Xseed

Curso de React _ Technisys

Xseed

Clase 04 -

Webpack, Babel, Proyecto desde cero, Lists, Props.children

by Diego Cáceres

Repaso

ES6

Se revisarán las siguientes características:

- Arrow functions.
- `var`, `let` y `const`.
- Default values.
- Object Destructuring.
- Enhanced Object Properties.
- Spread operator "...".
- String templates.
- Imports / Exports.

** Estas no son las únicas características nuevas en ES6.*

Arrow functions

Es una nueva forma de **declarar las funciones**, simplemente indicando los parámetros entre paréntesis. Ya no se necesita la palabra `function`, pero se agrega una flecha `=>`

```
// ES5:
var createGreeting = function(name, message) {
     var result = message + ' ' + name;
     return result;
}
// ES6:
var arrowGreeting = (name, message) => {
     var result = message + ' ' + name;
     return result;
}
```

Definición de variables: `var`, `let` y `const` (1)

En JavaScript siempre se utiliza ***`var`*** para declarar variables. En el siguiente ejemplo se puede pensar que en consola saldrá “Sunday” porque el resto está en un bloque, pero en realidad es la misma variable que se re-asigna.

```
var message = 'Sunday';
{
     var message = 'Monday';
}
console.log(message); // Imprime Monday
```

Lo que ya existe en JavaScript es “*function scoping*”, donde las variables existen dentro de la función únicamente. Con el resto de los bloques (***`for`***, ***`if`***, etc) no hay *scope* y sería la misma variable.

```
var message = 'Sunday';
function dayOfWeek() {
    var message = 'Monday';
}
console.log(message); // Imprime Sunday
```

Definición de variables: `var`, `let` y `const`

Hasta ES5 cuando se desea declarar algo como constante, como buena práctica se declara en mayúsculas, pero eso no impide que se vuelva a modificar su valor en el código.

```
var PI = 3.14;
PI = 10;
console.log(PI); // Imprime 10
```

En ES6 se introduce la palabra clave **const** para declarar variables que no se quiere que se les pueda modificar el valor. Son **Read-Only**.

```
const PI = 3.14;
PI = 10; // -> Esto tira un error de Read Only en consola
console.log(PI);
```

Object Destructuring

En JavaScript para acceder a una propiedad dentro de un objeto, se utiliza el punto "."
En ES6 también se puede hacer a través de “destructuring”.

```
var person = {
    name: 'Mike'
}
console.log(person.name);

// Con Destructuring:
var { name } = person;
console.log(name);
```

Spread Operator ...

Este nuevo operador permite tomar un array y separarlo en cada uno de sus ítems (spread = propagar).

```
console.log([1,2,3]); // [1, 2, 3]

console.log(...[4,5,6]); // 4 5 6

// Esto es util para unir de forma facil dos arrays

const person = { a: 1, b:2, c:3 }
const copy = { ...person }
console.log(first); //[1, 2, 3, [4, 5, 6]]

// Si en vez de eso uso spread

let first = [1,2,3];

let second = [4,5,6];

first.push(...second);

console.log(first); //[1, 2, 3, 4, 5, 6]
```

String Templates

Hasta ES5 es muy común el concatenar variables con strings de la siguiente forma.

```
let name = 'Diego';

let greeting = 'Hello ' + name + ', welcome aboard';

console.log(greeting); // Hello Diego, welcome aboard.
```

En ES6 se puede utilizar el tilde al revés ``` (Alt + 96) para envolver un string template, dentro del cual se puede acceder a evaluar JavaScript dentro de ${ .. }

```
let name = 'Diego';

let greeting = `Hello ${name}, welcome aboard`;

console.log(greeting); // Hello Diego, welcome aboard.
```

Imports - Exports

En ES6 se puede de forma muy simple exportar e importar valores y funciones, sin necesidad de acudir a global namespaces.

```
// En src/utils/math.js:
function square(a) { return a*a; }
export { square };

// En src/index.js:
import { square } from 'utils/math.js';
console.log(square(5)); // 25

// Otra forma de exportar:
// En src/utils/math.js
export function square(a) { return a*a; }
```

Conditional Rendering

En React se pueden crear distintos componentes para encapsular en cada uno el comportamiento necesario. Luego, se puede utilizar **Conditional Rendering** para dibujar solo alguno de ellos, dependiendo del estado de la aplicación.

```
const UserGreeting = (props) => { return <h1>Welcome back!</h1>; }
const GuestGreeting = (props) => { return <h1>Please sign up.</h1>; }

const Greeting = (props) => {
   const isLoggedIn = props.isLoggedIn;
   if (isLoggedIn) {
       return <UserGreeting />;
   }
   return <GuestGreeting />;
}
```

Conditional Rendering – Operador Lógico

Como todo lo que se escriba entre `{ }` será evaluado como una expresión JavaScript, también se puede utilizar el operador lógico de JavaScript `&&` lo que permite lograr condicionales más cortos.

```
function Mailbox(props) {
    const unreadMessages = props.unreadMessages;
    return (
        <div>
            <h1>Hola!</h1>
            {unreadMessages.length > 0 &&
                <h2>
                    Tenés {unreadMessages.length} mensajes sin leer.
                </h2>
            }
        </div>
    );
}
```

Webpack

Create React App

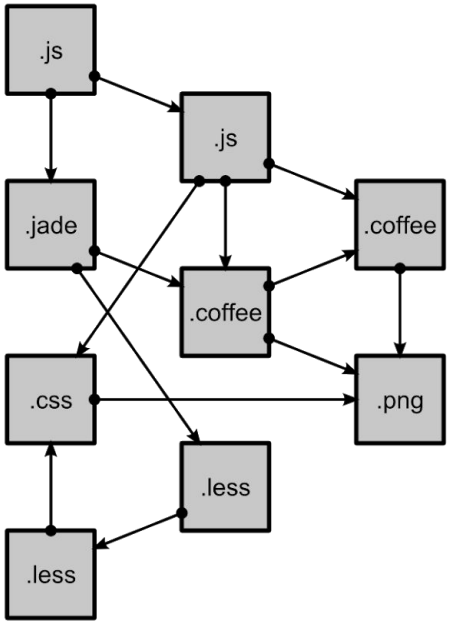
Hasta ahora se ha venido utilizando Create React App para crear nuevos proyectos y esto es muy conveniente ya que resuelve un montón de cosas por nosotros. Pero ¿qué es en realidad lo que resuelve?

Lo que hace por detrás es escribir archivos de configuración, instalar varias librerías que ayudan a facilitar el desarrollo (como Hot Reloading) y utiliza determinados procesadores por donde pasará el código (esto es lo que permite utilizar ES6 o JSX, por ejemplo). Luego esto queda oculto, pero en caso de querer modificar algo, se podría utilizar el comando `eject` y esto expondrá los archivos de configuración y demás, pero ojo, **es irreversible**.

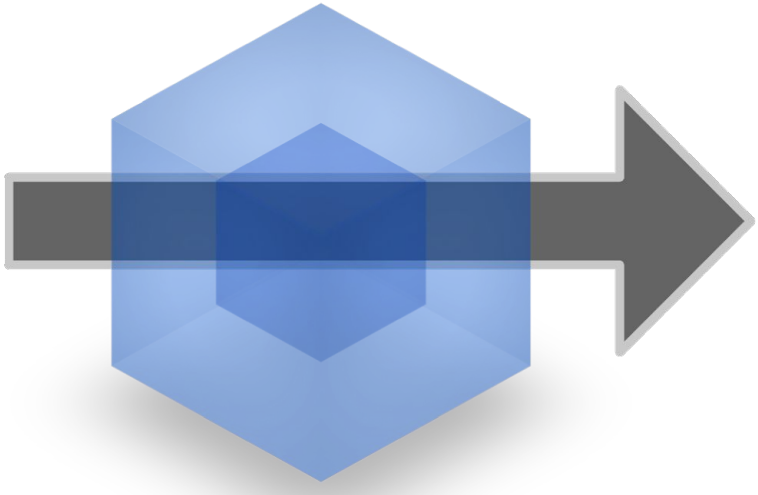
Webpack (1)

Webpack es un sistema de bundling que prepara el código desarrollado en una aplicación para *producción*.

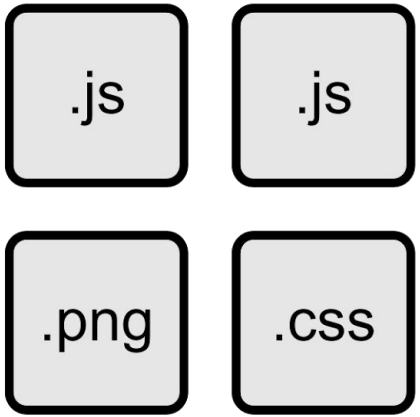
Se puede considerar como un Browserify avanzado ya que tiene muchas opciones de configuración. También es similar a Grunt y Gulp, ya que permite de alguna manera automatizar los procesos principales que son transpilar y preprocesar código de `.scss` a `.css`, de ES7 a ES5/6, etc.



modules
with dependencies



webpack
MODULE BUNDLER



static
assets

Webpack (2)

El comando más sencillo para realizar un bundle de los archivos JavaScript es el siguiente:

```
$ webpack ./index.js ./build/app.js
```

Este comando lo que hace es leer el archivo `index.js` (que sería el punto de entrada en la aplicación) e importar todos los módulos que estén definidos y crear el archivo de *producción* `app.js` en la carpeta `/build`.

En el caso de React esto no es suficiente, porque además se necesita aplicar loaders que procesen el código, como por ejemplo **Babel**.

Webpack (3)

En resumen, Webpack necesita saber:

1. Punto de entrada de la aplicación.
2. Transformaciones a realizar al código.
3. Ubicación donde guardar el código transformado.

Esto es lo que se escribe en el archivo de configuración de Webpack, y aunque al principio el código de este archivo parezca muy complicado, no es más que indicar los tres datos mencionados.

Babel

Babel

Babel es simplemente un compilador de JavaScript. Esto es lo que permite escribir código JavaScript de última generación, y estar seguros de que igual va a ser soportado por todos los navegadores.

También se utiliza para poder escribir JSX en los archivos JavaScript ya que es el encargado de convertirlo en JavaScript plano.

Babel funciona con plugins para indicarle qué cosas son necesarias que transforme.

Crear un proyecto de React desde cero

Ejercicio 1 (1)

Para que estos conceptos queden más claros, se va a crear un proyecto de React desde cero utilizando **Webpack** y **Babel** (no quiere decir que la mayoría de las veces sea más simple utilizar Create React App).

- Crear una carpeta `'ejercicio01'` en `EjerciciosClase04/ejercicio01` y abrir una consola en esa ubicación.
- Utilizar el comando `$ npm init` para crear el `package.json`. Pueden contestar las preguntas con un simple enter para aceptar el valor por defecto.
- Ahora instalar las primeras dependencias de nuestro proyecto, React y ReactDom:
  `$ npm install --save react react-dom` *(--save es opcional si usan npm mayor a version 5)*

  Esto creará la carpeta `node_modules`, que es donde quedarán guardadas las dependencias del proyecto. Revisar el `package.json` para comprobar que es similar al de la siguiente slide.

Ejercicio 1 (2)

```
{
   "name": "ejercicio01",
   "version": "1.0.0",
   "description": "",
   "main": "index.js",
   "scripts": {
       "test": "echo \"Error: no test specified\" && exit 1"
   },
   "author": "",
   "license": "ISC",
   "dependencies": {
       "react": "^16.4.0",
       "react-dom": "^16.4.0"
   }
}
```

~ significa que solo actualice en los cambios menores de versión.
~1.2.3 va a bajar todas las versiones 1.2.x pero no la 1.3.0

^ significa que también actualice los cambios dentro de la versión actual, es decir; ^1.2.3 podrá a bajar todas las versiones 1.x.x incluyendo 1.3.0, pero no la 2.0.0

Ejercicio 1 (3)

Cuando se desee subir este proyecto a un repositorio (ej: Github) no es necesario subir la carpeta `node_modules`. Esto se debe a que las dependencias están definidas en `package.json`. Cuando otro desarrollador precise descargarse las dependencias sólo deberá correr el comando: `$ npm install` (y esto le creará carpeta `node_modules`).

A continuación se creará un archivo llamado `.gitignore` para evitar que la carpeta `node_modules` se suba al repositorio. Lo mismo se hará con la carpeta `dist`, que aún no existe, pero es donde estará el código de producción. Escribir dentro del archivo las siguientes líneas:

```
node_modules
dist
```

Ahora se debe crear el archivo `index.js` que es el punto de entrada que se utilizará en el proyecto. Dentro del archivo crear el primer componente.

Ejercicio 1 (4)

```
// En index.js:
import React from "react";
import ReactDOM from "react-dom";

class App extends React.Component {
    render() {
        return (
            <div>
                Hello World!
            </div>
        );
    }
}
```

Ejercicio 1 (5)

Para mostrar el componente se debe utilizar `ReactDom.render`, que recibe dos parámetros: el primero es el elemento que se quiere renderizar y el segundo es dónde se lo quiere mostrar.

Agregar al final de `index.js` la siguiente línea para mostrar el componente:

```
ReactDOM.render(<App />, document.getElementById("root"));
```

Esto quiere decir que en el principal archivo html, debe existir un `div` con `id="root"` para que esto funcione. Se creará más adelante.

Ejercicio 1 (6)

Todavía queda por configurar Webpack y Babel para que quede funcionando.

Es necesario instalar algunas librerías más, con la diferencia de que ahora se usará `--save-dev` para que sean dependencias sólo mientras se está *desarrollando*. En el build final de *producción* estos paquetes no son necesarios.

```
$ npm install --save-dev babel-core babel-loader
babel-preset-env babel-preset-react css-loader style-loader
html-webpack-plugin webpack webpack-dev-server webpack-cli
```

Luego se verá para qué son cada uno de estos paquetes.

Ejercicio 1 (7)

Ahora se creará un archivo de configuración de Webpack.
Crear un archivo `webpack.config.js`

Hay que asegurarse de exportar un objeto desde el archivo, conteniendo la configuración:

```
// En webpack.config.js
module.exports = {}
```

Luego hay que indicarle el primero de los 3 pasos, es decir, cual es el archivo de ingreso al proyecto:

```
// En webpack.config.js
module.exports = {
   entry: './index.js',
}
```

Ejercicio 1 (8)

El segundo paso es indicarle qué transformaciones realizar al código. Para esto se utilizan los *loaders*. Primero se agrega una nueva propiedad al objeto que se exporta, llamada `module`. Y dentro de *module* se agrega otra propiedad llamada `rules` que será un array:

```
module.exports = {
   entry: './index.js',
   module: {
       rules: []
   },
}
```

Para cada transformación que se quiera que realice Webpack, se debe instalar el *loader* correspondiente, e indicarlo en el array de *rules*.

Ejercicio 1 (9)

Cada regla que se agrega tiene dos partes; la primera indica sobre qué archivos aplicarla y la segunda indica qué *loader* utilizar sobre esos archivos:

```
module.exports = {
   entry: './index.js',
   module: {
       rules: [
           { test: /\.js$/, use: "babel-loader" },
       ]
   },
}
```

En este caso se está indicando que en todos los archivos JavaScript, aplique el `babel-loader`, que es uno de los instalados previamente por `npm`.

Ejercicio 1 (10)

Para que el `babel-loader` funcione en el `package.json` se debe agregar una entrada más:

```
"babel": {
       "presets": [
           "env",
           "react"
      ]
},
```

El *loader* buscará la clave `babel`, verá que hay dos `presets` y los aplicará.

- `env` es el que transformara el código a la versión de JavaScript soportada por los navegadores.
- `react` es el que transforma JSX en JavaScript.

Ejercicio 1 (11)

Luego se va a agregar otra regla más para poder importar archivos CSS en los componentes (`css-loader`), y para que los estilos que se utilicen se inserten en la página (`style-loader`).

```
module.exports = {
   entry: './index.js',
   module: {
     rules: [
           { test: /\.js$/, use: "babel-loader" },
           { test: /\.css$/, use: ["style-loader", "css-loader"] }
       ]
   },
}
```

Ejercicio 1 (12)

El último paso es indicar en qué ruta dejar el código transformado. Para esto se agrega la propiedad `output` (hay que hacer un `require` del path, que ya está instalado en node, para trabajar con el file system):

```
var path = require("path");
module.exports = {
   entry: './index.js',
   module: {
       rules: [
           { test: /\.js$/, use: "babel-loader" },
           { test: /\.css$/, use: ["style-loader", "css-loader"] }
       ]
   },
   output: {
       path: path.resolve(__dirname, 'dist'),
       filename: 'index_bundle.js'
   }
```

Ejercicio 1 (13)

Un detalle más, a partir de Webpack 4 se debe indicar un modo: `development` o `production`.
De esta forma, Webpack puede aplicar las optimizaciones correspondientes.

```
var path = require("path");
module.exports = {
   entry: './index.js',
   module: {
       rules: [
           { test: /\.js$/, use: "babel-loader" },
           { test: /\.css$/, use: ["style-loader", "css-loader"] }
       ]
   },
   output: {
       path: path.resolve(__dirname, 'dist'),
       filename: 'index_bundle.js'
   },
   mode: 'development'
```

Ejercicio 1 (14)

En resumen, cuando Webpack se ejecute, dejará en `dist/index_bundle.js` el código transformado. Ahora lo que falta es que exista el html en `dist/index.html` que utilice este archivo.

Para esto se va a utilizar un plugin que permite tomar el `index.html` del proyecto como template, para crear uno nuevo en `dist`, que además contenga el script tag para utilizar `index_bundle.js`: `html-webpack-plugin`.

- Primero habría que instalarlo, pero esto ya se hizo anteriormente con:
  `$ npm install --save-dev html-webpack-plugin`
- Luego, incluirlo al comienzo del archivo de configuración:

```
var HtmlWebpackPlugin = require("html-webpack-plugin");
```

- Por último, se debe crear una nueva instancia de `HtmlWebpackPlugin` dentro de una propiedad `plugins` en el objeto de configuración. Ver código a continuación.

```
var path = require("path");
var HtmlWebpackPlugin = require("html-webpack-plugin");
module.exports = {
   entry: './index.js',
   module: {
       rules: [
           { test: /\.js$/, use: "babel-loader" },
           { test: /\.css$/, use: ["style-loader", "css-loader"] }
       ]
   },
   output: {
       path: path.resolve(__dirname, 'dist'),
       filename: 'index_bundle.js'
   },
   mode: 'development',
   plugins: [
       new HtmlWebpackPlugin({
         template: 'index.html'
       })
   ]
}
```

Ejercicio 1 (15)

Ahora se va a crear el `index.html` en la raíz del proyecto, que será además el template para webpack, con el siguiente código.
(Notar que acá está el `div` con `id="root"` al que se hizo referencia al utilizar `ReactDOM.render` en `index.js`).

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>Demo Project</title>
    </head>
    <body>
        <div id="root"></div>
    </body>
</html>
```

Ejercicio 1 (15)

Ahora ya se puede probar ejecutar el Webpack, para esto hay que editar el `package.json` de nuevo y agregar dentro del tag de `scripts` dos nuevos comandos:

- Con el `build` se está simplemente ejecutando Webpack, por lo que generará la carpeta `dist` con el código transformado.
  - `$ npm run build`
- Con `start` se inicia un servidor local de Webpack que es más útil para el desarrollo ya que se queda "mirando" los archivos y los compila cada vez que cambian.
  - `$ npm start`

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

Xseed

Lists en React

Lists – `map` (1)

En la mayoría de los frameworks existe alguna sintaxis especial para crear listas de elementos en la UI, por ejemplo, en Angular se usaría `ng-repeat`. En React se utiliza simplemente JavaScript, por ende, para crear una lista de elementos se puede utilizar el método nativo de JavaScript **`map`**.

`map` es un método (que tienen todos los arrays) que retorna un nuevo array con los elementos del array original habiéndose aplicado una función sobre cada uno de ellos.

```
var numbers = [1,2,3];
var numbersPlusTen = numbers.map(
    function (num) {
        return num + 10;
    }
);
console.log(numbersPlusTen);
// Resultado: [11, 12, 13]
```

```
// ES6:
const numbers = [1, 2, 3];
const numbersPlusTen = numbers.map(num => num + 10);
console.log(numbersPlusTen); // [11, 12, 13]
```

Lists – `map` (2)

Si se quiere crear una lista de elementos en la UI en React, se puede simplemente utilizar `map` y retornar elementos en la función. Por ejemplo, el siguiente componente recibe por *`props`* una lista de nombres de amigos:

```
const ShowList = (props) => {
   return (
       <div>
           <h3>Amigos:</h3>
           <ul>
               {props.names.map((friend) => {
                   return <li> {friend} </li>;
               })}
           </ul>
       </div>
   );
}
```

Lists – `map` (3)

Utilizando el componente anterior de la siguiente forma:

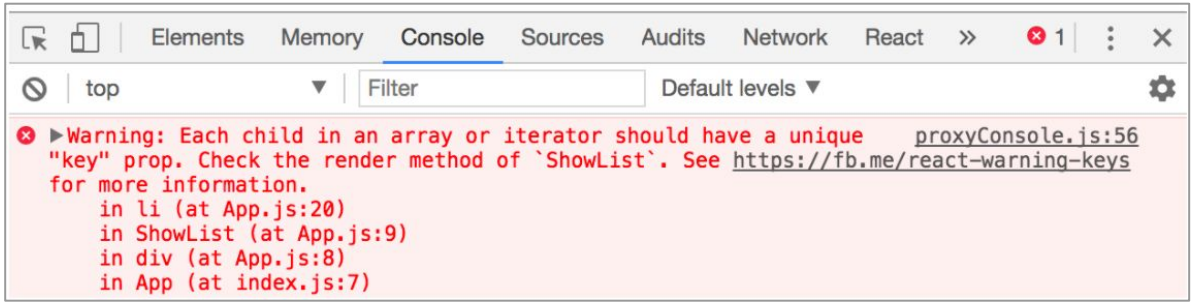
```
<ShowList names={["Diego", "Juan", "Pedro"]} />
```

En la página se verá el siguiente resultado:

Amigos:

- Diego
- Juan
- Pedro

Y en la consola se verá:



Lists – `key` (1)

Warning: Each child in an array or iterator should have a unique "key" prop

- Este es un *warning* que se verá siempre que un componente de React renderice una lista de elementos y no se le asigne una `key` propia a cada uno de ellos. Esto es un requerimiento de React para poder ser más eficiente a la hora de actualizar la UI.
- React utiliza esta `key` para identificar exactamente qué elemento fue modificado, eliminado o agregado en una lista de elementos, sin necesidad de volver a renderizar toda la lista.
- Esta `key` debe ser única entre los elementos de la lista. Si se cuenta con un `id` claramente es la mejor opción, pero si no, se puede utilizar el índice del elemento en el array como último recurso.

Lists – `key` (2)

En la función que se declaró para el `map` se puede recibir como segundo parámetro el índice del elemento en el array.

```
const ShowList = (props) => {
   return (
       <ul>
         {this.props.list.map((friend, index) => {
           return (
               <li key={index}>
                   {friend}
               </li>
           );
         })}
       </ul>
   );
}
```

Lists – `key` (3)

Las `key` sólo son necesarias entre los "hermanos" de la lista que se renderiza. Es decir, si se extrae un componente `ListItem`, el `key` pasaría del `<li>` al `<ListItem>`.

```
function ListItem(props) {
    return <li>{props.value}</li>;
}
function NumberList(props) {
    const numbers = props.numbers;
    const listItems = numbers.map((number, index) =>
        <ListItem key={index} value={number} />
    );
    return (
        <ul>{listItems}</ul>
    );
}
```

Lists – `filter`

Otro método de los arrays en JavaScript que suele ser útil en React es `filter`. Funciona de forma muy similar a `map` pero, en vez de aplicar una función sobre cada elemento, permite filtrar ciertos elementos que no cumplan la condición dada. En el siguiente ejemplo se obtienen sólo los nombres que empiecen con E:

```
var friends = ['Emiliano', 'Juan', 'Pablo', 'Ernesto', 'Tomás'];
var newFriends = friends.filter(function (name) {
    return name[0] === 'E'
});

console.log(newFriends) // Retorna: ['Emiliano', 'Ernesto']
```

Props.children

Props.children (1)

Cuando las expresiones JSX tienen tag de apertura y de cierre, el contenido entre ellas es enviado al componente como una propiedad especial, que se accede mediante `props.children`.

Estos "hijos" pueden ser de distintos tipos.

En el siguiente caso, `props.children` será simplemente un string:

```
<MyComponent>Hello world!</MyComponent>
```

Props.children (2)

En realidad, `props.children` es un array, lo cual es útil para poder brindar varios elementos como hijos.

Esto permite crear componentes que agreguen comportamiento a la aplicación y simplemente envuelvan al resto de los componentes. Pero para esto el "container" debe explícitamente renderizar a sus hijos, de la siguiente forma:

```
<MyContainer>
    <MyFirstComponent />
    <MySecondComponent />
</MyContainer>
```

```
class MyContainer extends Component {
    render() {
        return <div>{this.props.children}</div>
    }
}
```

Function as a child

Se puede pasar cualquier expresión JavaScript como hijo, incluso una función.

```
class Executioner extends React.Component {
    render() {
        // Ejecutamos a children como una función:
        return this.props.children()
    }
}
<Executioner>
    {() => <h1>Hello World!</h1>}
</Executioner>
```

Manipular los hijos

Como los hijos pueden ser literalmente cualquier cosa, React proporciona *helpers* para manipularlos, los cuales se encuentran en `React.Children`.
Por ejemplo, se puede utilizar `React.Children.map` y `React.Children.forEach` para recorrerlos, sin importar de qué tipo sean.

```
class IgnoreFirstChild extends Component {
   render() {
       const children = this.props.children
       return (
           <div>
               {React.Children.map(children, (child, i) => {
                   // Se ignora el primer hijo:
                   if (i < 1) return
                   return child
               })}
           </div>
       )
   }
}
```

```
<IgnoreFirstChild>
     <h1>First</h1>
     <h1>Second</h1>
</IgnoreFirstChild>
```

Ejercicios

Ejercicios

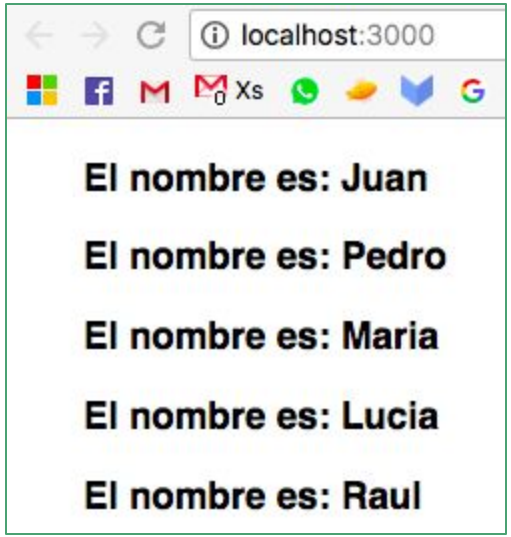
Descargar el zip `EjerciciosClase04`. Cada carpeta dentro corresponde a un ejercicio y es un proyecto creado con Create React App, por lo que para ejecutarlo es necesario:

- Si usan yarn:
  - `$ yarn install`
  - `$ yarn start`
- Si usan npm:
  - `$ npm install`
  - `$ npm start`

Se debería abrir automáticamente el explorador con el ejercicio corriendo en http://localhost:3000/

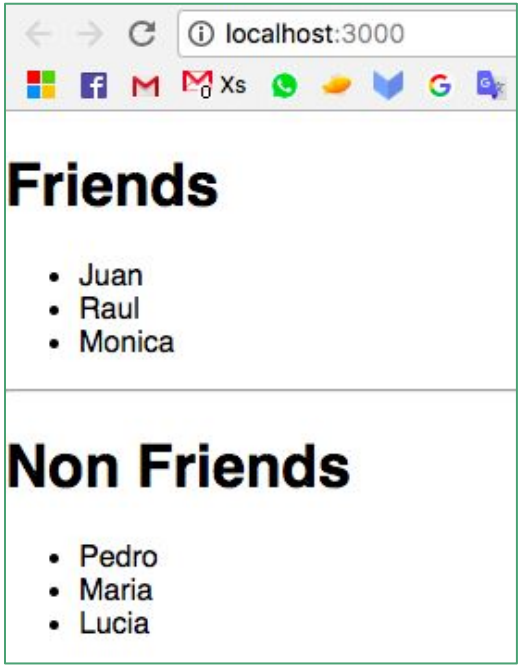
Ejercicio 2 - parte 1

- Editar el archivo `src/App_part1.js`.
- Completar el componente `Users` de forma que muestre todos los nombres recibidos por Props.



Ejercicio 2 - parte 2

- Editar el archivo `src/App_part2.js`.
- Completar el componente `Users` de forma que muestre correctamente la lista de Amigos y de No Amigos.



Ejercicio 3

- En este ejercicio se parte de la solución del primer ejercicio de la clase 2.
- La idea es modificar `App` para que no repita un `<li />` para cada lenguaje, sino que ahora utilice la función `map`.

Ejercicio 4

- Completar el componente `ShowHideBoard` para que decida si mostrar o no el tablero. El botón debe cambiar el valor de la bandera `showBoard`.
- Modificar `BoardSwitcher` y mostrar dentro del `div` que tiene `className="boards"` una lista de componentes `Board`, uno para cada uno de los que se reciben por Props.
- Asegurarse de indicarle a cada `Board` si es el seleccionado o no.
- Implementar el botón `Toggle` de forma que cambie el seleccionado al siguiente y comience de nuevo cuando llegue al final.

Ejercicio 4