



React

<https://facebook.github.io/react/>

© JMA 2016. All rights reserved

EVOLUCIÓN DEL DESARROLLO WEB

© JMA 2016. All rights reserved

1990

Cliente

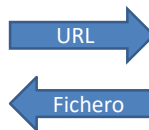
Tim Berners-Lee (CERN)

- ENQUIRE
 - Hipertexto
 - HTTP
 - HTML

Servidor

Robert Cailliau

- CERN httpd
 - Servidor de ficheros
 - HTTP – URL
 - Ficheros estáticos de texto



© JMA 2016. All rights reserved

Origen

- La Web no fue concebida para el desarrollo de aplicaciones. El problema que se pretendía resolver su inventor, Tim Berners-Lee, era el cómo organizar información a través de enlaces.
- De hecho la Web nació en el laboratorio de partículas CERN básicamente para agrupar un conjunto muy grande de información y datos del acelerador de partículas que se encontraba muy dispersa y aislada.
- Mediante un protocolo muy simple (HTTP), un sistema de localización de recursos (URL) y un lenguaje de marcas (HTML) se podía poner a disposición de todo científico en el mundo la información existente en el CERN de tal forma que mediante enlaces se pudiese acceder a información relacionada con la consultada.

© JMA 2016. All rights reserved

HTML

Cliente

- Navegadores
 - HTML

Servidor

- Servidor Web
 - Servidor de ficheros
 - Ficheros estáticos
 - Texto
 - HTML
 - Ficheros dinámicos
 - CGI (C, Perl)
 - ...
 - Servlet



© JMA 2016. All rights reserved

CGI

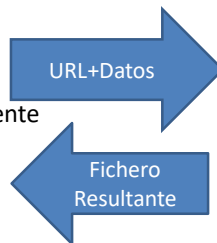
- Por la necesidad que el servidor Web pudiese devolver páginas Web dinámicas y no únicamente contenido estático residente en ficheros HTML se desarrolló la tecnología CGI (Common Gateway Interface) donde el servidor Web invocaba un programa el cual se ejecutaba, devolvía la página Web y el servidor Web remitía este flujo de datos al navegador.
- Un programa CGI podía ser cualquier programa que la máquina pudiese ejecutar: un programa en C, o en Visual Basic o en Perl. Normalmente se elegía este último por ser un lenguaje de script el cual podía ser traslado con facilidad de una arquitectura a otra. CGI era únicamente una pasarela que comunicaba el servidor Web con el ejecutable que devolvía la página Web.

© JMA 2016. All rights reserved

Formularios

Cliente

- Navegadores
 - HTML 3.2
 - Formularios
 - Plug-in
 - Applet
 - ActiveX
 - Scripting de cliente
 - JavaScript



Servidor

- Servidor Web
 - Servidor de ficheros
 - Ficheros estáticos
 - Texto, HTML, Imágenes, ...
 - Ficheros dinámicos
 - Scripting de servidor
 - ASP
 - PHP
 - ...
 - JSP

© JMA 2016. All rights reserved

Scripting

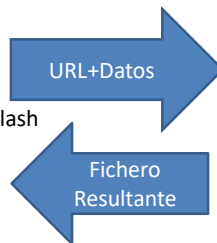
- CGI era una solución cómoda de realizar páginas Web dinámicas pero tenía un grave problema de rendimiento que lo hizo insostenible en cuanto la demanda de la Web comenzó a disparar las peticiones de los servidores Web.
- Para agilizar esto, los principales servidores Web del momento (Netscape e IIS) desarrollaron un sistema para la ejecución dinámica de aplicaciones usando el propio contexto del servidor Web. En el caso de Netscape se le denominó NSAPI (Netscape Server Application Program Interface) y en el caso de IIS se le llamó ISAPI.

© JMA 2016. All rights reserved

Web 2.0

Cliente

- Navegadores
 - HTML 4
 - CSS
 - DOM
 - AJAX
 - RIA
 - Shockwave Flash
 - Silverlight
 - JS Framework



Servidor

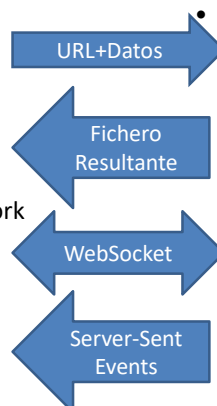
- Servidor Web
 - Servidor de ficheros
 - Ficheros estáticos
 - Ficheros dinámicos
 - Sricpting de servidor
 - Servidor de aplicaciones
 - ASP.NET
 - J2EE
 - Web Services
 - WS XML
 - RestFul

© JMA 2016. All rights reserved

Actualidad

Cliente

- Navegadores
 - HTML 5
 - CSS 3
 - ~~RIA~~
 - Móviles
 - JS RIA Framework
 - EcmaScript 6



Servidor

- Servidor Web
 - Servidor de ficheros
 - Ficheros estáticos
 - Ficheros dinámicos
 - Sricpting de servidor
 - Servidor de aplicaciones
 - NodeJS
 - Web Services
 - Server-Sent Events
 - Notificaciones PUSH

© JMA 2016. All rights reserved

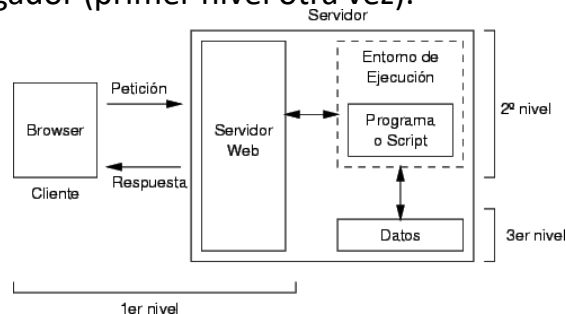
Aplicación Web

- Conjunto de páginas que residen en un directorio web y sus subdirectorios.
- Cualquier página puede ser el punto de entrada de la aplicación, no se debe confundir con el concepto de página principal con el de página por defecto.
- Externamente, no existe el concepto de aplicación como entidad única.
- Internamente, dependiendo de la tecnología utilizada una aplicación puede ser una entidad única o una colección de objetos independientes.

© JMA 2016. All rights reserved

Arquitectura

- Una aplicación Web típica recogerá datos del usuario (primer nivel), los enviará al servidor, que ejecutará un programa (segundo y tercer nivel) y cuyo resultado será formateado y presentado al usuario en el navegador (primer nivel otra vez).



© JMA 2016. All rights reserved

Ventajas e inconvenientes

Ventajas

- Inmediatez y accesibilidad
- No ocupan espacio local
- Actualizaciones inmediatas
- No hay problemas de compatibilidad
- Multiplataforma
- Consumo de recursos bajo
- Portables
- Alta escalabilidad y disponibilidad

Inconvenientes

- Interfaz de interacción con el usuario muy limitada
- Base tecnológica inadecuada
- Incompatibilidades entre navegadores
- Bajo rendimiento al ser interpretado
- Cesión tecnológica
- Seguridad menos robusta

© JMA 2016. All rights reserved

Single-page application (SPA)

- Un single-page application (SPA), o aplicación de página única es una aplicación web o es un sitio web que utiliza una sola página con el propósito de dar una experiencia más fluida a los usuarios como una aplicación de escritorio.
- En un SPA todo el código de HTML, JavaScript y CSS se carga de una sola vez o los recursos necesarios se cargan dinámicamente cuando lo requiera la página y se van agregando, normalmente como respuesta de los acciones del usuario.
- La página no se tiene que cargar otra vez en ningún punto del proceso, tampoco se transfiere a otra página, aunque las tecnologías modernas (como el `pushState()` API del HTML5) pueden permitir la navegabilidad en páginas lógicas dentro de la aplicación.
- La interacción con las aplicaciones de página única pueden involucrar comunicaciones dinámicas con el servidor web que está por detrás, habitualmente utilizando AJAX o WebSocket (HTML5).

© JMA 2016. All rights reserved

Estado actual de la adopción de los nuevos estándares

- <http://caniuse.com/>
- HTML 5
 - <http://html5test.com>
 - <http://html5demos.com>
- CSS
 - <http://css3test.com/>
- EcmaScript
 - <https://kangax.github.io/compat-table/es6/>
- Navegadores mas utilizados
 - <http://www.netmarketshare.com/>

© JMA 2016. All rights reserved

HERRAMIENTAS DE DESARROLLO

© JMA 2016. All rights reserved

IDEs

- Visual Studio Code - <http://code.visualstudio.com/>
 - VS Code is a Free, Lightweight Tool for Editing and Debugging Web Apps.
 - Extensiones: Web Dev Pack (JS/HTML/CSS), React Native Tools, ES7 React/Redux/GraphQL/React-Native snippets, Debugger for Chrome, EditorConfig for VS Code, ...
- StackBlitz - <https://stackblitz.com>
 - The online IDE for web applications. Powered by VS Code and GitHub.
- Eclipse - <https://www.eclipse.org/>
 - Eclipse IDE for Enterprise Java Developers, Eclipse IDE for JavaScript and Web Developers.
 - Marketplace > React :: CodeMix
- Webstorm - <https://www.jetbrains.com/webstorm/>
 - Lightweight yet powerful IDE, perfectly equipped for complex client-side development and server-side development with Node.js

© JMA 2016. All rights reserved

Instalación de utilidades

Consideraciones previas

- Las utilidades son de línea de comandos.
- Para ejecutar los comandos es necesario abrir la consola comandos (Símbolo del sistema)
- Siempre que se realice una instalación o creación es conveniente “Ejecutar como Administrador” para evitar otros problemas.
- En algunos casos el firewall de Windows, la configuración del proxy y las aplicaciones antivirus pueden dar problemas.

GIT: Software de control de versiones

- Descargar e instalar: <https://git-scm.com/>
- Verificar desde consola de comandos:
 - git

Node.js: Entorno en tiempo de ejecución

- Descargar e instalar: <https://nodejs.org>
- Verificar desde consola de comandos:
 - node --version

© JMA 2016. All rights reserved

npm: Node Package Manager

- Aunque se instala con el Node es conveniente actualizarlo:
 - `npm update -g npm`
- Verificar desde consola de comandos:
 - `npm --version`
- Configuración:
 - `npm config edit`
 - `proxy=http://usr:pwd@proxy.dominion.com:8080` ← Símbolos: %HEX ASCII
- Generar fichero de dependencias `package.json`:
 - `npm init`
- Instalación de paquetes:
 - `npm install -g grunt-cli karma karma-cli` ← Global (CLI)
 - `npm install jasmine-core tslint --save --save-dev`
 - `npm install` ← Dependencias en `package.json`
- Arranque del servidor:
 - `npm start`

© JMA 2016. All rights reserved

Generación del esqueleto de aplicación

- Configurar un nuevo proyecto de React puede ser un proceso complicado y tedioso, con tareas como:
 - Crear la estructura básica de archivos y bootstrap
 - Configurar Browserify o WebPack para transpilar el código
 - Crear scripts para ejecutar el servidor de desarrollo, tester, publicación, ...
- Disponemos de diferentes opciones de asistencia:
 - Proyectos semilla (seed) disponibles en github
 - Generadores basados en Yeoman
 - Herramienta oficial de gestión de proyectos: `create-react-app`.
- Creada por el equipo de React, `create-react-app` es una *Command Line Interface* que permite generar proyectos desde consola, así como ejecutar un servidor de desarrollo o lanzar los tests de la aplicación (la instalación es opcional y no se recomienda).
 - `npm install -g create-react-app`

© JMA 2016. All rights reserved

create-react-app

- Dependencia única: solo hay una dependencia de compilación. Utiliza Webpack, Babel, ESLint y otros proyectos increíbles, pero ofrece una experiencia coherente combinando todos ellos.
- No requiere configuración: no se necesita configurar nada. Se maneja una configuración razonablemente buena de compilaciones tanto de desarrollo como de producción para poderse concentrar en la escritura de código.
- Sin ataduras: se puede "expulsar" a la configuración personalizada en cualquier momento. Ejecuta un solo comando, y todas las dependencias de configuración y compilación se moverán directamente al proyecto, para que pueda reconfigurar como se desee.

© JMA 2016. All rights reserved

Características

- Soporte de sintaxis React, JSX, ES6 y Flow.
- Extras de lenguaje más allá del ES6 como el operador de propagación de objetos.
- Un servidor de desarrollo que busca errores comunes.
- Importa CSS e imágenes directamente desde JavaScript.
- Autoprefijado CSS, por lo que no se necesita -webkit u otros prefijos.
- Un comando para agrupar JS, CSS e imágenes para producción, con los mapeos de origen.
- Un service worker sin conexión y un manifiesto de aplicación web que cumple todos los criterios de una aplicación web progresiva .

© JMA 2016. All rights reserved

Creación y puesta en marcha

- Acceso al listado de comandos y opciones
 - <https://github.com/facebookincubator/create-react-app>
- Nuevo proyecto
 - \$ create-react-app myApp
 - \$ npx create-react-app myApp
 - Esto creará la carpeta myApp con un esqueleto de proyecto ya montado según la última versión de React y todo el trabajo sucio de configuración de WebPack para transpilar el código y generar un bundle, configuración de tests, lint, etc.
 - Además, instala todas las dependencias necesarias de npm.
- Servidor de desarrollo
 - \$ npm start
 - Esto lanza tu app en la URL <http://localhost:3000> y actualiza el contenido cada vez que guardas algún cambio.

© JMA 2016. All rights reserved

Estructura de directorios de soporte

- | | |
|---|---|
| <ul style="list-style-type: none"> 📁 public <ul style="list-style-type: none"> 📄 favicon.ico 📄 index.html 📄 manifest.json 📁 src <ul style="list-style-type: none"> 📄 App.css 📄 App.js 📄 App.test.js 📄 index.css 📄 index.js 📄 logo.svg 📄 registerServiceWorker.js 📄 .gitignore 📄 package.json 📄 README.md | <ul style="list-style-type: none"> • src: Fuentes de la aplicación • public: Ficheros públicos de la aplicación. • Ficheros de configuración de librerías y herramientas • Posteriormente aparecerán los siguientes directorios: <ul style="list-style-type: none"> – node_modules: Librerías y herramientas descargadas – build: Resultado para publicar en el servidor web |
|---|---|

© JMA 2016. All rights reserved

Pruebas y despliegue

\$ npm test

- Inicia el entorno de pruebas en modo interactivo.

\$ npm run build

- Genera la versión para producción en la carpeta /build, agrupando y minimizando para obtener un mejor rendimiento. Incluyen valores hash en los nombres de los archivos para impedir los problemas de la cache con el versionado.
- De forma predeterminada, también incluye un service worker para cargar desde la memoria caché local en las futuras visitas.

\$ npm run eject

- Elimina la herramienta y dependencias de compilación, después de copiar los archivos de configuración en el directorio de la aplicación.
- Una vez hecho no tiene vuelta atrás.

© JMA 2016. All rights reserved

Agregar a una aplicación existente

• Añadir las utilidades para desarrollar

- Un administrador de paquetes, como Yarn o npm. Permite aprovechar un vasto ecosistema de paquetes de terceros e instalarlos o actualizarlos fácilmente.
- Crear paquete de configuración:
 - npm init
- Un empaquetador, como webpack o Browserify. Permite escribir código modular y agruparlo en pequeños paquetes para optimizar el tiempo de carga.
 - npm install webpack webpack-dev-server --save
- Un compilador como Babel. Permite escribir código JavaScript moderno que aún funciona en navegadores más antiguos.
 - npm install -g babel babel-cli
 - npm install babel-core babel-loader babel-preset-react babel-preset-es2015 --save

© JMA 2016. All rights reserved

Webpack

- Webpack (<https://webpack.github.io/>) es un empaquetador de módulos, es decir, permite generar un archivo único con todos aquellos módulos que necesita la aplicación para funcionar.
- Toma módulos con dependencias y genera archivos estáticos correspondientes a dichos módulos.
- Webpack va mas allá y se ha convertido en una herramienta muy versátil. Entre otras cosas, destaca que:
 - Puede generar solo aquellos fragmentos de JS que realmente necesita cada página.
 - Dividir el árbol de dependencias en trozos cargados bajo demanda
 - Haciendo más rápida la carga inicial
 - Tiene varios loaders para importar y empaquetar también otros recursos (CSS, templates, ...) así como otros lenguajes (ES6 con Babel, TypeScript, SaSS, etc).
 - Sus plugins permiten hacer otras tareas importantes como por ejemplo minimizar y ofuscar el código.

© JMA 2016. All rights reserved

webpack.config.js

```
var config = {
  entry: './main.js',
  output: {
    path: './',
    filename: 'index.js',
  },
  devServer: {
    inline: true,
    port: 3000
  },
  module: {
    loaders: [
      {
        test: /\.jsx?$/,
        exclude: /node_modules/,
        loader: 'babel',
        query: { presets: ['es2015', 'react'] }
      }
    ]
  }
}
module.exports = config;
```

En package.json:

```
"scripts": {
  "start": "webpack-dev-server --hot",
  "test": "echo \"Error: no test specified\" && exit 1"
},
```

© JMA 2016. All rights reserved

Añadir React a una página

- Se pueden utilizar dos estrategias diferentes:
 - Incluir ficheros locales combinables en un bundle.
 - Incluir ficheros compartidos en una CDN (Content Delivery Network).
- Para descargar los ficheros:
 - `npm install react react-dom --save`
- Las CDN permiten compartir contenidos comunes entre diferentes sitios y evitar descargas al aprovechar la cache de los navegadores:
 - Versión de producción: se encuentra minimizada y compactada para reducir al máximo su tamaño.


```
<script crossorigin src="https://unpkg.com/react@16/umd/react.production.min.js"></script>
<script crossorigin src="https://unpkg.com/react-dom@16/umd/react-dom.production.min.js"></script>
```
 - Versión de desarrollo: para desarrollo y pruebas sin comprimir y depurable.


```
<script crossorigin src="https://unpkg.com/react@16/umd/react.development.js"></script>
<script crossorigin src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
```
 - Es conveniente incluirlos después de las hojas de estilos para asegurar la importación de todos los estilos.
 - El atributo `crossorigin` fuerza a verificar que el CDN está utilizando la cabecera `Access-Control-Allow-Origin: *`

© JMA 2016. All rights reserved

Aplicación

index.html

```
<!DOCTYPE html>
<html lang = "es">

  <head>
    <meta charset = "UTF-8">
    <title>React App</title>
  </head>

  <body>
    <div id = "app"></div>
    <script src = "index.js"></script>
  </body>

</html>
```

App.jsx

```
import React from 'react';
class App extends React.Component {
  render() {
    return (
      <div>Hola Mundo!!!</div>
    );
  }
}
export default App;
```

main.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.jsx';

ReactDOM.render(<App />,
  document.getElementById('app'));
```

© JMA 2016. All rights reserved

React Developer Tools

<https://github.com/facebook/react-devtools>

- React Developer Tools es una extensión de Chrome y Firefox para la biblioteca JavaScript de código abierto React.
- Muestra los componentes raíz de React que se representaron en la página, así como los subcomponentes que se terminaron representando.
- Al seleccionar uno de los componentes en el árbol, puede inspeccionar y editar sus propiedades y el estado actuales en el panel de la derecha. En breadcrumbs se puede inspeccionar el componente seleccionado, el componente que lo creó, el componente que creó a este último, y así sucesivamente.

© JMA 2016. All rights reserved

Web Worker

- Cuando se ejecuta una secuencia de comandos en una página HTML, la página no responde hasta que finalice la ejecución.
- La especificación de Web Workers recomienda un API para generar secuencias de comandos en segundo plano en la aplicación web.
- Los Web Workers permiten realizar acciones como activar secuencias de comandos con tiempos de ejecución largos para gestionar tareas intensivas de cálculo, pero sin bloquear la interfaz de usuario u otras secuencias de comandos para gestionar las interacciones del usuario.
- El usuario puede seguir haciendo lo que quiere: hacer clic, la selección de las etiquetas, etc., mientras que el Web Worker se ejecuta en segundo plano.
- Dado que los Web Workers están en archivos externos, no tienen acceso a los objetos JavaScript (hilo principal del JavaScript): DOM, window, document y parent.
- Esta limitación se puede solventar con el evento onmessage del objeto Web Worker, enviando los datos de salida del Web Worker en el event.data.

© JMA 2016. All rights reserved

Configurar VS Code

- Visualización de la salida Lint en el editor:
 - `.eslintrc.json`

```
{
  "extends": "react-app"
}
```
- Depuración en el editor:
 - `.vscode/launch.json`

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Chrome",
      "type": "chrome",
      "request": "launch",
      "url": "http://localhost:3000",
      "webRoot": "${workspaceFolder}/src",
      "sourceMapPathOverrides": {
        "webpack:///src/*": "${webRoot}/*"
      }
    }
  ]
}
```
- Formato de código automático cada vez que se hace un commit en git
 - `npm install --save husky lint-staged prettier`

© JMA 2016. All rights reserved

Instalar dependencias

- El proyecto generado incluye React y ReactDOM como dependencias.
- También incluye un conjunto de scripts utilizados por Create React App como dependencias de desarrollo.
- Se puede instalar otras dependencias (por ejemplo, React Router o Redux) con npm:
 - `npm install --save react-router-dom`
 - `npm i -S redux react-redux`
 - `npm i -D redux-devtools`
- Añadir Bootstrap
 - Dependencia:
 - `npm install --save bootstrap jquery popper.js reactstrap`
 - En `src/index.js`:
 - `import 'bootstrap/dist/css/bootstrap.css';`

© JMA 2016. All rights reserved

Entorno

- No se debe almacenar ningún secreto (como las claves privadas) en una aplicación React.
- El proyecto puede consumir variables declaradas en el entorno como si estuvieran declaradas localmente en sus archivos JS. De forma predeterminada, está definida `NODE_ENV` y cualquier otra variable de entorno que comience con `REACT_APP_`.
- Las variables son accesibles mediante:


```
if (process.env.NODE_ENV !== 'production')
  <input type="hidden" value={process.env.REACT_APP_SECRET} />
<title>%REACT_APP_WEBSITE_NAME%</title>
```
- Para definir variables de entorno permanentes se pueden crear archivos `.env` en la raíz del proyecto (las versiones local deberían excluirse en `.gitignore`):
 - `.env`: Defecto.
 - `.env.local`: Anulaciones locales (excepto prueba).
 - `.env.development`, `.env.test`, `.env.production`: Configuración específica del entorno.
 - `.env.development.local`, `.env.test.local`, `.env.production.local`: Versiones locales.
- Los archivos de la izquierda tienen más prioridad que los archivos de la derecha:
 - `npm start`: `.env.development.local`, `.env.development`, `.env.local`, `.env`
 - `npm run build`: `.env.production.local`, `.env.production`, `.env.local`, `.env`
 - `npm test`: `.env.test.local`, `.env.test`, `.env` (No utiliza `.env.local`)

© JMA 2016. All rights reserved

JavaScript 6: <http://www.ecma-international.org/ecma-262/6.0/>

ECMAScript 2015

© JMA 2016. All rights reserved

Introducción

- En Junio de 2015 aparece la 6ª edición del estándar ECMAScript (ES6), cuyo nombre oficial es ECMAScript 2015.
- Se está extendiendo progresivamente el soporte a ES6, pero aún queda trabajo por hacer.
- Las alternativas para empezar a utilizarlo son:
 - Transpiladores ("Transpiler": "Translator" y "Compiler"): Traducen o compilan un lenguaje de alto nivel a otro lenguaje de alto nivel, en este caso código ES6 a ES5. Los más conocidos y usados son Babel, TypeScript, Google Traceur, CoffeeScript.
 - Polyfill: Un trozo de código o un plugin que permite tener las nuevas funcionalidades de HTML5 o ES6 en aquellos navegadores que nativamente no lo soportan.

© JMA 2016. All rights reserved

EcmaScript 6

- <http://www.ecma-international.org/ecma-262/6.0/>
- <http://es6-features.org/>
- <https://kangax.github.io/compat-table/es6/>
- <https://babeljs.io/>
- <https://github.com/google/traceur-compiler>
- <https://www.typescriptlang.org/>
- <https://github.com/zloirock/core-js>

© JMA 2016. All rights reserved

Declaración de variables

- **let**: ámbito de bloque, solo accesible en el bloque donde está declarada.


```
(function() {
  if(true) {
    let x = "variable local";
  }
  console.log(x); // error, "x" definida dentro del "if"
})();
```
- **const**: constante, se asina valor al declararla y ya no puede cambiar de valor.


```
const PI = 3.15;
PI = 3.14159; // error, es de sólo-lectura
```
- Las constantes numérica ahora se pueden expresar en binario y octal.


```
0b111110111 === 503
0o767 === 503
```

© JMA 2016. All rights reserved

Template Strings

- **Interpolación**: sustituye dentro de la cadena las variable por su valor:


```
let var1 = "JavaScript";
let var2 = "Templates";
console.log(`El ${var1} ya tiene ${var2}.`);
// El JavaScript ya tiene Templates.
```
- Las constantes cadenas pueden ser multilínea sin necesidad de concatenarlos con **+**.


```
let var1 = " El JavaScript
ya tiene
Templates ";
```
- Incorpora soporte extendido para el uso de Unicode en cadenas y expresiones regulares (las cadenas pueden contener caracteres en hebreo, árabe, cirílico, ...).

© JMA 2016. All rights reserved

Destructuring

- Asignar (repartir) los valores de un objeto o array en varias variables:

```
var tab = ["hola", "adiós"];
var [a, b] = tab;
console.log(a); // "hola"
console.log(b); // "adiós"
[ b, a ] = [ a, b ]
```

```
var obj = { nombre: "Pepito", apellido: "Grillo" };
var { nombre, apellido } = obj;
console.log(nombre); // "Pepito"
```

© JMA 2016. All rights reserved

Parámetros de funciones

- Valores por defecto: Se pueden definir valores por defecto a los parámetros en las funciones.

```
function(valor = "foo") {...};
```

- Resto de los parámetros: Convierte una lista de parámetros en un array.

```
function f (x, y, ...a) {
  return (x + y) * a.length
}
f(1, 2, "hello", true, 7) === 9
```

- Operador de propagación: Convierte un array o cadena en una lista de parámetros.

```
var str = "foo"
var chars = [ ...str ] // [ "f", "o", "o" ]
```

© JMA 2016. All rights reserved

Función Arrow

- Funciones anónimas.

```
data.forEach(elem => {
  console.log(elem);
  // ...
});
var fn = (num1, num2) => num1 + num2;
pairs = evens.map(v => ({ even: v, odd: v + 1 }));
fn = () => {console.log("Error");};
```
- this: dentro de una función Arrow hace referencia al contenedor y no al contexto de la propia función.

```
bar : function() {
  document.addEventListener("click", (e) => this.foo());
}
```
- equivale a (ES5):

```
bar : function() {
  document.addEventListener("click", function(e) {
    this.foo();
  }).bind(this);
}
```

© JMA 2016. All rights reserved

Propiedades de objetos

ES 2015

```
obj = { x, y }
obj = {
  foo (a, b) {
    ...
  },
  bar (x, y) {
    ...
  },
  *iter (x, y) {
    ...
  }
}
```

Anteriormente:

```
obj = { x: x, y: y };
obj = {
  foo: function (a, b) {
    ...
  },
  bar: function (x, y) {
    ...
  },
  // iter: sin equivalencia
};
```

© JMA 2016. All rights reserved

Clases

- Ahora JavaScript tendrá clases, muy parecidas las funciones constructoras de objetos que realizábamos en el estándar anterior, pero ahora bajo el paradigma de clases, con todo lo que eso conlleva, como por ejemplo, herencia.

```
class LibroTecnico extends Libro {
  constructor(tematica, paginas) {
    super(tematica, paginas);
    this.capitulos = [];
    this.precio = "";
    // ...
  }
  metodo() {
    // ...
  }
}
```

© JMA 2016. All rights reserved

Static Members

```
class Rectangle extends Shape {
  ...
  static defaultRectangle () {
    return new Rectangle("default", 0, 0, 100, 100)
  }
}
class Circle extends Shape {
  ...
  static defaultCircle () {
    return new Circle("default", 0, 0, 100)
  }
}
var defRectangle = Rectangle.defaultRectangle()
var defCircle   = Circle.defaultCircle()
```

© JMA 2016. All rights reserved

Getter/Setter

```
class Rectangle {
  constructor (width, height) {
    this._width = width;
    this._height = height;
  }
  set width (width) { this._width = width; }
  get width () { return this._width; }
  set height (height) { this._height = height; }
  get height () { return this._height; }
  get area () { return this._width * this._height; }
}
var r = new Rectangle(50, 20)
if(r.area === 1000) r.width = 10;
```

© JMA 2016. All rights reserved

mixin

- Soporte para la herencia de estilo mixin mediante la ampliación de las expresiones que producen objetos de función.

```
var aggregation = (baseClass, ...mixins) => {
  let base = class _Combined extends baseClass {
    constructor (...args) {
      super(...args)
      mixins.forEach((mixin) => {
        mixin.prototype.initializer.call(this)
      })
    }
  }
  let copyProps = (target, source) => {
    Object.getOwnPropertyNames(source)
      .concat(Object.getOwnPropertySymbols(source))
      .forEach((prop) => {
        if (prop.match(/^?(?:constructor|prototype|arguments|caller|name|bind|call|apply|toString|length)$/))
          return
        Object.defineProperty(target, prop, Object.getOwnPropertyDescriptor(source, prop))
      })
  }
  mixins.forEach((mixin) => {
    copyProps(base.prototype, mixin.prototype)
    copyProps(base, mixin)
  })
  return base
}
```

© JMA 2016. All rights reserved

Módulos

- Estructura el código en módulos similares a los espacios de nombres
- Cada fichero se comporta como un módulo.
- Solo las partes marcadas como export pueden ser importadas en otros ficheros.
- Se puede llamar a las funciones desde los propios Scripts, sin tener que importarlos en el HTML, si usamos JavaScript en el navegador.

```
//File: lib/person.js
export function hello(nombre) {
  return nombre;
}
Y para importar en otro fichero:
//File: app.js
import { hello } from "lib/person";
var app = {
  foo: function() {
    hello("Carlos");
  }
}
export app;
```

© JMA 2016. All rights reserved

Iteradores y Generadores

- El patrón Iterador permite trabajar con colecciones por medio de abstracciones de alto nivel
- Un Iterador es un objeto que sabe como acceder a los elementos de una secuencia, uno cada vez, mientras que mantiene la referencia a su posición actual en la secuencia.
- En ES2015 las colecciones (arrays, maps, sets) son objetos iteradores.
- Los generadores permiten la implementación del patrón Iterador.
- Los Generadores son funciones que pueden ser detenidas y reanudadas en otro momento.
- Estas pausas en realidad ceden la ejecución al resto del programa, es decir no bloquean la ejecución.
- Los Generadores devuelven (generan) un objeto "Iterator" (iterador)

© JMA 2016. All rights reserved

Generadores

- Para crear una función generadora


```
function* myGenerator() {
  // ...
  yield value;
  // ...
}
```
- La instrucción `yield` devuelve el valor y queda a la espera de continuar cuando se solicite el siguiente valor.
- El método `next()` ejecuta el generador hasta el siguiente `yield` dentro del mismo y devuelve un objeto con el valor.


```
var iter = myGenerator();
// ...
rslt = iter.next();
// ...
```
- La nueva sintaxis del `for` permite recorrer el iterador completo:


```
for (let value of iter)
```

© JMA 2016. All rights reserved

Nuevos Objetos

- **Map**: Lista de pares clave-valor.
- **Set**: Colección de valores únicos que pueden ser de cualquier tipo.
- **WeakMap**: Colección de pares clave-valor en los que cada clave es una referencia de objeto.
- **WeakSet**: Colección de objetos únicos.
- **Promise**: Proporciona un mecanismo para programar el trabajo de modo que se lleve a cabo en un valor que todavía no se calculó.
- **Proxy**: Habilita el comportamiento personalizado de un objeto.
- **Reflect**: Proporciona métodos para su uso en las operaciones que se interceptan.
- **Symbol**: Permite crear un identificador único.
- **Intl.Collator**: Proporciona comparaciones de cadenas de configuración regional.
- **Intl.DateTimeFormat**: Proporciona formato de fecha y hora específico de la configuración regional.
- **Intl.NumberFormat**: Proporciona formato de número específico de la configuración regional.

© JMA 2016. All rights reserved

Nuevos Objetos

- **ArrayBuffer**: Representa un búfer sin formato de datos binarios, que se usa para almacenar datos de las diferentes matrices con tipo. No se puede leer directamente de **Arraybuffer** ni escribir directamente en **Arraybuffer**, pero se puede pasar a una matriz con tipo o un objeto **DataView** para interpretar el búfer sin formato según sea necesario.
- **DataView**: Se usa para leer y escribir diferentes tipos de datos binarios en cualquier ubicación de **ArrayBuffer**.
- **Float32Array**: Matriz con tipo de valores flotantes de 32 bits.
- **Float64Array**: Matriz con tipo de valores flotantes de 64 bits.
- **Int8Array**: Matriz con tipo de valores enteros de 8 bits.
- **Int16Array**: Matriz con tipo de valores enteros de 16 bits.
- **Int32Array**: Matriz con tipo de valores enteros de 32 bits.
- **Uint8Array**: Matriz con tipo de valores enteros sin signo de 8 bits.
- **Uint8ClampedArray**: Matriz con tipo de enteros sin signo de 8 bits con valores fijos.
- **Uint16Array**: Matriz con tipo de valores enteros sin signo de 16 bits.
- **Uint32Array**: Matriz con tipo de valores enteros sin signo de 32 bits.

© JMA 2016. All rights reserved

Promise Pattern

- El **Promise Pattern** es un patrón de organización de código que permite encadenar llamadas a métodos que se ejecutaran a la conclusión del anterior (flujos).
- Simplifica y soluciona los problemas comunes con el patrón **Callback**:
 - Llamadas anidadas
 - Complejidad de código
$$o.m(1, 2, f(m1(3, f1(4,5,ff(8)))) \rightarrow o.m(1, 2).f().m1(3).f1(4, 5).ff(8)$$
- Aunque se utiliza extensamente para las operaciones asíncronas, no es exclusivo de las mismas.
- El servicio **\$q** es un servicio de **AngularJS** que contiene toda la funcionalidad de las promesas (está basado en la implementación **Q** de **Kris Kowal**).
- La librería **jQuery** incluye el objeto **\$.Deferred** desde la versión 1.5.
- Las promesas se han incorporado a los objetos estándar de **JavaScript** en la versión 6.

© JMA 2016. All rights reserved

Objeto Promise

- Una “promesa” es un objeto que actúa como proxy en los casos en los que no se puede utilizar el verdadero valor porque aún no se conoce (no se ha generado, llegado, ...) pero se debe continuar sin esperar a que este disponible (no se puede bloquear la función esperando a su obtención).
- Una “promesa” puede tener los siguientes estados:
 - Pendiente: Aún no se sabe si se podrá o no obtener el resultado.
 - Resuelta: Se ha podido obtener el resultado (Promise.resolve())
 - Rechazada: Ha habido algún tipo de error y no se ha podido obtener el resultado (Promise.reject())
- Los métodos del objeto promesa devuelven al propio objeto para permitir apilar llamadas sucesivas.
- Como objeto, la promesa se puede almacenar en una variable, pasar como parámetro o devolver desde una función, lo que permite aplicar los métodos en distintos puntos del código.

© JMA 2016. All rights reserved

Crear promesas

- El objeto Promise gestiona la creación de la promesa y los cambios de estados de la misma.


```
list() {
  return new Promise((resolve, reject) => {
    this.http.get(this.baseUrl).subscribe(
      data => resolve(data),
      err => reject(err)
    )
  });
}
```
- Para crear promesas ya concluidas:
 - Promise.reject: Crea una promesa nueva como rechazada cuyo resultado es igual que el argumento pasado.
 - Promise.resolve: Crea una promesa nueva como resuelta cuyo resultado es igual que su argumento.

© JMA 2016. All rights reserved

Invocar promesas

- El objeto Promise creado expone los métodos:
 - `then(fnResuelta, fnRechazada)`: Recibe como parámetro la función a ejecutar cuando termine la anterior y, opcionalmente, la función a ejecutar en caso de que falle la anterior.
 - `catch(fnError)`: Recibe como parámetro la función a ejecutar en caso de que falle.
`list().then(calcular, ponError).then(guardar)`
- Otras formas de crear e invocar promesas son:
 - `Promise.all`: Combina dos o más promesas y realiza la devolución solo cuando todas las promesas especificadas se completan o alguna se rechaza.
 - `Promise.race`: Crea una nueva promesa que resolverá o rechazará con el mismo valor de resultado que la primera promesa que se va resolver o rechazar entre los argumentos pasados.

© JMA 2016. All rights reserved

INTRODUCCIÓN A REACT

© JMA 2016. All rights reserved

Introducción

- ReactJS es una librería Javascript de código abierto, enfocada en la visualización, para la creación interfaces de usuario.
- React fue creada por Jordan Walke, un ingeniero de software en Facebook para solucionar los problemas internos con XHP, un marco de componentes de HTML para PHP. Fue lanzada en 2013 y es mantenida por Facebook, Instagram y una amplia comunidad de desarrolladores.
- React intenta ayudar a los desarrolladores en la construcción de aplicaciones que usan datos que cambian todo el tiempo.
- Nace para resolver los problemas de escalabilidad en aplicaciones con un gran flujo de datos y permite crear componentes de interfaz de usuario reutilizables.
- Ofrece grandes beneficios en rendimiento y modularidad, que promueve un flujo muy claro de datos y eventos, facilitando la planificación y desarrollo de páginas web complejas.
- En la actualidad es una de las bibliotecas más populares de JavaScript, que cuenta con una sólida base y una gran comunidad detrás de él.

© JMA 2016. All rights reserved

Conceptos

- VirtualDOM: Es un objeto de JavaScript paralelo al árbol DOM que optimiza las actualizaciones del DOM real.
- JSX: Es una extensión que permite introducir XHTML embebido dentro del JavaScript. Aunque no es obligatorio su uso, es muy recomendable. Requiere transpilación.
- Sigue el paradigma de Orientación a Componentes, la tendencia de futuro: favorece la reutilización de código aumentando la velocidad de desarrollo y mejora la legibilidad que ayuda a mantener grandes aplicaciones.
- Se puede integrar con otros framework y utilizar tanto en el lado del cliente como en del servidor.
- Como inconvenientes cabe destacar:
 - la curva de aprendizaje,
 - se debe combinar con otros framework para disponer de un entorno de desarrollo completo,
 - se recomienda utilizar ECMAScript 2015 por lo que es necesaria la transpilación.

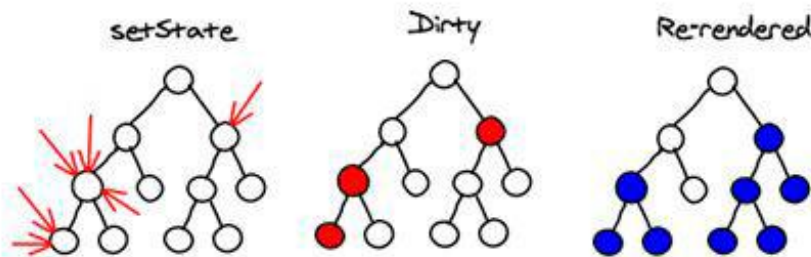
© JMA 2016. All rights reserved

Virtual DOM

- El secreto de ReactJS para tener un rendimiento muy alto, es que implementa algo denominado Virtual DOM: en vez de renderizar (repintar) todo el DOM en cada cambio, que es lo que normalmente se hace, este hace los cambios en una copia en memoria y después usa un algoritmo para comparar las propiedades de la copia en memoria con las de la versión del DOM y así aplicar cambios exclusivamente en las partes que varían.
- El algoritmo diferencial se encarga de actualizar el nuevo DOM comparándolo con el previo.
- En lugar de generar directamente elementos HTML y modificar el DOM, que es una operación muy lenta, los componentes de ReactJS generan (renderizan) un modelo del DOM en memoria. Una vez generado el Virtual DOM completo, ReactJS se encarga de buscar las diferencias entre el DOM real frente al virtual y realizar únicamente las modificaciones necesarias sobre el DOM real.

© JMA 2016. All rights reserved

Virtual DOM



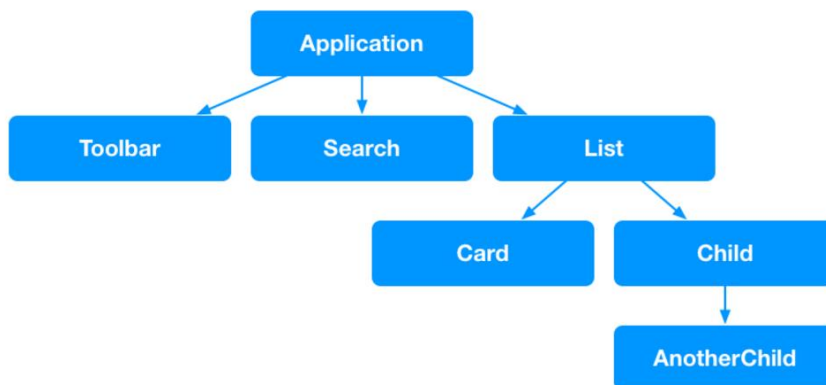
© JMA 2016. All rights reserved

Componentes

- Los sistemas de hoy en día son cada vez más complejos, deben ser construidos en tiempo récord y deben cumplir con los estándares más altos de calidad.
- El paradigma de ensamblar componentes y escribir código para hacer que estos componentes funcionen entre sí se conoce como Desarrollo de Software Basado en Componentes.
- Los Web Components nos ofrecen un estándar que va enfocado a la creación de todo tipo de componentes utilizables en una página web, para realizar interfaces de usuario y elementos que nos permitan presentar información (o sea, son tecnologías que se desarrollan en el lado del cliente). Los propios desarrolladores serán los que puedan, en base a las herramientas que incluye Web Components crear esos nuevos elementos y publicarlos para que otras personas también los puedan usar.
- Un componente puede estar compuesto por componentes que a su vez se compongan de otros componentes y así sucesivamente. Sigue un modelo de composición jerárquico con forma de árbol de componentes. La división sucesiva en componentes permite disminuir la complejidad funcional favoreciendo la reutilización y las pruebas.
- Los componentes establecen un cauce bien definido de entrada/salida para su comunicación con otros componentes.

© JMA 2016. All rights reserved

Árbol de componentes



© JMA 2016. All rights reserved

Tipos de componentes

- En React JS existen dos categorías recomendadas para los componentes:
 - Los **componentes de presentación** son aquellos que simplemente se limitan a mostrar datos y tienen poca o nula lógica asociada a manipulación del estado, es preferible que la mayoría de los componentes de una aplicación sean de este tipo porque son más fáciles de entender y analizar.
 - Los **componentes contenedores** tienen como propósito encapsular a otros componentes y proporcionarles las propiedades que necesitan, además se encargan de modificar el estado de la aplicación por ejemplo usando Flux o Redux para despachar alguna acción y que el usuario vea el cambio en los datos.

© JMA 2016. All rights reserved

Características

- Características de los componentes de presentación
 - Orientados al aspecto visual
 - No tienen dependencia con fuentes de datos (e.g. Flux)
 - Reciben callbacks por medio de props
 - Pueden ser escritos como componentes funcionales.
 - Normalmente no tienen estado
- Características de los componentes contenedores
 - Orientados al funcionamiento de la aplicación
 - Contienen componentes de presentación y/o otros contenedores
 - Se comunican con las fuentes de datos
 - Usualmente tienen estado para representar el cambio en los datos

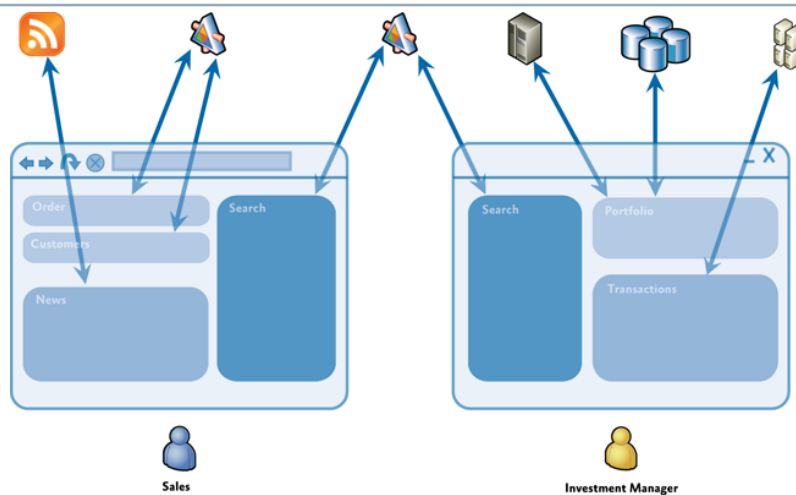
© JMA 2016. All rights reserved

Inmutable

- Una característica importante de ReactJS es que promueve el **flujo de datos en un solo sentido**, en lugar del flujo bidireccional típico en Frameworks modernos, esto hace más fácil la planificación y la detección de errores en aplicaciones complejas, en las que el flujo de información puede llegar a ser muy complejo, dando lugar a errores difíciles de ubicar que pueden hacernos la vida muy triste.
- **Flux** es la arquitectura de aplicaciones que Facebook usa para construir aplicaciones web del lado del cliente. Complementa los componentes de vista de React utilizando un flujo de datos unidireccional. Es más un patrón que un marco formal.

© JMA 2016. All rights reserved

Patrón Composite View



© JMA 2016. All rights reserved

Aplicaciones Isomorficas

- También conocidas como Universal JavaScript, son aquellas que nos permiten ejecutar el código JavaScript, que normalmente ejecutamos en el cliente, en el servidor.
- Una aplicación SPA (Single Page Application) aunque inicialmente tarda mas en cargarse, luego tiene una velocidad de página y experiencia de usuario muy buena. Como desventaja presentan la perdida de SEO dado que los motores de búsqueda no interaccionan con las páginas cuando las indexan y no generan los contenidos dinámicos por lo que gran parte del contenido puede quedar sin indexar, cosa que no pasa cuando se genera la pagina en el lado del servidor. Otro problema son los navegadores que no soportan o tienen de desactivado el JavaScript.
- Las aplicaciones isomorfas permiten:
 - Ejecutar el mismo código en el lado cliente y en el lado servido.
 - Decidir si modifica directamente el DOM o manda una cadena con el HTML modificado.
 - Obtener las ventajas de ambos modos.
- La forma de implementación de React permite usar la librería en la parte de servidor con Node JS, lo que nos permite junto con otras librerías como React-Router y React-Engine, crear componentes que puedan ser reutilizados tanto en el Frontend como en el Backend. React suministra las herramientas adecuadas para crear aplicaciones isomorfas pero hay que crearlas de forma especifica.

© JMA 2016. All rights reserved

Objetos fundamentales

- **React:** es el punto de entrada a la biblioteca React. Suministra el resto de objetos de la biblioteca.

```
const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Hola Mundo!!!'
);
```
- **ReactDOM:** proporciona métodos específicos de DOM que se utilizan en el nivel superior de la aplicación para exportar los elementos y componentes de React hacia el árbol DOM.

```
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

© JMA 2016. All rights reserved

JSX

© JMA 2016. All rights reserved

JSX

- Es una extensión de sintaxis de JavaScript que permite incrustar XHTML dentro de JavaScript sin tener que delimitarlo como cadenas. Invierte el concepto de scripting: HTML con JavaScript incrustado → JavaScript con HTML incrustado.

```
const element = (  
  <h1 className='greeting'>  
    Hola Mundo!!!  
  </h1>  
);
```
- Una vez compilado, el resultado queda en JavaScript puro.

```
const element = React.createElement(  
  'h1',  
  {className: 'greeting'},  
  'Hola Mundo!!!'  
);
```
- JSX puede recordar un lenguaje de plantilla, pero cuenta con todo el poder de JavaScript.

© JMA 2016. All rights reserved

JSX

- React recomienda utilizar JSX para crear plantillas en lugar de JavaScript regular. No es necesario utilizarlo, pero hay importantes ventajas en su utilización.
- Ventajas:
 - Es **más eficiente**, ya que se realiza una optimización al compilar código de JavaScript.
 - Es **mas seguro**, dado que la mayoría de los errores pueden ser capturados durante la compilación.
 - Es **más fácil y más rápido** escribir plantillas si se está familiarizado con el lenguaje HTML.
- Desventajas:
 - Requiere compilación.

© JMA 2016. All rights reserved

Sintaxis

- Utiliza la notación PascalCase en:
 - Componentes: MiComponente
- Utiliza la notación camelCase en:
 - Atributos: tabIndex
 - Eventos: onClick
- Tiene una notación alternativa para los atributos que son palabras reservadas:
 - Atributo class de HTML → className
 - Atributo for del Label → htmlFor
- Comentarios:
 - `{/* ... */}`
- Debe estar correctamente anidado y todas las etiquetas deben cerrarse, las etiquetas vacías se cierran con `/>` (similar a XML):
 - `<p>...
...</p>`
 - `<input ... />`

© JMA 2016. All rights reserved

Elementos

- Cada elemento representa una etiqueta HTML que puede contener otras etiquetas.

```
const element = (
  <p>
    Hola <b>Mundo!!!</b>
  </p>
);
```

- Se puede dividir JSX en varias líneas para facilitar la lectura. Si bien no es necesario, al hacer esto, también se recomienda envolverlo entre paréntesis para evitar los inconvenientes de la inserción automática de punto y coma.
- Por defecto solo puede contener una única etiqueta, para contener mas de una es necesario envolverlas en un contenedor, un array o un fragmento (sin nodo DOM):

```
const element = (
  <div>
    <h1>Hola</h1>
    <h2>Mundo</h2>
  </div>
);
```

```
const element = [
  <h1>Hola</h1>
  <h2>Mundo</h2>
];
```

```
const element = (
  <React.Fragment>
    <th>Hola</th>
    <td>Mundo</td>
  </React.Fragment>
);
```

© JMA 2016. All rights reserved

Expresiones

- Se puede incrustar cualquier expresión de JavaScript en JSX envolviéndola con llaves.

```
const calculo= <p>Calcula: 2 + 2 = {2 + 2}</p>;
function formatName(user) {
  return user.firstName + ' ' + user.lastName;
}
const saludo= <h1>Hola {formatName(user)}</h1>;
```

- Para asignar valores a los atributos se utilizan las comillas para los valores constantes y las llaves para las expresiones (y constantes aritméticas):


```
const constante = ;
const expresion = <img src={user.avatarUrl} />;
```
- De forma predeterminada, React DOM escapa cualquier valor incrustado en JSX antes de representarlos garantizando que nunca se inyectará nada que no esté escrito explícitamente en la aplicación. Todo se convierte en una cadena antes de ser renderizado. Esto ayuda a prevenir ataques XSS (cross-site-scripting).

© JMA 2016. All rights reserved

Expresiones

- Operador condicional simple:
`const saludo= <p>Hola {user && formatName(user)</p>;`
- Operador condicional ternario:
`const btnLogIn = <input type="button" value="Log In" />;`
`const btnLogOut = <input type="button" value="Log Out" />;`
`const saludo= <p>{user ? btnLogIn : btnLogOut}</p>;`
- Si se encapsula en una función o expresión lambda se pueden utilizar instrucciones condicionales y bucles:

```
function saludar(user) {
  if (user) {
    return <h1>Hola {formatName(user)}!</h1>;
  }
  return <h1>Hola desconocido.</h1>;
}
```
- No se renderizan los valores true, false, null y undefined.
`<div>{true}</div>` se visualiza como `<div></div>`

© JMA 2016. All rights reserved

Expresiones

- Se pueden crear colecciones de elementos e incluirlos en JSX con llaves {}.

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <li key={number.toString()}>{number}</li>
);
const list = <ul>{listItems}</ul>;
const otra = <ul>{numbers.map((number) => <li key={number.toString()}>{number}</li>)}</ul>;
```
- **key** es un atributo cadena especial que debe incluir al crear listas de elementos.
- Las claves ayudan a React (VirtualDOM) a identificar qué elementos han cambiado, se han agregado o se han eliminado. Las claves se deben asignar dentro de la matriz para dar a los elementos una identidad estable. La mejor forma de elegir una clave es usar una cadena que identifique de forma única cada elemento. Muy a menudo se utiliza como claves la clave primaria del modelo de datos.
- Cuando no hay IDs estables para los elementos representados y como último recurso, se puede usar el índice en el array como clave (no se debe ordenar ni eliminar elementos):

```
const listItems = numbers.map((number, indice) =>
  <li key={indice.toString()}>{number}</li>
);
```

© JMA 2016. All rights reserved

Renderizado

- Un elemento describe lo que quiere ver en la página. A diferencia de los elementos DOM del navegador, los elementos React son simples objetos que son baratos de crear. React DOM se encarga de actualizar el DOM para que coincida con los elementos React.
- En la página HTML se debe establecer un punto de entrada, el nodo raíz:

```
<div id="root"></div>
```
- Todo dentro de un nodo DOM raíz será administrado por React DOM.

```
ReactDOM.render(  
  element,  
  document.getElementById('root')  
);
```
- Generalmente se tiene un solo nodo DOM raíz pero se pueden tener tantos nodos raíz aislados como se desee.

© JMA 2016. All rights reserved

Actualización del renderizado

- Los elementos de React son inmutables. Una vez se crea un elemento, no puede cambiar sus elementos secundarios o atributos. Un elemento es como un único fotograma en una película: representa la UI en un momento determinado.
- La única forma de actualizar la UI es crear un nuevo elemento y volver a ejecutar ReactDOM.render().
- React DOM compara el elemento y sus elementos secundarios con el anterior (VirtualDOM), y solo aplica las actualizaciones DOM necesarias para llevar el DOM al nuevo estado.
- La filosofía de React es pensar en cómo debe verse la UI en un momento dado en lugar de cómo cambiarla a lo largo del tiempo, esto elimina muchos tipos de errores.

© JMA 2016. All rights reserved

COMPONENTES

© JMA 2016. All rights reserved

Introducción

- Los componentes permiten dividir la interfaz de usuario en piezas independientes y reutilizables, y pensar en cada pieza aisladamente.
- Conceptualmente, los componentes son como las funciones de JavaScript. Aceptan entradas arbitrarias (llamadas "props" abreviatura de propiedades) y devuelven elementos React que describen lo que debería aparecer en la pantalla.
- La forma más sencilla de definir un componente es escribir una función de JavaScript:

```
function Saludo(props) {  
  return <h1>Hola {props.name}</h1>;  
}
```

Esta función es un componente React válido porque acepta un único argumento de objeto "props" con datos y devuelve un elemento React.
- También puede usar una clase ES6 para definir un componente:

```
class Saludo extends React.Component {  
  render() {  
    return <h1>Hola {this.props.name}</h1>;  
  }  
}
```
- Los dos componentes anteriores son equivalentes desde el punto de vista de React. Las clases tienen múltiples características adicionales.

© JMA 2016. All rights reserved

Composición

- Un componente puede estar compuesto por componentes que a su vez se compongan de otros componentes y así sucesivamente.
- Sigue un modelo de composición jerárquico con forma de árbol de componentes.
- La división sucesiva en componentes permite disminuir la complejidad funcional favoreciendo la reutilización y las pruebas.
- Un botón, un formulario, un listado, una página: en las aplicaciones React, todos ellos se expresan comúnmente como componentes.

```
function App(props) {
  return (
    <div>
      <Saludo name="Don Pepito" />
      <Saludo name="Don Jose" />
    </div>
  );
}
```

© JMA 2016. All rights reserved

Renderizar el componente

- **Los nombres de los componentes deben comenzar siempre con una letra mayúscula (notación Pascal).**
- Los componentes se utilizan como si fueran nuevos tipos de etiquetas del HTML dentro de los elementos.
- Las propiedades toman la sintaxis de los atributos HTML, cuando React ve un elemento que representa un componente definido por el usuario, pasa los atributos JSX al componente como propiedades de un solo objeto.

```
const element = <Saludo name="mundo" />;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

© JMA 2016. All rights reserved

Propiedades

- Las propiedades permiten personalizar los componentes.
- Las propiedades toman la sintaxis de los atributos HTML.
- Las propiedades se encapsulan en un único objeto (array asociativo o diccionario), donde las claves son los nombres de los atributos y los valores son los valores asignados a los atributos, y se almacenan en la propiedad heredada `this.props`.
- Las propiedades son inmutables, de solo lectura. Aunque React es bastante flexible tiene algunas reglas estrictas: Los componentes no deben modificar las propiedades.
- Los componentes deben comportarse como “funciones puras” porque no deben cambiar sus entradas y siempre devuelven el mismo resultado para las mismas entradas.

© JMA 2016. All rights reserved

Propiedades

- Los valores de las propiedades pueden ser valores constantes o expresiones:
`const element = <Saludo name={formatName(user)} />;`
- Si no se asigna valor al atributo se establece de manera predeterminada con el valor `true`.
`<MyTextBox autocomplete />`
`<MyTextBox autocomplete={true} />`
- Si ya se dispone de las propiedades en un objeto se puede pasar utilizando el operador `...` de "propagación" (ES6):
`const attr = {init: 10, delta: '1'};`
`return <Contador {...attr} />; // <Contador init={10} delta="1" />`
- El operador de "propagación" también permite la des-estructuración:
`const { tipo, ...attr } = props;`
`if (tipo)`
`return <Contador {...attr} />;`
`return <Slider {...attr} />;`

© JMA 2016. All rights reserved

Contenido del componente

- Los componentes tienen una etiqueta de apertura y otra de cierre, el contenido entre estas etiquetas se pasa como una propiedad especial: `props.children`.
`<MiContenedor>Hola Mundo ...</MiContenedor>`
- Puede contener otros componentes, literales, etiquetas, expresiones, funciones o cualquier otra cosa válida dentro de cualquier etiqueta.
- El contenido puede ser múltiple y mezclado, en cuyo caso `props.children` toma la forma de un array.

```

<MiContenedor>
  Texto
  <Saluda nombre="mundo" />
  <Contador init={10} delta={1} />
  <h1>Etiqueta</h1>
  {this.props.nombre}
  {(item) => <div>Formato para {item}</div>}
</MiContenedor>

```
- Antes de pasar el contenido a `props.children` automáticamente se elimina: el espacio en blanco al principio y al final de una línea, las líneas en blanco, las nuevas líneas adyacentes a las etiquetas y los saltos de línea en medio de los literales cadena que se condensan en un solo espacio.

© JMA 2016. All rights reserved

PropTypes

- A medida que la aplicación crece, se pueden detectar muchos errores con la verificación de tipos de las propiedades de los componentes.
- `PropTypes` exporta una gama de validadores que se pueden usar para garantizar que los datos que reciba sean válidos. Cuando se proporciona un valor no válido, se mostrará una advertencia en la consola del navegador pero, por razones de rendimiento, solo se verifica en modo de desarrollo.
- Las validaciones se establecen como una estructura que se asigna al campo `static propTypes` del componente (solo clases).

```

import PropTypes from 'prop-types';
class Contador extends React.Component {
  // ...
}
Contador.propTypes = {
  init: PropTypes.number.isRequired,
  delta: PropTypes.any,
  onCambia: PropTypes.func
};
const element = <Contador init={10} delta="1" />;

```

© JMA 2016. All rights reserved

ProdTypes

- Tipos primitivos: array, bool, func, number, object, string, symbol, any, element (de React), node (cualquier valor que pueda ser renderizado).
- Tipos definidos:
 - `instanceOf(MyTipo)`: instancia de una determinada clase.
 - `arrayOf(PropTypes.???)`: Array con todos los valores de un determinado tipo.
 - `objectOf(PropTypes.???)`: Un objeto con valores de propiedad de cierto tipo.
 - `oneOf([...])`: Valor limitado a valores especificados en el array tratándolo como una enumeración.
 - `oneOfType([...])`: Tipo unión, uno de los tipos especificados en el array.
 - `shape({...})`: Objeto que al menos tiene las propiedades y tipos indicados en el array asociativo.
- También se puede especificar un validador personalizado que debería devolver un objeto Error si la validación falla. “`console.warn`” o `throw` no funcionarán dentro de “`oneOfType`”:


```
customProp: PropTypes.arrayOf(function(propValue, key, componentName, location,
propFullName) {
  if (!matchme/.test(propValue[key])) {
    return new Error('...');
  }
})
```

© JMA 2016. All rights reserved

ProdTypes

- Se puede definir una propiedad como obligatoria marcándola con `.isRequired` al fijar el tipo.


```
Contador.propTypes = {
  init: PropTypes.number.isRequired,
  delta: PropTypes.any,
  onCambia: PropTypes.func
};
```
- Los valores predeterminados de las propiedades se establecen como una estructura que se asigna al campo `static defaultProps` del componente.


```
Contador.defaultProps = {
  delta: 5
};
```

© JMA 2016. All rights reserved

Estado

- El estado es similar a las propiedades pero puede cambiar, es privado y está completamente controlado por el componente.
- El estado local es una característica disponible solo para las clases.
- El estado es la propiedad `this.state`, que se hereda de la clase componente, y es un objeto que encapsula todas las propiedades de estado que se vayan a utilizar en el `render()`.
- El estado se inicializa en el constructor, que obligatoriamente tiene que invocar al constructor del padre para propagar las propiedades:

```
class Contador extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      contador: +props.init,
      delta: +props.delta
    };
  }
}
```

© JMA 2016. All rights reserved

Estado

- El estado es accesible a través de las propiedades de `this.state`:

```
render() {
  return (
    <div>
      <h1>{this.state.contador}</h1>
      <p>
        <button onClick={this.suba}>Suba</button> &nbsp;
        <button onClick={this.baja.bind(this, 1, 2)}>Baja</button>
      </p>
    </div>
  );
}
```

- Los cambios en el estado provocan que se vuelva a ejecutar el método `render`.

© JMA 2016. All rights reserved

Estado

- El estado no debe ser modificado directamente:
`this.state.contador += 1;`
- Para modificar el estado se debe utilizar el método `this.setState()`:
`this.setState({ contador: this.state.contador + 1 });`
- `setState` recibe un objeto con todas las modificaciones del estado y realiza una mezcla, solo modifica las propiedades del estado que aparezcan en el objeto, creándolas si es necesario.
- React puede agrupar varias llamadas `setState()` en una sola actualización para mejorar el rendimiento, por lo que `this.props` y `this.state` pueden ser actualizados de forma asíncrona y no se debe utilizar directamente sus valores para calcular el siguiente estado, se debe utilizar una función que reciba los valores previos:
`this.setState((prev, props) => {
 return { contador: prev.contador + prev.delta }
});`

© JMA 2016. All rights reserved

Campos de la clase

- Las propiedades (`this.props`) y el estado tienen un significado especial y son configurados por React.
- Se pueden agregar manualmente campos (atributos) adicionales a la clase, para implementar la lógica interna, para almacenar todo aquello que no se usa en la salida visual, por lo que no requieren desencadenar el repintado al ser modificado.

```
constructor(props) {  
  super(props);  
  this.pulsaciones = 0;  
}
```

© JMA 2016. All rights reserved

Eventos

- En React los eventos de los elementos se manejan de una forma muy similar a como se manejan los eventos de los elementos DOM, pero con algunas diferencias importantes:
 - Los eventos React se nombran usando camelCase, en lugar de minúsculas.
 - Con JSX se pasa una función como controlador de eventos: el objeto, no la invocación.
 - Se debe llamar a `e.preventDefault()` explícitamente, devolver `false` no evita el comportamiento predeterminado (propagación de eventos) en React.
- Sintaxis:
 - HTML: `<button onclick="sube()">Sube</button>`
 - JSX: `<button onClick={this.sube}>Sube</button>`

© JMA 2016. All rights reserved

Eventos

- El controlador de evento se puede implementar como:
 - Método público de la clase:


```
baja(e) {
  e.preventDefault();
  this.setState((prev, props) => {
    return { contador: prev.contador - prev.delta }
  })
}
```
 - Campo público de la clase (experimental):


```
sube = (e) => {
  e.preventDefault();
  this.setState((prev, props) => {
    return { contador: prev.contador + prev.delta }
  })
};
```
 - Función flecha:


```
<button onClick={(e) => this.setState({ contador: 0 })}>Inicia</button>
```

© JMA 2016. All rights reserved

Eventos

- Hay que tener cuidado con el significado del `this` en los callbacks de JSX. En JavaScript, los métodos de clase no están vinculados por defecto. Si no se vincula explícitamente, `this` estará como `undefined` cuando se invoque a la función.


```
constructor(props) {
  super(props);
  // ...
  this.baja = this.baja.bind(this);
}
```
- Esto no es un comportamiento específico de React, es cómo funcionan los métodos en JavaScript, si se refiere a un método sin `()` se debe vincular dicho método.
- No es necesario vincular los campos públicos ni las funciones flecha.
- Las funciones flecha tienen el problema de que se crea una devolución de llamada diferente cada vez que se procesa. En la mayoría de los casos, esto no es un problema. Sin embargo, si este callback se pasa como una propiedad a los componentes anidados, esos componentes podrían hacer un render adicional. En general, es recomendable enlazar en el constructor o usar la sintaxis de los campos de clase, para evitar este tipo de problema de rendimiento.

© JMA 2016. All rights reserved

Eventos

- Por defecto el controlador de eventos recibe como argumento un objeto evento sintético (`SyntheticEvent`), un contenedor alrededor del evento nativo del navegador. React define estos eventos sintéticos de acuerdo con la especificación W3C, por lo que no hay que preocuparse por la compatibilidad entre navegadores.
- En React, generalmente no se necesita llamar `addEventListener` para agregar controladores al elemento DOM después de haber sido creado. Basta con proporcionar el controlador cuando el elemento se represente inicialmente.
- Para pasar argumentos a los controladores de eventos:


```
<button onClick={this.deleteRow.bind(this, id)}>Delete Row</button>
<button onClick={(e) => this.deleteRow(id, e)}>Delete Row</button>
```
- Son equivalentes pero el argumento `e`, que representa el evento React:
 - En el `bind`, se pasará automáticamente como último argumento.
 - En la función flecha, que ya ha recibido el valor del argumento `e`, hay que propagarlo explícitamente.
- El objeto `SyntheticEvent` se reutilizará y todas las propiedades se anularán después de que se haya invocado el controlador de eventos por lo que no se puede acceder al evento de manera asíncrona, hay que cachear su información.

© JMA 2016. All rights reserved

SyntheticEvent

- boolean bubbles
- boolean cancelable
- DOMEventTarget currentTarget
- boolean defaultPrevented
- number eventPhase
- boolean isTrusted
- **DOMEvent nativeEvent**
- void **preventDefault()**
- boolean isDefaultPrevented()
- void **stopPropagation()**
- boolean isPropagationStopped()
- DOMEventTarget target
- number timeStamp
- string type

© JMA 2016. All rights reserved

Eventos compatibles

Clipboard	onCopy onCut onPaste
Composition	onCompositionEnd onCompositionStart onCompositionUpdate
Keyboard	onKeyDown onKeyPress onKeyUp
Focus	onFocus onBlur
Mouse	onClick onContextMenu onDoubleClick onDrag onDragEnd onDragEnter onDragExit onDragLeave onDragOver onDragStart onDrop onMouseDown onMouseEnter onMouseLeave onMouseMove onMouseOut onMouseOver onMouseUp onWheel
Selection	onSelect
Touch	onTouchCancel onTouchEnd onTouchMove onTouchStart
UI	onScroll
Media	onAbort onCanPlay onCanPlayThrough onDurationChange onEmptied onEncrypted onEnded onError onLoadedData onLoadedMetadata onLoadStart onPause onPlay onPlaying onProgress onRateChange onSeeked onSeeking onStalled onSuspend onTimeUpdate onVolumeChange onWaiting
Image	onLoad onError
Animation	onAnimationStart onAnimationEnd onAnimationIteration
Transition	onTransitionEnd
Other	onToggle

© JMA 2016. All rights reserved

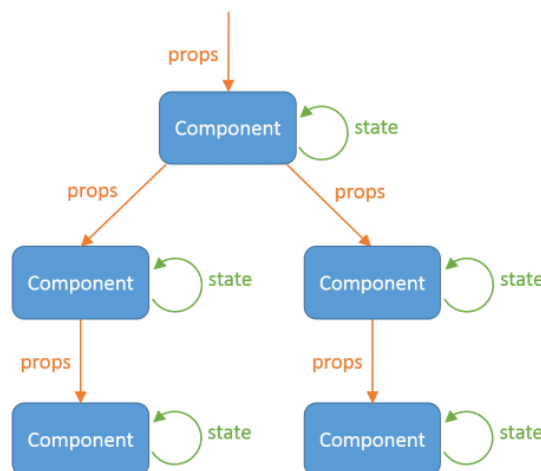
Flujo de datos

- La información fluye hacia abajo en el árbol de componentes.
- Los componentes, contenedores o contenidos, no pueden saber si un componente determinado tiene o no estado, sin importar si se define como una función o una clase.
- Esta es la razón por la cual a menudo se denomina al estado como local o encapsulado: no es accesible por ningún componente que no sea el que lo posee y establece.
- Un componente puede elegir pasar su estado como propiedad a sus componentes contenidos:


```
<Contador init={this.state.valor} delta="2" />
```
- Esto se conoce comúnmente como flujo de datos "descendente" o "unidireccional". Cualquier estado siempre es propiedad de algún componente específico, y cualquier dato o IU derivado de ese estado solo puede afectar a los componentes por "debajo" de ellos en el árbol de componentes.

© JMA 2016. All rights reserved

Flujo de datos



© JMA 2016. All rights reserved

Flujo de datos inverso

- En React se denomina *elevantado* al mecanismo mediante el cual un componente comunica sus cambios de estado a su contenedor.
- Similar a los eventos, se implementa mediante el paso de la función del contenedor en una propiedad del componente que el componente debe invocar cuando se produzca un cambio en su estado:


```
<Contador init="10" delta="2" onCambia={
  rslt => this.setState({cont: rslt})
} />
```
- Dado que son opcionales, el componente debe comprobar que le han pasado la función antes de invocarla:


```
if(this.props.onCambia)
  this.props.onCambia(this.state.contador);
```
- Hay que tener cuidado para no provocar un bucle infinito, que el cambio no reentre en el componente provocando un nuevo cambio.

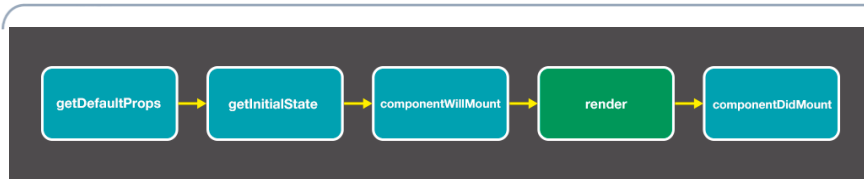
© JMA 2016. All rights reserved

Ciclo de vida del componente

- Para comprender el comportamiento de los componentes de React es importante comprender su ciclo de vida y los métodos que intervienen en dicho ciclo. Lo interesante de esto es que los métodos se pueden sobrescribir para conseguir que el componente se comporte de la manera que se espera.
- Podemos plantear cuatro fases que pueden darse en el ciclo de vida de un componente.
 - Montaje o inicialización
 - Actualización de estado
 - Actualización de propiedades
 - Desmontaje
- Cada método tiene un prefijo `will` o `did` dependiendo de si ocurren antes o después de cierta acción.

© JMA 2016. All rights reserved

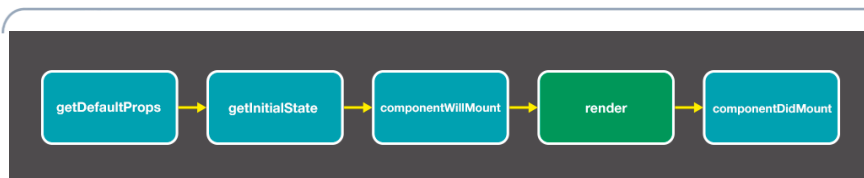
Montaje



- **getDefaultProps y getInitialState** o el **constructor(props)** de la clase: se ejecuta cuando se instancia un componente. Nos permite definir el estado inicial del componente, hacer el bind de métodos y definir los campos de la clase.
- **componentWillMount()**: se ejecuta cuando el componente se va a renderizar. En este punto es posible modificar el estado del componente sin causar una actualización (y por lo tanto no renderizar dos veces el componente).

© JMA 2016. All rights reserved

Montaje



- **render()**: único obligatorio, en esta fase genera la UI inicial del componente. Debe ser una función pura (no puede tener efectos secundarios) y no debe modificar nunca el estado del componente.
- **componentDidMount()**: se ejecuta cuando el componente ya se ha renderizado en el navegador y permite interactuar con el DOM o las otras APIs del navegador (geolocation, navigator, notificaciones, etc.). Es el mejor sitio para realizar peticiones HTTP o suscribirse a diferentes fuentes de datos (un Store o un WebSocket) y actualizar el estado al recibir una respuesta. Cambiar el estado en este método provoca que se vuelva a renderizar el componente.

© JMA 2016. All rights reserved

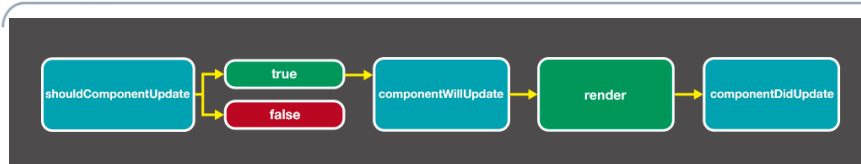
Actualización de estado



- **shouldComponentUpdate(next_props, next_state)**: decide si el componente se vuelve a renderizar o no. Recibe como parámetros las nuevas propiedades y el nuevo estado, debe devolver true para que se renderice. False puede afectar a sus componentes hijos que no reciben las nuevas propiedades.
- **componentWillUpdate(next_props, next_state)**: recibe como parámetros las nuevas propiedades y el nuevo estado para preparar al componente para su actualización por lo que no se debe modificar estados para evitar bucles infinitos.

© JMA 2016. All rights reserved

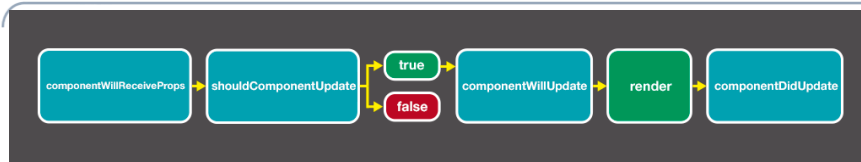
Actualización de estado



- **render()**: en esta fase, genera una nueva versión virtual del componente y modifica la versión actual en el DOM con las discrepancias con la nueva versión. Debe ser una función pura (no puede tener efectos secundarios) y no debe modificar nunca el estado del componente.
- **componentDidUpdate(next_props, next_state)**: al igual que componentDidMount complementa al render con las operaciones en las que intervengan elementos del DOM y necesitan que ya estén creados.

© JMA 2016. All rights reserved

Actualización de propiedades



- **componentWillReceiveProps(next_props)**: Se ejecuta tan sólo cuando las propiedades del componente cambian (flujo del contenedor), recibiendo como parámetros las nuevas propiedades y puede actualizar el estado sin desencadenar que se vuelva a renderizar el componente.
- Los pasos siguientes en este escenario, a partir de este punto, son iguales que cuando se actualiza el estado del componente.

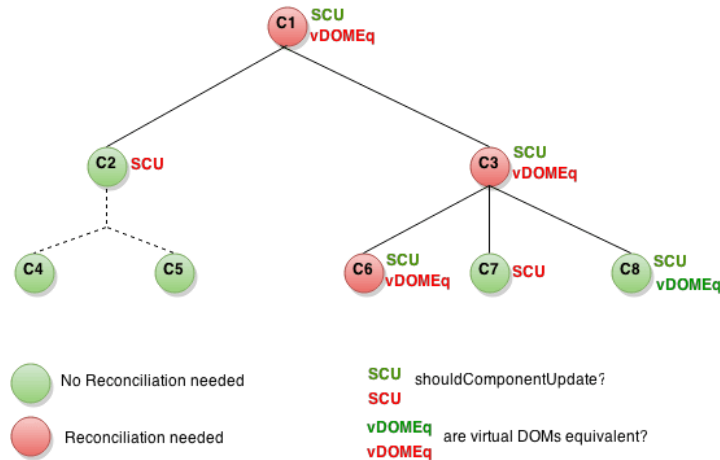
© JMA 2016. All rights reserved

Desmontaje

- En el desmontaje de un componente React interviene un único método, **componentWillUnmount()**, y es invocado justo antes de que el componente se desmonte: se quite del árbol DOM.
- Es el momento para realizar operaciones de limpieza como des suscribirse de los WebSocket o Stores, cancelar peticiones HTTP que hayan quedado pendientes, eliminar listeners, temporizadores o demás objetos que puedan quedar en memoria.
- Es muy importante para conservar los recursos y asegurar el rendimiento.

© JMA 2016. All rights reserved

Reconciliación del DOM



© JMA 2016. All rights reserved

Campos Referencia

- En el flujo de datos típico de React, las propiedades son la única forma en que los componentes principales interactúan con sus hijos. Para modificar un hijo, se vuelve a representar con nuevas propiedades.
- Sin embargo, hay algunos casos en los que se necesita modificar imperativamente a un hijo o ejecutar algunos de sus métodos, que podría ser un elemento DOM o un componente React, fuera del flujo de datos típico.
- Para estos casos React proporciona una vía de escape: refs. Mediante una función permite asociar un elemento DOM o componente React a un campo de la clase:


```
<button ref={(tag) => {this.btnSalir = tag; }}>Salir</button>
```
- Posteriormente, una vez ejecutado el render() apropiado, se puede acceder a el en otro punto del código:


```
this.btnSalir.focus();
```
- Hay que evitar usar refs para cualquier cosa que se pueda hacer directamente de manera declarativa, aunque algunos buenos escenarios para usarlas son:
 - Administrar el foco y la selección de texto o la reproducción de medios.
 - Controlar animaciones.
 - Integración con bibliotecas DOM de terceros.
- No se puede usar ref con las funciones componente porque no tienen instancias.

© JMA 2016. All rights reserved

Límites de Errores

- Los límites de errores son componentes de React que capturan errores de JavaScript (funcionan como un bloque `catch{}` de JavaScript, pero para componentes) en cualquier parte de su árbol de componentes hijo, registran esos errores y muestran una interfaz de repuesto en lugar del árbol de componentes que ha fallado.
- Los límites de errores capturan errores:
 - Durante el renderizado.
 - En métodos del ciclo de vida.
 - En constructores de todo el árbol bajo ellos.
- Los límites de errores no capturan errores de:
 - Manejadores de eventos
 - Código asíncrono
 - Renderizado en el servidor
 - Errores lanzados en el propio límite de errores (no puede capturar un error dentro de sí mismo)
- Sólo los componentes de clase pueden ser límites de errores.

© JMA 2016. All rights reserved

Límites de Errores

- Un componente de clase (class component) se convierte en límite de errores si define uno o ambos de los siguientes métodos del ciclo de vida:
 - `static getDerivedStateFromError(error)` para activar una interfaz de repuesto cuando se lance un error (se llama durante la fase “render”, por lo que los efectos secundarios no están permitidos).
 - `componentDidCatch(error, info)` para registrar información de los errores y se invoca después de que un error haya sido lanzado por un componente descendiente, por lo tanto, los efectos secundarios se permiten. Recibe dos parámetros:
 - `error`: Es el error (excepción) que ha sido lanzado.
 - `info`: Un objeto con una clave `componentStack` que contiene información sobre que componente ha devuelto un error.
- A partir de React 16, los errores que no sean capturados por ningún límite de error resultarán en el desmontado de todo el árbol de componentes de React.

© JMA 2016. All rights reserved

Límites de Errores

```
export class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false, error: null, errorInfo: null };
  }
  static getDerivedStateFromError(error) { // Actualiza el estado para que el siguiente renderizado lo muestre
    return { hasError: true };
  }
  componentDidCatch(error, info) { // También puedes registrar el error en un servicio de reporte de errores
    this.setState({ hasError: true, error: error, errorInfo: info })
  }
  render() {
    if (this.state.hasError) { // Puedes renderizar cualquier interfaz de repuesto
      return <div>
        <h1>ERROR</h1>
        {this.state.error && <p>this.state.error.toString()</p>}
        {this.state.errorInfo && <p>this.state.errorInfo.componentStack</p>}
      </div>;
    }
    return this.props.children;
  }
}
```

© JMA 2016. All rights reserved

FORMULARIOS

© JMA 2016. All rights reserved

Introducción

- Los elementos de formulario HTML funcionan de forma un poco diferente del resto de elementos DOM en React, porque los elementos de formulario ya mantienen un estado interno.
- El comportamiento predeterminado de los formularios de HTML es navegar a una nueva página cuando el usuario envía el formulario. Si este es el comportamiento deseado no es necesario hacer nada en React.
- En la mayoría de los casos, es conveniente tener una función de JavaScript que envíe los datos del formulario sin enviar el formulario, lo que desencadenaría navegar a una nueva página.
- La función debe tener acceso a los datos que el usuario introdujo en el formulario para enviarla al servidor vía Ajax o WebSocket.
- React permite utilizar dos técnicas diferentes:
 - componentes controlados
 - componentes no controlados.

© JMA 2016. All rights reserved

Componentes controlados

- En HTML, los elementos del formulario `<input>`, `<textarea>` y `<select>` mantienen su propio estado y lo actualizan en función de las entradas del usuario. En React, el estado mutable se mantiene en la propiedad `state` de los componentes y solo se actualiza con `setState()`.
- Para mantener sincronizados ambos estados, el componente React que representa un formulario también debe controlar lo que sucede en ese formulario después de la entrada del usuario mediante el uso de eventos.
- Un elemento de entrada de formulario cuyo valor es controlado por React de esta manera se denomina "componente controlado".
- La propiedad `value` de `<input>` permite establecer el valor inicial y recuperar el valor actual. React ha incorporado la pseudo-propiedad `value` a `<textarea>` y `<select>`, de tal forma que los tres tengan el mismo comportamiento.
- El evento `onChange` de React se dispara con cada pulsación de tecla o selección en el control. El controlador del evento recibe un argumento `SyntheticEvent` que, a través de su propiedad `target`, permite acceder al `HTMLInputElement` que disparó el evento.

© JMA 2016. All rights reserved

Inicialización

- Inicialización del estado:


```
constructor(props) {
  super(props);
  this.state = {
    nombre: props.nombre,
    // ...
  };
  this.handleChange = this.handleChange.bind(this);
}
```
- Para que sea un "componente controlado" se debe asignar al atributo value el state apropiado e interceptar el onChange.


```
render() {
  // ...
  <input type="text" name="nombre" value={this.state.nombre} onChange={this.handleChange} />
  // ...
}
```
- Es conveniente que el valor del atributo name coincida con el nombre de la propiedad en el state para facilitar el tratamiento posterior.

© JMA 2016. All rights reserved

Tratamiento de cambios

- El tratamiento del onChange debe incluir el paso del value al state.


```
handleChange(event) {
  this.setState(nombre: event.target.value);
}
```
- O de una forma genérica que atiende a múltiples controles:


```
handleChange(event) {
  this.setState({[event.target.name]: event.target.value});
}
```
- El tratamiento puede incluir transformaciones y validaciones pero si no se pasa el value al state se anulan los cambios.


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

© JMA 2016. All rights reserved

Validaciones

- HTML5 ya cuenta con validadores, pero cuando la validación sea personalizada o el navegador no soporte las validaciones HTML5 se incluirán en la captura de datos y se pasaran al estado para poder mostrar al usuario los errores:


```
let errors = {};
if (event.target.name === 'nombre') {
  if (event.target.value && event.target.value !== event.target.value.toUpperCase()) {
    errors.nombre = 'Debe ir en mayúsculas.';
  } else
    errors.nombre = null;
}
this.setState({ msgErr: errors, invalid: invalid });
```
- Cuando se mezcla la validación HTML5 y la personalizada hay que capturar los mensajes HTML para poder implementar una experiencia de usuario coherente y no mostrarlos de formas diferentes en función a la detección:


```
errors[event.target.name] = event.target.validationMessage;
if (!errors.nombre && event.target.name === 'nombre') {
  // ...
}
```

© JMA 2016. All rights reserved

Mostrar errores

- Los errores se pueden acumular dentro del estado en un objeto con los pares nombre/valor donde:
 - Nombre: name del control
 - Valor: cadena con el mensaje de error

```
this.state = { ... msgErr: {} ... };
```
- Es recomendable crear un componente que muestre los mensajes de validación:


```
class ValidationMessage extends React.Component {
  render() {
    if (this.props.msg) {
      return <span className="errorMsg">{this.props.msg}</span>;
    }
    return null;
  }
}
```
- Para posteriormente asociar los mensajes a los controles:


```
Nombre: <input type="text" id="nombre" name="nombre" value={this.state.elemento.nombre}
  required minLength="2" maxLength="10" onChange={this.handleChange} />
<ValidationMessage msg={this.state.msgErr.nombre} />
```

© JMA 2016. All rights reserved

Enviar el formulario

- Si los datos del formulario no se van a enviar directamente al servidor como respuesta del formulario, es necesario interceptar y detener el evento submit con preventDefault.


```

      sendClick(e) {
        e.preventDefault();
        // ...
      }

      <form name="miForm" onSubmit={this.sendClick}>
        // ...
        <input type="submit" value="Enviar" />
      </form>
    
```
- O no hacer submit:


```

      <button onClick={this.sendClick} >Enviar</button>
    
```
- Se puede deshabilitar el botón de envío si no pasa la validación, manteniendo en el estado la validez del formulario:


```

      <button disabled={this.state.invalid} onClick={this.sendClick} >Enviar</button>
    
```

© JMA 2016. All rights reserved

Control de Errores

```

handleChange(event) {
  const cmp = event.target.name;
  const valor = event.target.value;
  const errMsg = event.target.validationMessage;
  this.setState((prev, props) => {
    const ele = prev.elemento;
    let errors = prev.msgErr;
    let invalid = false;
    ele[cmp] = valor;
    errors[cmp] = errMsg;
    if (!errors.nombre && cmp === 'nombre') {
      if (valor && valor !== valor.toUpperCase()) {
        errors.nombre = 'Debe ir en mayúsculas.';
      } else {
        errors.nombre = '';
      }
    }
    for (var c in errors) {
      invalid = invalid || (errors[c] !== '' && errors[c] !== null && typeof(errors[c]) !== "undefined");
    }
    return { elemento: ele, msgErr: errors, invalid: invalid }
  })
}

```

© JMA 2016. All rights reserved

Componentes no controlados

- En la mayoría de los casos, es recomendable utilizar componentes controlados para implementar formularios. La alternativa son los componentes no controlados, donde el DOM maneja los datos del formulario.
- Dado que un componente no controlado mantiene la estado en el DOM, a veces es más fácil integrar el código React y no React cuando se usan componentes no controlados.
- Para escribir un componente no controlado, en lugar de escribir un controlador de eventos para cada actualización de estado se pueden usar un campos referencia para obtener del DOM los valores de formulario.
- En el ciclo de vida del render de React, el atributo value de los elementos del formulario anula el valor en el DOM. Con un componente no controlado, a menudo solo desea que React especifique el valor inicial, pero deje las actualizaciones posteriores sin control. Para manejar este caso, se debe especificar el atributo defaultValue en lugar de value.


```
<input type="text" ref={(tag) => this.txtNombre = tag}
      defaultValue={this.props.nombre} />
```

© JMA 2016. All rights reserved

Valor inicial null o undefined

- Para que una entrada sea controlada, el valor no puede ser null o undefined.
- En caso de que el value asignado desde el estado sea null o undefined, el componente se inicia como no controlado pese a tener un controlador en onChange.
- Cuando se cambia el valor la primera vez, se activa controlador en onChange y, si el valor del estado deja de estar null o undefined (aunque tenga la cadena vacía), vuelve a ser un componente controlado.
- Esta segunda transición de no controlado a controlado produce un aviso en la consola de depuración.

Warning: xxx is changing an uncontrolled input of type number to be controlled. Input elements should not switch from uncontrolled to controlled (or vice versa). Decide between using a controlled or uncontrolled input element for the lifetime of the component.
- Para evitar estos avisos hay que sustituir los valores null o undefined en el estado por la cadena vacía: "".

© JMA 2016. All rights reserved

Validación inicial del formulario

```
componentDidMount() {  
  if (this.form) {  
    const errors = {};  
    let invalid = false;  
    for (var cntr of this.form.elements) {  
      if (cntr.name) {  
        errors[cntr.name] = cntr.validationMessage;  
        invalid = invalid || (errors[cntr.name] !== "" && errors[cntr.name] !== null &&  
          typeof(errors[cntr.name]) !== "undefined");  
      }  
    }  
    this.setState({ msgErr: errors, invalid: invalid });  
  }  
}  
  
<form name="miForm" ref={tag => { this.form = tag; }}>
```

© JMA 2016. All rights reserved

ACCESO AL SERVIDOR

© JMA 2016. All rights reserved

Axios

- Cliente HTTP basado en promesas para el navegador y node.js
- Características:
 - Hacer XMLHttpRequests desde el navegador
 - Hacer solicitudes http desde node.js
 - Transformar los datos de solicitud y respuesta
 - Transformaciones automáticas para datos JSON
 - Soporte de API Promise
 - Interceptores de solicitudes y respuestas
 - Cancelar solicitudes
 - Soporte de cliente para proteger contra XSRF

© JMA 2016. All rights reserved

Axios

<https://github.com/axios/axios>

- Instalación:
 - \$ npm install -save axios
- Desde un CDN:
 - `<script src="https://unpkg.com/axios/dist/axios.min.js"></script>`
- Importación:
 - `import axios from 'axios';`
- Peticiones:


```

      axios.get('/user')
        .then(function (response) { console.log(response); })
        .catch(function (error) { console.log(error); });
      axios.post('/user', { firstName: 'Fred', lastName: 'Flintstone' })
        .then(function (response) { console.log(response); })
        .catch(function (error) { console.log(error); });
      
```

© JMA 2016. All rights reserved

Axios

Métodos

- axios(config)
- axios.request(config)
- axios.get(url[, config])
- axios.delete(url[, config])
- axios.head(url[, config])
- axios.options(url[, config])
- axios.post(url[, data[, config]])
- axios.put(url[, data[, config]])
- axios.patch(url[, data[, config]])

Response Schema

```
{
  data: {},
  status: 200,
  statusText: 'OK',
  headers: {},
  config: {},
  request: {}
}
```

© JMA 2016. All rights reserved

Componentes con AJAX

- Las peticiones AJAX son asíncronas y no deben interferir con el renderizado del componente.
- El resultado de la petición se pasa al estado y este dispara que se vuelva a renderizar la página para que se muestre la respuesta.


```
constructor(props) {
  super(props);
  this.state = { listado: [], loading: true };
}
```
- La primera petición se debe realizar en el hook componentDidMount() que se ejecuta después del render() inicial.


```
componentDidMount() {
  this.setState({ loading: true });
  axios.get('/ws/users').then(resp => { this.setState({ listado: resp.data, loading: false }); });
}
```
- Los controladores de eventos pueden hacer peticiones cuyas respuestas cuando pasen al estado re-renderizaran al componente.
- Se pueden utilizar avisos visuales.


```
render() {
  if (this.state.loading)
    return <p>Cargando ...</p>;
  // Pinta el listado
}
```

© JMA 2016. All rights reserved



<https://github.com/ReactTraining/react-router>

© JMA 2016. All rights reserved

Introducción

- En muchas aplicaciones web actuales se desea utilizar rutas desde la parte cliente, el mecanismo “natural” de navegación en la web, y esa es una de las cosas que ReactJS no incluye de serie, pero que se pueden implementar utilizando librerías de terceros. El React Router es una de las más conocidas.
- React Router es una librería para gestionar rutas en aplicaciones que utilicen ReactJS. Está inspirada en el sistema de enrutado de Ember.js y su forma de manejar las rutas es un poco diferente de lo que podemos ver en otros sistemas, como ASP.NET MVC, Angular, Express o Compojure.
- La idea fundamental es poder tener rutas anidadas mapeadas a componentes.
- Para instalar la versión estable:
 - `npm install -save react-router-dom`
- Importación:
 - `import { BrowserRouter as Router, Route, Link } from 'react-router-dom';`

© JMA 2016. All rights reserved

Componentes principales

- **BrowserRouter**
 - Es una envoltura para la zona de navegación. Esta envoltura da acceso al API de historial de HTML5 para mantener la interfaz gráfica sincronizada con la locación actual o URL. Solo puede tener un hijo, un contenedor con los enlaces y un componente Switch.
- **Switch**
 - Este componente, causa que solo se renderice el primer hijo Route o Redirect que coincida con la locación o URL actual.
 - En el caso que no usemos Switch todas las rutas que cumplan con la condición se renderizarán.
- **Route**
 - Permite definir las diferentes rutas de la aplicación y asociarla a un componente. La función de este componente es elegir que renderizar según la locación actual.
 - Todos los componentes Route siempre renderizan, si coincide la ruta renderizan un componente o null si no coincide.

© JMA 2016. All rights reserved

Route

- **path**: la ruta en la que se renderizará el componente en forma de cadena de texto
- **exact**: un booleano para definir si queremos que la ruta tiene o no que ser exacta para renderizar un componente. Ej: `/index !== /index/all`.
- **strict**: un booleano para definir si queremos que el último slash sea tomado en cuenta para renderizar un componente. Ej: `/index !== /index/`.
- **sensitive**: un booleano para definir si queremos distinguir entre minúsculas y mayúsculas, y tomar esto en cuenta para renderizar un componente. Ej: `/index !== /Index`
- **component**: recibe un componente a renderizar. Crea un nuevo elemento de React cada vez. Esto causa que el componente se monte y desmonte cada vez (no actualiza).
- **render**: recibe un método que retorna un componente.

© JMA 2016. All rights reserved

Redirect

- Componente que permite un redireccionamiento a una ruta diferente a la actual. La ruta a la que redirecciona reemplaza la locación actual en la historia del navegador.
- Redirect recibe las siguientes propiedades:
 - from: una locación en forma de cadena de texto u objeto desde la cual se va a hacer el redireccionamiento. Esta propiedad funciona de manera muy similar a la propiedad path de Route, cuando la locación coincide con from, se ejecuta el Redirect.
 - to: una locación en forma de cadena de texto u objeto hacia el cual se va a hacer el redireccionamiento.
 - push: un booleano que cuando este es verdadero en lugar de reemplazar la locación actual en la historia, este agrega una nueva locación.
 - sensitive: un booleano para definir si queremos distinguir entre minúsculas y mayúsculas.
 - exact: un booleano para definir si queremos que la ruta tiene o no que ser exacta para renderizar un componente.
 - strict: un booleano para definir si queremos que el último slash sea tomado en cuenta para renderizar un componente.

© JMA 2016. All rights reserved

Link

- El componente Link crea un hipervínculo que permite navegar por la aplicación sin las implicaciones de usar directamente la etiqueta ``. Al contrario de Redirect, este agrega una nueva locación a la historia.
- Dispone de las siguientes propiedades:
 - to: una ruta en formato cadena. Esta será la locación a la que navegara cuando se hace click en un hipervínculos.
 - replace: un booleano que cuando este es verdadero en lugar de agregar una nueva locación a la historia, este reemplaza la locación actual.
- Deben estar anidados dentro del BrowserRouter que contiene las rutas.
- La versión especial NavLink de Link agrega atributos de estilo (activeClassName, activeStyle, exact, strict) al elemento representado cuando coincida con la URL actual.

© JMA 2016. All rights reserved

BrowserRouter

```
<BrowserRouter>
  <div>
    <div>
      <Link to="/home">Inicio</Link>
      <Link to="/demos">Demo</Link>
      <Link to="/productos">Productos</Link>
    </div>
    <Switch>
      <Redirect from="/" to="/home" exact />
      <Route path="/home" render={() => <Saluda nombre="MUNDO" />} exact />
      <Route path="/demos" component={Demos} exact />
      <Route path="/productos" component={ProductosList} exact />
      <Route path="/productos/:id" component={ProductosList} exact />
      <Route component={PageNotFound} />
    </Switch>
  </div>
</BrowserRouter>
```

© JMA 2016. All rights reserved

match

- Las rutas pueden contar con segmentos dinámicos asignados a parámetros:
 - path='/producto/:id'
- El objeto match contiene información sobre las coincidencias de una ruta.
- Los objetos match contienen las siguientes propiedades:
 - params - (objeto) pares clave / valor analizados desde la ruta
 - isExact - (booleano) true si la URL completa coincide (sin caracteres finales)
 - path - (cadena) El patrón de ruta utilizado para coincidir. Útil para construir <Route> anidados.
 - url - (cadena) La parte coincidente de la URL. Útil para construir <Link> anidados.
- El objeto match es accesible mediante this.props.match en el componente.

© JMA 2016. All rights reserved



<https://redux.js.org/>

© JMA 2016. All rights reserved

Introducción

- Como los requisitos en aplicaciones JavaScript de una sola página se están volviendo cada vez más complicados, nuestro código, mas que nunca, debe manejar el estado. Este estado puede incluir respuestas del servidor y datos cacheados, así como datos creados localmente que todavía no fueron guardados en el servidor. El estado de la UI también se ha vuelto más complejo, al necesitar mantener la ruta activa, que está seleccionado actualmente, que mostrar o no...
- Controlar ese estado cambiante es difícil. Si un modelo puede actualizar otro modelo, entonces una vista puede actualizar un modelo, el cual actualiza otro modelo, y esto causa que otra vista se actualice. En cierto punto, ya no se entiende que esta pasando en la aplicación ya que pierdes el control sobre el cuándo, el por qué y el cómo de su estado. Cuando un sistema es opaco y no determinista, es difícil reproducir errores o agregar nuevas características.
- Siguiendo los pasos de Flux, CQRS y Event Sourcing, Redux intenta hacer predecibles las mutaciones del estado imponiendo ciertas restricciones en cómo y cuándo pueden realizarse las actualizaciones. Estas restricciones se reflejan en los tres principios de Redux.

© JMA 2016. All rights reserved

1º Única fuente de la verdad

- El estado de toda tu aplicación esta almacenado en un árbol guardado en un único store.
- Esto hace fácil crear aplicaciones universales, ya que el estado en tu servidor puede ser serializado y enviado al cliente sin ningún esfuerzo extra.
- Un único árbol de estado también hace más fácil depurar una aplicación; te permite también mantener el estado de la aplicación en desarrollo, para un ciclo de desarrollo más veloz.

© JMA 2016. All rights reserved

2º El estado es de solo lectura

- La única forma de modificar el estado es emitiendo una acción, un objeto describiendo que ocurrió.
- Esto te asegura que ni tu vista ni callbacks de red van a modificar el estado directamente.
- En vez de eso, expresan un intento de modificar el estado.
- Ya que todas las modificaciones están centralizadas y suceden en un orden estricto, no hay que preocuparse por una carrera entre las acciones.
- Como las acciones son objetos planos, pueden ser registrados, serializados y almacenados para volver a ejecutarlos por cuestiones de depuración y pruebas.

© JMA 2016. All rights reserved

3º Los cambios se realizan con funciones puras

- Para especificar como el árbol de estado es transformado por las acciones, se utilizan reducers que son funciones puras.
- Los reducers son funciones puras que toman el estado anterior y una acción, y devuelven un nuevo estado.
- El estado es inmutable, no se puede modificar, se debe crear un nuevo estado que refleje las modificaciones.
- El reducer debe devolver un nuevo objeto de estado en vez de modificar el anterior.
- Se puede empezar con un único reducer y, mientras la aplicación crece, dividirlo en varios reducers pequeños que manejan partes específicas del árbol de estado.
- Ya que los reducers son funciones puras, se puede controlar el orden en que se ejecutan, pasarle datos adicionales, o incluso hacer reducers reusables para tareas comunes como paginación.

© JMA 2016. All rights reserved

Instalación

- Para instalar la versión estable:
 - `npm i -S redux`
- Normalmente también vas a querer usar la conexión a React y las herramientas de desarrollo.
 - `npm i -S react-redux`
 - `npm i -D redux-devtools`
- Si no se quiere instalar Redux DevTools e integrarlo en el proyecto, se puede usar Redux DevTools Extension para Chrome y Firefox. Proporciona acceso a los monitores más populares, es fácil de configurar para filtrar acciones y no requiere instalar ningún paquete.

© JMA 2016. All rights reserved

Elementos básicos

- Acciones: Objetos planos (solo datos) que encapsulan el tipo de modificación a realizar en el almacenamiento con los datos necesarios.
- Creadores de acciones: funciones auxiliares que asisten en la creación de los objetos acción.
- Reducers: Funciones puras que reciben un objeto acción mas el estado actual y realizan la transición para generar el nuevo estado.
- Store: Objeto único, creado conteniendo los reducers, que contiene el estado de la aplicación y es el responsable de permitir su consulta, modificación mediante acciones y notificaciones de los cambios.

© JMA 2016. All rights reserved

Acciones

- Las acciones son objetos planos de JavaScript.
- Una acción debe tener una propiedad type que indica el tipo de operación a realizar sobre el estado.
- Los valores de la propiedad type, los tipos, normalmente se definen como strings constantes.
- La propiedad type irá acompañada de tantas propiedades como sean necesarias para completar la operación definida por la acción.
- Los creadores de acciones encapsulan en una única función la creación del objeto acción, evitando posibles errores y facilitando su manejo.
- Las funciones de envío de acciones no solo crean la acción, la ejecutan también sobre el store.

© JMA 2016. All rights reserved

Acciones

```
export const ADD_LIST = 'ADD_LIST';
export const MODIFY_LIST = 'MODIFY_LIST';
export const DELETE_LIST = 'DELETE_LIST';

export function addList(data) {
  return { type: ADD_LIST, value: data };
}
export function ModifyList(index, data) {
  return { type: MODIFY_LIST, index: index, value: data };
}
export function deleteList(index) {
  return { type: DELETE_LIST, index: index };
}

export const boundAddList = data => dispatch(addList(data));
export const boundDeleteList = index => dispatch({ type: DELETE_LIST, index: index });
```

© JMA 2016. All rights reserved

Reducers

- En Redux, todo el estado de la aplicación es almacenado en un único objeto. Es una buena idea pensar en su forma antes de escribir código. ¿Cuál es la mínima representación del estado de la aplicación como un objeto, como lo estructuramos?
- El reducer es la funciones pura que ejecuta el store para atender la petición dispatch(), toma el estado anterior y la acción suministrada, y devuelve en nuevo estado. Se llama reducer porque es el tipo de función que pasarías a `Array.prototype.reduce(reducer, ?initialValue)`.
- Es muy importante que los reducer se mantengan puros, nunca se debería hacer dentro de un reducer:
 - Modificar sus argumentos;
 - Realizar tareas con efectos secundarios como llamas a un API o transiciones de rutas.
 - Llamar una función no pura, por ejemplo `Date.now()` o `Math.random()`.
- Dados los mismos argumentos, debería calcular y devolver el siguiente estado. Sin sorpresas. Sin efectos secundarios. Sin llamadas a APIs. Sin mutaciones. Solo cálculos.

© JMA 2016. All rights reserved

Reducers

- Nunca se debe asignar nada a algo dentro de state antes de clonarlo primero.
- El método `Object.assign` permite realizar copias superficiales (primer nivel) a un objeto de destino desde un objeto origen
- El método `Object.assign()` permite realizar copias superficiales (primer nivel) de los valores de todas las propiedades enumerables de uno o más objetos fuente a un objeto destino, devolviendo el objeto destino.
`return Object.assign({}, state, modificaciones);`
- Si las operaciones requieren copias profundas, es una buena idea usar utilidades que tienen soporte nativo a actualizaciones profundas como:
 - <http://facebook.github.io/immutable-js/>
 - <https://github.com/kolodny/immutability-helper>
 - <https://github.com/substantial/updeep>

© JMA 2016. All rights reserved

Reducers

```
const initialState = { modo: 'add', listado: [], elemento: {} };
function myAppReducer(state = initialState, action) {
  switch (action.type) {
    case SET_MODAL:
      return Object.assign({}, state, { modo: action.modo });
    case ADD_LIST:
      return Object.assign({}, state, { listado: [ ...state.listado, action.value ] });
    case MODIFY_LIST:
      return Object.assign({}, state, {
        listado: state.listado.map((item, index) => Object.assign({}, item, { index: action.index ? action.value : item }))
      });
    case DELETE_LIST:
      return Object.assign({}, state, {
        listado: state.listado.filter((item, index) => index !== action.index)
      });
    // ...
    default:
      return state
  }
}
```

© JMA 2016. All rights reserved

Separando Reducers

- Cuando la complejidad así lo recomiende es conveniente dividir el reducer original en varios reducers con un subconjunto de acciones cada uno que gestione su propia parte del estado global.
- Esto se denomina composición de reducers, y es un patrón fundamental al construir aplicaciones de Redux.
- Redux viene con una utilidad llamada `combineReducers()` que combina múltiples reducers en un único reducer.

```
import { combineReducers } from 'redux'

export const globalReducer = combineReducers({
  vm: myAppReducer,
  IU: IUReducer,
  cache: cacheReducer
})
```

© JMA 2016. All rights reserved

Store

- El store tiene las siguientes responsabilidades:
 - Contiene el estado de la aplicación;
 - Permite el acceso al estado vía `getState()`;
 - Permite que el estado sea actualizado vía `dispatch(action)`;
 - Registra los listeners vía `subscribe(listener)`;
 - Permite anular el registro de los listeners mediante la función devuelta al realizar el `subscribe(listener)`.
- Es importante destacar que sólo se puede tener un único store en una aplicación Redux. Cuando desees dividir la lógica para el manejo de datos, se usará la composición de reductores en lugar de muchos stores.

© JMA 2016. All rights reserved

Store

- Es fácil crear un store, si ya se tiene el reducer, utilizando el método `createStore()`, opcionalmente se le puede pasar el estado inicial.

```
import { createStore } from 'redux'
import globalReducer from './reducers'
let store = createStore(globalReducer, initialState)
```
- Para consultar el estado:

```
state = store.getState();
```
- Para modificar el estado:

```
store.dispatch(deleteList(1))
```
- Para recibir notificaciones cada vez que cambia el estado:

```
let unsubscribe = store.subscribe(() => console.log(store.getState()))
```
- Para dejar de recibir las notificaciones de cambio de estado:

```
unsubscribe()
```

© JMA 2016. All rights reserved

Flujo de datos

- La arquitectura Redux gira en torno a un flujo de datos estrictamente unidireccional.
- Esto significa que todos los datos de una aplicación siguen el mismo patrón de ciclo de duración, haciendo que la lógica de la aplicación sea más predecible y más fácil de entender.
- También fomenta la normalización de los datos, de modo que no termines con múltiples copias independientes de la misma data sin que se entere una de la otra.
- El ciclo de duración de la data en cualquier aplicación Redux sigue estos 4 pasos:
 1. Haces una llamada a `store.dispatch(action)`.
 2. El store en Redux invoca a la función reductora que le indicaste.
 3. El reductor raíz puede combinar la salida de múltiples reductores en un único árbol de estado.
 4. El store en Redux guarda por completo el árbol de estado devuelto por el reductor raíz.

© JMA 2016. All rights reserved

Uso con React

- Redux no tiene relación alguna con React salvo la implementación del patrón Flux.
- React Redux no está incluido en Redux de manera predeterminada. Debe instalarse explícitamente:
 - `npm install --save react-redux`
- Para asociar React con Redux se recurre a la idea de separación de componentes de presentación y contenedores. Son los componentes contenedores los que se conectan al store que maneja Redux.
 - Para leer datos: Se suscribe al estado en Redux
 - Para manipular datos: Envía acciones a Redux
- Aunque se puede escribir un componente contenedor manualmente, se recomienda generar los componentes contenedores con la función `connect()` de la librería React Redux, ya que proporciona muchas optimizaciones útiles para evitar re-renders innecesarios.
- Un beneficio de utilizar `connect()` es no tener que preocuparse por la implementación del método `shouldComponentUpdate` para mejorar rendimiento.

© JMA 2016. All rights reserved

connect()

- Conecta un componente React a un store Redux.
- No modifica la clase del componente que se le pasa; en su lugar, la envuelve en una nueva clase de componente contenedor conectada para su uso.
- Recibe al menos un par de funciones que mapean las propiedades y las acciones del store al contenedor:
 - `mapStateToProps(state, ownProps)`: mapea las propiedades del estado del componente a las del estado del store mediante suscripciones al store Redux. Opcionalmente se reciben las propiedades del componente.
 - `mapDispatchToProps(dispatch, ownProps)`: mapea las propiedades de elevación de estado del componente a las funciones de envío de acciones mediante `dispatch` al store Redux.

© JMA 2016. All rights reserved

connect()

```
function contador(state = 0, action) {
  switch (action.type) {
    case "COUNTER_UP":
      return state + 1;
    case "COUNTER_DOWN":
      return state - 1;
    default:
      return 0
  }
}

export const globalReducer = combineReducers({
  // ...
  contador
})
```

© JMA 2016. All rights reserved

connect()

```
import React from 'react'
import { connect } from 'react-redux'
const myCounter = ({ contador, onSube, onBaja }) => (
  <div>
    <h1>{contador}</h1>
    <p><button onClick={onSube}>Sube</button>&nbsp;<button onClick={onBaja}>Baja</button></p>
  </div>
)

export const MyCounter = connect(
  (state, ownProps) => {
    return {
      contador: state.contador
    }
  },
  (dispatch, ownProps) => {
    return {
      onSube: () => { dispatch({type: "COUNTER_UP"}) },
      onBaja: () => { dispatch({type: "COUNTER_DOWN"}) },
    }
  }
)(myCounter)
```

© JMA 2016. All rights reserved

<Provider>

- Todos los componentes contenedores necesitan acceso al store Redux para que puedan suscribirse.
- La opción que recomendamos es usar un componente React Redux especial llamado <Provider> para hacer que el store esté disponible para todos los componentes de la aplicación sin pasarlo explícitamente:

```
import { createStore } from 'redux'  
import { Provider } from 'react-redux'  
import { globalReducer } from './reducers'
```

```
let store = createStore(globalReducer)
```

```
ReactDOM.render(  
  <Provider store={store}>  
    <App />  
  </Provider>, document.getElementById('root'));
```

© JMA 2016. All rights reserved



React
Apéndices



© JMA 2016. All rights reserved

JavaScript Object Notation

<http://tools.ietf.org/html/rfc4627>

JSON

© JMA 2016. All rights reserved

Introducción

- JSON (JavaScript Object Notation) es un formato sencillo para el intercambio de información.
- El formato JSON permite representar estructuras de datos (arrays) y objetos (arrays asociativos) en forma de texto.
- La notación de objetos mediante JSON es una de las características principales de JavaScript y es un mecanismo definido en los fundamentos básicos del lenguaje.
- En los últimos años, JSON se ha convertido en una alternativa al formato XML, ya que es más fácil de leer y escribir, además de ser mucho más conciso.
- No obstante, XML es superior técnicamente porque es un lenguaje de marcado, mientras que JSON es simplemente un formato para intercambiar datos.
- La especificación completa del JSON es la RFC 4627, su tipo MIME oficial es application/json y la extensión recomendada es .json.

© JMA 2016. All rights reserved

Estructuras

- JSON está constituido por dos estructuras:
 - Una colección de pares de nombre/valor. En los lenguajes esto es conocido como un objeto, registro, estructura, diccionario, tabla hash, lista de claves o un arreglo asociativo.
 - Una lista ordenada de valores. En la mayoría de los lenguajes, esto se implementa como tablas, arreglos, vectores, listas o secuencias.
- Estas son estructuras universales; virtualmente todos los lenguajes de programación las soportan de una forma u otra. Es razonable que un formato de intercambio de datos que es independiente del lenguaje de programación se base en estas estructuras.

© JMA 2016. All rights reserved

Sintaxis

- Un array es un conjunto de valores separados por comas (,) que se encierran entre corchetes [...]
- Un objeto es un conjunto de pares nombre:valor separados por comas (,) que se acotan entre llaves { ... }
- Los nombres son cadenas, entre comillas dobles (").
- El separador entre el nombre y el valor son los dos puntos (:)
- El valor debe ser un objeto, un array, un número, una cadena o uno de los tres nombres literales siguientes (en minúsculas):
 - true, false o null
- Se codifica en Unicode, la codificación predeterminada es UTF-8.

© JMA 2016. All rights reserved

Valores numéricos

- La representación de números es similar a la utilizada en la mayoría de los lenguajes de programación.
- Un número contiene una parte entera que puede ser prefijada con un signo menos opcional, que puede ser seguida por una parte fraccionaria y / o una parte exponencial.
- La parte fraccionaria comienza con un punto (como separador decimal) seguido de uno o más dígitos.
- La parte exponencial comienza con la letra E en mayúsculas o minúsculas, lo que puede ser seguido por un signo más o menos, y son seguidas por uno o más dígitos.
- Los formatos octales y hexadecimales no están permitidos. Los ceros iniciales no están permitidos.
- No se permiten valores numéricos que no se puedan representar como secuencias de dígitos (como infinito y NaN).

© JMA 2016. All rights reserved

Valores cadena

- La representación de las cadenas es similar a las convenciones utilizadas en la familia C de lenguajes de programación.
- Una cadena comienza y termina con comillas (").
- Se pueden utilizar todos los caracteres Unicode dentro de las comillas con excepción de los caracteres que se deben escapar: los caracteres de control (U + 0000 a U + 001F) y los caracteres con significado.
- Cuando un carácter se encuentra fuera del plano multilingüe básico (U + 0000 a U + FFFF), puede ser representado por su correspondiente valor hexadecimal. Las letras hexadecimales A-F puede ir en mayúsculas o en minúsculas.
- Secuencias de escape:
 - `\\`, `\`, `\"`, `\n`, `\r`, `\b`, `\f`, `\t`
 - `\u[0-9A-Fa-f]{4}`

© JMA 2016. All rights reserved

Objeto con anidamientos

```
{
  "Image": {
    "Width": 800,
    "Height": 600,
    "Title": "View from 15th Floor",
    "Thumbnail": {
      "Url": "/image/481989943",
      "Height": 125,
      "Width": "100"
    },
    "IDs": [116, 943, 234, 38793]
  }
}
```

© JMA 2016. All rights reserved

Array de objetos

```
[
  {
    "precision": "zip",
    "Latitude": 37.7668,
    "Longitude": -122.3959,
    "City": "SAN FRANCISCO",
    "State": "CA",
    "Zip": "94107"
  },
  {
    "precision": "zip",
    "Latitude": 37.371991,
    "Longitude": -122.026020,
    "City": "SUNNYVALE",
    "State": "CA",
    "Zip": "94085"
  }
]
```

© JMA 2016. All rights reserved

JSON en JavaScript

- El Standard Built-in ECMAScript Objects define que todo interprete de JavaScript debe contar con un objeto JSON como miembro del objeto Global.
- El objeto debe contener, al menos, los siguientes miembros:
 - **JSON.parse** (Función): Convierte una cadena de la notación de objetos de JavaScript (JSON) en un objeto de JavaScript.
 - **JSON.stringify** (Función): Convierte un valor de JavaScript en una cadena de la notación de objetos JavaScript (JSON).

© JMA 2016. All rights reserved

REpresentational State Transfer

SERVICIOS RESTFUL

© JMA 2016. All rights reserved

REST (REpresentational State Transfer)

- Un **estilo de arquitectura** para desarrollar aplicaciones web distribuidas que se basa en el uso del protocolo HTTP e Hypermedia.
- Definido en el 2000 por Roy Fielding, para no reinventar la rueda, se basa en aprovechar lo que ya estaba definido en el HTTP pero que no se utilizaba.
- El HTTP ya define 8 métodos (algunas veces referidos como "verbos") que indica la acción que desea que se efectúe sobre el recurso identificado:
 - HEAD, GET, POST, PUT, DELETE, TRACE, OPTIONS, CONNECT
- El HTTP permite en el encabezado transmitir la información de comportamiento:
 - Accept, Content-type, Response (códigos de estado), Authorization, Cache-control, ...

© JMA 2016. All rights reserved

Uso de la cabecera

- **Request:** Método /uri?parámetros
 - GET: Recupera el recurso
 - Todos: Sin parámetros
 - Uno: Con parámetros
 - POST: Crea un nuevo recurso
 - PUT: Edita el recurso
 - DELETE: Elimina el recurso
- **Accept:** Indica al servidor el formato o posibles formatos esperados, utilizando MIME.
- **Content-type:** Indica en que formato está codificado el cuerpo, utilizando MIME
- **Response:** Código de estado con el que el servidor informa del resultado de la petición.

© JMA 2016. All rights reserved

Peticiones

Request: GET /users
 Response: 200
 content-type:application/json
 Request: GET /users/11
 Response: 200
 content-type:application/json
 Request: POST /users
 Response: 201
 content-type:application/json
 body
 Request: PUT /users/11
 Response: 200
 content-type:application/json
 body
 Request: DELETE /users/11
 Response: 204 no content

© JMA 2016. All rights reserved

GraphQL

<http://graphql.org/>

- GraphQL es un lenguaje para ejecutar consultas en un API de servidor mediante el cual se provee al cliente de una descripción detallada de las estructuras de datos que ofrece.
- Este lenguaje fue creado por Facebook en 2012, cuya estandarización se comenzó a desarrollar en 2015, por lo que es un lenguaje en continuo desarrollo.
- Las principales características y funcionalidades que ofrece GraphQL son las siguientes:
 - Describe la estructura/organización de los datos ofrecidos por el API mediante un sistema de fuertemente tipado en el que se detallan los recursos y las relaciones existentes entre estos.
 - Permite solicitar al servidor tan sólo la información que necesitamos de un recurso en una sola petición incluyendo los datos de otros recursos que se relacionen con el recurso raíz.
 - Ofrece una única URI en la que se realizan todas las peticiones, por lo que no debemos preocuparnos de la semántica implementada por cada API.
 - Es un lenguaje independiente de la fuente de datos. Los datos pueden obtenerse de un servicio web, una base de datos, un fichero, etc.
 - Permite la evolución de un API sin crear nuevas versiones
 - Existen múltiples librerías para implementar en distintos lenguajes (JavaScript, .NET, Ruby, Java ...)

© JMA 2016. All rights reserved

GraphQL

Definición de datos

```
type Human implements Character {
  id: ID!
  name: String!
  friends: [Character]
  appearsIn: [Episode]!
  starships: [Starship]
  totalCredits: Int
}
enum Episode {
  NEWHOPE
  EMPIRE
  JEDI
}
type Starship {
  id: ID!
  name: String!
  length(unit: LengthUnit = METER): Float
}
```

Consulta

```
{
  human(id: 1002) {
    name
    appearsIn
    starships {
      name
    }
  }
}
```

<http://graphql.org/swapi-graphql/>

© JMA 2016. All rights reserved

GraphQL

Resultado

```
{
  "data": {
    "human": {
      "name": "Han Solo",
      "appearsIn": [
        "NEWHOPE",
        "EMPIRE",
        "JEDI"
      ],
      "starships": [
        {
          "name": "Millenium Falcon"
        },
        {
          "name": "Imperial shuttle"
        }
      ]
    }
  }
}
```

© JMA 2016. All rights reserved

Patrón Agregado (Aggregate)

- Una Agregación es un grupo de objetos asociados que deben tratarse como una unidad a la hora de manipular sus datos.
- El patrón Agregado es ampliamente utilizado en los modelos de datos basados en Diseños Orientados al Dominio (DDD).
- Proporciona una forma de encapsular nuestras entidades y los accesos y relaciones que se establecen entre las mismas de manera que se simplifique la complejidad del sistema en la medida de lo posible.
- Cada Agregación cuenta con una Entidad Raíz (root) y una Frontera (boundary):
 - La Entidad Raíz es una Entidad contenida en la Agregación de la que colgarán el resto de entidades del agregado y será el único punto de entrada a la Agregación.
 - La Frontera define qué está dentro de la Agregación y qué no.
- La Agregación es la unidad de persistencia, se recupera toda y se almacena toda.

© JMA 2016. All rights reserved

Seguridad

- La ejecución de aplicaciones JavaScript puede suponer un riesgo para el usuario que permite su ejecución.
- Por este motivo, los navegadores restringen la ejecución de todo código JavaScript a un entorno de ejecución limitado.
- Las aplicaciones JavaScript no pueden establecer conexiones de red con dominios distintos al dominio en el que se aloja la aplicación JavaScript.
- Los navegadores emplean un método estricto para diferenciar entre dos dominios ya que no permiten ni subdominios ni otros protocolos ni otros puertos.
- Si el código JavaScript se descarga desde la siguiente URL:
<http://www.ejemplo.com>
- Las funciones y métodos incluidos en ese código no pueden acceder a:
 - **https://www.ejemplo.com/scripts/codigo2.js**
 - **http://www.ejemplo.com:8080/scripts/codigo2.js**
 - **http://scripts.ejemplo.com/codigo2.js**
 - **http://192.168.0.1/scripts/codigo2.js**
- La propiedad `document.domain` se emplea para permitir el acceso entre subdominios del dominio principal de la aplicación.

© JMA 2015. All rights reserved

CORS

- Un recurso hace una solicitud HTTP de origen cruzado cuando solicita otro recurso de un dominio distinto al que pertenece y, por razones de seguridad, los exploradores restringen las solicitudes HTTP de origen cruzado iniciadas dentro de un script.
- XMLHttpRequest sigue la política de mismo-origen, por lo que, una aplicación usando XMLHttpRequest solo puede hacer solicitudes HTTP a su propio dominio. Para mejorar las aplicaciones web, los desarrolladores pidieron a los proveedores de navegadores que permitieran a XMLHttpRequest realizar solicitudes de dominio cruzado.
- El Grupo de Trabajo de Aplicaciones Web del W3C recomienda el nuevo mecanismo de Intercambio de Recursos de Origen Cruzado (CORS, Cross-origin resource sharing). Los servidores deben indicar al navegador mediante cabeceras si aceptan peticiones cruzadas y con que características:
 - "Access-Control-Allow-Origin", "*"
 - "Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type, Accept"
 - "Access-Control-Allow-Methods", "GET, POST, PUT, DELETE"
- Soporte: Chrome 3+ Firefox 3.5+ Opera 12+ Safari 4+ Internet Explorer 8+

© JMA 2016. All rights reserved

Desactivar la seguridad de Chrome

- Pasos para Windows:
 - Localizar el acceso directo al navegador (icono) y crear una copia como "Chrome Desarrollo".
 - Botón derecho -> Propiedades -> Destino
 - Editar el destino añadiendo el parámetro al final. ej: "C:\Program Files (x86)\Google\Chrome\Application\chrome.exe" --disable-web-security
 - Aceptar el cambio y lanzar Chrome
- Para desactivar parcialmente la seguridad:
 - allow-file-access
 - allow-file-access-from-files
 - allow-cross-origin-auth-prompt
- Referencia a otros parametros:
 - <http://peter.sh/experiments/chromium-command-line-switches/>

© JMA 2016. All rights reserved

JSONP (JSON con Padding)

- JSONP es una técnica de comunicación utilizada en los programas JavaScript para realizar llamadas asíncronas a dominios diferentes. JSONP es un método concebido para suplir la limitación de AJAX entre dominios por razones de seguridad. Esta restricción no se aplica a la etiqueta `<script>` de HTML, para la cual se puede especificar en su atributo `src` la URL de un script alojado en un servidor remoto.
- En esta técnica se devuelve un objeto JSON envuelto en la llamada de una función (debe ser código JavaScript válido), la función ya debe estar definida en el entorno de JavaScript y se encarga de manipular los datos JSON.
`miJsonCallback ({ "Nombre": "Carmelo", "Apellidos": "Cotón" });`
- Por convención, el nombre de la función de retorno se suele especificar mediante un parámetro de la consulta, normalmente, utilizando `jsonp` o `callback` como nombre del campo en la solicitud al servidor.
`<script type="text/javascript" src="http://otrodominio.com/datos.json?callback=miJsonCallback"></script>`