ReactJS (made by Facebook - https://facebook.github.io/react/)

React makes it painless to create interactive UIs. Design simple views for each state in your application, and React will efficiently update and render just the right components when your data changes.
Declarative views make your code more predictable and easier to debug.
Component-Based
Build encapsulated components that manage their own state, then compose them to make complex UIs.

Since component logic is written in JavaScript instead of templates, you can easily pass rich data through your app and keep state out of the DOM.

Learn Once, Write Anywhere
We don't make assumptions about the rest of your technology stack, so you can develop new features in React without rewriting existing code.

React can also render on the server using Node and power mobile apps using React Native.

It is a Javascript library.
In the MVC model, it is focussed on the View so narrow scope, and tightly focused.
The data state on one side (often on a server side - the model), is reflected in the user interface on the other side (on the client side - the view).

To update the view when the data change, there are various way.
The two way biding (see above in AngularJs) when one change the other one is updated. Angular JS do this the dirty way: it loops constantly between the data and the view to see if there is a change.
React, want the UI to be reactive and only react if the data has changed. It used the virtual DOM.

Basically, the code is rendered once, then if there is a change, the code is rendered a second time, and reactJS will identify the little parts of code that are different and will only update these instead of the whole document.

JSX = Javascript XML
the entire DOM is written in JSX.

Rendering an element:

render = taking the code and combining it with the data to produce the html, css and javascript.
browser can render anything in htlm + css + javascript, what ever device you're using, it will work. On the server side, you might have .NET, NodeJs, PHP, Java or python for example.
We need to look at the 'state' of the page on the client side, the state the user has made it take. The user is expecting the page to stick to whatever state they have mad it take at some point, not to hav the whole page rendered again.

Two way to render:
server side rendering: producing complete and parsable HTML, CSS and JS on the server and sending results to the client
client side rendering: leaving some of the rendering work to be performed on the client.

Rendering the DOM is slow and difficult. If every button click requires a rendering of the dom, this will be very inefficient.

React is dealing with the DOM for you, using the virtual dom.

The virtual dom is in memory, so fast rendering, can have many states and is easy to differentiate between states. It goes from the current state to the next state. It performs a diff between the virtual dom of these two states.
Compare to Jquery, it goes further, there is no html anymore, react is the html, it is doing all the dom rendering inside of javascript, so you don't worry about the html anymore. So you start and end in Javascript.

To do so, the render function is: ReactDOM.render takes 3 variables:
ReactElement (the element to render)
DOMElement (the target DOM element to render into
Callback (an option callback function)
example:

```
ReactDOM.render (   <h1>Hello world!</>,      // this is JSX could be:  React.DOM.h1(null. 'Hello, World'),
                    document.getElementById('container'),
                    function() {windows, alert("done'):}
                    );
```

Rendering a Component

The ReactComponent object is really the jet engine of our ReactJS application. We're going to take advantage of using JavaScript to render our DOM, as opposed to just a straight HTML file.
A ReactElement is the object that's rendered inside of the ReactDOM.render function, it's good for simple HTML elements.
ReactComponent is an object too, but it returns a ReactElement via the render method and is good for complex reusable components that have event handlers and property management.
ReactElements are really contained, or encapsulated by, ReactComponents.

```
var MyComponent = React.createClass({
        render: function (){
                return (
                        <p>
                        <h2> Paragraph 1 etc...
                        );
                }
});
```

MyComponent (a variable name for a piece of html) begins with a capital M. And that is an indicator for JSX that this should be rendered as a custom ReactJS component and not just a straight text into the browser.
Useful when you want to render similar piece of code like a calendar etc... , reusable, easy to read, compact. It helps turn our HTML into more declarative user interface.

The createClass function takes a single parameter which is the specification object. The render function here, the only member of our specification object that gets passed into our class, simply returns, a whole bunch of HTML.

Component Properties

HTML is your content, CSS is your presentation, and **JavaScript is your functionality**. So if we're going to move our code, our UI code, into JavaScript, lets take advantage of JavaScript. And we're going to do that in React with properties (=props), state, and events.

The props are like a properties bag, a collection of values that are associated with a component.

We can have dynamic rendering of our data when we reuse our components. So if we want to render this component three or four or five times, each of those renderings, each of those instances can have different properties, different property values. Even though we've only defined our component a single time, we're rendering it differently each time.
Properties are immutable. They are set by outside things: state changes, events. We should never inside of our component but instead write to this.props.

```
var HelloWorld= React.createClass({
        render: function (){
                return (
                        <h1>Hello {this.props.name}</h1>
                        );
                }
});

ReactDOM.render(<div> <Helloworld name="Mike" /></div>, document.getElementByID);
```

<u>to get a 'clock' component:</u>

```
var ItIsNow = React.createClass({
        render: function (){
                return ( <p> The current time is: {this.props.date.toTimeString()} </p>);
                          }  });
setIntervak (function(){
React.DOM.render(
        <ItIsNow date ={new Date ()()} />,
        document.getElementById('container')
        );
}, 500);
```

Using Component State

ReactJS is the perfect unidirectional, unchanging, immutable User Interface Generator inside of JavaScript.
Component state, at the technical level, is a collection of mutatable properties that affect the final rendered output of the component.
Component properties are intended to be immutable. They don't change. You set them once, and your components simply render the value of whatever is in those properties.
Nothing that happens inside of components should change its own properties. But state can change.

https://github.com/uberVU/react-guide/blob/master/props-vs-state.md

	props	**state**
Can get initial value from parent component	yes	yes
Can be changed by parent component	yes	no
Can set default values inside component	yes	yes
Can change inside component	no	yes
Can set initial value for child component	yes	yes
Can change in child Components	yes	no

As much as possible, you should build stateless components. Those are desirable when appropriate, and you really want to break down your user interface into as many stateless components as possible isolating and identifying the individual spaces within your UI where you need a stateful component. That's where you're going to respond to user interaction, or to outside events.

"get initial state" function is defined by us on our class, on our component which set the initial values of a component state
"Set state" is a function that is already on our class. It's inherited from React for us. We call it whenever we want to update the state, to inform React that it needs to rerender this component to the virtual DOM.

That's the whole React life cycle: calling "set state" in order to update the value of our state is really important to trigger this change throughout React JS.
Component state, like component props, are a collection of properties for our component that simply contain values that are going to use to affect how the component renders. Unlike props, state is meant to be changed over time, typically in response to user events.
So we can use component state to manage things like clicks or updates that happen from the user that the component, itself, needs to manage. We use "get initial state" in order to set the initial values for our state. And then we use something called "set state" in order to update those values and ensure that React JS re-renders the component appropriately.

Component Events

Components are not statics, they can change, their user interface elements and user interfaces often change over time. And to represent that changing state, we use something called state.
React JS events are just browser events that they've wrapped around. Any action, any user interaction, gets trapped as an event if you want it to inside of your browser. This includes mouse click, or touches, presses, it can also include non-user interactions-- things like timers or intervals. So you can have your application set up to trigger an event every 30 seconds, or every minute, or every hour, or whatever the case may be.
Each of these events is filtered through a function inside of your component, which then tells your component to re-render, redraw itself. And from there, it goes off to the virtual DOM, and React handles all of the mechanics of redrawing it to your HTML page

The difference between event from your browser and the synthetic event that React produces is that it kind of cleans up and normalizes all the different variances between browsers and between versions of browsers that you might be using. So what you know with the synthetic event is that it's always going to handle exactly the same, regardless of the browser, the client environment that might be using your application. And that's really ideal. You don't have to program for a whole bunch of different browsers.

React JS events are:
- uniform across browsers
- consistent with W3C spec
- simple JSON objects

more here: https://www.w3.org/TR/DOM-Level-3-Events/

Components are made up of props and state. Props are immutable values that get passed from a parent component of some kind that tell the object how to render. State are also properties that tell the object how to render, but they're not immutable.
State can be changed, specifically by the object itself. How do we know when to change, though? We respond to events. Events are any action, user-generated or otherwise, that might update the state of our component. If they do, we change the state accordingly, re-render to the virtual DOM, and carry on.

JSX

JSX is an XML-styled syntax for use inside of JavaScript, and it takes the best parts of HTML.
It takes that one job that HTML does, and it allows it to keep doing it, but this time, inside of our JavaScript code.
This has enormous advantages for us, it allows us to:

- Build unit-testable modules for our content, for our HTML inside of our JavaScript. So we're no longer reliant on simply seeing the output, but we can actually proactively test our HTML code.
- Integrate various properties and states and events with our backend very simply right inside of the HTML, or in this case, the XML itself.

We get nice, clean, easy-to-read code inside of our JavaScript that's still ultimately going to evaluate into JavaScript. **JSX ultimately gets turned back into this code, always**.

JSX is not HTML, it's XML
JSX avoid JS keywords like 'class' (uses ClassName instead)
Lower case for HTML Tags
Capital case forreact Components
Camel case for HTML attributes
Style (in html tags) is a JSON object in the javascript code

https://babeljs.io/repl = converts JSX to JS
http://facebook.github.io/react/html-jsx.html = http://magic.reactjs.net/htmltojsx.htlm = converts html to JSX

JSX is optional but highly recommended XML code that you can use directly inside of JavaScript. It makes your JavaScript a little bit easier to read when it comes to declaring and defining user interface elements. It basically takes the best part of HTML and moves it into your JavaScript so that you can build your user interface with React JS.

Compound Components

The first three life cycle methods or life cycle events:
componentWillMount, componentDidMount, componentWillUnmount.

componentWillMount is a function that is automatically fired right before the component is rendered into the virtual DOM. So you can go ahead and grab a hold of that component and perform some operation if you need to before it's rendered
ComponentDidMount just like that, happens after the component has been rendered to the virtual DOM. ComponentDidMount is the aftereffect.
componentWillUnmount is what gets fired right before the component is removed or deleted from the screen. If you get a really complicated web page, really complicated application, you might need to clean up and remove React-components from the page.

Now it's important to note, that these are one time fire only events. These top two get fired right at the beginning when the component is initially being created, initially being rendered, after that they don't fire.

Next event when components receives props:
shouldComponentUpdate, componentWillUpdate, and componentDidUpdate.

shouldComponentUpdate I think is really, really cool. If you don't erupted if you don't interact with it always, by default returns a true.
ComponentWillUpdate and componentDidUpdate are similar to the WillMount and DidMount functions: They will fire right before the component updates and right after the component updates

https://facebook.github.io/react/docs/component-specs.html

Children get their props from their parents, but manage their own state. (state is private, props is public)
Lifecycle methods fire top down before render, bottom up after.

Unit Testing

With React, we can make our user interface sort of interactive, more declarative, and more functional by moving it into the JavaScript. Well, another great feature, of moving it into JavaScript is it we can enable unit testing. Our user interface code, our HTML is now programmatic, which means we can run it through a unit test program. And we can ensure, we can validate as we develop it and make changes to it that it's running appropriately.

We want our unit tests to be automated. We want it to be scripted unit tests that run somewhere in the background and execute for us. React is running inside of JavaScript and most of the time, it runs in the browser.
Node.js is a JavaScript execution engine that is divorced from the browser. You can run it right on your desktop. It'll run in Windows, Linux, Mac OS, et cetera.
Node is going to have to have a handle or our unit testing is going to have to have a handle to the React JS library. We also need access to a library called **Babel**. It simply takes ES6 compatible code and translates it into ES5 compatible code. (ES stands for ECMAScript, which is a Javascript standard). That means it takes the future version of JavaScript, the one that is not supported in every browser yet. It's still being deployed and being developed. And it turns it into backwards compatible ES5 level code that is really widespread compatibility, across many different browsers, in many different versions.
Jest is the sort of official testing, unit testing harness for React JS. It's developed and released by the team at Facebook. You can check out the URLs for each of these things right here-- Node.js, Jest, and Babel:

http://babeljs.io http://facebook.github.io/jest https://nodejs.org/en/
Getting all these installed and set up could be a bit of a pain in the butt. But thankfully, we have a quick way to do it. We're going to use something called the **Node Package Manager, NPM**. And NPM is great because we can set up a little file-- and we will do this right here-- called the **package.JSON file**. This file is going to list all the configuration dependencies that we need in order to run our unit testing. Then we'll simply run a quick command NPM install. And NPM will make sure that all these are downloaded and installed from the worldwide web.
(npm is a command line utility)

Unit testing is all about testing the basic functionality of your React components.
We build avery simple scripts that rendered our items, our components, to the virtual DOM, and then expected different output based on the input then we put in.

Build Environment

To set up a really nice development-- or build environment-- for our React.js application development is challenging, because React doesn't have nice mature development tools-- build tools-- available to it, the way something like .NET does.

Well, we're going to start with Node,js. Node,js is a really important part of the React ecosystem. Node is what's going to execute everything else, including if we want to do server side isomorphic JavaScript rendering, we can do that with Node,js, but all the rest of the tools are actually JavaScript packages-- JavaScript libraries-- and Node, and especially npm, the node package manager.

The first one is called Gulp. **Gulp** is just a build automator. It's something like Grunt, if you've used Grunt, and the basic idea with Gulp is that we can script out the build process for our React.js application. We're going to create something called a gulpfile, and that gulpfile's going to describe what needs to happen to build our React application.

We need to run our React code through a couple of additional tools. One is called **Browserify.** It is going to take all of the different packages, all the things that we need for our React.js code and it's going to bundle it inside of a single JavaScript file that we can use on our web page when we deploy. Basically, it takes everything and it puts it together in one nice, neat little package for us. Actually, a lot of times it's called bundle.js-- the end result of Browserify. So you can imagine, as your dependencies grow, and you get many, many different JavaScript files, from a development perspective, it's good to have all those different files.

Another really important tool that we are working with-- and you've seen this come up a little bit when we've been working React.js-- is something called **Babel** (see above).

The last library you're going to see me install is something called **vinyl-source-stream**, and vinyl-source-stream just gets the code that comes out of Babel and out of Browserify and makes it consumable for Gulp.