Extra notes - Client-side Design and Development

- Dr Nick Hayward

JSX

A quick introduction to JSX. Further details on React 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.

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.

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...');
  }
});
```

References

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