React

https://es.reactjs.org/
https://beta.reactjs.org/learn

CLE

- Introducción a React
  - ▪ ¿Qué es MV * y SPA (aplicación de una sola página)?
  - ▪ Diferencia entre MV * frameworks y React
  - ▪ React vs. bibliotecas (por ejemplo, jQuery) y otros marcos (por ejemplo,
  - Backbone.js)
  - ▪ Descripción general de React
- React Avanzado
  - ▪ Fragments
  - ▪ Context API
  - ▪ Higher Order Components
  - ▪ React Hooks

INTRODUCCIÓN A REACT

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.

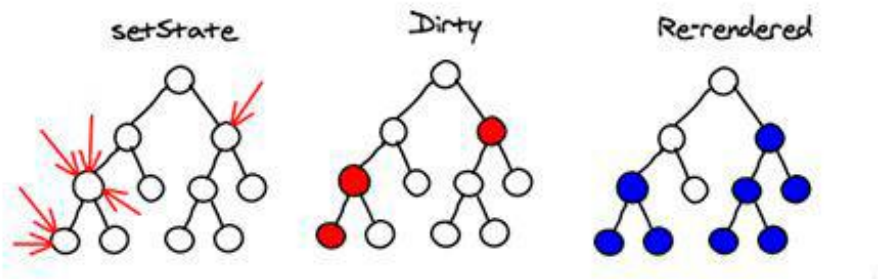
Conceptos

- 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 pero 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.

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, ReactJS hace los cambios en una copia en memoria y después usa un algoritmo para comparar las propiedades de la copia en memoria con las propiedades de la versión actual 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.

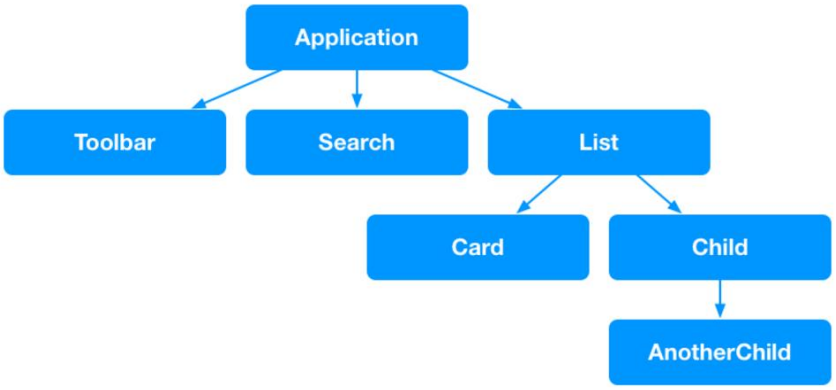
Virtual DOM



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 si 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.

Árbol de componentes



Tipos de componentes

- En React JS existen dos categorías recomendadas para los componentes (patrón contenedor/presentadores):
  - 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.

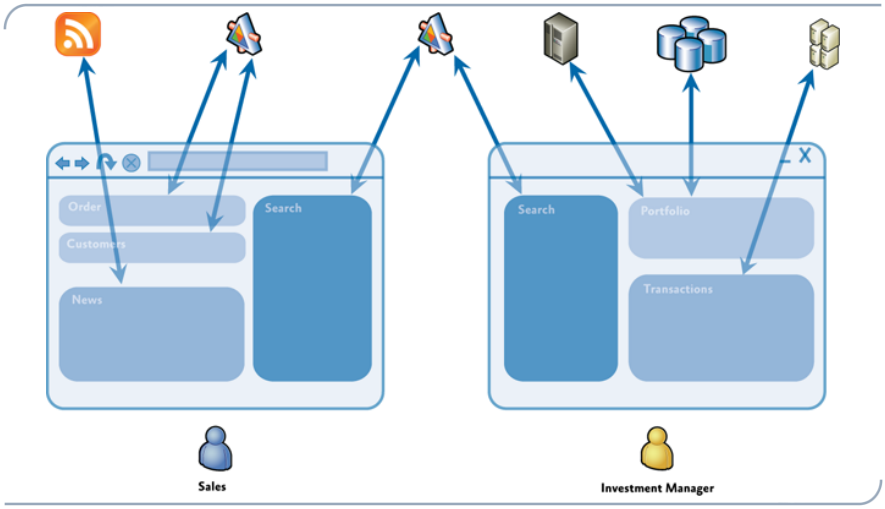
Características

- 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

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.

Patrón Composite View



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 códigos 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).

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.

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')
);
```

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

ECMASCRIPT 2015

Introducción

- En Junio de 2015 aparece la 6ª edición del estándar ECMAScript (JavaScript 6), cuyo nombre oficial es ECMAScript 2015.
- Se está extendiendo progresivamente el soporte a ES2015, pero aún queda trabajo por hacer.
- Las alternativas para empezar a utilizarlo son:
  - Transpiladores ("Transpiler": "Tanslator" y "Compiler"): Traducen o compilan un lenguaje de alto nivel a otro lenguaje de alto nivel, en este caso código ES2015 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 ES2015 en aquellos navegadores que nativamente no lo soportan.

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

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
```

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, …).

Destructuring

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

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

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

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.

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


Función Arrow

- Funciones anónimas.

```
data.forEach(elem => {
   console.log(elem);
   // …
});
let 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));
   }
```

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
   …
};
```


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() {
      // ...
   }
}
```

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)
   }
}
let defRectangle = Rectangle.defaultRectangle()
let defCircle    = Circle.defaultCircle()
```


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; }
}
let r = new Rectangle(50, 20)
if(r.area === 1000) r.width = 10;
```

mixin

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

```
let 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
}
```


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";
let app = {
  foo: function() {
    hello("Carlos");
  }
}
export app;
```

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)

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.

```
let iter = myGenerator();
// ...
rslt = iter.next();
// ...
```

- La nueva sintaxis del for permite recorrer el iterador completo:

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

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.

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.

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))) → 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.

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.

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.

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.

HERRAMIENTAS DE DESARROLLO

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

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

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 grunt-init karma karma-cli ← Global (CLI)
  - npm install jasmine-core tslint --save --save-dev
  - npm install ← Dependencias en package.json
- Arranque del servidor:
  - npm start

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

create-react-app

https://create-react-app.dev/

- 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.

Características

- Soporte de sintaxis React, JSX, ES2015 y Flow.
- Extras de lenguaje más allá del ES2015 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 .

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

  $ npm init react-app my-app *(disponible a partir de npm 6+)*
  - 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.

Estructura de directorios de soporte

- public
  - favicon.ico
  - index.html
  - logo192.png
  - logo512.png
  - manifest.json
  - robots.txt
- src
  - App.css
  - App.js
  - App.test.js
  - index.css
  - index.js
  - logo.svg
  - reportWebVitals.js
  - setupTests.js
- .gitignore
- package.json
- README.md

- src: Fuentes de la aplicación
- public: Ficheros públicos de la aplicación.
- Ficheros de configuración de librerías y herramientas
- Posteriormente aparecerán los siguientes directorios:
  - node_modules: Librerías y herramientas descargadas
  - build: Resultado para publicar en el servidor web

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 archivo para impedir los problemas de las 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.

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

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 (ES2015 con Babel, TypeScript, SaSS, etc).
  - Sus plugins permiten hacer otras tareas importantes como por ejemplo minimizar y ofuscar el código.

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“
},
```


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: *

Aplicación

index.html

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

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

  <body>
    <div id="root"></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('root'));
```


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.

Web Worker

- Cuando se ejecuta una secuencias 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.

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

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';

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} />
```

- 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.

```
REACT_APP_SECRET=Mi secreto
```

- 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 utilza .env.local)

Analizadores

- Los linters de código encuentran problemas en el código a medida que escribe, lo que ayuda a solucionarlos de forma temprana.
- ESLint es el mas popular analizador de código abierto para JavaScript.
  - npm i eslint @babel/eslint-parser eslint-config-standard eslint-config-standard-jsx eslint-plugin-promise eslint-plugin-import eslint-plugin-node eslint-plugin-react eslint-config-react-app eslint-plugin-jest --save-dev
- Configuración en .eslintrc.json:

```
{
  "parser": "@babel/eslint-parser",
  "extends": ["standard", "standard-jsx", "react-app", "react-app/jest"]
}
```

- Configuración en package.json:

```
"eslintConfig": {
  "parser": "@babel/eslint-parser",
  "extends": ["standard", "standard-jsx", "react-app", "react-app/jest"]
}
```


Formateo

- Lo último que se desea hacer cuando se comparte código con otro colaborador es entrar en una discusión sobre tabulaciones versus espacios, espaciado vertical, horizontal, …
- Afortunadamente, Prettier limpiará el código reformateándolo para que se ajuste a las reglas preestablecidas y configurables, unificando el formato para todos los participantes independientemente de sus preferencias personales.
  - npm i prettier --save-dev
- Ejecutando Prettier, y todas las tabulaciones se convertirán en espacios, y las sangrías, comillas, etc. también se cambiarán para ajustarse a la configuración.
- En la configuración ideal, Prettier se ejecutará cuando se guarde el archivo, haciendo estas ediciones rápidamente por nosotros.

JSX

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.

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 **rápido** escribir plantillas si se está familiarizado con el lenguaje HTML.
- Desventajas:
  - Requiere compilación.

Sintaxis

- Es obligatorio utilizar la notación PascalCase en:
  - Componentes: MiComponente
- Es obligatorio utilizar la notación camelCase en:
  - Atributos: tabIndex
  - Eventos: onClick
- Tiene una notación alternativa para los atributos HTML que son palabras reservadas en JavaScript:
  - 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):
  - …
…
  - <input … />
- Se puede usar un convertidor para traducir el HTML y SVG existente a JSX.

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 obligatorio, 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>
);
```

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


Expresiones

- Se puede incrustar cualquier expresión de JavaScript en un fragmento HTML 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), los objetos definidos se deben usar con dobles llaves:

```
const constante = <img src="http://lorempixel.com/64/64/people/" />;
const expresion = <img src={user.avatarUrl} width={200} />;
const error = <output style={{ color: 'red' }}>{msg}</output>
```

- Los comentarios son expresiones que se ignoran:

```
{/* … */}
```

Expresiones complejas

- Si no se asigna valor al atributo se establece el valor true de manera predeterminada.

```
<MyTextBox autocomplete />
<MyTextBox autocomplete={true} />
```

- Si ya se dispone de las propiedades en un objeto se puede pasar utilizando el operador ... de "propagación" (ES2015):

```
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} />;
```


Expresiones condicionales

- No se renderizan los valores true, false, null y undefined.

```
<div>{true}</div>  se visualiza como  <div></div>
```

- Operador condicional simple:

```
const saludo= <p>Hola {user && <b>formatName(user)</b>}</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 lamda se pueden utilizar instrucciones condicionales y bucles:

```
function saludar(user) {
  if (user) {
    return <h1>Hola {formatName(user)}!</h1>;
  }
  return <h1>Hola desconocido.</h1>;
}
```

dangerouslySetInnerHTML

- 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).
- dangerouslySetInnerHTML es el atributo que reemplaza a la propiedad DOM innerHTML, lo que significa “establecer HTML interno arriesgadamente”.
- En general, es arriesgado establecer contenido HTML desde el código sin el correspondiente saneamiento, porque se puede exponer inadvertidamente a los usuarios a un ataque cross-site scripting (XSS).
- Por lo tanto, para establecer contenido HTML directamente desde React, se debe usar el atributo dangerouslySetInnerHTML y pasarle un objeto con una propiedad __html, como recordatorio de que es peligroso.

```
function createMarkup() {
  return {__html: 'First &middot; Second'};
}
function MyComponent() {
  return <div dangerouslySetInnerHTML={createMarkup()} />;
}
```


Expresiones iterativas

- 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 identificadores 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>
);
```

Estilo

- En JSX, las propiedades style y class presentan algunas diferencias con el HTML/DOM.
- El propiedad style acepta una cadena CSS o un objeto JavaScript plano, pares nombre/valor, donde: el nombre de la propiedad que debe coincidir con el nombre de la propiedad CSS, entrecomillado si es kebab-case (aunque para ser consistentes con la forma en que se accede a los estilos de los nodos DOM en JS se dispone de sus correspondencias en camelCase), y el valor entrecomillado con el valor CSS.

  style={{'background-color': 'azure', fontWeight: 'bold', fontSize: '2em', opacity: '0.8'}}
- React adjuntará automáticamente el sufijo “px” a ciertas propiedades numéricas. Para usar unidades diferentes a “px” se debe especifica el valor como un string con la unidad deseada.
- Estos objeto pueden estar en un atributo o ser generados por métodos.
- Para especificar una clase CSS, recomendable, se usa el atributo className para evitar la palabra reservada class. Esto se aplica a todos los elementos regulares de DOM y SVG como , <a>, y otros.

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 del 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.

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.

COMPONENTES

Introducción

- Los componentes permiten dividir la interfaz de usuario en piezas independientes y reutilizables, se debe 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 en notación Pascal:

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

Esta función es un componente React válido porque acepta un único argumento objeto "props" con datos y devuelve un elemento React.

- También se puede usar una clase ES2015 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.

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>
  );
}
```

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')
);
```


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, en general, 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.

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" (ES2015):

```
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} />;
```


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 valida 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 salto de línea en medio de los literales cadena que se condensan en un solo espacio.

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" />;
```

- Desde la versión 15.5 de React hay que instalar la biblioteca prop-types a parte:
  - npm install prop-types

PropTypes

- Tipos primitivos: array, bool, func, number, object, string, symbol, any, element (de React), node (cualquier valor que pueda se 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”:

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

PropTypes

- 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 la propiedades se establecen como una estructura que se asigna al campo static defaultProps del componente.

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


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 la 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
    };
  }
```

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.sube}>Sube</button> 
        <button onClick={this.baja}>Baja</button>
      </p>
    </div>
  );
 }
}
```

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

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 }
  });
```

Campos de la clase

- Las propiedades (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 utilizar en la salida visual, por lo que no requieren desencadenar el repintado al ser modificado.

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

- Cuando se crea un constructor, los componentes de clase siempre deben invocar al constructor base para propagar las props.

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 (callback) 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>

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:

```
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>
```


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 las clases JavaScript, si se refiere a un método sin () se debe vincular dicho método.
- No es necesario vincular los campos públicos ni las funciones flecha.
- El uso de las funciones flecha o anónimas tienen la particularidad de crear un nuevo objeto función de devolución de llamada 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, dichos 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.

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. React genera JavaScript sin HTML por lo que no es intrusivo de cara a la accesibilidad.
- 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 asincrónica, hay que cachear su información.

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

Eventos compatibles

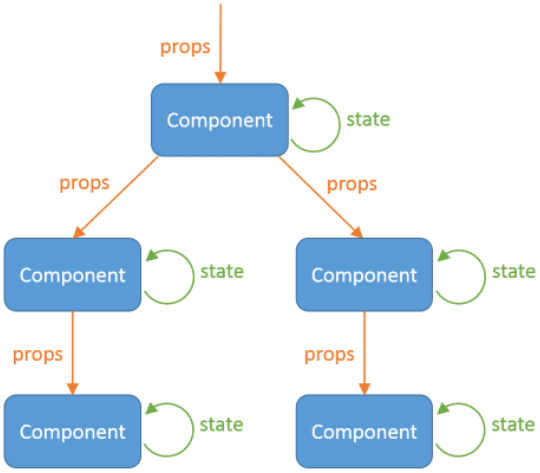
Clipboard	onCopy onCut onPaste
Composition	onCompositionEnd onCompositionStart onCompositionUpdate
Keyboard	onKeyDown onKeyPress onKeyUp
Focus	onFocus onBlur
Mouse	onClick onContextMenu onDoubleClick onDrag onDragEnd onDragEnter onDragExit onDragLeave onDragOver onDragStart onDrop onMouseDown onMouseEnter 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

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.

Flujo de datos



Flujo de datos inverso

- En React se denomina *elevar estado* 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:

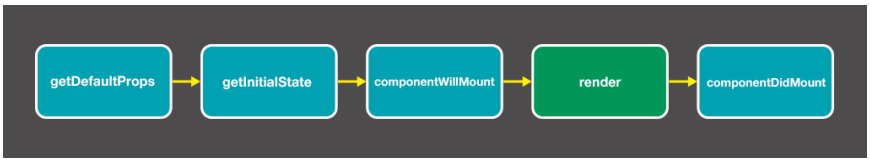
```
if(this.props.onCambia)
    this.props.onCambia(this.state.contador);
```

- Hay que tener cuidado para no provocar un bucle infinito, que el cambio no reentre en el componente provocando un nuevo cambio.

Ciclo de vida del componente

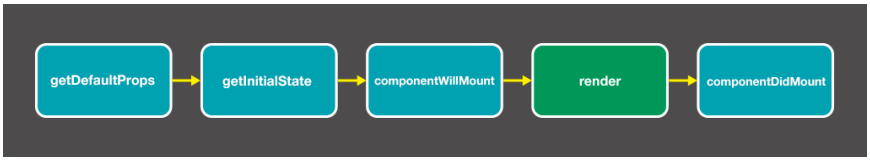
- Para comprender el comportamiento de los componentes de React es importante comprender su ciclo de vida y los métodos que intervienen en dicho ciclo. Lo interesante de esto es que los métodos se pueden sobreescribir para conseguir que el componente se comporte de la manera que se espera.
- Podemos 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.

Montaje



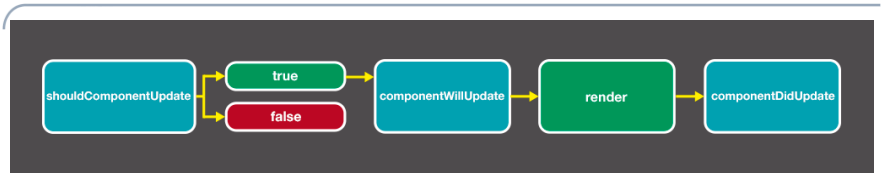
- **getDefaultProps y getInitialState** o el **constructor(props)** de la clase: se ejecuta cuando se instancia un componente. Nos permite definir el estado inicial del componente, hacer el bind de métodos y definir los campos de la clase.
- ~~**componentWillMount()**~~: se ejecuta cuando el componente 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).

Montaje



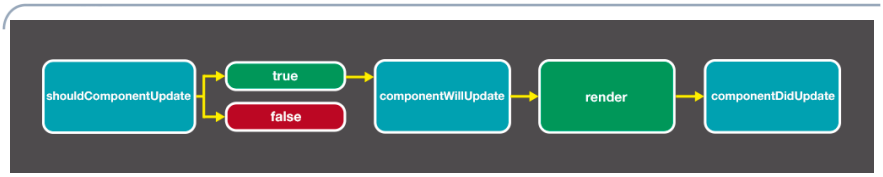
- **render()**: único obligatorio, en esta fase genera la UI inicial del componente. Debe ser una función pura (no puede tener efectos secundarios) y no debe modificar nunca el estado del componente.
- **componentDidMount()**: se ejecuta cuando el componente ya se ha renderizado en el navegador y permite 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.

Actualización de estado



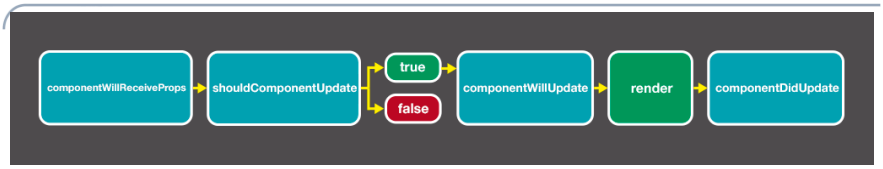
- **shouldComponentUpdate(next_props, next_state)**: decide si el componente se vuelve a renderizar o no. Recibe como parámetros las nuevas propiedades y el nuevo estado, debe devolver true para que se renderice. False puede afectar a sus componentes hijos que no reciben las nuevas propiedades.
- ~~**componentWillUpdate**~~**(next_props, 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.

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.

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.

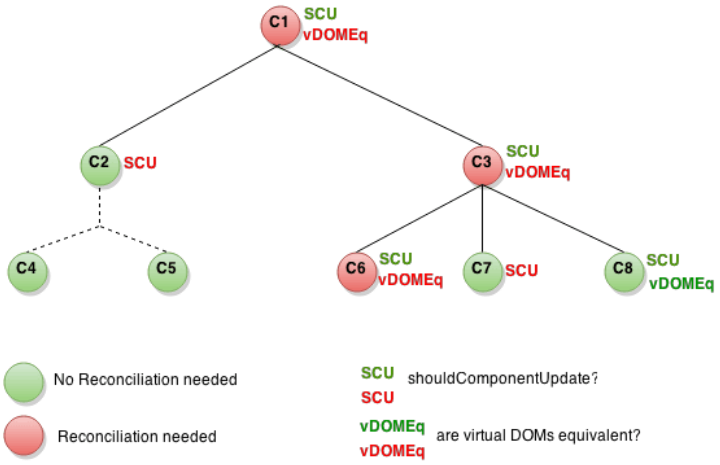
Desmontaje

- 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.

Nuevo ciclo de vida (v16.3)

- Se marcan como obsoletos (se prefijaran como "UNSAFE_" en una próxima versión v17.0) los métodos que fomentan prácticas de codificación inseguras: componentWillMount, componentWillReceiveProp y componentWillUpdate.
- Aparecen un par de métodos adicionales de uso muy poco común:
  - static getDerivedStateFromProps(props, state): Se invoca justo antes de llamar al método de render, tanto en la montaje inicial como en las actualizaciones posteriores. Debe devolver un objeto para actualizar el estado o null para no actualizar nada. Para casos de uso raros donde el estado depende de los cambios tempodependientes en props.
  - getSnapshotBeforeUpdate(prevProps, prevState): Se invoca justo antes de que la salida renderizada más reciente se entregue, por ejemplo, al DOM. Permite al componente capturar cierta información del DOM (por ejemplo, la posición del scroll) antes de que se cambie potencialmente. Cualquier valor que se devuelva en este ciclo de vida se pasará como parámetro al método componentDidUpdate().

Reconciliación del DOM



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.

Campos Referencia (v16.3)

- El API React.createRef() introducido en React 16.3 como alternativa a las referencias mediante callback.
- Las referencias se crean usando React.createRef() y, habitualmente, se asignan en el constructor a una propiedad de la instancia del componente.

```
constructor(props) {
  super(props);
  this.btnSalir = React.createRef();
}
```

- Mediante el atributo ref una función permite asociar un elemento DOM o componente de clase React a una propiedad creada con React.createRef():

```
<button ref={this.btnSalir}>Salir</button>
```

- Una vez renderizado el nodo es accesible a través de la propiedad current de la referencia:

```
this.btnSalir.current.focus();
```

- Dependiendo del tipo de nodo el contenido será el elemento HTML DOM de la etiqueta o la instancia montada del componente.

Reenvío de Referencias

- El reenvío de referencias es una técnica para pasar automáticamente una referencia a través de un componente a uno de sus hijos. Esto normalmente no es necesario para la mayoría de los componentes React en una aplicación: ocultan sus detalles de implementación, incluyendo su salida renderizada, con lo se que previene que los componentes dependan demasiado de la estructura DOM de otros. Sin embargo, puede ser útil para ciertos tipos de componentes, especialmente en bibliotecas de componentes reutilizables.
- Se usa React.forwardRef para definir la ref recibida y asociarla a un elemento que se renderiza:

```
const FancyButton = React.forwardRef((props, ref) => (
  <button ref={ref} className="FancyButton">
    {props.children}
  </button>
));
```

- Ahora se puede suministrar la referencia y obtener el nodo asignado:

```
this.miRef = React.createRef();
<FancyButton ref={this.miRef}>Click me!</FancyButton>;
this.miRef.current // <button className="FancyButton"
```

- El argumento ref solo existe si se define el componente con React.forwardRef.

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 los errores producidos:
  - Durante el renderizado.
  - En métodos del ciclo de vida.
  - En constructores de todo el árbol bajo ellos.
- Los límites de errores no capturan errores que deben tratarse con try en:
  - 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.

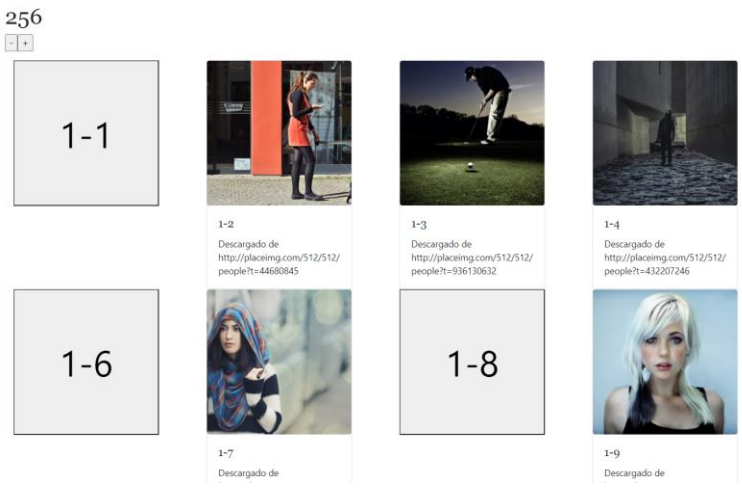
Límites de Errores

- 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 provocaran el desmontaje de todo el árbol de componentes de React.

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;
  }
}
```


Muro de imágenes



FORMULARIOS

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 diferente:
  - componentes controlados
  - componentes no controlados.

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 a 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 HTMLElement que disparo el evento.

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.

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()});
```


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') {
    // …
}
```

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 mensaje 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} />
```


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" />
```

- 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>
```

Control de Errores

```
handleChange(event) {
    const cmp = event.target.name;
    const valor = event.target.type === 'checkbox' ? event.target.checked : 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 (let c in errors) {
        invalid = invalid || (errors[c] !== '' && errors[c] !== null && typeof(errors[c]) !== "undefined");
      }
      return { elemento: ele, msgErr: errors, invalid: invalid }
    })
  }
```


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} />
```

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: ''.

Validación inicial del formulario

```
componentDidMount() {
   if (this.form) {
      const errors = {};
      let invalid = false;
      for (let 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; }}>
```

Frameworks

- Los formularios son muy farragosos en React, las partes más molestas son:
  - Obtener valores dentro y fuera del estado del formulario
  - Validar y mostrar mensajes de error
  - Manejar el envío del formularios
- Por ello existen multitud de frameworks para simplificar dichas tareas:
  - Formik (https://formik.org/)
  - Yup (https://github.com/jquense/yup)
  - React Hook Form (https://react-hook-form.com/)

ACCESO AL SERVIDOR

API Fetch

- La API Fetch proporciona una interfaz JavaScript para acceder y manipular partes del canal HTTP, como peticiones y respuestas.
- También provee un método global fetch() que proporciona una forma fácil y lógica de obtener recursos de forma asíncrona por la red, basado en promesas.
- Este tipo de funcionalidad se conseguía previamente haciendo uso de XMLHttpRequest.
- Fetch proporciona una mejor alternativa que puede ser empleada fácilmente por otras tecnologías como Service Workers.
- Fetch también aporta un único lugar lógico en el que definir otros conceptos relacionados con HTTP como CORS y extensiones para HTTP.
- Para utilizar fetch() en un explorador no soportado (IExplorer), hay disponible un Polyfill en https://github.com/github/fetch que recrea la funcionalidad para navegadores no soportados.

Petición GET

```
this.setState({ loading: true });
fetch(URL_BASE).then(
  resp => {
    if (resp.ok) {
      resp.json().then(data => this.setState({
        modo: 'list', listado: data, loading: false
      }));
    } else { // Error de petición
      console.error(`${res.status} - ${res.statusText}`);
      this.setState({ loading: false });
    }
  },
  err => { // Error de cliente
    console.error(err);
    this.setState({ loading: false });
  }
);
```

Petición POST

```
this.setState({ loading: true });
fetch(URL_BASE, {
   method: 'POST',
   body: JSON.stringify(this.state.elemento),
   headers: {
      'Content-Type': 'application/json'
   }
}).then(res => {
   if (res.ok) {
      this.cancel();
   } else {
      db.AddErrNotifyCmd(`${res.status} - ${res.statusText}`);
      this.setState({ loading: false });
   }
}).catch(err => {
   db.AddErrNotifyCmd(err);
   this.setState({ loading: false });
})
```


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

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); });
```


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: {}
}
```

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
}
```

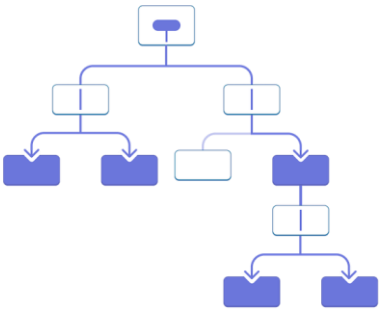

CONTEXT

Introducción

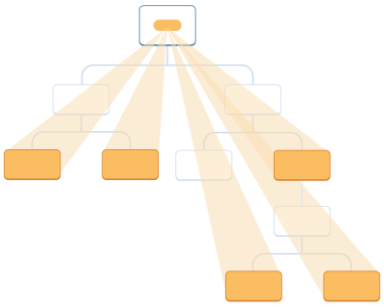
- Context provee una forma de pasar datos a través del árbol de componentes sin tener que pasar props manualmente en cada nivel.
- En una aplicación típica de React, los datos se pasan de arriba hacia abajo (de contenedor a contenido) a través de props, pero esta forma puede resultar incómoda para ciertos tipos de props que son necesarias para muchos componentes dentro de una aplicación, a diferentes niveles y necesitan ser propagadas a través de intermediarios que no las utilizan. Context proporciona una forma de compartir valores como estos entre componentes sin tener que pasar explícitamente una prop a través de cada nivel del árbol.
- Context está diseñado para compartir datos que pueden considerarse “globales” para un árbol de componentes en React, como pueden ser el usuario autenticado actual, la tabla de rutas, el tema seleccionado o el idioma preferido. Context se usa principalmente cuando algunos datos tienen que ser accesibles por muchos componentes en diferentes niveles de anidamiento pero hay que aplicarlo con moderación porque crea dependencias y hace que la reutilización de componentes sea más difícil.

Patrones

Propagar propiedades



Contexto



Crear el contexto

- React.createContext permite crear un contexto que los componentes pueden proporcionar o leer. Recibe el valor por defecto a utilizar si no hay un contexto disponible.

```
import { createContext } from 'react';
export const defaultAuthContext = {
   isAuth: false,
   authToken: '',
   name: '',
   roles: [],
};
export const AuthContext = createContext(defaultAuthContext);
```

- Se deberá exportar desde un archivo para que los componentes puedan usarlo.

Proporcionar el contexto

- Cada objeto Context viene con un componente Provider de React que permite que los componentes que lo consumen se suscriban a los cambios del contexto.
- El componente Provider acepta una prop value que se pasará a los componentes consumidores esten contenidos en el Provider. El valor puede ser una prop o un state para propagar los cambios. Un Provider puede estar conectado a muchos consumidores. Los Providers pueden estar anidados para sobreescribir los valores superiores dentro del árbol.

```
import { AuthContext, defaultAuthContext } from './utilidades/auth-context';

const [auth, setAuth] = useState(defaultAuthContext);

<AuthContext.Provider value={auth}>
:
</AuthContext.Provider>
```

Consumir el contexto

- A la propiedad contextType en una clase componente se le puede asignar un objeto Context creado por React.createContext(). Al usar esta propiedad se puede consumir el valor actual más cercano de ese Context utilizando this.context y se puede utilizar en cualquiera de los métodos del ciclo de vida, incluida la función de renderizado.

```
export class Cabecera extends React.Component {
  render() {
    let usr = this.context.name;
    :
  }
}
Cabecera.contextType = AuthContext;
```

- Como alternativa al contextType para los componentes funcion, cada objeto Context viene con un componente Consumer que permite que los componentes consuman el contexto. El value será igual a la prop value del Provider.

```
<AuthContext.Consumer>
  {value.name => /* renderiza algo basado en el valor de contexto */}
</AuthContext.Consumer>
```

- El hook useContext permite leer y suscribirse a un contexto desde un componente.

Actualizar el contexto

- A menudo, el contexto debe cambiar a través del tiempo. Para actualizar el contexto, es necesario combinarlo con el estado: se declara una variable de estado en el componente proveedor que pasa como el valor de contexto al proveedor.
- Si se suministra la función de cambio de estado, los consumidores podrán actualizar el valor a través del contexto.

```
import { AuthContext, defaultAuthContext } from './utilidades/auth-context';

const [auth, setAuth] = useState(defaultAuthContext);

<AuthContext.Provider value={{ auth, setAuth }}>
:
```

- Para actualizar el valor a través del contexto:

```
import { AuthContext, defaultAuthContext } from './utilidades/auth-context';

const { auth, setAuth } = useContext(AuthContext);
const login = usr => setAuth(usr);
const logout = () => setAuth(defaultAuthContext);
```

Composición de componentes

- Si solo se desea evitar pasar algunas props a través de muchos niveles, la composición de componentes suele ser una solución más simple que Context.
- Cuando se anida contenido dentro de una etiqueta JSX, el componente recibirá ese contenido en la propiedad children. Así mismo, el valor de las props pueden ser JSX.

```
function Seccion({ avatar, tipo, children }) {
  return (
    <div className={' Seccion Seccion-' + tipo}>
      {avatar} {children}
    </div>
  );
}
<Seccion tipo="Ficha" avatar={(<img className="avatar"
      src="https://i.imgur.com/MK3eW3As.jpg" width={size} height={size} />)}>
   <div className="light">El contenido</div>
</Seccion>
```


HOC: higher-order component

COMPONENTES DE ORDEN SUPERIOR

Introducción

- Los componentes de orden superior (HOC: higher-order component) son una técnica avanzada en React para la reutilización de la lógica de componentes. Los HOCs son un patrón que surge de la naturaleza composicional de React y no una parte de la API de React.
- Un componente de orden superior es una función que recibe un componente, lo envuelve y devuelve un nuevo componente, permitiendo manejar las preocupaciones transversales comunes. Como alternativa se pueden usar mixins del JavaScript para manejar dichas preocupaciones transversales pero la mayoría de las veces causan más problemas de los que resuelven.
- Los HOCs son comunes en bibliotecas de terceros usadas en React, tales como connect en Redux, Router o Relay.

Componente de orden superior

```
export function withErrorBoundary(WrappedComponent) {
  return class extends Component {
    constructor(props) {
      super(props);
      this.state = { hasError: false, error: null, errorInfo: null };
    }
    static getDerivedStateFromError(error) {return { hasError: true }; }
    componentDidCatch(error, info) { 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 <WrappedComponent {...this.props} />;
    }
  }
}
```

Restricciones

- Un HOC no debe modificar el componente de entrada, ni usar herencia para copiar su comportamiento. En su lugar, un HOC compone el componente original al envolverlo en un componente contenedor. Un HOC debe ser una función pura sin efectos colaterales.

```
export const ContadorWithErrorBoundary = withErrorBoundary(Contador)
```

- No deben existir dependencias entre el HOC y el componente envuelto, el HOC debe poder envolver a otros componentes (siempre que cumplan las expectativas) y el componente envuelto podrá ser usado sin envolver.
- Los HOCs añaden funcionalidad a un componente. No deberían alterar de forma drástica su contrato (props), se espera que el componente devuelto por un HOC tenga una interfaz similar al componente envuelto. Los HOCs deben pasar directamente las props al componente envuelto que no tengan relación con su preocupación específica, por lo que la mayoría de los HOCs lo resuelven en el método render.
- Los componentes envueltos cuando son creados por HOC generan nuevos objetos, por lo que hay que tenerlo en cuenta en la detección de cambios.
- Los HOC no reenvían las refs, se deben usar forwardRef.

(Versión 16.8+)

HOOKS

Introducción

- Hooks son una nueva característica en React 16.8 que permiten usar el estado, montaje/desmontaje y otras características en los componentes función sin escribir una clase.
- Los Hooks resuelven una amplia variedad de problemas aparentemente desconectados en React:
  - Es difícil reutilizar la lógica de estado entre componentes: No ofrece una forma de "acoplar" comportamientos re-utilizables a un componente, requiere reestructurar con envoltorios (providers, consumers, componentes de orden superior, render props, …).
  - Los componentes complejos se vuelven difíciles de entender: Con el paso del tiempo los componentes crecen y se convierten en un lío inmanejable de múltiples lógicas de estado y efectos secundarios.
  - Las clases JavaScript confunden tanto a las personas como a las máquinas: JavaScript es orientado a prototipos y su uso de this es muy diferente al de la mayoría de los lenguajes. Las clases no se minimizan muy bien y hacen que la recarga en caliente sea confusa y poco fiable.
  - En un futuro permitirá la compilación anticipada de componentes como Svelte, Angular, Glimmer y otros.

Introducción

- Los Hooks son funciones que permiten "enganchar" el estado de React y el ciclo de vida a los componentes función, no funcionan dentro de los componentes clases.
- Sus nombres siempre empiezan con use: useState, useEffect, …
- Los Hooks agregan características a los componentes función que antes solo estaban disponible en los componentes clases.
- Los Hooks son una alternativa a los componentes clases, son completamente opcionales y no un sustituto de los componentes clases.
- No hay planes para quitar el modelo de clases de React.

Hook de estado

- useState es un Hook (función) que se llama dentro de un componente de función para agregarle un estado local y que React lo mantenga
- useState recibe como argumento el estado inicial y devuelve un par variablefunción: el valor de estado actual (referencia) y la función que permite actualizarlo.

```
function Coordenadas() {
  let [punto, setPunto] = useState({x: 0, y: 0});
```

- La sintaxis de desestructuración de un array permite dar diferentes nombres a las variables y funciones de estado que declaramos llamando a useState.

Hook de estado

- En un componente de función no existe this por lo que no se puede asignar o leer this.state, se usan directamente las variables sin this.
- La función de actualización de estado recibe el nuevo valor de estado y maneja la complejidad del cambio de estado y re-renderizados. Se puede llamar desde un controlador de eventos o desde cualquier otro lugar.
- Se puede usar el Hook de estado más de una vez en un mismo componente.
- Se pueden crear Hooks personalizados para reutilizar el comportamiento con estado entre diferentes componentes.

Hook de estado

```
export default function Contador(props) {
  const [contador, setContador] = useState(props.init ?? 0);
  return (
    <div>
      <h1 data-testid="pantalla">{contador}</h1>
      <p>
        <input type="button" value="-"
          onClick={() => setContador(contador - 1)} />
        <input type="button" value="+"
          onClick={() => setContador(contador + 1)} />
      </p>
    </div>
  );
}
```


Hook de efecto

- Las operaciones de recuperación de datos, suscripciones o modificación manual del DOM realizadas en los componentes React se denominan “efectos secundarios” (o “efectos” para abreviar) porque pueden afectar a otros componentes y no se pueden hacer durante el renderizado.
- El Hook de efecto, useEffect, agrega la capacidad de realizar efectos secundarios desde un componente función. Tiene el mismo propósito que componentDidMount, componentDidUpdate y componentWillUnmount en las clases React, pero unificadas en una sola API.
- Los efectos se declaran dentro del componente función para que tengan acceso a sus props y estado. De forma predeterminada, React ejecuta los efectos después de cada renderizado, incluyendo el primer renderizado.
- Opcionalmente, los efectos pueden especificar cómo “limpiar” o “sanear” (desmontar) después de ellos devolviendo una función.

Hook de efecto

- useEffect es un procedimiento que recibe como argumento la función a ejecutara en componentDidMount y componentDidUpdate. Si la función pasada devuelve una función, la función devuelta se utilizará en componentWillUnmount.
- Se pueden usar más de un efecto en un componente y separar conceptos, permiten organizar efectos secundarios en un componente según qué partes están relacionadas (como agregar y eliminar una suscripción), en lugar de forzar una división basada en métodos del ciclo de vida.
- En algunos casos, sanear o aplicar el efecto después de cada renderizado puede crear problemas de rendimiento. Se puede indicar a React que omita aplicar un efecto si ciertos valores no han cambiado entre renderizados pasándole a useEffect como segundo argumento un array con dichos valores. El array debe contener todos los valores del ámbito del componente (como props y estado) que cambien a lo largo del tiempo y que sean usados por el efecto.

Hook de efecto

```
export function Coordenadas(props) {
  const [coordenadas, setCoordenadas] = useState({ latitud: null, longitud: null});
  let watchId = null;
  useEffect(() => {
    watchId = window.navigator.geolocation.watchPosition(pos => {
      setCoordenadas({latitud: pos.coords.latitude, longitud: pos.coords.longitude })
    });
    return () => window.navigator.geolocation.clearWatch(watchId);
  }, [coordenadas]);
  return coordenadas.latitud == null ? (<div>Cargando</div>) : (
     <div>
       <h1>Coordenadas</h1>
       <h2>Latitud: {coordenadas.latitud}</h2>
       <h2>Longitud: {coordenadas.longitud}</h2>
     </div>
    );
}
```

Otros Hooks

- **useContext**: Acepta un objeto de contexto (el valor devuelto de React.createContext) y devuelve el valor de contexto actual. Un componente que llama a useContext siempre se volverá a renderizar cuando el valor del contexto cambie.

```
const value = useContext(MyContext);
```

- **useReducer**: Una alternativa a useState. Acepta un reducer de tipo (state, action) => newState y devuelve el estado actual emparejado con un método dispatch (similar a Redux). Es mas indicado cuando se tiene una lógica compleja que involucra múltiples subvalores o cuando el próximo estado depende del anterior.
- **useCallback**: Devuelve una versión memorizada del callback que solo cambia si una de las dependencias ha cambiado. Esto es útil cuando se transfieren callbacks a componentes hijos optimizados que dependen de la igualdad de referencia para evitar renders innecesarias (similar a shouldComponentUpdate).

```
const memoizedCallback = useCallback(() => { fn(a, b); }, [a, b]);
```


Otros Hooks

- **useMemo**: Devuelve un valor memorizado. Recibe una función de “crear” y un array de dependencias, solo se volverá a calcular el valor memorizado cuando una de las dependencias haya cambiado. Esta optimización ayuda a evitar cálculos costosos en cada render.

```
const memoizedTotal = useMemo(() => a * b, [a, b]);
```

- **useRef**: Devuelve un objeto ref (campos referencia) mutable cuya propiedad .current se inicializa con el argumento pasado (initialValue). El objeto devuelto se mantendrá persistente durante la vida completa del componente.

```
const txtNombre = useRef(null);
:
<input ref={txtNombre} type="text" />
```

- **useImperativeHandle**: personaliza el valor de instancia que se expone a los componentes padres cuando se usa ref. El código imperativo que usa refs debe evitarse en la mayoría de los casos. useImperativeHandle debe usarse con forwardRef.

Otros Hooks

- **useLayoutEffect**: La firma es idéntica a useEffect, pero se dispara de forma síncrona después de todas las mutaciones de DOM. Se utiliza para leer el diseño del DOM y volver a renderizar de forma sincrónica. Las actualizaciones programadas dentro de useLayoutEffect se vaciarán sincrónicamente, antes de que el navegador tenga la oportunidad de pintar. Se utiliza como alternativa al useEffect cuando este da problemas.
- **useDebugValue**: Puede usarse para mostrar un mensaje en React DevTools para los Hooks personalizados.

```
useDebugValue(isOnline ? 'Online' : 'Offline');
```


Hooks personalizados

- A veces, se quiere reutilizar alguna lógica de estado entre componentes. Tradicionalmente, había dos soluciones populares para este problema: componente de orden superior y render props. Los Hooks personalizados permiten hacer esto, pero sin agregar más componentes al árbol.
- Un Hook personalizado es una función de JavaScript cuyo nombre comienza con ”use” (convención de nomenclatura), que puede llamar a otros Hooks y debe ser llamada dentro de un componente función. Los Hooks personalizados son más una convención que una funcionalidad.
- El estado de cada componente es completamente independiente. Los Hooks son una forma de reutilizar la lógica de estado, no el estado en sí. De hecho, cada llamada a un Hook tiene un estado completamente aislado, por lo que incluso se puede usar el mismo Hook personalizado dos veces en un componente.

Hooks personalizados

```
export function useCoordenadas() {
  const [coordenadas, setCoordenadas] = useState({ latitud: null, longitud: null});
  const watchId = useRef(0);
  useEffect(() => {
    watchId.current = window.navigator.geolocation.watchPosition(pos => {
      setCoordenadas({
        latitud: pos.coords.latitude,
        longitud: pos.coords.longitude
      })
    });
    return () => window.navigator.geolocation.clearWatch(watchId.current);
  });
  return coordenadas;
}
```


React Router

https://reactrouter.com/

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 de pueden implementar utilizando librerías de terceros. El React Router es una de las mas 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';

Componentes principales

- BrowserRouter (Router)
  - 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 los componente Routes.
- Routes
  - Este componente, causa que solo se renderice el Route que coincida con la locación o URL actual (elegirá la combinación más específica).
  - En versiones anteriores, se tenía que ordenar las rutas de cierta manera para obtener la correcta para renderizar cuando varias rutas coincidían con una URL ambigua.
- 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.
- Link
  - Permite al usuario navegar a otra página haciendo clic o tocándola.

Componentes principales

```
<BrowserRouter>
  <nav>
    <Link to='/'>Home</Link> | <Link to='/teams'>Teams</Link> | <Link to='/about'>About</Link>
  </nav>
  <Routes>
    <Route path='/' element={<App />}>
      <Route index element={<Home />} />
      <Route path='teams' element={<Teams />}>
        <Route path=':teamId' element={<Team />} />
        <Route path='new' element={<NewTeamForm />} />
        <Route index element={<LeagueStandings />} />
      </Route>
      <Route path='about' element={<About />} />
      <Route path='*' element={<NotFound />} />
    </Route>
  </Routes>
</BrowserRouter>
```


Route

- Permite definir las diferentes rutas de la aplicación y asociarla a un componente.
  - path: la ruta en la que se renderizará el componente en forma de cadena de texto
  - element: 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).
  - caseSensitive: un booleano para definir si queremos distinguir entre minúsculas y mayúsculas, y tomar esto en cuenta para renderizar un componente. Ej: /index !== /Index
- Cuando ninguna otra ruta coincida con la URL, se puede representar una ruta "no encontrada" usando path="*".
- La rutas pueden contar con segmentos dinámicos asignados a parámetros:
  `<Route path='/producto/:id'`
- Para recuperar los parámetros de la ruta:
  `const { id } = useParams()`

Rutas anidas

- Las rutas se pueden anidar unas dentro de otras, sus path y sus element también se anidarán.

```
<Routes>
    <Route path="invoices" element={<Invoices />}>
      <Route path=":invoiceId" element={<Invoice />} />
      <Route path="sent" element={<SentInvoices />} />
    </Route>
</Routes>
```

- Esta configuración de ruta define tres rutas:
  - "/invoices"
  - "/invoices/sent"
  - "/invoices/:invoiceId"
- Los segmentos de URL anidados se asignan a árboles de componentes anidados. Esto es perfecto para crear una interfaz de usuario que tenga navegación persistente en diseños con una sección interna que cambia con la URL.

Rutas anidas

- Se muestra el elemento padre y la ruta secundaria coincidente se represente con <Outlet>. Si se omite, el element predeterminado para la ruta padre es <Outlet>.

```
function Invoices() {
  return (
    <div>
      <h1>Invoices</h1>
      <Outlet />
    </div>
  );
}
```

- Las rutas sin path marcadas con index se pueden considerar como "rutas secundarias predeterminadas". Cuando una ruta principal tiene varios elementos secundarios, pero la URL se encuentra justo en la ruta principal:

```
<Routes>
    <Route path="invoices">
      <Route index element={<InvoicesList />} />
      <Route path=":invoiceId" element={<InvoiceView />} />
    </Route>
</Routes>
```

Rutas anidas

- A menudo, las rutas principales administran el estado u otros valores que desean compartir con las rutas secundarias. El hook useOutletContext permite recuperar el contexto.

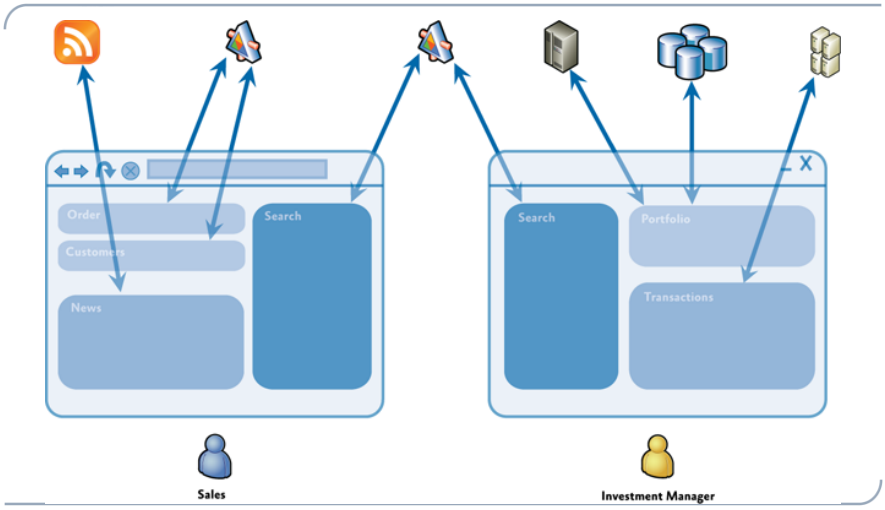
```
function Parent() {
  const [count, setCount] = React.useState(0);
  return <Outlet context={[count, setCount]} />;
}
function Child() {
  const [count, setCount] = useOutletContext();
  const increment = () => setCount((c) => c + 1);
  return <button onClick={increment}>{count}</button>;
}
```


Routes en componentes anidados

- Se puede representar un elemento <Routes> en cualquier lugar donde se necesite, incluso en un componentes anidado y en otro archivo. Estos funcionarán igual que cualquier otro anidamiento, excepto que se construirán automáticamente en la ruta que representó el componente. Si hace esto, hay que poner un * al final de la ruta principal o, de lo contrario, las rutas anidada no coincidirá con la ruta principal y sus descendientes <Routes> nunca aparecerán.

```
function App() {
  return (
    <Routes>
      <Route path="/" element={<Home />} />
      <Route path="dashboard/*" element={<Dashboard />} />
    </Routes>
  );
}
function Dashboard() {
  return (
     <Routes>
       <Route path="/" element={<DashboardGraphs />} />
       <Route path="invoices" element={<InvoiceList />} />
     </Routes>
  );
}
```

Patrón Composite View



Conjuntos de Routes

- Aunque solo debe tener uno <Router> (<BrowserRouter>) en una aplicación, se puede tener tantos <Routes> como y donde se necesiten. Cada elemento <Routes> opera independientemente de los demás y elige una ruta secundaria para representar.

```
<div>
    <Sidebar>
     <Routes>
       <Route path="/" element={<MainNav />} />
       <Route path="dashboard" element={<DashboardNav />} />
     </Routes>
    </Sidebar>
    <MainContent>
     <Routes>
       <Route path="/" element={<Home />}>
         <Route path="about" element={<About />} />
         <Route path="support" element={<Support />} />
       </Route>
       <Route path="dashboard" element={<Dashboard />}>
         <Route path="invoices" element={<Invoices />} />
         <Route path="team" element={<Team />} />
       </Route>
     </Routes>
    </MainContent>
</div>
```

Link

- El componente Link crea un hipervínculo que permite navegar por la aplicación sin las implicaciones de usar directamente la etiqueta <a href= />.
- Dispone de las siguientes propiedades:
  - to: una ruta en formato cadena. Esta será la localización 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 localización a la historia, este reemplaza la localización actual.
- Deben estar anidados dentro del BrowserRouter que contiene las rutas.
- Un valor relativo <Link to> (que no comienza con /) se resuelve en relación con la ruta principal, lo que significa que se basa en la ruta de URL que coincidió con la ruta que representó ese <Link>. Puede contener con .. enlaces a rutas más arriba en la jerarquía.
- La versión especial NavLink de Link agrega atributos de estilo (activeClassName, activeStyle, exact, strict) al elemento representado cuando coincida con la URL actual.

Navegación

- El hook useNavigate devuelve una función que permite navegar mediante programación. La función tiene dos firmas:
  - Pasar un valor To (del mismo tipo que <Link to>) con un segundo argumento opcional { replace, state }
  - Pasar el delta al que desea ir en la pila de historial (-1 es equivalente a presionar Atrás).

```
function Invoices() {
  const navigate = useNavigate();
  return (
    <div>
      <NewInvoiceForm  onSubmit={async (event) => {
          let newInvoice = await createInvoice(event.target);
          navigate('/invoices/${newInvoice.id});
        }} />
    </div>
  );
}
```

Navegación

- Para cuando no se pueda utilizar useNavigate, el elemento <Navigate> cambia la ubicación actual cuando se representa. Es un contenedor de componentes useNavigate y acepta todos los mismos argumentos como props.

```
<div>
    {error && <p>{error.message}</p>}
    {user && (
      <Navigate to="/dashboard" replace={true} />
    )}
    <form  onSubmit={(event) => this.handleSubmit(event)} >
      <input type="text" name="username" />
      <input type="password" name="password" />
    </form>
</div>
```

- El elemento <Redirect> de v5 ya no se admite como parte de la configuración de las rutas (dentro de un <Routes>). Si se necesita redirigir inmediatamente, se puede utiliza un elemento <Navigate> en el componente de la ruta.

```
<Route path='personas' element={<Navigate to='/contactos' />} />
```


Hooks

- Hay algunas API de bajo nivel que se usan internamente que también pueden resultar útiles al crear tus propias interfaces de navegación.
  - useResolvedPath: resuelve una ruta relativa contra la ubicación actual
  - useHref: resuelve una ruta relativa adecuada para su uso como <a href>
  - useLocation y useNavigationType: estos describen la ubicación actual y cómo llegamos allí
  - useLinkClickHandler: devuelve un controlador de eventos para la navegación al crear un <Link> personalizado en react-router-dom
  - useLinkPressHandler: devuelve un controlador de eventos para la navegación al crear un <Link> personalizado en react-router-native
  - resolvePath: resuelve una ruta relativa contra un nombre de ruta de URL dado
  - useRoutes: es el equivalente funcional de <Routes>, pero usa objetos de JavaScript en lugar de elementos <Route> para definir las rutas.

https://redux.js.org/

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.

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.

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.

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.

Instalación

- Para instalar la versión estable del nucleo:
  - 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
- Redux Toolkit incluye el núcleo de Redux, así como otros paquetes clave que consideramos esenciales para crear aplicaciones de Redux (como Redux Thunk y Reselect).
  - npm install @reduxjs/toolkit
- 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.
- La forma recomendada de iniciar nuevas aplicaciones con React y Redux es mediante el uso de la plantilla oficial:
  - npx create-react-app my-app --template redux

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 mantiene el estado de la aplicación y es el responsable de permitir su consulta, modificación mediante acciones y notificaciones de los cambios.

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.

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 });
```


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 dispach(), toma el estado anterior y la acción suministrada para devolver el 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.

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 la 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 profunda, 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

Reducers

```
const initialState = { modo: 'add', listado: [], elemento: {} };
function myAppReducer(state = initialState, action) {
  switch (action.type) {
    case SET_MODO:
      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({}, 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
  }
}
```

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
})
```


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.

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()
```


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.

Ecosistema

- Redux es una biblioteca pequeña, pero sus contratos y API se eligen cuidadosamente para generar un ecosistema de herramientas y extensiones, y la comunidad ha creado una amplia variedad de complementos, bibliotecas y herramientas útiles.
- No se necesita usar ninguno de estos complementos para usar Redux, pero pueden ayudar a que sea más fácil implementar funciones y resolver problemas en su aplicación.
- Están disponibles las integraciones con React, Angular, Ember, Glimmer, Polymer, …
- Así mismo existen múltiples complementos que aplican, especializan o simplifican diferentes aspectos: Reductores, Stores, Datos inmutables, Efectos secundarios, Software intermedio, Entidades y Colecciones, Estado del componente y encapsulación, Enrutamiento, Formularios, Utilidades
- Se puede complementa las Redux DevTools con herramientas de desarrollo: Depuradores y visores, Monitores, Registros, Detección de mutaciones, Pruebas

Uso con React

- Redux no tiene relación alguna con React salvo la implementación del patrón Flux.
- React Redux, mantenidos por el equipo de 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.

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.

connect()

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

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

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>  <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)
```


Hooks

- useSelector: permite extraer datos del estado del store utilizando una función.

```
export const CounterComponent = () => {
 const counter = useSelector((state) => state.counter)
 return <div>{counter}</div>
}
```

- useDispatch: devuelve una referencia a la función dispatch del store.

```
export const CounterComponent = ({ value }) => {
 const dispatch = useDispatch()
 return (
  <div>
   <span>{value}</span>
   <button onClick={() => dispatch({ type: 'increment-counter' })}>
    Increment counter
   </button>
  </div>
 )
}
```

- useStore: devuelve una referencia al propio store.

```
const store = useStore()
```

<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'));
```


Redux Toolkit

- El paquete Redux Toolkit ("RTK" para abreviar) está destinado a ser la forma estándar de escribir la lógica de Redux. Originalmente se creó para ayudar a abordar tres preocupaciones comunes sobre Redux:
  - Configurar un almacenamiento Redux es demasiado complicado
  - Hay que agregar muchos paquetes adicionales para que Redux haga algo útil
  - Redux requiere demasiado código repetitivo
- RTK incluye utilidades que ayudan a simplificar muchos casos de uso comunes, incluida la configuración del almacenamiento, la creación de reductores y la escritura de una lógica de actualización inmutable e incluso la creación de "porciones" completas de estado a la vez .

APIs

- configureStore(): ajusta createStore para proporcionar opciones de configuración simplificadas y buenos valores predeterminados.
  - Se puede combinar automáticamente las porciones de reductores, agregar cualquier middleware, incluye redux-thunk, uso de Redux DevTools, …
- createReducer(): Permite proporcionar una tabla de búsqueda de tipos de acción para funciones de reducción de casos, en lugar de escribir instrucciones de cambio.
  - Además, utiliza automáticamente la biblioteca immer para permitir escribir actualizaciones inmutables más simples como código mutable normal, como state.value += state.delta.
- createAction(): Genera una función creadora de acción para la cadena de tipo de acción dada.
  - La función misma se ha definido con toString(), por lo que se puede usar en lugar de la constante de tipo.
- createSlice(): Combina createReducer() + createAction().
  - Acepta un objeto de funciones reductoras, un nombre de segmento y un valor de estado inicial, y genera automáticamente un reductor de segmento con los creadores de acción y los tipos de acción correspondientes.

createSlice

```
import { createSlice } from '@reduxjs/toolkit';
const initialState = { value: 0, delta: 1 };

export const contadorSlice = createSlice({
   name: 'contador',
   initialState,
   reducers: {
     increment: (state) => { state.value += state.delta; },
     decrement: (state) => { state.value -= state.delta; },
     init: (state, action) => {
       state.value = action.payload.value;
       state.delta = action.payload.delta;
     },
   },
 });
```

createSlice

```
export const { increment, decrement, init } = contadorSlice.actions;

export const selectCount = (state) => state.contador.value;

export const initCount = (value, delta = 1) => (dispatch, getState) => {
   dispatch(init({ value, delta }));
};

export const defaultInitCount = () => (dispatch, getState) => {
   dispatch(init(initialState));
};

export default contadorSlice.reducer;
```


configureStore

```
import { configureStore } from '@reduxjs/toolkit';
import authReducer from './authSlice';
import contadorReducer from './contadorSlice';
import notificationReducer from './notificationSlice';

export const store = configureStore({
  reducer: {
    contador: contadorReducer,
    notification: notificationReducer,
    auth: authReducer
  },
});
```

componente

```
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { selectCount, initCount, increment, decrement, } from './contadorSlice'

export default function Contador() {
  const count = useSelector(selectCount);
  const dispatch = useDispatch();

  return (
    <div>
      <h1>{count}</h1>
      <p>
        <input type="button" value="-" onClick={() => dispatch(decrement())} />
        <input type="button" value="init" onClick={() => dispatch(initCount(10, 2))} />
        <input type="button" value="+" onClick={() => dispatch(increment())} />
      </p>
    </div>
  )
}
```