CODE > WEB DEVELOPMENT

Intro to the React Framework

by Pavan Podila 15 Nov 2013

Difficulty: Intermediate Length: Long Languages: English

Web Development React JavaScript Node.js

In today's world of Javascript Application frameworks, design philosophy is the key differentiating factor. If you compare the popular JS frameworks, such as EmberJS, AngularJS, Backbone, Knockout, etc. you are sure to find differences in their abstractions, thinking models, and of course the terminology. This is a direct consequence of the underlying design philosophy. But, in principle, they all do one thing, which is to abstract out the DOM in such a way that you don't deal directly with HTML Elements.

I personally think that a framework becomes interesting when it provides a set of abstractions that enable a different mode of thinking. In this aspect, react, the new JS framework from the folks at Facebook, will force you to rethink (to some extent) how you decompose the UI and interactions of your application. Having reached version 0.4.1 (as of this writing), React provides a surprisingly simple, yet effective model for building JS apps that mixes a delightful cocktail of a different kind.

In this article, we'll explore the building blocks of React and embrace a style of thinking that may seem counter-intuitive on the first go. But, as the React docs say: "Give it Five Minutes" and then you will see how this approach will become more natural.

Motivations

The story of React started within the confines of Facebook, where it brewed for a while. Having reached a stable-enough state, the developers decided to open-source it a few months back. Interestingly the Instagram website is also powered by the React Framework.

React approaches the DOM-abstraction problem with a slightly different take. To understand how this is different, lets quickly gloss over the techniques adopted by the frameworks I mentioned earlier.

A High Level Overview of JS Application Frameworks

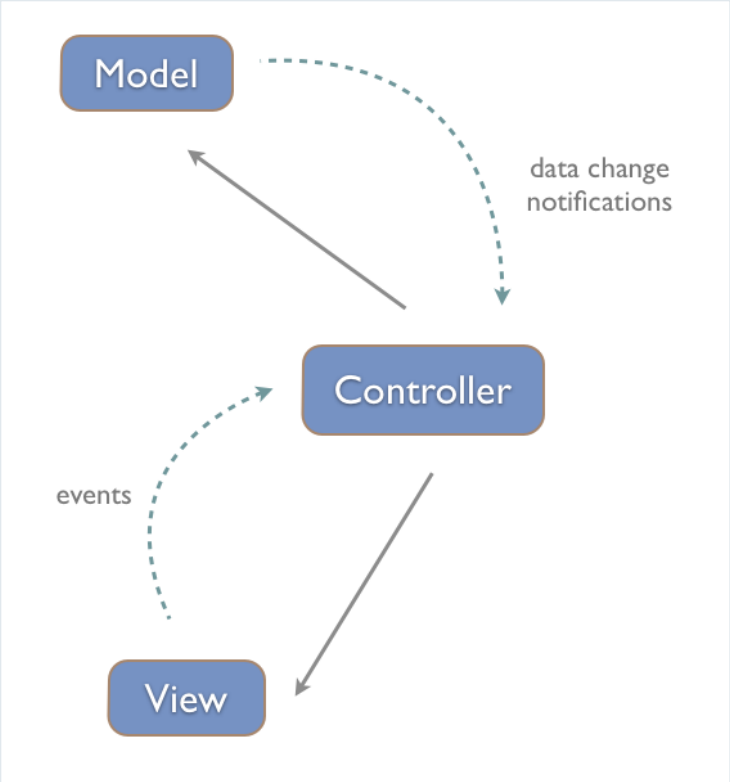
The MVC (Model-View-Controller) design pattern is fundamental to UI development, not just in web apps, but in front-end applications on any platform. In case of web apps, the DOM is the physical representation of a View. The DOM itself is generated from a textual html-template that is pulled from a different file, script-block or a precompiled template function. The `View` is an entity that brings the textual template to life as a DOM fragment. It also sets up event-handlers and takes care of manipulating the DOM tree as part of its lifecycle.

For the `View` to be useful, it needs to show some data, and possibly allow user interaction. The data is the `Model`, which comes from some data-source (a database, web-service, local-storage, etc.). Frameworks provide a way of "binding" the data to the view, such that

changes in data are automatically reflected with changes on the view. This automatic process is called *data-binding* and there are APIs/techniques to make this as seamless as possible.

The MVC triad is completed by the `Controller`, which engages the `View` and the `Model` and orchestrates the flow of data (`Model`) into

the `View` and user-events out from the `View`, possibly leading to changes in the `Model`.



Frameworks that automatically handle the flow of data back and forth between the View and Model maintain an internal event-loop. This event-loop is needed to listen to certain user events, data-change events, external triggers, etc and then determine if there is any change from the previous run of the loop. If there are changes, at either end (View or Model), the framework ensures that both are brought back in sync.

What Makes React Different?

With React, the View-part of the MVC triad takes prominence and is rolled into an entity called the `Component`. The Component maintains an immutable property bag called `props`, and a `state` that represents the user-driven state of the UI. The view-generation part of the `Component` is rather interesting and possibly the reason that makes React stand out compared to other frameworks. Instead of constructing a physical DOM directly from a template file/script/function, the `Component` generates an intermediate DOM that is a stand-in for the real HTML DOM. An additional step is then taken to translate this intermediate DOM into the real HTML DOM.

As part of the intermediate DOM generation, the `Component` also attaches event-handlers and binds the data contained in `props` and `state`.

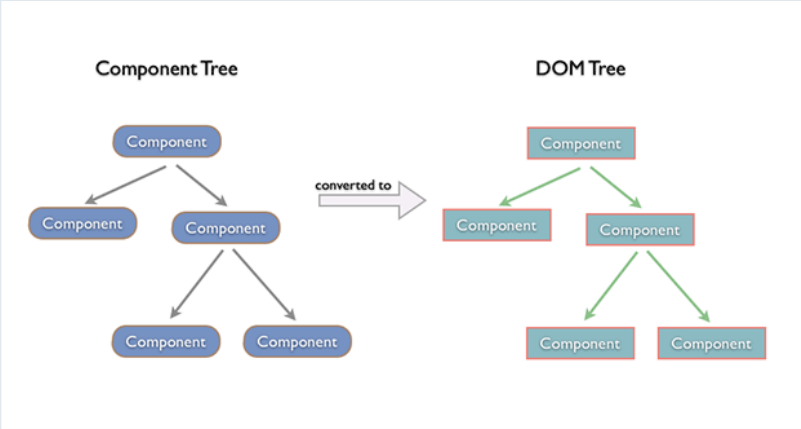
If the idea of an intermediate-DOM sounds a little alien, don't be too alarmed. You have already seen this strategy adopted by language runtimes (aka Virtual Machines) for interpreted languages. Our very own JavaScript runtime, first generates an intermediate representation before spitting out the native code. This is also true for other VM-based languages such as Java, C#, Ruby, Python, etc.

React cleverly adopts this strategy to create an intermediate DOM before generating the final HTML DOM. The intermediate-DOM is just a JavaScript object graph and is not rendered directly. There is a translation step that creates the real DOM. This is the underlying technique that makes React do fast DOM manipulations.

React In Depth

To get a better picture of how React makes it all work, let's dive a little deeper; starting with the `Component`. The Component is the primary building block in React. You can compose the UI of your application by assembling a tree of Components. Each Component

primary building block in React. You can compose the UI of your application by assembling a tree of Components. Each Component provides an implementation for the `render()` method, where it creates the intermediate-DOM. Calling `React.renderComponent()` on the root Component results in recursively going down the Component-tree and building up the intermediate-DOM. The intermediate-DOM is then converted into the real HTML DOM.



Since the intermediate-DOM creation is an integral part of the Component, React provides a convenient XML-based extension to JavaScript, called JSX, to build the component tree as a set of XML nodes. This makes it easier to visualize and reason about the DOM. JSX also simplifies the association of event-handlers and properties as xml attributes. Since JSX is an extension language, there is a tool (command-line and in-browser) to generate the final JavaScript. A JSX XML node maps directly to a Component. It is worth pointing out that React works independent of JSX and the JSX language only makes it easy to create the intermediate DOM.

Tooling

The core React framework can be downloaded from their website. Additionally, for the JSX → JS transformation, you can either use the in-browser JSXTransformer or use the command line tool, called react-tools (installed via NPM). You will need an installation of Node.js to download it. The command-line tool allows you to precompile the JSX files and avoid the translation within the browser. This is definitely recommended if your JSX files are large or many in number.

A Simple Component

Alright, we have seen a lot of theory so far, and I am sure you are itching to see some real code. Let's dive into our first example:

```
/** @jsx React.DOM */

var Simple = React.createClass({

  getInitialState: function(){
    return { count: 0 };
  },

  handleMouseDown: function(){
    alert('I was told: ' + this.props.message);
    this.setState({ count: this.state.count + 1});
  },

  render: function(){

    return <div>

      <div class="clicker" onMouseDown={this.handleMouseDown}>
        Give me the message!
      </div>
      <div class="message">Message conveyed
```

```
        <span class="count">{this.state.count}</span> time(s)</div>
    </div>
    ;
  }
});

React.renderComponent(<Simple message="Keep it Simple"/>,
                  document.body);
```

Although simple, the code above does cover a good amount of the React surface area:

- We create the Simple component by using `React.createClass` and passing in an object that implements some core functions. The most important one is the `render()`, which generates the intermediate-DOM.
- Here we are using JSX to define the DOM and also attach the mousedown event-handler. The `{}` syntax is useful for incorporating JavaScript expressions for attributes (`onMouseDown={this.handleClick}`) and child-nodes (`<span class="count">{this.state.count}</span>`). Event handlers associated using the {} syntax are automatically bound to the instance of the component. Thus `this` inside the event-handler function refers to the component instance. The comment on the first line `/** @jsx React.DOM */` is a cue for the JSX transformer to do the translation to JS. *Without this comment line, no translation will take place*.

We can run the command-line tool (*jsx*) in watch mode and auto-compile changes from JSX → JS. The source files are in */src* folder and the output is generated in */build*.

```
1 jsx --watch src/ build/
```

Here is the generated JS file:

```
/** @jsx React.DOM */

var Simple = React.createClass({displayName: 'Simple',

  getInitialState: function(){
    return { count: 0 };
  },

  handleMouseDown: function(){
    alert('I was told: ' + this.props.message);
    this.setState({ count: this.state.count + 1});
  },

  render: function(){

    return React.DOM.div(null,
      React.DOM.div( {className:"clicker", onMouseDown:this.handleMouseDown},
" Give me the message! "      ),
      React.DOM.div( {className:"message"}, "Message conveyed ",        React.DOM.span( {className:"c
    )
    ;
  }
});

React.renderComponent(Simple( {message:"Keep it Simple"}),
                  document.body);
```

Notice how the `<div/>` and `<span/>` tags map to instances of `React.DOM.div` and `React.DOM.span`.

- Now let's get back to our code example. Inside `handleMouseDown`, we make use of `this.props` to read the *message* property that was passed in. We set the *message* on the last line of the snippet, in the call to `React.renderComponent()` where we create the `<Simple/>` component. The purpose of `this.props` is to store the data that was passed in to the component. It is considered immutable and only a higher-level component is allowed to make changes and pass it down the component tree.
- Inside `handleMouseDown` we also set some user state with `this.setState()` to track the number of times the message was displayed. You will notice that we use `this.state` in the `render()` method. Anytime you call `setState()`, React also triggers the `render()` method to keep the DOM in sync. Besides `React.renderComponent()`, `setState()` is another way to force a visual refresh.

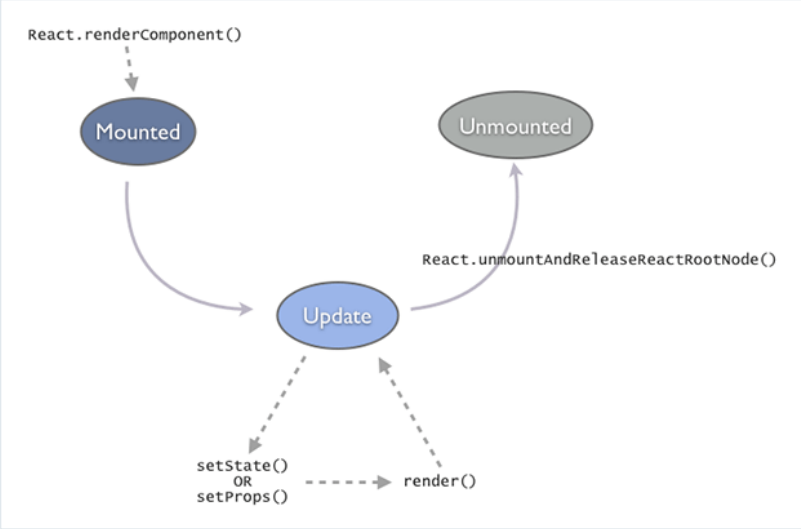
Synthetic Events

The events exposed on the intermediate-DOM, such as the `onMouseDown`, also act as a layer of indirection before they get set on the real-DOM. These events are thus refered to as *Synthetic Events*. React adopts event-delegation, which is a well-known technique, and attaches events only at the root-level of the real-DOM. Thus there is only one true event-handler on the real-DOM. Additionally these synthetic events also provide a level of consistency by hiding browser and element differences.

The combination of the intermediate-DOM and synthetic events gives you a standard and consistent way of defining UIs across different browsers and even devices.

Component Lifecycle

Components in the React framework have a specific lifecycle and embody a state-machine that has three distinct states.



The Component comes to life after being *Mounted*. Mounting results in going through a render-pass that generates the component-tree (intermediate-DOM). This tree is converted and placed into a container-node of the real DOM. This is a direct outcome of the call to `React.renderComponent()`.

Once mounted, the component stays in the *Update* state. A component gets updated when you change state using `setState()` or change props using `setProps()`. This in turn results in calling `render()`, which brings the DOM in sync with the data (`props` + `state`). Between subsequent updates, React will calculate the delta between the previous component-tree and the newly generated tree. This is a highly optimized step (and a flagship feature) that minimizes the manipulation on the real DOM.

The final state is *Unmounted*. This happens when you explicitly call `React.unmountAndReleaseReactRootNode()` or automatically if a component was a child that was no longer generated in a `render()` call. Most often you don't have to deal with this and just let React do the proper thing.

Now it would have been a big remiss, if React didn't tell you when it moved between the *Mounted-Update-Unmounted* states. Thankfully that is not the case and there are hooks you can override to get notified of lifecycle changes. The names speak for themselves:

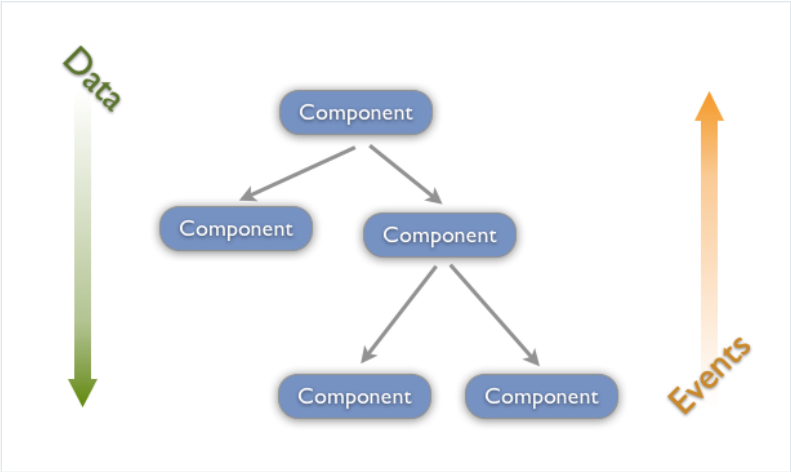
- `getInitialState()`: prepare initial state of the Component
- `componentWillMount()`
- `componentDidMount()`
- `componentWillReceiveProps()`
- `shouldComponentUpdate()`: useful if you want to control when a render should be skipped.
- `componentWillUpdate()`
- `render()`
- `componentDidUpdate()`
- `componentWillUnmount()`

The `componentWill*` methods are called before the state change and the `componentDid*` methods are called after.

Some of the method names do seem to have taken a cue from the Cocoa frameworks in Mac and iOS

Miscellaneous Features

Within a component-tree, data should always flow down. A parent-component should set the `props` of a child-component to pass any data from the parent to the child. This is termed as the *Owner-Owned* pair. On the other hand user-events (mouse, keyboard, touches) will always bubble up from the child all the way to the root component, unless handled in between.



When you create the intermediate-DOM in `render()`, you can also assign a `ref` property to a child component. You can then refer to it from the parent using the `refs` property. This is depicted in the snippet below.

```
render: function(){
  // Set a ref
  return <div>
      <span ref="counter" class="count">{this.state.count}</span>
      </div>;
}

handleMouseDown: function(){
  // Use the ref
  console.log(this.refs.counter.innerHTML);
},
```

As part of the component metadata, you can set the initial-state (`getInitialState()`), which we saw earlier within the lifecycle methods. You can also set the default values of the props with `getDefaultProps()` and also establish some validation rules on these props using `propTypes`. The docs give a nice overview of the different kinds of validations (type checks, required, etc.) you can perform.

React also supports the concept of a *Mixin* to extract reusable pieces of behavior that can be injected into disparate Components. You can pass the mixins using the `mixins` property of a Component.

Now, lets get real and build a more comprehensive Component that uses these features.

A Shape Editor Built Using React

In this example, we will build an editor that accepts a simple DSL (Domain Specific Language) for creating shapes. As you type in, you will see the corresponding output on the side, giving you live feedback.

The DSL allows you to create three kinds of shapes: Ellipse, Rectangle and Text. Each shape is specified on a separate line along with a bunch of styling properties. The syntax is straightforward and borrows a bit from CSS. To parse a line, we use a Regex that looks like:

```
  var shapeRegex = /(rect|ellipse|text)(\s[a-z]+:\s[a-z0-9]+;)*/i;
```

As an example, the following set of lines describe two rectangles and a text label...

```
// React label
text value:React; color: #00D8FF; font-size: 48px; text-shadow: 1px 1px 3px #555; padding: 10px; left

// left logo
rect background:url(react.png) no-repeat; border: none; width: 38; height: 38; left: 60px; top: 120px

// right logo
rect background:url(react.png) no-repeat; border: none; width: 38; height: 38; left: 250px; top: 120p
```

...generating the output shown below:

Setting Up

Alright, lets go ahead and build this editor. We will start out with the HTML file (`index.html`), where we put in the top-level markup and include the libraries and application scripts. I am only showing the relevant parts here:

```
<body>
<select class="shapes-picker">
  <option value="--">-- Select a sample --</option>
  <option value="react">React</option>
  <option value="robot">Robot</option>
</select>
<div class="container"></div>

<!-- Libraries -->
<script src="../../lib/jquery-2.0.3.min.js"></script>
<script src="../../lib/react.js"></script>


<!-- Application Scripts -->
<script src="../../build/shape-editor/ShapePropertyMixin.js"></script>
<script src="../../build/shape-editor/shapes/Ellipse.js"></script>
<script src="../../build/shape-editor/shapes/Rectangle.js"></script>
<script src="../../build/shape-editor/shapes/Text.js"></script>

<script src="../../build/shape-editor/ShapeParser.js"></script>
<script src="../../build/shape-editor/ShapeCanvas.js"></script>
<script src="../../build/shape-editor/ShapeEditor.js"></script>

```

```
<script src="../../build/shape-editor/shapes.js"></script>
<script src="../../build/shape-editor/app.js"></script>
</body>
```

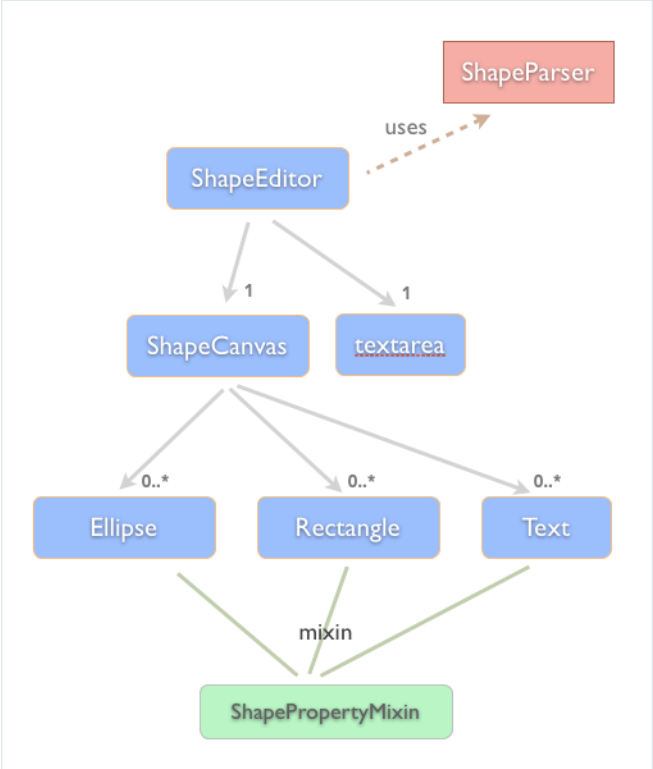
In the above snippet, the `container` div holds our React generated DOM. Our application scripts are included from the `/build` directory. We are using JSX within our components and the command line watcher (`jsx`), puts the converted JS files into `/build`. Note that this watcher command is part of the `react-tools` NPM module.

```
jsx --watch src/ build/
```

The editor is broken down into a set of components, which are listed below:

- **ShapeEditor**: the root Component in the component tree
- **ShapeCanvas**: responsible for generating the shape-Components (Ellipse, Rectangle, Text). It is contained within the ShapeEditor.
- **ShapeParser**: responsible for parsing text and extracting the list of shape definitions. It parses line by line with the Regex we saw earlier. Invalid lines are ignored. This is not really a component, but a helper JS object, used by the ShapeEditor.
- **Ellipse, Rectangle, Text**: the shape Components. These become children of the ShapeCanvas.
- **ShapePropertyMixin**: provides helper functions for extracting styles found in the shape definitions. This is mixed-into the three shape-Components using the `mixins` property.
- **app**: the entry-point for the editor. It generates the root component (ShapeEditor) and allows you to pick a shape sample from the drop-down.

The relationship of these entities is shown in the annotated component-tree:



The ShapeEditor Component

Lets look at the implementation of some of these components, starting with the ShapeEditor.

```
/** @jsx React.DOM */
var ShapeEditor = React.createClass({

  componentWillMount: function () {
```

```
    this._parser = new ShapeParser();
  },

  getInitialState: function () {
    return { text: '' };
  },

  render: function () {
    var shapes = this._parser.parse(this.state.text);

    var tree = (
      <div>
        <textarea class="editor" onChange={this.handleTextChange} />
        <ShapeCanvas shapes={shapes} />
      </div>);

    return tree;
  },

  handleTextChange: function (event) {
    this.setState({ text: event.target.value })
  }

});
```

As the name suggests, the ShapeEditor provides the editing experience by generating the `<textarea/>` and the live feedback on the `<ShapeCanvas/<`. It listens to the `onChange` event (events in React are always named with camel case) on the `<textarea/>` and on every change, sets the `text` property of the component's `state`. As mentioned earlier, whenever you set the state using `setState()`, render is called automatically. In this case, the `render()` of the ShapeEditor gets called where we parse the text from the state and rebuild the shapes. Note that we are starting with an initial state of empty text, which is set in the `getInitialState()` hook.

For parsing the text into a set of shapes, We use an instance of the `ShapeParser`. I've left out the details of the parser to keep the discussion focused on React. The parser instance is created in the `componentWillMount()` hook. This is called just before the component mounts and is a good place to do any initializations before the first render happens.

It is generally recommended that you funnel all your complex processing through the `render()` method. Event handlers just set the state while `render()` is the hub for all your core logic.

The `ShapeEditor` uses this idea to do the parsing inside of its `render()` and forwards the detected shapes by setting the `shapes` property of the `ShapeCanvas`. This is how data flows down into the component tree, from the owner (`ShapeEditor`) to the owned (`ShapeCanvas`).

One last thing to note in here is that we have the first line comment to indicate JSX → JS translation.

ShapeCanvas to Generate the Shapes

Next, we will move on to the ShapeCanvas and the Ellipse, Rectangle and Text components.

p>The `ShapeCanvas` is rather straightforward with its core responsibility of generating the respective `<Ellipse/>`, `<Rectangle/>` and `<Text/>` components from the passed in shape definitions (`this.props.shapes`). For each shape, we pass in the parsed properties with the attribute expression: `properties={shape.properties}`.

```
/** @jsx React.DOM */
var ShapeCanvas = React.createClass({

  getDefaultProps: function(){
    return {
      shapes: []
    };
  },

  render: function () {
    var self = this;
    var shapeTree = <div class="shape-canvas">
    {
      this.props.shapes.map(function(s) {
        return self._createShape(s);
      })
    }
      </div>;

    var noTree = <div class="shape-canvas no-shapes">No Shapes Found</div>;

    return this.props.shapes.length > 0 ? shapeTree : noTree;
  },

  _createShape: function(shape) {
    return this._shapeMap[shape.type](shape);
  },

  _shapeMap: {
    ellipse: function (shape) {
      return <Ellipse properties={shape.properties} />;
    },

    rect: function (shape) {
      return <Rectangle properties={shape.properties} />;
    },

    text: function (shape) {
      return <Text properties={shape.properties} />;
    }
  }

});
```

One thing different here is that our component tree is not static, like we have in ShapeEditor. Instead it's dynamically generated by looping over the passed in shapes. We also show the `"No Shapes Found"` message if there is nothing to show.

The Shapes: Ellipse, Rectangle, Text

All of the shapes have a similar structure and differ only in the styling. They also make use of the `ShapePropertyMixin` to handle the style generation.

Here's Ellipse:

```
/** @jsx React.DOM */

var Ellipse = React.createClass({


  mixins: [ShapePropertyMixin],

  render:function(){
    var style = this.extractStyle(true);
    style['border-radius'] = '50% 50%';

    return <div style={style} class="shape" />;
  }
});
```

The implementation for `extractStyle()` is provided by the `ShapePropertyMixin`.

The Rectangle component follows suit, of course without the *border-radius* style. The Text component has an extra property called `value` which sets the inner text for the `<div/>`.

Here's Text, to make this clear:

```
/** @jsx React.DOM */

var Text = React.createClass({

  mixins: [ShapePropertyMixin],

  render:function(){
    var style = this.extractStyle(false);
    return <div style={style} class="shape">{this.props.properties.value}</div>;
  }

});
```

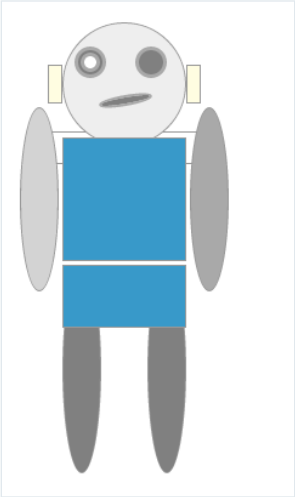
Tying It All Together With App.js

`app.js` is where we bring it all together. Here we render the root component, the `ShapeEditor` and also provide support to switch between a few sample shapes. When you pick a different sample from the drop down, we load some predefined text into the `ShapeEditor` and cause the `ShapeCanvas` to update. This happens in the `readShapes()` method.

```
/** @jsx React.DOM */

var shapeEditor = <ShapeEditor />;
React.renderComponent(
  shapeEditor,
  document.getElementsByClassName('container')[0]
);


function readShapes() {
  var file = $('.shapes-picker').val(),
    text = SHAPES[file] || '';
```

```
  $('.editor').val(text);
  shapeEditor.setState({ text: text }); // force a render
}

$('.shapes-picker').change(readShapes);
readShapes(); // load time
```

To exercise the creative side, here is a robot built using the Shape Editor:



And That's React for you!

Phew! This has been a rather long article and having reached this point, you should have a sense of achievement!

We have explored a lot of concepts here: the integral role of Components in the framework, use of JSX to easily describe a component tree (aka intermediate-DOM), various hooks to plug into the component lifecyle, use of `state` and `props` to drive the render process, use of Mixins to factor out reusable behavior and finally pulling all of this together with the Shape Editor example.

I hope this article gives you enough boost to build a few React apps for yourself. To continue your exploration, here are few handy links:

- Docs
- Source on Github
- Blog