UNIDAD 1: HTML, CSS Y JAVASCRIPT

HTML (HYPERTEXT MARKUP LANGUAGE)

¿Qué es HTML?

HTML es el lenguaje de marcado estándar para la creación de paginas web que indica a los navegadores web como estructurar y presentar el contenido a los usuarios.

- HyperText: Es la capacidad de enlazar un documento con otros permitiendo la navegación entre páginas.
- Markup Language: Significa que se utilizan etiquetas (tags) para definir la estructura y formato del contenido.

Estructura Basica de un documento HTML

Todo documento HTML moderno debe comenzar con la declaración del tipo de documento **(<!DOCTUPE html>)**, que indica que usaremos **HTML5**.

Principales etiquetas HTML

HTML dispone de múltiples etiquetas, las más básicas y fundamentales se pueden agrupar en las siguientes categorías:

1) **Estructurales**
   - **<html>**: Contenedor raíz del documento.
   - **<head>**: Incluye metadatos, enlaces a hojas de estilo, scripts y configuración inicial del documento.
   - **<body>**: Contiene el contenido visible que se muestra en pantalla.
2) **Texto**
   - **<h1> a <h6>**: Encabezados de diferentes niveles o tamaños en orden de jerarquía, <h1> mas importante y <h6> menos importante.
   - **<p>**: Parrafos de texto para oraciones, descripciones y cualquier texto largo.
   - **
**: Inserta un salto de línea.
   - **<hr>**: Dibuja una línea horizontal para separa secciones.
3) **Enlaces e Imágenes**
   - **<a>**: Crea un hipervínculo a otra pagina o recurso. El atributo mas importante es el **href** donde se coloca la URL.
   - **<img>**: inserta una imagen en la pagina, requiere como mínimo el atributo **src** para indicar la ruta de la imagen.

Etiquetas Semánticas

Las etiquetas semánticas en HTML se diseñan para describir de manera clara y concisa el propósito y la estructura del contenido. Sirven para:

- **Mejorar la accesibilidad:** Herramientas como lectores de pantalla comprenden mejor la organización del contenido y ofrecen una experiencia de navegación más precisa.
- **Optimizar SEO:** Los motores de búsqueda pueden analizar con mayor precisión la importancia y relevancia de cada sección, lo cual favorece la indexación y la posición en los resultados.
- **Mantener el código limpio y comprensible:** Otros desarrolladores (e incluso tú mismo en el futuro) pueden entender rápidamente la estructura de la página.

Principales Etiquetas Semánticas

- **<header>**: Representa la cabecera de la página o de una sección concreta.
- **<nav>**: Indica una sección destinada a la navegación (menú).
- **<section>**: Agrupa contenido temáticamente relacionado y se usa frecuentemente para dividir la página en bloques lógicos.
- **<article>**: Representa contenido independiente y autosuficiente.
- **<aside>**: Se utiliza para contenido secundario o complementario que está relacionado con el contenido principal.
- **<footer>**: Define la sección que normalmente aparece al final de un documento o sección.
- **<main>**: Elemento único para cada página que encapsula el contenido principal, aquello que es esencial y único de la misma, sirve para ubicar rápidamente el núcleo de la información.

CSS (CASCADING STYLE SHEETS)

¿Qué es CSS?

CSS es el lenguaje utilizado para definir la presentación de documentos HTML. Aporta estilos, diseños y efectos visuales, separando de manera clara el contenido (HTML) de la presentación (CSS).

Ventajas

- **Separación de contenido y presentación:** Se gestionan estilos de forma independiente del contenido.
- **Reutilización de estilos:** Permite definir estilos una sola vez y aplicarlos en múltiples lugares.
- **Rendimiento:** Al mantener estilos en un único archivo CSS, se reduce la redundancia y se optimiza la carga de la página.
- **Consistencia:** Garantiza una apariencia uniforme en todas las páginas de un sitio web.

Selectores de CSS

Los selectores indican a que elementos HTML se les aplicaran determinados estilos. Los más comunes son:

- **Selector de tipo (elemento):** Aplica estilos a todos los elementos de un tipo en particular.
- **Selector de clase (.clase):** Comienza con un punto (.) y se aplica a todos los elementos que tengan esa clase.
- **Selector de ID (#id):** Comienza con el símbolo # y se aplica a un único elemento que tenga ese ID.
- **Selector de atributo:** Selecciona elementos basados en sus atributos

Propiedades Comunes

- **Color (color):** Cambia el color de texto.
- **Fondo (background-color):** Cambia el color de fondo.
- **Ancho (width) y Alto (height):** Define el tamaño de los elementos.
- **Margen (margin) y relleno (padding):** Controlan el espacio alrededor y dentro de los elementos.
- **Borde (border):** Controla el borde.

Modelo de Caja (Box Model)

En CSS, todos los elementos se representan como cajas rectangulares. La caja consta de cuatro partes principales:

- **Content (Contenido):** El área donde reside el texto, imágenes o elementos hijos.
- **Padding (Relleno):** Espacio entre el contenido y el borde de la caja.
- **Border (Borde):** Línea que rodea la caja, afectando su tamaño total.
- **Margin (Margen):** Espacio externo entre el borde de la caja y otros elementos.

Diseño Responsivo

El diseño responsivo (responsive design) permite que un sitio web se adapte de manera automática a distintos tamaños de pantalla, desde móviles hasta monitores de escritorio. Algunas técnicas clave incluyen:

- **Media Queries:** Aplican estilos específicos cuando se cumplen ciertas condiciones (por ejemplo, un ancho máximo de pantalla).
- **Unidades relativas**
  - **% (porcentaje):** relativo al contenedor padre.
  - **vw (viewport width) y vh (viewport height):** relativos al ancho/alto de la ventana del navegador.
  - **em y rem:** relativos al tamaño de fuente establecido (o al elemento raíz en el caso de rem).

TAILWIND CSS

¿Qué es Tailwind CSS?

Es un framework de utilidades para CSS que te permite diseñar interfaces de usuario con gran flexibilidad. Proporciona una extensa colección de clases de utilidad que brindan un control total sobre el estilo de cada elemento.

Beneficios clave

- **Personalización absoluta:** Permite crear diseños únicos, partiendo de utilidades en lugar de "recetas fijas".
- **Desarrollo rápido y ágil:** Mediante el uso de clases de utilidad, es muy sencillo pasar de prototipos a diseños finales.
- **Diseño responsivo simple:** Tailwind incluye prefijos de responsive (como **sm:, md:, lg:, xl:**) que facilitan la adaptación de tu diseño a distintas pantallas.
- **Mantenimiento sencillo:** Al basarse en un enfoque puramente utilitario, los estilos se definen directamente en las clases.

JAVASCRIPT

¿Qué es JavaScript?

Es un leguaje de programación de alto nivel, dinamico y no tipado, que se utiliza principalmente para añadir interactividad y dinamismo a las paginas web.

Se ejecuta en el navegador del cliente permitiendo:

- **Manipular el DOM (Document Object Model):** Cambiar o actualizar elementos en tiempo real.
- **Responder a eventos de usuario:** Manejar acciones como clics de botón, movimientos de ratón o envíos de formularios.
- **Realizar peticiones asíncronas:** Interactuar con APIs, Servidores, BD y otros servicios sin recargar la pagina.

Modelo de programación

JavaScript adopta un enfoque orientado a objetos basado en prototipos en lugar del tradicional modelo de clases puro. Esto significa que los objetos pueden heredar directamente de otros objetos.

Sintaxis – Declaración de variables

Existen varias maneras de declarar variables:

- **Let:** Variables de ámbito de bloque cuyo valor es reasignable o cambia.
- **Const:** Variables constantes de ámbito de bloque cuyo valor no puede ser reasignado o no cambia.
- **Var:** Variable de alcance de función (no de bloque).

UNIDAD 2: DOM, FETCH y API

Modelo Cliente-Servidor

- Es un **modelo de red** donde el **cliente** (navegador, app móvil) hace solicitudes y el **servidor** (Apache, AWS) responde con recursos o servicios.
- **Proceso**:
    1. Cliente envía una solicitud HTTP.
    2. Servidor procesa la solicitud.
    3. Servidor devuelve una respuesta HTTP.
- **Roles**:
    - **Cliente**: Envía solicitudes y renderiza páginas (HTML, CSS, JS).
    - **Servidor**: Almacena archivos, bases de datos y procesa peticiones.

Servidores Web y Protocolo HTTP

- **Servidor Web**: Software que maneja solicitudes HTTP y entrega archivos web (HTML, CSS, imágenes).
- Ejemplos: **Apache**, **Nginx**, **IIS**.
- **HTTP** (HyperText Transfer Protocol):
    - Comunicación basada en **solicitud/respuesta**.
    - **Métodos comunes**:
        - **GET** (obtener datos)
        - **POST** (enviar datos)
        - **PUT** (actualizar)
        - **DELETE** (eliminar).
    - **Códigos de estado**:
        - 200 OK (éxito)
        - 404 Not Found (no encontrado)
        - 500 Internal Server Error (error servidor).

Sitios Estáticos y Dinámicos

- **Sitio Estático**:
  - Contenido fijo.
  - Servidor envía archivos tal cual (HTML, CSS, imágenes).
  - Rápidos, seguros y fáciles de alojar.
- **Sitio Dinámico**:
  - Contenido generado en tiempo real (ej: redes sociales).
  - Usa servidores que procesan datos (PHP, Node.js, Python).
  - Permite interactividad, login, formularios, bases de datos.
- **Comparativa**:

Característica	**Estático**	**Dinámico**
Contenido	Fijo	Generado en tiempo real
Interactividad	Limitada	Alta
Hosting	Simple	Requiere backend

DOM (Document Object Model)

- El **DOM** representa una página HTML como un **árbol de nodos**.
- Cada etiqueta HTML es un **nodo** en el árbol (ej: , <p>, <img>).
- **Manipulación del DOM**:
  - **Acceder elementos**:
    - getElementById
    - querySelector
  - **Crear elementos**: createElement, appendChild.
  - **Modificar atributos y propiedades**: cambiar src, href, textContent, etc.
  - **Recorrer el DOM**: moverse entre padres, hijos, hermanos.

- **Errores comunes**:
    - Modificar el DOM antes de que cargue (solución: usar DOMContentLoaded).
    - Acceso excesivo e ineficiente al DOM (causa bajo rendimiento).

Manejo de Eventos en la Web

- **Eventos**: Acciones que ocurren en la web (clics, movimientos, teclado, envíos de formularios).
- **Modelo de Eventos**:
    - **Captura**: El evento baja desde window hasta el objetivo.
    - **Objetivo**: El evento ocurre en el elemento objetivo.
    - **Burbuja**: El evento sube desde el objetivo hacia window.
- **Eventos Comunes**:
    - **Ratón:** click, dblclick, mouseover.
    - **Teclado:** keydown, keyup.
    - **Formularios:** submit, change, input.
    - **Página:** load, DOMContentLoaded, scroll.
- **Cómo manejar eventos**:
    - **Usar** addEventListener.
    - **Delegación de eventos:** escuchar eventos en un contenedor padre.
- **Objeto event**:
    - **Propiedades importantes:** event.target, event.type.
    - **Métodos útiles:** preventDefault() (anular comportamiento), stopPropagation() (detener propagación).

Fetch API (Peticiones HTTP)

- **Fetch API** permite hacer solicitudes HTTP desde JavaScript **de manera moderna** (sustituye XMLHttpRequest).
- **Características**:
  - Basado en **Promesas** → más limpio y legible.
  - Compatible con async/await.
  - Útil para consumir **APIs REST**.
- **Errores comunes**:
  - No convertir respuesta a .json().
  - No manejar errores de red o de respuesta.
  - Problemas de **CORS** al pedir datos de otro dominio.
- **Buenas prácticas**:
  - Comprobar response.ok.
  - Manejar try/catch si usas async/await.

Promesas en JavaScript

- **Promesas**: objeto que representa un valor futuro de una operación asíncrona.
- **Estados de una Promesa**:
  1. **Pending** (pendiente)
  2. **Fulfilled** (cumplida)
  3. **Rejected** (rechazada)
- **Flujo de trabajo**:
  - then() → ejecutar si éxito.
  - catch() → ejecutar si error.
  - finally() → ejecutar siempre.

- **Importante**:
    - Las Promesas **se pueden encadenar**.
    - **async/await** simplifica su uso (await "espera" que la promesa se resuelva).
- **Errores comunes**:
    - No encadenar correctamente promesas.
    - No capturar errores (importante usar .catch() o try/catch).

Body y Headers en HTTP

- **Headers**: metadatos en las solicitudes/respuestas HTTP (tipo de contenido, autenticación, control de caché).
    - **Ejemplos:** Content-Type, Authorization, Accept.
- **Body**: contenido enviado en solicitudes como POST, PUT.
    - Puede ser en formato JSON, FormData o XML.
- **Errores comunes**:
    - No especificar Content-Type correcto.
    - No validar datos del body en el backend.

Obtener Parámetros desde la URL

- **Query String**: parte de la URL que comienza con ? y tiene parámetros separados por &.
    - **Ejemplo:** ?nombre=Juan&edad=30.
- **Uso de URLSearchParams**:
    - **get():** obtener el valor de un parámetro.
    - **has():** verificar existencia de un parámetro.
    - **append(), set(), delete():** manipular parámetros.

- **Buenas prácticas**:
    - Validar los parámetros obtenidos.
    - Codificar/decodificar valores especiales.
    - Tener cuidado con navegadores antiguos (IE11 no soporta URLSearchParams).

Web Storage API

- Permite **guardar datos en el navegador** en clave-valor.
- **Tipos**:
    - **localStorage:** datos persistentes que sobreviven cierres de navegador.
    - **sessionStorage:** datos temporales que se borran al cerrar la pestaña.
- **Métodos comunes**:
    - **setItem**(key, value), **getItem**(key), **removeItem**(key), **clear**().
- **Diferencias**:

Aspecto	**sessionStorage**	**localStorage**
Duración	Solo durante la pestaña abierta	Persistente
Alcance	Por pestaña	Por dominio
Uso típico	Datos temporales de la sesión	Preferencias, datos persistentes

- **Errores comunes**:
    - Olvidar serializar objetos con JSON.stringify.
    - Pensar que sessionStorage comparte datos entre pestañas (no lo hace).
    - Guardar datos sensibles sin protección.

Introducción a TypeScript

Definición y Relación con JavaScript (superset): TypeScript es un superconjunto (superset) de JavaScript desarrollado por Microsoft. Esto implica que todo código JavaScript válido es también TypeScript, pero TypeScript añade características adicionales, principalmente el tipado estático.

Propósito Principal: Su objetivo es añadir tipado estático a JavaScript, permitiendo detectar errores de tipo durante la fase de compilación, antes de que el código se ejecute. El código TypeScript no se ejecuta directamente; se transpila (compila) a JavaScript puro (.js) usando el compilador tsc.

Beneficios Clave:

- **Detección de Errores Temprana:** Evita errores comunes en tiempo de ejecución al verificar los tipos durante el desarrollo.
- **Mejor Autocompletado (IntelliSense):** Los editores de código aprovechan la información de tipos para ofrecer sugerencias más precisas, acelerando el desarrollo.
- **Código más Mantenible y Autodocumentado:** Al definir explícitamente las estructuras de datos, el código se vuelve más legible y fácil de mantener, especialmente en proyectos grandes.

Tipos Básicos

Tipos Primitivos: TypeScript utiliza los tipos primitivos de JavaScript, permitiendo su anotación explícita:

- **string:** para cadenas de texto.
- **number:** para valores numéricos (enteros y flotantes).
- **boolean:** para true o false.

Arreglos (Arrays): Se puede especificar el tipo de los elementos de un arreglo de dos maneras, que son intercambiables:

- string[]: let numeros: number[] = [1, 2, 3];
- Array<string>: let palabras: Array<string> = ["uno", "dos"];

El tipo any: Es un tipo especial que desactiva la verificación de tipos de TypeScript. Una variable any puede recibir cualquier valor y ser usada de cualquier forma. Aunque es útil para migrar código JavaScript, se recomienda evitar su uso ya que se pierden las ventajas de la detección de errores.

Anotaciones de Tipo vs. Inferencia:

Anotación: Declarar explícitamente el tipo de una variable usando la sintaxis : Tipo.

Inferencia: Cuando se inicializa una variable, TypeScript deduce su tipo automáticamente. Es la práctica recomendada para mantener el código conciso cuando el tipo es obvio.

Funciones

Parámetros: Se debe especificar el tipo de cada parámetro que recibe una función, asegurando que solo se pasen los datos correctos.

Tipo de Retorno: Se puede declarar el tipo de valor que una función retorna usando la sintaxis : Tipo después de los parámetros. Si no retorna nada, se usa void. TypeScript también puede inferir el tipo de retorno.

Tipos Avanzados

Tipos de Objetos: Se puede definir la "forma" de un objeto especificando sus propiedades y tipos de manera literal.

Propiedades Opcionales: Se usa el símbolo ? para indicar que una propiedad puede no estar presente en un objeto.

Tipos de Unión (Union Types): El operador | permite que una variable pueda ser de uno de varios tipos. Para usar métodos específicos de un tipo, se debe hacer "narrowing" (acotación) usando una comprobación como typeof.

Alias de Tipo (Type Aliases): La palabra clave type permite crear un nombre personalizado para cualquier tipo (primitivo, unión, objeto, etc.), mejorando la legibilidad y reutilización.

Interfaces: Una interface es otra forma de nombrar y definir un contrato para la estructura de un objeto. Su propósito principal es describir la forma que deben tener los objetos o las clases.

Diferencias entre Alias de Tipo e Interfaces:

- **Extensibilidad:** Las interfaces pueden ser extendidas por otras (extends) y fusionadas (declaration merging). Los alias de tipo no pueden ser modificados una vez creados.
- **Flexibilidad:** Los alias de tipo son más generales y pueden representar uniones (string | number), tuplas y otros tipos que las interfaces no pueden.
- **Implementación en Clases:** Las clases usan implements con interfaces para asegurar que cumplen un contrato, lo cual no es posible directamente con un alias de tipo de objeto.
- **null y undefined:** Con la opción strictNullChecks activada (recomendado), null y undefined no son asignables a otros tipos a menos que se declaren explícitamente en una unión (ej. string | null). Esto previene errores de acceso a valores nulos.
- **Enumeraciones (enum):** Permiten crear un conjunto de constantes con nombre, reemplazando "números mágicos" o cadenas por nombres descriptivos. Pueden ser numéricas (con valores auto-incrementales por defecto) o de cadena.

Clases y Programación Orientada a Objetos (POO)

Definición de Clases y Propiedades: Se usa la palabra clave class. Las propiedades deben ser declaradas con su tipo.

Modificadores de Acceso:

- **public:** (Por defecto) Accesible desde cualquier lugar.
- **private:** Accesible solo desde dentro de la misma clase.
- **protected:** Accesible desde la clase y sus subclases.

Constructores, Métodos, Getters y Setters:

- **Constructor:** Método especial para inicializar instancias.
- **Métodos:** Funciones dentro de una clase que operan sobre la instancia (this).
- **Getters y Setters:** Métodos especiales para controlar el acceso a las propiedades, permitiendo lógica de validación o cálculo.

Herencia (extends): Una clase (subclase) puede heredar propiedades y métodos de otra (superclase). Se debe usar super() en el constructor de la subclase para llamar al constructor de la superclase.

Implementación de Interfaces (implements): Una clase puede adherirse a un contrato definido por una interfaz, lo que obliga a la clase a proveer una implementación para todas las propiedades y métodos de la interfaz.

Módulos

Exportando e Importando: TypeScript usa la sintaxis de módulos de ES6. Un archivo se convierte en un módulo si contiene export o import.

- **Exportación nombrada:** export const PI = 3.14;
- **Importación nombrada:** import { PI } from './matematicas';
- **Exportación por defecto:** export default function saludar() { ... }
- **Importación por defecto:** import miFuncion from './saludo';

Exportar/Importar Tipos: Las interface y type se exportan e importan con la misma sintaxis, permitiendo compartir definiciones de tipos en todo el proyecto. Estas importaciones se eliminan en el JavaScript compilado.

Decoradores

Un decorador es una función especial que se aplica a clases, métodos, propiedades o parámetros usando la sintaxis @nombreDecorador. Su propósito es la metaprogramación: alterar o extender el comportamiento de forma declarativa. Son una característica experimental que debe ser habilitada en tsconfig.json.

tsconfig.json y Configuración del Proyecto

Definición y Propósito: Es el archivo de configuración fundamental de un proyecto TypeScript. Indica al compilador qué archivos incluir, qué opciones de compilación usar y dónde generar el resultado. Se crea con tsc --init.

Opciones Importantes:

- **target:** La versión de JavaScript a la que se compilará (ej. "ES2015").
- **module:** El sistema de módulos del código de salida (ej. "commonjs" para Node.js).
- **outDir:** El directorio donde se guardarán los archivos .js compilados (ej. "./dist").

- **strict:** Habilita un conjunto de reglas estrictas de verificación de tipos (muy recomendado).
- **noEmitOnError:** Evita que se generen archivos .js si hay errores de compilación.

Comparación entre JavaScript y TypeScript

Característica	JavaScript (JS)	TypeScript (TS)
Tipado	Dinámico (los tipos se verifican en tiempo de ejecución).	Estático (los tipos se verifican en tiempo de compilación).
Detección de Errores	En tiempo de ejecución, a menudo en producción.	En tiempo de compilación, antes de ejecutar el código.
Sintaxis Extra	No tiene sintaxis nativa para interfaces, enums, etc.	Añade interface, type, enum, modificadores de acceso, etc.
Ejecución	Se ejecuta directamente en el navegador o Node.js.	Requiere un paso de compilación para generar JavaScript.
Flujo de Trabajo	Más simple para scripts pequeños.	Añade un paso de compilación, pero mejora la robustez en proyectos grandes.

Administración del Proyecto con npm

Importancia del package.json: Es el archivo de configuración principal de un proyecto Node.js/frontend. Define metadatos, dependencias (dependencies y devDependencies como typescript) y scripts para automatizar tareas.

Comandos de npm Comunes:

- **npm install typescript --save-dev:** Instala TypeScript como dependencia de desarrollo.
- **npm run build:** Un script personalizado, usualmente definido como "build": "tsc", para compilar el proyecto.
- **npm run start:** Un script para ejecutar la aplicación (ej. node dist/index.js).

Introducción a las SPAs y el Contexto de Angular

¿Qué es una Single Page Application (SPA)?
Una Single Page Application (SPA) o "Aplicación de una Sola Página" es una aplicación web que funciona cargando una única página HTML. En lugar de recargar la página entera cada vez que el usuario interactúa (por ejemplo, al hacer clic en un enlace), la SPA actualiza dinámicamente solo las partes del contenido que necesitan cambiar.

Son relevantes en el desarrollo moderno porque ofrecen:

- **Mayor Rendimiento y Velocidad:** Al no tener que recargar toda la página, la experiencia es mucho más rápida y fluida, parecida a la de una aplicación de escritorio.
- **Mejor Experiencia de Usuario:** La navegación es continua y sin interrupciones.
- **Arquitectura Modular:** Facilitan la organización del código, haciendo que las aplicaciones sean más fáciles de desarrollar, mantener y escalar

Presentación de Angular

Angular es un framework de código abierto, mantenido por Google, diseñado específicamente para construir SPAs complejas y de alto rendimiento.

Sus características clave son:

- **Basado en Componentes:** La interfaz de usuario se construye a partir de piezas reutilizables y autocontenidas.
- **Uso de TypeScript:** Es un superconjunto de JavaScript que añade tipado estático, lo que ayuda a detectar errores de forma temprana y a escribir un código más robusto y mantenible.
- **Robusto y Completo:** Ofrece un conjunto integral de herramientas para el ruteo, manejo de formularios, comunicación con servidores y mucho más.

Conceptos Fundamentales de Angular

Componentes: Los Ladrillos de la Aplicación

Un Componente es la unidad fundamental de una aplicación Angular. Cada componente es una pieza de la interfaz de usuario que encapsula su propia lógica, vista y estilos.

Propósito: Dividir la interfaz en partes manejables, organizadas y reutilizables.

Estructura Típica:

- Un archivo TypeScript (.ts) que contiene la lógica y los datos.
- Un archivo HTML (.html) que define la estructura o plantilla (la vista).
- Un archivo CSS (.css) para los estilos específicos de ese componente.

Standalone Components (Componentes Autónomos): Desde Angular 17, esta es la forma moderna de crear componentes. La principal ventaja es que no necesitan ser declarados en módulos, lo que simplifica la arquitectura, reduce el código repetitivo y los hace más fáciles de reutilizar.

Data Binding: Sincronizando Datos y Vista
El Data Binding es el proceso que mantiene sincronizados los datos de la lógica del componente con lo que se muestra en la vista (HTML).

One-Way Binding (Enlace Unidireccional): Los datos fluyen en una sola dirección.

- **De la lógica a la vista:** {{ dato }} (interpolación) o [propiedad]="dato". Sirve para mostrar información.
- **De la vista a la lógica:** (evento)="funcion()". Sirve para reaccionar a acciones del usuario, como clics.

Two-Way Binding (Enlace Bidireccional): Los datos fluyen en ambas direcciones simultáneamente. Se utiliza la sintaxis [(ngModel)]. Si el usuario cambia un valor en un campo de texto, la propiedad en el componente se actualiza automáticamente, y viceversa. Es ideal para formularios.

Directivas: Manipulando el DOM
Una Directiva es una instrucción en la plantilla HTML que le dice a Angular cómo manipular el DOM (la estructura de la página).

Directivas Estructurales: Modifican la estructura del DOM agregando, eliminando o repitiendo elementos. Las nuevas directivas de control de flujo son la forma moderna y recomendada de hacerlo:

- @if: Muestra un bloque de HTML si una condición es verdadera.
- @for: Itera sobre una lista y repite un bloque de HTML para cada elemento.
- @switch: Muestra un bloque de HTML entre varias opciones, según el valor de una expresión.

Directivas de Atributo: Cambian la apariencia o el comportamiento de un elemento existente. Ejemplos comunes son ngClass (para añadir clases CSS dinámicamente) y ngStyle (para aplicar estilos CSS dinámicamente).

Servicios e Inyección de Dependencias

Un Servicio es una clase cuyo propósito principal es compartir lógica de negocio y datos entre diferentes componentes. Por ejemplo, un servicio puede encargarse de obtener datos de un servidor.

Para que un componente pueda usar un servicio, Angular utiliza un mecanismo llamado Inyección de Dependencias (DI). En lugar de que el componente cree una instancia del servicio por sí mismo, simplemente lo "pide" en su constructor, y Angular se encarga de "inyectarle" la instancia única del servicio. Esto promueve un código más limpio, desacoplado y fácil de probar.

Comunicación y Datos

Peticiones HTTP: Hablando con el Servidor
Para comunicarse con un servidor y obtener o enviar datos, Angular utiliza el servicio HttpClient. Este servicio permite realizar peticiones HTTP (como GET, POST, PUT, DELETE) a una API externa.

Asincronismo: Manejando Operaciones que Toman Tiempo
El asincronismo se refiere a tareas que no se completan de inmediato, como una petición a una API. Angular necesita una forma de manejar esto sin bloquear la aplicación.

- **Observables:** Son el mecanismo principal que Angular utiliza para manejar operaciones asíncronas. Cuando HttpClient realiza una petición, no devuelve los datos directamente, sino un Observable.
- **Suscripciones:** Un Observable es como un "flujo de datos" al que te puedes suscribir. Cuando los datos de la API finalmente llegan, el Observable los emite, y el código dentro de tu suscripción se ejecuta para procesarlos.

Interacción y UI

Comunicación entre Componentes

Para construir aplicaciones complejas, los componentes necesitan pasarse información entre sí.

- @Input(): Permite que un componente padre pase datos a un componente hijo. Se usa un decorador en una propiedad del hijo para marcarla como una "entrada".
- @Output(): Permite que un componente hijo emita un evento hacia su padre. El hijo usa un EventEmitter para enviar la señal, y el padre la escucha.

Pipes: Transformando Datos en la Vista

Un Pipe es una herramienta que permite transformar datos directamente en la plantilla HTML para cambiar su formato. Son muy útiles para mejorar la legibilidad sin añadir lógica al componente.

Ejemplos: Formatear una fecha (| date), un número como moneda (| currency), o convertir un texto a mayúsculas (| uppercase).

También es posible crear pipes personalizados para lógicas de transformación más específicas.

Formularios en Angular

Template-Driven Forms (Formularios Basados en Plantilla)

Es uno de los dos enfoques de Angular para manejar formularios. En este método, la mayor parte de la lógica y la estructura del formulario se definen directamente en la plantilla HTML usando directivas.

- ngForm: Se aplica a la etiqueta <form> para que Angular reconozca y gestione el formulario.
- ngModel: Se aplica a los campos de entrada (<input>) para vincularlos a propiedades del componente y para que Angular rastree su valor y estado.
- ngSubmit: Se usa en la etiqueta <form> para asociar el evento de envío del formulario a una función en el componente.

Validaciones: Angular aprovecha los atributos de validación de HTML5 (como required, minlength) y actualiza automáticamente el estado del formulario (por ejemplo, valid o invalid), lo que facilita mostrar mensajes de error o deshabilitar el botón de envío.

Guía de Estudio Rápida: RxJS en Angular
Introducción a RxJS en Angular

¿Qué es RxJS?

De forma sencilla, **RxJS** (Reactive Extensions for JavaScript) es una librería para trabajar con programación asíncrona mediante el uso de **Observables**. Piensa en un **Observable** como un "stream" o flujo de datos al que te puedes suscribir para recibir valores a medida que llegan en el tiempo (como eventos, respuestas HTTP, etc.). RxJS te da herramientas poderosas para crear, transformar, filtrar y combinar estos flujos.

¿Por qué es una herramienta fundamental en Angular?

Angular no solo recomienda RxJS, sino que lo integra en su núcleo. Es la forma estándar de manejar operaciones asíncronas en el framework. Lo encontrarás en todas partes:

- **Peticiones HTTP:** El HttpClient de Angular devuelve **Observables**, lo que nos permite manejar las respuestas de forma reactiva.
- **Formularios Reactivos y Eventos:** Permite escuchar cambios en los valores o estados de los formularios, así como eventos del DOM o WebSockets.
- **Enrutamiento:** Puedes suscribirte a cambios en los parámetros de una ruta.
- **Programación Reactiva:** Facilita la creación de interfaces de usuario (UI) que se actualizan automáticamente cuando los datos cambian, sin necesidad de sondeos manuales (*polling*).

Conceptos Fundamentales y Gestión de Suscripciones

Pilares de RxJS

El documento se centra en el pilar principal, pero el ecosistema se compone de tres:

- **Observable:** Es el productor del flujo de datos. Representa una colección de valores futuros. Es "perezoso" (*lazy*), lo que significa que no hace nada hasta que alguien se suscribe a él con el método .subscribe().
- **Observer:** Es el consumidor. Es un objeto con hasta tres *callbacks* (funciones) que le pasas al método subscribe:
  - next(valor): Se ejecuta por cada valor que emite el Observable.
  - error(err): Se ejecuta si ocurre un error, deteniendo el flujo.
  - complete(): Se ejecuta cuando el Observable termina de emitir valores.
- **Operator:** Son funciones que permiten manipular los datos que fluyen a través de un Observable.

¿Qué es una suscripción (Subscription)?

Una **Subscription** es el objeto que se obtiene al llamar a .subscribe(). Representa la ejecución activa del Observable y es el vínculo entre el productor (Observable) y el consumidor (Observer). Es crucial para poder "cancelar" la escucha de datos más tarde.

Punto Clave: Por qué es crucial desuscribirse

Es fundamental desuscribirse para **evitar fugas de memoria (memory leaks)**. Si un componente de Angular es destruido, pero una suscripción a un Observable de larga duración (como un interval o un Subject de un servicio) sigue activa, el componente no puede ser eliminado de la memoria por el recolector de basura. Esto provoca que:

- Se sigan ejecutando procesos innecesarios en segundo plano.
- Se consuman recursos de memoria que no se liberan.
- Se generen bugs difíciles de rastrear, como intentar actualizar un componente que ya no existe en la vista.

Métodos de Desuscripción

El documento destaca dos enfoques principales:

1. **Manual (.unsubscribe()):**
    - **Cómo funciona:** Guardas la **Subscription** en una variable del componente y llamas a su método .unsubscribe() dentro del hook ngOnDestroy(). Si tienes múltiples suscripciones, puedes agruparlas con el método .add().
    - **Desventajas:** Es un proceso manual, propenso a errores (es fácil olvidarse de una suscripción) y puede hacer que el código en ngOnDestroy() sea más complejo y difícil de mantener.
2. **Automática con takeUntil (Mejor Práctica):**
    - **Mecanismo:** Se crea un **Subject** (ej. destroy$) en el componente. Todas las suscripciones se encadenan con el operador **takeUntil(this.destroy$)** a través del método .pipe(). En ngOnDestroy(), simplemente se emite un valor desde destroy$ (con .next()) y se completa (.complete()).
    - **Ventajas:** Es una solución declarativa, limpia y escalable. Centraliza la lógica de desuscripción en un único punto, sin importar cuántas suscripciones tengas, previniendo eficazmente los *memory leaks*.

Observables que se completan solos

No siempre es necesario desuscribirse. Los Observables que emiten un número finito de valores y luego se completan automáticamente no generan fugas de memoria.

- **Se completan solos:** El **HttpClient** de Angular es el ejemplo perfecto. Emite un único valor (la respuesta HTTP) y se completa. También los creados con of o timer.
- **NO se completan solos:** Observables de eventos, **Subject**, **BehaviorSubject**, interval o WebSockets. Estos deben ser gestionados manualmente o con takeUntil.

El Poder de los Operadores y .pipe()

¿Qué son los Operadores de RxJS?

Los operadores son el corazón de RxJS. Son **funciones puras** que procesan los flujos de datos. Toman un Observable de entrada, realizan una operación (transformar, filtrar, combinar, etc.) y devuelven un nuevo Observable de salida, sin modificar el original.

La función del método .pipe()

El método **.pipe()** es el mecanismo que nos permite **encadenar operadores** de forma secuencial, legible y mantenible. Cada operador dentro del .pipe() procesa el resultado del anterior, creando una "tubería" de manipulación de datos.

Ejemplo de Operadores Principales (mencionados en el documento)

- **Operador de Creación (of):** Crea un Observable a partir de una lista de valores que le pasas como argumento. Emite cada valor en secuencia y luego se completa.
    - **Ejemplo:** of(1, 2, 3) emite 1, luego 2, luego 3 y finaliza.
- **Operador de Transformación (map):** Aplica una función a cada valor emitido por el Observable, devolviendo un nuevo valor transformado.
    - **Ejemplo:** map(user => user.name) transformaría un flujo de objetos user en un flujo con solo sus nombres.

- **Operador de Filtrado (filter):** Emite solo los valores del flujo original que cumplan con una condición booleana.
    - **Ejemplo:** filter(n => n > 10) solo dejaría pasar los números mayores a 10.

Subjects: Canales de Comunicación Reactiva

¿Qué es un Subject?

Un **Subject** es un tipo especial de Observable con una característica clave: es **multicasting**. Esto significa que múltiples suscriptores escuchan la misma y única ejecución del Subject. Actúa como un canal o bus de eventos: puedes emitirle valores desde cualquier parte usando su método .next(valor), y todos los componentes suscritos recibirán esa emisión.

¿Qué es un BehaviorSubject?

Un **BehaviorSubject** es un tipo de **Subject** que va un paso más allá: tiene "memoria". Su diferencia fundamental es que **almacena el último valor emitido** y lo entrega inmediatamente a cualquier nuevo suscriptor. Por esta razón, siempre debe ser inicializado con un valor.

Tabla Comparativa: Subject vs. BehaviorSubject

Característica	Subject	BehaviorSubject
Valor Inicial	No necesita un valor inicial.	**Requiere un valor inicial** obligatorio.
Comportamiento	Es "sin estado". Un nuevo suscriptor **no recibe** valores anteriores; solo los que se emitan *después* de su suscripción.	Es "con estado". Un nuevo suscriptor **recibe inmediatamente el último valor** emitido (o el valor inicial si no ha habido emisiones).
Acceso al Valor	No se puede obtener el valor actual de forma síncrona.	Se puede acceder al último valor de forma síncrona usando el método **.getValue()**.
Caso de Uso	Ideal para **eventos** y notificaciones que no necesitan estado, como "Guardado con éxito" o el destroy$ para desuscripciones.	Ideal para **gestionar un estado** que se comparte en la aplicación, como el usuario autenticado, el tema de la UI o el contenido de un carrito de compras.

Conclusión General

Como resume el documento, dominar estos conceptos es esencial para el desarrollo profesional con Angular. Entender cómo gestionar **suscripciones** para evitar *memory leaks*, utilizar el poder de los **operadores** con **.pipe()** para manipular datos de forma limpia, y saber elegir entre un **Subject** para eventos y un **BehaviorSubject** para estados, te permitirá construir aplicaciones Angular que no solo funcionan, sino que son robustas, eficientes, fáciles de mantener y verdaderamente reactivas.