

# REACT

# ¿POR QUÉ REACT?

# POPULARIDAD

- Lo demanda el mercado
- Gran comunidad
  - Stackoverflow
  - Librerías de terceros

- Muchas empresas
  - Facebook, WhatsApp, Instagram
  - Uber
  - Khan Academy
  - Airbnb
  - Dropbox
  - Netflix
  - PayPal

# A NIVEL TÉCNICO

- Excelente rendimiento: Virtual DOM
- Componentes reusables
  - Y existen muchos componentes
- Curva de aprendizaje pequeña
  - Fácil escribir con JSX
  - No salimos del estándar ES2015 y más allá

# CONCEPTO DE REACT

- Se considera la V en un modelo MVC
- Cada vista está constituida por componentes
  - Header
  - Footer
  - LoginForm
  - ...
- Cada componente se puede reutilizar en:
  - Otra vista
  - Otra aplicación

# INSTALACIÓN DE PAQUETES

- Trabajaremos con el ecosistema node:
  - Instalaremos node, npm a través de nvm
  - Se puede usar [yarn](#) como gestor de paquetes en vez de npm

# VENTAJAS DE YARN

- Instantánea de versiones mediante *yarn.lock*
- Instalar dependencias en paralelo
  - Mayor rapidez
- Se empezó a usar mucho en el ecosistema React
  - React viene de Facebook y yarn también
- Con npm v6 no se aprecian ventajas

```
npm i -g yarn  
yarn --version
```



# INTRODUCCIÓN

## REACT Y JSX

# SCRIPTS DE REACT

- No usaremos webpack
- CDN relacionado con los paquetes de npm

```
unpkg.com/:package@:version/:file
```

- Utilizaremos CDN

```
<script crossorigin src="https://unpkg.com/react@16/umd/react.js">  
<script crossorigin src="https://unpkg.com/react-dom@16/umd/react-dom.js">
```

# ESQUELETO

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="UTF-8">
  <title>Código inicial</title>
</head>

<body>
  <div id="app"></div>
  <script crossorigin src="https://unpkg.com/react@16/umd/react">
  <script crossorigin src="https://unpkg.com/react-dom@16/umd/">
  <script src="/scripts/app.js"></script>
</body>
```

# ESTRUCTURA HTML

- Cargamos los siguientes scripts:
  - React y React-dom
- Generarán dos variables en el objeto Window:
  - React y ReactDOM
- El segundo script está asociado a la web
  - No es necesario si trabajamos con VR ([react-360](#))
  - No es necesario con móviles ([react native](#))

# LINTER

- Para usar eslint, utilizaremos node:

```
npm init  
npm i -D eslint  
npx eslint --init
```

- Yo elijo standard

# BABEL

- Es un compilador de JavaScript
  - Escribimos código JavaScript de última generación
  - Babel lo traduce a código ES5
- Prueba a escribir código [en su web](#) y observa su traducción

# HOLA MUNDO MEDIANTE JS

```
console.log('App running')

const p = document.createElement('p') // Creamos un elemento p
const text = document.createTextNode('Hola Mundo') // Creamos
p.appendChild(text) // Añadimos el texto al párrafo>
const appRoot = document.getElementById('app')
appRoot.appendChild(p)
```

# ¿QUÉ TENEMOS?

- Inyección de código html a partir de creación y relación de nodos
- Puede haber muchos nodos y se puede volver complejo



# ¿QUÉ NOS GUSTARÍA?

- Elegir un nodo (app)
- Inyectar directamente el código dentro del nodo como si fuera código *html*
- ¡JSX al rescate!

# LENGUAJE JSX

- JSX denota una ampliación de JavaScript a JavaScript con XML
  - Facilita el uso de templates

```
/* global ReactDOM */  
console.log('Aplicación en ejecución')  
  
var template = <p>¡Hola Mundo!</p>  
var appRoot = document.getElementById('app')  
ReactDOM.render(template, appRoot)
```

# NECESITAMOS BABEL

- Si ejecutamos en un navegador vemos que la definición del template da error:

```
var template = <p>¡Hola Mundo!</p>  
Unexpected token <
```

- No es código JavaScript
- Es una extensión (JSX) que nosotros debemos compilar previamente a JS para que funcione
- De eso se encarga Babel con el preset adecuado (react)

# USAR BABEL

- Traduce el código anterior mediante Babel y comprueba que funcione en el navegador:

```
/* global ReactDOM */  
console.log('Aplicación en ejecución')  
  
var template = <p>iHola Mundo!</p>  
var appRoot = document.getElementById('app')  
ReactDOM.render(template, appRoot)
```

# RESULTADO BABEL

- Este es el código que deberíamos poner en el fichero *app.js*:

```
/* global React, ReactDOM */
'use strict'

console.log('Aplicación en ejecución')

var template = React.createElement('p', null, '\xA1Hola Mundo!')
var appRoot = document.getElementById('app')
ReactDOM.render(template, appRoot)
```

# USO AUTOMÁTICO DE BABEL

- Objetivos:
  - Escribir JSX y que se traduzca a código JavaScript
  - Escribir ES2015 y posterior y que se traduzca a ES5
  - ¡Qué sea automático!

# PLUGINS DE BABEL

- Babel está compuesto por un conjunto de plugins
- Cada plugin se encarga de hacer una traducción determinada
- **Plugin de funciones flecha**
- Se puede volver un poco locura seleccionar los plugins necesarios para traducir nuestro código a ES5
  - Utilizaremos **presets**
    - Son colecciones de plugins agrupados de forma lógica

# PRESETS DE BABEL

- Conjunto de plugins que hacen que Babel nos de una funcionalidad determinada.
- Nos facilitan la selección de los plugins
- Utilizaremos
  - **react**: que da acceso a JSX
  - **env**: da acceso a ES2015, ES2016 y ES2017



# ORGANIZACIÓN DEL PROYECTO

- Creamos carpeta *public* para el proyecto final
  - Fichero *index.html* en la carpeta *public*
  - Solo modificaremos los *js*
- Creamos carpeta *public/scripts* para los *js* en ES5
- Nuestros fuentes en el directorio *src*
  - Fichero *app.js*

# DEPENDENCIAS

- babel para compilación a ES5
- [live-server](#) para visualización

```
npm i -D babel-cli  
npm i -D babel-preset-react babel-preset  
npm i -D live-server
```

# CONFIGURACIÓN DEL WORKFLOW

- livenesserver coge los cambios y los muestra de forma automática en el navegador
- babel observa los cambios y compila para que livenesserver funcione en tiempo real (--watch o -w)
- npm scripts en *package.json*:

```
"build": "babel -w src/app.js --out-file public/scripts/app.js",  
"start": "live-server ./public",  
"start-dev": "npm run build & npm start"
```

# EJECUCIÓN Y PARADA DEL PROYECTO

- Mediante dos comandos:

```
npm run build  
npm start
```

- De una sola vez:

```
npm run start-dev
```

# EXPLORAR JSX

- Instalamos Babel JavaScript
  - Mejora colores sintáxis
- Escribiremos el código con paréntesis y salto de línea para mejor lectura:

```
const template = (  
  <div>  
    <h1>Lista de tareas</h1>  
    <p>Pendientes:</p>  
    <ol>  
      <li>Planchar</li>  
      <li>Comprar leche</li>  
    </ol>  
  </div>  
)
```



# USO DE EXPRESIONES

```
const user = {  
  name: 'Pepe',  
  age: 30,  
  city: 'Zaragoza'  
}  
const template = (  
  <div>  
    <h1>{user.name}</h1>  
    <p>Age: {user.age}</p>  
    <p>City: {user.city}</p>  
  </div>  
)
```

# USO DE CONDICIONALES

```
const user = {
  name: 'Pepe',
  age: 30,
  city: 'Zaragoza'
}
function getCity (city) {
  if (city) {
    return <p>Ciudad: {city}</p>
  }
}
const template = (
  <div>
    <h1>{user.name ? user.name : 'Desconocido'}</h1>
    {user.age && user.age >= 18 && <p>Edad: {user.age}</p>}
    {getCity(user.city)}
  </div>
)
```



# CREAR UN HEADER

- Crea un template para header con las siguientes propiedades:
  - title: *Añadir cervezas*
  - subtitle: *Usa el formulario para añadir tus cervezas preferidas*
- Crea un template que muestre:
  - El título mediante etiqueta h1
  - El subtítulo en caso de existir, mediante etiqueta de párrafo

# SOLUCIÓN HEADER

- ¡Ojo, en principio se necesita una etiqueta que haga de wrapper!

```
/* global ReactDOM */

const header = {
  title: 'Añadir cervezas',
  subtitle: 'Usa el formulario para añadir tus cervezas preferidas'
}

const template = (
  <header>
    <h1>{header.title}</h1>
    {header.subtitle ? <p>{header.subtitle}</p> : ''}
  </header>
)

var appRoot = document.getElementById('app')
```

# VAR, CONST Y LET

- Redefinir variables
  - Es posible con var (let y const no dejan)
  - La compilación de lo siguiente en babel dará error:

```
let nombre = 'pepe'  
let nombre = 'pepe'
```

# AMBITO DE SCOPE

- var dentro de funciones
- let y const dentro de llaves
  - Por eso definimos firstName fuera del if

```
const fullName = 'Pedro Pérez'
let firstName

if (fullName) {
  firstName = fullName.split(' ')[0]
  console.log(firstName)
}
console.log(firstName)
```

# FUNCIONES TRADICIONALES

- Forma tradicional:

```
function square(x) {  
  return x * x;  
}  
console.log(square(3));
```

- La función podría ser anónima
  - En este caso la guardamos en una variable para poder usarla

```
const square = function (x) {  
  return x * x;  
};  
console.log(square(3));
```

# ARROW FUNCTIONS

- Son funciones anónimas
  - En caso de usarlas posteriormente, hay que asignarlas a una variable
- Los parámetros van entre parentesis
  - Si hay un solo parametro puede ir sin paréntesis
  - Si no hay parámetros hay que poner paréntesis sin argumentos

```
const square = x => {  
  return x * x  
}  
console.log(square(3))
```

- Arrow functions simplificadas:
  - Se quitan las {} y el return va implícito

```
const square = (x) => x * x  
console.log(square(3))
```

- Si se devuelve un json, se deben añadir paréntesis:

```
const square = (x) => ({doble: x * x})
```



# CONVERSIÓN A ARROW FUNCTIONS

- Crea una arrow function que devuelva la primera parte de un nombre completo

```
getFirstName('Pedro Pérez') -> "Pedro"  
// Create regular arrow function  
// Create arrow function using shorthand syntax
```

# SOLUCIÓN ARROW FUNCTION

```
const getFirstName = fullName => fullName.split(' ')[0]  
console.log(getFirstName('Pedro Pérez'))
```

# ARGUMENTS

- No se puede acceder al array de arguments desde las arrow functions:

```
const add = (a, b) => {  
  // console.log(arguments)  
  return a + b  
};  
console.log(add(55, 1, 1001))
```

# THIS

- Se comporta algo diferente en JavaScript respecto a otros lenguajes
  - Su valor depende de como se ha llamado, del contexto de la función
- Usando arrow functions siempre depende del elemento superior
- Conclusiones:
  - Normalmente se deben utilizar arrow functions
  - Pero en ocasiones no

# EJEMPLO USO THIS

- ¿Funcionará el siguiente código?
  - ¿Sabes arreglarlo en caso de que no funcione?

```
const user = {  
  name: 'Pepe',  
  cities: ['Zaragoza', 'Madrid', 'Lérida'],  
  showCities: function () {  
    this.cities.forEach(function (city) {  
      console.log(`${this.name} ha vivido en ${city}\n`)  
    })  
  }  
}
```

# RESULTADO

- La variable `this` no tiene valor en el `console.log`

```
app.js:11 Uncaught TypeError: Cannot read property 'name' of u  
    at app.js:11
```

# POSIBLE SOLUCIÓN:

```
const user = {  
  name: 'Pepe',  
  cities: ['Zaragoza', 'Madrid', 'Lérida'],  
  showCities: function () {  
    const that = this  
    this.cities.forEach(function (city) {  
      console.log(`${that.name} ha vivido en ${city}\n`)  
    })  
  }  
}
```

# SOLUCIÓN CON ARROW FUNCIONS

```
const user = {  
  name: 'Pepe',  
  cities: ['Zaragoza', 'Madrid', 'Lérida'],  
  showCities: function () {  
    this.cities.forEach((city) => {  
      console.log(`${this.name} ha vivido en ${city}\n`)  
    })  
  }  
}
```



# ¿FUNCIONARÁ?

```
const user = {  
  name: 'Pepe',  
  cities: ['Zaragoza', 'Madrid', 'Lérida'],  
  showCities: () => {  
    this.cities.forEach(city => {  
      console.log(`${this.name} ha vivido en ${city}\n`)  
    })  
  }  
}
```

# SOLUCIÓN:

- No funciona porque la función flecha coge *this* de su *parent scope*
- En este caso su *parent scope* es el *global scope* :-)
- Afortunadamente tenemos una nueva sintáxis para los métodos en las clases de ES6:

```
const user = {  
  name: 'Pepe',  
  cities: ['Zaragoza', 'Madrid', 'Lérida'],  
  showCities () {  
    this.cities.forEach(city => {  
      console.log(`${this.name} ha vivido en ${city}\n`)  
    })  
  }  
}
```

# EVENTOS Y ATRIBUTOS

- Algunos se deben reescribir
  - `class` -> `className` (`class` es palabra reservada en JavaScript)
- Se escriben con notación camelCase
- Elementos que podemos usar en React

# EJEMPLO

- Completa el código siguiente para que los botones funcionen:

```
/* global ReactDOM */
console.log('Aplicación en ejecución')

let count = 0
const addOne = () => {
  console.log('addOne')
}

const template = (
  <div>
    <h1>Count: {count}</h1>
    <button onClick={addOne}>+1</button>
    <button onClick={minusOne}>-1</button>
    <button onClick={reset}>reset</button>
  </div>
```

# SOLUCIÓN CÓDIGO ANTERIOR

```
/* global ReactDOM */
console.log('Aplicación en ejecución')

let count = 0
const addOne = () => {
  count++
}
const minusOne = () => {
  count--
}
const reset = () => {
  console.log(count)
  count = 0
}
const template = (
```

# ANÁLISIS CÓDIGO ANTERIOR

- Los botones funcionan
- Sin embargo no se renderiza de nuevo al cambiar el valor de *count*

# IMPLEMENTACIÓN DEL RENDER

- Tenemos que llamar al render de forma manual
  - Al inicio de la aplicación
  - Cada vez que se pulsa un botón
- ReactDOM.render es muy eficiente:
  - Solo actualizará la parte del DOM necesaria
  - Utiliza un DOM virtual para ver diferencias
  - Actualizará solo el h1, no los botones
- ¿Lo intentas?

# SOLUCIÓN

```
/* global ReactDOM */
console.log('Aplicación en ejecución')

let count = 0
const addOne = () => {
  count++
  renderApp()
}
const minusOne = () => {
  count--
  renderApp()
}
const reset = () => {
  count = 0
  renderApp()
}
```



# EVENTOS

- Lista de eventos
- Eventos del formulario

```
/* global ReactDOM */

const app = {
  title: 'Añadir cervezas',
  subtitle: 'Usa el formulario para añadir tus cervezas preferidas',
  cervezas: []
}

const onFormSubmit = e => {
  e.preventDefault()
  const cerveza = e.target.elements['cerveza'].value
  if (cerveza) {
    app.cervezas.push(cerveza)
    e.target.elements.cerveza.value = ''
    renderApp()
  }
}
```

# EJERCICIO: BORRAR LISTA

- Implementar un botón de borrado
- Implementar la acción del botón
  - Inicializar el array de cervezas a vacío.
  - Renderizar de nuevo

```
const template = (  
  ....  
  <button onClick={onRemoveAll}>Remove All</button>  
  ...  
);  
  
const onRemoveAll = () => {  
  app.cervezas = []  
  renderApp()  
}
```

# SOLUCIÓN EJERCICIO

```
/* global ReactDOM */

const app = {
  title: 'Añadir cervezas',
  subtitle: 'Usa el formulario para añadir tus cervezas preferidas',
  cervezas: []
}

const onFormSubmit = e => {
  e.preventDefault()
  const cerveza = e.target.elements['cerveza'].value
  if (cerveza) {
    app.cervezas.push(cerveza)
    e.target.elements.cerveza.value = ''
    renderApp()
  }
}
```

# USO DE ARRAYS CON JSX

- Basta con poner en el template algo como:

```
{  
  ["cerveza1", "cerveza2", "cerveza3"]  
}
```

- JSX lo sabe pintar
- Aunque normalmente necesitaremos añadir marcas de párrafo:

```
{  
  [<p>cerveza1</p>, <p>cerveza2</p>, <p>cerveza3</p>]  
}
```

- Observa el warning: *Each child in an array or iterator should have a unique key prop...*
  - NO tenemos comments de react
  - Añadimos el key que luego no se visualiza pero lo usa internamente para chequear los cambios del DOM

```
<ol>
  {app.cervezas.map(cerveza => (
    <li key={cerveza}>{cerveza}</li>
  ))}
</ol>
```

# EJERCICIO

- Añadimos más campos a las cervezas y gestionamos un array de objetos
- Para mostrarlas en la lista ponemos solo el nombre, pero creamos botones que permitan ver el resto
- Puedes utilizar las siguientes slides de ayuda
  - Como hacer visible un elemento
  - Como leer los datos de un formulario y convertirlos a un json

# AYUDA VISIBILIDAD FORMULARIO

```
let visibility = false

const toggleVisibility = () => {
  visibility = !visibility
  render()
}

const render = () => {
  const template = (
    <div>
      <h1>Visibility Toggle</h1>
      <button onClick={toggleVisibility}>
        {visibility ? "Ocultar detalles" : "Ver detalles"}
      </button>
      {visibility && (
```

# AYUDA: FORMULARIO A JSON

- Los datos se recogen en una colección (formulario)

```
const elementsForm = e.target
```

- Para obtener los datos en JSON:
  - Se convierte la colección a un array
  - Se ejecuta el método `reduce` del array y cada elemento del array se convierte a un campo del objeto final
  - Podemos utilizar *Array.from* o *call* que hace una conversión de tipos implícita



```
const formToJSON = elements =>
[].reduce.call(
  elements,
  (data, element) => {
    if (element.value) data[element.name] = element.value
    return data
  },
  {}
)
```

```
const formToJSON = elements => {
  const newElements = Array.from(elements)
  return newElements.reduce((data, element) => {
    if (element.value) data[element.name] = element.value
    return data
  }, {})
```

# CLASES ES6 Y REACT COMPONENTS

# CLASES ES6

- Creamos la clase persona:
  - Campo nombre, por defecto anónimo
  - Método saludar: *¡Hola soy xxxx!*
- Instanciamos para comprobar

```
class Persona {  
  constructor (name = 'Anónimo') {  
    this.name = name  
  }  
  saludar () {  
    return `Hola, soy ${this.name}`  
  }  
}  
  
const pepe = new Persona('Pepe')  
console.log(pepe.saludar())  
  
const desconocido = new Persona()  
console.log(desconocido.saludar())
```

# EJERCICIO CLASES

- Añadimos el campo *edad*, por defecto 0
- Añadimos el método *presentarse* donde diga nombre y edad

# SOLUCIÓN CLASE

```
class Persona {  
  constructor (nombre = 'Anónimo', edad = 0) {  
    this.nombre = nombre  
    this.edad = edad  
  }  
  saludar () {  
    return `Hola, soy ${this.nombre}`  
  }  
  presentarse () {  
    return `Me llamo ${this.nombre} y tengo ${this.edad} años`  
  }  
}  
  
const pepe = new Persona('Pepe', 26)  
console.log(pepe.saludar())
```

# SUBCLASES

- Creamos la clase Estudiante que tiene los siguientes atributos:
  - Nombre
  - Edad
  - Grado (Informática, Inglés...)

```
class Estudiante extends Persona {  
  constructor (nombre, edad, grado) {  
    super(nombre, edad)  
    this.grado = grado  
  }  
  tieneGrado () {  
    return !!this.grado  
  }  
  presentarse () {  
    let presentación = super.presentarse()  
  
    if (this.tieneGrado()) {  
      presentación += ` He estudiado la carrera de ${this.grado}`  
    }  
  }  
}
```



# EJERCICIO SUBCLASES

- Crea la clase viajero según los requerimientos siguientes:
  - Atributos:
    - Nombre
    - Edad
    - Ciudad de origen
  - Métodos:
    - Saludar, donde diga la ciudad de donde procede.

# SOLUCIÓN: CLASE VIAJERO

```
class Viajero extends Persona {  
  constructor(nombre, edad, ciudadDeOrigen) {  
    super(nombre, edad)  
    this.ciudadDeOrigen = ciudadDeOrigen  
  }  
  saludar() {  
    let saludo = super.saludar()  
  
    if (this.ciudadDeOrigen) {  
      saludo += ` Procedo de ${this.ciudadDeOrigen}.`  
    }  
  
    return saludo;  
  }  
}
```

# CREATE-REACT-APP

- Nos evita tener que configurar Webpack o Babel
- ¿Qué incluye?
  - Soporte a React, JSX, ES6, TypeScript y Flow .
  - Soporte a E6 y superior mediante Babel
  - Prefijos CSS automáticos
  - Test runner con soporte de informe de coverage
  - Servidor de desarrollo en vivo
  - Webpack configurado para empaquetar JS, CSS e imágenes para producción

# CREAR UNA NUEVA APLICACIÓN

```
npx create-react-app cervezas  
cd my-app  
npm start # ejecución  
npm test # tests  
npm run build # compilación
```

# CONFIGURAR LINTER

- create-react-app ya viene "predefinida"
  - Opciones de Babel
  - Webpack
  - eslint
  - ....
- Podemos hacer *npm run eject* para ver lo que hay por debajo
  - ¡No hay vuelta atrás!
- Pero nos da la opción de configurar prettier

# PRETTIER

- Instalación:

```
npm i -D prettier
```

- Configuración (fichero prettierrc.json)
  - Por ejemplo:

```
{  
  "tabWidth": 2,  
  "semi": false,  
  "singleQuote": true  
}
```

- Observa autocompletado del editor para personalizar prettier

# VISTAZO A LA APLICACIÓN POR DEFECTO

- Se utilizan imports y exports de ES6
- Se importan CSS e imágenes gracias a Webpack
- Se da estilos a los elementos vía el atributo *className*
- Nuestros componentes extienden de `React.Component`
  - Utilizan el método `render` para representarlos en el navegador (vía JSX)
  - Heredan otros métodos que no aparecen aquí

# COMPILACIÓN

```
npm run build  
npm i -g serve  
serve -s build
```



# PREPARACIÓN INTERFAZ



# COMPONENTES REACT

- Dividiremos nuestra interfaz en varios componentes

```
<CervezasView />  
<Header />  
<Sidebar />  
<Menu />  
<CervezasList />  
<CervezaSnippet />  
....
```

# EXTENSIONES PARA REACT

- Configuramos emmet para que funcione con jsx:

```
"emmet.includeLanguages": {"javascript": "javascriptreact"}
```

- Usaremos una extensión de Visual Code Editor para React
  - [React Standard Style code snippets](#)

# CREAR COMPONENTE HEADER

- Con el disparador *rce* tenemos la plantilla para nuestro primer componente

```
import React, { Component } from 'react'
import Header from './Header'

class App extends Component {
  render() {
    return <Header />
  }
}

export default App
```

# MODIFICACIÓN APP.JS

- Modificamos *app.js* para que cargue el Header:

```
import React, { Component } from 'react'
import Header from './Header'

class App extends Component {
  render() {
    return <Header />
  }
}

export default App
```

# HEADER CONFIGURABLE

- El Header va a ser un componente reusable entre otros sitios de mi web
- Me interesa que el título se pueda cambiar
  - Se envía como una propiedad
  - React permite hacer un checking de las propiedades para el desarrollo
    - Ese código se elimina cuando se hace el build
    - Se deben disparar los snippets mediante *pts*, *ptn*, *ptsr*...

# CÓDIGO HEADER CONFIGURABLE

```
import React, { Component } from 'react'
import PropTypes from 'prop-types'

export class Header extends Component {
  static propTypes = {
    title: PropTypes.string.isRequired
  }

  render() {
    return (
      <header>
        <h1>{this.props.title}</h1>
      </header>
    )
  }
}
```



# LLAMADA AL HEADER CON PROPIEDADES

- Se utilizan atributos "custom"
- Se sigue la sintaxis de html

```
import React, { Component } from 'react'
import Header from './Header'

class App extends Component {
  render() {
    return <Header title="Buscador de cervezas" />
  }
}

export default App
```

# EJERCICIO COMPLETAR HEADER

- Añade la propiedad *subtitle* que, **si existe**, se debe renderizar con un párrafo

# SOLUCIÓN HEADER

```
import React, { Component } from 'react'
import Header from './Header'

class App extends Component {
  render() {
    return (
      <Header
        title="Buscador de cervezas"
        subtitle="Elije la cerveza que más te guste para ver s
      />
    )
  }
}

export default App
```

# APLICAR ESTILOS

- Creamos un fichero Header.css:

```
header {  
  display: flex;  
  flex-direction: column;  
  align-items: center;  
}
```

- Importamos el css dentro de nuestro componente Header:

```
import React, { Component } from 'react'
import PropTypes from 'prop-types'
import './Header.css'

export class Header extends Component {
  static propTypes = {
    title: PropTypes.string.isRequired,
    subtitle: PropTypes.string
  }

  render() {
    return (
      <header className="header">
        <h1>{this.props.title}</h1>
        {this.props.subtitle && <p>{this.props.subtitle}</p>}
      </header>
    )
  }
}
```

# EJERCICIO

- Crea el resto de componentes y estilos para lograr alguna aplicación de *cervezas*
  - breadcrumbs no es necesario
    - Importaremos un componente que interaccione con el backend
  - Los datos se pueden importar del fichero `cervezas.json`

```
wget https://raw.githubusercontent.com/juanda99/curso-node-  
js/
```

# **PROPIEDADES, ESTADO Y CICLO DE VIDA DE LOS COMPONENTES**

# ESTADO DE UN COMPONENTE

- Un componente puede guardar un estado en *this.state*:

```
class Clock extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {date: new Date()};  
  }  
  
  render() {  
    return (  
      <div>  
        <h1>Hello, world!</h1>  
        <h2>It is {this.state.date.toLocaleTimeString()}</h2>  
      </div>  
    );  
  }  
}
```



# MÁS CORTO

- No es necesario llamar al constructor

```
class Clock extends React.Component {  
  
  state = {date: new Date()}  
  
  render() {  
    return (  
      <div>  
        <h1>Hello, world!</h1>  
        <h2>It is {this.state.date.toLocaleTimeString()}</h2>  
      </div>  
    );  
  }  
}
```

# MÉTODOS

- Cada componente tiene unos **métodos asociados a su tiempo de vida**
- El estado se puede modificar con *setState*
- Cada vez que el estado cambia, se vuelve a renderizar el componente
  - Cuando las propiedades cambian también

# EJEMPLO

```
class Clock extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {date: new Date()};  
  }  
  
  componentDidMount() {  
    this.timerID = setInterval(  
      () => this.tick(),  
      1000  
    );  
  }  
  
  componentWillUnmount() {  
    clearInterval(this.timerID);  
  }  
}
```

# BINDING DE MÉTODOS ES2015

```
class Foo extends Component {  
  constructor(props) {  
    super(props);  
    this.handleClick = this.handleClick.bind(this);  
  }  
  handleClick() {  
    console.log('Click happened');  
  }  
  render() {  
    return <button onClick={this.handleClick}>Click Me</button>  
  }  
}
```

# BINDING EN EL RENDER

- Nos ahorramos el constructor

```
class Foo extends Component {  
  handleClick() {  
    console.log('Click happened');  
  }  
  render() {  
    return <button onClick={this.handleClick.bind(this)}>Click  
  }  
}
```

# MÉTODOS COMO PROPIEDADES DE CLASE

- Está en Stage 3 (proposal)
- El más corto

```
handleClick = () => {  
  console.log('Click happened');  
}  
render() {  
  return <button onClick={this.handleClick}>Click Me</button>  
}
```

# EJERCICIO

- Crea un componente llamado SearchBox
  - El componente estará constituido de:
    - Un botón con el texto Buscar
    - Una caja de texto
  - El botón de buscar estará habilitado sólo si hay un texto escrito
  - Al pulsar el botón se debe escribir el texto que se busca en la pantalla.

# SOLUCIÓN

```
import React, { Component } from 'react'

export class CervezasPage extends Component {
  state = {
    disabled: true
  }

  handleSubmit = event => {
    event.preventDefault()
  }

  handleChange = event => {
    this.setState({ searchText: event.target.value }, () =>
      console.log(this.state.searchText)
    )
  }
}
```



# OBSERVACIONES EJERCICIO

- `setState` es asíncrono
- `bind` de `this`

# SETSTATE ES ASÍNCRONO

```
handleChange = event => {  
  this.setState({ searchText: event.target.value }, () =>  
    console.log(this.state.searchText)  
  )  
}
```

```
handleChange = event => {  
  this.setState({ searchText: event.target.value })  
  console.log(this.state.searchText)  
}
```

# BIND DE THIS

- Si no utilizamos nada experimental:
  - En la llamada:

```
<input
  type="text"
  name="busqueda"
  ref={this.textInput}
  onChange={this.handleChange.bind(this)}
/>
```

- En el constructor:

```
constructor() {
  this.handleChange = this.handleChange.bind(this)
}
```

# STATELESS FUNCTIONAL COMPONENTS

- Muchos componentes son sencillos y no tienen estado
  - Deberían ser la gran mayoría de nuestra aplicación
- Se pueden crear mediante clases pero mejor como funciones
  - Más sencillos de leer
  - Más fáciles para hacer tests

# EJEMPLO COMPONENTE COMO FUNCIÓN

```
const User = (props) => {  
  return (  
    <div>  
      <p>Nombre: {props.nombre}</p>  
      <p>Edad: {props.edad}</p>  
    </div>  
  )  
}  
  
ReactDOM.render(  
  <User nombre='Pepe' edad={20} />,  
  document.getElementById('app')  
)
```

# CLASES O FUNCIONES

- Si tu componente necesita acceder a los métodos que tienen las clases de React utiliza una clase
- Si necesitas acceder a this, usa una clase

```
function IsThisUndefined(props) {  
  return <div>{props.title} {this === undefined ? 'Yes' : 'No'}  
}  
  
ReactDOM.render(  
  <IsThisUndefined title='Is this undefined?' />,  
  document.getElementById('root')  
)
```

- En otro caso, usa una función

# DEBUG

- Utilizaremos las [React Dev Tools](#)

# EJERCICIO

- Cambia todos los componentes que puedas de clases a funciones



# ENRUTADO

# CONFIGURAR LAYOUT

- Podemos definir el main de nuestra aplicación mediante el paso de una propiedad:

```
const Layout = (props) => {  
  return (  
    <div>  
      <Header />  
      {this.props.template}  
      <Footer />  
    </div>  
  )  
}  
  
ReactDOM.render(  
  <Layout content={template} />,  
  document.getElementById( 'app' )  
)
```

# PROPS.CHILDREN

- Normalmente se hace utilizando la propiedad *children* que viene por defecto

```
const Layout = (props) => {  
  return (  
    <div>  
      <Header />  
      {this.props.children}  
      <Footer />  
    </div>  
  )  
}  
  
ReactDOM.render(  
  <Layout>  
    <div>  
      <p>Esto sería el main</p>  
    </div>  
  )  
)
```

# ENRUTADO EN CLIENTE

- La generación del html ocurre en el cliente
  - Se usa la API de historial del navegador
    - Si no, no funcionan los botones de retroceso y avance del historial
  - También puede ser que el usuario acceda directamente a una URL específica
    - El servidor web debe delegar en el cliente (no mostrar un 404)
- Se busca una opción más específica (un componente de React)

# NUESTRAS RUTAS

- /
  - Página de inicio
- */cervezas*
  - Sacaremos un listado de las cervezas
- */cervezas/id*
  - Veremos la ficha de una cerveza
- */contactar*
  - Veremos la página de contactar

# ENRUTADO EN CLIENTE

- Utilizaremos [react-router](#)

```
npm i -S react-router-dom
```

- Otros paquetes:
  - *react-router* para nativo y DOM
  - *react-router-native* solo nativo

# ENRUTADO EN SERVIDOR

- Solicitud *midominio/*
  - El servidor devuelve nuestro index.html y su js asociado
  - Todo funciona
- ¿Y si solicitamos *midominio/cervezas*?
  - La ruta no existe en el servidor....

- Solicitud *midominio/cervezas*
  - El servidor no encuentra la ruta/fichero
  - Es una ruta "local"
  - ¿Devuelve un 404?
    - ¡No! Siempre devuelve el index.html y su js asociado
    - El enrutador en local se encargará de resolver en función de *window.location.url*



# CONFIGURACIÓN DE ENRUTADO EN SERVIDOR (APACHE)

```
# To host on root path just use "<Location />" for http://myd
# To host on non-root path use "<Location /myreactapp>" for ht
# If non-root path, don't forgot to add "homepage": "/myreacta
<VirtualHost *:80>
    ServerName mydomainname.in
    DirectoryIndex index.html
    DocumentRoot /var/www/html

    ErrorLog ${APACHE_LOG_DIR}/error.log
    CustomLog ${APACHE_LOG_DIR}/access.log combined

    <Location /myreactapp >
        RewriteEngine on
        RewriteCond %{REQUEST_FILENAME} !-d
        RewriteCond %{REQUEST_FILENAME} !-f
```

# CONFIGURACIÓN DE ENRUTADO EN SERVIDOR (NGINX)

```
# To host on root path just use "location /" for http://mydom
# To host on non-root path use "location /myreactapp" for http
# If non-root path, don't forgot to add "homepage": "/myreacta
server {
    server_name mydomainname.in;
    index index.html;
    error_log /var/log/nginx/error.log;
    access_log /var/log/nginx/access.log;
    root /usr/share/nginx/html;

    location /myreactapp {
        try_files $uri $uri/ /myreactapp/index.html;
    }
}
```

# COMPONENTES REACT ROUTER

# BROWSERROUTER

- Encargado de comunicarse con la API de historial de HTML5 del navegador

```
import { BrowserRouter as Router } from 'react-router-dom'  
ReactDOM.render(  
  <Router>  
    <App />  
  </Router>,  
  document.getElementById('root')  
)
```

# ROUTE

- Encargado de renderizar algo en función de la ruta

```
<Route exact path="/" component={HomePage} />  
<Route path="/contactar" component={ContactPage} />  
....
```

# SWITCH

- Por defecto las rutas son inclusivas
  - Podemos utilizar *Switch* para que sean exclusivas
  - En el siguiente caso, la ruta */about* renderizaría todos los componentes:

```
<Route path="/about" component={About}/>  
<Route path="/:user" component={User}/>  
<Route component={NoMatch}/>
```

- En este caso */about* solo el componente *About*:

```
<Switch>  
  <Route path="/about" component={About}/>  
  <Route path="/:user" component={User}/>  
  <Route component={NoMatch}/>  
</Switch>
```

# LINK

- Comportamiento hipervínculos:
  - El navegador busca en un servidor la página según el atributo href
  - Se resuelve en remoto
- Queremos que resuelva "en local"
  - Mediante las **Route** que haya definidas
  - Usaremos el componente **Link**
  - Link se renderiza como un hipervínculo



# EJEMPLO USO LINK

```
import { Link } from 'react-router-dom'
const Nav = () => (
  <Link to='/'>Home</Link>
)
```

# EJERCICIO

- Utiliza los componentes BrowserRouter, Route, Switch y Link para renderizar las siguientes páginas:
  - Inicio
  - Cervezas
  - Cerveza
  - Contactar
- Crea un componente de tipo catchAll para las rutas que no existan

# APP.JS

- Inyectando React Router queda así:

```
import React, { Component } from 'react'
import { BrowserRouter as Router } from 'react-router-dom'
import './App.css'
import Header from './Header'
import Footer from './Footer'
import Sidebar from './Sidebar'
import Main from './Main'

class App extends Component {
  render() {
    return (
      <Router>
        <div>
          <Header
            titulo="Buscador de cervezas"

```

# SIDEBAR.JS

```
import React, { Component } from 'react'
import './Sidebar.css'
import { Link } from 'react-router-dom'

export class Sidebar extends Component {
  render() {
    return (
      <aside>
        <nav>
          <ul>
            <li>
              <Link to="/">Inicio</Link>
            </li>
            <li>
              <Link to="/cervezas">Buscador de Cervezas</Link>
            </li>
          </ul>
        </nav>
      </aside>
    )
  }
}
```

# MAIN.JS

```
import React, { Component } from 'react'
import IndexPage from './IndexPage'
import CervezaPage from './CervezaPage'
import CervezasPage from './CervezasPage'
import ContactPage from './ContactPage'
import NoMatch from './NoMatch'
import { Route, Switch } from 'react-router-dom'
import './Main.css'

export class Main extends Component {
  render() {
    return (
      <main>
        <Switch>
          <Route path="/" exact component={IndexPage} />

```

# REFACTORIZACIÓN DE CÓDIGO

# CONTANEDORES Y COMPONENTES

- Los contenedores son los que reciben los datos
  - Se encargan de recibir los datos
  - Renderizan componentes (sus hijos), pasándoles las props que necesitan
- Los componentes "son tontos"
  - Reciben unos datos
  - A partir de dichos datos su estado/representación es siempre el mismo.

# EN NUESTRO CASO PARTICULAR

- Creamos vistas para cada página que actúan como contenedores
  - Agrupan componentes
  - Les mandan como deben representarse
- Agrupamos los componentes en una carpeta propia
- Creamos otra carpeta (opcional) para containers
- Cada componente en su carpeta con su nombre (opcional)
  - Se crea un index.js por componente para ahorrar código



# CERVEZASVIEW

- Contiene los componentes **\*SearchBox** y *CervezasList*
- Obtiene vía AJAX las búsquedas de la cervezas
  - Realmente no lo obtiene el, lo hace searchBox pero porque le pasa la función...
- Cambia su estado
- Envía esos datos al componente CervezasList para que los represente
  - Este a su vez se lo manda a cada componente CervezaSnippet

# ¿REFACTORIZAMOS?

- ¿Aprovechamos para solicitar los datos a la API REST?
- Ejemplo solicitud API

# WEBPACK

# ¿QUÉ ES WEBPACK?

- Un empaquetador de módulos (module bundler)
  - Genera un archivo único con los módulos que necesita tu aplicación para funcionar.
  - Permite generar bundles independientes por cada página de tu app
    - Suelen ser aplicaciones con mucho JavaScript
    - Hace más rápida la carga

