Universidad Tecnológica Nacional

Facultad Regional Rosario

Tecnicatura Universitaria de la Programación

Programación III

Unidad 1.0

Introducción a React y JS moderno

Introducción a React

¿Qué es React?

React, o React.js como es correctamente llamado, nos anuncia en su propia página como debemos definirlo:

Una biblioteca Javascript para construir interfaces de usuario

El nombre de React viene del adjetivo *reactive* que nos da a entender que lo que buscamos son interfaces de usuario reactivas al *input* del mismo, que se comporten de manera similar a una app de nuestro teléfono.

En tiempos anteriores, las páginas web poseían un cierto *delay* en la obtención de contenidos, esto es debido a que se debía realizar un "pedido" al servidor y este a su vez tenía que devolver la página HTML con el contenido deseado.

Entonces, Javascript viene a salvarnos de ese *delay*, ya que corre en el navegador, es decir, en la página ya cargada. Con este lenguaje podemos manipular el DOM de HTML (DOM,*Document Object Model*, que analizaremos en profundidad más adelante) y así lograr que el usuario sienta una mayor fluidez dentro del sitio web, obteniendo una experiencia mucho más refinada y eficiente.

Pero entonces, ¿Por qué no usamos directamente Javascript en vez de React?

¿Por qué usar React?

La razón por la que usamos React y no Vainilla JS es la misma razón de porqué utilizamos C# en vez de C o C++, o por qué utilizamos Python en vez de Pascal o Node.js en vez de PHP. **Como desarrolladores de software, siempre vamos a buscar agilizar nuestros procesos para conseguir escribir mejor código en menos tiempo**. Menos código a su vez nos permite obtener un código **más prolijo, más legible y más sencillo de testear**.

La implementación de React nos permite además la reutilización de bloques de código funcionales.

Construir SPA’s con React

Las *single page application* o SPA (aplicaciones de única página) son aplicaciones web que se traen del servidor una sola página (llamada *index.html* o *main.html*) que se encargará de realizar la totalidad del manejo del sistema web *front-end.* Esto es ampliamente diferente al manejo de páginas múltiples antiguo, donde cada página de HTML era servida al usuario para su navegación. La posibilidad de que la navegación (movernos entre distintas páginas internas

del sistema web) sea tan fluida es porque siempre estamos navegando dentro del mismo documento HTML.

Javascript moderno

Para poder escribir código React de manera eficiente, es muy común utilizar lo que se conoce como "Javascript moderno", que refiere a funcionalidades ES6+ (*ecmascript 6 y posteriores*). Esto es debido a que dichas funcionalidades nos aportan mejoras (tal como la conocida *arrow function* que además de ser más sencilla de escribir nos libera del concepto de *this* en JS) como atajos de manera que escribamos menos código (conceptos como *destructuring*).

let vs *const*

Dos palabras claves altamente utilizadas, en especial *const*. Ellas nos permiten crear variables de manera similar a *var.* Justamente, lo que antes se conocía como *var* en JS fue reemplazado por *let* y *const*. Además, detrás de escena, ellas también están atadas al concepto de *scope* que refiere a que pertenecen al bloque de código donde fueron creadas.

- **let**: Se utiliza para valores variables.
- **const**: Se utiliza para declarar un valor "constante", es decir, un valor que no planea ser cambiado en el futuro.

```
1  var myName = "Gaby";
2  console.log(myName);
3
4  myName = "Dante";
5  console.log(myName);
6
```

El código anterior nos imprimirá en la primera línea Gaby y luego en la segunda Eze, es decir, la variable "myName" poseía un valor y luego ese valor fue reemplazado sin problemas. Lo mismo sucederá si ponemos:

```
1  let myName = "Gaby";
2  console.log(myName);
3
4  myName = "Dante";
5  console.log(myName);
6
```

Solo cambiará el *scope* de la variable. Ahora, si lo intercambiamos por *const*:

```
const myName = "Gaby";
console.log(myName);

myName = "Dante";
console.log(myName);

```

Nos devolverá el siguiente error:

```
Const.js:4
myName = "Dante";
       ^

TypeError: Assignment to constant
variable.
```

Que justamente nos dice que no podemos reasignar el valor en una constante variable (si bien, es medio contradictoria la frase).

Arrow functions

Es una sintaxis diferente para crear funciones JS. Una función estándar en JS se ve así:

```
function myFunc(){

    // ...

}
```

Pero, de la manera *arrow function* se vería así:

```
const myFunc = () => {
  //...
};

```

Analizemos sus partes:

1. La función se guarda en una variable constante (*const*).
2. A la derecha del signo igual, se encuentran los paréntesis donde allí colocaremos los argumentos en caso que los necesitemos.
3. Luego le sigue la "flecha", conformada por un igual y un signo de mayor.
4. Dentro de las llaves colocaremos nuestro código.

Este tipo de declaración no solo nos ahorra tiempo si no que además resuelve varios problemas relacionados con la palabra clave *this,* debido a que *this* en versiones anteriores de JS era medio ambiguo a lo que refería.

Comparativa de dos funciones que imprimen un argumento:

```
9   function printName(name) {
10    console.log(name);
11  }
12
13  const printName = (name) => {
14    console.log(name);
15  };
16
```

Algunas aclaraciones

- **1 solo argumento**: en el caso de que pasemos un solo argumento, la función puede escribirse de la siguiente manera:

```
17  const printMyName = (name) => {
18    console.log(name);
19  };
20
21  printMyName("Gaby");
22
```

- **Ningún argumento**: en el caso que la función no posea argumentos, de la misma forma es obligatorio escribir los paréntesis. Es decir:

```
23  const sayHello = () => {
24    console.log("Hola Mundo!");
25  };
26
27  sayHello();
28
```

- **Solo retorna un valor**: cuando la función solo retorna un valor (es decir, no posee otra línea de código antes de la palabra clave *return*), es posible obviar las llaves luego de la *arrow*.

```
29  const duplicateNumber = (number) => number * 2;
30
31  console.log(duplicateNumber(7));
32
33  // ¡Incluso podemos eliminar los paréntesis en el argumento!
34  const duplicateNumber2 = (number) => number * 2;
35
36  console.log(duplicateNumber2(7));
37
```

Exportar e importar módulos

La principal idea es poder beneficiarse de importar (o exportar) código JS entre archivos para su uso. Veamos un ejemplo:

Yo poseo el archivo *cat.js* con el siguiente código:

```
1  const cat = {
2    name: "Michelle",
3  };
4
5  export default cat;
6
```

No es necesario exportar una sola cosa por archivo. Imaginemos que existe un archivo llamado *catThings.js* que posee una función y un valor, los dos exportables:

```
1  export const feedCat = () => {
2    console.log("Cat feeded!");
3  };
4
5  export const favouriteFood = "DeliCat";
6
```

Veamos cómo podemos importar esto en nuestro archivo *app.js*

```
import cat from "./cat.js";
import c from ".cat.js";

// Ya que el exportado en el archivo cat.js es por default, no es necesario poner el
// nombre tal cual como lo exportamos, ya que JS va a saber que lo único que tiene que
// exportar es cat.

// La otra forma de importar es:

import { feedCat } from "./catThings.js";
import { favouriteFood } from "./catThings.js";

// Aquí si debemos poner el nombre del módulo a importar exacto ya que JS no posee un
// default para exportar, sino que debe hacer match entre el export y el nombre que le damos nosotros.

// Lo anterior lo podemos reducir a una sola línea de código de la siguiente manera:

import { feedCat, favouriteFood } from "./catThings.js";

// También podemos aplicar alias:

import { favouriteFood as FF } from "./catThings.js";

// O podemos traernos todos los módulos exportados como un objeto JS, donde cada módulo terminará siendo
// una propiedad

import * as catThings from "./catThings.js";

```

Esta modalidad no es compatible con todos los navegadores. En clases siguientes veremos cómo esto puede ser aplicado a todos los navegadores sin dejar usuarios atrás.

Clases

Seguro el alumno en algún otro lenguaje ha escuchado hablar del concepto de clases. Aquí, en JS, las clases son usadas con menor frecuencia ya que JS es un lenguaje principalmente funcional, y luego al modernizarse a incluido clases (**Nota**: el concepto faltante de clases en JS fue una de las principales razones por las que Microsoft desarrollo *typescript*)

Veamos un ejemplo de clase:

```
class Person {
  constructor() {
    this.name = "Gaby";
  }

  cook(food) {
    console.log("Cooking " + food);
  }
}

```

Como vemos las clases pueden poseer propiedades y métodos. De la siguiente forma las inicializamos:

```
11  const myPerson = new Person();
12  console.log(myPerson.name);
13  myPerson.cook("pizzas");
14
```

Las clases en JS también aceptan el concepto de herencia para heredar propiedades o métodos:

```
1  class Human {
2    constructor() {
3      this.hairColor = "Negro";
4    }
5
6    printHairColor() {
7      console.log(this.hairColor);
8    }
9  }
10
11 class Person extends Human {
12   constructor() {
13     this.name = "Gaby";
14   }
15
16   cook(food) {
17     console.log("Cooking " + food);
18   }
19 }
20
21 const myPerson = new Person();
22 myPerson.cook("pizzas");
23 myPerson.printHairColor();
```

El código anterior debería imprimirnos el nombre y el género, ya que hereda las propiedades y métodos de la clase padre. Sin embargo, ese código nos tirará el siguiente error:

```
ReferenceError: Must call super constructor in derived class before accessing 'this' or returning from derived constructor
```

Esto refiere a que debemos agregar *super()* al comienzo del constructor de *Person*, para que tome toda la información de la herencia.

Modernizando las clases

A partir de ES7, se introdujo una manera más moderna de realizar la declaración de propiedades y métodos. Para las propiedades se evita la declaración del constructor:

```
class Human {
  hairColor = "Negro";

  printHairColor = () => {
    console.log(this.hairColor);
  };
}

class Person extends Human {
  name = "Gaby";

  cook = (food) => {
    console.log("Cooking " + food);
  };
}

const myPerson = new Person();
myPerson.cook("pizzas");
myPerson.printHairColor();

```

Y los métodos pasaron a utilizar *arrow functions*.

Spread & Rest operators

En realidad, es un solo operador, denotado por el símbolo de puntos suspensivos (...). Lo llamaremos de manera *Spread* (la más utilizada) o *Rest* según el uso que le demos.

Spread

Se utiliza para "distribuir" los valores (en caso de utilizarse en un arreglo) o las propiedades (en el caso de utilizarse en un objeto)

```
const newArray = [...oldArray,1,2];
const newObject = {...oldObject, newProp: 'dog'}
// En el caso de que oldObject tuviera una propiedad llamada newProp,
// la que posee la cadena 'dog' la reemplazaría.
```

Ejemplo:

```
// En arreglos
const someNumbers = [1, 2, 3];
const someMoreNumbers = [...someNumbers, 4, 5];
console.log(someMoreNumbers);
// Esto imprimirá '[1,2,3,4,5]'

const numbers = [1, 2, 3];
const moreNumbers = [numbers, 4, 5];
console.log(moreNumbers);
// Esto imprimirá '[[1,2,3],4,5]', que no es lo que buscamos.

//En objetos
const person = {
  name: "Gaby",
};

const newPerson = {
  ...person,
  favouriteColor: "Verde bosque",
};
console.log(newPerson);
// Esto imprimirá [object,Object]{ name:'Gaby', favouriteColor:'Verde bosque' }
```

Rest

Se utiliza para elaborar un *merge* entre los argumentos de una función, resultando en la conformación de un arreglo con los mismos.

```
const sortArgs = (...args) => {
    return args.sort()
}
// Aquí todos los argumentos (sean cuales sean) seran parte de un arreglo llamado
// args, y por ende podemos aplicarle métodos de arreglos (como sort())
```

Ejemplo:

```
const filter = (...args) => {
  return args.filter((el) => el === 1);
};
console.log(filter(1, 2, 3));
// Esto imprimirá [1], que es el único argumento que cumple el filtrado

```

Destructuring

El proceso de *destructuring* nos facilita extraer elementos de un arreglo o propiedades de un objeto.

Array destructuring

```
const [a, b] = ["cat", "dog"];
console.log(a); // Imprime 'cat'
console.log(b); // Imprime 'dog'

```

Object destructuring

```
const { userName } = { userName: "Gaby", favouriteColor: "Verde bosque" };
console.log(userName); // Imprime 'Gaby'
console.log(favouriteColor); // Ya que no fue destructurado, devuelve undefined

```

Primitivas y variables de referencia

Si bien esto no es pertinente a ES6+ (JS moderno), realizaremos un breve repaso del concepto de variables primitivas y de referencia.

Variables primitivas

Las variables primitivas (por ejemplo, las de tipo *string*, *boolean* o *number*) son variables que en el caso de duplicarse, copian todo el contenido alojado en la variable original.

```
const number = 1;
const number2 = number;
console.log(number2); // Esto imprimirá el valor 1, osea es una copia exacta de lo
// que había en la primera variable

```

Variables de referencia

Esto no sucede con las variables de tipo *referencia* (arreglos y objetos). Veamos un ejemplo:

```
const person = {
  name: "Gaby",
};
const person2 = person;
console.log(person2); // Esto imprimirá [object Object] { name: "Gaby" }

```

Si bien ahí pareciera que estamos imprimiendo una copia del primer objeto. Lo que sucede es que lo que guardamos en memoria es el objeto *person* pero en la variable person, guardamos un puntero que apunta a ese objeto. Entonces, lo que allí copiamos a *person2* es el puntero, no el objeto.

```
const person = {
  name: "Gaby",
};
const person2 = person;
person.name = "Eze";
console.log(person2); // Esto imprimirá [object Object] { name: "Eze" }, sin tener
// en cuenta que yo deseo imprimir la version anterior copiada del objeto.

```

Esto se relaciona entonces con el concepto de **mutabilidad** dentro de los objeto y arreglos en JS. La mutabilidad es una propiedad que refiere a que los objetos o arreglo pueden mutar sus valores originales, mientras que las cadenas, los números o los valores *booleanos* no (solo puedo crear nuevos a partir de los viejos, no puedo **mutar** los viejos)

Utilizaremos diferentes técnicas para crear copias reales de los objetos y arreglos. Por ejemplo, una de ellas es el *spread operator* discutido más arriba:

```
const person = {
  name: "Gaby",
};
const person2 = {
  ...person,
};
person.name = "Eze";
console.log(person2); // Esto imprimirá [object Object] { name: "Gaby" }

```

Array functions

Las funciones aplicadas a arreglos (*array functions*) son funciones que nos permiten modificar un arreglo original. Hay una amplia variedad de ellas, aquí mostraremos solo el funcionamiento de 3 y nombraremos algunas más.

map()

Esta función nos permite realizar cambios sobre cada uno de los elementos en un arreglo. Retorna un nuevo arreglo con la forma deseada.

```
const numbers = [1, 2, 3];
const doubleNumbers = numbers.map((num) => {
  return num * 2;
});
console.log(doubleNumbers);

// Eso nos imprimirá [2,4,6]

```

filter()

Esta función nos permite crear un sub-arreglo que contenga todos los elementos que cumplen cierta condición. También nos devuelve un arreglo.

```
const numbers = [1, 2, 3];
const filteredNumbers = numbers.filter((num) => {
  return num >= 2;
});
console.log(filteredNumbers);

// Eso nos imprimirá [2,3]

```

reduce()

Es una función que justamente nos permite reducir un arreglo de valores a un solo valor (útil para realizar sumatorias, o promedios, por ejemplo). Nos devuelve un valor.

```
const numbers = [1, 2, 3];
const sumNumbers = numbers.reduce((numb, el) => {
  return numb + el;
}, 0);
console.log(sumNumbers);

// Eso nos imprimirá 6
```

- *concat()*: se utiliza para concatenar dos arreglos.
- *slice()*: se utiliza para crear nuevos arreglos cortando en una posición de inicio y de fin al anterior.
- *find()*: devuelve el **primer elemento encontrado que cumpla una condición** en la lista de elementos.

Fecha	Versionado actual	Autor	Observaciones
08/03/2022	1.0.0	Gabriel Golzman	Primera versión
25/11/2024	1.1.0	Gabriel Golzman	Revisión