Chapter 2: React at a High Level

What Is React?

React is an open-source JavaScript library for creating user interfaces. It aims to address the problems of building large applications with data that changes over time. Originally developed at Facebook, it is now used in a number of companies including Netflix, Instagram, Airbnb, and HelloSign.

React is only concerned with rendering the UI from a data model and is mostly not opinionated about how you structure your applications, leaving that up to the developer. It is mostly unconcerned with how data is fetched and handled. That makes React extremely flexible. You can use it with other unrelated libraries/frameworks like Backbone and Angular, with Flux[1], or with your own custom solution. It is very simple yet incredibly powerful. Its simplicity makes it easy to pick up in a short amount of time.

How React Views the World

React's beauty is in it's philosophy toward applicaiton development. React introduces a whole new set of ideas to web development which go against many established best practices, which we as an industry have come to accept in front-end web development. Its approach is very different to what we have seen on the web so far. React encourages developers to move away from MVC (Model-View-Controller) and instead adopt a UI component based approach. To React, the client should not be thought of in terms of MVC, but rather a collection of nested UI components.

While MVC JavaScript applications can be described in terms of Object Oriented Programming, React moves more toward a functional approach based user interfaces. Just like simple reusable functions, React helps organize our applications into reusable UI components. That is why React is primarily concerned with an application's presentation layer since that is what should drive an application's design. By taking this approach, React design decision helps simplify the problem of managing state in large scale web applications.

React Is Not MVC

The fact that React only cares about presenting the UI has led some to conclude that React should be considered the "V" in "MVC", which is somewhat misleading. While you *could* use React as the

[1]https://facebook.github.io/flux/docs/overview.html

view layer of an MVC architecture, that would negate many of the benefits of using React. It is much more than just a templating library. It is a completely different way of building web applications.

Its philosophy mostly rejects MVC/MV* in the browser. That is because the browser environment is very different from the server, where we traditionally use MVC. On the server, MVC is an excellent idea. The server's job is to create state out of the stateless environment of HTTP. In such an environment, MVC makes a lot of sense. Given the success of MVC on the server, it is no wonder why we have been trying to use it on the browser.

These days, web applications are more than just a server serving static HTML and CSS. With the vast improvement of browser JavaScript engines along with the rise of excellent user experiences found in native applications, we as an industry have increasingly started pushing data as well as behavior to the client. This was a great step forward for the web, but as web developers this left us scrambling to find ways to work in the hostile browser environment.

We understandably took the ideas that served us well on the server, such as MVC, and have attempted to apply them to the browser. If MVC helped separate concerns on the server, it seems logical to use it to separate concerns in the browser. At the time, this seemed to make a lot of sense and indeed was a great improvement for client-side JavaScript. While they have served a very useful purpose, MVC frameworks are not the end of history for building web applications. Much like how jQuery helped move the web forward, MVC frameworks helped the push to build rich user experiences for the web but is not the final word in that space.

Not all web developers have adopted the idea that client-side web applications are the future or even useful. This is because development in MVC JavaScript frameworks is painfully slow and complicated, due to the application of an inappropriate paradigm to a different environment. The unique challenges presented by browser side JavaScript require a different set of ideas to tackle them effectively. The problem with forcing MVC on the browser is; it attempts to separate concerns that are inextricably linked.

How to Actually Separate Concerns: Write Components, Not Templates

MVC makes sense in the stateless environment of HTTP and webservers but it makes less sense in single page applications. That is because unlike the server environment, the browser is full of state. The server is concerned with requests and responses, but the browser is a user interface with a lot of events and user interaction. MVC on the browser tries to separate things that actually belong together, which in turn obscures the state of an application.

One thing JavaScript MVC frameworks try to separate which actually belong together is display logic and markup. This is a mistake. React thinks that you should stop writing templates: you should be writing components. This means that we need to stop separating out HTML from our JavaScript.

This goes against everything we have come to believe in the JavaScript community. In fact, the above statement is enough for many to dismiss React. For just a moment, let us set dogma aside and analyze the issue as if approaching it for the first time.

React asserts markup and display logic are inextricably linked. Whenever you change markup, you are bound to affect the behavior you wrote in JavaScript. And when you change the behavior, you are undoubtedly going to update the markup. That is not always the case, but that is a concern. This looks a lot like tight coupling in a system. By trying to separate a UI between templates and controllers/view-models, we are not separating concerns â€“ we are separating technologies.

Before we continue, we should define what "separation of concerns" means. Separation of concerns is really about reducing coupling while simultaneously increasing cohesion. Coupling is how much pieces of a system depend on each other. Cohesion refers to the degree to which elements of a system belong together. Templates and display logic have implicit agreements among themselves, meaning; if you change one, you might have to change the other, which leads to cascading changes. This is coupling. So by separating markup and display logic, you are actually increasing coupling throughout your system. Meanwhile, you are also decreasing the amount of cohesion in your application. Display logic and markup are very cohesive; they are both concerned with showing the UI to the user. By separating them, you are reducing cohesion and the developer's ability to reason about the structure of the application.

To be fair, seperating templates and display logic usually turns out fine. But there is a better approach. To truly separate concerns, you should be writing components. Frameworks cannot separate concerns for you. You can do a much better job of separating concerns yourself. You separate concerns by creating small components responsible for rendering just a small piece of the UI. Displaying a piece of the UI is now encapsulated in one place. Such a component is both decoupled from the rest of your system and cohesive.

In addition, small components are reusable, composable, and above all, unit testable. Essentially, components are idempotent functions that when given a set of data will always return the same UI result, making them much easier to test. React essentially introduced functional programing to the UI.

Let's take a look at what a component looks like in React. Here's a small example.

```
var HelloWorld = React.createClass({
  render: function() {
    return <h1 className="header">Hello, World!</h1>
  }
});
```

The first thing you will most likely notice is what looks like HTML in our JavaScript. It is not actually HTML. This is JSX. It is syntactic sugar on top of plain JavaScript that makes it easy to declare markup. When compiled to regular JavaScript, it will look like this:

```
var HelloWorld = React.createClass({
  render: function() {
    return React.createElement('h1', {className: 'header'}, 'Hello, World!')
  }
});
```

JSX is not needed to work with React, but using it makes it much easier to create components that are semantic and easy to understand at a glance. These days, most JavaScript projects are using some kind of build system and perhaps using that build system to use ES2015, so adding a small compilation step for JSX is trivial in comparison. In fact, Babel[2] supports JSX, which is arguably the most popular JavaScript compiler today.

We are passing a `render()` method in an object to `React.createClass` to create the `HelloWorld` component. `render()` is the most important method in React. It is the heart of a React component. `render()` gets called every time the React components gets rendered to HTML.

It is important to point out that `render()` does not return an `h1` – it will return an in memory representation of a DOM node that will eventually be used to create an in memory representation of the current state of the DOM, the Virtual DOM. More about that in a future chapter.

Now that we have this component, we can reuse it elsewhere:

```
var Header = React.createClass({
  render: function() {
    return (
      <div>
        <HelloWorld />
        <h2>React all the things!</h2>
      </div>
    );
  }
});
```

Above, we created a new `Header` component that is reusing the `HelloWorld` component. Instead of using a template partial, we can simply reuse components anywhere we like. We have the accessibility of templates with the power of JavaScript. You can write your whole app in JavaScript and function calls that look like markup, allowing you to clearly refactor without worrying about breaking your templates or vice versa.

But this is still a pretty small example without any data, state, or behavior. Let's dive into how React thinks about all those things.

[2]https://babeljs.io/

Data Binding

User interfaces are very difficult to build due to the amount of state found in it. State is the root of all evil, as many have observered[3]. All the state in user interfaces intertwines to create a complex system that is very difficult for humans to understand.

Why is state so hard to understand? Essentially, state is your data at any given point in time. In other words, state is data that changes over time – it is mutable. Human brains, as it turn out, are not very well geared toward reasoning about systems that change over time. We are masters of understanding static relationships and have no problem keeping those relationships in our brains. However, we have a really hard time at keeping track of dynamic relationships that change over time (see race condition[4]). When data is constantly in flux, we humans have a hard time trying to reason about exactly what is going on at any given point in time.

So how do we, as programmers, master state in user interfaces? Traditionally, we have done this via data binding. This helps to transform a dynamic process into a more static one. Data binding makes the UI look more like a static program relative to our data. In other words, data binding syncs UI state with a data model.

Data binding is one of many great ideas to come from the 1970's but in JavaScript it is not perfect. The problem with data binding in JavaScript is that it is a polyfill for reactive programming in the DOM. What do we mean by that?

Reactive programming deserves its own book, so we will not talk about it too much in this book. Reactive programming is a little difficult to grasp at first, but it is basically programming with asynchronous data streams. Unfortunately, JavaScript is not actually reactive. Take a look at this very simple example:

```
var a = 1;
var b = 1;
var c = a + b;
var a = 2;
```

In the above example, `c` will still equal 2 even after we change the value of `a`. If JavaScript were truly reactive, the value of `c` would be automatically be in sync with the sum of `a` and `b`. This is a very simple example. If you want to learn more about reactive programming take a look at this excellent resource: The Introduction to Reactive Programming You've Been Missing[5].

Needless to say, this is not necessarily a mistake or bug in the language, but it does present an obstacle if we want a reactive implementation of JavaScript in the DOM. In order to achieve this, most JavaScript frameworks implement some form of data binding.

[3]https://twitter.com/steveklabnik/status/343179719136649216
[4]https://en.wikipedia.org/wiki/Race_condition
[5]https://gist.github.com/staltz/868e7e9bc2a7b8c1f754

This is not a trivial abstraction on top of JavaScript, and like all abstractions, these implementations leak. The difference is; how leaky are they, when do they leak, and how easy is it to spot and fix the leak?

Pretty much every JavaScript frameworks uses Key-Value Observation (KVO) for data binding (most prominently Ember and Meteor) except Angular (which uses dirty checking). We will not dive into too much detail into these systems in this book. The main problem with these systems is that they diverge too far away from something that looks a lot more like vanilla JavaScript. They also require deep understanding of the inner workings of these systems.

This is far from ideal. The ideal data binding system tries to stick to plain old JavaScript functions as much as possible, while giving you simple tools for reactivity. You should not have to think about how the data binding works. Leaks in the abstraction should be predictable every time and easy to solve.

React fits this description almost perfectly. Its data binding system is not perfect, but those leaks are very well understood and predictable.

Virtual DOM

React's approach to data binding relies on much simpler abstractions: the Virtual DOM. Essentially, React's Virtual DOM system abstracts the DOM by keeping a virtual representation of the DOM in memory and whenever the data model changes, React triggers a re-render of the components that rely on that data.

At a high level, React give you tools to describe what your component should look like at any given time. So, whenever the state changes, React will begin re-rendering the component that contains that state, diff the previous virtual representation of the DOM with the new virtual representation, and only update the DOM with what actually changed. You as a developer do not need to worry about how this really works, you just need to describe what your component looks like and let React know when your state has changed. That being said, we will dive deeper into React's Virtual DOM diff algorithm shortly.

Let's look at a very simple example and see how this works in detail. You can play the code here: Demo[6]

[6]http://codepen.io/anon/pen/VvrqxO

```
var Cat = React.createClass({
  getInitialState: function() {
    return { name: null };
  },

  render: function() {
    return (
      <div>
        <p>Your cat name is: {this.state.name}</p>
        <input type="text" onChange={this.updateName} />
      </div>
    );
  },

  updateName: function(event) {
    this.setState({name: event.target.value});
  }
});
```

You can figure out what this simple example does yourself without much need for explanation. First, we are setting the initial state of the component in `getInitialState`. You can think of this method like a constructor function for the component, where we are setting the default value of `name` to null. `getInitialState` is not required for every React component, just like a constructor function is not always needed to create a class in Object Oriented Programming.

Then we are describing what our component will look like in the `render()` method. We can access the current state of the component via `this.state`. Inside of `render()`, we are returning the current state of the cat name via `this.state.name`. We are also rendering an input field to the user to change the name of the cat.

Notice that we are adding an event handler to the input field using `onChange={this.updateName}`. Whenever there is a change to the input field, `updateName()` will be called. `updateName()` calls a special React method called `setstate()`. If `render()` can be considered the heart of a component, `setState()` can be considered the brain. `setState()` will update the value of the data in the components state, which in turn will trigger a re-render of the app.

If you type in the letter "M" into the input field, `updateName()` will be called, which will grab that value from the input field and update the state of the cat name with `setState()`. React will then call `render()` on `Cat` again with the new state. It will create two virtual representations of the DOM: the old DOM and the new DOM. It will batch all updates together and figure out the most optimal DOM mutations necessary to reach the new state of the UI.

By the time you are finished typing "Mittens", React would have triggered numerous re-renders on `Cat`. On the surface, this seems like it would be slow. In fact, this design decision makes React extremely fast. Because this is all done in JavaScript, this is all performed within one repaint of the browser. Even in the worst case scenario involving tens of thousands of DOM mutations, React will perform a re-render within the repaint time of the browser. This is due to the speed of modern JavaScript engines and React's clever diffing algorithm, which we will go into more detail shortly.

Data in React

In addition to the Virtual DOM, React treats data differently from other JavaScript projects like Angular or Ember. React separates data that changes over time (state) from data that stays the same. It does this via two concepts: `state` and `props`. `state` is data that will change over time such as user interaction.

Components should strive to contain as little state as possible. However, sometimes a component needs to respond to external events such as user interaction, a server request, or the passage of time. For that kind of data, components should keep that in `state`. For data that will not change, components should keep that in `props`.

React receive `props` from a parent component. By relying on props, a component essentially becomes an idempotent function. React sees components as functions, therefore it forces data to only flow from parent to child. Just like a function, data should flow into it as arguments. Data always flow just one way in React from parent to child.

Unidirectional data flow limits the messages and data being passed inside the system, making it easier to debug. Since data can only pass from parent to child, it is easy to trace the flow of data in your system. When you come across a bug, first you look at the component you think is causing the bug. If you can't find it there, you turn to its parent and so on until you find the culprit. In an MVC pattern, since messages travel all over the place, it is much more difficult to figure out the flow of your data.

Let's take a look at an example of `state`, `props`, and unidirectional data flow. You can follow along here: Demo[7].

```
var AnimalRescue = React.createClass({
  getInitialState: function() {
    return { catCount: 0, dogCount: 0 }
  },

  render: function() {
    return (
      <div>
        <ul>
          <Cats count={this.state.catCount}/>
          <Dogs count={this.state.dogCount}/>
```

[7]http://codepen.io/anon/pen/OyzowP

```
        </ul>
        <button onClick={this.moreAnimals}>Save more animals</button>
      </div>
    );
  },

  moreAnimals: function() {
    var catCount = this.state.catCount + 1;
    var dogCount = this.state.dogCount + 1;
    this.setState({
      catCount: catCount,
      dogCount: dogCount
    });
  }
});

var Cats = React.createClass({
  render: function() {
    return (
      <li># of Cats: {this.props.count}</li>
    );
  }
});

var Dogs = React.createClass({
  render: function() {
    return (
      <li># of Dogs: {this.props.count}</li>
    );
  }
});
```

In this example, the number of dogs and cats changes over time. Therefore, this data should be kept in `state`. `AnimalRescue` will be in charge of keeping this state, as well as rendering two child components: `Cats` and `Dogs` and a button to increment the number of animals. `getInitialState` is setting the initial values of `catCount` and `dogCount` to zero. Inside of the `render()` function of `AnimalRescue`, we are passing the current state of `catCount` and `dogCount` to its child components as `props`. We do this by declaring `<Cats count={this.state.catCount}/>`. Now the `Cat` component has access to that value as `props`. From the point of view of the child components, these values are like function arguments. As far as they are concerned, this data is never going to change. They should be treated as immutable inside of the child components.

Notice how in the parent component we are calling this value `catCount` but we are calling it `count` in Cats. That means that in order to access this value, Cats will need to call this.props.count and not this.props.catCount. This is done deliberately to point out that you can pass in any data as props and call it whatever you like.

When the button is clicked, moreAnimals() is called, which is in charge of calculating the new state and updating the state via setState(). Once setState() is called, React now knows that the state has changed and figures out which parts of the UI that need to change.

You may have spotted a refactoring opportunity here. Because React sticks to regular JavaScript as much possible and components are idempotent, you can easily refactor your code and create reusable components.

```
var AnimalRescue = React.createClass({
  getInitialState: function() {
    return { catCount: 0, dogCount: 0 }
  },

  render: function() {
    return (
      <div>
        <AnimalCount catCount={this.state.catCount} dogCount={this.state.dogCoun\
t}/>
        <button onClick={this.moreAnimals}>Save more animals</button>
      </div>
    );
  },

  moreAnimals: function() {
    var catCount = this.state.catCount + 1;
    var dogCount = this.state.dogCount + 1;
    this.setState({
      catCount: catCount,
      dogCount: dogCount
    });
  }
});

var AnimalCount = React.createClass({
  render: function() {
    return (
      <ul>
        <AnimalListItem animalName="Cats" count={this.props.catCount} />
        <AnimalListItem animalName="Dogs" count={this.props.dogCount} />
      </ul>
    );
  }
});

var AnimalListItem = React.createClass({
  render: function() {
    return (
      <li># of {this.props.animalName}: {this.props.count}</li>
    );
  }
});
```

Now AnimalRescue only cares about state, behavior, and delegating rendering of animal count to the `AnimalCount` component. `AnimalCount` only worries about rendering an unordered list of animal data. Rendering of the list items is delegated to the `AnimalListItem` component. Now we can reuse `AnimalListItem` to render more animal data should we start rescuing more types of animals.

With the Virtual DOM and unidirectional data flow, React turns the user interface into a state machine. Whenever your data changes, the rest of your application snaps into the proper place.

While this seems like a novel approach, it is actually not unique – it is just new to web development. React took a look at another area of software development, the gaming industry, and brought those ideas to the web.

In many ways, web based UIs have more in common with games than with the server. Games are also packed full of state (number of weapons, vehicles, sounds, events based on user interaction, etc…) and tons of behavior that needs to be constantly recalculated and re-render the player at 60 frames per second. On top of that, there are tough hardware constraints that still require excellent performance despite those limitations.

Indeed, the React lifecycle is very similar to the Doom 3 engine. In Doom, all the state of the game is kept at a high level container, then passes it to its front-end components which contain the game logic, followed by the creation of an intermediate representation of the scene, which then gets translated into OpenGL operations and finally flushed out to the graphics card. React's frow is very similar: the state of the application lives in higher components which then gets passed down as immutable data to child components which actually house the behavior, then a Virtual DOM is created as an intermediate representation of the DOM, finally batching of the DOM mutations and updating the real DOM itself.