPM we may use the following commands,

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

As with plain JavaScript, we can choose a number of different ways to store and access the JavaScript code. We can save it in the HTML file itself, in a separate file, or reference from a third party remote source.

We may also add any required React files using standard content delivery networks, or CDNs. For example, we may use the following for development,

```
<script src="https://unpkg.com/react@15/dist/react.js"></script>
<script src="https://unpkg.com/react-dom@15/dist/react-dom.js"></script>
```

and then the minified versions for production,

```
<script src="https://unpkg.com/react@15/dist/react.min.js"></script>
<script src="https://unpkg.com/react-dom@15/dist/react-dom.min.js"></script>
```

Don't forget to update the version number according. We may also use *Bower* to manage such package installs and requirements, which is similar to patterns we use for AngularJS, for example.

part 2

For earlier versions of React and JSX, we would pre-compile JSX into JavaScript before deploying our application.

Using React's JSXTransformer option allowed us to compile and monitor JSX as we developed and tested our React app.

As React has evolved over the year or so, we still effectively follow this underlying concept and process. So, instead of using a custom JSXTransformer, we may now use *Babel's* in-browser JSX transformer for explicit ES6 support where required.

Effectively, Babel will add a check to our app to allow us to use JSX syntax, which is then understood by the browser.

This dynamic transformation works well for most test scenarios, but it is preferable to pre-compile for production apps. This should help to make an app faster for production usage.

JSX

JSX stands for **JavaScript XML**. It follows an XML familiar syntax for developing markup within components in React.

NB: JSX is not compulsory within React, but inherently it makes components easier to read and understand. Its structure is more succinct and less verbose.

A few defining characteristics of JSX are as follows,

- 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, instead relying upon simple function calls.

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. These benefits can include the following,

- 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

composite components

Let us now look at some example composite components using React.

custom component definition

An 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>
    );
  }
});
```

NB: this component is hard coded to simply output the specified heading 'Hello World'.

So, we can now update this example to work with dynamic values. JSX considers values dynamic if they are placed between curly brackets `{..}` . Effectively, these curly brackets are treated as a JavaScript context, which means content will be evaluated and the returned results rendered as nodes in the markup.

For example, for text, numbers etc we can simply refer to the appropriate variable for that value.

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

more dynamic values

We can also call functions, which allows us to move a lot of the logic for the component to a standard JavaScript function. We can then call this function, plus any supplied parameters, within the curly brackets of the React component.

React can also evaluate arrays, and then output each value. For example,

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

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

conditionals

A component's markup and its logic are inherently linked in React. This naturally includes *conditionals*, *loops* etc. However, adding `if` statements directly to JSX will create invalid JavaScript. Therefore, we can use the following options,

- ternary operator

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

- variable

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

- function call

```
getIsComplete: function() {
  return this.state.isComplete ? 'is-complete' : '';
},
render: function() {
```

```
    return <div className={this.getIsComplete()}>...</div>;
}
```

- double && operator
  To handle React's lack of output for *null* or *false* values, we can use a boolean value and follow it with the desired output.

non-DOM attributes

In JSX, there are currently three special attribute names,

- `key`
- `ref`
- `dangerouslySetInnerHTML`

`key`

In React, this is an optional unique identifier that remains consistent throughout render passes. Effectively, it informs React so it can more efficiently select when to reuse or destroy a component. Naturally, this helps improve the rendering performance of the application.

For example, if two elements already in the DOM need to switch position, React is able to match the keys and move them without any unnecessary re-rendering of the complete DOM.

`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>;
}
```

Later, you are able to access this `ref` using the defined `this.refs.myInput` anywhere in the component. The object accessed through this `ref` is known as a *backing instance*.

NB: this is not the actual DOM. Instead, it is a description of the component React uses to create the DOM when necessary.

To access the DOM itself for this `ref`, use `this.refs.myInput.getDOMNode()`, where *myInput* is the name of the previously defined ref.

`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>;
}
```

reserved words

JSX transforms to plain JavaScript functions, which means there are some reserved or special attributes. For example, we can't use `class` or `for`.

Therefore, to create a form label with the `for` attribute we can use `htmlFor` instead. eg:

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

To create a custom class we can use `className`. eg:

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

data flow

Data flows in one direction in React, namely from parent to child. This helps to make components nice and simple, and predictable as well.

In essence, 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, and then re-render any components that used that property.

Internal state also exists for each component, and should only be updated within the component itself.

`props`

Properties, or `props`, can hold any data and are passed to a component for usage. As developers, we can set `props` on a component during instantiation

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

We can 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 });
```

However, this option should only really be used to set `props` on a child component or outside of the component tree.

NB: we can access `props` via `this.props`, but it should not be used to write directly to `props`. In effect, a component should not modify its own props.

`props` **with JSX**

We can set the `props` using the `{}` syntax, which allows us to pass variables of any type via JavaScript injection.

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

We can 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...');
  }
});
```

State

A component in React is able to house *state*. *State* is inherently different from `props` because it is internal to the component. However, it is particularly useful for deciding a view state on an element.

For example, we could use *state* to track when to show certain options within a hidden list or menu. We can track the current state, change it relative to component requirements, and then show options based upon this amended state.

NB: it is considered bad practice to update state directly using `this.state`. Instead, we should use the method `this.setState`. For example,

As developers, we need to try to avoid storing computed values or components directly in *state*. Instead, we should be focusing upon using simple data directly required for the given component to function correctly.

It is considered good practice to perform required calculations in the `render` function.

We should also try to avoid duplicating `prop` data into `state`. Use the `props` data instead.

```
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 });
  }
});
```

When designing React apps, we often think about **stateless children** and a **stateful parent**. As noted in the React documentation,

> 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.

So, we need to carefully consider how to identify and implement this type of component hierarchy. We can consider,

stateless child components

These components should be passed data via `props` from the parent. More importantly, to remain *stateless* they should not manipulate their `state`.

Instead, they should send a callback to the parent informing it of a change, update etc. The parent will then decide whether it should result in a `state` change, and a re-rendering of the DOM.

stateful parent component

A *stateful* parent component can exist at any level of the hierarchy. It does not have to be the root component for the app, but instead can exist as a child to other parents.

We use the parent component to pass `props` to its children, and maintain and update state for the applicable components.

props vs state

In React, we can often consider two types of *model* data, which include `props` and `state`. Most components will normally take their data from `props`, which allow them to render the required data.

However, as we work with users, add interactivity, and query and respond to servers, we also need to consider the `state` of the application. Whilst `state` is very useful and important in React, it's also important to try and keep many of our components *stateless*.

state

React considers user interfaces, UIs, as simple state machines, acting in various states and then rendering as required. This means that in React, we simply update a component's state and then render the new corresponding UI.

Therefore, to make a UI interactive, we can trigger data changes using React's `state`.

concepts in state

how state works

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

In state

Try to keep data in `state` to a minimum, in effect considering the minimal possible representation of an application's state to help build a *stateful* component. Therefore, this `state` should try to just contain data that is 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`. Then, as we render the updated UI, we can simply compute any other required information in the `render()` method based on this `state`. This is beneficial as we can avoid the need to keep computed values in sync in state, instead relying on React to compute them for us.

Out of state

In React, `this.state` should only contain the minimal data necessary to represent an application's UI state. Therefore, this should not contain the following

- computed value
- React components
- duplicated data from `props`

State - a basic example

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

We can start with our test JSON,

```
//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'}
];
```

We'll start with some static data to help populate our app. `categoryId` will be used to filter unique categories, and again to help get all of our authors per category.

For `stateless` child components, we need to be able to output a list of filtered, unique categories, and then a list of authors for each selected category.

Our first child component is the `CategoryList`, which filters and renders our list of unique categories. The `onClick` attribute is included, and state is therefore passed via callback to the `stateful` parent.

```
//output unique categories from passed data...
var CategoryList = React.createClass({
 render: function() {
  var category = [];
   return (
    <div id="left-titles" className="col-md-6 col-sm-6 col-xs-6">
     <ul>
      {this.props.data.map(function(item) {
        //plain JS - no support for legacy browsers...
        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>
   );
  }
});
```

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

We now need to consider our second `stateless` child component, which renders the user's chosen authors per category. In effect, a user clicks on their chosen category, and 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>
    );
  }
});
```

Again, this component does not set any state. It is simply rendering the passed `props` data for viewing.

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, and 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.

In this case, we only need to store the `selectedCategoryAuthors` in `state` to be able to update the UI for our app.

```
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)
    });
  }
});
```

Our `stateful` parent component sets its initial state, including passed data and the app's selected category for authors. This helps set a default state for the app, which we can then modify as a user selects their chosen category.

The callback for this user selected category is handled in the `onCategorySelected` method, which updates the app's state for the chosen `categoryId`. This then leads to the app re-rendering the DOM for any changes.

However, 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...

Therefore, we should now move our computations to the `render` method of the parent component, and update `state` accordingly.

```
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, which can be modified and the DOM re-rendered correctly.

We can then load this application, passing data as props to the `Container`.

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

- DEMO - state example

Minimal state

As noted above, to help make our UI interactive we can use React's `state` to trigger changes to the underlying data model of an application. We need to keep a minimal set of mutable state. **DRY**, or *don't repeat yourself*, is often cited as a good rule of thumb for this minimal set.

Effectively, we need to decide upon an absolute minimal representation of the `state` of the application, and then compute everything else as required. For example, if we maintain an array of items, it is React common practice to calculate the length of this array as needed instead of maintaining a counter.

As we develop an application, we should start dividing our data into logical pieces. We can 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

Again, if data is parred as `props` it is not `state` in React.

We also need to decide upon our minimal set of components that mutate, or *own* state. As React is based on the premise of one-way data flow down the hierarchy of components, this can often be quite tricky to determine. To helps us, we can initially 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 at the top of the state hierarchy

Component lifecycle

React components include a minimal lifecycle API. This provides the developer with enough without being overwhelming, at least in theory.

React provides what are known as *will* and *did* methods. These work as follows,

- *will* - called right before something happens
- *did* - called right after something happens

Relative to the lifecycle, we can consider the following methods

- Instantiation (mounting)
- Lifetime (updating)
- Teardown (unmounting)
- Anti-pattern - calculated values

method groupings - Instantiation (mounting)

instantiation (mounting)

This includes methods called upon instantiation, and they will be called initially in order for the selected component class. For example, we can `getDefaultProps` or `getInitialState`. So, we can use such methods to set default values for new instances or initialise a custom state of each instance, and so on. We also have the important `render` method itself, which builds our application's virtual DOM. It is also the only required method for a component. There are a number of rules this method needs to follow, such as accessible data, return values and, importantly, must remain `pure`. In effect, it cannot change the state or modify the DOM output. The returned result is the virtual DOM, which is compared with the actual DOM to determine if changes are required for the application.

method groupings - lifetime (updating)

At this stage, the component has now been rendered to the user for viewing and interaction. So, as a user interacts with the component, they are changing the state of that component or application, which allows us as developers to act on the relevant points in the component tree. State changes for the application, which affect the component, may result in update methods being called. In effect, we're telling the component how and when to update.

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, which provides an opportunity to perform any cleanup and teardown as necessary.

- `componentWillUnmount`
  - as the life of a component ends, it is removed from the component hierarchy. This method gives us a chance to clean up just before the component is removed. For example, any custom work performed during the component's instantiation needs to be undone here.

method groupings - anti-pattern (calculated values)

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

Additional reading, material, and samples

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

References

- React
  - Homepage
  - API Reference
  - Interactive and Dynamic UIs
  - Starter Kit