

# React



# Redux

Carlos de la Orden | [carlos@redradix.com](mailto:carlos@redradix.com)

**\*redradix**

# Contenido

- 1. ¿Qué es React?**
2. Construyendo componentes React
3. UIs complejas mediante composición
4. Redux
5. React y Redux en el mundo real
6. Testing

# 1. ¿Qué es React?

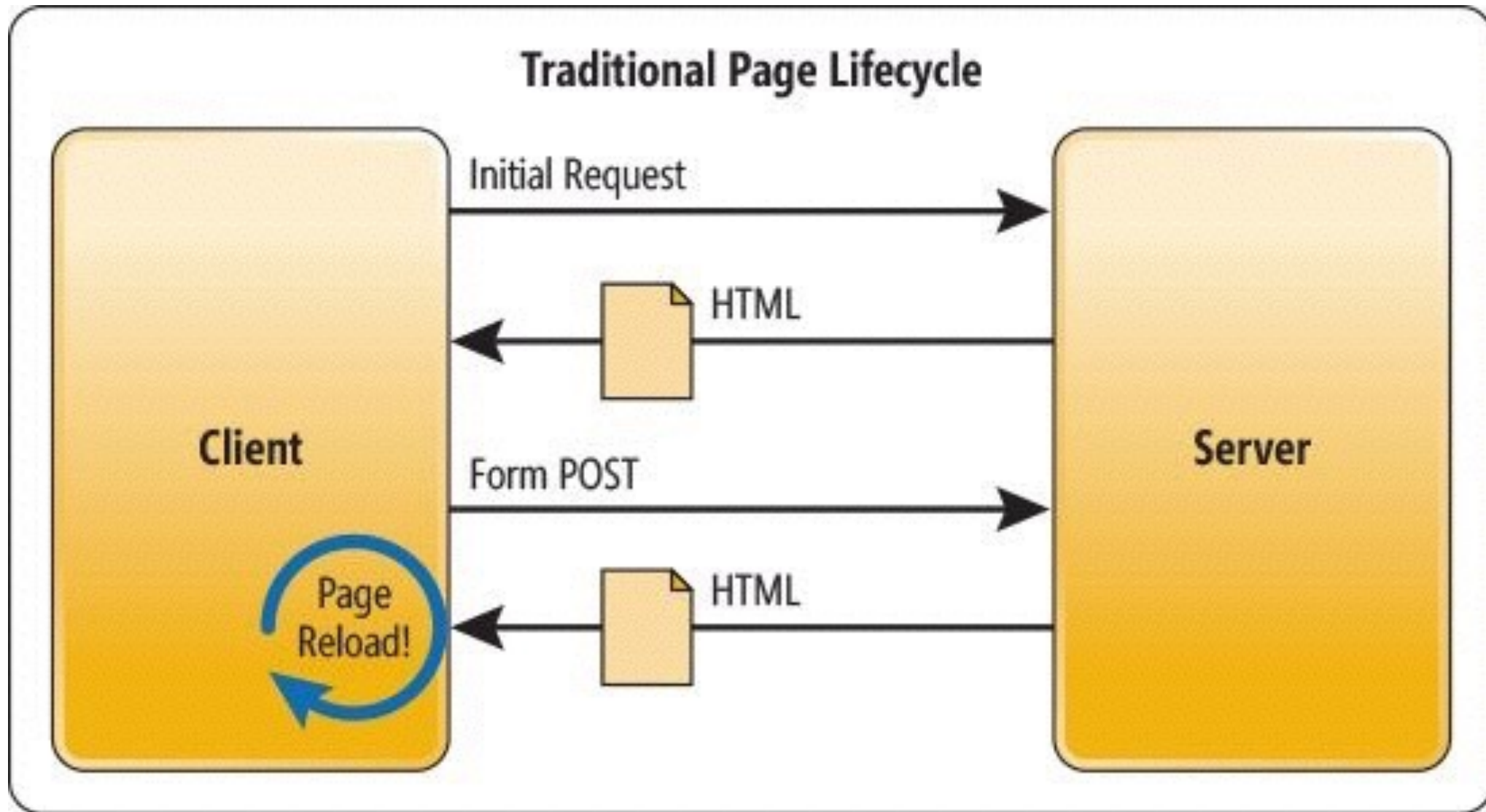
# Origen: de página a aplicación

El pasado: páginas web

- Server-side: TODA la lógica en el servidor
- Cliente sólo pide y visualiza páginas completas
- Para interactuar:
  - Cambiar URL (HTTP GET)
  - Formulario (HTTP POST)

# Origen: de página a aplicación

El pasado: páginas web



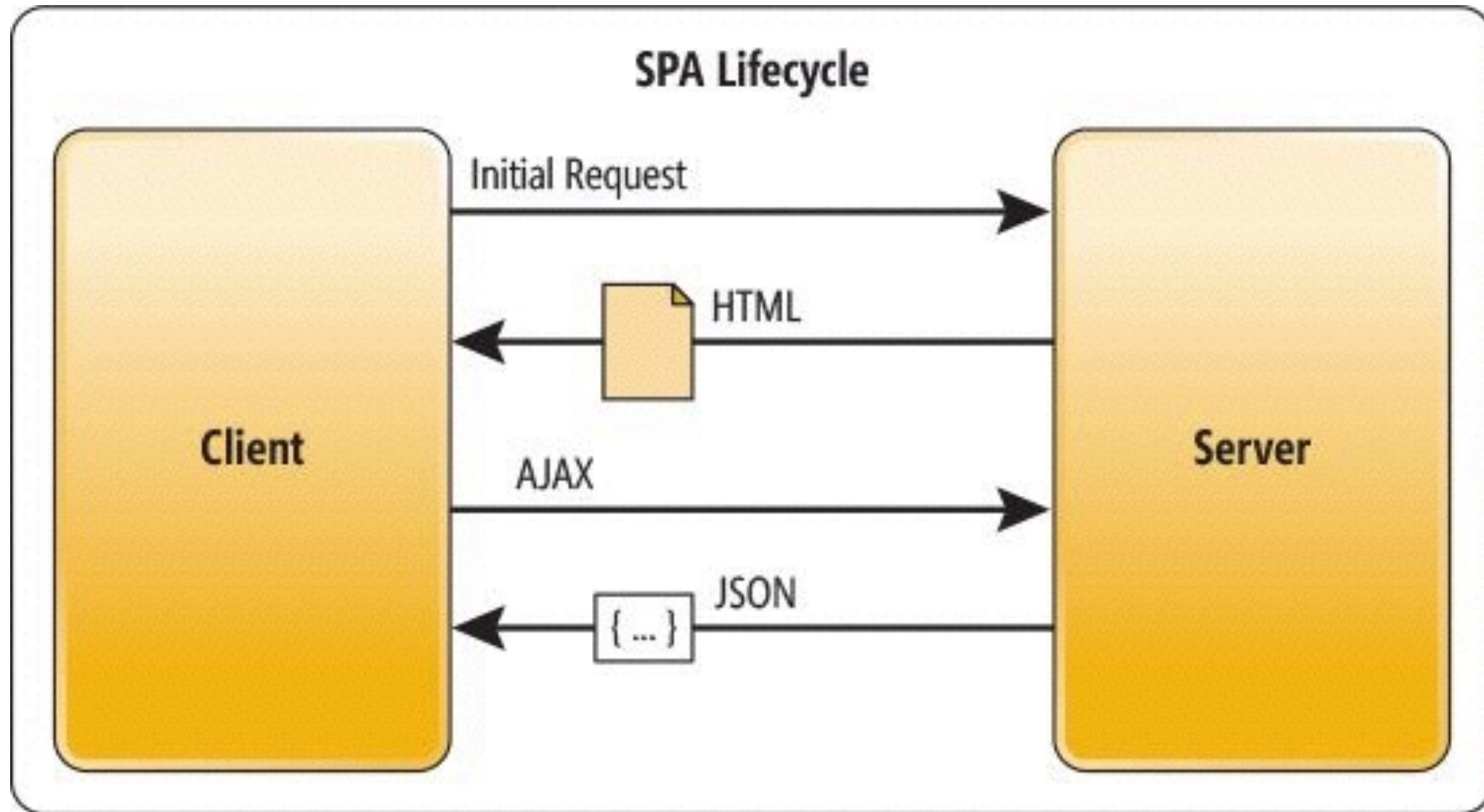
# Origen: de página a aplicación

El presente y futuro: Single Page Applications

- Actualmente: aplicaciones Web
- Cliente **construye** la aplicación completa con componentes, estado, acceso a datos...
- Para interactuar: peticiones asíncronas de datos
  - Ajax: HTTP Request + JSON / XML (RESTful APIs)
  - Websockets (Realtime APIs)
- Ejemplos: Gmail, Facebook, Google Maps...

# Origen: de página a aplicación

Interacción cliente <-> servidor



# Retos

- Las aplicaciones Web se parecen cada vez más a aplicaciones de escritorio
- Utilizamos elementos HTML como elementos básicos de UI: texto, botones, listas, formularios, etc.
- Utilizamos Javascript para manipular estos elementos, mediante el Document Object Model
- La UI reacciona a eventos
- Gestionamos diferentes pantallas (“páginas”), datos, estado...



# Retos

- El problema es que HTML es un lenguaje para crear documentos, no aplicaciones
- Además, existen diferencias en el DOM y las APIs existentes en Javascript entre navegadores
- Soluciones: ¡**frameworks**!

# Frameworks - greatest hits

- Javascript puro
- jQuery
- Knockout
- Backbone
- Angular
- Ember
- ...

How MVVM Frameworks proliferate:

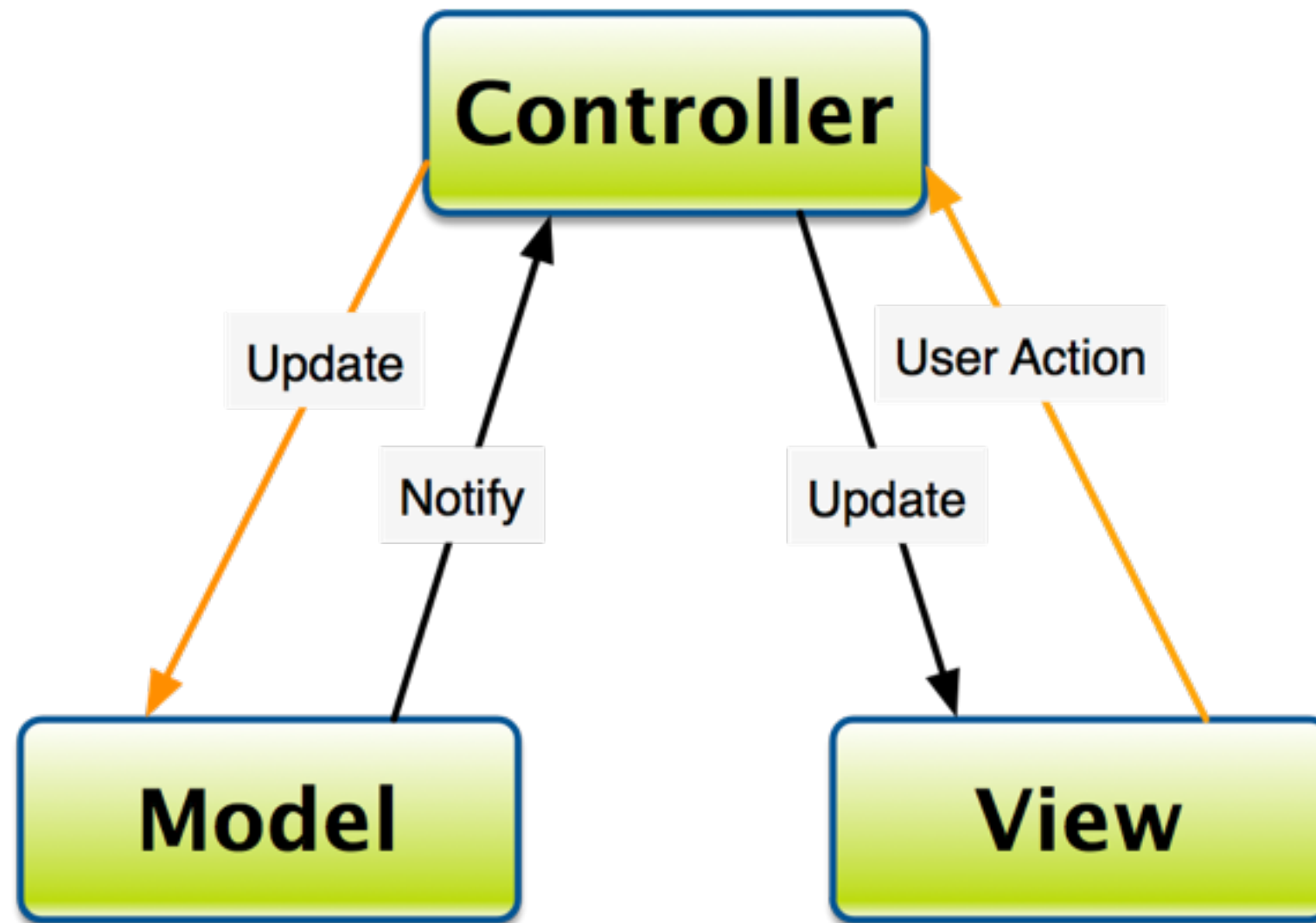


# Origen: de página a aplicación

Frameworks

- Mismas soluciones
  - Model View Controller (o variación)
  - Data-binding más o menos automático
  - Eventos “change”
  - Mismas ventajas / inconvenientes

# MVC



# MVC - problemas

- Tenemos diferentes lenguajes / mecanismos en cada punto
- Controlador -> Javascript en todo su esplendor
- Vista -> HTML o plantilla (Handlebars, moustache, underscore... con su propias expresiones y restricciones)
- Modelo -> su propio DSL para gestionar cambios, normalmente vía patrón Observer (evento “change”)

# MVC - problemas

- Acabamos normalmente con cascadas de eventos
- Difíciles de depurar, de comprender y seguir **su flujo** (¿quién ha cambiado qué? ¿por qué re-render? ¿cuándo? ¿cuántas veces?)
- La manipulación del DOM es costosa y compleja
- Aplicaciones grandes -> rendimiento mejorable

# React

# React

A JAVASCRIPT LIBRARY FOR BUILDING USER INTERFACES

[Get Started](#)

[Download React v15.0.1](#)

# React

- Librería para construir **interfaces de usuario**
- No es un framework, es una librería para UI



# React

- Nos va a ayudar **exclusivamente** con las Vistas
- NS/NC sobre datos, APIs REST, routing, arquitectura, organización
- Pero aporta una forma de razonar sobre la UI interesante e innovadora

# React

- Modificar el DOM es una operación **costosa**
- **Queremos** modificaciones selectivas
- **No** queremos escribir nosotros esa lógica
- Con React nos despreocupamos: que decida React
- ¿Cómo lo hace? Virtual DOM

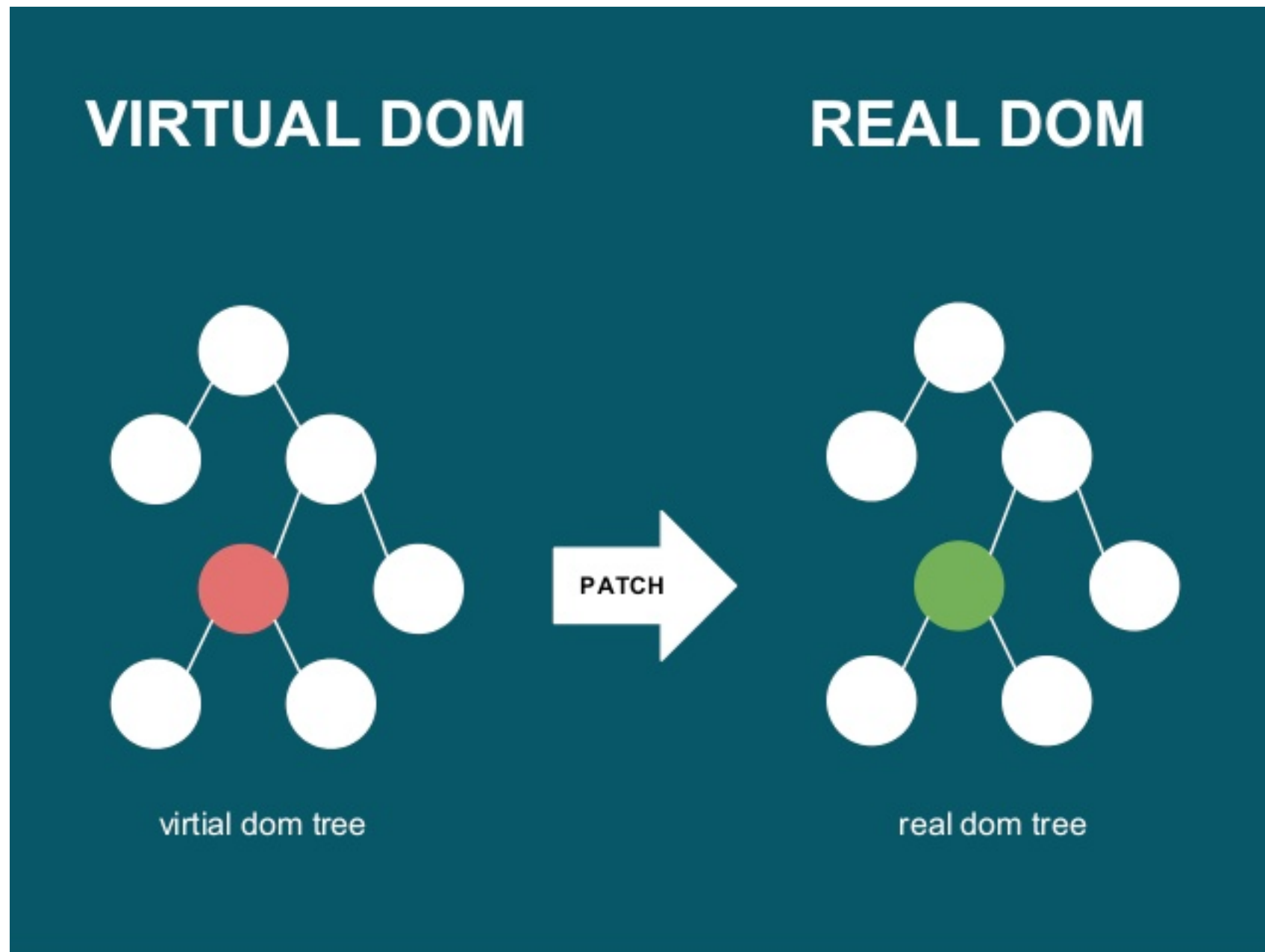
# Virtual DOM

- Los componentes de React no generan HTML directamente
- Los componentes se estructuran en forma de árbol con un único nodo raíz (la vista principal).
- Generan **código**: una descripción virtual del DOM
- Cuando React ejecuta render, se guarda esta descripción **en memoria**

# Virtual DOM

- En el próximo **render**, React compara la nueva **descripción** del árbol de componentes con la anterior para decidir **qué debe cambiar en el DOM real**, hasta el mínimo detalle
- React garantiza que se realizarán las mínimas operaciones necesarias en el DOM, de la forma más eficiente
- Para el programador funciona como un **render completo**, cada vez. **Simplicidad.**

# Virtual DOM



# Virtual DOM

- React nos proporciona una API que **parece** que **repinta** la aplicación completa con cada actualización
- Nuestros componentes definen **su representación en un momento dado**
- Una aplicación React es un componente React que incluye muchos otros -> composición

# React

- ¿Cómo hacer esto manteniendo una experiencia decente para el desarrollador?
- React utiliza una sintaxis especial, **JSX**
- Y **JSX** necesita un transpilador

# React

```
var HolaMundo = React.createClass({  
  render: function() {  
    ➔ return (  
      <div className='wrapper'>  
        <h1 className='title'>Hola mundo</h1>  
        <p>Mi primer componente React</p>  
      </div>  
    );  
  }  
});
```



# React

```
var HolaMundo = React.createClass({
  displayName: 'HolaMundo',

  render: function render() {
    ➔ return React.createElement(
      'div',
      { className: 'wrapper' },
      React.createElement(
        'h1',
        { className: 'title' },
        'Hola mundo'
      ),
      React.createElement(
        'p',
        null,
        'Mi primer componente React'
      )
    );
  }
});
```

# Herramientas

- Para transpilar JSX se utiliza **babel.js**
- Que además nos permite utilizar ES2015 (ES6) traduciéndolo a ES5 compatible con la mayoría de los navegadores

# Herramientas

**Babel is a JavaScript compiler.**

Use next generation JavaScript, today.

6.0.0 Released!



Star

11,431

# Herramientas

- Para hacer un desarrollo modular, utilizaremos **webpack**
- Esto nos permite separar nuestro código en archivos y carpetas, utilizar **npm** como gestor de dependencias...
- Y generar un **bundle**: único archivo JS con toda nuestra aplicación
- O varios **bundles** con carga dinámica, librerías comunes, etc.

# Ejercicio - infraestructura con Babel y Webpack

- Requisitos: **node.js y npm** instalados (versión 0.12, 4 o superior mejor)
- Recomendable: **nvm** (node version manager)  
Nos permite tener varias versiones simultáneas e instalar o desinstalar desde la línea de comandos

# Paso 1: npm

- En la carpeta que queramos, inicializamos nuestras dependencias con

➔ **npm init**

# Paso 2: dependencias

- Ahora que tenemos el **package.json** podemos ir añadiendo dependencias (y guardándolas) de producción y desarrollo:
  - ➔ **npm install --save <nombre\_dependencia>**  
(atajo: **npm i -S <dep> <dep> ...**)
  - ➔ **npm install --save-dev <nombre\_dependencia>**  
(atajo: **npm i -D <dep>**)

# Paso 3: dependencias

- Instalamos como dependencias:
- **react**
- **react-dom**



# Paso 4: dependencias de desarrollo

- Instalamos como dependencias de desarrollo
- **babel-core**
- **babel-loader**
- **babel-preset-es2015**
- **babel-preset-react**
- **webpack**

# Paso 5: configurar Babel

- Creamos un archivo **.babelrc** en nuestra carpeta
- Con esto indicamos a Babel que queremos que nos procese JSX (preset “react”) así como ES2015
- Más info: <http://babeljs.io/docs/plugins/#presets>

```
{  
  "presets": ["es2015", "react"]  
}
```

# Paso 6: configurar webpack

- Creamos un archivo **webpack.config.js** en nuestra carpeta

```
module.exports = {  
  entry: './src/app.js',  
  output: {  
    path: __dirname + '/dist',  
    filename: 'bundle.js'  
  },  
  module: {  
    loaders: [  
      {  
        test: /\.js$/,  
        exclude: /node_modules/,  
        loader: 'babel'  
      }  
    ]  
  }  
}
```

# Paso 7: crear nuestro index.html

- Creamos un archivo **index.html** en la carpeta **./dist/index.html**

```
<!doctype html>
<html>
<head>
  <title></title>
</head>
<body>
  <div id="app">
    Si ves esto, React no funciona
  </div>
  <script src="bundle.js"></script>
</body>
</html>
```

# Paso 8: crear nuestra app.js

- Creamos nuestra aplicación en **./src/app.js**

```
import React from 'react';
import ReactDOM from 'react-dom';

const HolaMundo = React.createClass({
  render() {
    return (
      <div>
        <h1>Hola mundo</h1>
      </div>
    )
  }
});

window.onload = function() {
  ReactDOM.render(<HolaMundo />, document.getElementById('app'));
}
```

# Paso 9: “compilar”

- Simplemente ejecutamos **webpack** para generar el bundle
- ➔ **./node\_modules/.bin/webpack**
- Webpack admite diferentes opciones:
- -w (Watch, recompila cuando detecta cambios)
- -d (Agrega source maps para depuración, **muy útil**)
- -p Crea un bundle minificado, uglificado y optimizado (muy lento)

# Paso 10: scripts npm

- Para que esto sea más cómodo, vamos a utilizar scripts de **npm** en lugar de **grunt/gulp**
- En **package.json**:

```
"scripts": {  
  "test": "echo \"Error: no test specified\" && exit 1",  
  "build": "webpack -p",  
  "start": "webpack -w -d"  
},
```

# Paso 11: fin

- Ahora podemos programar ejecutando en consola **npm start**
- Y abriendo **./dist/index.html** en el navegador
- Cualquier modificación re-compilará (con cache, muy rápido) nuestro bundle
- Cuando queramos una versión optimizada, ejecutaremos **npm run build**



# Paso 12: detalle de calidad

- Refrescar en cada cambio es bastante pesado
- Y desarrollar con URI tipo file:// tampoco es recomendable
- Nos gustaría tener un servidor Web de desarrollo que nos sirva y automáticamente nos refresque nuestra aplicación: **webpack-dev-server**
- Además, rizando el rizo, queremos que los cambios no reinicien la aplicación, sino que se apliquen “en caliente”: **react-hot-loader**


# Paso 13: webpack-dev-server

- ➔ **npm install --D react-hot-loader webpack-dev-server**
- A webpack-dev-server tenemos que decirle qué directorio queremos que nos sirva como “public” (por defecto, el directorio actual)
- ➔ **webpack-dev-server -d --content-base ./dist --hot --inline**
- Prepara el bundle con watch y nos sirve ./dist en <http://localhost:8080> con hot reloading

# Paso 14: react-hot-loader

- Si queremos mantener el estado de nuestra aplicación entre recargas, añadimos **react-hot** como cargador adicional en webpack.config.js:

```
module: {  
  loaders: [  
    {  
      test: /\.js*/,  
      include: path.join(__dirname, 'src'),  
      loaders: ['react-hot', 'babel']  
    }  
  ]  
}
```



# Paso 15: package.json final

```
{
  "name": "cursoredux",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "webpack -p",
    "start": "webpack-dev-server -d --hot --inline"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "react": "^15.0.2",
    "react-dom": "^15.0.2"
  },
  "devDependencies": {
    "babel-core": "^6.8.0",
    "babel-loader": "^6.2.4",
    "babel-preset-es2015": "^6.6.0",
    "babel-preset-react": "^6.5.0",
    "react-hot-loader": "^1.3.0",
    "webpack": "^1.13.0",
    "webpack-dev-server": "^1.14.1"
  }
}
```

# Paso 15: webpack.config.js

```
var path = require('path');

module.exports = {
  entry: './src/app.js',
  output: {
    path: path.join(__dirname, 'dist'),
    filename: 'bundle.js'
  },
  module: {
    loaders: [
      {
        loaders: ['react-hot', 'babel'],
        test: /\.js?$/,
        exclude: /node_modules/
      }
    ]
  },
  devServer: {
    contentBase: path.join(__dirname, 'dist')
  }
}
```

# Probar hot loading

- En la consola de Javascript, tenemos mensajes de "HMR" (hot module replacement)
- Modificar el texto en app.js y comprobar que cambia en el navegador
- Hot-loading no es perfecto y a veces tendremos que recargar la página, y otras él la recarga por nosotros
- Vigila la consola de webpack, los errores de sintaxis de JSX aparecen ahí!!!

# Más herramientas

- Un consejo: cuando desarrolles con React, ten la consola Javascript abierta
- React nos indica muchos errores y warnings con mensajes en consola
- Además, es muy recomendable instalar **React Dev Tools** (Chrome, Mozilla)
- Para los editores, buscar archivos de sintaxis de JSX (Sublime, Atom, WebStorm, etc... hay para todos)