
UTN
TÉCNICO UNIVERSITARIO EN SISTEMAS INFORMÁTICOS
PROGRAMACIÓN AVANZADA II

Nosotros

Comisiones y datos de contacto

Prof. Juan José Azar
juan.azar@gmail.com

Ayte. Ignacio Casales
casales.Ignacio@gmail.com

RESPETA A TUS COMPAÑEROS

ESTUDIA CON INTENSIDAD

JÚNTATE CON LOS QUE CONSIDERES MEJORES Y MAS EXPERIMENTADOS

“ESTO FUNCIONA” NO ES DONDE TE DETIENES, ES DONDE COMIENZAS

CONÉCTATE CON LOS DEMÁS, TRABAJA EN EQUIPO

APRENDE TÉCNICAS, NO HERRAMIENTAS

PERMANECE CURIOSO

PIDE AYUDA Y OFRECE AYUDA

AMA LO QUE HACES

DIVIÉRTETE!

Sobre esta materia

Cómo aprobarla?

· Aprobar los 2 (dos) parciales o recuperatorios con 6 (seis) o mas puntos en cada uno.

Cómo son los Recuperatorios?

· Hay una única instancia al final de la cursada.

· Todos los parciales pueden recuperarse en esta instancia.

Cómo es la Aprobación Directa?

· Todos los parciales deben ser aprobados con 8 (ocho) o mas puntos en cada uno.

· 75% de asistencia es requerida.

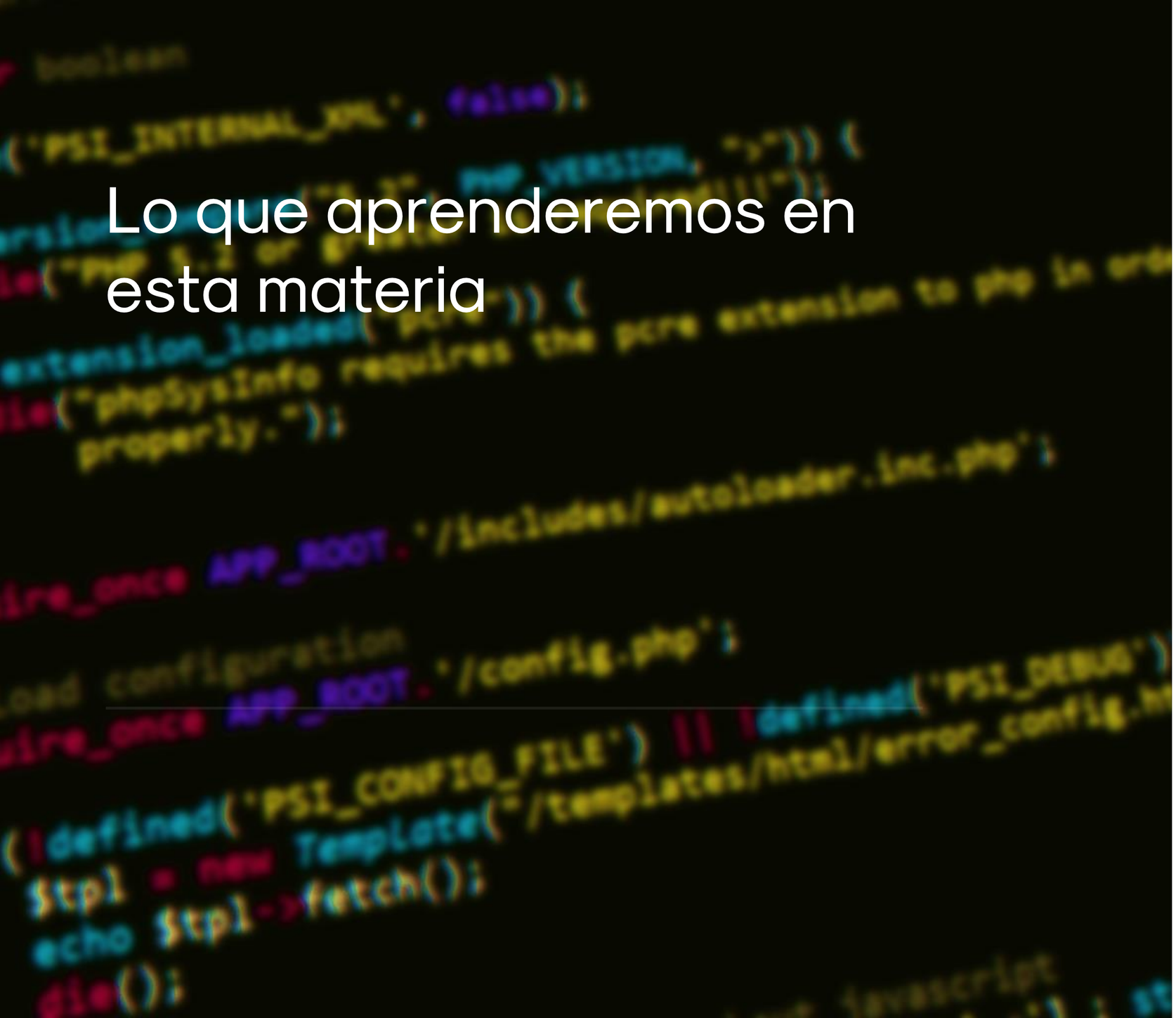
· El alumno podrá tener sólo 1 (uno) recuperatorio.

· El alumno no podrá tener materias correlativas sin aprobar.

Se puede repetir un Parcial?

· Si, solo 1 (una) chance para repetir un Parcial con el objetivo de obtener un puntaje mayor, pero el alumno no deberá tener recuperatorios previos.

· La nota quedará firme aunque ésta sea menor que la de la primera instancia.



Lo que aprenderemos en esta materia

FUNDAMENTALS

JavaScript, CSS, HTML5, DOM

WEB UI FRAMEWORK

Angular, NPM

INTEGRACION

Integración Angular Application con REST API

HTML Basics

HTML es el lenguaje de etiquetado para Páginas Web.

Con HTML puedes crear tu propio Web Site.

Qué es HTML?

HTML es el lenguaje de etiquetado para la creación de Páginas Web.

- HTML viene de Hyper Text Markup Language
- HTML describe la estructura de una Página Web
- HTML consiste en una serie de elementos
- Los elementos HTML le dicen al browser cómo mostrar el contenido
- Los elementos HTML están representados por etiquetas (**tags**)
- Las HTML tags etiquetan partes del contenido como por ejemplo "heading", "paragraph", "table", etc.
- Los browsers no muestran las tags HTML, pero las usan para renderizar el contenido de la página

index.html

```html
<!DOCTYPE html>
<html>
    <head>
        <title>HTML fundamentals</title>
    </head>
    <body>

        <h1>My First Heading</h1>

        <p>My first paragraph.</p>

        <!-- This is a comment -->

    </body>
</html>
```

Estructura básica de un documento HTML

- **<!DOCTYPE html>** define que el documento es de tipo HTML5
- **<html>** es el elemento raíz de una página HTML
- **<head>** contiene meta información sobre el documento
- **<title>** especifica el título del documento (title bar)
- **<body>** posee el contenido visible de la página
- **<h1>** define un encabezado grande
- ** ** define un párrafo
- **<!–** y **-->** apertura y cierre de código comentado

HTML Tags: **<tagname>**contenido…**</tagname>**

- Generalmente van de a pares, ej.: ** ** y ** **
- Cada una se conoce como *opening tag* y *closing tag*

```html
<body>
    <h1>This is heading 1</h1>
    <h2>This is heading 2</h2>
    <h3>This is heading 3</h3>

    <a href="https://google.com">This is a link to Google</a>

    <br>

    <img
    src="https://raw.githubusercontent.com/JuanAzar/UTN-LabIV/master/Common/Assets/logo-landscape.jpg"
    alt="logo-landscape image" title="UTN Mar del Plata" width="340" height="68">
</body>
```

Estructura básica de un documento HTML

<h1> al **<h6>** definen encabezados de mayor a menor tamaño respectivamente

- **<a>** define un hipervínculo. El atributo **href** indica la URL de destino y entre las tags se especifica el texto a mostrar en el hipervínculo
- **
** salto de línea en HTML, no lleva closing tag
- **<img>** inserta una imagen en el documento, no lleva closing tag y posee los siguientes atributos:

  src origen de la imagen, puede ser un archivo local o un archivo remoto

  alt texto alternativo que se muestra cuando la imagen no logra cargarse

  title texto que se muestra al hacer un **hover over** en la imagen

  width y **height** ancho y alto respectivamente para cambiar el tamaño de forma manual. Si no se especifica ninguno, la imagen se muestra en tamaño original

```html
<body>
    <h2>Unordered List example</h2>
    <ul>
        <li>Coffee</li>
        <li>Tea</li>
        <li>Milk</li>
    </ul>

    <h2>Ordered List example</h2>
    <ol>
        <li>Coffee</li>
        <li>Tea</li>
        <li>Milk</li>
    </ol>

    <h2>Description list example</h2>
    <dl>
        <dt>Coffee</dt>
        <dd>- black hot drink</dd>
        <dt>Milk</dt>
        <dd>- white cold drink</dd>
    </dl>
</body>
```

Trabajando con listas

- **`<ul>`** define una lista desordenada
- **`list-style-type`** es una propiedad CSS que define el tipo de viñeta
- **`<ol>`** define una lista ordenada
- **`type`** es un atributo que define el tipo de numeración
- **`<li>`** corresponde a un list item o elemento de lista
- **`<dl>`** define una lista de descripción
- **`<dt>`** corresponde a un término
- **`<dd>`** corresponde a la descripción de un término
- Las listas pueden anidarse entre sí
- Las listas pueden contener otros elementos HTML
- Se pueden usar las propiedades CSS **`float:left`** o **`display:inline`** para mostrar una lista de manera horizontal

```html
<body>
    <b>This text is bold</b>

    <br>

    <strong>This text is strong</strong>

    <br>

    <i>This text is italic</i>

    <br>

    <em>This text is emphasized</em>

    <h2>HTML <small>Small</small> Formatting</h2>

    <h2>HTML <mark>Marked</mark> Formatting</h2>

    <p>My favorite color is <del>blue</del> red.</p>

    <p>My favorite <ins>color</ins> is red.</p>

    <p>This is <sub>subscripted</sub> text.</p>

    <p>This is <sup>superscripted</sup> text.</p>
</body>
```

Dando formato al texto

- **<b>** define texto en negrita
- **** define texto importante
- **<i>** define texto en cursiva
- **** define texto enfatizado
- **<small>** define texto mas pequeño
- **<mark>** define texto resaltado
- **<del>** define texto tachado
- **<ins>** define texto subrayado
- **<sub>** define texto en subíndice
- **<sup>** define texto en superíndice

```
<body>

    <div>Hello World</div>

    <span>Hello World</span>

</body>
```

Elementos de bloque

- `<div>` define una sección en un documento (**block-level**). Este elemento es utilizado a menudo como contenedor de otros elementos HTML.

- `<span>` define una sección en un documento (**inline**). Este elemento es utilizado a menudo como contenedor de texto.

Nota: Ambos elementos se utilizan con atributos de tipo **style**, **class** y **id**.

Un elemento de tipo **block-level** comienza siempre en una nueva línea y ocupa todo el ancho disponible (se extiende hacia la izquierda y hacia la derecha todo lo que pueda). Ej.: **`<address>` `<article>` `<aside>` `<dd>` `<div>` `<dl>` `<dt>` `<footer>` `<form>` `<h1>-<h6>` `<header>` `<hr>` `<li>` `<main>` `<nav>` `<ol>` `<p>` `<section>` `<table>` `<ul>`**

Un elemento de tipo **inline** no comienza en una nueva línea y ocupa sólo el ancho que sea necesario. Ej.: **`<a>` `<b>` `<br>` `<button>` `<cite>` `<code>` `<em>` `<i>` `<img>` `<input>` `<label>` `<q>` `<script>` `<select>` `<small>` `<span>` `<strong>` `<sub>` `<sup>` `<textarea>`**

```html
<body>
    <table>
        <caption>Employee Information</caption>
        <thead>
            <tr>
                <th>Firstname</th>
                <th>Lastname</th>
                <th>Age</th>
            </tr>
        </thead>
        <tbody>
            <tr>
                <td>Jill</td>
                <td>Smith</td>
                <td>50</td>
            </tr>
            <tr>
                <td>Eve</td>
                <td>Jackson</td>
                <td>94</td>
            </tr>
            <tr>
                <td>John</td>
                <td>Doe</td>
                <td>80</td>
            </tr>
        </tbody>
        <tfoot>
            <tr>
                <td colspan="3"> </td>
            </tr>
        </tfoot>
    </table>
</body>
```

Tablas

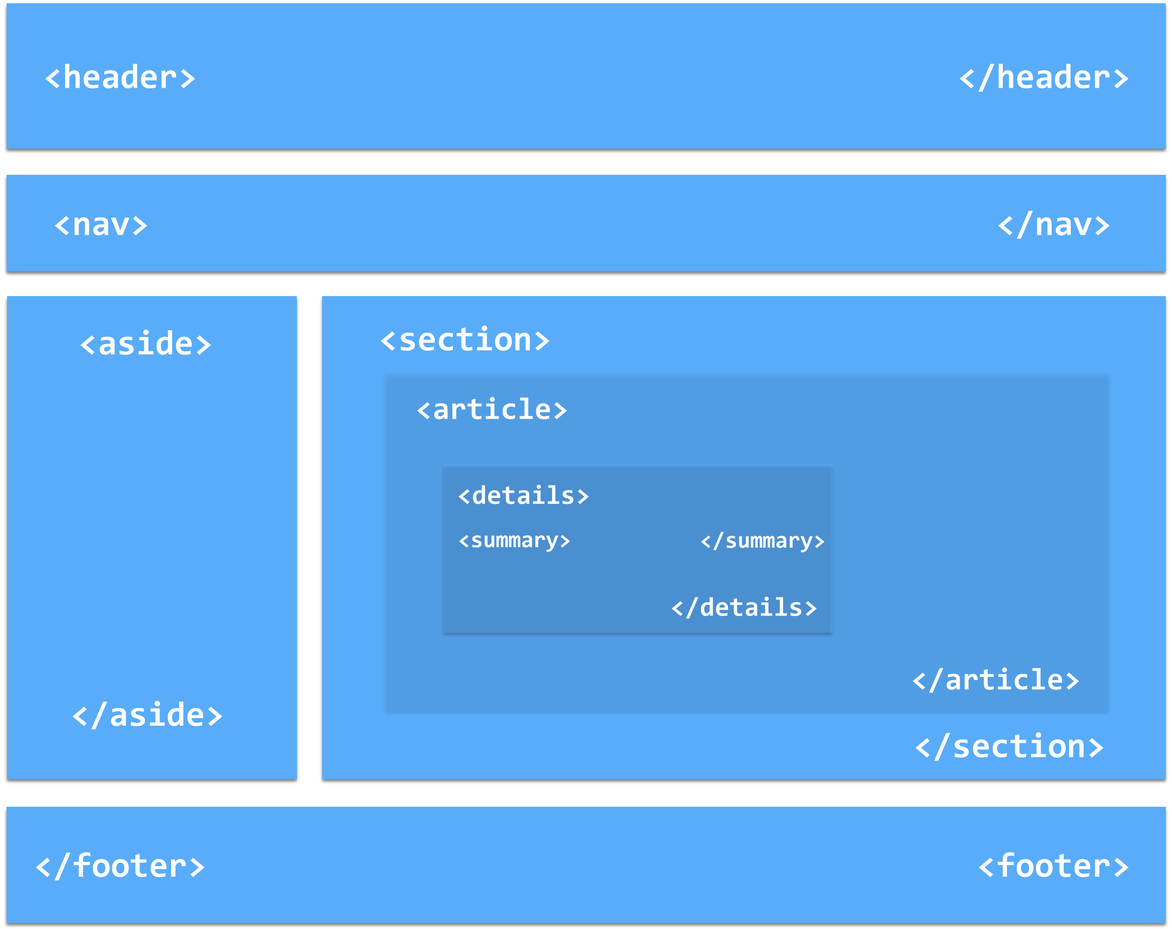
- **`<table>`** define una tabla
- **`<caption>`** define el título de una tabla
- **`<tr>`** define una fila de tabla
- **`<th>`** define una celda de encabezado de tabla
- **`<td>`** define una celda de tabla
- **`<thead>`** agrupa el contenido del **header** en una tabla
- **`<tbody>`** agrupa el contenido del **body** en una tabla
- **`<tfoot>`** agrupa el contenido del **footer** en una tabla

Nota: Los atributos **`colspan`** y **`rowspan`** permiten hacer que una columna o fila abarque mas de una columna o fila respectivamente

```html
<head>
    <title>HTML fundamentals</title>
    <base href="https://raw.githubusercontent.com/JuanAzar/UTN-LabIV/master/Common/Assets/">
    <link rel="stylesheet" href="mystyles.css">
    <meta charset="UTF-8">
    <script>
        function myFunction() {
            document.getElementById("demo").innerHTML = "Hello JavaScript!";
        }
    </script>
    <style>
        body {
            background-color: powderblue;
        }

        h1 {
            color: red;
        }

        p {
            color: blue;
        }
    </style>
</head>
<body>
    <h1>This is a heading</h1>

    <p id="demo">This is a paragraph</p>

    <button onclick="myFunction()">Click here</button>

    <br><br>

    <img src="logo-portrait.jpg" width="230" height="383">
</body>
```

Head

- **`<head>`** define información sobre el documento
- **`<title>`** especifica el título del documento (title bar)
- **`<base>`** define una dirección/destino por defecto para todos los links relativos de la página
- **`<link>`** especifica una relación entre el documento y un recurso externo
- **`<meta>`** define metadata sobre un documento HTML
- **`<script>`** define un script client-side
- **`<style>`** especifica estilos para un documento



Layout

- **<header>** define el encabezado para un documento o sección
- **<nav>** define un contenedor para links de navegación (ej.: menú)
- **<section>** define una sección en un documento
- **<article>** define un artículo independiente y auto contenido
- **<aside>** define contenido que no forma parte del contenido principal (ej.: sidebar)
- **<footer>** define el pie de un documento o sección
- **<details>** especifica detalles adicionales. Permite expandir o colapsar su contenido sin necesidad de agregar lógica adicional (*no soportado en IE y Edge*).
- **<summary>** especifica un encabezado para el elemento **<details>**

Result	Description	Entity Name	Entity Number
	non-breaking space		
<	less than	&lt;	&#60;
>	greater than	&gt;	&#62;
&	ampersand	&amp;	&#38;
"	double quotation mark	&quot;	&#34;
'	single quotation mark	&apos;	&#39;
á	small letter a with accent	&aacute;	&#225;
é	small letter e with accent	&eacute;	&#233;
í	small letter i with accent	&iacute;	&#237;
ó	small letter o with accent	&oacute;	&#243;
ú	small letter u with accent	&uacute;	&#250;
ñ	n tilde	&ntilde;	&#241;
©	copyright	&copy;	&#169;
®	registered trademark	&red;	&#174;

Entidades

- Algunos caracteres son reservados en HTML
- Las entities se utilizan para poder mostrar caracteres reservados en HTML
- Se utiliza **`&entityName`** o **`&#entityNumber`**. Ej.: para mostrar (<) se usa **`<`** o **`&#60;`**
- Ventajas de usar un entity name: Es fácil de recordar
- Desventajas de usar un entity name: Los Browsers pueden no soportar todos los entity names, pero el soporte de entity numbers es muy bueno.

Formularios

```
<body>
    <form action="action.php" method="post">
        <input type="text" name="firstName" placeholder="First Name">
        <input type="text" name="lastName" value="Your Last Name">
        <input type="password" name="password" placeholder="Enter password">
        <br>
        <input type="radio" name="answer" value="yes" checked> Yes
        <input type="radio" name="answer" value="no"> No
        <input type="radio" name="answer" value="na"> N/A
        <br>
        <select name="cars">
          <option value="peugeot">Peugeot</option>
          <option value="chevrolet">Chevrolet</option>
          <option value="ford">Ford</option>
          <option value="volkswagen">Volkswagen</option>
        </select>
        <br>
        <textarea name="comments" cols="50" rows="10"></textarea>
        <br>
        <input type="checkbox" name="vehicle1" value="Bike"> I have a bike<br>
        <input type="checkbox" name="vehicle2" value="Car"> I have a car
        <br>
        <button type="submit">Send</button>
        <button type="reset">Reset</button>
        <button type="button" onclick="alert('Hello World!')">Say Hello</button>
      </form>
</body>
```

- **<form>** define un formulario para recolectar información.
- **<input>** depende del atributo type para determinar cómo se muestra:
  - **text** especifica un input text de una sola línea
  - **password** define un input de tipo password
  - **radio** define un radio button. Aquellos radio buttons con mismo **name** trabajan en conjunto
  - **checkbox** permite elegir cero o mas opciones
- **<select>** define un drop-down list. Contiene varios **<option>**
- **<textarea>** define un input de tipo multilínea
- **<submit>** envía el contenido de los inputs de un form a un **form-handler**
- **<reset>** restablece el contenido de los inputs de un form
- **<button>** botón genérico al cual se le puede añadir acción

ECMAScript - JavaScript

JavaScript es el lenguaje de programación de HTML y la Web inventado en 1995.

ECMAScript es el nombre oficial del lenguaje.

ECMA (*European Computer Manufacturer's Association*) es un estándar para *scripting languages*

A partir de 2015 ECMAScript se llama según el año (Ej.: ECMAScript 2015)

Qué es ECMAScript?

ECMAScript es el lenguaje de scripting para páginas Web.

- Trabaja del lado del cliente (*client-side)*, es decir en el browser.
- Puede cambiar contenido HTML
- Puede cambiar valores de atributos HTML
- Puede cambiar estilos CSS
- En síntesis puede cambiar cualquier cosa dentro del **DOM** (***Document Object Model***)

ECMAScript Basics

Actualmente ECMAScript se encuentra en la versión 2018.

Veremos las versiones principales con sus diferencias y su sintaxis básica

ECMAScript Versiones

ECMAScript y sus versiones mas relevantes

- **JavaScript ES5(2009)**: Soporte para, **`strict mode`**, **`JSON support`**, **`String.trim()`**, **`Aray.isArray()`**, **`Array iteration methods`**
- **JavaScript ES6 o ECMAScript 2015**: Soporte para **`let`** y **`const`**, **`default parameter values`**, **`Array.find()`**, **`Array.findIndex()`**
- **ECMAScript 2016:** Soporte para **`exponential operator (**)`**, **`Array.prototype.includes`**
- **ECMAScript 2017:** Soporte para **`string padding`**, **`new Object proeprtie`**s, **`async functions`**, **`shared memory`**
- **ECMAScript 2018:** Soporte para **`rest/spread properties`**, **`async itearion`**, **`Promise.finally()`**, agregados a **`RegExp`**

```html
<html>
    <head>
        <title>JavaScript Fundamentals</title>
        <script src="myScript.js"></script>
        <script src="https://www.w3schools.com/js/myScript1.js"></script>
    </head>
    <body>
        <p id="demo"></p>

        <button type="button" onclick="myFunction()">Say Hello</button>

        <script>
            var x, y, z;
            x = 5;
            y = 6;
            z = x + y;

            document.getElementById("demo").innerHTML =
            "The value of z is " + z + ".";

            function myFunction()
            {
                document.getElementById("demo").innerHTML =
                "Hello World!.";
            }

            var intValue = parseInt("1234");
            var floatValue = parseFloat("12.34");
            var parseError = parseInt("John");

            //Single line Commented

            /*
            Multiline commented
            block
            */
        </script>
    </body>
</html>
```

Sintaxis básica de JavaScript

- Debe ser insertado entre tags **<script>** y **</script>**
- Las funciones JavaScript pueden ser llamadas a través de **eventos**, por ejemplo cuando un usuario hace click en un botón
- Los scripts pueden ir dentro del **<head>** o el **<body>** y se pueden colocar todos los scripts que queramos
- Los scripts pueden colocarse en archivos externos. Esto mejora entre otras cosas que las páginas carguen más rápido al ser cacheados. Ej.: **myScript.js**
- **getElememntById()**: permite acceder a un elemento HTML a través de su **id** y luego ser manipulado.
- Se puede comentar una línea con **//** y en bloque con **/* */**
- Las variables pueden contener letras, números, underscores (_) y signos de dólar $
- Las variables no pueden comenzar con un número y son case sensitive
- Es case sensitive
- Una variable que no tiene valor, tendrá el valor **undefined**
- **parseInt() y parseFloat()** permiten convertir texto en **integer** o **float**

```html
<html>
    <head>
        <title>JavaScript Fundamentals</title>
    </head>
    <body>
        <h1>My First Web Page</h1>
        <p>My First Paragraph</p>

        <p id="demo"></p>

        <script>
            document.getElementById("demo").innerHTML = 5 + 6;

            document.write(5 + 6);

            console.log("Logging some data");
        </script>

        <button type="button" onclick="document.write('This is JavaScript!')">Write</button>
        <button type="button" onclick="window.alert('This is JavaScript!')">Alert</button>
    </body>
</html>
```

Output

- **`innerHTML:`** utilizando **`getElementById()`** podemos acceder a un elemento HTML y con la propiedad **`innerHTML`** acceder a su contenido HTML. Esta es una manera tradicional de mostrar información
- **`document.write()`**: permite mostrar contenido, se usa mas comúnmente para propósitos de testing. Si usamos **`document.write()`** luego de que el documento HTML se haya cargado, se elimina todo el HTML existente.
- **`window.alert():`** Permite disparar un **`alert box`** para mostrar información.
- **`console.log():`** es utilizado para debugging y permite mostrar por consola determinada información. Es muy utilizado en el proceso de desarrollo.

Operadores Aritméticos

Operador	Nombre	Ejemplo	Resultado
+	Suma	x + y	Suma de x e y
-	Resta	x - y	Diferencia entre x e y
*	Multiplicación	x * y	Producto de x e y
/	División	x / y	Cociente de x dividido y
%	Módulo	x % y	Resto de x dividido y
**	Potenciación	x ** y	Resultado de elevar x a la y potencia

Operadores de Asignación

Asignación	Equivalente	Descripción
x = y	x = y	La variable de la izquierda recibe el valor de la expresión de la derecha
x += y	x = x + y	Suma de x e y con asignación en x
x -= y	x = x - y	Resta entre x e y con asignación en x
x *= y	x = x * y	Multiplicación de x e y con asignación en x
x /= y	x = x / y	Cociente x dividido y con asignación en x
x %= y	x = x % y	Resto de x dividido y con asignación en x

Operadores de Comparación

Operador	Nombre	Ejemplo	Resultado
==	Igualdad	x == y	Retorna true si x es igual a y
===	Identidad	x === y	Retorna true si x es igual a y, y además son del mismo tipo
!=	Desigualdad	x != y	Retorna true si x no es igual a y
<>	Desigualdad	x <> y	Retorna true si x no es igual a y
!==	No Identidad	x !== y	Retorna true si x no es igual a y, o si no son del mismo tipo
>	Mayor	x > y	Retorna true si x es mayor que y
<	Menor	x < y	Retorna true si x es menor que y
>=	Mayor o igual	x >= y	Retorna true si x es mayor o igual que y
<=	Menor o igual	x <= y	Retorna true si x es menor o igual que y

Operadores de Incremento y Decremento

Operador	Nombre	Descripción
++x	Pre-incremento	Incrementa x en 1, luego retorna x
x++	Post-incremento	Retorna x, luego incrementa x en uno
--x	Pre-decremento	Decrementa x en 1, luego retorna x
x--	Post-decremento	Retorna x, luego decrementa x en uno

Operadores Lógicos

Operador	Nombre	Ejemplo	Resultado
`&&`	And	`x && y`	True si `x` e `y` son true
`||`	Or	`x || y`	True si `x` o `y` es true
`!`	Not	`!x`	True si `x` no es true

Operadores de Strings

Operador	Nombre	Ejemplo	Resultado
`+`	Concatenación	`txt1 + txt2`	Concatenación de `txt1` y `txt2`
`+=`	Asignación de concatenación	`txt1 += txt2`	Agrega `txt2` a `txt1`

Operadores de Tipo

Operador	Descripción
`typeOf`	Retorna el tipo de una variable
`instanceOf`	Retorna true si un objeto es una instancia de un tipo específico

```
<script>
    var length = 16;

    var lastName = "Johnson";

    var person = { firstName :"John", lastName : "Doe"};

    //Output: 16Volvo
    var a = "16" + "Volvo";
    console.log(a);

    //Output: 16Volvo
    var b = 16 + "Volvo";
    console.log(b);

    //Output: Volvo16
    var c = "Volvo" + 16;
    console.log(c);

    //Output: 20Volvo
    var d = 16 + 4 + "Volvo";
    console.log(d);

    //Output: Volvo164
    var e = "Volvo" + 16 + 4;
    console.log(e);

    var carName1 = "Volvo XC60";
    var carName2 = 'Volvo XC60';

    var answer1 = "It's alright";
    var answer2 = "He is called 'Johnny'";
    var answer3 = 'He is called "Johnny"';
</script>
```

Tipos de Datos

- Sumando número y string, se tratará como la concatenación de dos strings
- Concatenando un string a un número, se tratará como la concatenación de dos strings
- JavaScript evalúa las expresiones de izquierda a derecha, por lo tanto diferentes secuencias pueden producir diferentes resultados.
- Sumar dos números y un string, producirá la suma algebraica y posteriormente la concatenación
- Concatenar un string a la suma de dos números, producirá la concatenación de los tres términos
- Los strings pueden declararse con comillas simples o dobles
- Se pueden poner comillas simples dentro de dobles y viceversa

```html
<body onload="myFunction1()">
    <script>
        function myFunction1() {
            alert("Hello World!")
        }

        function myFunction2() {
            console.log("Hello World!");
        }
    </script>

    <p onmouseover="myFunction1()">I love JavaScript!! Best language ever!!</p>

    <form onsubmit="myFunction1()" onreset="myFunction2()">
        KeyUp: <input type="text" name="a" onkeyup="myFunction1()"/>
        <br><br>
        Change: <input type="text" name="b" onchange="myFunction1()"/>
        <br><br>
        Select: <input type="text" name="c" onselect="myFunction1()"/>
        <br><br>
        Focus: <input type="text" name="d" onfocus="myFunction2()"/>
        <br><br>
        Blur: <input type="text" name="d" onblur="myFunction2()"/>
        <br><br>
        <button type="button" onclick="myFunction1()">Click</button>
        <button type="submit">Submit</button>
        <button type="reset">Reset</button>
        <br><br>
        <span>This is a span with an onclick event bound to a function</span>
        <div id="myDiv">This div has a onclick function bound to it</div>
    </form>
    <script>
        document.getElementById("myDiv").onclick = myFunction1;

        var element = document.getElementsByTagName('span')[0];
        element.addEventListener('click', myFunction1, false);
    </script>
</body>
```

Funciones y Eventos

- Las funciones pueden llamarse entre sí o al producirse un evento
- Un evento detecta la acción de un usuario y permite disparar una acción determinada
- Tipos de manejadores de eventos: en línea, propiedad y método
  `AddEventListener()`
- Eventos comunes:

  `onLoad:` Terminar de cargar una página o frame (entrar)

  `onMouseOver:` Pasar el mouse por encima de un elemento

  `onMouseOut:` Quitar el mouse de encima del elemento

  `onMouseMove:` Mover el mouse sobre el documento

  `onKeyUp:` Presionar una tecla

  `onClick:` Hacer click con el mouse

  `onChange:` Modificar texto en un control de edición. Sucede al perder el foco

  `onSelect:` Seleccionar texto en un control de edición

  `onFocus:` Situar el foco en un control

  `onBlur:` Perder el foco un control

  `onSubmit:` Enviar un formulario

  `onReset:` Restablecer un formulario

Referenciando elementos

```
<script>
    function getById() {
        document.getElementById('myTextBox').value = 'Hello World!';
    }

    function getByName() {
        document.getElementsByName('firstName')[0].value = 'Hello World';
    }

    function getBygetByClassName() {
        document.getElementsByClassName('formClass')[0].value = 'Hellow World!';
    }

    function getByTagName() {
        var htmlCollection = document.getElementsByTagName('span');

        var elements = Array.from(htmlCollection);

        elements.forEach(function(element) {
            element.innerHTML = 'Hello World!';
        });
    }

    function getBySelector() {
        document.querySelector('h1, h2').innerHTML = 'Hello World';
    }

    function getBySelectorAll() {
        var htmlCollection = document.querySelectorAll('h2, h3');

        var elements = Array.from(htmlCollection);

        elements.forEach(function(element) {
            element.innerHTML = 'Hello World!';
        });
    }
</script>
```

- **`getElementById():`** Obtiene un elemento a través de su id. Los id deberían ser únicos para cada elemento, pero si hubiese mas de uno con el mismo id, este método retorna el primero
- **`getElementsbyName():`** Obtiene una colección de elementos que tienen el **`name`** especificado. Se puede usar la propiedad **`length`** para determinar la cantidad de elementos devueltos y poder iterar sobre estos
- **`getElementsByClassName():`** Obtiene una colección de elementos que tienen el **`name`** especificado. Se puede usar la propiedad **`length`** para determinar la cantidad de elementos devueltos
- **`getElementsByTagName():`** Obtiene una colección de elementos que corresponden al **`tag`** especificado. Se puede usar la propiedad **`length`** para determinar la cantidad de elementos devueltos. Si se especifica el parámetro '*' se retorna todos los elementos del documento
- **`querySelector():`** Obtiene el primer elemento que coincide con el selector CSS especificado. Se pueden seleccionar elementos HTML a través de **`id`**, **`class`**, **`type`**, **`attribute`**, **`attribute values`**, etc.
- **`querySelectorAll()`**: Similar al anterior, solo que en lugar de retornar la primer ocurrencia, retorna todas

```html
<p id="demo"></p>
<script>
    var cars = ['Saab', 'Volvo', 'BMW'];
    var fruits = new Array('Apple', 'Orange', 'Strawberry');

    //Output: Saab
    document.getElementById("demo").innerHTML = cars[0];

    cars[0] = "Peugeot";

    //Output: Peugeot,Volvo,BMW
    document.getElementById("demo").innerHTML = cars;

    //Output: object
    document.getElementById("demo").innerHTML = typeof(cars);

    cars[3] = { firstName: 'John', lastName: 'Doe' };

    //Output: Peugeot,Volvo,BMW,[object Object]
    document.getElementById("demo").innerHTML = cars;

    //Outputs: 3
    console.log(fruits.length);

    //Outputs: Strawberry
    console.log(fruits[fruits.length - 1]);

    fruits.push('Banana');
    fruits[fruits.length] = 'Lemon';

    for(i = 0; i < fruits.length; i++) {
        console.log(fruits[i]);
    }

    fruits.forEach(fruit => {
        console.log(fruit);
    })

    //JavaScript does not support associative arrays
    //This is an object not an array
    var person = [];
    person["firstName"] = "John";
    person["lastName"] = "Doe";
    person["age"] = 46;
</script>
```

Arrays

- Los arreglos podemos declararlos de la forma **`var array_name = [item1, item2, …]`** o bien **`var array_name = new Array(item1, item2, …)`**
- Para acceder a un array indicamos la posición del elemento deseado. Ej.: **`array_name[0]`**
- Si deseamos acceder al array completo podemos referenciar su nombre directamente
- Un array puede contener elementos que sean objetos
- En JavaScript los arrays son un tipo especial de objetos, de hecho **`typeof(array_name)`** retorna **`Object`**
- La propiedad **`length`** nos retorna la cantidad de elementos que posee el array
- Podemos iterar sobre un array con **`for()`** o **`forEach()`**
- Para agregar un elemento, la forma mas sencilla es **`array_name.push(newItem)`** pero también podemos hacerlo de la forma **`array_name[array_name.length - 1] = newItem`** sin embargo, manejar los índices a mano puede generar “*vacíos*” en el array
- JavaScript no soporta arrays asociativos, los arrays poseen índices numéricos

```
<p id="demo"></p>
<script>
    var fruits = ['Banana', 'Orange', 'Apple', 'Mango'];

    console.log(Array.isArray(fruits));
    if(fruits instanceof Array)
        console.log(true);
    else
        console.log(false);

    document.getElementById('demo').innerHTML = fruits;
    document.getElementById('demo').innerHTML = fruits.toString();
    document.getElementById('demo').innerHTML = fruits.join(' | ');

    var fruit = fruits.pop();

    fruits.push('Lemon');

    var fruit = fruits.shift();

    fruits.unshift('Banana');

    delete fruits[0];

    var items = fruits.splice(2, 0, 'Kiwi', 'Pinapple');

    var items = fruits.splice(0, 1);

    var someFruits = fruits.slice(3);

    fruits.sort();

    fruits.reverse();

    var person1 = { firstName : 'Lewis', lastName : 'Ziggy' };
    var person2 = { firstName : 'John', lastName : 'Doe' };

    var personArray = [person1, person2];
    personArray.sort(function (a, b) {
        return (b.lastName < a.firstName) ? 1 : -1;
    });
</script>
```

Arrays – Métodos

- Para poder reconocer un array, podemos utilizar **`Array.isArray(array_name)`** o **`array_name instanceof Array`**
- **`toString()`** y **`join()`** son equivalentes. **`join()`** permite especificar el separador
- **`pop()`** remueve el último elemento del array y lo retorna
- **`push()`** agrega un elemento al final del array, retorna el length del array
- **`shift()`** remueve y retorna el primer elemento del array. Reordena los índices
- **`unshift()`** agrega un elemento al comienzo del array. Reordena los índices
- Como los arrays son objects podemos utilizar **`delete`** para eliminar un elemento. No recomendado dado que puede provocar “vacíos”
- **`splice()`** puede utilizarse tanto para agregar como para eliminar elementos. El primer parámetro define la posición a partir de la cual se agregarán los elementos, el segundo parámetro define cuántos elementos serán removidos y el resto de los parámetros indica los elementos a agregar. Retorna los elementos removidos
- **`slice()`** retorna un subconjunto de elementos del array. El primer parámetro indica la posición inicial y el segundo parámetro opcional indica el final (no inclusivo)
- **`sort()`** y **`reverse()`** permite ordenar un array de forma ascendente o descendente

```
<script>
    function Person(firstName, lastName)
    {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    var Employee = function (employeeId, firstName, lastName)
    {
        this.employeeId = employeeId;
        this.firstName = firstName;
        this.lastName = lastName;
        this.fullName = function()
        {
            return this.firstName + ' ' + this.lastName;
        }
    }

    Employee.prototype.getEmployeeId = function()
    {
        return this.employeeId;
    }

    var employee = new Employee();
    employee.employeeId = 1;
    employee.firstName = 'John';
    employee.lastName = 'Doe';

    console.log(employee);
    console.log('Full name is: ' + employee.fullName());
    console.log('EmployeeId is: ' + employee.getEmployeeId());
</script>
```

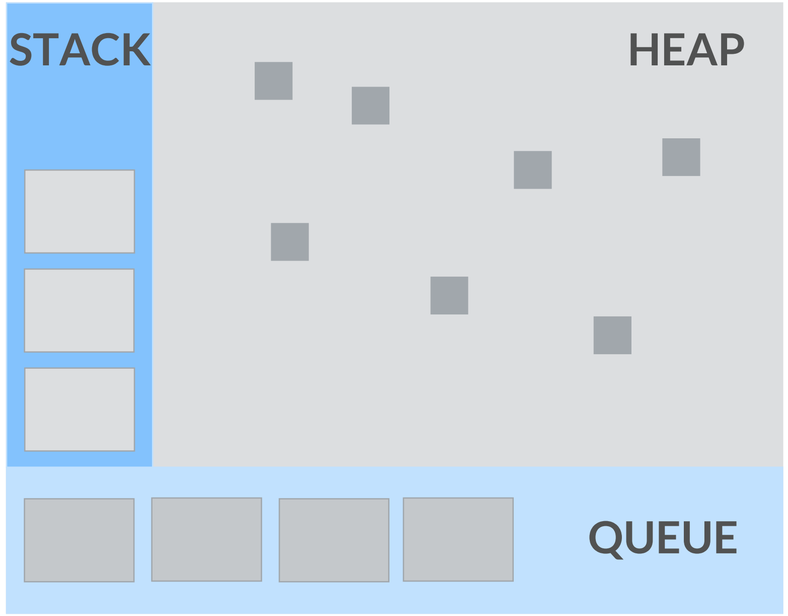
Objetos

- Existen distintas formas de construir objetos en JavaScript. Las dos primeras en este ejemplo sólo permiten crear una única instancia de objetos
- Utilizando **`function Person()`** o **`var Employee = Function()`** nos permite crear una definición de la clase que puede ser reutilizada para múltiples instancias de objetos
- Asignar un objeto a otra variable no genera una copia del objeto, sino que es el objeto en sí (*person3 es person2*)
- Los métodos se declaran como una función asociada a una **`property`** del objeto. Los métodos son funciones guardadas como propiedades de los objetos
- En la definición de una función **`this`** hace referencia al **`owner`** de la función, en este caso el objeto mismo
- Los métodos se invocan utilizando paréntesis (**`employee.fullName()`**), si lo hacemos in los paréntesis nos retorna la definición de la función
- Los objetos en JavaScript heredan propiedades y métodos de un **`prototype`**
- Un **`prototype`** es un objeto asociado por defecto a cada función y objeto en JavaScript y nos permite agregar propiedades y métodos a constructores de objetos

```
<script>
    //ES2015/ES6 Class
    class Person {
        constructor(firstName, lastName, age){
            this.firstName = firstName;
            this.lastName = lastName;
            this._age = age;
        }

        fullName() {
            return this.firstName + ' ' + this.lastName;
        }

        set age(age) {
            this._age = age;
        }

        get age() {
            return this._age;
        }
    }

    class Student extends Person{
        constructor(firstName, lastName, age, studentId) {
            super(firstName, lastName, age);
            this.studentId = studentId;
        }
    }

    var person = new Person('John', 'Doe', 20);
    var student = new Student('John', 'Doe', 20, 123456);

    console.log(person);
    console.log(person.fullName());
    console.log(student);
</script>
```

Objetos y Herencia

- ES2015/ES6 agrega la palabra reservada class que posee una sentencia de tipo constructor lo que nos permite declarar clases de manera similar a otros lenguajes de programación
- Además la forma para declarar un método como **`fullName()`** también resulta más intuitiva respecto a versiones anteriores
- Para el caso de **`setters`** y **`getters`** también proporciona una mejora en su declaración
- Los **`setters`** y **`getters`** en contrapartida con el uso de **`functions`** nos permite acceder a una property de una manera más intuitiva. Ej.: **`person.age`** vs **`person.age()`**
- Cuando veamos **`TypeScript`** veremos una similitud con esta forma de declarar clases y además veremos otros agregados que potenciarán nuestra codificación
- Para Herencia utilizamos la palabra reservada **`extends`** y **`super`** para hacer referencia a la clase base
- Si no se declara un constructor en la clase derivada, el constructor de la clase base se llama por defecto




```
<script>
    function myFunctionA(a){
        var b = 2;

        return myFunctionB(a * b);
    }

    function myFunctionB(x){
        var y = 10;

        return x + y + 20;
    }

    console.log(myFunctionA(20));
</script>
```

Concepto de un programa en ejecución

- **`Stack (Pila):`** Las llamadas a una **`function`** generan un **`stack`** de **`frames`**
- Un **`frame`** encapsula información como el contexto y las variables locales de una función
- **`Heap (Montículo):`** Los objetos son colocados en el **`heap`**. Tal como su nombre lo indica es una vasta región de memoria que generalmente no tiene estructura ni orden
- **`Queue (Cola):`** Un programa JavaScript en ejecución contiene una cola de mensajes, la cual es una listas de mensajes a ser procesados. Cada mensaje está asociado a una función. Cuando la está vacía, un mensaje es sacado de la cola y procesado. Esto es, llamar a una función asociada al mensaje (y por ende crear un **`frame`** en el **`stack`**). El mensaje termina cuando la pila está vacía nuevamante
- Cuando se llama a **`myFunctionA`**, un primer **`frame`** es creado, el cual contiene **`myFunctionA`** argumentos y variables locales. Cuando **`myFunctionA`** llama **`myFunctionB`** otro **`frame`** es creado y colocado por encima del primero en el **`stack`**. Cuando **`myFunctionB`** termina de ejecutarse, el último **`frame`** (**`myFunctionB`**) es quitado del **`stack`** quedando **`myFunctionB`**. Cuando **`myFunctionA`** termina de ejecutarse, el **`stack`** está vacío.

```
<script>
    function doSomething() {
        console.log('this is the start');

        setTimeout(function callBack1() {
            console.log('this is a msg from call back 1');
        });

        console.log('this is just a message');

        setTimeout(function callBack2() {
            console.log('this is a msg from call back 2');
        }, 0);

        console.log('this is the  end');
    }

    doSomething();

    // "this is the start"
    // "this is just a message"
    // "this is the end"
    // "this is a msg from call back 1"
    // "this is a msg from call back 2"
</script>
```

Event Loop

- **Event Loop** posee este nombre por la forma en la que trabaja: Esperar a que llegue un mensaje y procesarlo

- **Run to Completion:** Cada mensaje es procesado completamente antes que cualquier otro mensaje sea procesado. Esto es que cada vez que una función se ejecuta, no puede ser interrumpida hasta completarse.
  La desventaja de este modelo, es que si la función demora mucho en completarse, no se pueden procesar otras interacciones del usuario. Aquí es donde vemos el famoso mensaje *“Un script está tomando mucho tiempo en ejecutarse”*

- **Añadir mensajes:** En los browsers, los mensajes son añadidos cada vez que ocurre un evento y éste tiene un **`EventListener`** asociado. Por ejemplo, si un botón tiene un **`EventListener`** asociado a su evento **`click`**, se generará un mensaje

- **`setTimeOut()`** añade un mensaje al **`queue`** después del tiempo especificado como parámetro. Si no existe un mensaje previo, se procesará en el momento. Caso contrario, tendrá que esperar que los mensajes previos sean completados. Por esta razón el Segundo parámetro indica el tiempo mínimo de espera y no es una garantía de que esto se cumpla

```
<script>
    var timeOutId;

    function mySetTimeOutFunction() {
        timeOutId = setTimeout(alertTimeOut, 3000);
    }

    function alertTimeOut() {
        alert("I Love JavaScript!");
    }

    function myStopSetTimeOutFunction() {
        clearTimeout(timeOutId);
    }

    mySetTimeOutFunction();
    myStopSetTimeOutFunction();


    var intervalId;

    function mySetIntervalFunction() {
        intervalId = setInterval(alertInterval, 3000);
    }

    function alertInterval() {
        console.log("I Love JavaScript!");
    }

    function myStopSetIntervalFunction() {
        clearInterval(intervalId);
    }

    mySetIntervalFunction();
    myStopSetIntervalFunction();
</script>
```

setTimeOut vs setInterval

- **`setTimeOut()`** invoca una función o evalúa una expresión luego de una cantidad de milisegundos especificados
- **`setTimeOut`** se ejecuta una sola vez
- **`setTimeOut`** retorna el id del timer creado que puede utilizarse con **`clearTimeOut()`** para cancelarlo
- **`setInterval()`** invoca una función o evalúa una expresión en intervalos específicos (en milisegundos)
- **`setInterval`** se ejecuta reiteradas veces
- **`setInterval`** retorna el id del timer creado que puede utilizarse con **`clearInterval()`** para cancelarlo
- **`setInterval`** se cancela también al cerrarse la ventana

```
<script>
    console.log("Start message!!");

    var myFirstPromise = new Promise((resolve, reject) => {
        setTimeout(function(){
            resolve("This was successful!!");
            //reject("Rejected!!");
        }, 250);
    });

    myFirstPromise
        .then((successMessage) => {
            console.log("Sweet! " + successMessage);
        })
        .catch((reason) => {
            console.log("Oops! " + reason);
        });

    console.log("End message!!");
</script>
```

Promises

- Una **Promise** representa el ***completion*** o ***failure*** de una operación asincrónica, y su valor de retorno correspondiente
- Recibe un parámetro conocido como **`executor`** que es una función con argumentos **`resolve`** y **`reject`**
- La función **`executor`** se ejecuta inmediatamente pasando los parámetros **`resolve`** y **`reject`** que son también funciones
- Estas funciones al ser llamadas, resuelven o rechazan la promesa respectivamente. El **`executor`** inicia una tarea asincrónica, y una vez que se completa, llama a la función **`resolve`** si se ejecutó correctamente o a la función **`reject`** si ocurrió algún error
- Una Promise actúa como un **`proxy`** para un valor que no se conoce al momento de crear la promise
- Permite que métodos asincrónicos retornen valores como si fueran sincrónicos: En lugar de retornar un valor inmediatamente, el método asincrónico retorna una promesa que proveerá el valor esperado posteriormente en algún momento.
- Posee 3 estados: **`pending`** (estado inicial), **`fullfilled`** (completada satisfactoriamente), **`rejected`** (operación fallida)

```
<script>
  function loadImage(url){
    return new Promise(function(resolve, reject){
      var request = new XMLHttpRequest();
      request.open('GET', url);
      request.responseType = 'blob';
      request.onload = function(){
        if(request.status == 200){
          resolve(request.response);
        }
        else{
          reject(Error('Image couldn\'t be loaded. Error: ' + request.statusText));
        }
      }

      request.onerror = function(){
        reject(Error('Oops!, there was a network error.'));
      }

      request.send();
    });
  }

  var body = document.querySelector('body');
  var image = new Image();
  loadImage('logo-landscape.jpg')
    .then((response) => {
      var imageURL = window.URL.createObjectURL(response);
      image.src = imageURL;
      body.append(image);
    })
    .catch((reason) => {
      console.log(Error(reason));
    })
</script>
```

Promises - XHR

- **`XMLHttpRequest (XHR)`** es una interfaz que permite realizar peticiones **`HTTP`** y **`HTTPS`** obteniendo respuestas en **`XML`**, **`JSON`**, **`HTML`** y **`Plain Text`** entre otras
- Nuestra **`Promise`** genera un objeto **`XMLHttpRequest`** que utilizará un método **`GET`** y cuya respuesta será un **`blob`** (Tipo de dato necesario para recibir una imagen)
- Especificamos qué acciones tomaremos cuando el **`Request`** se ejecute correctamente y cuando se produzca error (**`onload`**, **`onerror`**)
- En caso de que se ejecute correctamente, controlaremos el **`status code`**
- **`Error`** representa un objeto de tipo **`Error`** en tiempo de ejecución
- Con **`querySelector`** accedemos al **`body`** y creamos un objeto **`Image`** cuyo **`source`** será la imagen resuelta de la llamada asincrónica
- En caso de errores utilizamos el **`catch`** de la **`promise`**

```
<script>
  console.log('This is the start');
  var p1 = Promise.resolve(3);
  //var p1 = Promise.reject('Promise 1 rejected');
  var p2 = 42;
  var p3 = new Promise(function(resolve, reject) {
    setTimeout(resolve, 2000, 'This is Promise 3');
  });

  console.log('This is one message');
  var p = Promise.all([p1, p2, p3])
    .then(response => {
      console.log(response);
    })
    .catch(reason => {
      console.log(reason);
    });

  console.log('This is another message');
  console.log(p);

  setTimeout(function() {
      console.log('The stack is now empty');
      console.log(p);
  });
  console.log('This is the end');

  //This is the start
  //This is one message
  //This is another message
  //Promise {<pending>}
  //This is the end
  //The stack is now empty
  //Promise {<resolved>}
  //[3, 42, "This is Promise 3"]
</script>
```

Promises - Methods

- **`Promise.all()`** retorna una sola **`Promise`** resuelta/rechazada en base a un conjunto de Promises pasadas en un **`iterable`**
- No hay un orden específico de ejecución/resolución de sus Promises, por lo tanto no debe generarse dependencia entre ellas
- Retornos: **`already resolved Promise`**, si el iterable está vacío **`Asynchronously resolved Promise`**, si el iterable no contiene Promises **`Pending Promise`** en los demás casos. Esta Promise se resuelve si todas las Promises se resolvieron, o se rechaza cuando una Promise es rechazada. El orden de los valores devueltos es según se pasaron las Promises, sin importar el orden de completitud
- **`Promise.allSettled()`** retorna una sola Promise que se resuelve luego de que todas las Promises dadas se resuelvan o rechacen. Retorna un array con el resultado de cada Promise
- **`Promise.race()`** retorna una Promise con el mismo resultado que la primera Promise pasada por parámetro que se resuelva/rechace primero
- **`Promise.resolve()`** y **`Promise.reject()`** retornan una promesa resuelta/rechazada respectivamente

```
<script>
  function resolveAfter2Seconds() {
    return new Promise(resolve => {
      setTimeout(() => {
        resolve('resolved');
        console.log('Promise is done');
      }, 2000);
    });
  }

  async function asyncCall() {
    console.log('calling');
    var result = await resolveAfter2Seconds();
    console.log(result);
  }

  asyncCall();
  //Outputs: calling
  //waits 2 seconds
  //Outputs: resolved
</script>
```

Async - Await

- **async** define una función asincrónica, la cual devuelve un objeto **AsyncFunction**
- Una función asincrónica es una función que opera de forma asincrónica a través del **evento loop**, usando una **Promise** implícita para retornar el resultado
- Una **async** function puede contener una expresión **await** que pausa la ejecución de la función asincrónica y espera la resolución de la **Promise**. Luego finaliza la ejecución y evalúa el valor resuelto
- El operador **await** sólo es válido dentro de una **async** function
- Mientras la **async function** está pausada, el programa sigue corriendo ya que recibió la **Promise** implícita retornada por la **async function**
- El uso de **async/await** permite simplificar el uso de **Promises** y realizar alguna tarea sobre un conjunto de **Promises**
- En el ejemplo, se al ejecutarse la **async function**, se espera 2 segundos hasta que se resuelva la función y luego muestra el resultado

```
<script>
  var resolveAfter2Seconds = function() {
    console.log("starting slow promise");
    return new Promise(resolve => {
      setTimeout(function() {
        resolve("slow");
        console.log("slow promise is done");
      }, 2000);
    });
  };

  var resolveAfter1Second = function() {
    console.log("starting fast promise");
    return new Promise(resolve => {
      setTimeout(function() {
        resolve("fast");
        console.log("fast promise is done");
      }, 1000);
    });
  };

  var sequentialStart = async function() {
    console.log('==SEQUENTIAL START==');

    // 1. Execution gets here almost instantly
    const slow = await resolveAfter2Seconds();
    console.log(slow); // 2. this runs 2 seconds after 1.

    const fast = await resolveAfter1Second();
    console.log(fast); // 3. this runs 3 seconds after 1.
  }

  sequentialStart();
  // after 2 seconds, logs "slow", then after 1 more second, "fast"
</script>
```

Async - Secuencial

- En este ejemplo se realiza una operación secuencial
- **`secuentialStart()`** se ejecuta y se suspende 2 segundos a causa del primer **`await`**, luego se suspende 1 segundo a causa del segundo **`await`**
- El segundo **`timer`** no se crea hasta que el primero haya sido disparado
- El código finaliza luego de 3 segundos

```
<script>
  var resolveAfter2Seconds = function() {
    console.log("starting slow promise");
    return new Promise(resolve => {
      setTimeout(function() {
        resolve("slow");
        console.log("slow promise is done");
      }, 2000);
    });
  };

  var resolveAfter1Second = function() {
    console.log("starting fast promise");
    return new Promise(resolve => {
      setTimeout(function() {
        resolve("fast");
        console.log("fast promise is done");
      }, 1000);
    });
  };

  var concurrentStart = async function() {
    console.log('==CONCURRENT START with await==');
    const slow = resolveAfter2Seconds();
    const fast = resolveAfter1Second();

    console.log(await slow);
    console.log(await fast);
  }

  var concurrentPromise = function() {
    console.log('==CONCURRENT START with Promise.all==');
    return Promise.all([resolveAfter2Seconds(), resolveAfter1Second()]).then((messages) => {
      console.log(messages[0]);
      console.log(messages[1]);
    });
  }

  concurrentStart();
  // after 2 seconds, logs "slow" and then "fast"

  // wait above to finish
  setTimeout(concurrentPromise, 7000);
  // same as concurrentStart
</script>
```

Async - Concurrente

- En este ejemplo se realiza una operación concurrente
- **`concurrentStart()`** se ejecuta y ambos **`timers`** son creados y luego entran en el **`await`**. Los **`timers`** corren concurrentemente, por lo tanto el código finaliza en 2 segundos en lugar de 3. Es decir en el tiempo que lleva el **`timer`** mas lento
- De todas formas las llamadas a **`await`** corren en serie. Esto es, el segundo **`await`** espera al primero a que finalice. Eso significa que el timer mas rápido será procesado luego del más lento
- Muchas **`async functions`** pueden escribirse utilizando **`Promises`**, solo que las **`async functions`** son un poco menos susceptible a errores cuando debemos hacer ***error handling***

  Si algún **`await`** falla en **`concurrentStart()`**, la excepción se captura automáticamente, la **`async function`** se interrumpe y el **`Error`** se propaga

  Sin embargo para **`concurrentPromise()`**, la función debe encargarse de la **`Promise`** resultante que captura la completitud de la función.

```
<script>
  var resolveAfter2Seconds = function() {
    console.log("starting slow promise");
    return new Promise(resolve => {
      setTimeout(function() {
        resolve("slow");
        console.log("slow promise is done");
      }, 2000);
    });
  };

  var resolveAfter1Second = function() {
    console.log("starting fast promise");
    return new Promise(resolve => {
      setTimeout(function() {
        resolve("fast");
        console.log("fast promise is done");
      }, 1000);
    });
  };

  var parallel = async function() {
    console.log('==PARALLEL with await Promise.all==');

    await Promise.all([
        (async()=>console.log(await resolveAfter2Seconds()))(),
        (async()=>console.log(await resolveAfter1Second()))()
    ]);
  }

  var parallelPromise = function() {
    console.log('==PARALLEL with Promise.then==');
    resolveAfter2Seconds().then((message)=>console.log(message));
    resolveAfter1Second().then((message)=>console.log(message));
  }

  parallel();
  // truly parallel: after 1 second, logs "fast", then after 1 more second, "slow"

  // wait above to finish
  setTimeout(parallelPromise, 13000);
  // same as parallel
</script>
```

Async - Paralelo

- En este ejemplo se realiza una operación en paralelo
- **`parallel()`** se ejecuta utilizando un await Promise.all() y cada job se corre con un await también
- **`parallelPromise()`** se ejecuta utilizando **`Promise.then()`** y produce el mismo resultado solo que se escribe como una función que ejecuta dos **`Promises.then()`**
- En este caso, también se pueden cometer errores de ***error handling*** respecto al uso de **`async functions`**.
  Si no utilizáramos **`await`** o return para el resultado de **`Promise.all()`**, cualquier **`Error`** no sería propagado.
  A su vez mientras que **`parallelPromise()`** parece sencillo, tampoco maneja errores. Para ello deberíamos utilizar un **`return Promise.All([])`**

Angular

Angular es un framework open-source basado en TypeScript
que permite crear aplicaciones web de tipo SPA (*Single Page Application*)

Qué es Angular?

Angular es un Framework basado en TypeScript

- Convierte plantillas en código JavaScript que es optimizado
- Sirve la vista en Node.js, .Net, PHP y otros servidores transformando el código en solo HTML, JavaScript y CSS
- Permite cargar el código de los componentes que solo requiere en el momento, haciendo las cargas mucho más rápidas
- Permite crear vistas de manera muy sencilla gracias a su sintaxis
- Con Angular CLI, se puede desarrollar de manera mucho más rápida

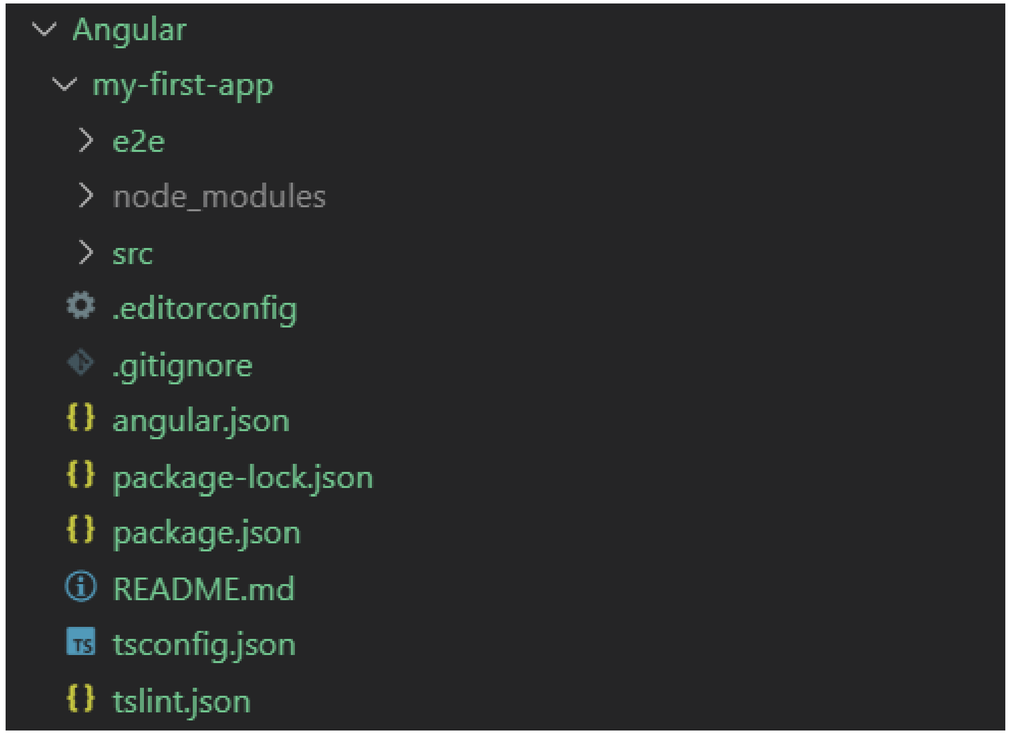
Pre requisitos

Node.js y NPM

- Angular require **`Node.js`**
- **`Node.js`** es un runtime que permite ejecutar **`JavaScript`** del lado del servidor a gran velocidad
- URL oficial de **`Node.js`**: https://nodejs.org/en/
- **`NPM`** es un gestor de paquetes que permite instalar librerías para sr usadas en **`Node.js`**
- **`NPM`** se instala junto con **`Node.js`**
- URL oficial de **`NPM`**: https://www.npmjs.com/

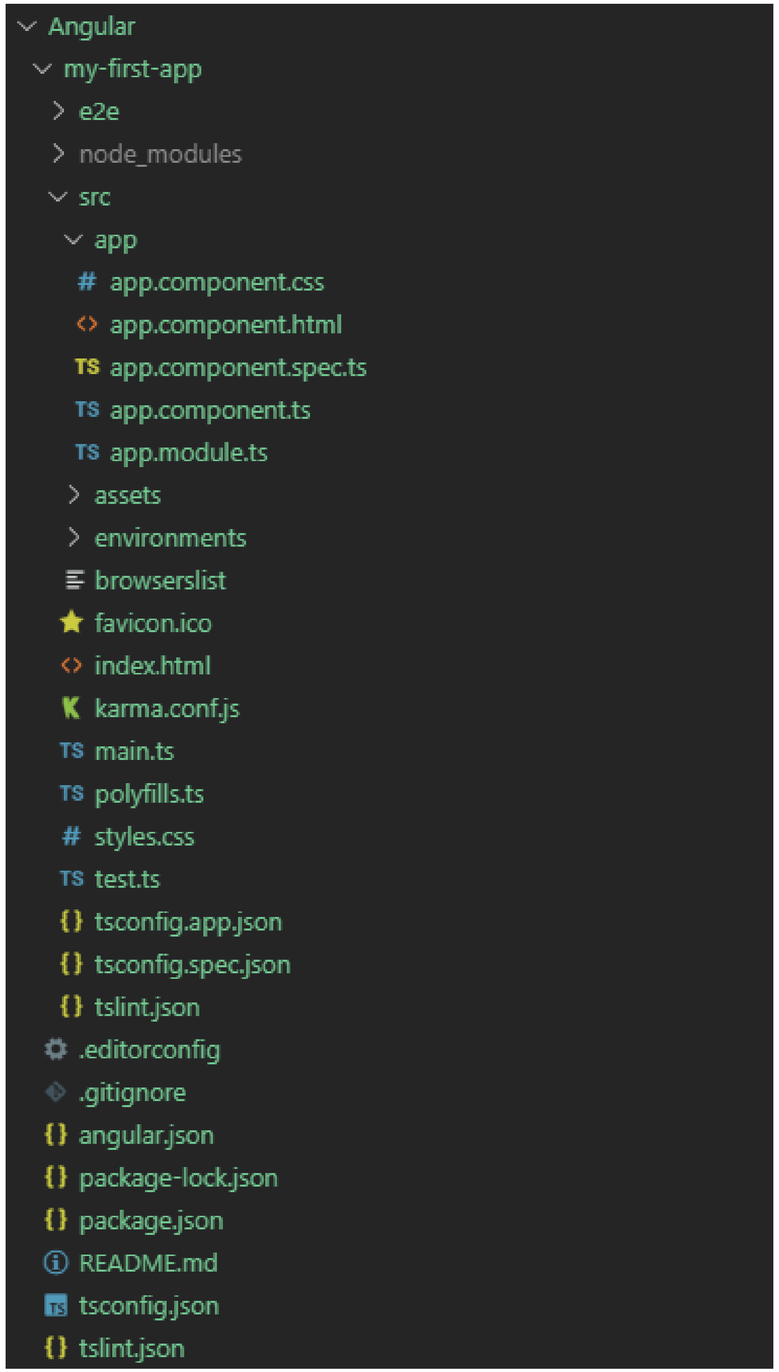
Angular CLI

- **`Angular CLI`** (Command Line Interface) se usa para crear proyectos, generar código por nosotros y realizar varias tareas importantes como testing, bundling y deployment
- Para instalar Angular CLI de manera global ejecutamos el siguiente comando vía NPM: **`npm install-g @angular/cli`**



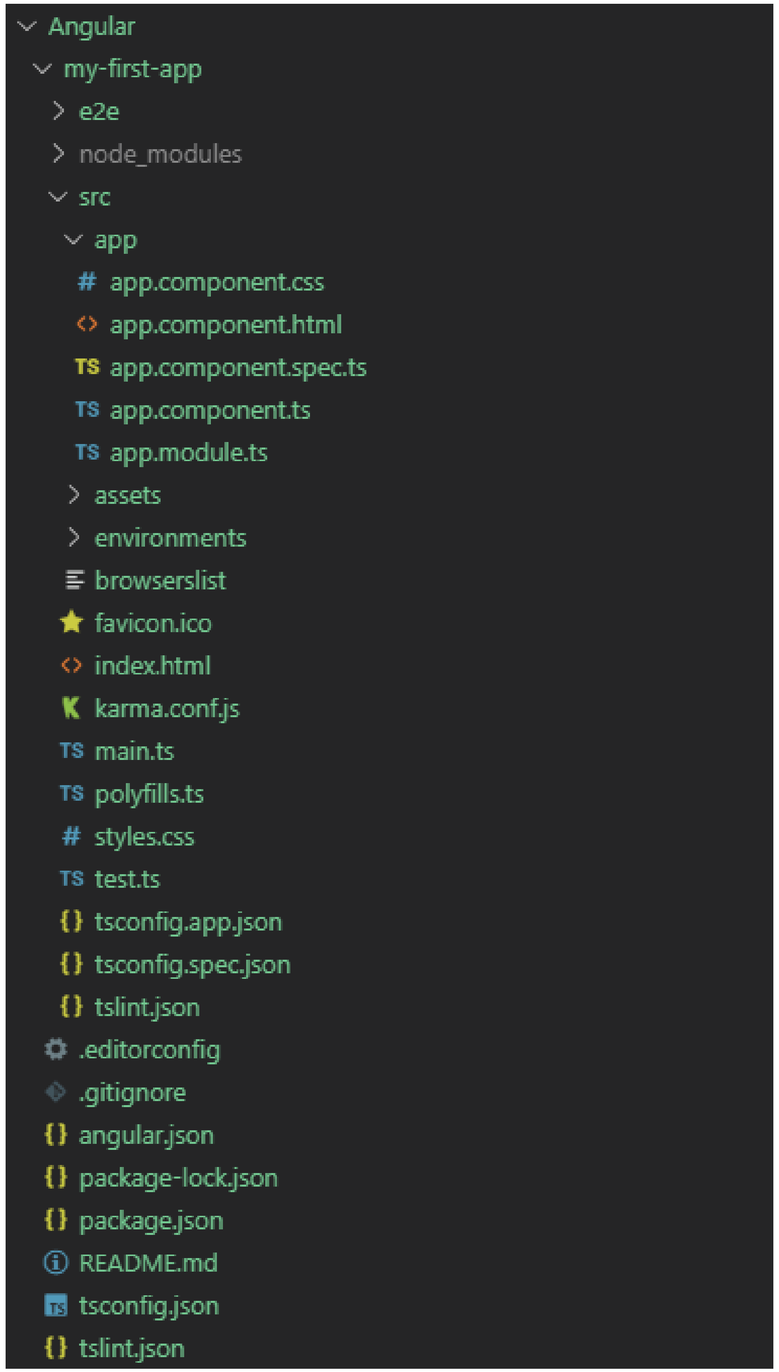
Creando nuestra primera app Angular

- Comenzaremos definiendo una carpeta raíz donde alojaremos nuestros proyectos Angular para mejor organización. En este caso la carpeta raíz se llamará **`Angular`**
- **Angular CLI** nos permite crear un proyecto nuevo con el comando: **`ng new <project-name>`**
- Este comando ejecutará comandos vía NPM que descargarán todas las librerías necesarias para comenzar nuestra primera aplicación
- **Angular CLI** nos crea un nuevo ***workspace*** y una ***Welcome app*** lista para correr
- Para correr nuestra app, debemos ingresar a la carpeta del proyecto y ejecutar el comando: **`ng serve --open`** (*--open o –o es opcional y abre automáticamente el browser en http://localhost:4200*)
- **`ng serve`** lanza una instancia del server Node.js y monitorea los archivos de la aplicación mientras hacemos cambios en ellos



Estructura de archivos

- **`e2e`** contiene archivos para realizar tests de tipo end-to-end junto con archivos para la configuración de tests
- **`src`** es la carpeta raíz donde se alojará nuestro código y su configuración
- **`app`** contiene los archivos de los **`Components`** con la lógica de la aplicación
- **`assets`** contiene imágenes y archivos estáticos que serán copiados tal como están al buildear nuestra aplicación
- **`environments`** contiene configuración para realizar builds en diferentes environments
- **`favicon.ico`** es el ícono usado para el bookmark bar
- **`index.html`** archivo HTML principal que es servido al ingresar a la aplicación. Generalmente no hace falta generar ningún cambio aquí
- **`main.ts`** es el punto de entrada de nuestra app. Compila la app y levanta también el **`AppModule`** (Módulo root de la aplicación) para correr en el browser
- **`polyfills.ts`** provee polyfill scripts para soporte de browsers (*compatibilidad*)
- **`styles.css`** Contiene los estilos de la aplicación
- **`test.ts`** punto de entrada para nuestros test unitarios

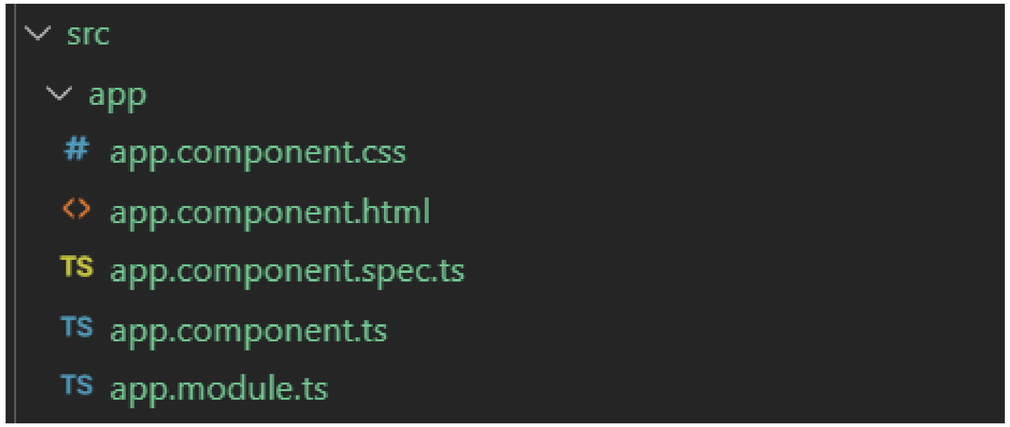


Estructura de archivos

- **`package.json`** es un archivo indica qué dependencias son requeridas en nuestro proyecto. En este caso, qué librearías (que son un conjunto de archivos js y css) son requeridos para que nuestra aplicación funcione
  Este archivo contiene un objeto JSON con todas las dependencias de nuestro proyecto y la versión requerida para las mismas
  Con el comando npm install, NPM lee este archivo y se conecta a los repositorios para bajar las dependencias generando de esta manera una carpeta node_modules que contiene los archivos descargados
  Para el control de versionado y para portabilidad, la carpeta node_modules no es tenida en cuenta

- **`angular.json`** es un archivo que contiene información específica de nuestro proyecto para el build y deploy del mismo
  En este archivo podemos entre otras cosas, setear nuestros archivos de environments que serán utilizados con una configuración específica dependiendo el ambiente en el cual queramos hacer un build o deploy

- **`tslint.json`** es un archivo que permite chequear y controlar la calidad del código escrito en TypeScript. Se pueden definir distintas reglas que nos ayudan a mejorar la lectura, mantenimiento y funcionalidad del código



App Component

- Como vimos anteriormente, la carpeta app contiene la lógica de nuestro proyecto. Los **`Components`**, **`Templates`** y **`styles`** van aquí
- **`app.component.ts`** define la lógica para el componente raíz de nuestra aplicación llamado **`AppComponent`**. La vista asociada a este componente equivale a la vista raíz de nuestra aplicación
- **`app.component.html`** define el template HTML asociado a nuestro **`AppComponent`**
- **`app.component.css`** define el archivo CSS asociado a nuestro **`AppComponent`**
- **`app.component.spec.ts`** define el archivo de Unit Test para nuestro **`AppComponent`**
- **`app.module.ts`** define el root module de nuestra aplicación que indica a Angular como ensamblar nuestra aplicación. A medida que vayamos agregando nuevos componentes, deberemos declararlos en este archivo

```html
<!-- index.html -->
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>MyFirstApp</title>
  <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

```html
<!-- app.component.html -->
<div style="text-align:center">
  <h1>
    Welcome to {{ title }}!
  </h1>
  <img width="300" alt="Angular Logo" src="
</div>
<h2>Here are some links to help you start: </h2>
<ul>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://angular.io/tutorial">Tour of Heroes</
  </li>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://github.com/angular/angular-cli/wiki">
  </li>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://blog.angular.io/">Angular blog</a></h
  </li>
</ul>
```

SPA – Single Page Application

- Nuestro proyecto Angular posee un index.html que es el punto de partida de nuestra aplicación

  Este archivo contiene la declaración básica de un documento HTML solo que nuestro **`<body>`** posee una referencia a una etiqueta especial llamada **`<app-root>`**

  Esta etiqueta referencia a nuestro AppComponent el cual será renderizado en lugar de esta

- **`app.component.html`** en este caso tiene un conjunto de etiquetas base de ejemplo que muestran el logotipo de Angular y contiene links de utilidad

  Reemplazaremos este componente con el HTML que queramos para nuestro proyecto específico

- En conclusión, veremos mas adelante como cada componente posee una etiqueta asociada que indicará a Angular cuando y como renderizar el HTML correspondiente

```html
<!-- index.html -->
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>MyFirstApp</title>
  <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

```html
<!-- app.component.html -->
<div style="text-align:center">
  <h1>
    Welcome to {{ title }}!
  </h1>
  <img width="300" alt="Angular Logo" src="
</div>
<h2>Here are some links to help you start: </h2>
<ul>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://angular.io/tutorial">Tour of Heroes</
  </li>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://github.com/angular/angular-cli/wiki">
  </li>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://blog.angular.io/">Angular blog</a></h
  </li>
</ul>
```

SPA – Single Page Application

- Nuestro proyecto Angular posee un index.html que es el punto de partida de nuestra aplicación

  Este archivo contiene la declaración básica de un documento HTML solo que nuestro **`<body>`** posee una referencia a una etiqueta especial llamada **`<app-root>`**

  Esta etiqueta referencia a nuestro AppComponent el cual será renderizado en lugar de esta

- **`app.component.html`** en este caso tiene un conjunto de etiquetas base de ejemplo que muestran el logotipo de Angular y contiene links de utilidad

  Reemplazaremos este componente con el HTML que queramos para nuestro proyecto específico

- En conclusión, veremos mas adelante como cada componente posee una etiqueta asociada que indicará a Angular cuando y como renderizar el HTML correspondiente

Templates - Introducción

```ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'my first app using Angular';
  name = 'John Doe';
  cars = ['Peugeot', 'Volkswagen', 'Ford', 'BMW', 'Audi'];
  visible = false;
}
```

```html
<!-- app.component.html -->
<div style="text-align:center">
  <h1>
    Welcome to {{ title }}!
  </h1>
</div>
<h2>Hi, my name is: {{ name }}</h2>
<ul>
  <li *ngFor="let car of cars">{{ car }}</li>
</ul>

<span *ngIf="visible">{{ cars }}</span>
```

- La sintaxis de los Templates en Angular, extienden HTML y JavaScript y mediante el uso de **`interpolation`** podemos mostrar en nuestro template, contenido dinámico de manera mas sencilla e intuitiva
- En el ejemplo siguiente vamos a modificar el template y el componente para mostrar un conjunto de propiedades
- El template puede estar en un archivo externo como este caso o estar incluido directamente dentro del **`component.ts`** reemplazando **`templateUrl`** por **`template`** y pegando el código HTML directamente allí encapsulado en **`backticks (`)`**
- Para mostrar contenido podemos usar interpolation. Ej.: **`{{ title }}`**
- La directiva **`ngFor`** nos permite iterar sobre una lista de elementos. En este cas indica que el **`<li>`** y todos sus *Children elements* serán repetidos
- La directiva **`ngIf`** permite insertar o remover un elemento a partir de una condición **`true/false`**. En este caso vemos como la etiqueta **`<span>`** se renderiza o no dependiendo el valor de la propiedad visible

```ts
//models/car.ts
export class Car {
    brand : string;
    model : string;
}
```

```ts
//app.component.ts
import { Component } from '@angular/core';
import { Car } from './models/car';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'my first app using Angular';
  name = 'John Doe';
  cars = [];
  visible = false;

  constructor()
  {
    let car1 = new Car(); car1.brand = 'Peugeot'; car1.model = '508';

    let car2 = new Car(); car2.brand = 'Volkswagen'; car2.model = 'Vento';

    let car3 = new Car(); car3.brand = 'Ford'; car3.model = 'Focus';

    this.cars.push(car1);
    this.cars.push(car2);
    this.cars.push(car3);
  }
}
```

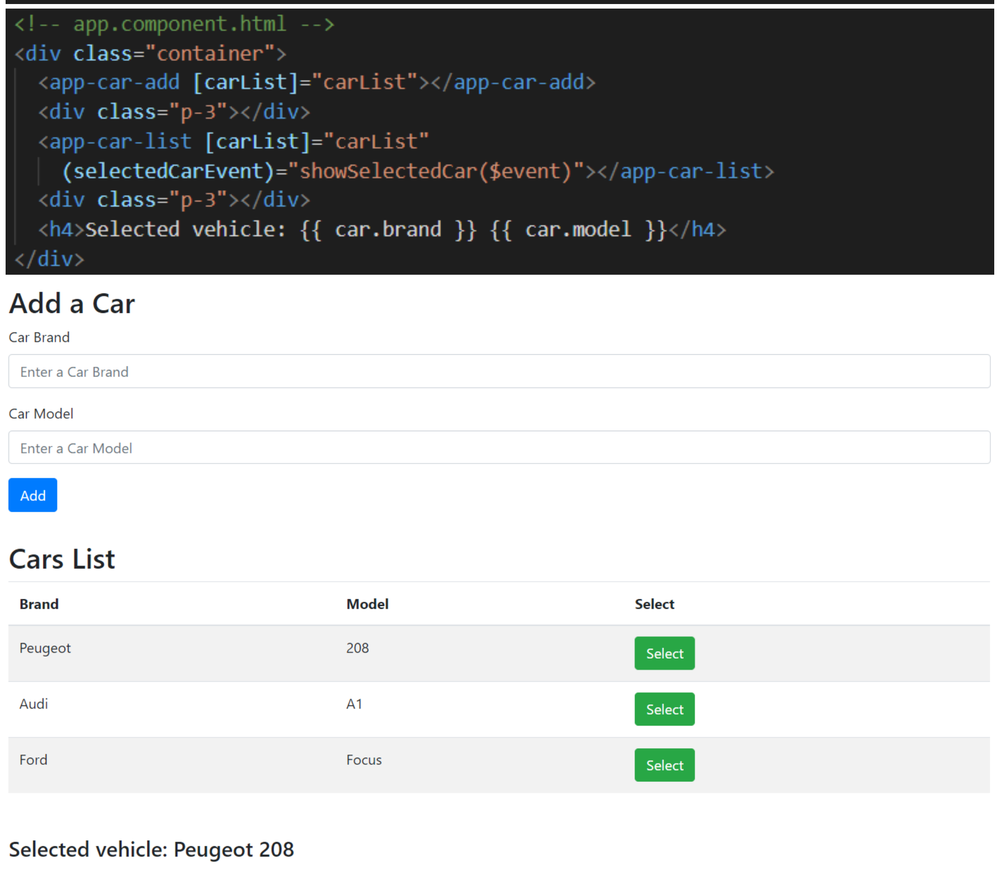
```html
<!-- app.component.html -->
<ul>
  <li *ngFor="let car of cars">{{ car.brand }} {{ car.model }}</li>
</ul>
```

Templates - Introducción

- Con **`Angular CLI`** podemos crear una clase de manera sencilla: **`ng generate class models/car`**
  Esto creará la clase **`Car`** dentro de un folder llamado **`Models`**
- Con la clausula **`import`** referenciamos nuestra clase dentro del **`AppComponent`** y luego en su constructor podemos instanciar un conjunto de objetos que agregaremos a nuestro **`array`** de **`cars`**
- Finalmente modificamos nuestro template para mostrar las dos properties de cada objeto **`Car`** iterado en nuestro array

```ts
//app.component.ts
import { Component } from '@angular/core';
import { Car } from './models/car';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Cars Application Example';
  carList = new Array<Car>();
  car = new Car();

  showSelectedCar(car : Car) {
    this.car = car;
  }
}
```

```html
<!-- app.component.html -->
<div class="container">
  <app-car-add [carList]="carList"></app-car-add>
  <div class="p-3"></div>
  <app-car-list [carList]="carList"
    (selectedCarEvent)="showSelectedCar($event)"></app-car-list>
  <div class="p-3"></div>
  <h4>Selected vehicle: {{ car.brand }} {{ car.model }}</h4>
</div>
```



Compartiendo datos entre Componentes

- **`@Input()`** y **`@Output`**: Permiten compartir datos entre el `parent` y `child components`. `@Input()` es `writable` y `@Output()` es un `observable`
- **`@Input()`** y **`@Output()`** actúan como una API en el child component en el sentido que permiten al parent component comunicarse con este
- **`@Input()`** permite recibir cualquier tipo de dato desde el parent component. Durante la detección de cambios en el valor que el parent component le establece a la property, Angular actualiza automáticamente este valor en el child component
- **`@Output()`** permite al parent component recibir datos desde el child component. En este caso, la property output debe ser de `tipo` **`EventEmitter<>`** que puede tiparse a un tipo específico: ej. **`new EventEmitter<string>()`**
- En este ejemplo se declara en `AppComponent` una property **`carList`** que contendrá los vehículos y una property **`car`** que contendrá el vehículo seleccionado de una lista
- El `app.component.html` pasará el `array` de `Car` usando data binding **`[carList]="carList"`** como una input property
  A su vez tendremos un **método `showSelectedCat($event)`** que capturará el vehículo enviado desde el child component en la output property

```html
<!-- app.component.html -->
<div class="container">
  <app-car-add [carList]="carList"></app-car-add>
  <div class="p-3"></div>
  <app-car-list [carList]="carList"
    (selectedCarEvent)="showSelectedCar($event)"></app-car-list>
  <div class="p-3"></div>
  <h4>Selected vehicle: {{ car.brand }} {{ car.model }}</h4>
</div>
```

```typescript
//car-add.component.ts
export class CarAddComponent implements OnInit {
  @Input()
  carList : Array<Car> = new Array<Car>();
  brand : string;
  model : string;

  constructor() { }

  ngOnInit() {
  }

  addCar()
  {
    let car = new Car();
    car.brand = this.brand;
    car.model = this.model;

    this.carList.push(car);
  }
}
```

```html
<!-- car-add.component.html -->
<form>
  <h2>Add a Car</h2>
  <div class="form-group">
    <label for="brand">Car Brand</label>
    <input type="text" class="form-control"
    name="brand" id="brand" [(ngModel)]="brand" placeholder="Enter a Car Brand">
  </div>
  <div class="form-group">
    <label for="model">Car Model</label>
    <input type="text" class="form-control"
    name="model" id="brand" [(ngModel)]="model" placeholder="Enter a Car Model">
  </div>
  <button type="button" (click)="addCar()" class="btn btn-primary">Add</button>
</form>
```

Compartiendo datos entre Componentes

- **`CarAddComponent`** declara su input property carList a través del decorador **`@Input()`**
- El método **`addCar()`** se dispara en el `click` del botón, instancia un objeto `Car` nuevo y luego lo agrega al array **`carList`** que al ser una `input property` es compartida por el parent component por ser una referencia al array
- **`[(ngModel)]`** nos permite establecer `two-way data binding`, de manera tal que **`brand`** y **`model`** contendrán los valores que el usuario cargue en el formulario
- Finalmente podemos decir que la estructura de uso de @Input() es: <app-car-add [carList]="carList"></app-car-add>, donde:
  app-car-add: es el child component selector
  [carList] :es el target (@Input() property del child)
  “carList”: es el source (property del parent)

```html
<!-- app.component.html -->
<div class="container">
  <app-car-add [carList]="carList"></app-car-add>
  <div class="p-3"></div>
  <app-car-list [carList]="carList"
    (selectedCarEvent)="showSelectedCar($event)"></app-car-list>
  <div class="p-3"></div>
  <h4>Selected vehicle: {{ car.brand }} {{ car.model }}</h4>
</div>
```

```html
<!-- car-list.component.html -->
<h2>Cars List</h2>
<table class="table table-striped">
  <thead>
    <tr>
      <th scope="col">Brand</th>
      <th scope="col">Model</th>
      <th scope="col">Select</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let car of carList">
      <td>{{ car.brand }}</td>
      <td>{{ car.model }}</td>
      <td><button type="button" class="btn btn-success"
          (click)="selectCar(car)">Select</button></td>
    </tr>
  </tbody>
</table>
```

```typescript
//car-list.component.ts
export class CarListComponent implements OnInit {
  @Input()
  carList : Array<Car> = new Array<Car>();

  @Output()
  selectedCarEvent = new EventEmitter<Car>();

  constructor() { }

  ngOnInit() {
  }

  selectCar(car : Car){
    this.selectedCarEvent.emit(car);
  }
}
```

Compartiendo datos entre Componentes

- **`CarAddComponent`** declara su input property carList a través del decorador **`@Input()`**. También se declara una property **`selectedCarEvent`** de tipo **`EventEmitter<Car>`** a través del decorador **`@Output()`**
- La property carList permite obtener el listado de vehículos proveniente de su parent component (AppComponent) y que fueron previamente cargados por otro componente (CarAddComponent)
- CarListComponent a través de su EventEmitter levanta un evento para hacerle saber a su parent component que el valor de la property ha cambiado
- Cuando el usuario selecciona un vehículo de la lista, se dispara el método selectCar(car: Car) que recibe como parámetro el vehículo seleccionado
  Este método emite el evento de la ouput property a través de this.selectedCarEvent.emit(car) y le pasa el vehículo
- AppComponent realiza un event binding a través de (selectedCarEvent)=“showSelectedCar($event) esto conecta el evento emitido por el child component a un método del parent component llamado showSelectedCar(). El $event contiene el dato enviado por el child component
- Finalmente AppComponent muestra los datos del vehículo seleccionado

```
//app-routing.module.ts
import { NgModule }              from '@angular/core';
import { RouterModule, Routes }  from '@angular/router';
import { StudentAddComponent } from './components/student-add/student-add.component';
import { StudentViewComponent } from './components/student-view/student-view.component';
import { StudentListComponent } from './components/student-list/student-list.component';
import { PageNotFoundComponent } from './components/page-not-found/page-not-found.component';

const appRoutes: Routes = [
  { path: 'add', component: StudentAddComponent },
  { path: 'view/:id', component: StudentViewComponent },
  { path: 'list', component: StudentListComponent },
  { path: '', redirectTo: '/list', pathMatch: 'full' },
  { path: '**', component : PageNotFoundComponent }
];

@NgModule({
  imports: [
    RouterModule.forRoot(
      appRoutes,
      { enableTracing: true } // <-- debugging purposes only
    )
  ],
  exports: [
    RouterModule
  ]
})
export class AppRoutingModule {}
```

Angular Routing

- El **`Router`** de Angular interpreta una URL como una instrucción para navegar a una vista específica (**`Component`**) de nuestra aplicación así como también permite pasar parámetros adicionales para presentar un contenido específico.
  El **`Router`** también realiza logs de su actividad en la history del browser permitiendo el go back y forward entre otras cosas.

- El **`Router`** de Angular no forma parte del Angular Core, se encuentra en su propia library **`@angular/Router`** por lo tanto necesitamos impórtalo.

- Es ideal crear un módulo propio para alojar nuestro **`Router`** con todas las rutas que serán detectadas.
  Podemos crear el módulo con el comando **`ng generate module app-routing`**

- El **`Router`** por defecto no posee rutas, debemos agregarlas a través de un arreglo de routes llamado appRoutes.

Angular Routing

```
//app-routing.module.ts
import { NgModule }              from '@angular/core';
import { RouterModule, Routes }  from '@angular/router';
import { StudentAddComponent } from './components/student-add/student-add.component';
import { StudentViewComponent } from './components/student-view/student-view.component';
import { StudentListComponent } from './components/student-list/student-list.component';
import { PageNotFoundComponent } from './components/page-not-found/page-not-found.component';

const appRoutes: Routes = [
  { path: 'add', component: StudentAddComponent },
  { path: 'view/:id', component: StudentViewComponent },
  { path: 'list', component: StudentListComponent },
  { path: '', redirectTo: '/list', pathMatch: 'full' },
  { path: '**', component : PageNotFoundComponent }
];

@NgModule({
  imports: [
    RouterModule.forRoot(
      appRoutes,
      { enableTracing: true } // <-- debugging purposes only
    )
  ],
  exports: [
    RouterModule
  ]
})
export class AppRoutingModule {}
```

```
<!-- app.component.html -->
<nav class="navbar navbar-expand-lg  navbar-dark bg-dark">
     <ul class="navbar-nav ml-auto">
          <li class="nav-item">
               <a class="nav-link" [routerLink]="['/add']" routerLinkActive="active">Add Students</a>
          </li>
          <li class="nav-item">
             <a class="nav-link" [routerLink]="['/list']" routerLinkActive="active">List Students</a>
          </li>
     </ul>
 </nav>

<router-outlet></router-outlet>
```

- Debemos generar un arreglo para configurar las rutas de nuestro **`Router`**.
- Cada ruta mapea una URL con un Component. Ej.:
  `{ path: 'add', component: StudentAddComponent }` para la ruta **`/add`** se cargará el component **`StudentAddComponent`**.
- Para **`{ path: 'view/:id', component: StudentViewComponent }`** el token **`:id`** especifica un parámetro para la ruta, es decir **`view/42`** cargará el componente **`StudentViewComponent`** y el parámetro id contendrá el valor 42. Veremos como recuperarlo a continuación.
- Para **`{ path: '**', component : PageNotFoundComponent }`** el **`valor '**'`** indica una wildcard que hará match con cualquier ruta que no coincide con ninguna de las anteriores y en este caso cargar el componente **`PageNotFoundException`**.
- La directiva **`<router-outlet></router-outlet>`** de la Router library se usa como un componente. Funciona como un placeholder que será utilizado por el Router para mostrar el Component que está asociado a la URL matcheada.
- La directiva **`RouterLink`** en los links permite darle control al **`Router`** sobre estos elementos.
- La directiva **`RouterLinkActive`** permite cambiar la class css de un link que se encuentra activado.

Angular Routing

```typescript
//student-view.component.ts
import { Component, OnInit } from '@angular/core';
import { StudentService } from 'src/app/services/student.service';
import { Student } from 'src/app/models/student';
import { ActivatedRoute } from '@angular/router';

@Component({
  selector: 'app-student-view',
  templateUrl: './student-view.component.html',
  styleUrls: ['./student-view.component.css']
})
export class StudentViewComponent implements OnInit {
  private student: Student;

  constructor(private studentService: StudentService, private route: ActivatedRoute) { }

  ngOnInit() {
    let studentId = Number(this.route.snapshot.paramMap.get('id'));

    this.student = this.studentService.getById(studentId);
  }

}
```

- Finalmente el path de la ruta está disponible a través un servicio del Router inyectado llamado **`ActivatedRoute`**. Este servicio posee mucha información de utilidad para el manejo de nuestras rutas.
- Entre ellas tenemos **`snapshot`** que nos provee el valor inicial de la ruta disparada.
- Luego **`paramMap.get()`** nos permite obtener el/los parámetro/s que acompañaban esa ruta.
- En este ejemplo, para **`view/42`** estaríamos capturando el parámetro id cuyo valor será 42 y esto nos permite en el **`StudentViewComponent`** ir a buscar el **`Student`** con id 42 y mostrarlo.

Angular Routing

```typescript
//student-view.component.ts
import { Component, OnInit } from '@angular/core';
import { StudentService } from 'src/app/services/student.service';
import { Student } from 'src/app/models/student';
import { ActivatedRoute } from '@angular/router';

@Component({
  selector: 'app-student-view',
  templateUrl: './student-view.component.html',
  styleUrls: ['./student-view.component.css']
})
export class StudentViewComponent implements OnInit {
  private student: Student;

  constructor(private studentService: StudentService, private route: ActivatedRoute) { }

  ngOnInit() {
    let studentId = Number(this.route.snapshot.paramMap.get('id'));

    this.student = this.studentService.getById(studentId);
  }

}
```

- Finalmente el path de la ruta está disponible a través un servicio del Router inyectado llamado **`ActivatedRoute`**. Este servicio posee mucha información de utilidad para el manejo de nuestras rutas.
- Entre ellas tenemos **`snapshot`** que nos provee el valor inicial de la ruta disparada.
- Luego **`paramMap.get()`** nos permite obtener el/los parámetro/s que acompañaban esa ruta.
- En este ejemplo, para **`view/42`** estaríamos capturando el parámetro id cuyo valor será 42 y esto nos permite en el **`StudentViewComponent`** ir a buscar el **`Student`** con id 42 y mostrarlo.

```ts
//student.service.ts
import { Injectable } from '@angular/core';
import { Student } from '../models/student';

@Injectable({
  providedIn: 'root'
})
export class StudentService {
  private studentList = new Array<Student>();
  private studentId = 0;

  constructor() { }

  add(student: Student){
    this.studentId++;
    student.studentId = this.studentId;
    this.studentList.push(student);
  }

  getAll(){
    return this.studentList;
  }

  getById(studentId: number){
    let students = this.studentList.filter(student => {
      return student.studentId == studentId;
    });

    return (students.length > 0) ? students[0] : null;
  }
}
```

```ts
//student-view-component.ts
@Component({
  selector: 'app-student-view',
  templateUrl: './student-view.component.html',
  styleUrls: ['./student-view.component.css']
})
export class StudentViewComponent implements OnInit {
  private student: Student;

  constructor(private studentService: StudentService, private route: ActivatedRoute) { }
```

```ts
//student-view-component.ts
@Component({
  selector: 'app-student-view',
  templateUrl: './student-view.component.html',
  styleUrls: ['./student-view.component.css'],
  providers: [ StudentService ] //Generate a specific instance instead of Singleton
})
export class StudentViewComponent implements OnInit {
  private student: Student;

  constructor(private studentService: StudentService, private route: ActivatedRoute) { }
```

Servicios

- Los **`Components`** deben enfocarse solo en presentar información y solicitar datos a través de los inputs. Las tareas de guardar y obtener datos, es responsabilidad de los **`Services`**.
- Los **`Services`** permiten compartir datos entre **`Components`**. Para crear componentes podemos utilizar el **`comando ng generate service services/student-service`**
- El decorador **`@Injectable()`** indicará que ese servicio será utilizado a través de **`Dependency Injection`** por lo tanto Angular se encargará de generar su instancia.
- Aquellos **`Components`** que necesiten hacer uso del **`Service`** deberán declararlo dentro del constructor y lo utilizarán como un atributo más.
- Por defecto el **`Service`** funcionará como un **`Singleton`**, es decir que la misma instancia será compartida por todos los **`Components`** que lo referencien.
- Si deseamos que un **`Service`** funcione como una instancia diferente, es decir no como **`Singleton`**, debemos anotarlo en el **`@Component`** como **`provider`**. Ej.: **`providers: [StudentService]`**

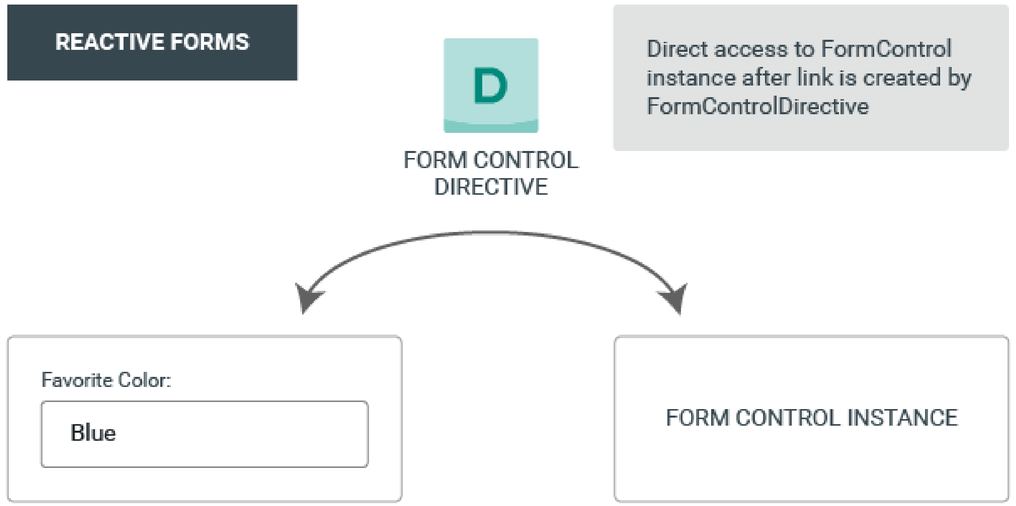
```typescript
//student-async.service.ts
import { Injectable } from '@angular/core';
import { Student } from '../models/student';
import { HttpClient, HttpHeaders } from '@angular/common/http';


@Injectable({
  providedIn: 'root'
})
export class StudentAsyncService {
  private apiURL = 'https://utn-adv.com/api/students/'
  constructor(private http: HttpClient) { }

  getAll(): Promise<any>{
    return this.http.get(this.apiURL)
      .toPromise();
  }

  getById(studentId: number): Promise<any>{
    return this.http.get(this.apiURL + studentId)
      .toPromise();
  }

  add(student: Student): Promise<any>{
    const httpOptions = {
      headers: new HttpHeaders({
        'Content-Type':  'application/json'
      })
    };

    return this.http.post(this.apiURL, student, httpOptions)
      .toPromise();
  }
}
```

```typescript
ngOnInit() {
  //student-list.component.ts
  this.studentService.getAll()
    .then(response => {
      this.studentList = response;
    })
    .catch(error => {
    });
}
```

HttpClient

- El servicio **`HttpClient`** de **`@angular/common/http`** no ofrece una forma simplificada de hacer llamadas de tipo REST.
- Para poder utilizarlo, primero debemos importar **`HttpClientModule`** en nuestro **`AppModule`** y declararlo en nuestro array de **`imports`**.
- Luego ya podremos inyectar el **`HttpClient`** en nuestro servicio.
- El servicio **`HttpClient`** ofrece métodos que representan los diferentes **`verbs`** utilizados en **`REST`**: **`Post, Get, Put, Patch, Delete, etc`**.
- Estos métodos retorna un **`Observable`** el cual abordaremos mas adelante.
- En este ejemplo utilizaremos **`Promises`** para realizar nuestras llamadas asincrónicas a la API que deseamos consumir. El método **`toPromise()`** luego de la llamada el método correspondiente del **`HttpClient`**, transforma transforma la respuesta en una **`Promise`** que será tratada en el **`Component`** que utilice nuestro **`Service`**.
- Para el caso del método **`Put`** del **`HttpClient`**, necesitaremos además especificar los **`headers`** a ser enviados, donde mínimamente deberemos indicar en qué formato enviaremos la información. Usualmente será **`application/json`**.

Reactive Forms - Introducción




```
<!-- app.component.html -->
Favorite Color: <input type="text" [formControl]="favoriteColorControl">
```

```
//app.component.ts
import { Component } from '@angular/core';
import { FormControl } from '@angular/forms';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  favoriteColorControl = new FormControl('');
}
```

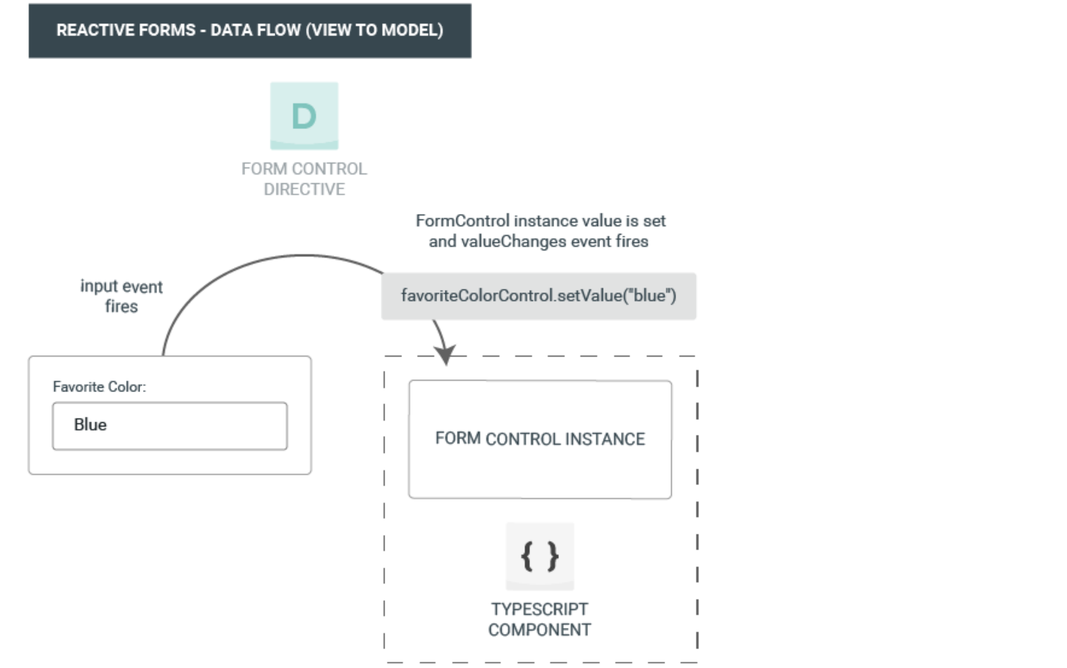
- Angular provee dos mecanismos para manejar formularios y el ingreso de datos de los usuarios: **`Reactive`** y **`Template-Driven forms`**. En esta materia nos centraremos en **`Reactive Forms`** dado que es más robusto, escalable, reusable y testeable.

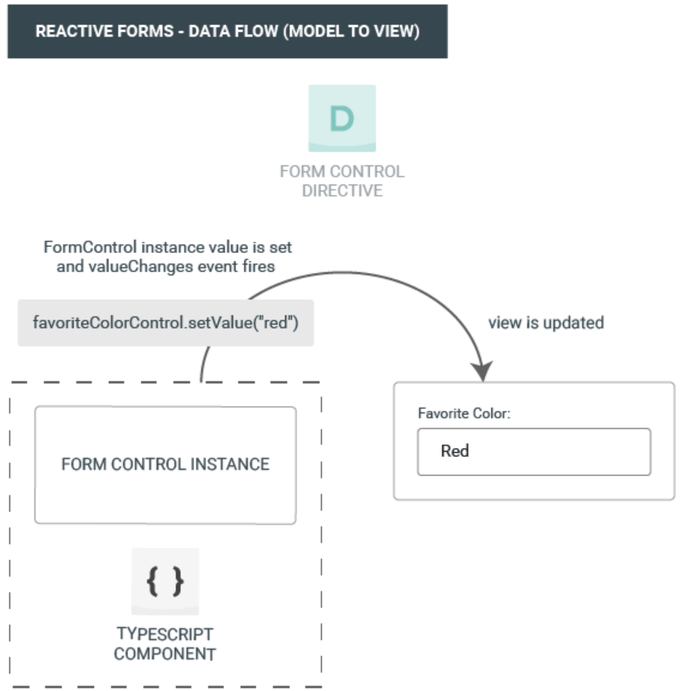
- Reactive Forms utiliza los siguientes building blocks:
  - **`FormControl`**: Trackea el valor y el validation status de un form control.
  - **`FormGroup`**: Trackea el valor y el validation status de una colección de form controls.
  - **`FormArray`**: Trackea el valor y el validation status de un array de form controls.
  - **`ControlValueAccessor`**: Actúa como puente entre un **`Angular FormControl`** y elementos nativos del **`DOM`**.

- Con **`Reactive Forms`** el **`form model`** se define dentro del **`Component`**. La directiva **`FromControlDirective`** linkea un **`FormControl`** a un **`form element`** del HTML utilizando el **`ControlValueAccessor.`**

- En el ejemplo utilizaremos un Component con un **`input field`** sobre el cual utilizaremos **`Reactive Forms`**.

- Debemos importar en nuestro **`AppModule`**, el módulo **`FormsModule`** y declararlo dentro de nuestros **`imports`**





Reactive Forms – Data Flow

- Una de las principales diferencias entre Reactive Forms y **`Template-Driven Forms`** es la forma en la que manejan el flujo de datos de los cambios que realiza el usuario o la aplicación.

- Como vimos, en **`Reactive Forms`** cada form element es linkeado directamente a un **`form model`** (**`FormControl`**).

- Cambios de la View al Model:

  1 - El usuario ingresa un valor en el **`input element`**.

  2 - El **`input`** emite un evento con el value más reciente.

  3 - El **`ControlValueAccessor`** que espera eventos inmediatamente envía el valor al **`FormControl`** asociado.

  4 - El **`FormControl`** emite el nuevo valor a través del **`observable ValueChanges`**.

  5 - Cualquier **`subscriber`** de **`ValueChanges`** recibe el valor emitido.

- Cambios del Model a la View:

  1 – El usuario invoca el método **`favoriteColorControl.setValue()`** que actualiza el valor del **`FormControl`**.

  2 – El **`FormControl`** emite el valor a través del **`observable ValueChanges`**.

  3 – Cualquier **`subscriber`** de **`ValueChanges`** recibe el valor emitido.

  4 – El **`ControlValueAccessor`** del **`input element`**, actualiza su valor.

```html
<!-- student-add.component.html -->
<form [formGroup]="studentForm" (ngSubmit)="onSubmit()">
    <input type="text" formControlName="lastName" class="form-control">
    <input type="text" formControlName="firstName" class="form-control">
    <input type="text" formControlName="dni" class="form-control">
    <input type="email" formControlName="email" class="form-control">
    <input type="text" formControlName="address" class="form-control">

    <div *ngIf="firstName.invalid && (firstName.dirty || firstName.touched)"
        class="alert alert-danger">
        <div *ngIf="firstName.errors.required">
            First Name is required.
        </div>
        <div *ngIf="firstName.errors.minlength">
            First Name must be at least 20 characters long.
        </div>
    </div>
</form>
```

```typescript
//student-add.component.ts
import { Component, OnInit, Input } from '@angular/core';
import { Student } from 'src/app/models/student';
import { FormGroup, FormControl, Validators } from '@angular/forms';

@Component({
  selector: 'app-student-add',
  templateUrl: './student-add.component.html',
  styleUrls: ['./student-add.component.css']
})
export class StudentAddComponent implements OnInit {
  student: Student = new Student();
  studentForm: FormGroup;

  constructor() { }

  ngOnInit(): void {
    this.studentForm = new FormGroup({
      'firstName': new FormControl(this.student.firstName,
        [Validators.required, Validators.minLength(20)],
        [ /* Async Validators */ ]),
      'lastName': new FormControl(this.student.lastName),
      'email': new FormControl(this.student.email),
      'dni': new FormControl(this.student.dni),
      'address': new FormControl(this.student.address)
    });
  }

  onSubmit(){ }

  get firstName() { return this.studentForm.get('firstName'); }
}
```

Reactive Forms – Validtaors

- Con **Reactive Forms**, en lugar de agregar atributos de validación a los input elements, agregamos **Validator functions** directamente al **FormControl**. Angular se encarga de llamar a estas funciones cuando el valor del input cambia.

- Hay dos tipos de Validator Functions:
  - **Sync Validators**: toman una instancia del control e inmediatamente retornan **errores** o **null**. Se pueden pasar como segundo parámetro al instanciar el **FormControl**.
  - **Async Validators**: toman una instancia del control y retornan una **Promise** u **Observable** que luego emitirá **errores** o **null**. Se pueden pasar como tercer parámetro al instanciar el **FormControl**.

- Debemos declarar un **form** con la directiva **[formGroup]** donde indicaremos el **FormGroup** que contiene nuestros **FormControls**.

- Debemos hacer un import de **ReactiveFormsModue** en nuestro **AppModule**.

- La propiedad **invalid** retorna true si alguno de los **Validators** aplicados sobre el **FormControl** retornó errors.

- La propiedad **dirty** retorna true si el valor del **FormControl** fue modificado.

- La propiedad **touched** retorna true si el **FormControl** fue visitado.

Reactive Forms – Custom [Async] Validators

```ts
//custom-validator.ts
import { ValidatorFn, AbstractControl } from "@angular/forms";

export class CustomValidator {
    static forbiddenNames(forbiddenNameRegEx: RegExp): ValidatorFn{
        return (control : AbstractControl): {[key: string]: any} | null => {
            const forbidden = forbiddenNameRegEx.test(control.value);

            return forbidden ? {'forbiddenName': { value: control.value }} : null;
        }
    }
```

```ts
//student-add.component.ts
ngOnInit(): void {
  this.studentForm = new FormGroup({
    'firstName': new FormControl(this.student.firstName,
      [Validators.required, Validators.minLength(20),
        CustomValidator.forbiddenNames(/Juan/)],
      [ /* Async Validators */ ]),
    'lastName': new FormControl(this.student.lastName),
    'email': new FormControl(this.student.email),
    'dni': new FormControl(this.student.dni),
    'address': new FormControl(this.student.address)
  });
}
```

```html
<!-- student-add.component.html -->
<div *ngIf="firstName.invalid && (firstName.dirty || firstName.touched)"
    class="alert alert-danger">
    <div *ngIf="firstName.errors.required">
        First Name is required.
    </div>
    <div *ngIf="firstName.errors.minlength">
        First Name must be at least 20 characters long.
    </div>
    <div *ngIf="firstName.errors.forbiddenName">
        {{ firstName.errors.forbiddenName.value }} is a forbidden name
    </div>
</div>
```

- Podemos crear nuestros propios Validators para realizar validaciones especiales.
- La función **`forbiddenNames`** en este caso, es una *factory* que toma una *regular expression* para detectar un nombre prohibido que es especificado y retorna una validator function.
- En este ejemplo, el forbidden name es **“Juan”**, por lo tanto el **`Validator`** rechaza cualquier **`firstName`** que contenga **“Juan”**.
- La función **`forbiddenNames`** retorna el **`validator function`** configurado. Esta función toma un **`Control Object`** de **`Angular`** y retorna **`null`** si el control es válido o un **`Validation Error Object`** en caso contrario.
- El **`Validation Error Object`** generalmente tiene una property cuyo name es la **`Validation Key (‘forbiddenName’)`** y cuyo valor es un **`dictionary (Key/Value)`** que puede usarse dentro de un error message en el template.
- Los **`Custom Async Validators`**, son similares a los **`Custom Validators`**, pero deben devolver una **`Promise`** u **`Observable`** que luego emitirán **`null`** o un **`Validation Error Object`**.

Reactive Forms – Custom [Async] Validators

```ts
//custom-validator.ts
import { ValidatorFn, AbstractControl } from "@angular/forms";

export class CustomValidator {
    static forbiddenNames(forbiddenNameRegEx: RegExp): ValidatorFn{
        return (control : AbstractControl): {[key: string]: any} | null => {
            const forbidden = forbiddenNameRegEx.test(control.value);

            return forbidden ? {'forbiddenName': { value: control.value }} : null;
        }
    }
```

```ts
//student-add.component.ts
ngOnInit(): void {
  this.studentForm = new FormGroup({
    'firstName': new FormControl(this.student.firstName,
      [Validators.required, Validators.minLength(20),
        CustomValidator.forbiddenNames(/Juan/)],
      [ /* Async Validators */ ]),
    'lastName': new FormControl(this.student.lastName),
    'email': new FormControl(this.student.email),
    'dni': new FormControl(this.student.dni),
    'address': new FormControl(this.student.address)
  });
}
```

```html
<!-- student-add.component.html -->
<div *ngIf="firstName.invalid && (firstName.dirty || firstName.touched)"
    class="alert alert-danger">
    <div *ngIf="firstName.errors.required">
        First Name is required.
    </div>
    <div *ngIf="firstName.errors.minlength">
        First Name must be at least 20 characters long.
    </div>
    <div *ngIf="firstName.errors.forbiddenName">
        {{ firstName.errors.forbiddenName.value }} is a forbidden name
    </div>
</div>
```

- Podemos crear nuestros propios Validators para realizar validaciones especiales.
- La función **`forbiddenNames`** en este caso, es una *factory* que toma una *regular expression* para detectar un nombre prohibido que es especificado y retorna una validator function.
- En este ejemplo, el forbidden name es **“Juan”**, por lo tanto el **`Validator`** rechaza cualquier **`firstName`** que contenga **“Juan”**.
- La función **`forbiddenNames`** retorna el **`validator function`** configurado. Esta función toma un **`Control Object`** de **`Angular`** y retorna **`null`** si el control es válido o un **`Validation Error Object`** en caso contrario.
- El **`Validation Error Object`** generalmente tiene una property cuyo name es la **`Validation Key (‘forbiddenName’)`** y cuyo valor es un **`dictionary (Key/Value)`** que puede usarse dentro de un error message en el template.
- Los **`Custom Async Validators`**, son similares a los **`Custom Validators`**, pero deben devolver una **`Promise`** u **`Observable`** que luego emitirán **`null`** o un **`Validation Error Object`**.

Observables - Introducción

```
// Create an Observable that will start listening to geolocation updates
// when a consumer subscribes.
const locations = new Observable((observer) => {
  // Get the next and error callbacks. These will be passed in when
  // the consumer subscribes.
  const {next, error} = observer;
  let watchId;

  // Simple geolocation API check provides values to publish
  if ('geolocation' in navigator) {
    watchId = navigator.geolocation.watchPosition(next, error);
  } else {
    error('Geolocation not available');
  }

  // When the consumer unsubscribes, clean up data ready for next subscription.
  return {unsubscribe() { navigator.geolocation.clearWatch(watchId); }};
});

// Call subscribe() to start listening for updates.
const locationsSubscription = locations.subscribe({
  next(position) { console.log('Current Position: ', position); },
  error(msg) { console.log('Error Getting Location: ', msg); }
});

// Stop listening for location after 10 seconds
setTimeout(() => { locationsSubscription.unsubscribe(); }, 10000);
```

- Los **`Observables`** proveen soporte para pasar mensajes entre **`publishers`** y **`subscribers`** dentro de una aplicación.
- Los **`Observables`** ofrecen beneficios significativos sobre otras técnicas de manejo de eventos, asincronismo y manejo de múltiples valores.
- Los **`Observables`** son declarativos, es decir, se define una función que publicará valores pero no se ejecuta hasta que un consumidor se suscriba a ella. Luego el suscriptor recibirá notificaciones hasta que la función complete o hasta que cancele la suscripción.
- La forma de recibir valores es la misma ya sean sincrónicos o asincrónicos por lo tanto la aplicación solo debe preocuparse por suscribirse para consumir valores.
- Como **`Publisher`** se crea un **`Observable`** que define una **`subscriber function`**. Esta función es ejecutada cuando un consumidor llama al método **`subscribe()`**.
- Para ejecutar el **`observable`** y recibir notificaciones, se llama el método **`subscribe()`** pasando un **`observer`** como parámetro. De esta manera definimos los **`handlers`** para las notificaciones recibidas. La llamada a **`subscribe()`** retorna un objeto que posee un **`unsubscribe()`** method para detener la recepción de notificaciones.

Observers y Suscripción

```
// Create simple observable that emits three values
/*of(...items)–Returns an Observable instance that
synchronously delivers the values provided as arguments.
from(iterable)–Converts its argument to an Observable instance.
This method is commonly used to convert an array to an observable.*/
const myObservable = of(1, 2, 3);

// Create observer object
const myObserver = {
  next: x => console.log('Observer got a next value: ' + x),
  error: err => console.error('Observer got an error: ' + err),
  complete: () => console.log('Observer got a complete notification'),
};

// Execute with the observer object
myObservable.subscribe(myObserver);
// Logs:
// Observer got a next value: 1
// Observer got a next value: 2
// Observer got a next value: 3
// Observer got a complete notification

//or instead of passing an observer object, just passing callback functions
myObservable.subscribe(
  x => console.log('Observer got a next value: ' + x),
  err => console.error('Observer got an error: ' + err),
  () => console.log('Observer got a complete notification')
);
```

- **Definiendo `Observers`:** Un **`handler`** para recibir notificaciones de un **`Observable`** implementa la interfaz **`Observer`**. Es un objeto que define **`callback methods`** para manejar los tres tipos de notificaciones que un observable puede enviar:
  - **`next`**: Requerido. Un **`handler`** para cada valor entregado. Llamado cero o mas veces luego de que comienza la ejecución.
  - **`error`**: Opcional. Un **`handler`** para una notificación de error. Un error detiene la ejecución del observable.
  - **`complete`**: Opcional. Un **`handler`** para la notificación de ejecución completa. Valores con delay puede continuar siendo entregados al “**`next handler`**” luego de que la ejecución se completó.

- **Suscripción:** Un **`Observable`** comienza publicando valores solo cuando un **`Observer`** se suscribe a este. La suscripción se realiza llamando al método **`subscribe()`** en el observable y pasando el **`Observer`** como parámetro.

- Alternativamente el método **`subscribe()`** puede recibir **`callback functions`** en lugar de un observer.

- La función **`next()`** puede recibir por ejemplo, strings con mensajes, objects, valores numéricos, etc. dependiendo el contexto. Nos referimos a los datos publicados por un observable como un **`stream`**.

```
// This function runs when subscribe() is called
function sequenceSubscriber(observer) {
  // synchronously deliver 1, 2, and 3, then complete
  observer.next(1);
  observer.next(2);
  observer.next(3);
  observer.complete();

  // unsubscribe function doesn't need to do anything in this
  // because values are delivered synchronously
  return {unsubscribe() {}};
}

// Create a new Observable that will deliver the above sequence
const sequence = new Observable(sequenceSubscriber);

// execute the Observable and print the result of each notification
sequence.subscribe({
  next(num) { console.log(num); },
  complete() { console.log('Finished sequence'); }
});

// Logs:
// 1
// 2
// 3
// Finished sequence
```

Observables

- Se utiliza el constructor **`Observable`** para crear un **`stream`** de cualquier tipo. Toma como parámetro la **`subscriber function`** a correr cuando el método **`subscribe()`** del **`observable`** se ejecuta. La **`subscriber function`** recibe un **`Observer`** como parámetro y puede publicar valores al método **`next()`** del **`observer`**.
- En este ejemplo para crear un **`observable`** equivalente al **`of(1, 2, 3)`** anterior, podemos definirlo como un **`new Observable()`** que recibe una **`subscriber function`**.

```
// Create an Observable out of a promise
const data = from(fetch('/api/endpoint'));
// Subscribe to begin listening for async result
data.subscribe({
  next(response) { console.log(response); },
  error(err) { console.error('Error: ' + err); },
  complete() { console.log('Completed'); }
});

// Create an Observable that will publish a value on an interval
const secondsCounter = interval(1000);
// Subscribe to begin publishing values
secondsCounter.subscribe(n =>
console.log(`It's been ${n} seconds since subscribing!`));


// Create an Observable that will publish mouse movements
const el = document.getElementById('my-element');
const mouseMoves = fromEvent(el, 'mousemove');


// Subscribe to start listening for mouse-move events
const subscription = mouseMoves.subscribe((evt: MouseEvent) => {
  // Log coords of mouse movements
  console.log(`Coords: ${evt.clientX} X ${evt.clientY}`);

  // When the mouse is over the upper-left of the screen,
  // unsubscribe to stop listening for mouse movements
  if (evt.clientX < 40 && evt.clientY < 40) {
    subscription.unsubscribe();
  }
});
```

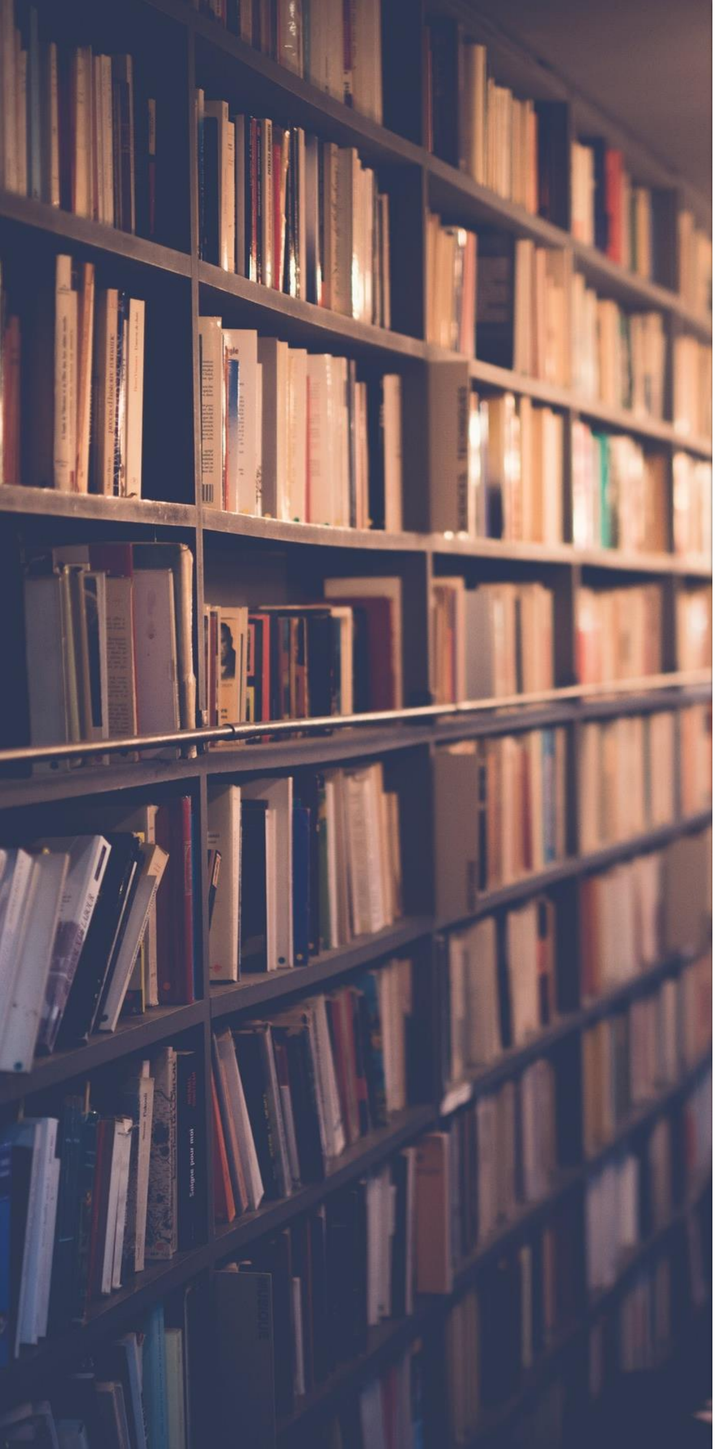
RxJS – Reactive Extensions for JavaScript

- **Reactive programming** es un paradigma de programación asincrónica enfocado en **data streams** y **propagación de cambios** (Wikipedia).
- **RxJS** es una librería para **reactive programming** que utiliza **observables** para componer llamadas asincrónicas de manera mas sencilla (RxJS Docs.).
- **RxJS** provee funciones de utilidad que nos permiten realizar las siguientes tareas con observables entre otras:
  - Convertir operaciones asincrónicas en observables.
  - Iterar a través de valores en un stream.
  - Mapear valores a diferentes tipos.
  - Filtrar streams.
  - Crear múltiples streams.
- **RxJS** ofrece funciones que pueden usarse para crear nuevos observables simplificando esta tarea. Como por ejemplo crear observables a partir de una **Promise**, de un **Counter**, de un **Event**, etc.

```
//Map Operator
const nums = of(1, 2, 3);

const squareValues = map((val: number) => val * val);
const squaredNums = squareValues(nums);

squaredNums.subscribe(x => console.log(x));

// Logs 1 , 4, 9


//Pipe function
// Create a function that accepts an Observable.
const nums = of(1, 2, 3, 4, 5);

const squareOddVals = pipe(
  filter((n: number) => n % 2 !== 0),
  map(n => n * n)
);

// Create an Observable that will run the filter and map functions
const squareOdd = squareOddVals(nums);

// Subscribe to run the combined functions
squareOdd.subscribe(x => console.log(x));


//Pipe function
const squareOdd = of(1, 2, 3, 4, 5)
  .pipe(
    filter(n => n % 2 !== 0),
    map(n => n * n)
  );

// Subscribe to get values
squareOdd.subscribe(x => console.log(x));
```

RxJS – Operators

- Los **`Operators`** son funciones que trabajan con **`observables`** para permitir una manipulación avanzada de colecciones. Por ejemplo, **`RxJS`** define operadores como **`map()`**, **`filter()`**, **`concat()`**, etc.
- Los operadores toman opciones de configuración y retornan una función que toma un **`observable`** como **`source`**. Cuando se ejecuta esta función de retorno, el **`operator`** observa los valores emitidos por el **`source`**, los transforma y retorna un nuevo observable.
- Se puede utilizar **`pipes`** para linkear operadores. **`Pipes`** permite combinar múltiples funciones en una sola función. La función **`pipe()`** toma como parámetros las funciones que se desean combinar y retorna una nueva función que al ser ejecutada, corre las funciones compuestas en secuencia.



Bibliografía y recursos

HTML Basics, JavaScript, CSS : https://www.w3schools.com/

Campus: http://campus.mdp.utn.edu.ar/course/edit.php?id=169

GitHub Repository: https://github.com/JuanAzar


¿Preguntas?


END
GRACIAS POR ACOMPAÑARNOS
EN ESTE CAMINO