This is a collection of simple demos of React.js.

These demos are purposely written in a simple and clear style. You will find no difficulty in following them to learn the powerful library.

**HTML Template**

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<script src="../build/react.development.js"></script>

<script src="../build/react-dom.development.js"></script>

<script src="../build/babel.min.js"></script>

</head>

<body>

<div id="example"></div>

<script type="text/babel">

// \*\* Our code goes here! \*\*

</script>

</body>

</html>

**Index**

1. Render JSX
2. Use JavaScript in JSX
3. Use array in JSX
4. Define a component
5. this.props.children
6. PropTypes
7. Finding a DOM node
8. this.state
9. Form
10. Component Lifecycle
11. Ajax
12. Display value from a Promise
13. Server-side rendering

**Demo01: Render JSX**

The template syntax in React is called JSX. It is allowed in JSX to put HTML tags directly into JavaScript codes. ReactDOM.render() is the method which translates JSX into HTML, and renders it into the specified DOM node.

ReactDOM.render(

<h1>Hello, world!</h1>,

document.getElementById('example')

);

Attention, you have to use <script type="text/babel"> to indicate JSX codes, and include babel.min.js, which is a browser version of Babel and could be get inside a babel-core@6 npm release, to actually perform the transformation in the browser.

Before v0.14, React use JSTransform.js to translate <script type="text/jsx">. It has been deprecated (more info).

**Demo02: Use JavaScript in JSX**

You could also use JavaScript in JSX. It takes angle brackets (<) as the beginning of HTML syntax, and curly brackets ({) as the beginning of JavaScript syntax.

var names = ['Alice', 'Emily', 'Kate'];

ReactDOM.render(

<div>

{

names.map(function (name) {

return <div>Hello, {name}!</div>

})

}

</div>,

document.getElementById('example')

);

**Demo03: Use array in JSX**

If a JavaScript variable is an array, JSX will implicitly concat all members of the array.

var arr = [

<h1>Hello world!</h1>,

<h2>React is awesome</h2>,

];

ReactDOM.render(

<div>{arr}</div>,

document.getElementById('example')

);

**Demo04: Define a component**

class ComponentName extends React.Component creates a component class, which implements a render method to return an component instance of the class.

Before v16.0, React use React.createClass() to create a component class. It has been deprecated (more info).

class HelloMessage extends React.Component {

render() {

return <h1>Hello {this.props.name}</h1>;

}

}

ReactDOM.render(

<HelloMessage name="John" />,

document.getElementById('example')

);

Components would have attributes, and you can use this.props.[attribute] to access them, just like this.props.name of <HelloMessage name="John" /> is John.

Please remember the first letter of the component's name must be capitalized, otherwise React will throw an error. For instance, HelloMessage as a component's name is OK, but helloMessage is not allowed. And a React component should only have one top child node.

// wrong

class HelloMessage extends React.Component {

render() {

return <h1>

Hello {this.props.name}

</h1><p>

some text

</p>;

}

}

// correct

class HelloMessage extends React.Component {

render() {

return <div>

<h1>Hello {this.props.name}</h1>

<p>some text</p>

</div>;

}

}

**Demo05: this.props.children**

React uses this.props.children to access a component's children nodes.

class NotesList extends React.Component {

render() {

return (

<ol>

{

React.Children.map(this.props.children, function (child) {

return <li>{child}</li>;

})

}

</ol>

);

}

}

ReactDOM.render(

<NotesList>

<span>hello</span>

<span>world</span>

</NotesList>,

document.getElementById('example')

);

Please be mindful that the value of this.props.children has three possibilities. If the component has no children node, the value is undefined; If single children node, an object; If multiple children nodes, an array. You should be careful to handle it.

React gave us an utility React.Children for dealing with the this.props.children's opaque data structure. You could use React.Children.map to iterate this.props.children without worring its data type being undefined or object. Check official document for more methods React.Children offers.

**Demo06: PropTypes**

Components have many specific attributes which are called props in React and can be of any type.

Sometimes you need a way to validate these props. You don't want users have the freedom to input anything into your components.

React has a solution for this and it's called PropTypes.

class MyTitle extends React.Component {

static propTypes = {

title: PropTypes.string.isRequired,

}

render() {

return <h1> {this.props.title} </h1>;

}

}

The above component of MyTitle has a props of title. PropTypes tells React that the title is required and its value should be a string.

Now we give Title a number value.

var data = 123;

ReactDOM.render(

<MyTitle title={data} />,

document.getElementById('example')

);

It means the props doesn't pass the validation, and the console will show you an error message.

Warning: Failed propType: Invalid prop `title` of type `number` supplied to `MyTitle`, expected `string`.

Visit official doc for more PropTypes options.

P.S. If you want to give the props a default value, use defaultProps.

class MyTitle extends React.Component {

constructor(props) {

super(props)

}

static defaultProps = {

title: 'Hello World',

}

render() {

return <h1> {this.props.title} </h1>;

}

}

ReactDOM.render(

<MyTitle />,

document.getElementById('example')

);

React.PropTypes has moved into a different package since React v15.5. (more info).

**Demo07: Finding a DOM node**

Sometimes you need to reference a DOM node in a component. React gives you the ref attribute to attach a DOM node to instance created by React.createRef().

class MyComponent extends React.Component {

constructor(props) {

super(props);

this.myTextInput = React.createRef();

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

}

handleClick() {

this.myTextInput.current.focus();

}

render() {

return (

<div>

<input type="text" ref={this.myTextInput} />

<input type="button" value="Focus the text input" onClick={this.handleClick} />

</div>

);

}

}

ReactDOM.render(

<MyComponent />,

document.getElementById('example')

);

Please be mindful that you could do that only after this component has been mounted into the DOM, otherwise you get null.

**Demo08: this.state**

React thinks of component as state machines, and uses this.state to hold component's state, this.setState() to update this.state and re-render the component.

class LikeButton extends React.Component {

constructor(props) {

super(props)

this.state = {

liked: false

}

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

}

handleClick(event) {

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

}

render() {

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

return (

<p onClick={this.handleClick}>

You {text} this. Click to toggle.

</p>

);

}

}

ReactDOM.render(

<LikeButton />,

document.getElementById('example')

);

You could use component attributes to register event handlers, just like onClick, onKeyDown, onCopy, etc. Official Document has all supported events.

**Demo09: Form**

According to React's design philosophy, this.state describes the state of component and is mutated via user interactions, and this.props describes the properties of component and is stable and immutable.

Since that, the value attribute of Form components, such as <input>, <textarea>, and <option>, is unaffected by any user input. If you wanted to access or update the value in response to user input, you could use the onChange event.

class Input extends React.Component {

constructor(props) {

super(props)

this.state = {value: 'Hello!'}

this.handleChange = this.handleChange.bind(this)

}

handleChange(event) {

this.setState({value: event.target.value});

}

render() {

var value = this.state.value;

return (

<div>

<input type="text" value={value} onChange={this.handleChange} />

<p>{value}</p>

</div>

);

}

}

ReactDOM.render(<Input/>, document.getElementById('example'));

More information on official document.

**Demo10: Component Lifecycle**

Components have three main parts of their lifecycle: Mounting(being inserted into the DOM), Updating(being re-rendered) and Unmounting(being removed from the DOM). React provides hooks into these lifecycle part. will methods are called right before something happens, and did methods which are called right after something happens.

class Hello extends React.Component {

constructor(props) {

super(props)

this.state = {opacity: 1.0};

}

componentDidMount() {

this.timer = setInterval(function () {

var opacity = this.state.opacity;

opacity -= .05;

if (opacity < 0.1) {

opacity = 1.0;

}

this.setState({

opacity: opacity

});

}.bind(this), 100);

}

render() {

return (

<div style={{opacity: this.state.opacity}}>

Hello {this.props.name}

</div>

);

}

}

ReactDOM.render(

<Hello name="world"/>,

document.getElementById('example')

);

The following is a whole list of lifecycle methods.

* **componentWillMount()**: Fired once, before initial rendering occurs. Good place to wire-up message listeners. this.setState doesn't work here.
* **componentDidMount()**: Fired once, after initial rendering occurs. Can use this.getDOMNode().
* **componentWillUpdate(object nextProps, object nextState)**: Fired after the component's updates are made to the DOM. Can use this.getDOMNode() for updates.
* **componentDidUpdate(object prevProps, object prevState)**: Invoked immediately after the component's updates are flushed to the DOM. This method is not called for the initial render. Use this as an opportunity to operate on the DOM when the component has been updated.
* **componentWillUnmount()**: Fired immediately before a component is unmounted from the DOM. Good place to remove message listeners or general clean up.
* **componentWillReceiveProps(object nextProps)**: Fired when a component is receiving new props. You might want to this.setState depending on the props.
* **shouldComponentUpdate(object nextProps, object nextState)**: Fired before rendering when new props or state are received. return false if you know an update isn't needed.

**Demo11: Ajax**

How to get the data of a component from a server or an API provider? The answer is using Ajax to fetch data in the event handler of componentDidMount. When the server response arrives, store the data with this.setState() to trigger a re-render of your UI.

class UserGist extends React.Component {

constructor(props) {

super(props)

this.state = {

username: '',

lastGistUrl: ''

};

}

componentDidMount() {

$.get(this.props.source, function(result) {

var lastGist = result[0];

this.setState({

username: lastGist.owner.login,

lastGistUrl: lastGist.html\_url

});

}.bind(this));

}

render() {

return (

<div>

{this.state.username}'s last gist is

<a href={this.state.lastGistUrl}>here</a>.

</div>

);

}

}

ReactDOM.render(

<UserGist source="https://api.github.com/users/octocat/gists" />,

document.getElementById('example')

);

**Demo12: Display value from a Promise**

This demo is inspired by Nat Pryce's article "Higher Order React Components".

If a React component's data is received asynchronously, we can use a Promise object as the component's property also, just as the following.

ReactDOM.render(

<RepoList promise={$.getJSON('https://api.github.com/search/repositories?q=javascript&sort=stars')} />,

document.getElementById('example')

);

The above code takes data from Github's API, and the RepoList component gets a Promise object as its property.

Now, while the promise is pending, the component displays a loading indicator. When the promise is resolved successfully, the component displays a list of repository information. If the promise is rejected, the component displays an error message.

class RepoList extends React.Component {

constructor(props) {

super(props)

this.state = {

loading: true,

error: null,

data: null

};

}

componentDidMount() {

this.props.promise.then(

value => this.setState({loading: false, data: value}),

error => this.setState({loading: false, error: error}));

}

render() {

if (this.state.loading) {

return <span>Loading...</span>;

}

else if (this.state.error !== null) {

return <span>Error: {this.state.error.message}</span>;

}

else {

var repos = this.state.data.items;

var repoList = repos.map(function (repo, index) {

return (

<li key={index}><a href={repo.html\_url}>{repo.name}</a> ({repo.stargazers\_count} stars) <br/> {repo.description}</li>

);

});

return (

<main>

<h1>Most Popular JavaScript Projects in Github</h1>

<ol>{repoList}</ol>

</main>

);

}

}

}

**Demo13: Server-side rendering**

JSX syntax.

# install the dependencies in demo13 directory

$ npm install

# translate all jsx file in src subdirectory to js file

$ npm run build

# launch http server

$ node server.js

**Extras**

**Precompiling JSX**

All above demos don't use JSX compilation for clarity. In production environment, ensure to precompile JSX files before putting them online.

First, install the command-line tools Babel.

$ npm install -g babel

Then precompile your JSX files(.jsx) into JavaScript(.js). Compiling the entire src directory and output it to the build directory, you may use the option --out-dir or -d.

$ babel src --out-dir build

Put the compiled JS files into HTML.

<!DOCTYPE html>

<html>

<head>

<title>Hello React!</title>

<script src="build/react.js"></script>

<script src="build/react-dom.js"></script>

<!-- No need for Browser.js! -->

</head>

<body>

<div id="example"></div>

<script src="build/helloworld.js"></script>

</body>

</html>