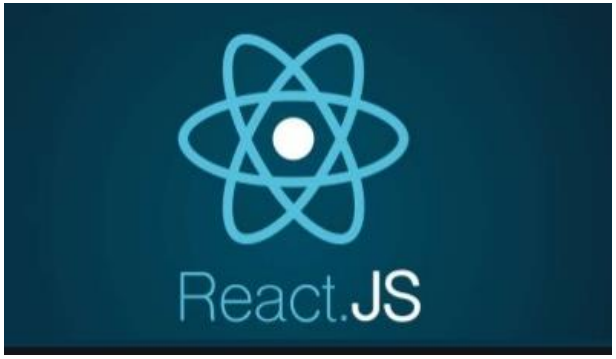


Presentation by Uplatz

Contact Us: https://training.uplatz.com/

Email: info@uplatz.com

Phone:+44 7836 212635

Uplatz

Table Of Contents:

React Without ES6

- Normally you would define a React component as a plain JavaScript class:

```
class Greeting extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

- If you don't use ES6 yet, you may use the create-react-class module instead:

```
var createReactClass = require('create-react-class');
var Greeting = createReactClass({
  render: function() {
```

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

- The API of ES6 classes is similar to createReactClass() with a few exceptions.

Declaring Default Props

- With functions and ES6 classes defaultProps is defined as a property on the component itself:

```
class Greeting extends React.Component {
  // ...
}
Greeting.defaultProps = {
  name: 'Mary'
};
```

- With createReactClass(), you need to define getDefaultProps() as a function on the passed object:

```
var Greeting = createReactClass({
  getDefaultProps: function() {
    return {
      name: 'Mary'
    };
  },// ...});
```

Setting the Initial State

- In ES6 classes, you can define the initial state by assigning this.state in the constructor:

```
class Counter extends React.Component {
```

```
constructor(props) {
    super(props);
    this.state = {count: props.initialCount};
  }
  // ...}
```

- With createReactClass(), you have to provide a separate getInitialState method that returns the initial state:

```
var Counter = createReactClass({
  getInitialState: function() {
    return {count: this.props.initialCount};
  },
  // ...
});
```

Autobinding

- In React components declared as ES6 classes, methods follow the same semantics as regular ES6 classes.
- This means that they don't automatically bind this to the instance.
- You'll have to explicitly use .bind(this) in the constructor:

```
class SayHello extends React.Component {
  constructor(props) {
    super(props);
    this.state = {message: 'Hello!'};
    // This line is important!
    this.handleClick = this.handleClick.bind(this);
  }
```

Uplatz

```
 handleClick() {
    alert(this.state.message);
  }
 render() {
    // Because `this.handleClick` is bound, we can use it as an event handler.
    return (
      <button onClick={this.handleClick}>
        Say hello
      </button>
    );
  }
}
```

With createReactClass(), this is not necessary because it binds all methods:

```
var SayHello = createReactClass({
  getInitialState: function() {
    return {message: 'Hello!'};
  },
handleClick: function() {
    alert(this.state.message);
  },
render: function() {
    return (
      <button onClick={this.handleClick}>
        Say hello
```

```
</button>
    );
  }
});
```

- This means writing ES6 classes comes with a little more boilerplate code for event handlers, but the upside is slightly better performance in large applications.
- If the boilerplate code is too unattractive to you, you may enable the experimental Class Properties syntax proposal with Babel:

```
class SayHello extends React.Component {
  constructor(props) {
    super(props);
```

```
this.state = {message: 'Hello!'};
  }
  // WARNING: this syntax is experimental!
  // Using an arrow here binds the method:
  handleClick = () => {
    alert(this.state.message);
  }
render() {
    return (
      <button onClick={this.handleClick}>
        Say hello
      </button>
    );
  } }
```

- Please note that the syntax above is experimental and the syntax may change, or the proposal might not make it into the language.
- If you'd rather play it safe, you have a few options:
- Bind methods in the constructor.

Use arrow functions,

e.g. onClick={(e) => this.handleClick(e)}.

- Keep using createReactClass.

Mixins

Note:

- ES6 launched without any mixin support.
- Therefore, there is no support for mixins when you use React with ES6 classes.
- We also found numerous issues in codebases using mixins, and don't recommend using them in the new code.

- This section exists only for the reference.
- Sometimes very different components may share some common functionality.
- These are sometimes called cross-cutting concerns. createReactClass lets you use a legacy mixins system for that.
- One common use case is a component wanting to update itself on a time interval.
- It's easy to use setInterval(), but it's important to cancel your interval when you don't need it anymore to save memory.
- React provides lifecycle methods that let you know when a component is about to be created or destroyed.

- Let's create a simple mixin that uses these methods to provide an easy setInterval() function that will automatically get cleaned up when your component is destroyed.

```
var SetIntervalMixin = {
  componentWillMount: function() {
    this.intervals = [];
  },
  setInterval: function() {
    this.intervals.push(setInterval.apply(null,
arguments));
  },
  componentWillUnmount: function() {
    this.intervals.forEach(clearInterval);
  }
};
```

```
var createReactClass = require('create-react-class');
var TickTock = createReactClass({
  mixins: [SetIntervalMixin], // Use the mixin
  getInitialState: function() {
    return {seconds: 0};
  },
  componentDidMount: function() {
    this.setInterval(this.tick, 1000); // Call a method on the
mixin
  },
  tick: function() {
    this.setState({seconds: this.state.seconds + 1});
  },
  render: function() {
```

```
return (
      <p>
        React has been running for {this.state.seconds}
seconds.
      </p>
    );
  }
});
ReactDOM.render(
  <TickTock />,
  document.getElementById('example')
);
```

- If a component is using multiple mixins and several mixins define the same lifecycle method (i.e. several

- mixins want to do some cleanup when the component is destroyed), all of the lifecycle methods are guaranteed to be called.
- Methods defined on mixins run in the order mixins were listed, followed by a method call on the component.

Introduction into ES6 and React

- By looking at ES6 compatibility table we could notice that not all the browsers support every single feature of ES2015.
- Luckily, you don't have to spend ages until vendors will ship ECMAScript 6 features to the browsers.
- There are already existing solutions called transpilers that convert your code written in ES6 to ES5-compatible code.

- That's very similar to how CoffeeScript turns your Coffee code into JavaScript.
- One of these solutions is Babel, really amazing tool. Many thanks to its authors.
- What's good, Babel supports a huge amount of different frameworks, build systems, test frameworks, template engines - look here.
- To give you a quick idea of how babel works here is an example. Say, we have following code written in ECMAScript 6:

```
var evenNumbers = numbers.filter((num) => num % 2 === 0);
```

- After running babel for that code you will have:

```
var evenNumbers = numbers.filter(function (num) {
  return num % 2 === 0;
});
```

Preparing development environment

- In order to have a continuous workflow with babel, we will use Gulp.
- This is task runner built on top of node.js which could improve your life by automating tedious tasks.
- If you heard about Grunt then Gulp has the same purpose.

Obviously, you will need node.js. Install it on your system if you don't have it.

- Next, you will need to install Gulp globally:

npm install --g gulp.

- Switch to your project's directory. Initialize your package.json file by using npm init and answering appeared questions.

- Run npm install --save react react-dom.
- This will install react npm module into your node_modules folder and save React library as dependency inside your package.json file.
- Run npm install --save-dev gulp browserify babelify vinyl-source-stream babel-preset-es2015 babel-preset-react.
- This will install more development dependencies to your node_modules.

Creating gulpfile.js

- Create file named gulpfile.js in the root directory of your project with following content:

```
var gulp = require('gulp');
var browserify = require('browserify');
```

```
var babelify = require('babelify');
var source = require('vinyl-source-stream');
gulp.task('build', function () {
    return browserify({entries: './app.jsx', extensions: ['.jsx'],
debug: true})
        .transform('babelify', {presets: ['es2015', 'react']})
        .bundle()
        .pipe(source('bundle.js'))
        .pipe(gulp.dest('dist'));
});
gulp.task('watch', ['build'], function () {
    gulp.watch('*.jsx', ['build']);
});
gulp.task('default', ['watch']);
```

some explanations would be helpful.

- **Lines 1-4**. We require installed node.js modules and assign them to variables.
- **Line 6.** We define gulp task named build that could be run by typing gulp build.
- **Line 7**. We start to describe what our build task will do. We tell Gulp to use Browserify for app.jsx. Additionally, we turn on debug mode which is beneficial for development.
- **Lines 8-11**. We apply Babelify transform to our code. This allows us to convert code written in ECMAScript6 to ECMAScript5.
- Next we output the result to dist/bundle.js file. Update from 03.11.2015: These lines now use new feature of Babel 6 called presets.

Lines 14-16. We define gulp task named watch that could be run by typing gulp watch.

- This task will run build task whenever any of jsx file changes.
- **Line 18**. We define default gulp task that could be run by typing gulp. This task simply executes watch task.
- Your typical workflow will include typing gulp in a terminal and pressing Enter key.
- It will watch for changes inside your React components and re-build everything continuously.

JSX and Babel

- You might already notice that we are using .jsx extension instead of .js.
- JSX is special JavaScript syntax extension developed by ReactJS team.

- This format is delivered especially for the more convenient development of ReactJS components.
- **Writing first React component using ECMAScript 6**

```
import React from 'react';
class HelloWorld extends React.Component {
    render() {
        return <h1>Hello from {this.props.phrase}!</h1>;
    }
}
export default HelloWorld;
```

Some explanation:

- **Line 1.** We import React library and put it to a variable named React.
- **Lines 3-8.** Creation of React component using ES6 class by extending React.Component class.

- We add very simple render method which returns <h1> tag containing phrase prop.
- **Line 9.** We export just created a component to outside world using export default HelloWorld.
- For simpler understanding, I placed the code for the same component, but written without usage

For simpler understanding, I placed the code for the same component, but written without usage of ES6 classes:

```
import React from 'react';
var HelloWorld = React.createClass({
    render: function() {
        return (
            <h1>Hello from {this.props.phrase}!</h1>
        );}
```

```
});
export default HelloWorld;
```

Create file named **app.jsx**:

```
import HelloWorld from './hello-world';
import React from 'react';
import ReactDOM from 'react-dom';
ReactDOM.render(
    <HelloWorld phrase="ES6"/>,
    document.querySelector('.root')
);
```

- Here we import HelloWorld component created on the previous step and set phrase prop of that component.

- Please also note that we use special react-dom package for rendering of the HelloWorld component.
- That's because core React package is separated from rendering package starting from React version 0.14.

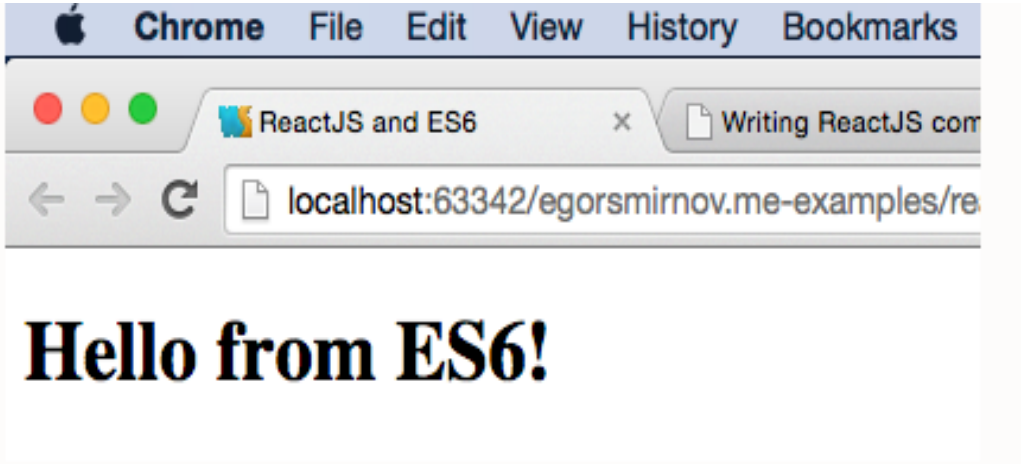
Next, let's create index.html:

```
<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8">
    <title>ReactJS and ES6</title>
</head>
<body>
```

```
<div class="root"></div>
<script src="dist/bundle.js"></script>
</body>
</html>
```

- Now run gulp from a terminal (it will create dist/bundle.js file) and open this HTML file in your browser.

You should see below image.



React Classes and ES7 Property Initializers

Creation of index.html file

- Let's start with the creation of simple HTML file that will be our HTML template file.

```
<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8">
    <title>React and ES6 Part 2</title>
    <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/foundation/
5.5.2/css/foundation.min.css">
</head>
<body>
```

** **

<script src="dist/bundle.js"></script>

</body>

</html>

Gulpfile.js

- Just copy and paste gulpfile.js from the previous part of series. No changes to it by now.
- Same copy & pasting for package.json.
- Next, run npm install (to install node dependencies) and then gulp. Gulp will continuously watch for your changes.

Main application React Component

Create app.jsx:

import React from 'react';

import ReactDOM from 'react-dom';

```
import CartItem from './cartItem';
const order = {
    title: 'Fresh fruits package',
    image: 'http://images.all-free-
download.com/images/graphiclarge/citrus_fruit_184416.jpg',
    initialQty: 3,
    price: 8
};
ReactDOM.render(
    <CartItem title={order.title}
              image={order.image}
              initialQty={order.initialQty}
              price={order.price}/>,
    document.querySelector('.root')
```

);

What's going on here:

- **Lines 1-2.** We import React / ReactDOM libraries.
- **Line 3.** Import CartItem React component that we'll create later.
- **Lines 5-10.** Sending props to CartItem component (props include item title, image, initial quantity and price).
- **Lines 12-18.** Mounting CartItem component to DOM element specified by its CSS class .root.

CartItem React Component skeleton

- Now it's time to create component responsible for displaying item's data and also for interaction with a user.

Add this code to file named cartItem.jsx:

Uplatz

```
import React from 'react';
export default class CartItem extends
React.Component {
    constructor(props) {
        super(props);
        this.state = {
            qty: props.initialQty,
            total: 0
        };
    }
    componentWillMount() {
        this.recalculateTotal();
    }
    increaseQty() {
```

```
this.setState({qty: this.state.qty + 1},
this.recalculateTotal);
    }
    decreaseQty() {
        let newQty = this.state.qty > 0 ? this.state.qty - 1 : 0;
        this.setState({qty: newQty}, this.recalculateTotal);
    }
    recalculateTotal() {
        this.setState({total: this.state.qty * this.props.price});
    }
}
```

Explanation:

Lines 4-10. This is the constructor of newly added React class. The first thing to note here is call to super(props) which is required.

- Another thing - instead of using React **getInitialState**() lifecycle method, React team recommends to use instance property called **this.state**.
- We initialize our state with props sent from app.jsx file. Personally, I like this improvement.
- **Lines 11-13**. Declaration of componentWillMount() lifecycle method (which is eventually just CartItem class method).
- Inside this method, we do a recalculation of the total price.
- To calculate total price we use recalculateTotal() which multiplies quantity (which is state of our component) by price (which is props).
- **Lines 14-20**. These are methods for increasing or decreasing the amount of items by a user.

- These methods will be called when user clicks corresponding buttons in the view (refer to the screenshot of the application at the beginning of post).
- Also notice that we are using second callback parameter of this.setState() method to recalculate the total price.

CartItem render method

Add this new method inside CartItem class:

```
export default class CartItem extends
React.Component {
// previous code we wrote here
    render() {
        return <article className="row large-4">
            <figure className="text-center">
```

```
<p>
    <img src={this.props.image}/>
        </p>
        <figcaption>
            <h2>{this.props.title}</h2>
        </figcaption>
    </figure>
    <p className="large-4 column"><strong>Quantity: {this.state.qty}</strong></p>
<p className="large-4 column">
        <button onClick={this.increaseQty.bind(this)} className="button success">+</button>
        <button onClick={this.decreaseQty.bind(this)} className="button alert">-</button> </p>
```

```
<p className="large-4 column"><strong>Price per item:</strong> ${this.props.price}</p>
<h3 className="large-12 column text-center">
            Total: ${this.state.total}
        </h3>

    </article>;
  }
}
```

- Here we just output tags using JSX + component state and apply some Foundation CSS beauty.
- Don't worry about {this.increaseQty.bind(this)} - I am going to cover this in the next part of series.
- For now just believe me that this line will call increaseQty() method of CartItem class.

- So, now we have pretty simple application interacting with a user.
- Even this example is simple, it showed us how to write React components using ECMAScript6.
- Personally I like the new syntax that ES6 brings to us.

Default Props and Prop Types for ES6 React classes

- Imagine we want to add some validation and default values for CartItem component.
- Luckily, this is built-in into React and called Default Props and Prop Types.
- You could familiarize yourself with both of them here.

Right after CartItem class add these lines:

```
CartItem.propTypes = {
    title: React.PropTypes.string.isRequired,
```

```
price: React.PropTypes.number.isRequired,
    initialQty: React.PropTypes.number
};
CartItem.defaultProps = {
    title: 'Undefined Product',
    price: 100,
    initialQty: 0
};
```

- As a result, if you now send numeric title prop inside app.jsx you'll have a warning in console. This is the power of React props validation.
- Bringing ES7 into the project
- Even if ES7 is still in the very early stage, there is number of proposed features that are already implemented in Babel.

- And these experimental features are transpiled to valid ES5 code from ES7. Awesome!
- First, install the missing npm module by running npm install --save-dev babel-preset-stage-0.
- Next, in order to bring that additional syntax sugar to our project, we'll need to make the very small change on line 8 of gulpfile.js.

Substitute this line with the code below:

.transform('babelify', {presets: ['react', 'es2015', 'stage-0']})

ES7 Property Initializers for Default Props and Prop Types of React component

Inside CartItem class add this right above constructor:

export default class CartItem extends React.Component {

```
 static propTypes = {
        title: React.PropTypes.string.isRequired,
        price: React.PropTypes.number.isRequired,
        initialQty: React.PropTypes.number
    };
    static defaultProps = {
        title: 'Undefined Product',
        price: 100,
        initialQty: 0
    };constructor() {
        ...
    }
 // .. all other code
}
```

- This is doing the same thing as code after the class definition, just in a bit more neatly way (in my opinion). You could also delete code related to propTypes / defaultProps that goes after CartItem class definition as it is not needed anymore.
- ES7 Property Initializers for initial state of React component
- The finishing step will be transferring initial state from the constructor to property initializer.

Add this right before constructor of CartItem class:

```
export default class CartItem extends
React.Component {
    // .. some code here
    state = {
        qty: this.props.initialQty,
```

```
total: 0
};
// .. constructor starts here
```

- You could also delete code related to initializing of state from the constructor.Done.
- You could check out final cartItem.jsx for this part on the series on the GitHub.
- In this part, we familiarized ourselves with writing React components using ES6 classes and introduced ES7 property initializers
- If you look at paragraph "CartItem render method" of the previous article in the series you might be confused by the usage of {this.increaseQty.bind(this)}.

- if we try the same example with just {this.increaseQty} we'll see Uncaught TypeError: Cannot read property 'setState' of undefined in browser console:

- This is because when we call a function in that way binding to this is not a class itself, it's undefined.
- It's default JavaScript behavior and is quite expected.
- In opposite to this, in case you use React.createClass() all the methods are autobinded to the instance of an object.

- Which might be counter-intuitive for some developers.
- No autobinding was the decision of React team when they implemented support of ES6 classes for React components.

Method 1. Using of Function.prototype.bind().

We've already seen this:

```
export default class CartItem extends
React.Component {
    render() {
        <button onClick={this.increaseQty.bind(this)}
className="button success">+</button>
    }
}
```

- As any method of ES6 class is plain JavaScript function it inherits bind() from Function prototype.
- So now when we call increaseQty() inside JSX, this will point to our class instance.

Method 2. Using function defined in constructor.

This method is a mix of the previous one with usage of class constructor function:

```
export default class CartItem extends
React.Component {
    constructor(props) {
        super(props);
        this.increaseQty = this.increaseQty.bind(this);
    }
 render() {
```

```
<button onClick={this.increaseQty} className="button success">+</button>
    }
}
```

- You don't have to use bind() inside your JSX anymore, but this comes with the cost of increasing the size of constructor code.

Method 3.

- Using fat arrow function and constructor.
- ES6 fat arrow functions preserve this context when they are called.
- We could use this feature and redefine increaseQty() inside constructor in the following way:

```
export default class CartItem extends React.Component {
```

```
 constructor(props) {
        super(props);
        this._increaseQty = () => this.increaseQty();
    }
render() {
        <button onClick={_this.increaseQty}
className="button success">+</button>
    }
}
```

Method 4.

- Using fat arrow function and ES2015+ class properties.
- Additionally, you could use fat arrow function in combination with experimental ES2015+ class properties syntax:

Uplatz

```
export default class CartItem extends
React.Component {
      increaseQty = () => this.increaseQty();
 render() {
       <button onClick={this.increaseQty}
className="button success">+</button>
   }
}
```

- So, instead of defining our class method in a constructor as in method number 3, we use property initializer.
- **Warning**: Class properties are not yet part of current JavaScript standard.
- But your are free to use them in Babel using corresponding experimental flag (stage 0 in our case).

Method 5.

- Using ES2015+ function bind syntax.
- Quite recently Babel introduced syntactic sugar for Function.prototype.bind() with the usage of ::. I won't go into details of how it works here.
- Other guys already have done the pretty good explanation
- Below is code with usage of ES2015+ function bind syntax:

```
export default class CartItem extends
React.Component {
    constructor(props) {
        super(props);
        this.increaseQty = ::this.increaseQty;
        // line above is an equivalent to this.increas
this.increaseQty.bind(this);
```

```
} render() {
        <button onClick={this.increaseQty}
className="button success">+</button>
    }
}
```

Method 6.

- Using ES2015+ Function Bind Syntax in-place.
- You also have a possibility to use ES2015+ function bind syntax directly in your JSX without touching constructor.
- It will look like:

```
export default class CartItem extends
React.Component {
    render() {
```

```
<button onClick={::this.increaseQty}
className="button success">+</button>
    }
}
```

- Very concise, the only drawback is that this function will be re-created on each subsequent render of the component. This is not optimal.
- What is more important that will cause problems if you use something like PureRenderMixin (or its equivalent for ES2015 classes).

What is JSPM?

- Nowadays to start a new frontend project we as developers have to do too many actions before writing any line of code.

- You have to think what JavaScript module loader system you are going to use, prepare gulp / grunt / npm / whatever script to prepare assets for production, you have to select CSS pre-processing tool, you have to think about sourcemaps integration and a lot of other things.
- **One of such a tools is JSPM (JavaScript Package Manager).**
- JSPM allows you to load JavaScript modules in different formats (ES6, CommonJS, AMD and global).
- It allows you to install dependencies from different registries - npm or github.
- JSPM has ES6+ support out of the box.

- You are able to load CSS / images / fonts / other formats of files without additional hassle.
- That is possible with a help of plugins (described later in this article).
- JSPM makes it easy to prepare your files for production use.

Initial setup our JSPM + React project

First of all run this command:

- install -g jspm@0.16.12
- update from 18.06.2016 we have to fixate the version because of some further problems with newer version of JSPM.
- This will install JSPM as a global npm module.
- After that, it would be possible to use jspm command from any place in your system.

- Next, go to your project directory and run jspm init and provide default answers for all prompts.
- The output in your terminal should look like this:

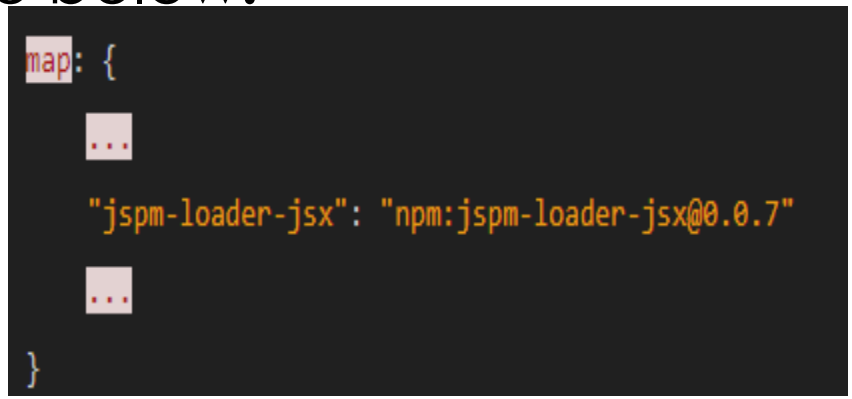
```
Package.json file does not exist, create it? [yes]:

Would you like jspm to prefix the jspm package.json properties under jspm? [yes]:

Enter server baseURL (public folder path) [./]:

Enter jspm packages folder [./jspm_packages]:

Enter config file path [./config.js]:

Configuration file config.js doesn't exist, create it? [yes]:

Enter client baseURL (public folder URL) [/]:

Do you wish to use a transpiler? [yes]:

Which ES6 transpiler would you like to use, Babel, TypeScript or Traceur? [babel]:
```

- Next thing that will be produced after running jspm init is a file named config.js.
- First of all, config.js provides configuration for versions of dependencies we install.
- Additionally, you are able to rename your packages to the name you like the most.

Let's understand what I mean by all that. Run this command

jspm install npm:jspm-loader-jsx

- If you'll take a look at config.js now you should see excerpt like below:



- Now it's possible to use this module inside the code by its name jspm-loader-jsx. Let's say we don't like this name and would like to reference it as jsx inside our code.

Run following commands:

jspm uninstall jspm-loader-jsx

jspm install jsx=npm:jspm-loader-jsx

- Our plugin will be now referenced as jsx and you could include it into your code by import jsx (though we don't need to do that as this package is JSPM plugin).
- By the way, the package we've just installed will be needed for our project (JSPM doesn't support loading of JSX files by default).
- So don't delete jspm-loader-jsx package :)

Let's install other packages we are going to use:

jspm install react react-dom

- We haven't used npm: or github: prefix here because JSPM has the registry of commonly used packages.
- It's located here and you could always add a package to the list by opening a new PR.
- Inside config.js find key named babelOptions and replace it with following lines:

```
babelOptions: {
    "stage": 0,
    "optional": [
      "runtime",
      "optimisation.modules.system"
    ] }
```

Summary:

- React without ES6
- React with ES6
- React Classes and ES7 Property Initializers
- Binding to methods of React class (ES7 included)
- React and ES6 Workflow with JSPM

Uplatz

Thank You.....

If you have any quries please write to info@uplatz.com".

Uplatz