<https://codepen.io/bestjon/pen/RPXoaW/>

<https://codedump.io/share/8PXZ0Gh6Pfw/1/loading-json-data-from-local-file-into-react-js>

ReactJS:

Works on the principle of "Components" single responsibility principle from SOLID.

DRY: Dont Repeat Yourself.

One-way data binding.

Two types of data model: prop and state

**props** are a way of passing data from parent to child.

This.props : immutable; never directly write to these objects.

**State** is reserved only for interactivity, that is, data that changes over time.

State: not provided by the parent, changes with time, cannot be computed based on values of other states or props.

For building applications it is preferred to go top-bottom the components hierarchy.

Test environment: bottom-top

=====================================================================================

Building app:

=====================================================================================

1. Identify the components

2. build static with props

3. identify the state objects

4. identify the place where it would be placed.

5. and reverse data flow: onChange() and setState()

Component at the top of hierarchy has the "render()" method which accepts prop as data model.

ReactDOM.render further in the UI will update the UI.

Main render:

ReactDOM.render(<parent data model>, element);

React.createClass({

getInitialState: function(){},

handleUserInput: function(p1, p2){

this.setState({});

}

render: function(){

return ();

}

})

getInitialState() goes in the common component higher up in the hierarchy should own the state.

handleChange: function(){}

ref name to the control

onchange function in your component

**Reactive Updates:**

It uses a fast, internal mock DOM to perform diffs and computes the most efficient DOM mutation for you.

The owner-ownee relationship is specific to React, while the parent-child relationship is simply the one you know and love from the DOM

AngularJS believes in templates and view model

ReactJS: generated HTML and component from Javascript So often considered as V in MVC

**JSX lets you create JavaScript objects using HTML syntax**

React.createElement(‘type’, {attributes: values}, default\_value);

React.createElement, which takes a tag name or component, a properties object, and variable number of optional child arguments.

ar child1 = React.createElement('li', null, 'First Text Content');

var child2 = React.createElement('li', null, 'Second Text Content');

var root = React.createElement('ul', { className: 'my-list' }, child1, child2);

ReactDOM.render(root, document.getElementById('example'));

**JSX in Depth:**

* Javascript extension which looks similar to XML
* React can either render HTML tags (strings) or React components (classes).
* HTML strings:
* var myDivElement = <div className="foo" />;
* ReactDOM.render(myDivElement, document.getElementById('example'));
* React components:
* var MyComponent = React.createClass({/\*...\*/});
* var myElement = <MyComponent someProperty={true} />;
* ReactDOM.render(myElement, document.getElementById('example'));

React DOM components expect DOM property names likeclassName and htmlFor

Instead of class and for

If you are not sure of converting HTML to JSX: Can use the compiler link below:

[HTML to JSX converter](http://magic.reactjs.net/htmltojsx.htm)

Attribute Expressions: {} instead of “”

**Boolean Attributes**[**#**](https://facebook.github.io/react/docs/jsx-in-depth.html#boolean-attributes)

disabled, required, checked and readOnly. {true}

Typescript syntax with JSX

class LikeButton extends React.Component {

constructor() {

super();

this.state = {

liked: false

};

this.handleClick = this.handleClick.bind(this);

}

handleClick() {

this.setState({liked: !this.state.liked});

}

render() {

const text = this.state.liked ? 'like' : 'haven\'t liked';

return (

<div onClick={this.handleClick}>

You {text} this. Click to toggle.

</div>

);

}

}

Parent can read its children by accessing the special this.props.children prop

**Prop Validation**

React.PropTypes exports a range of validators that can be used to make sure the data you receive is valid

Array, arrayOf, bool, func,

Symbol, element, instanceOf, one of(enum)

getDefaultProps: function() {

return {

value: 'default value'

};

}

**Mixins**[**#**](https://facebook.github.io/react/docs/reusable-components.html#mixins)

sometimes very different components may share some common functionality.

## ES6 Classes

class HelloMessage extends React.Component {

render() {

return <div>Hello {this.props.name}</div>;

}

}

ReactDOM.render(<HelloMessage name="Sebastian" />, mountNode);

The API is similar to React.createClass with the exception of getInitialState. Instead of providing a separate getInitialState method, you set up your own state property in the constructor. Just like the return value of getInitialState, the value you assign tothis.state will be used as the initial state for your component.

Another difference is that propTypes and defaultProps are defined as properties on the constructor instead of in the class body.

Counter.propTypes = { initialCount: React.PropTypes.number };

Counter.defaultProps = { initialCount: 0 };

**No Mixins**

render() methods actually return a description of the DOM, and React can compare this description with the in-memory representation to compute the fastest way to update the browser.

**Component Lifecycle**

Components have three main parts of their lifecycle:

* **Mounting:** A component is being inserted into the DOM.
* **Updating:** A component is being re-rendered to determine if the DOM should be updated.
* **Unmounting:** A component is being removed from the DOM.

React provides lifecycle methods that you can specify to hook into this process. We provide**will** methods, which are called right before something happens, and **did** methods which are called right after something happens.

**Mounting**

* getInitialState(): object is invoked before a component is mounted. Stateful components should implement this and return the initial state data.
* componentWillMount() is invoked immediately before mounting occurs.
* componentDidMount() is invoked immediately after mounting occurs. Initialization that requires DOM nodes should go here.

**Updating**

* componentWillReceiveProps(object nextProps) is invoked when a mounted component receives new props. This method should be used to compare this.propsand nextProps to perform state transitions using this.setState().
* shouldComponentUpdate(object nextProps, object nextState): boolean is invoked when a component decides whether any changes warrant an update to the DOM. Implement this as an optimization to compare this.props with nextProps andthis.state with nextState and return false if React should skip updating.
* componentWillUpdate(object nextProps, object nextState) is invoked immediately before updating occurs. You cannot call this.setState() here.
* componentDidUpdate(object prevProps, object prevState) is invoked immediately after updating occurs.

**Unmounting**

* componentWillUnmount() is invoked immediately before a component is unmounted and destroyed. Cleanup should go here.

**Mounted Methods**

*Mounted* composite components also support the following method:

* component.forceUpdate() can be invoked on any mounted component when you know that some deeper aspect of the component's state has changed without usingthis.setState().

The ref attribute can be a callback function, and this callback will be executed immediately after the component is mounted.

Jest for unit testing

ReactTestUtils makes it easy to test React components in the testing framework of your choice (we use [Jest](https://facebook.github.io/jest/)).