

FrameWorks

GraphQL - Bootstrap – Laravel - Angular

I.E.S. Arcipreste de Hita

César San Juan Pastor

Dpto. de Informática

marzo 2025 – junio 2025

API DEVELOPMENT
WITH PHP USING GRAPHQL

Contenido

Capítulo I. Introducción	6
Historia de las Apis.....	.6
Problema que resuelve.....	7
Diseño Rest vs GraphQL.....	7
Por qué GraphQL	9
Preparando el entorno para PHP.....	10
Capítulo II. GraphQL: Introducción	14
Qué es GraphQL.....	14
Características.....	15
Arquitectura de GraphQL	16
Primer ejemplo	16
Introspección	18
Uso de la IA	19
Generación de un cliente html	22
Capítulo III. GraphQL: Esquemas y tipos	24
Esquema	24
Tipos.....	25
Capítulo IV. GraphQL: Consultas	39
Qué es el lenguaje de consultas	39
Estructura de una consulta desde el cliente.....	39
Componentes de una consulta GraphQL.....	40
Ejemplos de consultas GraphQL de introspección	45
Ejemplos de consultas GraphQL mutaciones	47
Ejemplos de consultas GraphQL suscripciones	48
Validaciones.....	49
Ejecución de las consultas	49
Capítulo V. Bootstrap	51
Configurar PhpStorm para Bootstrap	51
Configurar Bootstrap en mis páginas	51
Configuración inicial de una pagina.....	52
Disposición de la página	53
Formularios.....	59

Componentes Bootstrap.....	65
Clases de ayuda	66
Utilidades	67
Capítulo VI. Composer PHP	71
Uso de Composer.....	71
Capítulo VII. Laravel.....	76
Preparando un entorno de desarrollo.....	76
Inicializando Laravel.....	76
El patrón Modelo-Vista-Controlador (MVC).....	77
Tipos de programación	78
Estructura de directorios de la aplicación Laravel.....	79
Opciones de configuración	80
Rutas	82
Entendiendo la utilidad Artisan	89
Blade	90
Bases de datos	94
Controladores y Modelos	114
Autentificación bajo Laravel	117
GraphQL bajo Laravel	124
Desplegar a producción	127
Capítulo VIII. Proyecto Laravel – Bootstrap: Web de Reservas de Turismo Espacial: Estrella Viajera .	128
Requisitos Funcionales -	128
Requisitos No Funcionales (RNF)	128
Requisitos Software	129
Casos de Uso	130
Diagrama E-R simplificado	130
Modelo Relacional	130
Diagrama de clases	131
Desarrollo.....	132
Trabajo para el alumno	136
Capítulo IX. Angular.....	137
Introducción.....	137
Establecer el entorno de desarrollo	137

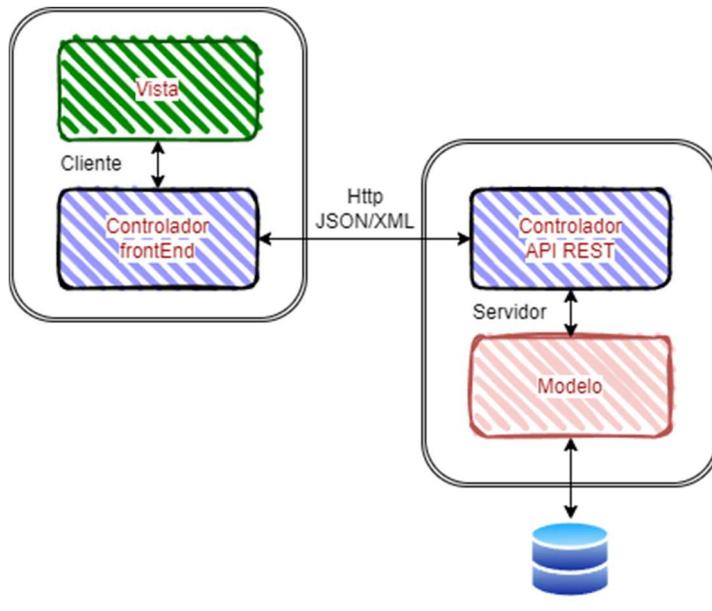
Estructura de un proyecto Angular	139
Introducción a TypeScript.....	140
Componentes.....	146
Tuberías y directivas	155
Servicios	158
Señales	163
Comunicación con servicios externos.....	164
Rutas	167
Formularios.....	176
Llevar a Producción.....	183
Capítulo X. Proyecto Laravel (GraphQL) - Angular.....	184
Servidor Laravel	184
Frontend Angular.....	186
Trabajo para el alumno.....	187
Anexo I. Recursos	189
Fuente de datos JSON.....	189
Anexo II. Índice completo	190

Ilustraciones

Ilustración 1- Paradigma MVC - API REST	6
Ilustración 2 - Historia del API Web	6
Ilustración 3 - Problemas del diseño en APIs.....	7
Ilustración 4 - REST vs GraphQL.....	8
Ilustración 5 - Salida de info.php con xdebug habilitado	12
Ilustración 6 - Primer ejemplo GraphQL.....	18
Ilustración 7 - Ejemplo Esquema	25
Ilustración 8 - Definición de tipo de objeto	26
Ilustración 9 - Ejemplo root operations.....	28
Ilustración 10 - Petición diseño con interfaces.....	30
Ilustración 11 - Diseño con uniones.....	31
Ilustración 12 - Transmisión Cliente Servidor.....	39
Ilustración 13 - Configuración de Bootstrap con CDN	51
Ilustración 14 – Patrón MVC.....	78
Ilustración 15 – Arquitectura Laravel MLP	78
Ilustración 16 - Herencia Blade.....	93
Ilustración 17 - La orden dd	117
Ilustración 18 - Herramienta Grapiql.....	125
Ilustración 19 - Comunicación entre componentes	163

Capítulo I. Introducción

Historia de las Apis



La programación web ha migrado desde aplicaciones estáticas a dinámicas a lo largo de décadas. Y estas últimas, de aplicaciones de múltiples páginas a las actuales compuestas por una única página. La necesidad de separar la información manejada por la aplicación del interfaz gráfico ha hecho que se desarrollen mecanismos en los que el servidor transportara de forma independiente los datos, de aquellos que “dibujan” el interfaz gráfico, lo que hoy se conoce comúnmente como el MVC (Modelo Vista Controlador), permitiendo independencia tanto en el desarrollo como en el transporte de la información.

Ilustración 1- Paradigma MVC - API REST

Podríamos definir API (del inglés: Application Programming Interface) dentro de una aplicación WEB como un interfaz de programación (o protocolo) para la interacción entre el cliente (frontend) y el servidor (backend).

A lo largo de los años se han presentado tres soluciones distintas para la implementación del paradigma MVC. En la imagen siguiente podemos apreciar la evolución de las mismas, junto a sus ventajas e inconvenientes.

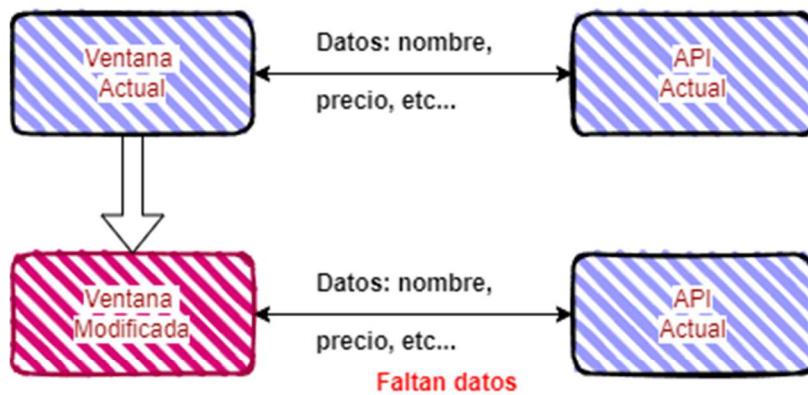


Ilustración 2 - Historia del API Web

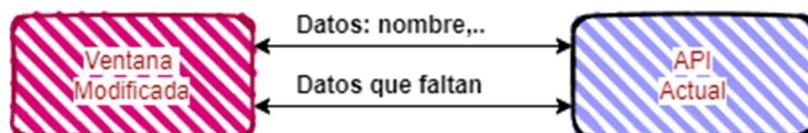
Los servicios web fueron los primeros en aparecer, pero su creación y mantenimiento es muy tediosa, sobre todo la obligación de la generación de los ficheros wsdl de descripción de los servicios de forma estática o dinámica. El segundo intento, el API REST, utiliza los comandos http GET/POST/PATCH/PUT/DELETE para dar una semántica a las transacciones que haga el cliente, pero el mayor inconveniente que presenta la necesidad de presentar un conjunto de puntos diferenciados de entrada al API para funcionalidades diferentes de la misma aplicación, es decir, tiene diferentes direcciones url y formas de las mismas además de los comandos http, impidiendo una gestión coherente

y siendo muy costoso el mantenimiento de diversas versiones del API así como su documentación. Por último, GraphQL aparece para intentar dar solución a ambos problemas, y añadir nuevas facilidades: presenta un único punto de entrada, permite mecanismos de descubrimiento implementado para los clientes, es autodocumentada, utiliza un formato JSON para el transporte liviano y directamente entendible por el cliente y algunas ventajas más que veremos a lo largo de todo el libro.

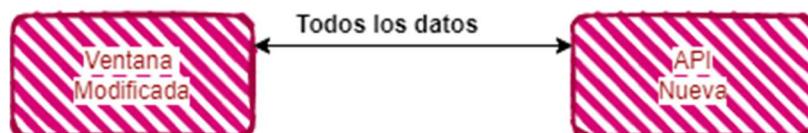
Problema que resuelve



Solución 1: Dos peticiones



Solución 2: Cambiar el API



Solución 3: Añadir un flag a la petición API

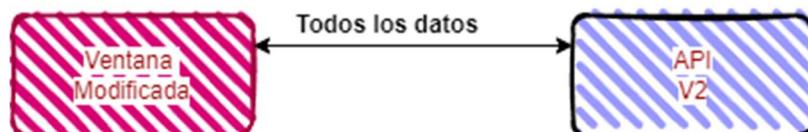


Ilustración 3 - Problemas del diseño en APIs

aproximaciones, por lo que parece lógico que haya que buscar una nueva: GraphQL.

Diseño Rest vs GraphQL

GraphQL y REST son dos enfoques distintos para diseñar una API para el intercambio de datos a través de Internet. REST permite a las aplicaciones cliente intercambiar datos con un servidor mediante verbos HTTP, que es el protocolo de comunicación estándar de Internet. Por otro lado, GraphQL es un lenguaje de consulta de API que define las especificaciones relativas a cómo una aplicación cliente debe solicitar

Vamos a describir un problema muy común que se da en los entornos de desarrollo hoy en día, teniendo en cuenta la tasa de cambio que existe en las aplicaciones web.

Partiremos de una empresa que ya tiene establecida una aplicación web con un API REST para dar servicio a los clientes. Se desea ahora que aparezca en una ventana información de la compañía que actualmente no existe.

Con la primera solución aumentamos el número de peticiones y por tanto la latencia de nuestra aplicación, puede no ser factible. Con la segunda generaremos incertidumbre a los clientes que usen nuestra API ya que será inconsistente con la anterior versión y mandamos más datos de los que realmente son necesarios. Con la tercera cada cliente tendrá que conocer el valor de un flag para los datos que necesita, generando inconsistencias y probables cambios en el código ya existente. Como podemos ver, no se soluciona completamente el problema con ninguna de las tres

datos de un servidor remoto. Puede usar GraphQL en sus llamadas a la API sin depender de la aplicación del lado del servidor para definir la solicitud.

Una API de REST es un concepto arquitectónico para la comunicación de aplicaciones. Por otro lado, GraphQL es una especificación, un lenguaje de consulta de API y un conjunto de herramientas. GraphQL funciona en un único punto de conexión mediante HTTP.

Diferencias en la solicitud del cliente

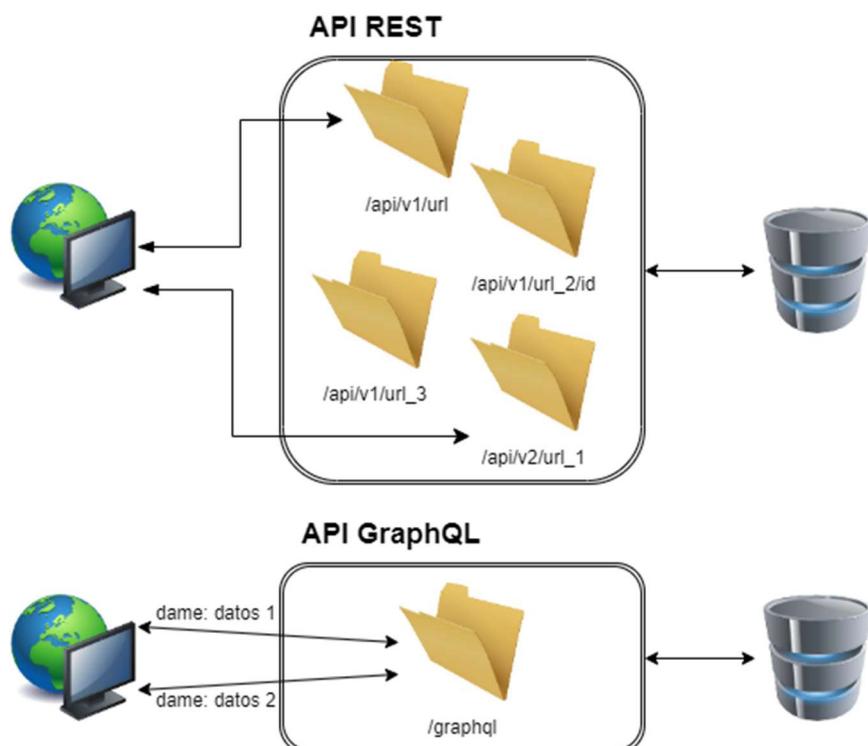


Ilustración 4 - REST vs GraphQL

Esto es lo que utiliza una solicitud de **REST** para funcionar:

- Verbos HTTP que determinan la acción (GET/POST/PATCH/PUT/DELETE).
- Una URL que identifica cada recurso en el que se realizará una acción con un verbo HTTP.
- Parámetros y valores para analizar, si desea crear o modificar un objeto en un recurso existente del servidor.
- Por ejemplo, use GET para obtener datos de solo lectura de un recurso, POST para agregar una nueva entrada de recurso o PUT para actualizar un recurso.

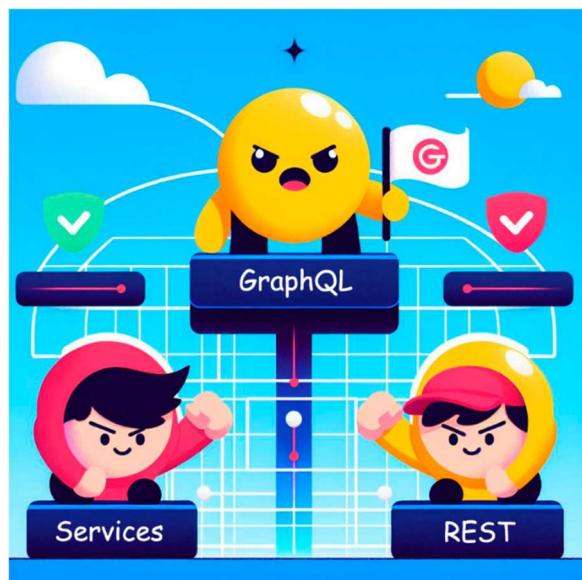
Por el contrario, esto es lo que usan las solicitudes de **GraphQL**:

- Consulta para obtener datos de solo lectura.
- Mutación para modificar datos.
- Suscripción para recibir actualizaciones de datos de streaming o basadas en eventos.
- Un formato de datos describe cómo desea que el servidor devuelva los datos, incluidos los objetos y campos que coinciden con el esquema del servidor. También puede introducir datos nuevos. Internamente, GraphQL envía cada solicitud de cliente como una solicitud HTTP POST.

Resumen de las diferencias

	REST	GraphQL
¿Qué es?	REST es un conjunto de reglas que define el intercambio de datos estructurados entre un cliente y un servidor.	GraphQL es un lenguaje de consulta, un estilo de arquitectura y un conjunto de herramientas para crear y manipular las API.
Más adecuada para lo siguiente:	REST va bien para orígenes de datos simples donde los recursos están bien definidos.	GraphQL es ideal para orígenes de datos grandes, complejos e interrelacionados.
Acceso a los datos	REST tiene varios puntos de conexión en forma de URL para definir los recursos.	GraphQL tiene un único punto de conexión de URL.
Datos devueltos	REST devuelve los datos en una estructura fija definida por el servidor.	GraphQL devuelve los datos en una estructura flexible definida por el cliente.
Cómo se estructuran y definen los datos	Los datos de REST son de tipado débil. Por lo tanto, el cliente debe decidir cómo interpretar los datos formateados cuando se devuelvan.	Los datos de GraphQL son de tipado fuerte. De este modo, el cliente recibe los datos en formatos predeterminados y mutuamente comprendidos.
Comprobación de errores	Con REST, el cliente debe comprobar si los datos devueltos son válidos.	Con GraphQL, las solicitudes no válidas suelen ser rechazadas por la estructura del esquema. Esto da lugar a un mensaje de error autogenerado.

Por qué GraphQL



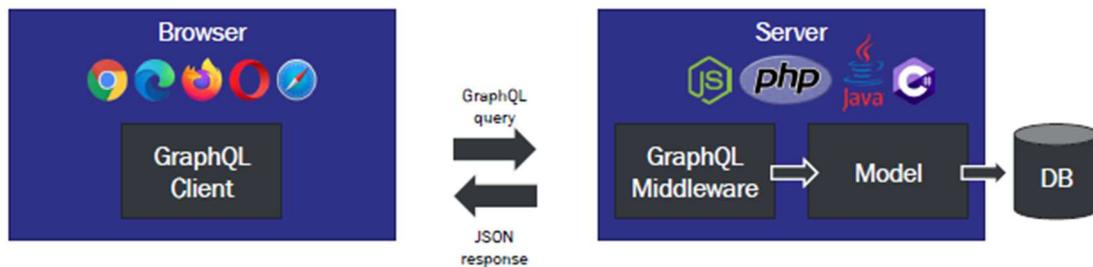
GraphQL es una nueva manera de comunicarse entre el frontend y backend solucionando los problemas que presentaban las implementaciones REST y WebServices. Este nuevo paradigma permite desarrollos flexibles y mantenibles a lo largo del tiempo, reforzando la documentación del API y permitiendo a los equipos el traspaso de información clara. Además, contiene una sintaxis fácil y clara recopilando la misma información que los correspondientes esquemas UML. Facilita la migración entre versiones del API mediante directivas y sin modificación de los puntos de acceso, así como el movimiento de la información estrictamente necesaria. Además, se adapta muy bien a las metodologías actuales ágiles, así como a las fases de análisis y diseño de las más tradicionales. En definitiva, muestra mejoras significativas con respecto a las soluciones anteriores.

Aun así, no es apta para todos los proyectos, debiendo evaluarse su uso en función de las características de los mismos. Para desarrollos pequeños, o cuyos datos sean pocos o muy estáticos a lo largo del tiempo, es posible que una aproximación a través de API REST sea más adecuada por la rápida implantación de esta tecnología y su sencillez de uso.

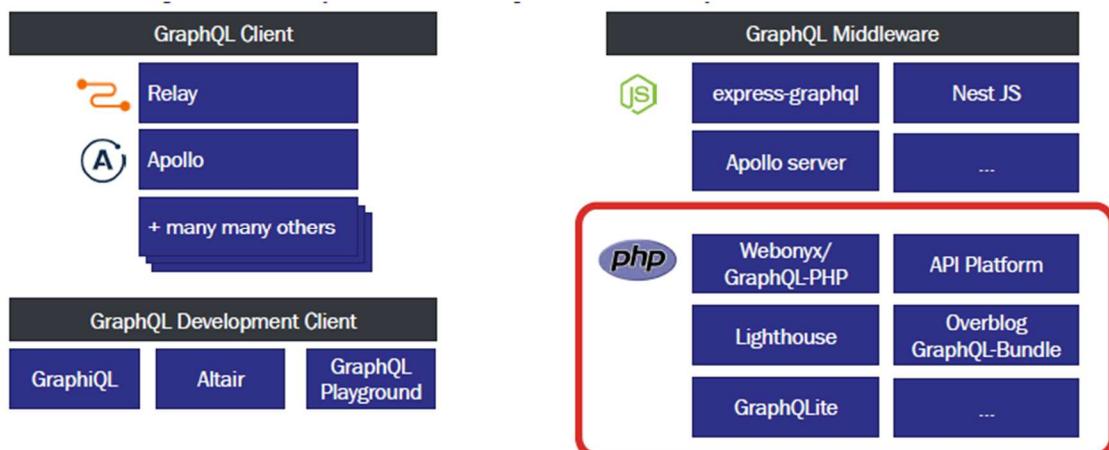
Preparando el entorno para PHP

Antes de entrar de lleno en la configuración de un entorno de desarrollo para GraphQL/PHP debemos introducir cómo se ha implantado este protocolo en las diferentes tecnologías de desarrollo web. GraphQL se desarrolla sobre todo para Nodejs, pasándose posteriormente a otros entornos.

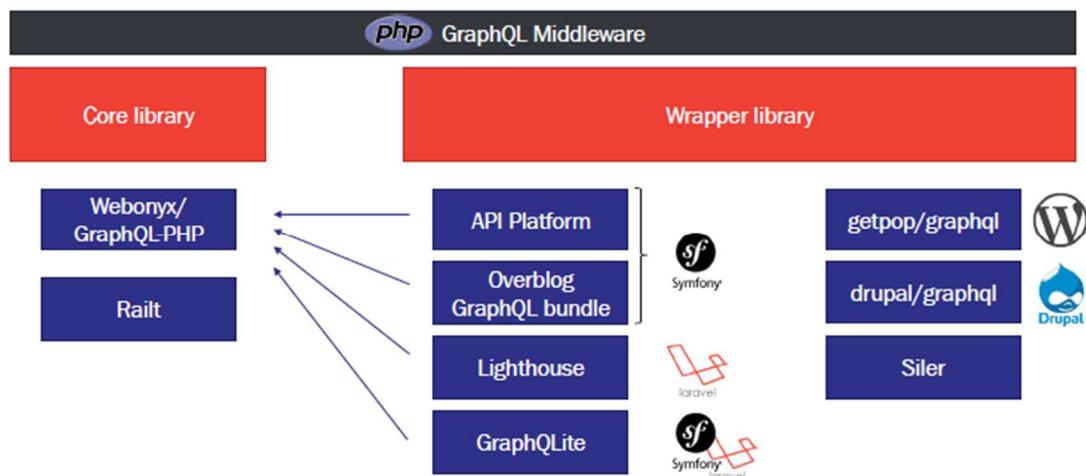
La estructura general de un proyecto sería similar a la mostrada en la siguiente figura.



Si vemos tecnologías actuales, la estructura actual del desarrollo sería similar a la siguiente.



Ya centrados exclusivamente en el backend de PHP se podría resumir como



Como podemos ver, la primera elección a realizar es si usamos la librería de base o una librería que encapsula (wrapper) la principal. En esta primera parte, se utilizará la librería base para entender todos los conceptos asociados al protocolo, valorando el uso de alguna otra librería en la parte final del libro.

Para poder desarrollar el proyecto enmarcado en este libro, deberemos crear un entorno de desarrollo similar al siguiente:

- Servidor Web Apache con Php 8.2.
- Servidor MySQL o MaríaDB.
- Composer instalado en el servidor local.
- IDE de desarrollo: Vs Code o algún otro.
- Opcional: configurar un repositorio bajo GitHub.

Servidor web

A la hora de instalar el servidor web podemos usar tanto Windows como Linux, así como WSL (Linux bajo Linux) o Docker. Elegiremos con la que estemos más cómodos en nuestro desarrollo. Solo un apunte, si elegimos Windows, tenemos que ser muy cuidadosos con los nombres de los ficheros y urls en el proyecto ya que Windows no diferencia mayúsculas y minúsculas pero los SSOO basados en Unix sí.

Una vez instalado y probado que funciona (accederemos a <http://localhost>) estableceremos dos servidores virtuales en los puertos 8080 y 8081. El primero para el frontend que apuntará a la raíz de la ruta del proyecto (dentro del directorio del servidor web htdocs), el segundo como punto de acceso del API GraphQL que señalará a un directorio llamado **graphql** dentro del raíz de la ruta de nuestro proyecto.

Como ejemplo para una instalación bajo Windows de XAMPP por defecto, buscaremos la ruta **C:\xampp\apache\conf\extra** y dentro el fichero **httpd-vhosts.conf** crearemos:

```
Listen 8080
<VirtualHost *:8080>
    ServerAdmin webmaster@www.prorutas.com
    DocumentRoot "C:/xampp/htdocs/pro_rutas"
    ServerName localhost
    ErrorLog "logs/www.prorutas.com-error.log"
    CustomLog "logs/www.prorutas.com-access.log" common
</VirtualHost>

Listen 8081
<VirtualHost *:8081>
    ServerAdmin webmaster@www.prorutas.com
    DocumentRoot "C:/xampp/htdocs/pro_rutas/graphql"
    ServerName localhost
    ErrorLog "logs/www.prorutas.com-error_graphql.log"
    CustomLog "logs/www.prorutas.com-access_graphql.log" common
</VirtualHost>
```

En este caso el directorio del servidor web está en **C:/xampp/htdocs/**, el proyecto residirá en **C:/xampp/htdocs/pro_rutas/**, el servidor virtual 8080 apunta a **C:/xampp/htdocs/pro_rutas/** y el servidor virtual 8081 apunta a **C:/xampp/htdocs/pro_rutas/graphql**.

Ahora probaremos el funcionamiento de php, con lo que crearemos un fichero llamado **info.php** en la raíz del proyecto con el siguiente contenido, accediendo posteriormente a <http://localhost/info.php>.

```
<?php  
phpinfo();
```

Como resultado aparecerá algo similar a la siguiente imagen

PHP Version 8.2.0		
System	Windows NT LAPTOP-GTVANV4I 10.0 build 22631 (Windows 11) AMD64	
Build Date	Dec 6 2022 15:26:23	
Build System	Microsoft Windows Server 2019 Datacenter [10.0.17763]	

No estaría de más que se configurara el servidor para la depuración a través de xdebug, para más información sobre este paso visitar la url <https://xdebug.org/wizard>, en la que en la caja de texto se pega la salida html de la página anterior y nos facilita instrucciones exactas de cómo realizarlo.

Instructions

1. Download [php_xdebug-3.4.2-8.2-ts-vs16-x86_64.dll](#)
2. Move the downloaded file to C:\xampp\php\ext, and rename it to `php_xdebug.dll`
3. Update `C:\xampp\php\php.ini` to have the line:
`zend_extension = xdebug`
4. Restart the Apache Webserver

El último paso sería instalar la extensión Xdebug en el navegador correspondiente y configurar el IDE para la depuración.

xdebug

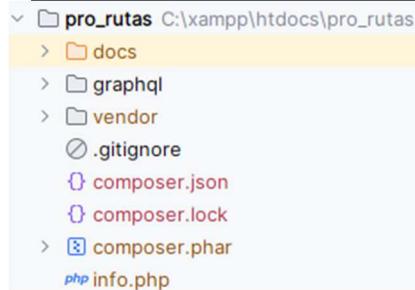
		
Version	3.2.0	
Support Xdebug on Patreon, GitHub, or as a business		
Enabled Features (through 'xdebug.mode' setting)		
Feature	Enabled/Disabled	Docs
Development Helpers	✓ enabled	[1]
Coverage	✓ enabled	[2]
GC Stats	✓ enabled	[3]
Profiler	✓ enabled	[4]
Step Debugger	✓ enabled	[5]
Tracing	✓ enabled	[6]

Ilustración 5 - Salida de info.php con xdebug habilitado

Instalación de composer

Composer es uno de los gestores de paquetes más usados en el desarrollo de aplicaciones web con php. Nos permite gestionar los paquetes de terceros, así como mapear espacios de nombre propios a directorios del proyecto evitando el uso de includes o requires. Podemos ver toda la información en la url <https://getcomposer.org/download/>. A modo de ejemplo, bajo un sistema Windows con XAMP instalado en la ruta por defecto, ejecutaremos los siguientes pasos en una ventana de comandos (cmd.exe) en la que el directorio actual sea el directorio raíz del proyecto.

```
>cd C:\xampp\htdocs\pro_rutas
>c:\xampp\php\php.exe -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"
>c:\xampp\php\php.exe -r "if (hash_file('sha384', 'composer-setup.php') ===
'dac665fdc30fd8ec78b38b9800061b4150413ff2e3b6f88543c636f7cd84f6db9189d43a81e5503cda447da73
c7e5b6') { echo 'Installer verified'.PHP_EOL; } else { echo 'Installer corrupt'.PHP_EOL; unlink('composer-
setup.php'); exit(1); }"
>c:\xampp\php\php.exe composer-setup.php
>c:\xampp\php\php.exe -r "unlink('composer-setup.php');"
```



Si el proceso ha sido correcto, podremos encontrar el fichero **composer.phar** instalado.

Tras la instalación hay que inicializar el proyecto para que use **composer** y realizar la instalación inicial de paquetes. Para este fin, ejecutaremos en el mismo terminal seleccionando los valores por defecto

```
>c:\xampp\php\php.exe composer.phar init
```

que creará el resto de la estructura (directorio vendor, composer.json) y nos dejará en disposición de realizar instalaciones de paquetes, en concreto configuraremos la librería GraphQL con:

```
>c:\xampp\php\php.exe composer.phar require webonyx/graphql-php
>c:\xampp\php\php.exe composer.phar install
```

Consideraciones sobre la instalación del resto de elementos

Con las configuraciones anteriores tendremos un entorno de desarrollo funcional, solo queda realizar unos pequeños pasos para terminar. El primero la instalación de un servidor de bases de datos basado en MySQL o MaríaDB, podemos optar por usar una instalación a través de XAMPP que nos realizará todos los pasos en cualquier plataforma, o ser nosotros mismos los que instalemos el servidor de forma manual descargado desde su página. Por último, tendremos que utilizar un IDE de desarrollo para facilitarnos la edición y ejecución de código, podemos optar por aquel que tengamos más experiencia, pero recomendaría usar VS Code o JetBrains PhpStorm.

Para finalizar el proceso, de forma totalmente optativa, podemos configurar un repositorio bajo GitHub y dar soporte al proyecto en la nube, si este es el caso, es imprescindible crear un fichero de exclusión para github (.gitignore) que no sincronice el directorio vendor, el fichero composer.phar y cualquier otro que contenga contraseñas almacenadas.

Capítulo II. GraphQL: Introducción

Qué es GraphQL

De la documentación oficial GraphQL:

GraphQL es un lenguaje de consulta para su API y un entorno de ejecución del lado del servidor para ejecutar consultas mediante un sistema de tipos estrictamente definido para los datos, de código abierto, implementado en una variedad de lenguajes de programación. GraphQL no está vinculado a ninguna base de datos o motor de almacenamiento específico, sino que está respaldado por su código y datos existentes.

GraphQL está formado por:

- Lenguaje de Esquema.
- Lenguaje de consultas.

Qué es un lenguaje de consultas

Es un sistema formal utilizado para formular solicitudes y extraer información de una fuente de datos. Proporciona una sintaxis y estructura específicas para expresar consultas, así como para definir criterios de búsqueda y filtrado de datos.

En términos más concretos, un lenguaje de consultas suele incluir comandos o expresiones para especificar los campos o atributos deseados, establecer criterios de filtrado para seleccionar registros específicos, definir condiciones de ordenamiento para clasificar los resultados y, en algunos casos, realizar operaciones de agregación o combinación de datos.

Ejemplos conocidos de lenguajes de consultas abarcan SQL (Structured Query Language), empleado en bases de datos relacionales; XQuery, utilizado para consultas en datos XML; SPARQL, utilizado para consultas en grafos RDF. Estos lenguajes permiten a los desarrolladores interactuar de manera efectiva con diversas fuentes de datos según las necesidades específicas de sus aplicaciones.

Conceptos clave

- **Esquema (Schema):** Un esquema en GraphQL es un plan estructurado que especifica los tipos de datos y las operaciones disponibles en una API GraphQL. En este esquema, se definen objetos, interfaces y enumeraciones. En términos sencillos, un esquema es una composición de diferentes definiciones que describen cómo interactuar con la API GraphQL, proporcionando un marco claro para la consulta y manipulación de datos.
- **Tipos de datos (Types):** En GraphQL, los tipos de datos son estructuras que nos dicen qué información podemos pedir y cómo podemos pedirla en una API de GraphQL. La palabra

```

1 schema {
2   query: Query
3   mutation: Mutation
4 }
5 type Mutation {
6   createOffice(input: OfficeInput!): Office
7 }
8 type Query {
9   company(id: ID!): Company
10 }
11 type Company {
12   id: ID!
13   name: String
14   address: String
15   age: Int @deprecated(reason: "No longer relevant.")
16   offices(limit: Int!, after: ID): OfficeConnection
17 }
18
19 type OfficeConnection {
20   totalCount: Int!
21   nodes: [Office]
22   edges: [OfficeEdge]
23 }
24 type OfficeEdge {
25   node: Office
26   cursor: ID
27 }
28 type Office {
29   id: ID!
30   name: String
31 }
32 input OfficeInput {
33   name: String!
34 }

```

clave type en GraphQL se utiliza para definir nuevos tipos de datos dentro de un esquema. Los tipos de datos están compuestos por escalares, que son datos básicos que representa un valor único y no tiene subcampos.

- **Consultas (Queries):** Las consultas son solicitudes enviadas por los clientes para obtener datos específicos de la API de GraphQL. Utilizando la sintaxis de consulta de GraphQL, los clientes pueden especificar los campos y relaciones que desean obtener. Las consultas permiten a los clientes obtener datos de manera precisa y eficiente. Las consultas de GraphQL son una homologación de REST de una consulta del tipo GET.
- **Mutaciones (Mutations):** Las mutaciones son operaciones de escritura utilizadas para crear, actualizar o eliminar datos en el servidor. Al igual que las consultas, las mutaciones se definen en el esquema y permiten a los clientes realizar cambios en los datos. Las mutaciones de GraphQL son una homologación de REST de una consulta del tipo PUT, POST, PATCH y DELETE.
- **Resolutores (Resolvers):** Los resolvers son funciones o métodos que se utilizan para resolver (devolver un valor asociado) los campos solicitados en una consulta. Cada campo en el esquema de GraphQL está asociado a un resolver que se encarga de obtener los datos correspondientes. Los resolvers pueden interactuar con bases de datos, servicios externos u otras fuentes de datos para obtener la información solicitada. Si no se asigna uno, se llamará al resolutor por defecto para el campo, que devuelve el valor del mismo.
- **Directivas:** Las directivas son elementos opcionales en GraphQL que se utilizan para modificar el comportamiento de las consultas y mutaciones. Permiten realizar acciones como condicionales, fragmentación y cacheo de resultados.
- **Introspección:** GraphQL proporciona la capacidad de introspección, lo que significa que se puede consultar el propio esquema de GraphQL para obtener información sobre los tipos de datos y las capacidades de la API. Esto permite a los clientes explorar y descubrir las capacidades de la API de forma dinámica.

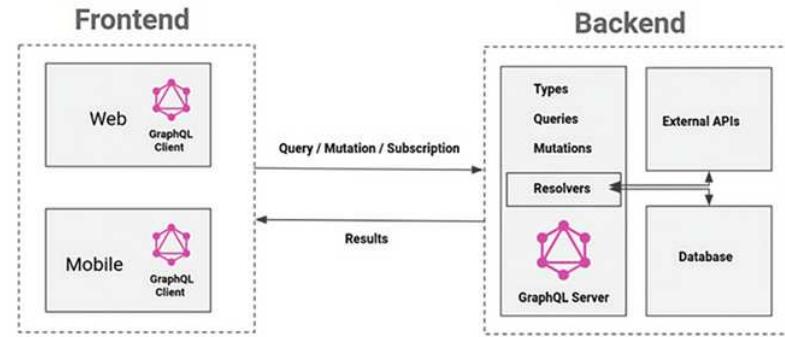
Características

- **Consulta precisa:** Permite a los clientes solicitar solo los datos específicos que necesitan, evitando la sobrecarga de datos innecesarios.
- **Tipado fuerte:** Utiliza un sistema de tipos para definir la estructura y los campos de los datos, lo que proporciona un esquema sólido y consistente.
- **Múltiples fuentes de datos:** Permite combinar y obtener datos de múltiples fuentes en una sola consulta.
- **Versionado flexible:** No requiere versionado explícito de la API, ya que los clientes pueden solicitar solo los campos que necesitan en lugar de depender de versiones específicas.
- **Realización eficiente:** Realiza múltiples solicitudes de datos en paralelo y devuelve solo los resultados necesarios, lo que mejora la eficiencia de la red.
- **Documentación automática:** El esquema GraphQL se puede utilizar para generar automáticamente documentación precisa y actualizada de la API.
- **Introspección:** Permite a los clientes obtener información sobre el esquema y las capacidades de la API en tiempo de ejecución.
- **Validación de consultas:** Permite validar las consultas en tiempo de compilación para asegurarse de que cumplan con el esquema definido.

- **Suscripciones en tiempo real:** Admite suscripciones para recibir actualizaciones en tiempo real cuando los datos cambian.
- **Granularidad en las respuestas:** Permite a los clientes especificar los campos exactos que desean recibir en la respuesta, lo que mejora la eficiencia y reduce la cantidad de datos transmitidos.

Arquitectura de GraphQL

GraphQL se basa en una arquitectura cliente-servidor. En un entorno de GraphQL, hay un servidor que implementa la API GraphQL y se encarga de recibir y procesar las consultas. Por otro lado, existe un cliente GraphQL el cual es el sistema o aplicación que envía las solicitudes a través de HTTP o cualquier otro protocolo compatible con GraphQL.



Algunos elementos clave de la arquitectura:

- **Cliente GraphQL:** Un cliente de GraphQL puede ser cualquier aplicación web, aplicación móvil u otros sistemas que consumen los datos de la API GraphQL directamente. Para interactuar con una API GraphQL, generalmente es necesario utilizar una biblioteca o cliente GraphQL que comprenda y facilite la comunicación con GraphQL. **Para realizar una consulta es obligatorio el campo query con el contenido de la misma (utilizando el lenguaje de consultas).**
- **Servidor GraphQL:** Un servidor de GraphQL es un componente o una aplicación que implementa la funcionalidad de GraphQL y permite a los clientes realizar consultas, mutaciones y suscripciones a través de una API GraphQL. Su principal función es recibir las solicitudes GraphQL, interpretarlas y proporcionar las respuestas correspondientes. Un servidor de GraphQL puede ser construido utilizando diferentes lenguajes de programación y frameworks. **Todo servidor debe implementar al menos un esquema (con el lenguaje de esquemas o mediante programación) y los resolutores asociados al mismo.**

Primer ejemplo

No hay mejor manera de entender un desarrollo que practicando, con lo que vamos a crear un ejemplo muy sencillo en el que nuestra API contestará solo a una petición: **hello** a la que responderá con “Hola Mundo”. Hay que recordar que todo el tránsito de datos se hace codificado en JSON y que la petición por defecto se tiene que realizar mediante POST (aunque no es necesario como vamos a ver).

Paso 1: Creación de esquema usando el lenguaje de esquemas. En el directorio **graphql** creamos el fichero **schema.graphql** con el siguiente contenido (Creamos un tipo consulta con una función llamada hello que devuelve una cadena).

```

type Query {
  hello: String!
}

```

Paso 2: Creamos el resolver de la función **hello**. En el directorio **graphql** creamos el fichero **rootresolver.php** con el siguiente contenido

```
<?php

$root_fileds_Resolver = [
    'hello' => 'Hola Mundo!',
];
```

Paso 3: Creamos el servidor GraphQL. En el directorio **graphql** creamos el fichero **index.php** con el siguiente contenido

```
<?php
declare(strict_types=1);
global $root_fileds_Resolver;

require_once __DIR__ . '/../vendor/autoload.php';
include_once __DIR__ . '/rootresolver.php';

use GraphQL\Utils\BuildSchema;
use GraphQL\Server\StandardServer;

$contents = file_get_contents(__DIR__. '/schema.graphql');
$schema = BuildSchema::build($contents);
$server = new StandardServer([
    'schema' => $schema,
    'rootValue' => $root_fileds_Resolver,
]);

try{
    $server->handleRequest();
}
catch (Exception $e){
    echo json_encode(['error' => $e->getMessage()]);
}
```

Paso 4: Ejecutamos desde el cliente consultas al servidor. En este caso no hay cliente y vamos a utilizar la herramienta **curl** (habrá que instalarlo si no lo tenemos ya) para realizar dichas consultas. Desde la consola de comandos ejecutamos:

```
>curl http://localhost:8081 -H "Content-Type: application/json" -H "Accept: application/json" --data-binary "{\"query\":\"query {hello}\\"}"
```

Comprobamos el resultado:

```
C:\xampp\htdocs\pro_rutas>curl http://localhost:8081 -H "Content-Type: application/json" -H "Accept: application/json" --data-binary "{\"query\":\"query {hello}\\"}"
{"data":{"hello":"Hola Mundo!"}}
C:\xampp\htdocs\pro_rutas>
```

Podemos usar también <http://localhost:8080/graphql/> como url de acceso a GraphQL

Expliquemos lo que ha pasado

Desde nuestro cliente web (**curl**) hemos mandado una petición (en este caso GET) al servidor en formato

```
{
  "query": "query {hello}"
}
```

JSON. Las consultas como veremos más adelante tienen un único campo obligatorio (**query**, en rojo en la imagen) y otro optativo (**variables**). En el campo query debemos especificar, utilizando el lenguaje de consulta GraphQL, qué es lo que queremos conseguir al llamar al API, en este caso una query llamada hello (en verde). Hay que tener cuidado no confundir el campo **query** (en rojo) que pertenece a la estructura de la consulta, este valor no puede cambiar, con el de la ejecución que solicitamos al API, el **query** que está en verde y puede ser también **_schema**, **mutation** o **subscription**.

```
{
  "data": {
    "hello": "Hola Mundo!"
  }
}
```

A dicha petición, el servidor ha respondido también en formato JSON con los datos que le hemos programado, la frase “Hola Mundo”. Como vemos la respuesta incluye un campo **data** (en rojo) con la correspondiente respuesta que incluye el nombre la acción requerida (**hello**) y los datos. En caso que existan errores en la ejecución, aparecería también al primer nivel un campo **errors**.

Como podemos ver, al ser JSON esta cadena, se puede convertir en objeto directamente en el cliente si ejecuta JavaScript.

La comunicación realizada la podemos apreciar mejor en la siguiente imagen

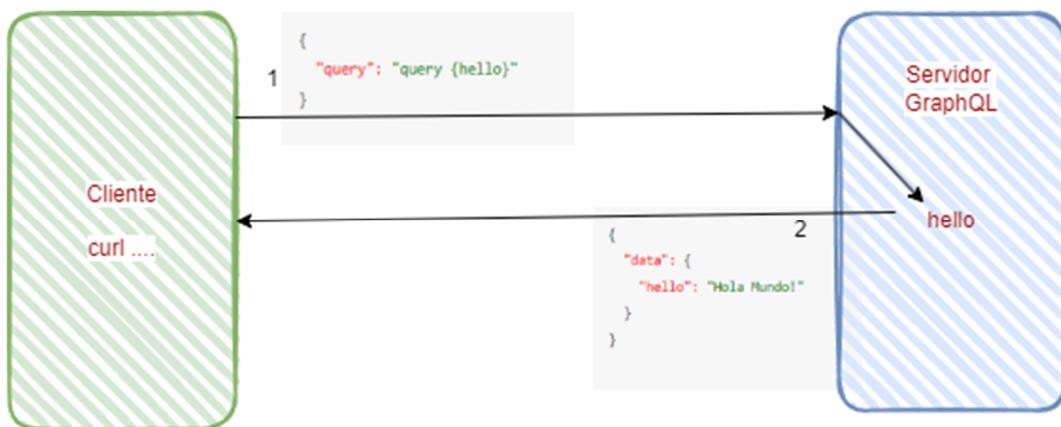


Ilustración 6 - Primer ejemplo GraphQL

Introspección

Las consultas de introspección son tipos especiales de consultas que permiten comprender el esquema de una API de GraphQL. Utilizan un tipo especial de consulta **_schema**. A continuación, vemos un ejemplo sobre el servidor que hemos creado en el capítulo

```
curl http://localhost:8081 -H "Content-Type: application/json" -H "Accept: application/json" --data-binary "{\"query\":\"{ __schema {queryType {fields {name}}}}\""}
```

```
C:\xampp\htdocs\pro_rutas>curl http://localhost:8081 -H "Content-Type: application/json" -H "Accept: application/json" --data-binary "{\"query\":\"{ __schema {queryType {fields {name}}}}\"\""} {"data": {"__schema": {"queryType": {"fields": [{"name": "hello"}, {"name": "echo"}]}}}}
```



En la respuesta vemos como existe un punto de entrada para una Query que se llama **hello**, utilizando todas las facilidades que nos brinda GraphQL podemos encontrar toda la información del API.

La capacidad de introspección que proporciona GraphQL es muy útil tanto para herramientas (IDEs, etc...) como fuente de documentación para los desarrolladores del frontend.

No vamos a profundizar en este apartado, para una referencia más exhaustiva se puede visitar la url <https://spec.graphql.org/draft/#sec-Introspection>.

Uso de la IA

El uso de los asistentes virtuales basados en IA se está extendiendo a pasos agigantados en el mundo, y la programación no es menos. Dentro de este entorno podemos utilizarlo para cuatro tareas:

- Generar consultas.
- Documentar un esquema existente.
- Generar datos de prueba
- Generar de esquemas

Vamos a ver cómo se podría usar el asistente de GitHub para estas tareas.

Generación de Consultas

A screenshot of a GitHub AI interface. At the top, there is a text input box with the following content:

Hazme una consulta graphql para la query echo con el id 34 GPT-4o

Below the input box, the AI has generated the following GraphQL query:

```
type Query {
  hello: String!
  echo(name:ID!): String!
}
```



Generación de documentación

Genérame la documentación de los tipos Graphql GPT-4o

```
1  type Query {
2    hello: String!
3    echo(name:ID!): String!
4  }
5
6  type Mutation {
7    add(id:Int!): Int!
8  }
9
10

.....
El tipo Query define las operaciones de lectura disponibles en la API GraphQL.
.....
type Query {
  .....
  Retorna un saludo en forma de cadena de texto.
  .....
  hello: String!
  .....
  Retorna un saludo personalizado utilizando el nombre proporcionado.
  .....
  echo(name: ID!): String!
}

.....
El tipo Mutation define las operaciones de escritura disponibles en la API GraphQL.
.....
type Mutation {
  .....
  Suma 5 al valor del ID proporcionado y retorna el resultado.
  .....
  add(id: Int!): Int!
}
```

Generación de datos de prueba

```

1 type Query {
2   hello: String!
3   echo(name:ID!): String!
4 }
5
6 type Mutation {
7   add(id:Int!): Int!
8 }

```

```

$testData = [
  ['query' => '{ hello }', 'expected' => 'Hola Mundo!'],
  ['query' => '{ echo(name: "Juan") }', 'expected' => 'Hello Juan!'],
  ['query' => '{ echo(name: "Maria") }', 'expected' => 'Hello Maria!'],
  ['query' => '{ echo(name: "123") }', 'expected' => 'Hello 123!'],
  ['query' => '{ echo(name: "Test") }', 'expected' => 'Hello Test!'],
];

```

Generación de esquemas



Para crear un esquema GraphQL para un médico, puedes definir los tipos **Query** y **Mutation** necesarios para manejar la información del médico. Aquí tienes un ejemplo de cómo podrías estructurarlo:

```

type Query {
  getDoctor(id: ID!): Doctor
  listDoctors: [Doctor]
}

type Mutation {
  addDoctor(name: String!, specialty: String!, yearsOfExperience: Int!):
    updateDoctor(id: ID!, name: String, specialty: String, yearsOfExperience: Int!):
    deleteDoctor(id: ID!): Boolean
}

type Doctor {
  id: ID!
  name: String!
  specialty: String!
  yearsOfExperience: Int!
}

```

Generación de un cliente html

Realizar consultas mediante la instrucción **curl** puede ser bastante tedioso, por lo que vamos a crear un cliente html/javascript para facilitarnos la labor. Para tal fin, en el directorio raíz crearemos un fichero llamado **client_graphql.html** con el siguiente contenido

```
<!DOCTYPE html>
<html lang="es">
<head>
<meta charset="UTF-8">
<title>GraphQL Client</title>
<script>
async function getQueryString() {
  const url = 'http://localhost:8080/graphql/';
  const response_id = document.getElementById('response_id');
  const query = document.getElementById('send_id').value || 'query { hello }';
  const variables = document.getElementById('variables_id').value || {};
  // {query} crear una propiedad "query" con el valor de la variable query
  console.log(JSON.stringify({query, variables}));

  fetch(url, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({query, variables}) // query: query
  })
  .then(result => result.json())
  .then(json => {
    response_id.value = JSON.stringify(json,null, ' ');
  })
  .catch(error => {
    response_id.value = error.stack;
  });
}
</script>
</head>
<body>
<table>
<tr>
<td>
<label for="send_id">query</label><br>
<textarea id="send_id" rows="10" cols="80">
```

```

query { hello }

</textarea><br>
<label for="variables_id">variables</label><br>
<textarea id="variables_id" rows="10" cols="80"></textarea>

</td>
<td><button onclick="getQueryString()">--></button></td>
<td>
  <label for="response_id">respuesta</label><br>
  <textarea id="response_id" rows="22" cols="80" readonly>
    </textarea>
</td>
</tr>
</table>
</body>
</html>

```

Accederemos mediante la url http://localhost:8080/client_graphql.html, escribiendo nuestras consultas en la parte de query, dejando de momento en blanco el campo de variables y tras pulsar en el botón podremos apreciar el resultado.

The screenshot shows a browser window with the title 'GraphQL Client'. The address bar displays the URL 'localhost:8080/client_graphql.html'. Below the address bar, there are standard navigation buttons (back, forward, search, etc.). The main content area has three sections: a 'query' input field containing the GraphQL query 'query { hello }', an empty 'variables' input field, and a 'respuesta' output field displaying the JSON response '{ "data": { "hello": "Hola Mundo!" } }'. A button labeled '-->' is positioned between the 'variables' and 'respuesta' fields.

Es muy importante no utilizar una url basada en el puerto 8081 ya que nos dará un error, hay que utilizar obligatoriamente **const url = 'http://localhost:8080/graphql/'**.

Capítulo III. GraphQL: Esquemas y tipos

Esquema

Qué es

El esquema GraphQL es la base de cualquier implementación de servidor GraphQL. Cada API de GraphQL se define mediante un solo esquema que contiene tipos y campos que describen cómo se rellenarán los datos de las solicitudes. Los datos que fluyen a través de la API y las operaciones realizadas deben validarse con respecto al esquema.

En general, el sistema de tipos de GraphQL describe las funcionalidades de un servidor GraphQL y se utiliza para determinar si una consulta es válida. El sistema de tipos de un servidor suele denominarse esquema de ese servidor y puede constar de diferentes tipos de objetos, tipos escalares, tipos de entradas, etc. GraphQL es declarativo y tiene establecimiento inflexible de tipos, lo que significa que los tipos estarán bien definidos en tiempo de ejecución y solo devolverán lo que se haya especificado.

Los esquemas de GraphQL se escriben en el lenguaje de definición de esquema (SDL). SDL está compuesto por tipos y campos con una estructura establecida:

- **Tipos:** los tipos son la forma en que GraphQL define la forma y el comportamiento de los datos. GraphQL admite una multitud de tipos. Cada tipo que se defina en su esquema tendrá su propio ámbito. Dentro del ámbito habrá uno o más campos que pueden contener un valor o una lógica que se utilice en el servicio GraphQL. Los tipos cumplen muchos roles diferentes, siendo las más comunes los objetos o los escalares (tipos de valores primitivos).
- **Campos:** los campos existen dentro del ámbito de un tipo y contienen el valor que se solicita al servicio GraphQL. Se parecen mucho a las variables de otros lenguajes de programación. La forma de los datos que defina en sus campos determinará cómo se estructuren los datos en una operación de solicitud/respuesta. Esto permite a los desarrolladores predecir lo que se va a devolver sin saber cómo se implementa el backend del servicio.

Los esquemas más simples contendrán tres categorías de datos diferentes:

- **Raíces del esquema:** las raíces definen los puntos de entrada de su esquema. Señala los campos que realizarán alguna operación con los datos, como añadir, eliminar o modificar algo. Las raíces son
 - **Query:** Representa las operaciones de lectura y recuperación de datos.
 - **Mutation:** Representa las operaciones de escritura, como crear, actualizar o eliminar datos.
 - **Subscription:** Permite suscribirse a eventos que ocurren en tiempo real.
- **Tipos:** son tipos básicos que se utilizan para representar la forma de los datos. Casi se puede pensar en ellos como objetos o representaciones abstractas de algo con características definidas. Por ejemplo, podría crear un objeto Persona que represente a una persona en una base de datos. Las características de cada persona se definirán dentro de Persona como campos. Pueden ser cualquier cosa, como el nombre, la edad, el trabajo, la dirección, etc. de la persona.

- **Tipos de objetos especiales:** son los tipos que definen el comportamiento de las operaciones del esquema. Cada tipo de objeto especial se define una vez por cada esquema. Primero se colocan en la raíz del esquema y, a continuación, se definen en el cuerpo del esquema. Cada campo de un tipo de objeto especial define una sola operación que debe implementar el resolutor (resolver).

```
type Query {  
    hello: String  
    user(id: ID!): User  
    users: [User]  
}  
  
type Mutation {  
    createUser(name: String!, email: String!): User  
}  
  
type User {  
    id: ID  
    name: String  
    email: String  
}
```

Ilustración 7 - Ejemplo Esquema

Tipos

El sistema de tipos de GraphQL describe qué datos se pueden consultar desde la API. El conjunto de esas capacidades se conoce como el esquema del servicio y los clientes pueden utilizar ese esquema para enviar consultas a la API que devuelvan resultados predecibles.

Tipos escalares predefinidos

Como todo lenguaje, GraphQL incorpora un conjunto de tipos predefinidos a usar en nuestras definiciones, intencionadamente se han limitado a cinco el número de estos tipos y el nombre es autodescriptivo. Así los tipos básicos son: **Int**, **Float**, **String**, **Boolean** e **ID**.

El tipo Int se implementa como un entero de 32 bits como signo; el tipo Float como un valor en coma flotante de doble precisión con signo; String es una cadena UTF-8 de caracteres y Boolean contendrá un valor true o false.

El tipo ID requiere un poco más de explicación. Este valor es un identificador único, generalmente unido a un objeto como clave primaria, en implementación es similar a un String, pero sin significado para la lectura.

Tipos lista

Es posible definir un tipo de un campo como una lista de objetos, utilizaremos los corchetes para esa finalidad ([]). De tal manera, un campo definido como **libros: [Libro]**, indicará que el campo libros es una lista del tipo Libro.

Tipos de objetos

```
type Doctor {
  id: ID!
  name: String!
  specialty: String!
  yearsOfExperience: Int!
}
```

Los tipos de objetos son los componentes más básicos de un esquema, representan un tipo de dato que puedes obtener, se define con la palabra clave **type** seguido del nombre del tipo. En el cuerpo aparecerán todos los campos constitutivos con el tipo correspondiente.

Los tipos se pueden usar otros tipos, tanto definidos por nosotros como existentes para crear estructuras complejas.

Componentes



Ilustración 8 - Definición de tipo de objeto

En la ilustración anterior podemos ver los componentes de un tipo de objeto. Los distintos elementos se explican por su nombre, solo el **modificador** requiere más desarrollo. La existencia de una admiración cerrada (!) al final de un tipo indicará que el valor correspondiente no puede ser nulo, siempre devolverá algún valor, este modificador es opcional. Veamos algunos ejemplos:

- **String!**, Indica que será del tipo String y no puede ser nulo.
- **[Usuario]!**, obligaría a que se devuelva al menos una lista vacía, y si no está vacía tendrá elementos del tipo Usuario.
- **[Usuario!]!**. En este caso no se podría devolver l elemento nulo, hay que devolver una lista con al menos un elemento.

Argumentos

<pre> 1 type Query { 2 hello: String! 3 echo(name:ID!): String! 4 } 5 6 type Mutation { 7 add(id:Int!): Int! 8 } </pre>	query <pre>query { echo(name:23) }</pre>	respuesta <pre>{ "data": { "echo": "Hello 23!" } }</pre>
---	--	--

Cada campo del tipo puede contener cero o más argumentos que modifiquen el valor devuelto en función de dicho parámetro. Los argumentos tienen que tener un nombre y opcionalmente un valor por defecto para definir un argumento opcional.

Como se puede ver en el ejemplo anterior, se define un campo **echo** con un parámetro obligatorio **name** del tipo **ID**. En la query se le tiene que pasar obligatoriamente el parámetro y el valor, pudiendo comprobar en la respuesta que efectivamente se utiliza.

Enumeraciones

Las enumeraciones permiten restringir el valor de un campo a los elementos descritos en dicha enumeración. Conceden un significado más concreto a los campos y expresivo.

<pre> 1 enum Episode { 2 NEWHOPE 3 EMPIRE 4 JEDI 5 } 6 7 type Query { 8 echo(name:ID!): Episode! 9 } </pre>	query <pre>query { echo(name:0) }</pre>	respuesta <pre>{ "data": { "echo": "JEDI" } }</pre>
---	---	---

Los tipos Query, Mutation y Subscription

Un esquema GraphQL soporta al menos tres tipos de operaciones al más alto nivel (**root operations**). Estas son las **Querys**, las **Mutations** y las **Subscriptions** (Consultas, Mutaciones y Suscripciones). Es obligatorio al menos definir un tipo Query, en el esquema, las mutaciones y suscripciones son optativas. Estas operaciones son las formas de ejecutar acciones sobre el API.

```

type Droid {
  id: ID!
  name: String!
  friends: [Character]
  appearsIn: [Episode]!
  primaryFunction: String
}

type Query {
  droid (id: ID!): Droid
}

```

Operación (desde cliente)	Respuesta (servidor)
<pre>{ droid (id: "2000") { name } }</pre>	<pre>{ "data": { "droid": { "name": "C-3PO" } } }</pre>

Una Query indica al sistema que se desea un conjunto de datos, dicha petición vendrá en formato JSON indicando exactamente la estructura desde el cliente y qué datos necesitamos, se devolverá el conjunto resultante de los mismos también en formato JSON. Ver figura anterior.

Una **Mutation** se define por parte del API cuando permite operaciones de modificación sobre algún elemento del sistema, por ejemplo, inserción de datos, modificación, etc. Por último, una **Subscription** indica una operación de suscripción (sindicación) en la que el servidor se compromete a enviar datos en streaming al cliente. Definiremos estos tipos y sus campos como cualquier tipo de objeto con el nombre que deseemos y a la hora de crear el esquema se le facilitará, como mínimo la Query.

```

enum Episode {
  NEWHOPE
  EMPIRE
  JEDI
}

type MyQuery {
  echo(name:ID!): Episode!
}

type MyMutation {
  echo(name:ID!): Episode!
}

type MySubscription {
  echo(name:ID!): Episode!
}

schema {
  query: MyQuery,
  mutation: MyMutation,
  subscription: MySubscription
}

```

Ilustración 9 - Ejemplo root operations

Interfaces

En un lenguaje de programación, un interfaz es un conjunto de funciones o métodos que deben ser implementados obligatoriamente por la clase que lo usa. Eso facilita la implementación de protocolos al compartir una especificación única. En el caso de GraphQL que trabajamos con tipos, el interfaz definirá un conjunto de campos que debe **obligatoriamente** implementar el tipo que lo use. Los interfaces son muy útiles para devolver diferentes tipos, siempre que todos ellos implemente el mismo interfaz.

Los interfaces pueden a su vez implementar otros interfaces, generándose un grafo de uso.

En la imagen en la cabecera de la sección se definen los tipos necesarios para crear el esquema, a continuación, vemos el interfaz (**LibraryItem**) que es implementado por **Book** y por **BoardGame**. Para finalizar se crea un punto de entrada **libraryItems** en las consultas que devolverá una lista de LibraryItem (el interfaz) permitiendo que esta consulta retorne tanto Book como BoardGame.

```

scalar Date

type Publisher {
    name: String!
}

type Author {
    books: [Book!]!
    firstName: String!
    lastName: String!
}

interface LibraryItem {
    id: ID!
    title: String!
}

type Book implements LibraryItem {
    id: ID!
    title: String!
    authors: [Author!]!
}

type BoardGame implements LibraryItem {
    id: ID!
    title: String!
    publisher: Publisher!
}

type Query {
    libraryItems: [LibraryItem!]
}

schema {
    query: Query
}

```

Definición de interfaz

Book lo implementa

BoardGame lo implementa

obligatorio para implementarlo

Como nota final, hay que indicar que los interfaces no pueden implementarse así mismos de forma cíclica o contenerla dentro.

La consulta realizada en el cliente tiene que explicar un poco, aunque se dedicará un capítulo completo a la creación de consultas más adelante. Se pide una lista de elementos (**libraryItems**) y de cada elemento necesitamos los campos **__typename**, **id** y **title**. Además, dependiendo de que el tipo devuelto (... on ...) sea un libro (**Book**) que se devolverá todos los autores con su nombre (**firstName**) o sea un juego de mesa (**BoardGame**) se devolverá el editor con su nombre (**name**). El campo **__typename** es un metacampo generado por el API que siempre existe que contendrá la clase del elemento devuelto.



Ilustración 10 - Petición diseño con interfaces

Uniones

Si los interfaces se definen cuando queremos compartir información entre tipos, el concepto de unión es la posibilidad de utilizar dos tipos diferenciados a la vez sin compartir información. Siendo posible también devolver cualquier de ellos. Los tipos definición en una unión tienen que ser concretos, no pueden ser interfaces u otras uniones.

```

union LibraryItem = Book | BoardGame

type Query {
  libraryItems: [LibraryItem!]! Devuelve tanto Book como BoardGame
}

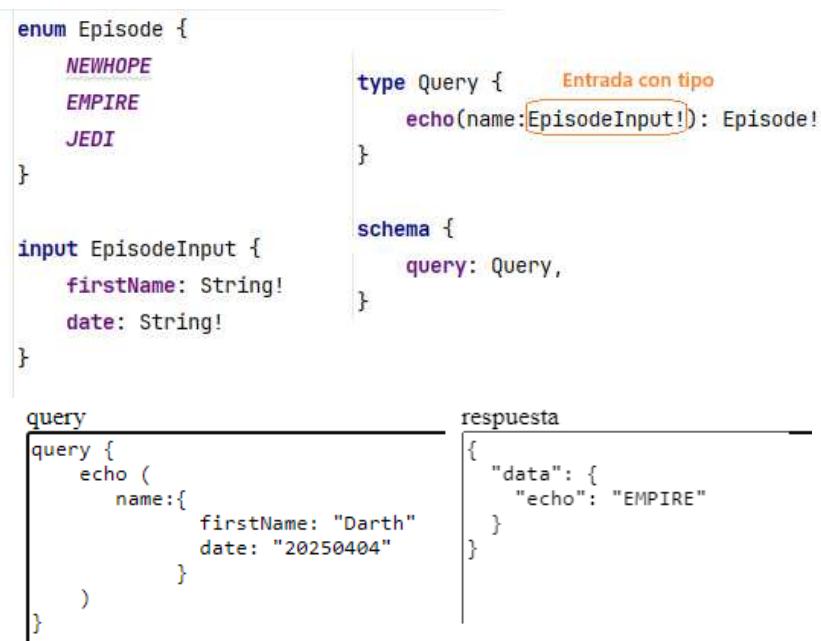
schema {
  query: Query
}
  
```



Ilustración 11 - Diseño con uniones

Si nos fijamos muy bien en la query hecha desde el cliente podemos apreciar un pequeño cambio con respecto a la que hicimos en los interfaces, en esta es obligatorio repetir cada campo en ambos tipos (**id**, **title**) mientras que en la de los interfaces no era necesario, aparecía en la parte común.

Inputs



Hasta ahora los argumentos de los campos solo podían ser tipos simples, un lenguaje con esa limitación no sería adecuado, por lo que GraphQL soporta el paso de argumentos complejos a través de los inputs. Definiremos un tipo input con los campos que deberán pasarse al parámetro correspondiente y a continuación lo usaremos en la definición del campo, como se puede ver en el ejemplo siguiente.

Esta facilidad es muy útil sobre todo en las mutaciones, en la que hay que pasar gran información al API para que realice su trabajo como, por ejemplo, todos los datos de una persona a dar de alta. **Los inputs no pueden tener definidos argumentos.**

Directivas

Las directivas son anotaciones que hacemos a los tipos, campos o argumentos de un esquema para validar o ejecutar código sobre ellos. El estándar GraphQL tiene varios implementados de uso muy común. Por ejemplo:

- **@deprecated.** Esta directiva se utilizará en vez de eliminar un campo o tipo que haya cambiado a lo largo del tiempo, de esta manera damos a los clientes información sobre la evolución del API y permitimos que se adapten a ella de forma eficiente.

```
type User {  
    fullName: String  
    name: String @deprecated(reason: "Use `fullName`.")  
}
```

Existe la posibilidad también de definir nuestras propias directivas para que modifique el comportamiento de los tipos cuando sean requeridos. Así podríamos hacer que un campo solo se pudiera acceder si se tiene un nivel de privilegio concreto.

Documentación y comentarios

GraphQL le permite agregar documentación a los tipos, campos y argumentos de un esquema. De hecho, la especificación GraphQL lo alienta a hacer esto en todos los casos, a menos que el nombre del tipo, campo o argumento sea autodescriptivo. Las descripciones de esquema se definen mediante sintaxis Markdown y pueden ser de una o varias líneas (para un ejemplo ver Generación de documentación).

En ocasiones, es posible que necesite agregar comentarios en un esquema que no describan tipos, campos o argumentos y que no estén destinados a ser vistos por los clientes. En esos casos, puede agregar un comentario de una sola línea al SDL anteponiendo un # carácter al texto. También se puede añadir comentarios a las consultas cliente.

Diseño de los esquemas con POO en vez de SDL

El diseño a través del lenguaje SDL es muy útil para la descripción, pero generalmente se utilizan técnicas basadas en programación en vez de diseño. Podemos utilizar cualquier de las dos aproximaciones tradicionales: programación estructurada o programación orientada a objetos. En este caso, por claridad nos vamos a centrar en la POO por la claridad de diseño. Veamos un ejemplo.

SDL DE PARTIDA

scalar Date

type Publisher {

name: String!

}

type Author {

books: [Book!]!

firstName: String!

lastName: String!

}

type Book {

id: ID!

title: String!

authors: [Author!]

}

type BoardGame {

id: ID!

title: String!

publisher: Publisher!

}

union LibraryItem = Book | BoardGame

type Query {

libraryItems: [LibraryItem!]

}

schema {

query: Query

}

CREACIÓN BAJO POO

<?php

```
use GraphQL\Type\Definition\Type;
```

```
use GraphQL\Type\Definition\ObjectType;
```

```
use GraphQL\Type\Definition\UnionType;
```

```
use GraphQL\Type\Schema;
```

```
class Author extends ObjectType
```

```
{  
    public function __construct()  
    {  
        parent::__construct([  
            'fields' => [  
                'firstName' => Type::nonNull(Type::string()),  
                'lastName' => Type::nonNull(Type::string()),  
                'books' => Type::listOf(Type::string())  
            ]  
        ]);  
    }  
}  
  
class Publisher extends ObjectType  
{  
    public function __construct()  
    {  
        parent::__construct([  
            'fields' => [  
                'name' => Type::nonNull(Type::string())  
            ]  
        ]);  
    }  
}  
  
class Book extends ObjectType  
{  
    public function __construct()  
    {  
        parent::__construct([  
            'fields' => [  
                'id' => Type::nonNull(Type::id()),  
                'title' => Type::nonNull(Type::string()),  
                'authors' => Type::listOf(MyTypes::autor())  
            ]  
        ]);  
    }  
}  
  
class BoardGame extends ObjectType  
{  
    public function __construct()  
    {
```

```

parent::__construct([
    'fields' => [
        'id' => Type::nonNull(Type::id()),
        'title' => Type::nonNull(Type::string()),
        'publisher' => MyTypes::publisher()
    ],
]);
}

}

class UnionLibraryItem extends UnionType {
    public function __construct() {
        parent::__construct(
            [
                'types' => [
                    MyTypes::book(),
                    MyTypes::boardGame()
                ],
                'resolveType' => function ($value) : ObjectType {
                    if (isset($value['authors'])) {
                        return MyTypes::book();
                    }
                    if (isset($value['publisher'])) {
                        return MyTypes::boardGame();
                    }
                    throw new Exception("Unknown type");
                },
            ]
        );
    }
}

class MyTypes
{
    /*
     * Creamos esta clase porque el tipo debe ser único en el esquema
     *
     * y usamos el patrón singleton
     */
    private static UnionLibraryItem $unionLibraryItem;
    private static Book $bookObjectType;
    private static BoardGame $boardGameObjectType;
    private static Author $authorObjectType;
    private static Publisher $publisherObjectType;
}

```

```
public static function unionLibraryItem(): UnionLibraryItem
{
    return self::$unionLibraryItem ??= new UnionLibraryItem();
}

public static function book(): Book{
    return self::$bookObjectType ??= new Book();
}

public static function boardGame(): BoardGame{
    return self::$boardGameObjectType ??= new BoardGame();
}

public static function autor(): Author{
    return self::$autorObjectType ??= new Author();
}

public static function publisher(): Publisher{
    return self::$publisherObjectType ??= new Publisher();
}

}

$schema = new Schema([
    'query' => new ObjectType([
        'name' => 'Query',
        'fields' => [
            'libraryItems' => [
                'type' => Type::listOf(MyTypes::unionLibraryItem()),
                'resolve' => function () {
                    $books = [
                        [
                            'id' => '1',
                            'title' => 'Book 1',
                            'authors' => [
                                ['firstName' => 'Author',
                                    'lastName' => 'One', 'books' => []]
                            ]
                        ],
                        [
                            'id' => '2',
                            'title' => 'Book 2',
                            'authors' => [
                                ['firstName' => 'Author']
                            ]
                        ]
                    ];
                    return $books;
                }
            ]
        ]
    ])
]);
```

```

        'lastName' => 'Two', 'books' => []
    ]
]
];
};

$boardGames = [
[
'id' => '1',
'title' => 'Board Game 1',
'publisher' => ['name' => 'Publisher 1']
],
[
'id' => '2',
'title' => 'Board Game 2',
'publisher' => ['name' => 'Publisher 2']
]
];
return (array_merge($books, $boardGames));
}
]
],
]);
]);

```

Cada tipo se convierte en una clase en la que, en su constructor, al llamar al parente, se establece la configuración de campos y atributos, esta definición tendrá un campo **resolve** con la función que será llamada en el campo cuando este tenga que resolverse en una consulta, si no se crea este campo se llamará al resolver por defecto que devuelve el valor de la propiedad correspondiente.

Lo más notable de este tipo de programación es la necesidad de una clase que implemente el patrón Singleton para cada uno de los tipos. Cuando se define el tipo, solo se puede crear un objeto por cada clase, que se deberá usar a lo largo de toda la definición. Con esta clase (MyTypes) nos aseguramos que solo haya un objeto por cada tipo de objeto.

En el ejemplo anterior no se muestra cómo crear interfaces, lo mostramos a continuación:

```

class LibraryItem extends InterfaceType{
public function __construct()
{
parent::__construct([
'fields' => [
'id' => Type::nonNull(Type::id()),
'title' => Type::nonNull(Type::string())
],
'resolveType' => function ($value) : ObjectType {

```

```
if (isset($value['authors'])) {
    return MyTypes::book();
}
if (isset($value['publisher'])) {
    return MyTypes::boardGame();
}
throw new Exception("Unknown type");
});
}
}

class Book extends ObjectType
{
    public function __construct()
    {
        parent::__construct([
            'interfaces' => [MyTypes::LibraryItem()],
            'fields' => [
                'id' => Type::nonNull(Type::id()),
                'title' => Type::nonNull(Type::string()),
                'authors' => Type::listOf(MyTypes::autor())
            ]
        ]);
    }
}
```

En este caso, a la hora de definir el interfaz se tiene que implementar un campo `resolveType` en el que determinaremos el tipo del valor que nos pasan (`$value`).

Capítulo IV. GraphQL: Consultas

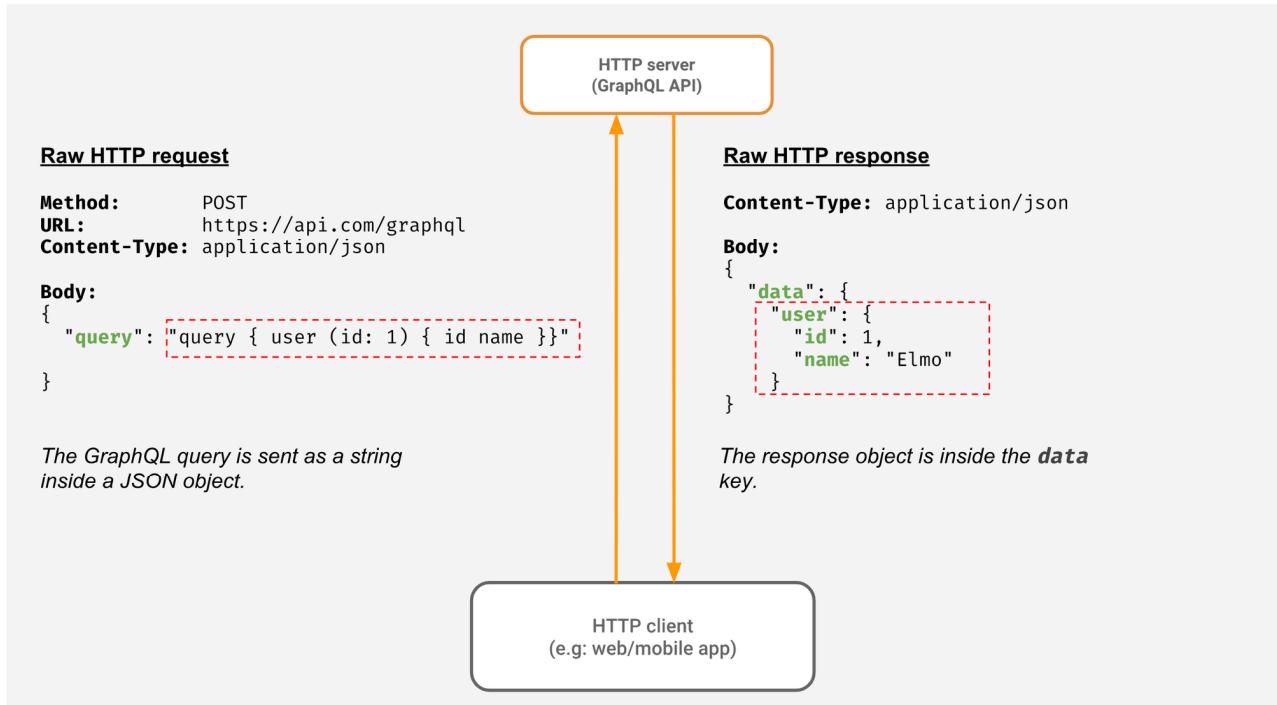


Ilustración 12 - Transmisión Cliente Servidor

Qué es el lenguaje de consultas

Es un sistema formal utilizado para realizar solicitudes y extraer información de una fuente de datos. Proporciona una sintaxis y estructura específicas para expresar consultas, así como para definir criterios de búsqueda y filtrado de datos. En el desarrollo de APIs bajo GraphQL el formato que se debe transmitir la consulta es JSON.

Estructura de una consulta desde el cliente

Para poder realizar consultas a través de un cliente tenemos que definir una estructura muy concreta para que se entienda por parte del servidor, esta estructura está marcada en el estándar. Crearemos un objeto en formato JSON formado por dos campos, uno: **query** (obligatorio) y otro: **variables** (optativo). Recordar que, si dentro de algún campo tenemos que usar las comillas dobles, habrá que anteponer la barra invertida (\). Estructura de la consulta cliente:

```
{
  "query": "consulta GraphQL (Aquí pondremos la definición de la consulta)",
  "variables": "definición de variables y sus valores"
}
```

Un ejemplo real de código a mandar desde cliente sería:

```
{ "query": "
  query {
    hello
  }
  "}
```

{ }

Los clientes y servidores GraphQL deben admitir JSON para la serialización y pueden admitir otros formatos. Los clientes también deben indicar qué tipos de medios admiten en las respuestas mediante el Accept encabezado HTTP. En concreto, el cliente debe incluir el application/graphql-response+json en el Accept encabezado.

Si no se incluye información de codificación con este encabezado, se asumirá que es utf-8 . Sin embargo, el servidor puede responder con un error si un cliente no proporciona el Accept encabezado con una solicitud.

Para garantizar la compatibilidad, si un cliente envía una solicitud a un servidor GraphQL heredado, el encabezado Accept también debe incluir el tipo de medio application/json de la siguiente manera:

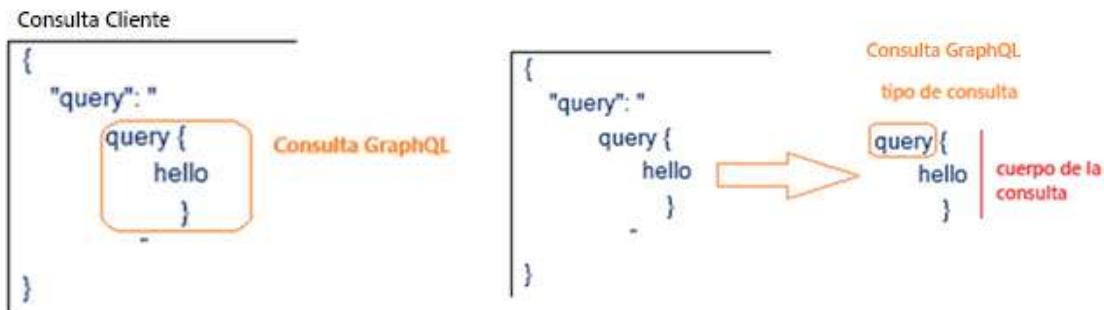
```
application/graphql-response+json; charset=utf-8,  
application/json; charset=utf-8
```

El servidor HTTP de GraphQL debe gestionar el método HTTP POST para las operaciones de consulta y mutación, y también puede aceptar el método GET para las operaciones de consulta.

Componentes de una consulta GraphQL

De la consulta anterior (consulta cliente), el campo **query**, contendrá el texto que desarrollaremos en este punto y siguientes, la consulta GraphQL, remarcada en negrita.

La estructura de una consulta GraphQL es siempre la misma: primero el tipo de operación raíz (tipo de consulta) y a continuación el cuerpo de la consulta en la que se definirán los campos que queremos recoger. En caso que el tipo sea **query**, se puede omitir, pero no lo vamos a hacer en este manual por claridad.



Los **tipos de consulta** coinciden con las operaciones raíz: query, mutation y subscription. Teniendo el significado ya explicado anteriormente en el capítulo anterior.

Campos

En su forma más simple, GraphQL consiste en solicitar campos específicos en objetos llegando de forma recursiva hasta un campo simple o una enumeración. Si los campos tienen argumentos, y son obligatorios, deberán aparecer a continuación del nombre del campo, entre paréntesis, con el nombre del argumento.

En un sistema como REST, solo se puede pasar un único conjunto de argumentos (los parámetros de consulta y los segmentos de URL de la solicitud). Pero en GraphQL, cada campo y objeto anidado puede obtener su propio conjunto de argumentos, lo que convierte a GraphQL en un reemplazo completo para realizar múltiples búsquedas de API.

| Operación | Respuesta |
|--|--|
| <pre>query { human(id: "1000") { name height } }</pre> | <pre>{ "data": { "human": { "name": "Luke Skywalker", "height": 1.72 } } }</pre> |

La especificación GraphQL indica que el resultado de una solicitud se devolverá en una clave **data** de nivel superior en la respuesta. Si la solicitud generó algún error, habrá información sobre qué salió mal en una clave **errors** de nivel superior. La especificación GraphQL gestiona los errores de resolución como hemos dicho, pero aún siguen quedando otro tipo de errores que el cliente tendrá que controlar, todos aquellos relacionados con el protocolo que sean diferentes al OK – 200 y posibles problemas de red.

| query | respuesta |
|---|---|
| <pre>query { human(idx:"1000"){ name height } }</pre> | <pre>{ "errors": [{ "message": "Unknown argument \\"idx\\" on field \\"human\\" of type \\"Query\\". Did you mean \\"id\\"?", "locations": [{ "line": 2, "column": 11 }] }] }</pre> |

Las consultas GraphQL pueden recorrer objetos relacionados y sus campos, lo que permite a los clientes obtener una gran cantidad de datos relacionados en una sola solicitud, en lugar de realizar varios viajes de ida y vuelta como se necesitaría en una arquitectura REST clásica.

Aliases

Una de las facilidades que proporciona GraphQL es la capacidad de realizar en una misma consulta varias consultas que de otra forma serían diferentes en un API REST. Por ejemplo, pedir dos usuarios diferentes a la vez, etc. El problema que se plantea es la identificación de cada una de las respuestas dadas por el servidor. Para este fin aparecen los **Alias**, por el que anteponemos un nombre a la subconsulta que

queremos realizar, y el API nos lo incluirá como campo antes del correspondiente resultado. Veamos un ejemplo.

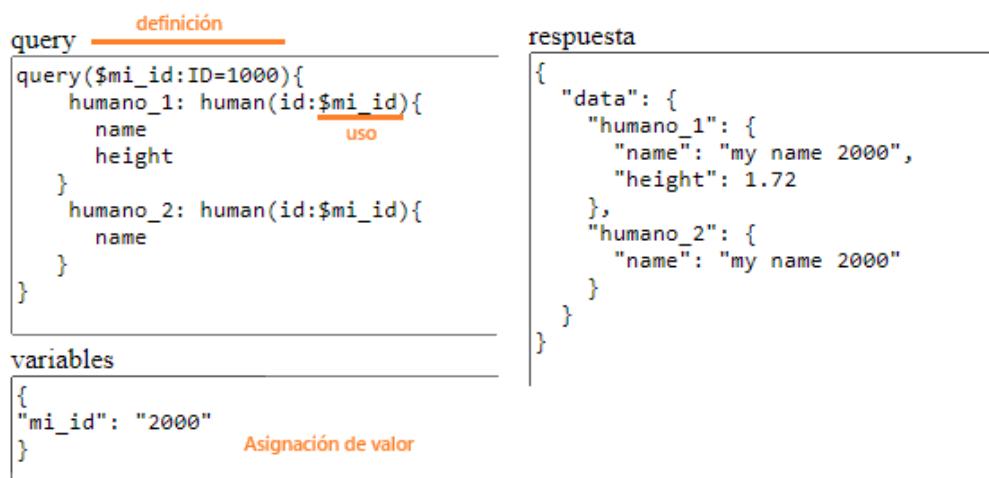


Variabes

En la mayoría de las consultas, los valores que deseamos utilizar como filtro suelen ser variables, en los ejemplos anteriores los hemos escrito en hardcode dentro de la misma consulta, pero esto nos llevaría a tener que escribir en el cliente la consulta cada vez que cambie el valor y generarla nueva. Para evitar este problema, podemos usar dentro de la consulta variables, y añadir el campo: **variables** al más alto nivel con la correspondencia de las variables con sus valores, para que sea el servidor el que haga uso de dicho valor en el lugar que corresponde, permitiendo que la consulta no cambie a lo largo del programa.

Para usar de forma correcta las variables hay que seguir tres pasos

1. Definir el uso de la variable de la consulta en la cabecera, el nombre utilizará el \$ (dólar) obligatoriamente como primer carácter y a continuación un tipo que podrá ser simple o uno definido, en caso que necesitemos un tipo más complejo, deberá ser un **input** existente en el servidor. También podremos asignar un valor por defecto a la misma para que su uso se opcional.
2. Usar la variable dentro de la consulta, en el lugar que corresponda.
3. Crear el campo: **variables** y asignarlo con el valor correspondiente para la variable



Podemos ver cómo se define la variable en la cabecera y se le otorga un valor por defecto, esto es obligatorio, a continuación, como se usa en la propia consulta y como se le asigna valor en la sección de variables.

La petición real que se hace desde el cliente de la primera consulta es la que se muestra a continuación:

```
{
  "query":"
    query($mi_id:ID=1000){
      humano_1: human(id:$mi_id){
        name
        height
      }
      humano_2: human(id:$mi_id){
        name
      }
    },
    "variables":{
      \"mi_id\": \"2000\"
    }
  "
}
```

En la que se puede apreciar la existencia la existencia de los campos **query** y **variables** para acomodar los datos que necesita el servidor GraphQL. En la imagen siguiente se muestra un uso un poco más avanzado, con dos variables distintas.

The diagram illustrates a GraphQL request and its corresponding response. On the left, under 'query', there is a complex query involving two variables (\$mi_id and \$mi_id_2). The 'variables' section shows the values assigned to these variables. An arrow points from the 'query' section to the 'resposta' section on the right, which displays the JSON data returned by the server. The JSON data contains a 'data' field with two entries: 'humano_1' and 'humano_2', each containing their respective names and heights.

```

query
query($mi_id:ID=1000 $mi_id_2:ID=1000){
  humano_1: human(id:$mi_id){
    name
    height
  }
  humano_2: human(id:$mi_id_2){
    name
  }
}

variables
{
  "mi_id": "2000",
  "mi_id_2": "3000"
}

resposta
{
  "data": {
    "humano_1": {
      "name": "my name 2000",
      "height": 1.72
    },
    "humano_2": {
      "name": "my name 3000"
    }
  }
}
  
```

Es posible que algunas implementaciones del servidor obliguen a nombrar la consulta, es decir después del tipo, en este caso **query**, debe aparecer un nombre que nos será devuelto en los resultados.

```
{
  "query":"
    query nombre_de_la_consulta ($mi_id:ID=1000){
```

Fragmentos

Cuando las consultas se complican, es muy normal que tengamos que repetir campos a lo largo de la consulta, cuando el número de campos repetidos es mayor de dos, aparece la necesidad de evitar esta repetición. Para solucionar este problema se crean los fragmentos.

| | |
|--|---|
| query
<pre>query(\$mi_id:ID=1000){ humano_1: human(id:\$mi_id){ ...campos } humano_2: human(id:\$mi_id){ ...campos } } fragment campos on Human { name height }</pre> | respuesta
<pre>{ "data": ["humano_1": { "name": "my name 2000", "height": 1.72 }, "humano_2": { "name": "my name 2000", "height": 1.72 }] }</pre> |
| variables
<pre>{ "mi_id": "2000" }</pre> | |

La definición de un fragmento se hace fuera de la consulta, aunque se utilice en ella. Comienza con la palabra clave **fragment**, seguido del nombre que queremos utilizar (campos) y a continuación el tipo sobre el que estamos creando el fragmento (**on Human**). No es necesario incluir todos los campos del tipo en el fragmento, solo aquellos que necesitemos, y podremos crear tantos fragmentos como creamos convenientes referidos al mismo tipo u a otros. Para usarlo en la consulta, utilizaremos tres puntos (...) para indicar la posición en donde se insertará el fragmento seguido del nombre del fragmento.

Directivas

El concepto de directiva ya se vio en el capítulo anterior (@deprecated), por lo que es conocido. En este caso las directivas ya implementadas son las siguientes:

- **@include(if: Boolean)**. Solo incluye este campo o fragmento en el resultado si el argumento es verdadero.

```
query Hero($episode: Episode, $withFriends: Boolean!) {
  hero(episode: $episode) {
    name
    friends @include(if: $withFriends) {
      name
    }
  }
}
```

- **@skip(if: Boolean.)** Salta el campo o fragmento si la condición es verdadera.

Es posible definir nuestras propias directivas en el esquema, y estas permitirán también el uso de parámetros. Tendremos que decidir cómo implementarlas durante la ejecución de la consulta

Ejemplos de consultas GraphQL de introspección

Para tener una referencia completa sobre este tipo de consultas, visitar:

- <https://spec.graphql.org/June2018/#sec-Introspection>
- <https://graphql.org/learn/introspection/>

A continuación, vemos la consulta para recoger la mayoría de información, se rellena solo la sección **types**, pero se puede extender al resto de elementos: **queryType**, **mutationType**, **subscriptionType** y **directives**. En este tipo de consultas, el campo raíz a utilizar es **__schema**, realizando dentro la selección de información, no es necesario ni mucho menos incluir todos los campos que aparecen.

```
query{  
  __schema{  
    types{  
      name  
      kind  
      description @include(if: false)  
      interfaces{  
        name  
      }  
      fields (includeDeprecated:true){  
        name  
        type{  
          kind  
        }  
        description  
        isDeprecated  
        deprecationReason  
        args{ name type {kind}}  
      }  
    }  
    queryType{  
      name  
      description  
    }  
    mutationType{name}  
    subscriptionType{name}  
    directives{name locations}  
  }  
}
```

Parte del resultado devuelto de la consulta de introspección:

```
{ "data": {  
    "__schema": {  
        "types": [  
            {  
                "name": "Query",  
                "kind": "OBJECT",  
                "interfaces": [],  
                "fields": [  
                    {  
                        "name": "human",  
                        "type": {  
                            "kind": "NON_NULL"  
                        },  
                        "description": null,  
                        "isDeprecated": false,  
                        "deprecationReason": null,  
                        "args": [  
                            {  
                                "name": "id",  
                                "type": {  
                                    "kind": "NON_NULL"  
                                }  
                            ...  
                        },  
                        ...  
                    ],  
                    "queryType": {  
                        "name": "Query",  
                        "description": null  
                    },  
                    "mutationType": null,  
                    "subscriptionType": null,  
                    "directives": [  
                        ...  
                    ]  
                }  
            }  
        ]  
    }  
}
```

Ejemplos de consultas GraphQL mutaciones

Las mutaciones son las operaciones que presenta el API para realizar cambios en el sistema subyacente. Hay que tener en cuenta que según la especificación GraphQL, la resolución de campos distintos de los campos de mutación (**mutation**) de nivel superior siempre debe estar libre de efectos secundarios, por lo que solo este tipo es el que puede realizar cualquier tipo modificaciones. Todo lo que se aplica a las consultas se puede aplicar a este apartado.

```

enum Episode {
  NEWHOPE
  EMPIRE
  JEDI
}

input ReviewInput {
  stars: Int!
  commentary: String
}

type Mutation {
  createReview(episode: Episode, review: ReviewInput!): Review
}

```

| Operation | Response |
|---|---|
| <pre> mutation CreateReviewForEpisode(\$ep: Episode!, \$review: ReviewInput!) { createReview(episode: \$ep, review: \$review) { stars commentary } } </pre> | <pre>{ "data": { "createReview": { "stars": 5, "commentary": "This is a great movie!" } } }</pre> |
| Variables | |
| <pre>{ "ep": "JEDI", "review": { "stars": 5, "commentary": "This is a great movie!" } }</pre> | |

Tenemos que tener en cuenta que el servidor GraphQL por sí solo no realiza ninguna acción de la **mutation**, debe ser en el correspondiente resolutor (resolver) el que nosotros programemos el código que realice la acción y devuelva un resultado según el tipo indicado en el esquema. Se abordará este tema en un capítulo posterior.

El ejemplo anterior muestra una distinción importante con respecto al API REST. Para actualizar las propiedades de un objeto mediante una API de REST, es probable que envíe los datos actualizados a un punto de conexión generalizado para ese recurso mediante una solicitud http **PATCH**. Con GraphQL,

simplemente crearemos una mutación, con campos diseñados para la tarea en cuestión. GraphQL también nos permite expresar relaciones entre datos que serían más difíciles de modelar semánticamente con una solicitud básica de estilo CRUD. Por ejemplo, es posible que un usuario desee guardar una valoración personal de una película. Las mutaciones permiten implementar el patrón CRUD directamente en distintas mutaciones, sin la necesidad de usar distintos tipos de solicitud http (DELETE, POST, GET, PATCH, PUT).

Las mutaciones al igual que las consultas, se pueden mandar en una única petición varias, nombrándolas con un alias tal y como se vio en las consultas, pero hay que tener en cuenta que estas peticiones se ejecutarán de forma síncrona según el orden que hayamos creado la mutación de arriba abajo, pero cada una de forma independiente.

Ejemplos de consultas GraphQL suscripciones

la especificación GraphQL también describe cómo recibir actualizaciones en tiempo real mediante solicitudes de larga duración.

```
type Subscription {  
    reviewCreated: Review  
}
```

```
subscription NewReviewCreated {  
    reviewCreated {  
        rating  
        commentary  
    }  
}
```

Las suscripciones de GraphQL generalmente están respaldadas por un sistema de publicación/suscripción independiente para que los mensajes sobre datos actualizados puedan publicarse según sea necesario durante el tiempo de ejecución y luego puedan ser consumidos por las funciones de resolución para los campos de suscripción en la API.

Al igual que con las operaciones de consulta y mutación, GraphQL no especifica qué protocolo de transporte utilizar, por lo que es el servidor quien debe decidirlo. En la práctica, a menudo verá que se implementan con WebSockets o eventos enviados por el servidor. Los clientes que desean enviar operaciones de suscripción también deberán admitir el protocolo elegido. Existen especificaciones mantenidas por la comunidad para implementar suscripciones GraphQL con WebSockets y eventos enviados por el servidor.

Las operaciones de suscripción son una característica poderosa de GraphQL, pero son más complicadas de implementar que las consultas o mutaciones porque se deben mantener el documento, las variables y otros contextos de GraphQL durante la vida útil de la suscripción. Estos requisitos pueden ser un desafío al escalar servidores GraphQL horizontalmente porque cada cliente suscrito debe estar vinculado a una instancia específica del servidor.

En la práctica, una API GraphQL que admita suscripciones requerirá una arquitectura más complicada que una que solo exponga campos de consulta y mutación. La forma en que se diseñe esta arquitectura dependerá de la implementación específica de GraphQL, el sistema de publicación/suscripción que admite las suscripciones y el protocolo de transporte elegido, así como otros requisitos relacionados con la disponibilidad, la escalabilidad y la seguridad de los datos en tiempo real.

Las operaciones de suscripción son adecuadas para datos que cambian con frecuencia y de forma incremental, y para clientes que necesitan recibir esas actualizaciones incrementales lo más cerca posible del tiempo real para brindar la experiencia de usuario esperada. Para datos con actualizaciones menos frecuentes, el sondeo periódico, las notificaciones push móviles o la recuperación de consultas basadas en la interacción del usuario pueden ser las mejores soluciones para mantener actualizada la interfaz de usuario de un cliente.

Validaciones

Una solicitud debe ser sintácticamente correcta para ejecutarse, pero también debe ser válida cuando se la compara con el esquema de la API.

En la práctica, cuando una operación GraphQL llega al servidor, primero se analiza el documento y luego se valida mediante el sistema de tipos. Esto permite que los servidores y los clientes informen de manera eficaz a los desarrolladores cuando se ha creado una consulta no válida, sin depender de comprobaciones en tiempo de ejecución. Una vez que se valida la operación, se puede ejecutar en el servidor y se enviará una respuesta al cliente.

La especificación GraphQL describe las condiciones detalladas que se deben cumplir para que una solicitud se considere válida.

Dentro de los errores comunes aparecen

- Solicitar campos inexistentes.
- Omisión de las selecciones de campos.
- Fragmentos faltantes para campos que generan tipos abstractos.
- Propagación de fragmentos cíclicos, un fragmento se hace referencia a sí mismo.

Ejecución de las consultas

Una vez que se valida un documento analizado, el servidor GraphQL ejecutará la solicitud de un cliente y el resultado devuelto reflejará la forma de la consulta solicitada.

Resolutores de campos

En cada campo de una consulta GraphQL se tiene que ejecutar una función o método del tipo superior que devuelve el siguiente tipo. Cada campo de cada tipo está respaldado por una función de resolución escrita por el desarrollador del servidor GraphQL. Cuando se ejecuta un campo, se llama a la función de resolución correspondiente para generar el siguiente valor.

Si un campo produce un valor escalar, como una cadena o un número, la ejecución se completa. Sin embargo, si un campo produce un valor de objeto, la consulta contendrá otra selección de campos que

se aplican a ese objeto. Esto continúa hasta que se alcanzan los valores de hoja. Las consultas GraphQL siempre terminan en tipos escalares o de enumeración.

Resolutores de los campos raíz

En el nivel superior de cada servidor GraphQL hay un tipo de objeto que representa los posibles puntos de entrada a la API GraphQL. A menudo se lo denomina “query tipo de operación raíz” o el Query tipo. Si la API también admite mutaciones para escribir datos y suscripciones para obtener datos en tiempo real, entonces tendrá Mutation tipos Subscription que expongan campos para realizar este tipo de operaciones también.

En la implementación de referencia, una función de resolución recibe cuatro argumentos

- **obj**. El objeto anterior (para un campo del tipo raíz Query, este argumento normalmente no se utiliza).
- **args**. Los argumentos proporcionados al campo en la operación GraphQL.
- **context**. Un valor proporcionado a cada solucionador que puede contener información contextual importante, como el usuario que ha iniciado sesión actualmente o el acceso a una base de datos.
- **info**. Por lo general, solo se usa en casos de uso avanzados, este es un campo de retención de valor. Contiene información específica relevante para la operación actual, así como los detalles del esquema.

```
$root_files_Resolver = [
    'human' => static fn ($obj, $args, $context, $info) =>
        ["name" => "my name {$args['id']}"], "height" => 1.72],
];
```

Resolutores por defecto

Todo campo tiene que tener su resolutor, pero en aquellos casos en donde la resolución es trivial, no hace falta que se proporcione en la mayoría de las librerías. En los casos en donde el valor a devolver es el nombre de una propiedad del objeto (**\$obj**) con el mismo nombre (**\$obj->nombre**), el propio sistema es capaz de hacerlo. Además, la implementación sobre php, también proporciona un resolutor por defecto para los casos en lo que el objeto es un Array asociativo y la clave es igual al nombre a resolver(**\$obj['nombre']**).

Producción del resultado

A medida que se resuelve cada campo, el valor resultante se coloca en un diccionario (array asociativo) clave-valor con el nombre del campo (o alias) como clave y el valor resuelto como valor. Esto continúa desde los campos hoja inferiores de la consulta hasta el campo original en el tipo de consulta raíz. En conjunto, estos producen una estructura jerárquica que refleja la consulta original que luego se puede enviar (normalmente como JSON) al cliente que la solicitó.

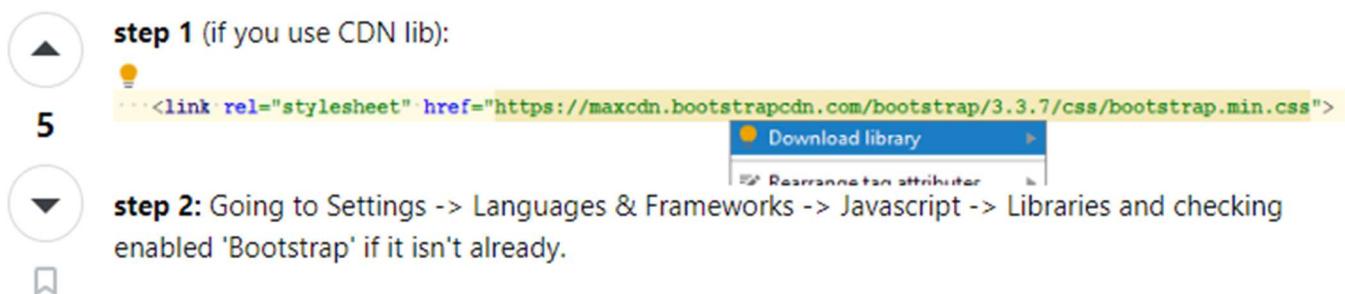
Capítulo V. Bootstrap

Bootstrap es un Framework multiplataforma o conjunto de herramientas de código abierto para diseño de sitios y aplicaciones web. El desarrollo de este capítulo está basado en la versión 5.3.3 de Bootstrap, última en el momento de desarrollarlo.

<https://getbootstrap.com/docs/5.3/getting-started/introduction/>

Configurar PhpStorm para Bootstrap

Una vez creado el enlace en cualquier página, con el botón derecho sobre la url, aparecerá un menú desde el que podemos descargar la librería para que el Ide la reconozca. Una vez descargada, lo general es que se active automáticamente, pero lo comprobaremos siguiendo los pasos del punto 2 de la imagen siguiente.

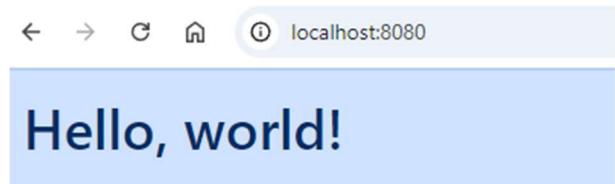


Configurar Bootstrap en mis páginas

La forma más fácil de usar Bootstrap en un proyecto es incluir la ruta CDN correspondiente en la parte HEAD (el fichero css) y al final de la misma (el fichero javascript, también podemos usar la opción **defer** en la carga y hacerla en el Head). Veremos cuál utilizar visitando la url <https://getbootstrap.com/docs/5.3/getting-started/introduction/>. Una vez mi página contenga tanto el acceso al css como al js, podremos hacer uso de las clases Bootstrap, tal y como vemos en la imagen siguiente.

```
<> index.html <
1  <!doctype html>
2  <html lang="en">
3  <head>
4      <meta charset="utf-8">
5      <meta name="viewport" content="width=device-width, initial-scale=1">
6      <title>Bootstrap demo</title>
7      <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet"
8          integrity="sha384-QWTKZyjpPEjISv5WaRU90FeRpok6YctnYmDr5pNlyT2bRjXh0JMhjY6hW+ALEwIH" crossorigin="anonymous">
9  </head>
10 <body>
11 <h1 class="alert alert-primary">Hello, world!</h1>
12 <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js"
13     integrity="sha384-YvpcrYf0tY3lHB60NNkmXc5s9fDVZLESAA55NDz0xhy9GkcIdsLK1eN7N6jIeHz" crossorigin="anonymous"></script>
14 </body>
15 </html>
```

Ilustración 13 - Configuración de Bootstrap con CDN



Existen más formas de instalar la librería en nuestros proyectos, sobre todo si usamos Composer con php o el gestor npm para javascript.

```
$ composer require twbs/bootstrap:5.3.3
```

```
$ npm install bootstrap@5.3.3
```

Recurriremos a estas técnicas cuando estemos desarrollando con Laravel en el capítulo siguiente.

Configuración inicial de una página

Para usar correctamente la librería, las páginas html deberán de contener algunas configuraciones imprescindibles. Se deberá hacer uso de la versión 5 del html y añadir la directiva meta que se muestra a continuación.

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

Si bien el CSS de Bootstrap se puede usar con cualquier Framework, el JavaScript de Bootstrap no es totalmente compatible con los Frameworks React, Vue y Angular, que asumen el conocimiento completo del DOM. Tanto Bootstrap como el Framework pueden intentar cambiar el mismo elemento DOM, lo que producirá errores en los menús desplegables. Es una mejor alternativa para aquellos que usan este tipo de Frameworks, usar un paquete específico desarrollado para él en lugar del JavaScript de Bootstrap. Estas son algunas de las opciones más populares. Para Angular: **ng-bootstrap**.

En principio, todos los complementos de Bootstrap se pueden habilitar y configurar solo a través de HTML con atributos de datos (**data-**). Hay que asegurarse de usar solo un conjunto de atributos de datos por cada elemento (por ejemplo, no se puede activar información sobre herramientas y un formulario modal desde el mismo botón).

Como las opciones se pueden pasar a través de atributos de datos o JavaScript, puede agregar un nombre de opción a **data-bs-**, como en **data-bs-animation="{value}"**. Deberemos fijarnos en cambiar el tipo de mayúsculas y minúsculas del nombre de la opción de "camelCase" a "kebab-case" al pasar las opciones a

través de atributos de datos. Por ejemplo, usar `data-bs-custom-class="beautifier"` en lugar de `data-bs-customClass="beautifier"`.

Disposición de la página

Puntos de ruptura

Los puntos de ruptura son los bloques constitutivos de Bootstrap. Se utilizan para controlar cómo la página se adapta al tamaño de cada uno de los dispositivos. Para implementar los puntos de ruptura se usa las consultas @media de CSS, incluyendo código de forma condicional en función del tamaño. Así es posible distinguir entre seis tamaños predefinidos

| Punto de ruptura | clase | Tamaño |
|--------------------------|---------|---------|
| Extra small | ninguno | <576px |
| Small | sm | ≥576px |
| Medium | md | ≥768px |
| Large | lg | ≥992px |
| Extra large | xl | ≥1200px |
| Extra extra large | xxl | ≥1400px |

Cada punto de ruptura se elegirá en función del tamaño del dispositivo, y permitirá amoldar dentro contenedores cuyo tamaño sea múltiplo de 12.

Contenedores

Los contenedores son los elementos más básicos que crea Bootstrap en una página. Es obligatorio su uso para utilizar el sistema de Grid por defecto de Bootstrap, por lo que todo contenido HTML estará definido dentro de uno. Este Grid está basado en flexbox con un número indeterminado de filas, estando cada fila dividida a su vez en 12 columnas. Para crear un contenedor, añadiremos una etiqueta div con el atributo `class = "container"`.

```

10 <body>
11 <div class="container">
12   <h1 class="alert alert-primary">Hello, world!</h1>
13 </div>

<--> localhost:8080

```

Podemos apreciar cómo cambia la estructura de la página, añadiendo espacio en ambos laterales. Además de la clase anterior, existen algunas otras para contenedores, en la tabla siguiente se muestra los tamaños que tendrá el contenedor en función del punto de ruptura:

| | Extra small
<576px | Small
≥576px | Medium
≥768px | Large
≥992px | X-Large
≥1200px | XX-Large
≥1400px |
|------------------------|-----------------------|-----------------|------------------|-----------------|--------------------|---------------------|
| container | 100% | 540px | 720px | 960px | 1140px | 1320px |
| container-sm | 100% | 540px | 720px | 960px | 1140px | 1320px |
| container-md | 100% | 100% | 720px | 960px | 1140px | 1320px |
| container-lg | 100% | 100% | 100% | 960px | 1140px | 1320px |
| container-xl | 100% | 100% | 100% | 100% | 1140px | 1320px |
| container-xxl | 100% | 100% | 100% | 100% | 100% | 1320px |
| container-fluid | 100% | 100% | 100% | 100% | 100% | 100% |

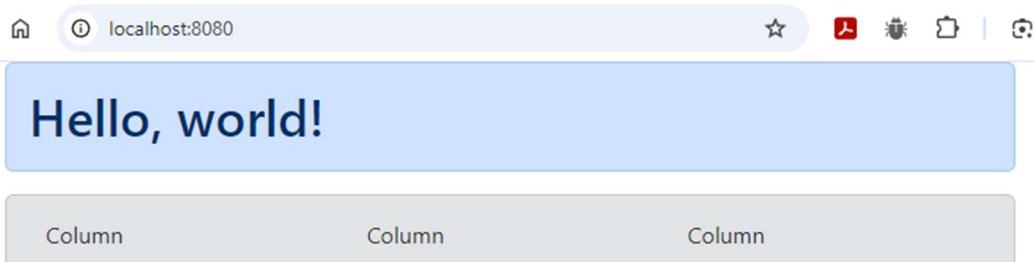
Las clases de contenedores con los puntos de ruptura son responsives hasta el ancho total definido para esa clase, así una clase **container-lg**, recogerá todo el tamaño hasta 768px y dejará espacio en el resto. Para terminar, decir que la clase **container-fluid** siempre utilizará todo el espacio disponible.

Grid

El sistema de Grid de Bootstrap utiliza una serie de contenedores, filas y columnas, para diseñar y alinear el contenido. Está construido con flexbox y es totalmente responsive.

```

11 <div class="container">
12   <div class="row">
13     <h1 class="alert alert-primary">Hello, world!</h1>
14   </div>
15   <div class="row alert alert-secondary">
16     <div class="col">
17       Column
18     </div>
19     <div class="col">
20       Column
21     </div>
22     <div class="col">
23       Column
24     </div>
25   </div>
26 </div>
```



Podemos apreciar como el diseño es por filas (ver el atributo **row** de dos div) y que, a su vez, cada fila está compuesta por columnas (atributo **col** de tres div). De momento las clases **alert-*** podemos ignorarlas, se añaden solo por mayor claridad.

Pero, ¿cómo funciona realmente el diseño Grid de Bootstrap?:

- El Grid soporta los seis puntos de ruptura definidos al comienzo del capítulo. Así, para utilizarlos los añadiremos a continuación de **col**. Por ejemplo, **col-sm**. Cuando escribimos un punto de ruptura de este tipo, significa que se aplicará al que hemos escrito y a todos los siguientes mayores, en este ejemplo también a **-md**, **-lg**, **-xl** y **-xxl**.
- Las clases container, centrarán vertical y horizontalmente el contenido mediante Padding.
- Las columnas dentro de una fila se centrarán del mismo modo.
- Se ha diseñado un Grid con doce columnas, podemos determinar el ancho de una de ellas añadiendo el número que queremos con un guion (**col-sm-6**). En este caso la columna ocupará seis columnas del Grid, el resto se repartirá entre las existentes.

```
<div class="row alert alert-secondary">
  <div class="col-6">
    Column
  </div>
  <div class="col">
    Column
  </div>
  <div class="col">
    Column
  </div>
</div>
```

Vemos como la primera columna ocupa en este caso seis celdas, la mitad de todo el tamaño y el resto (las otras seis celdas) de dividen de forma adecuada entre las dos columnas restantes.

Hello, world!

Column

Column

Column

- La gestión de flexbox, permite establecer el tamaño a una o varias columnas, no necesariamente todas, y el tamaño del resto se establecerá de forma automática.
- Se puede usar el modificar **-auto**, para hacer que la columna se adapte a su contenido, pudiendo cambiar de tamaño durante la ejecución, ya que se modificará de forma dinámica.

```
<div class="row alert alert-secondary">
  <div class="col">
    Column
  </div>
  <div class="col-auto">
    Column muy larga para hacer que cambie de tamaño
  </div>
  <div class="col">
    Column
  </div>
</div>
```

Hello, world!

Column Column muy larga para hacer que cambie de tamaño Column

Hello, world!

Column Column más corta Column

- Cuando una columna no se puede acomodar horizontalmente en el tamaño, pasa automáticamente a la siguiente fila.

Hello, world!

Column Column esta ya es mucho más larga que antes y hace saltar de fila a la siguiente
Column

- Se puede anidar el contenido añadiendo una nueva fila o más dentro de la columna.

```
<div class="row alert alert-secondary">
  <div class="col">
    Column 0
  </div>
  <div class="col-auto">
    <div class="row">
      <div class="col">
        Column 1 - 1
      </div>
      <div class="col">
        Column 2 - 1
      </div>
    </div>
  </div>
</div>
```

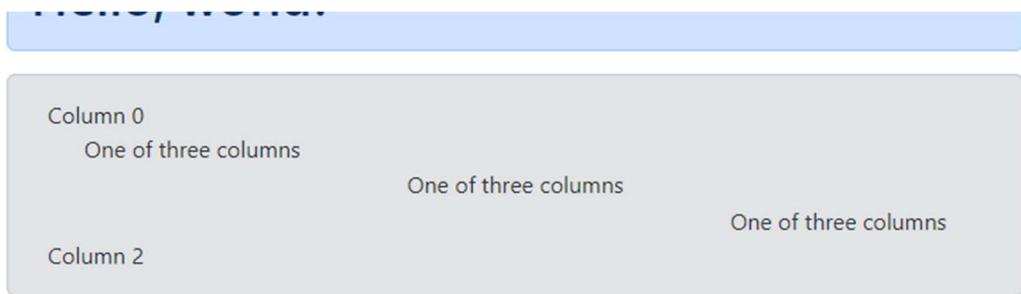
<div class="row">
 <div class="col">
 Column 3 - 2
 </div>
</div>
<div class="col">
 Column 2
</div>

Hello, world!

Column 0 Column 1 - 1 Column 2 - 1
Column 3 - 2

ALINEAMIENTO DE LAS COLUMNAS

Para alinear las columnas dentro de la fila, podemos usar cualquiera de las siguientes clases: **align-items-start**, para alinearlo a la arriba, **align-items-center** para alinearlo en el centro, y **align-items-end** para alinearlas en la parte inferior en la definición de la fila (row). Si lo que queremos es que se alineen las celdas de forma individual en vez de todas, usaremos **align-self-start**, **align-self-center** o **align-self-end** en la definición de la columna (**col**).



Las directivas anteriores alinean verticalmente las celdas, pero en caso de una alineación horizontal de los contenidos de cada celda podremos usar las siguientes clases: **justify-content-start**, **justify-content-center**, **justify-content-end**, **justify-content-around**, **justify-content-between** o **justify-content-evenly**.

DESPLAZAMIENTO DE COLUMNAS

Se pueden desplazar las columnas de la cuadrícula de dos maneras: mediante clases de cuadrícula **offset** responsivas y las utilidades de margen. Las clases de cuadrícula tienen un tamaño que coincide con las columnas, mientras que los márgenes son más útiles para diseños rápidos en los que el ancho del desplazamiento es variable.

```
<div class="row alert alert-secondary">
  <div class="container text-center">
    <div class="row">
      <div class="col-md-4".col-md-4</div>
      <div class="col-md-4 offset-md-4".col-md-4 .offset-md-4</div>
    </div>
    <div class="row">
      <div class="col-md-3 offset-md-3".col-md-3 .offset-md-3</div>
      <div class="col-md-3 offset-md-3".col-md-3 .offset-md-3</div>
    </div>
    <div class="row">
      <div class="col-md-6 offset-md-3".col-md-6 .offset-md-3</div>
    </div>
  </div>
</div>
```

TICHO, WOTCH:



Podemos mover las columnas a la derecha mediante las clases **offset-md-***. Estas clases aumentan el margen izquierdo de una columna en * columnas. Por ejemplo, **offset-md-4** mueve **col-md-4** a cuatro columnas. Con el cambio a flexbox se pueden usar las utilidades de margen como **me-auto** para forzar que las columnas hermanas se alejen entre sí. Las clases **col-*** también se pueden usar fuera de una fila para dar a un elemento un ancho específico.

Hello, world!

```

<div class="container">
  <div class="row">
    <h1 class="alert alert-primary">Hello, world!</h1>
  </div>
  <div class="row alert alert-secondary">
    <div class="container text-center">
      <div class="row">
        <div class="col-md-4">.col-md-4</div>
        <div class="col-md-4 ms-auto">.col-md-4 .ms-auto</div>
      </div>
      <div class="row">
        <div class="col-md-3 ms-md-auto">.col-md-3 .ms-md-auto</div>
        <div class="col-md-3 ms-md-auto">.col-md-3 .ms-md-auto</div>
      </div>
      <div class="row">
        <div class="col-auto me-auto">.col-auto .me-auto</div>
        <div class="col-auto">.col-auto</div>
      </div>
    </div>
  </div>
</div>

```

ESPACIADO (GUTTERS)

Los gutters son el relleno personalizado entre las columnas, se utiliza para espaciar y alinear el contenido en el sistema de Grid de Bootstrap, se crean por el Padding horizontal y se pueden ajustar de forma responsive. Se consigue ajustar tanto los rellenos horizontales como los verticales. Se pueden usar las clases **gx-{pto ruputra}-{0..5}** para controlar el ajuste horizontal o los espacios verticales con las clases **gy-{pto ruputra}-{0..5}**. Si necesitamos controlar ambos espaciados a la vez, se utilizarán las clases **g--{pto ruputra}-{0..5}* (En todos los casos el punto de ruptura es optativo)=**.

ORDEN DE PROFUNDIDAD

Varios componentes de Bootstrap utilizan la propiedad **z-index**. Esta propiedad CSS ayuda a controlar el diseño al proporcionar un tercer eje para organizar el contenido. Utilizamos una escala de índice z predeterminada en Bootstrap que se ha diseñado para superponer correctamente la navegación, la información sobre herramientas, las ventanas emergentes, los modales y más.

Los valores más altos comienzan en un número arbitrario, alto y lo suficientemente específico como para evitar conflictos. Necesitamos un conjunto estándar de estos valores en todos nuestros componentes (información sobre herramientas, ventanas emergentes, barras de navegación, menús desplegables, modales) para que podamos ser razonablemente coherentes en los comportamientos. No hay razón para cambiar los valores preestablecidos.

Formularios

Las clases dedicadas a los formularios permiten dotar de un estilo visual coherente a lo largo de la aplicación en la petición de datos.

| Descripción | Clases | Ejemplo |
|--|------------------------------------|---|
| Dar formato a una etiqueta label asociada a un control | form-label | <label for="exampleFormControlInput1" class="form-label">Email address</label> |
| Dar formato a un campo de texto, o textarea | form-control | <input type="email" class="form-control" id="exampleFormControlInput1" placeholder="name@example.com"> |
| Cambiar el tamaño del texto de un control | form-control-lg
form-control-sm | <input class="form-control form-control-lg" type="text" placeholder=".form-control-lg" aria-label=".form-control-lg example"> |
| Mensaje del campo de texto aclarativo, situado debajo de él. | form-text | <input type="password" id="inputPassword5" class="form-control" aria-describedby="passwordHelpBlock"> <div id="passwordHelpBlock" class="form-text"> Your password must be 8-20 characters long, contain letters and numbers, and must not contain spaces, special characters, or emoji. </div> |
| Desactivar un campo con el atributo disabled | | <input class="form-control" type="text" placeholder="Disabled input" aria-label="Disabled input example" disabled> |
| Campo de solo lectura con el atributo readonly | | <input class="form-control" type="text" value="Readonly input here..." aria-label="readonly input example" readonly> |
| Selector de color, dentro de un campo el atributo value junto con la clase | form-control-color | <input type="color" class="form-control form-control-color" id="exampleColorInput" value="#563d7c" title="Choose your color"> |
| Dar estilo a la etiqueta select | form-select | <select class="form-select" multiple aria-label="Multiple select example"> |

TIEMPO, WORDS:

Email address

.form-control-lg

Your password must be 8-20 characters long, contain letters and numbers, and must not contain spaces, special characters, or emoji.

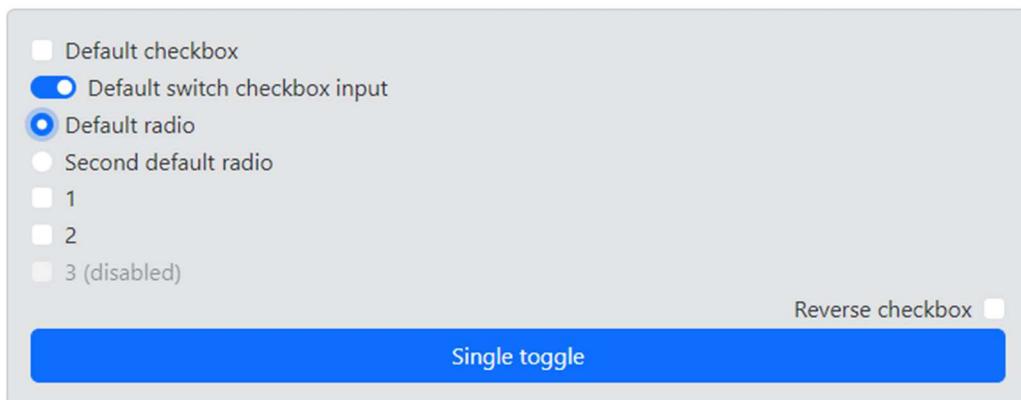
Disabled input

Readonly input here...

Checkboxes y options

| Descripción | Clases | Ejemplo |
|---|--|--|
| Dar estilo a una checkbox | form-check
form-check-input
form-check-label | <pre><div class="form-check"> <input class="form-check-input" type="checkbox" value="" id="flexCheckDefault"> <label class="form-check-label" for="flexCheckDefault"> Default checkbox </label> </div></pre> |
| Dar estilo a una option, usar el atributo type="radio" | | <pre><input class="form-check-input" type="radio" name="flexRadioDefault" id="flexRadioDefault1"></pre> |
| Crear un botón de checkbox. Usar la clase form-switch y el role switch. | form-switch
role="switch" | <pre><div class="form-check form-switch"> <input class="form-check-input" type="checkbox" role="switch" id="flexSwitchCheckDefault"></pre> |
| Agrupar en estilo horizontal | form-check-inline
form-check | <pre><div class="form-check form-check-inline"></pre> |
| Situar los elementos en la parte derecha del área | form-check-reverse | <pre><div class="form-check form-check-reverse"></pre> |
| Crear estilo de botón tanto en checkboxes como en radio, ver las clases de botones más adelante | btn-check
btn
btn-check-* | <pre><input type="checkbox" class="btn-check" id="btn-check" autocomplete="off"> <label class="btn btn-primary" for="btn-check">Single toggle</label></pre> |
| | | |

TIPO, WORDS:



Rangos

| Descripción | Clases | Ejemplo |
|--------------------------------|------------|---|
| Para el tipo range de un input | form-range | <input type="range" class="form-range" id="customRange1"> |

Grupos de controles

Podemos crear grupos de controles para dar un significado semántico a la entrada para que sean tratados todos los elementos que forman el grupo de manera similar. Para la creación de un grupo se utilizará la clase **input-group** y en el div correspondiente, a continuación, se formará el grupo con una etiqueta **span** en el que se utilice la clase **input-group-text** y otro elemento de formulario. Si queremos evitar que el grupo se deshaga en varias líneas por el tamaño, tendremos que establecer la clase **flexnowrap** en el contenedor.

Uso básico:

- La gestión del tamaño del grupo, se puede realizar añadiendo los puntos de ruptura a la clase **input-group**.
- Para crear un grupo con checkboxes u options en vez de texto, es recomendable añadir la clase **mt-0** al elemento cuando no hay texto a continuación del campo input.
- Es posible crear grupos con varios controles en la misma línea, siguiendo la lógica del código vista hasta ahora.
- Podemos jugar con la posición de los elementos de grupo para establecerlos dónde nos convenga con sólo ordenarlos según la presentación de arriba a bajo en código HTML.
- Incluso es factible crear botones con listas desplegables con la clase **dropdown-toggle** en la etiqueta button y a continuación una definición de lista desordenada (ul) en la que se establecerá la clase **dropdown-menu** (podemos añadir además **dropdown-menu-end** si deseamos que se sitúe el menú a la derecha en vez de a la izquierda).
- Podemos crear grupos con listas de selección usando la etiqueta select y options en vez del button más ul, en este caso utilizaremos una label como texto de presentación.

- También se puede agrupar un tipo file, usando como campo anterior al mismo una label o un button.

The screenshot shows a horizontal input group with the following components from left to right:

- A text input with placeholder "Username".
- A text input with placeholder "Username".
- A small text input with placeholder "Small".
- A checkbox input.
- A text input with placeholder "First and last name".
- A text input with placeholder "\$ 0.00".
- A button labeled "Button".
- A button labeled "Button".
- A dropdown menu labeled "Dropdown ▾".
- A button labeled "Options".
- A button labeled "Choose...".
- A file input with placeholder "Seleccionar archivo" and a status message "Ningún archivo seleccionado".
- An "Upload" button.
- Three additional buttons labeled "Button", "Seleccionar archivo", and "Ningún archivo seleccionado".

```
<div class="input-group">
  <span class="input-group-text" id="basic-addon1">@</span>
  <input type="text" class="form-control" placeholder="Username"
    aria-label="Username" aria-describedby="basic-addon1">
</div>
<div class="input-group flexnowrap">
  <span class="input-group-text" id="addon-wrapping">@</span>
  <input type="text" class="form-control" placeholder="Username"
    aria-label="Username" aria-describedby="addon-wrapping">
</div>
<div class="input-group input-group-sm">
  <span class="input-group-text" id="inputGroup-sizing-sm">Small</span>
  <input type="text" class="form-control" aria-label="Sizing example input"
    aria-describedby="inputGroup-sizing-sm">
</div>
<div class="input-group">
  <div class="input-group-text">
    <input class="form-check-input mt-0" type="checkbox" value="" 
      aria-label="Checkbox for following text input">
  </div>
  <input type="text" class="form-control">
</div>
```

```
    aria-label="Text input with checkbox">
</div>
<div class="input-group">
  <span class="input-group-text">First and last name</span>
  <input type="text" aria-label="First name" class="form-control">
  <input type="text" aria-label="Last name" class="form-control">
</div>

<div class="input-group">
  <span class="input-group-text">$</span>
  <span class="input-group-text">0.00</span>
  <input type="text" class="form-control"

    aria-label="Dollar amount (with dot and two decimal places)">
</div>

<div class="input-group">
  <button class="btn btn-outline-secondary" type="button">Button</button>
  <input type="text" class="form-control" placeholder="" 

    aria-label="Example text with two button addons">
  <button class="btn btn-outline-secondary" type="button">Button</button>
</div>

<div class="input-group">
  <button class="btn btn-outline-secondary dropdown-toggle" type="button"

    data-bs-toggle="dropdown" aria-expanded="false">Dropdown</button>
  <ul class="dropdown-menu">
    <li><a class="dropdown-item" href="#">Action</a></li>
    <li><a class="dropdown-item" href="#">Another action</a></li>
    <li><a class="dropdown-item" href="#">Something else here</a></li>
    <li><hr class="dropdown-divider"></li>
    <li><a class="dropdown-item" href="#">Separated link</a></li>
  </ul>
  <input type="text" class="form-control"

    aria-label="Text input with dropdown button">
</div>

<div class="input-group">
  <label class="input-group-text" for="inputGroupSelect01">Options</label>
  <select class="form-select" id="inputGroupSelect01">
    <option selected>Choose...</option>
    <option value="1">One</option>
```

```

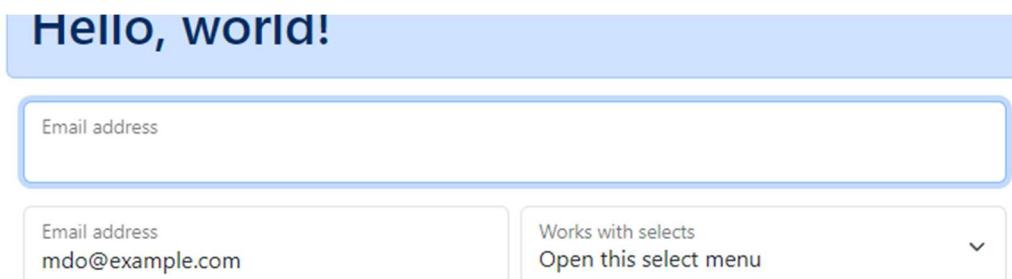
<option value="2">Two</option>
<option value="3">Three</option>
</select>
</div>

<div class="input-group">
  <input type="file" class="form-control" id="inputGroupFile02">
  <label class="input-group-text" for="inputGroupFile02">Upload</label>
</div>
<div class="input-group">
  <button class="btn btn-outline-secondary" type="button"
    id="inputGroupFileAddon03">Button</button>
  <input type="file" class="form-control" id="inputGroupFile03"
    aria-describedby="inputGroupFileAddon03" aria-label="Upload">
</div>

```

Etiquetas en las cabeceras

Este tipo de ayuda, permite situar un texto en la parte superior del campo de entrada mostrando una ayuda al mismo. Para activar esta opción crearemos los campos de la forma tradicional, agrupándolos dentro de un div en el que se definirá la clase **form-floating**. Se puede usar además de con etiquetas input, textarea, select, grupos; opciones disabled, validaciones y readonly. La única precaución a tener en cuenta, es que el grupo completo debería estar dentro de div que defina la clase **col** para asegurarnos el buen funcionamiento cuando se quieran situar varios en la misma fila.



```

<div class="form-floating mb-3">
  <input type="email" class="form-control" id="floatingInput"
    placeholder="name@example.com">
  <label for="floatingInput">Email address</label>
</div>

<div class="row g-2">
  <div class="col-md">
    <div class="form-floating">
      <input type="email" class="form-control" id="floatingInputGrid"

```

```

placeholder="name@example.com" value="mdo@example.com">
    <label for="floatingInputGrid">Email address</label>
    </div>
</div>
<div class="col-md">
    <div class="form-floating">
        <select class="form-select" id="floatingSelectGrid">
            <option selected>Open this select menu</option>
            <option value="1">One</option>
            <option value="2">Two</option>
            <option value="3">Three</option>
        </select>
        <label for="floatingSelectGrid">Works with selects</label>
    </div>
</div>
</div>

```

Componentes Bootstrap

<https://getbootstrap.com/docs/5.3/components/>

Bootstrap añade gran cantidad de componentes predefinidos a su repertorio, es una de las características del Framework que lo distinguen de algunos otros. No es necesario detallar todos y cada uno de ellos, en la documentación está muy bien explicados, con todas las opciones y ejemplos. Mostramos a continuación un listado de los más usados, recomiendo que se estudien las alertas, botones, menús como mínimo.

| Componente | url de la documentación |
|-----------------------|--|
| Acordeón | https://getbootstrap.com/docs/5.3/components/accordion/ |
| Alertas | https://getbootstrap.com/docs/5.3/components/alerts/ |
| Botones | https://getbootstrap.com/docs/5.3/components/buttons/
https://getbootstrap.com/docs/5.3/components/close-button/ |
| Grupos de botones | https://getbootstrap.com/docs/5.3/components/button-group/ |
| Elementos contraídos | https://getbootstrap.com/docs/5.3/components/collapse/ |
| Botones de lista | https://getbootstrap.com/docs/5.3/components/dropdowns/ |
| Grupos de lista | https://getbootstrap.com/docs/5.3/components/list-group/ |
| Menús | https://getbootstrap.com/docs/5.3/components/navbar/
https://getbootstrap.com/docs/5.3/components/navs-tabs/ |
| Paginación | https://getbootstrap.com/docs/5.3/components/pagination/ |
| Barras de progreso | https://getbootstrap.com/docs/5.3/components/progress/ |
| Indicador de progreso | https://getbootstrap.com/docs/5.3/components/spinners/ |
| Tooltips | https://getbootstrap.com/docs/5.3/components/tooltips/ |

Clases de ayuda

El Framework nos proporciona un conjunto de clases de ayuda para tareas muy comunes, vamos a verlas

| Clase | Descripción | |
|--|---|--|
| link-* | Mostrar un enlace de un color concreto pero se tendrá en cuenta los selectores :hover y :focus, algo que no lo hacen las clases text-*. | |
| icon-link | Permite añadir un ícono a un enlace. El ícono será un svg que aparecerá en el cuerpo del texto mediante la etiqueta <svg> | |
| fixed-top
sticky-top | fixed-bottom
sticky-bottom | Posicionamiento rápido de elementos independiente del scroll del contenido. |
| ratio-1x1
ratio-16x9 | ratio-4x3
ratio-21x9 | Hace que el elemento mantenga dicha proporción visual. |
| vstack | hstack | Apilamiento vertical-horizontal de los elementos siguientes. |
| text-truncate | | Trunca el contenido añadiendo puntos suspensivos cuando hay gran cantidad, es obligatorio display: inline-block o display:block |
| vr | | Dibuja una línea vertical, similar a la generada por <hr> |
| visually-hidden
visually-hidden-focusable | | Oculta el campo, pero lo mantiene activo para las tecnologías de accesibilidad, tales como lectores. La segunda clase hace que el elemento esté oculto hasta que se le pase el foco, momento en el que se muestra. |
| text-bg-{def_color} | | Establece el text-color y bg-color a la vez. Cambia el color de fondo y el de primer plano a la vez. |

Utilidades

Las siguientes clases se usan muy a menudo y debemos conocerlas.

| Clase | Descripción |
|---|---|
| bg-primary bg-primary-subtle
bg-secondary bg-* | Establece el color de fondo |
| bg-primary-gradient
bg-*-gradient | Partiendo de las clases anteriores añadiendo -gradient, genera un gradiente en el color de fondo en vez de uno sólido. |
| style="--bs-bg-opacity: .5;" | Junto con una clase de color de fondo, redefine la opacidad. |
| border border-top border-end
border-bottom border-start | Añade bordes al elemento, todos o uno en concreto en función de su posición. |
| Border-*-0 | En vez de añadir, en este caso lo elimina, es necesario que tenga border antes. |
| border-primary
border-secondary
border-primary-subtle ... | Establece el color del borde |
| rounded rounded-top
rounded-end rounded-circle
rounded-pill rounded-* | Redondea las esquinas de los bordes |
| rounded-[0..5]-* | Define el tamaño del redondeo en los bordes, desde 0 hasta el máximo de 5 |
| text-primary text-primary-emphasis text-success ... | Establece el color de primer plano de la fuente. |
| d-{punto_ruptura}-{valor} | Define el valor de la propiedad display de CSS. El punto de ruptura es obligatorio para sm, md, lg, xl y xxl. El valor es cualquiera de la propiedad: none, inline, inline-block, grid, inline-grid, table, table-cell, table-row, flex, inline-flex, |
| d-print-{valor} | Clases para una mejor impresión en un documento (informes, impresiones en pdf, etc.). Valor contendrá lo explicado en la línea anterior. |
| d-{punto_ruptura}-flex-{valor} | Permite gestionar el grid flex de forma más precisa: flex-row, flex-row-reverse, flex-column, flex-column-reverse |
| justify-content-{pto_r.}-{valor} | En un contenedor con d-flex establece la justificación de las celdas, valor puede ser: start, end, center, between, around o evenly. |
| align-content-{pto_r.}-{valor} | Alinea todos los elementos según valor. Este podrá ser: start, end, center, between, around o stretch. |
| align-items-{pto_rup.}-{valor} | En un contenedor con d-flex marca la alineación de sus elementos, valor puede ser: start, end, center, baseline, stretch, start |
| align-self-{pto_rup.}-{valor} | Permite la lineación de un elemento tomándose a él como referencia. Valor puede ser: start, end, center, baseline o stretch. |
| flex-{pto_ruptura}-fill | Obliga a ocupar todo el espacio del padre. |
| flex-grow-1 | Utiliza todo el espacio que quede libre en horizontal. |

| | |
|---|---|
| flex-shrink-1 | Utiliza todo el espacio que quede libre en vertical. |
| me-auto | Añade margen al final del elemento para utilizar el mayor espacio posible. |
| ms-auto | Añade margen al comienzo del elemento para utilizar el mayor espacio posible. |
| flexnowrap | Evita que sus elementos salten de línea si no caben. |
| flex-wrap-reverse | Hace que los elementos salten de línea si no caben en orden inverso. |
| flex-wrap | Valor por defecto, hace que salten los elementos a la siguiente línea si no entran. |
| order-{punto ruptura}-n | El valor de n será un numérico de 0 a 5 que se utilizará para mostrar los elementos de menor a mayor siempre independientemente de la definición de arriba a abajo. N también podrá tomar los valores first y last. |
| float-{punto_ruptura}-{valor} | Establece un elemento flotante según valor: start, end o none. |
| user-select-{valor} | Cambia la forma en la que se selecciona el texto. All: todo a la vez, auto: valor por defecto, none: no se selecciona. |
| link-opacity-{valor}
link-opacity-{valor}-hover
link-underline-opacity-{valor} | Transparencia del enlace al modtrar, al estar encima de él, o del subrayado. Valor será uno de: 10, 25,50,75 o 100, medidos en porcentajes. |
| link-{def_color} | Establece el color del enlace o del subrayado, def_color será uno de los valores de color reconocido: primary, etc... |
| link-underline-{def_color} | |
| link-offset-{valor} | Posición del subrayado en un enlace. Valor será 1, 2 o 3. |
| opacity-{valor} | Transparencia del elemento. Valor será uno de: 10, 25,50,75 o 100. |
| w-{valor} | Ancho relativo del elemento en porcentaje, donde valor será: auto, 25, 50, 75, 100. |
| h-{valor} | Alto relativo del elemento en porcentaje, donde valor será: auto, 25, 50, 75, 100. |
| mw-100 mh-100 | Establece el ancho y el alto relativo al 100 por 100. |

Espaciado

Las utilidades de espaciado se aplican a todos los puntos de ruptura, desde xs hasta xxl, no tienen ninguna abreviatura. Esto se debe a que esas clases se aplican desde min-width: 0 y superior y, por lo tanto, no están enlazadas por una consulta de medios. Sin embargo, los puntos de interrupción restantes incluyen una abreviatura de punto de interrupción. Las clases se nombran con el formato **{propiedad}{lado}-{tamaño}** para xs y **{propiedad}{lado}-{punto_ruptura}-{tamaño}** para sm, md, lg, xl y xxl.

Donde la **propiedad** es una de las siguientes:

- m - para las clases que establecen el margen.
- p - para las clases que establecen el relleno.

Donde **lado** es uno de los siguientes:

- t - para las clases que establecen margin-top o padding-top.

- b - para clases que establecen margin-bottom o padding-bottom.
- s - (inicio) para las clases que establecen margin-left o padding-left en LTR, margin-right o padding-right en RTL.
- e - (fin) para las clases que establecen margin-right o padding-right en LTR, margin-left o padding-left en RTL.
- x - para las clases que establecen tanto *-left como *-right.
- y - para las clases que establecen tanto *-top como *-bottom.
- En blanco: para clases que establecen un margen o relleno en los 4 lados del elemento.

Donde el **tamaño** es uno de los siguientes:

- 0 - para las clases que eliminan el margen o el relleno estableciéndolo en 0.
- 1 - (de forma predeterminada) para las clases que establecen el margen o el relleno en 25%.
- 2 - (de forma predeterminada) para las clases que establecen el margen o el relleno en 50%.
- 3 – espaciado por defecto.
- 4 - (de forma predeterminada) para las clases que establecen el margen o el relleno en 150%.
- 5 - (de forma predeterminada) para las clases que establecen el margen o el relleno en 300%.
- auto: para las clases que establecen el margen en auto.

Se puede establecer también el espaciado entre celdas del Grid y en las filas del mismo. Para ello usaremos **g-col-{0..5}** o **row-gap-{0..5}** en el contenedor padre.

Utilidades de texto

| Clase | Descripción |
|---|---|
| text-{pto_ruptura}-{valor} | Establece la alineación. Valor uno de: start, end, center. |
| text-wrap | Hace saltar el texto si no entra. |
| text-nowrap | Evita que el texto salte de línea. |
| text-break | Evitar que la disposición general se deshaga por un texto muy largo. |
| text-lowercase text-uppercase
text-capitalize | Transformar la visualización del texto en minúsculas, mayúsculas o capitalizadas. |
| fs-{1--6} | Tamaño del texto. |
| fw-{valor}
fst-italic fst-normal | Formato de la fuente. Valor será uno de: bold, bolder, semibold, médium, normal, light, lighter; fst-* lo establece al estilo indicado. |
| lh-{valor} | Establece el interlineado. Valor será: sm, base, lg o 1. |
| font-monospace | Establece la fuente en monoespaciada. |
| text-reset | Resetea el color. |
| text-decoration-underline
text-decoration-line-through | Subraya un texto o lo tacha. |

Alineación vertical

| Clase | Descripción |
|-------|-------------|
|-------|-------------|

| | |
|----------------------|--|
| align-{valor} | Establece la alineación vertical. Valor: baseline, top, middle, bottom, text-top y text-bottom |
|----------------------|--|

Visibilidad

| Clase | Descripción |
|--------------------------|---|
| visible invisible | Establece la visibilidad sin cambiar la disposición de los elementos, no como cuando se cambia el display a none. |

Orden de profundidad

| Clase | Descripción |
|------------------|--|
| z-{valor} | Establece el nivel de posicionamiento en el eje Z, los valores pueden ser: n1, 0, 1, 2, 3. |

Prácticas Bootstrap: enunciados.docx

Capítulo VI. Composer PHP

Uso de Composer

<https://getcomposer.org/>

Instalación en local bajo Windows

<https://getcomposer.org/download/>

INSTALACIÓN WINDOWS GLOBAL

El instalador, que requiere que ya tenga PHP instalado, descargará Composer y configurará su variable de entorno PATH para que simplemente pueda llamar a Composer desde cualquier directorio. Descargue y ejecute Composer-Setup.exe: instalará la última versión del compositor cada vez que se ejecute.

INSTALACIÓN LOCAL

Para instalar rápidamente Composer en el directorio actual, ejecute el siguiente script en su terminal. Para automatizar la instalación.

```
>php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"  
>php -r "if (hash_file('sha384', 'composer-setup.php') ===  
'dac665fdc30fdd8ec78b38b9800061b4150413ff2e3b6f88543c636f7cd84f6db9189d43a81e5503cda447da73  
c7e5b6') { echo 'Installer verified'.PHP_EOL; } else { echo 'Installer corrupt'.PHP_EOL; unlink('composer-  
setup.php'); exit(1); }"  
>php composer-setup.php  
>php -r "unlink('composer-setup.php');"
```

Este script de instalación simplemente verificará algunas configuraciones php.ini, le advertirá si están configuradas incorrectamente y luego descargará el último composer.phar en el directorio actual. Las 4 líneas anteriores, en orden realizan:

- Descarga del instalador en el directorio actual.
- Verificación del instalador SHA-384.
- Ejecución del instalador.
- Eliminación del instalador.

Lo más probable es que desee poner **composer.phar** en un directorio en su PATH, para poder llamar a Composer desde cualquier directorio (instalación global), **en Windows lo mejor es una instalación global con el ejecutable (exe)** y en sistemas Unix añadirlo a la variable de entorno PATH dentro del script de configuración de usuario. En todo caso, nos podremos situar en la raíz del proyecto y ejecutar el Composer correspondiente.

Generando y comprendiendo composer.json

Para lograr esto, debes generar un archivo **composer.json**. Puedes considerarlo como una forma de buscar datos de una lista para Composer. Este archivo contiene paquetes (dependencias) que deben descargarse.

Además, **composer.json** también verifica la compatibilidad de la versión con tu proyecto. Esto significa que si estás utilizando un paquete anterior, **composer.json** te lo informará para evitar problemas futuros.

Tienes la opción de crear y actualizar **composer.json** tú mismo pero se recomienda seguir estos pasos:

Crea un nuevo directorio para el proyecto, cambia al nuevo directorio creado.

Encuentra un paquete o biblioteca para el proyecto. El mejor lugar para lograrlo es Packagist, donde encontrarás toneladas de bibliotecas para ayudar al desarrollo de tu proyecto. Como puedes ver, hay varios paquetes de temporizadores disponibles y cada uno tiene un nombre y una pequeña descripción de lo que hace

Especifica el paquete deseado para que Composer pueda agregarlo a tu proyecto:

composer require nombre

Después de ejecutar el comando anterior, el directorio de su proyecto tendrá dos archivos nuevos: **composer.json** y **composer.lock**, y una carpeta llamada **vendor**. Este es el directorio donde **Composer** almacenará todos tus paquetes y dependencias. Procedimiento inicial en el proyecto.

```
composer.phar init
composer.phar require nombre/nombre_paquete
composer.phar install
```

Usando el Script Autoload

El proyecto está casi configurado, y lo único que queda por hacer es cargar la dependencia en tu script PHP. Y, afortunadamente, el **archivo de carga automática de Composer** te ayuda a completar este proceso más rápido.

Para usar la carga automática, se escribe la siguiente línea antes de declarar o instanciar nuevas variables en tu script php:

```
<?php
require __DIR__ . '/vendor/autoload.php'
```

Actualización de las dependencias de tu proyecto

Por último, debes saber cómo actualizar tus paquetes. Esto lo puedes hacer de dos maneras:

- **Actualización universal.** Para verificar e instalar actualizaciones para todos tus paquetes y dependencias a la vez, escribe el siguiente comando:

```
composer update
```

- **Actualización específica del paquete.** Ejecuta este comando para verificar las actualizaciones de uno o más paquetes específicos:

```
composer update vendor/package vendor2/package2
```

Recuerda reemplazar la parte de **vendor/package** con el nombre del paquete que deseas actualizar. Al ejecutar el comando de **update**, Composer también actualiza los archivos **composer.json** y **composer.lock** para que coincidan con el estado actual de las dependencias de tu proyecto.

Configurar el sistema de autocarga de clases

El proceso descrito antes es el trabajo básico que tendremos que realizar para cargar las librerías, de una manera ágil, gracias a Composer. Pero podría ser necesario para nuestras aplicaciones configurar el sistema de autocarga de clases. En la mayoría de los casos no será necesario, pero puede venirnos bien si queremos aprovecharnos del autoload de Composer para cargar de manera automática las clases de nuestro propio proyecto. Para ello se define un nuevo campo, llamado "autoload" en el archivo **composer.json**.

Ese campo nos permite indicar un namespace y las clases que se encuentran detrás de ese namespace.

```
"autoload": {  
    "psr-4": {"MiNamespace\\": "mi_carpeta/"}  
}
```

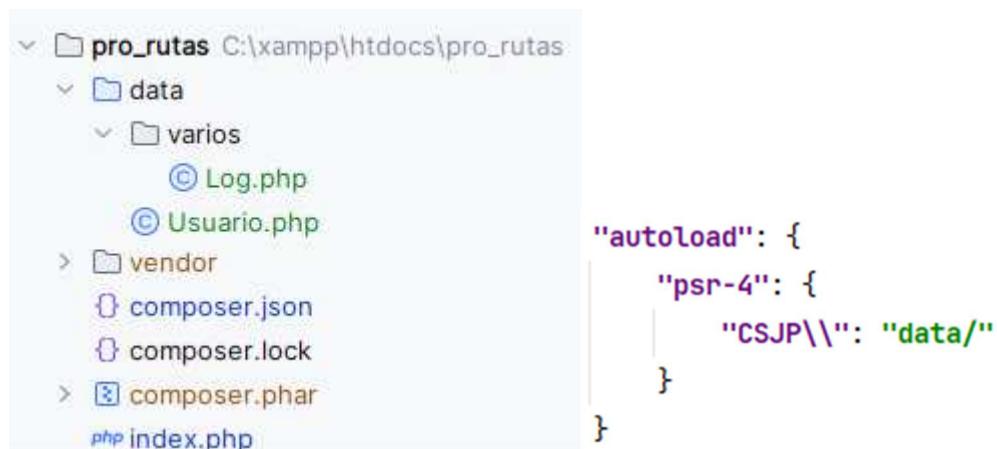
Esto indica que las clases del namespace "MiNamespace" las tiene que ir a buscar al directorio "mi_carpeta" (suponiendo que "mi_carpeta" se encuentra colgando la raíz, por lo que sería un hermano del directorio "vendor", las dos barras son obligatorias como separador).

Solo un único detalle. Una vez has configurado el sistema de Autoload de Composer, debes correr el comando "dump-autoload", para que se vuelva a generar y optimizar todo el proceso de Autoload de clases de Composer. Este paso es tan sencillo como lanzar el comando:

```
composer dumpautoload
```

Ahora que tenemos el archivo "vendor/autoload.php" regenerado, ya podremos tener disponibles las clases de este namespace para su autocarga.

Ejemplo



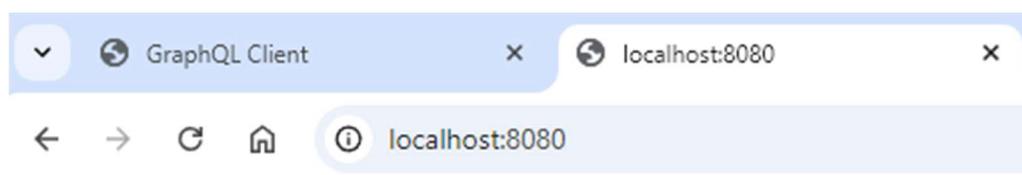
© Log.php ×

```
1 <?php  
2  
3 namespace CSJP\varios;  
4  
5 class Log {  
6     public string $val="";  
7 }
```

© Usuario.php ×

```
1 <?php  
2 declare(strict_types=1);  
3 namespace CSJP;  
4  
5 use CSJP\varios\Log;  
6  
7 class Usuario {  
8     public string $nombre="";  
9     public Log $log;  
10  
11     public function __construct(string $nombre){  
12         $this->nombre = $nombre;  
13     }  
14     public function __toString():string {  
15         return $this->nombre . " - " . $this->log->val;  
16     }  
17 }  
18 }
```

```
php index.php ×  
1 <?php  
2 declare(strict_types=1);  
3  
4 require_once __DIR__ . './vendor/autoload.php';  
5  
6 use CSJP\Usuario;  
7 use CSJP\Varios\Log;  
8  
9 $usuario = new Usuario( nombre: "Carlos");  
10 $usuario->log = new Log();  
11 $usuario->log->val = "Valores de log";  
12  
13 echo $usuario;
```



Carlos - Valores de log

Hacer el empleo anterior. En Composer Practica_01.zip

Capítulo VII. Laravel

<https://laravel.com/docs/12.x/readme>

Preparando un entorno de desarrollo

A la hora de crear aplicaciones bajo Laravel, podemos optar en función del Sistema Operativo de varias opciones para preparar el entorno, vamos a presentar las más comunes de ellas.

- **XAMPP / WAMPP.** Es un paquete que instala el servidor Apache, PHP y MariaDB, junto algún servidor más como Tomcat y Fake Sendmail, tiene muchos años a sus espaldas. Está orientado al desarrollo web de forma generalista y no exclusivo de Laravel, al ser más antiguo no realiza ninguna gestión de servidores o paquetes, más allá de un panel de control para la gestión. Es necesario configurar posteriormente tanto los servidores virtuales dentro de Apache como la librería Xdebug para depuración de PHP, además tendremos que instalar de forma global Composer y el Instalador de Laravel, así como Nodejs. Es el más veterano y el que nos ofrece un entorno más parecido a un servidor real de producción.
- **Herd.** Permite cambiar fácilmente la versión de PHP y admite Node.js junto con varias bases de datos como PostgreSQL, MongoDB, Redis, SQLite, MySQL y más. Una diferencia notable con XAMPP es que no es compatible con Apache; en su lugar, utiliza Nginx. Es el más moderno de todos, pero no permite la portabilidad que da Laragon.
- **Docker Sail.** Da la misma flexibilidad que los anteriores, pero está basado en Docker, siendo obligatorio la instalación de Docker Desktop para su uso. Probablemente es el más parecido a un entorno de desarrollo remoto. Además, es la única opción junto a XAMP para entornos Unix.
- **Laragon,** permite la flexibilidad de Herd junto a todas las opciones ya mencionadas de todos los entornos de desarrollo, pero añade la posibilidad de cambiar el directorio de instalación simplemente moviendo o renombrando y convivir con XAMPP a la vez, siempre que no se ejecuten simultáneamente.

En este capítulo nos decantamos por elegir la opción tradicional XAMPP, junto con Nodejs, Composer o por usar la opción de Herd al ser la recomendación oficial.

Inicializando Laravel

Antes de poder realizar alguna aplicación hay que realizar una serie de procedimientos comunes a todos los desarrollos. Se tiene que instalar Composer y Nodejs, así como el instalador de aplicaciones Laravel. Veamos el procedimiento completo.

La instalación de paquetes vía Composer, una vez ya configurado XAMPP, obliga a tener en el sistema un entorno node.js activo, con lo que descargaremos de la url del Nodejs en <https://nodejs.org/en/download>, el último entorno LTS (22.14.0) y lo instalaremos. Nodejs es una aplicación muy común entre programadores JavaScript, no solo para crear entornos en Backend. Instalaremos como paso final el asistente de Laravel para crear aplicaciones con el comando siguiente desde la consola:

`composer global require laravel/installer`

```
C:\Users\arcipreste>composer global require laravel/installer
Changed current directory to C:/Users/arcipreste/AppData/Roaming/Composer
./composer.json has been created
Running composer update laravel/installer
Loading composer repositories with package information
```

Este se utiliza para crear la estructura de archivos y directorios requerida. Además, antes de realizar la inicialización de la aplicación, si vamos a utilizar un servidor de base de datos que no sea SQLite, deberemos tener todos los datos de acceso, siendo obligatorio que esté arrancado el correspondiente servidor. Así en nuestro caso abriremos el panel de control de XAMPP y lanzaremos Mysql. Con todo esto ya podemos pasar a crear el esqueleto de la aplicación desde la consola:

laravel new app_01

Podemos dejar los valores por defecto o establecer algunas configuraciones básicas como son el Framework cliente que vamos a usar, el Framework para las pruebas y el servidor de base de datos en el que desarrollar.

Una vez terminada la configuración, podremos lanzar el entorno de desarrollo con el siguiente comando desde un terminal. Si todo es correcto, usando el explorador accederemos a la dirección correspondiente (<http://127.0.0.1:8000>) para ver si está funcionando.

```
composer run dev  
http://127.0.0.1:8000/
```

Si todo ha ido correctamente, en la dirección anterior nos encontraremos el logo de Laravel en el navegador. Si usamos para el desarrollo Herd y hemos creado el proyecto desde las utilidades del mismo, el acceso será diferente, a través de la url formada por el nombre del proyecto y el dominio **test**, es decir, http://app_01.test.

El patrón Modelo-Vista-Controlador (MVC)

El Modelo-Vista-Controlador (MVC) es un patrón de arquitectura de software que separa principalmente, lo que es la lógica de negocio de una aplicación, de su representación y del módulo encargado de gestionar los eventos y las comunicaciones. Para ello MVC propone la construcción de tres componentes distintos que son el modelo, la vista y el controlador; es decir: por un lado, define componentes para la representación de la información, por otro lado elementos para la interacción del usuario y por otro bloques encargados de unir unos con otros. Este patrón se basa en las ideas de reutilización de código y la separación de conceptos, características que buscan facilitar la tarea de desarrollo de aplicaciones y su posterior mantenimiento.

- **Modelo:** implementa la lógica de negocio, es decir la funcionalidad que ofrece la aplicación. Administra los datos de la aplicación.
- **Vista:** son componentes que despliegan la interfaz de usuario. No suelen tener casi lógica, y están implementados casi exclusivamente por HTML / CSS / JavaScript. Se suele usar un motor de plantillas para mejorar su creación.
- **Controlador:** son elementos que manejan la interfaz de usuario (seleccionan la vista correcta a desplegar) y trabajan con el modelo. En esta parte se realizará la mayoría de la programación, validación, etc.

Laravel implementa el patrón MVC y más allá, permitiendo por ejemplo que los controladores se definan o bien dentro de la carpeta **controllers**, mediante funciones anónimas, o componentes.

Tipos de programación

Laravel se puede adoptar para diversos tipos de aplicación, pero destacan dos modelos de programación: desarrollo completo y servidor API.

Desarrollo completo

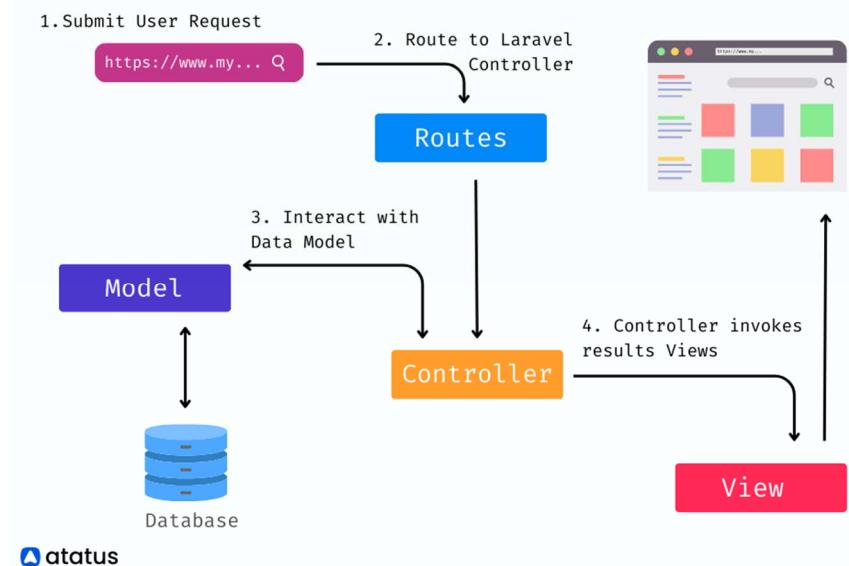


Ilustración 15 – Arquitectura Laravel MLP

recibirá una página HTML nueva.

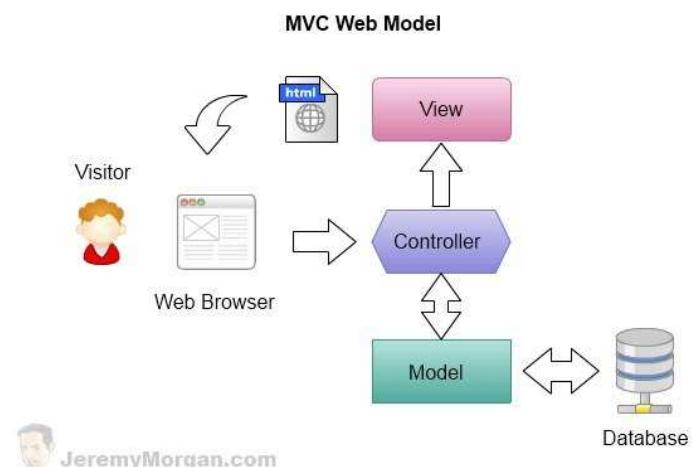
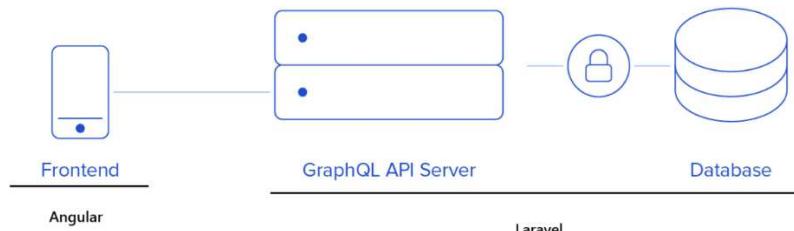


Ilustración 14 – Patrón MVC

Con este paradigma, el programador Laravel crea la aplicación de forma completa, apoyándose en plantillas de desarrollo para la vista y JavaScript embebido. Con este enfoque se adopta un desarrollo de ventana múltiple, en la que se creará un nuevo código HTML con cada petición que el cliente haga al servidor. Se dará uso completo al patrón MVC dentro del proyecto. Como vemos en la imagen, el usuario lanza una petición, que tras pasar por todo el modelo MVC

Servidor API

En el lado opuesto, está el modelo de ventana única. Bajo Laravel se desarrollará exclusivamente la parte CV del patrón, creando un API de acceso a los datos (REST o GraphQL). Por otro lado, se realizará el desarrollo del Frontend bajo Angular en el que se implementará la parte CV del paradigma.



Estructura de directorios de la aplicación Laravel

| Project | |
|-----------------|--------------|
| pr01 | C:\Herd\pr01 |
| app | |
| Http | |
| Controllers | |
| Models | |
| Providers | |
| bootstrap | |
| cache | |
| app.php | |
| providers.php | |
| config | |
| database | |
| factories | |
| migrations | |
| seeders | |
| database.sqlite | |
| public | |
| resources | |
| css | |
| js | |
| views | |
| routes | |
| console.php | |
| web.php | |
| storage | |
| app | |
| framework | |
| logs | |
| tests | |

Cuando se crea una aplicación con el asistente se define una estructura de directorios y carpetas con funcionalidades establecidas. Vamos a ver cada una de ellas.

La carpeta **app** es la carpeta central del código proyecto que nosotros desarrollaremos, tiene un directorio para cada parte del paradigma Modelo y Controlador. La subcarpeta **Models** implementará una clase por cada una de las tablas de la BBDD, y la subcarpeta **Controllers** recogerá toda la lógica de ejecución de las peticiones del cliente. La subcarpeta **Http** contiene también el middleware (Software que se ejecuta después de recibir una petición pero antes de llamar al controlador, permite controlar el flujo de ejecución de una llamada, por ejemplo, que se esté autorizado).

La carpeta **bootstrap** es el punto central de una aplicación Laravel, raramente deberemos tocar algo en ella.

La carpeta **config** contiene todos los ficheros de configuración, se explicarán las opciones más importantes en el punto siguiente.

En la carpeta **database** aparecerá todo lo relacionado con la base de datos, las migraciones de las tablas, los datos de prueba o seeders, las clases factoría, y en caso de usar una base de datos SQLite se almacenará también aquí.

La carpeta **lang**, por defecto no aparece, recogerá todos los ficheros de traducciones de nuestra aplicación, si queremos configurarla hay que indicarlo a Laravel mediante **php artisan Lang:publish**

La carpeta **public** contiene el punto de entrada a nuestra aplicación, el fichero index.php, es la ruta a configurar en el servidor web como directorio raíz para que funcione correctamente la aplicación. También

contendrá todos los ficheros estáticos que se generen para su funcionamiento, tales como CSS, imágenes, Javascript, etc.

La carpeta **resources** alberga los recursos de nuestra aplicación sin compilar, en concreto recoge las vistas (views) que se utilizarán para generar las páginas HTML de regreso al usuario.

La carpeta **routes** contiene las urls de entrada a nuestra aplicación. En concreto en el fichero **web.php** deben aparecer todas las rutas disponibles de forma directa o mediante inclusión de otro fichero. En caso que queramos crear un API Rest, las urls se definirán en el fichero **api.php**. Todos los ficheros aquí definidos podrán tener rutas y se cargarán automáticamente en el sistema.

La carpeta **storage** recogerá todos los ficheros de registros, plantillas compiladas, ficheros de sesión, ficheros de cache y otros ficheros generados por el Framework. Se puede utilizar para almacenar cualquier archivo generado por la aplicación.

En la carpeta **test** crearemos los test de la aplicación implementados bajo el Framework correspondiente configurado al inicializar el proyecto.

Opciones de configuración

Configuración del entorno

En el raíz del proyecto se habrá creado un fichero llamado: **.env** en el que podemos configurar nuestra aplicación, este fichero es utilizará durante el desarrollo ya que en el entorno de producción se deberá borrar, siendo obligatorio dicha configuración dentro de los ficheros de la carpeta **config**.

```
APP_NAME=Laravel
APP_ENV=local
APP_KEY=base64:v3yf0BwG6MU0iHMiJ+gSQC FUHV2yMZA H6F3Qi(
APP_DEBUG=true
APP_URL=http://pr01.test

APP_LOCALE=es
APP_FALLBACK_LOCALE=es
APP_FAKE_RER_LOCALE=es_ES
```

La primera sección es la configuración de la Aplicación que deberá coincidir con los datos de nuestro proyecto, es muy importante que la configuración del idioma sea adecuada. En esta parte está el nombre de la aplicación y el tipo de ejecución local (desarrollo) o production (producción), una clave de aplicación usada para cifrar (debería implementarse una política de cambio de clave en entornos de producción) por ejemplo, los tokens del API y si está en modo depuración o no. Para finalizar se configura la url de la App.

En el segundo bloque aparecen las configuraciones de idioma, en este caso para una aplicación en español.

Por defecto Laravel utilizará este fichero si existe, en caso que no sea así, en entornos de producción usará los valores del directorio **config**.

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=laravel
DB_USERNAME=root
DB_PASSWORD=
```

```
DB_CONNECTION=sqlite
# DB_HOST=127.0.0.1
# DB_PORT=3306
# DB_DATABASE=laravel
# DB_USERNAME=root
# DB_PASSWORD=
```

Otra sección importante del archivo de entorno es la parte de la gestión de la Base de Datos. En la imagen se ven dos posibles configuraciones. Es obligatorio tener al menos una Base de datos configurada para el funcionamiento de cualquier aplicación Laravel. En la primera se ve una conexión con Mysql, en la segunda una bajo SQLite. Podemos cambiar en cualquier momento de una a otra modificando los valores, pero hay que tener en cuenta que solo existirán las tablas obligatorias de **Laravel** en aquella configuración de BBDD elegida durante el proceso de instalación, en el resto de configuraciones deberemos lanzar las ordenes: **php artisan migrate:install** y **php artisan migrate** en este orden para que realicen todas las configuraciones en la Base de datos iniciales. **Se recomienda no cambiar de gestor de BBDD en medio de un proyecto.** El fichero de producción se encuentra en **config/database.php**.

Configurar Bootstrap para Laravel

Práctica Laravel 1

INSTALAR LOS FICHEROS NECESARIOS

```
composer require laravel/ui
php artisan ui bootstrap
npm install
```

MODIFICAR LOS SIGUIENTES ARCHIVOS A TRAVÉS DEL IDE

Debemos indicar al sistema que vamos a usar Bootstrap en vez de tailwind, por lo que nos dirigiremos al fichero **resources/css/app.css** y cambiaremos la primera línea como se muestra en la imagen de al lado. Con esto se cargarán los archivos correctamente en el sistema, solo tendremos que hacer uso de ellos en las páginas. En la plantilla base generalmente en:

```
resources/view/layouts/base.blade.php
```

(si no existe la ruta y el fichero lo crearemos) hay que añadir en la cabecera (<HEAD>) al final, la sentencia:

```
@vite(['resources/css/app.css', 'resources/js/app.js'])
```

por supuesto, no nos podemos olvidar, tal y como se vio en el capítulo de Bootstrap establecer la etiqueta **meta viewport** de forma adecuada.

Con estos pasos, podremos hacer uso de las clases Bootstrap en nuestras etiquetas en cualquier parte de la aplicación. Para servir contenido estático se crearán directorios dentro de **resources** y para incluirlo en las plantillas Blade (ver más adelante) se usará código similar a:

```
{{ Vite::asset('resources/images/logo.png') }}
```

Práctica Laravel 2

Rutas

Nomenclatura Laravel: **Facade**, clase propia de Laravel que permite el acceso a servicios del Framework a través de métodos estáticos.

El enrutamiento es una parte fundamental de cualquier aplicación web. Es el proceso de dirigir solicitudes de los usuarios a diferentes acciones que se ejecutan en el servidor. El enrutamiento en Laravel es el proceso de definir las rutas de una aplicación web y cómo deben ser manejadas. Una ruta se define como una URL y una acción que se ejecuta en el servidor cuando se accede a ella. La acción puede ser una función, un método de un controlador que devuelve una respuesta HTTP o de un componente. El enrutador de Laravel toma la URL solicitada por el usuario y la compara con las rutas definidas en la aplicación para determinar la acción que debe ser ejecutada.

Todas las rutas de Laravel están definidas en sus archivos de ruta, que se encuentran en el directorio de **routes**. Estos archivos son cargados automáticamente por Laravel utilizando la configuración especificada en el archivo de arranque **bootstrap/app.php** de la aplicación. El archivo **routes/web.php** define las rutas que son para su interfaz web.

El fichero **routes/api.php** define las rutas para un API REST. Este último mecanismo permitirá conexiones sin estado per necesitas una instalación inicial, para la que hay que lanzar el siguiente comando

```
php artisan install:api
```

que instalará el sistema de autenticación Laravel Sanctum más potente que el sistema de autorización que se configura por defecto.

Cómo definir rutas en Laravel

Para definir una ruta en Laravel, se utilizan los métodos: **Route::get()**, **Route::post()**, **Route::put()**, **Route::patch()**, **Route::delete()** o **Route::options()**. Estos métodos corresponden a los diferentes verbos HTTP (GET, POST, PUT, PATCH y DELETE) que se utilizan para manipular recursos en la web. Cada uno de estos verbos HTTP tiene una función específica en la manipulación de los datos en la aplicación.

- **GET**: se utiliza para obtener recursos del servidor. Por ejemplo, si queremos obtener información de un usuario, podemos utilizar una ruta con el verbo GET.
- **POST**: se utiliza para crear recursos en el servidor. Por ejemplo, si queremos crear un nuevo usuario en la base de datos, podemos utilizar una ruta con el verbo POST.
- **PUT**: se utiliza para actualizar recursos completos en el servidor. Por ejemplo, si queremos actualizar toda la información de un usuario en la base de datos, podemos utilizar una ruta con el verbo PUT.
- **PATCH**: se utiliza para actualizar parte de un recurso en el servidor. Por ejemplo, si queremos actualizar solamente el correo electrónico de un usuario en la base de datos, podemos utilizar una ruta con el verbo PATCH.
- **DELETE**: se utiliza para eliminar recursos del servidor. Por ejemplo, si queremos eliminar un usuario de la base de datos, podemos utilizar una ruta con el verbo DELETE.
- **OPTIONS**: solicita las opciones de comunicación permitidas para una URL o un servidor determinado. Puede utilizarse para comprobar los métodos HTTP permitidos para una petición, o

para determinar si una petición tendría éxito al realizar una petición CORS. Un cliente puede especificar una URL con este método, o un asterisco (*) para referirse a todo el servicio. Esta opción no se suele implementar generalmente.

Para crear una ruta en Laravel, simplemente llamamos al método correspondiente según el verbo HTTP que queremos utilizar en el fichero **web.php**. El primer parámetro es la URL que se desea gestionar y el segundo es la acción que se ejecutará. Por ejemplo, si queremos manejar la URL "/inicio" y devolver una vista de bienvenida, podemos definir la ruta de la siguiente manera utilizando el verbo GET:

```
Route::get('/inicio', function () {
    return view('welcome');
});
```

En este ejemplo, estamos definiendo una ruta para manejar la URL "/inicio" y estamos devolviendo la vista "welcome" a través de una función anónima. La vista "welcome" es un archivo de plantilla que se encuentra en la carpeta "**resources/views**" con el nombre **welcome.blade.php**.

Para crear una ruta con un verbo HTTP diferente, simplemente cambiamos el método Route correspondiente. Por ejemplo, si queremos crear un nuevo usuario en la base de datos utilizando el verbo POST, podemos definir la ruta de la siguiente manera:

```
Route::post('/usuarios', function () {
    // Código para crear un nuevo usuario en la base de datos
});
```

En este ejemplo, estamos definiendo una ruta para manejar la URL "/usuarios" con el verbo POST. La función anónima que se ejecuta al acceder a esta dirección contiene el código para crear un nuevo usuario en la base de datos. De esta forma, podemos utilizar diferentes verbos HTTP en nuestras aplicaciones para manipular recursos de nuestra aplicación de acuerdo a la función específica de cada acción.

A veces, es posible que se deba registrar una ruta que responda a varios verbos HTTP. Puede hacerlo utilizando el método **match()**. O incluso puede registrar una ruta que responda a todos los verbos HTTP mediante el método **any()**.

```
Route::match(['get', 'post'], '/', function () {
    // ...
});
Route::any('/', function () {
    // ...
});
```

Todos los formularios HTML que apunten a rutas POST, PUT, PATCH o DELETE que se definen en el archivo de rutas web.php deben incluir un campo oculto de token CSRF, para que el middleware de protección pueda validar la solicitud. De lo contrario, la petición será rechazada. Para mayor comodidad, se puede utilizar la directiva Blade **@csrf** para generar el campo de entrada de token oculto:

```
<form method="POST" action="/profile">
    @csrf
    ...
</form>
```

Redirección a otra URL

Para crear una redirección, podemos usar el método de ayuda **redirect** en el controlador o en la ruta directamente con **Route::redirect**(en este caso el primer parámetro será la ruta de entrada), este método proporciona facilidades para realizarla sin tener que definir un controlador para la acción.

```
return redirect('/donde');
```

Por defecto redirect devuelve el código de estado HTTP 302 que puede ser configurado en la llamada.

```
return redirect( '/donde', 301);
```

O si el cambio es definitivo usar el método **Route::permanentRedirect** que devuelve un código de estado 301 en el fichero de rutas.

```
return Route::permanentRedirect('/aqui', '/donde');
```

Devolviendo vistas

Si la ruta necesita devolver una vista, puede usar el método de ayuda **view**. Al igual que el método de redireccionamiento, este método proporciona un acceso directo sencillo para que no se tenga que definir una ruta o un controlador completo. El método view acepta un nombre de vista como primer argumento. Además, puede proporcionar un Array de datos para pasar a la vista como segundo argumento opcional:

```
return view( 'welcome', [ 'name' => 'Taylor' ]);
```

Cómo pasar parámetros en las rutas

Es posible que se necesite pasar parámetros a la acción que se ejecuta cuando se accede a una ruta. Por ejemplo, si queremos mostrar el perfil de un usuario en una URL como "/usuarios/1", podemos definir la ruta de la siguiente manera:

```
Route::get('/usuarios/{id}', function ($id) { return "Mostrando perfil del usuario: ".$id; });
```

En este ejemplo, estamos definiendo una ruta para manejar direcciones que tienen un parámetro "id". Cuando un usuario accede a una URL que coincide con esta ruta, Laravel pasará automáticamente el valor del parámetro a la acción que se ejecuta. Los parámetros que se definen en la Uri, también hay que establecerlos en la función anónima o en el método del controlador, en el mismo orden y tipo que se quieren definir. El esquema se puede repetir a lo largo de las URLs, tantas veces como necesitemos.

```
Route::get('/post/{post}/{id}', function ($post,$id) {
    return "Mostrando perfil del usuario: $post..$id;
});
```

PARÁMETROS OPCIONALES

En ocasiones, es posible que se deba especificar un parámetro de ruta que no siempre esté presente en la URL. Podemos hacerlo colocando una interrogación(?) después del nombre del parámetro. Hay que asegurarse de asignar un valor predeterminado a la variable correspondiente de la ruta:

```
Route::get('/user/{name?}', function (?string $name = null) {
    return $name;
});
```

```
Route::get('/user/{name?}', function (?string $name = 'John') {
    return $name;
});
```

Cómo utilizar controladores en Laravel

Para mantener el código limpio y organizado, es una buena práctica utilizar controladores en lugar de definir todas las acciones en las rutas. Un controlador es una clase que contiene métodos que manejan las solicitudes de una ruta específica.

Para crear un nuevo controlador en Laravel, podemos usar el comando "**make:controller**" de Artisan. Este comando creará un nuevo archivo de controlador en el directorio "**app/Http/Controllers**" con un esqueleto básico del controlador. Para crear un controlador llamado "UsuariosController", podemos ejecutar el siguiente comando:

```
php artisan make:controller UsuariosController
```

Una vez que hemos creado el controlador, podemos agregar los métodos que necesitamos para manejar las solicitudes en las rutas correspondientes. Por ejemplo, si queremos manejar una solicitud GET en la ruta "/usuarios" con el método "index", podemos añadir el siguiente método al controlador:

```
public function index() {
    // Código para manejar la solicitud GET en la ruta "/usuarios"
}
```

Luego, para utilizar el controlador en una ruta, primero debemos importar la clase del controlador al archivo de rutas utilizando la directiva "**use**". Por ejemplo, si hemos creado la clase "UsuariosController" en el archivo "app/Http/Controllers/UsuariosController.php", podemos agregar la siguiente línea al comienzo del archivo de rutas:

```
use App\Http\Controllers\UsuariosController;
```

Después, para utilizar el controlador en una ruta, tenemos que especificar la clase del controlador y el nombre del método separados por una coma dentro de un array en el segundo parámetro de la ruta. Por ejemplo, si queremos utilizar el método "index" del controlador "UsuariosController" para manejar la ruta "/usuarios", podemos definir la ruta de la siguiente manera:

```
Route::get('/usuarios', [UsuariosController::class, 'index']);
```

En este ejemplo, estamos creando una ruta para manejar la URL "/usuarios" y estamos utilizando el método "index" del controlador "UsuariosController" para gestionar la acción.

Cómo utilizar grupos de rutas en Laravel

A veces es necesario aplicar ciertas funcionalidades a varias rutas de manera simultánea, como autenticación o autorización. En Laravel, es posible utilizar grupos de rutas para aplicar estas funcionalidades a varias rutas al mismo tiempo. Un grupo de rutas es simplemente un conjunto de rutas que comparten una característica en común.

Para establecer un grupo de rutas, se utiliza el método **Route::group()**. Dentro de este método, se puede definir las rutas que forman parte del grupo y las funcionalidades que se aplicarán a todas ellas. Por ejemplo, si queremos aplicar autenticación a todas las rutas que comienzan con "/admin", podemos definir el grupo de rutas de la siguiente manera:

```
Route::group(['prefix' => 'admin', 'middleware' => 'auth'], function () {  
    Route::get('/', [AdminController::class, 'index']);  
    Route::get('/usuarios', [UsuariosController::class, 'index']);  
    Route::get('/productos', [ProductosController::class, 'index']);  
});
```

En este ejemplo, hemos definido un grupo de tres rutas que comparten el prefijo "/admin". Además, estamos aplicando el middleware "auth" a todas las rutas del grupo, lo que significa que el usuario debe estar autenticado para acceder a ellas. Dentro del grupo, estamos definiendoUrls diferentes que utilizan los métodos "index" de otros tres controladores.

Cómo definir rutas con **Route::resource()**

Continuando con el tema de las rutas en Laravel, también existe una forma de generar varias rutas CRUD (Create, Read, Update, Delete) a través de una sola línea de código, y esto se logra mediante el uso de la función **Route::resource()**. Esta función permite definir un conjunto completo de rutas para un recurso de la aplicación.

Por ejemplo, si tenemos un recurso "producto" que necesita rutas CRUD, podemos definirlo de la siguiente manera:

```
Route::resource('productos', ProductosController::class);
```

En el ejemplo, estamos definiendo un conjunto completo de rutas para el recurso "productos", que estarán gestionadas por el controlador "ProductosController". Estas rutas incluyen las siguientes acciones:

- **GET /productos** - muestra una lista de todos los productos.
- **GET /productos/create** - muestra un formulario para crear un nuevo producto.
- **POST /productos** - guarda un nuevo producto.
- **GET /productos/{producto}** - muestra un producto específico.
- **GET /productos/{producto}/edit** - muestra un formulario para editar un producto existente.
- **PUT/PATCH /productos/{producto}** - actualiza un producto existente.
- **DELETE /productos/{producto}** - elimina un producto existente.

Es importante tener en cuenta que la función **Route::resource()** sigue una convención de nomenclatura específica. En este caso, el primer parámetro debe ser el nombre del recurso en plural, mientras que el segundo parámetro es el nombre del controlador que manejará las rutas. Además, todas las rutas generadas por **Route::resource()** se pueden modificar utilizando los métodos `except` o `only`, según sea necesario. **Ver en la parte de controladores la asociación con los métodos correspondientes del controlador las direcciones anteriores.**

En cuanto a la pregunta sobre cuándo conviene usar un grupo de rutas versus **Route::resource()**, depende del contexto y de las necesidades específicas de la aplicación. El uso de un grupo de rutas es útil cuando

se necesita aplicar ciertas funcionalidades, como autenticación o autorización, a varias rutas al mismo tiempo. Por otro lado, `Route::resource()` es útil cuando se necesitan rutas CRUD para un recurso específico en la aplicación. En general, se puede utilizar una combinación de ambos en una aplicación de Laravel para cubrir todas las necesidades de nuestra aplicación. La definición del controlador sería algo similar a:

```
class PostController extends Controller {
    /**
     * Display a listing of the resource.
     */
    public function index():View  { }

    /**
     * Show the form for creating a new resource.
     */
    public function create() { }

    /**
     * Store a newly created resource in storage.
     */
    public function store(Request $request) { }

    /**
     * Display the specified resource.
     */
    public function show(int $post_id) { }

    /**
     * Show the form for editing the specified resource.
     */
    public function edit(int $post_id){ }

    /**
     * Update the specified resource in storage.
     */
    public function update(Request $request, int $post_id) { }

    /**
     * Remove the specified resource from storage.
     */
    public function destroy(int $post_id) { }
}
```

Cómo utilizar rutas con nombres en Laravel

A veces es útil asignar un nombre a una ruta para poder hacer referencia a ella de manera más sencilla en otras partes de la aplicación. En Laravel, puedes asignar un nombre a una ruta utilizando el método `name()`. Por ejemplo, si queremos asignar un nombre a la ruta "/usuarios", podemos definirla de la siguiente manera:

```
Route::get('/usuarios', [UsuariosController::class, 'index'])->name('usuarios.index');
```

Aquí, estamos asignando el nombre "usuarios.index" a la ruta "/usuarios". Ahora podemos hacer referencia a ella en otras partes de la aplicación utilizando este nombre en lugar de la URL completa.

Cómo generar URLs en Laravel

En Laravel se pueden generar URLs para las rutas con nombre utilizando el método **route()**. Este método toma el nombre de la ruta que deseamos generar y cualquier parámetro necesario. Por ejemplo, si queremos crear la URL de la ruta con el nombre "usuarios.index", podemos hacerlo de la siguiente manera:

```
$url = route('usuarios.index');
```

Se está asignando la URL de la ruta con el nombre "usuarios.index" y almacenándola en la variable "\$url". Si la ruta requiere algún parámetro, podemos pasarlo como segundo argumento del método.

También se puede usar la función **url()**, que recoge la dirección tal y como la hemos definido en los ficheros dentro del directorio **routes** para crearla de forma dinámica. Es posible hacer una redirección a una ruta con nombre, similar a la función redirect, pero con nombres de ruta con el método **to_route**:

```
return to_route("nombre.ruta");
```

Práctica Laravel 3

Práctica Laravel 4

VALIDACIÓN

Laravel proporciona varios enfoques diferentes para validar los datos entrantes de su aplicación. Lo más habitual es utilizar el método **validate** disponible en todas las solicitudes HTTP entrantes.

Si se produce un error en la validación durante una solicitud HTTP tradicional, se generará una respuesta de redireccionamiento a la URL anterior. Si la solicitud entrante es una solicitud XHR, se devolverá una respuesta JSON que contenga los mensajes de error de validación. Para comprender mejor el método de validación, volvamos al método de store:

```
public function store(Request $request): RedirectResponse {
    $validated = $request->validate([
        'title' => 'required|unique:posts|max:255',
        'body' => 'required',
    ]);
    return redirect('/posts');
}
```

Como puede ver, las reglas de validación se pasan a la validación. Alternativamente, las reglas de validación se pueden especificar como matrices de reglas en lugar de un solo | Cadena delimitada:

```
$validatedData = $request->validate([
    'title' => ['required', 'unique:posts', 'max:255'],
    'body' => ['required'],
```

]);

Existen gran cantidad de reglas que se pueden validar, se puede ver en <https://laravel.com/docs/12.x/validation#available-validation-rules>.

En caso de error existe una variable \$errors compartida con todas las vistas de la aplicación mediante el middleware Illuminate\View\Middleware\ShareErrorsFromSession, que proporciona el grupo de middleware web. Cuando se aplica este middleware, una variable \$errors siempre estará disponible en sus vistas, lo que le permite asumir convenientemente que la variable \$errors siempre está definida y se puede usar de manera segura.

```
@if ($errors->any())
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif
```

Cuando Laravel genera una respuesta de redirecciónamiento debido a un error de validación, el marco mostrará automáticamente todas las entradas de la solicitud a la sesión. Esto se hace para que pueda acceder cómodamente a la entrada durante la próxima solicitud y volver a llenar el formulario que el usuario intentó enviar.

```
$title = $request->old('title');
```

Para recuperar la entrada de la solicitud anterior, invoque el método anterior en una instancia de Illuminate\Http\Request. El método anterior extraerá los datos de entrada previamente flasheados de la sesión. Laravel también proporciona un método **old** global. Si muestra una entrada antigua dentro de una plantilla de Blade, es más conveniente utilizar el asistente **old** para volver a llenar el formulario. Si no existe ninguna entrada antigua para el campo dado, se devolverá null:

```
<input type="text" name="title" value="{{ old('title') }}">
```

Para escenarios de validación más complejos, es posible que se desee crear un "form request". Son clases de que extienden del formulario personalizadas que encapsulan su propia lógica de validación y autorización. Para más información visitar la url: <https://laravel.com/docs/12.x/validation#form-request-validation>.

Entendiendo la utilidad Artisan

Artisan es un interfaz de línea de comandos incluida con Laravel. Se instala en la raíz de la aplicación como un script llamado **artisan.php** y proporciona una serie de comandos útiles que pueden ayudar mientras se desarrolla la aplicación. Para ver una lista de todos los comandos de Artisan disponibles, se puede usar el comando **list**.

```
php artisan list
```

Cada comando incluye una pantalla de "ayuda" que muestra y describe los argumentos y opciones disponibles del comando. Para ver la pantalla de ayuda, se precede el nombre del comando con help.

```
php artisan help migrate
```

Dentro de los comandos que proporciona Artisan están:

- Crear migraciones de bases de datos, **php artisan make:migration nombre**.
- Generar conjuntos de datos de prueba (seeds), **php artisan make:seeder Nombre**.
- Generar controladores y otros tipos de archivos, **php artisan make:controler NombreControlador**.
- Levantar un servidor de pruebas, **php artisan serve**.
- Ejecutar migraciones, **php artisan migrate**.
- Limpiar cachés.
 - **php artisan cache:clear**
 - **php artisan config:clear**
- Manejar la base de datos. Se verán más adelante.
- Crear un modelo de BBDD, **php artisan make:model [nombre_de_Modelo_en_Singular] -a**.
- Creación de siete rutas para CRUD, **php artisan make:resource [nombre]**.

Blade

Introducción

```
https://laravel.com/docs/12.x/blade
```

Blade es el simple, pero potente motor de plantillas que se incluye con Laravel. A diferencia de otros motores de plantillas PHP, Blade no restringe el uso de código PHP plano en las plantillas. De hecho, todas las plantillas Blade se compilan en código PHP plano y se almacenan en caché hasta que se modifican, lo que significa que Blade no añade sobrecarga a la aplicación. Los archivos de plantilla Blade utilizan la extensión: **.blade.php** y se almacenan normalmente en el directorio **resources/views**.

Para poder usar una vista desde un controlador, el nombre de la plantilla tiene que coincidir con el que pasamos en la función **view** más la extensión blade.php.

Directivas

Blade tiene dos tipos de patrones de escape para su gestión, el que se utiliza para devolver código HTML y el que se utiliza para controlar la ejecución. El primer caso hace uso de la estructura **{{ }}**. Todo código incluido dentro se devolverá como una cadena en dicha posición para ser interpretado como HTML. Podemos incluir solo variables, expresiones o cualquier elemento que se pueda resolver a una cadena de texto. Por defecto la estructura con llaves utiliza la función **htmlspecialchars** para evitar ataques XSS. Si no deseamos que sea llamada dicha función de php habrá que usar: **{!! \$name !!}**.

Para controlar el flujo de ejecución de la plantilla se usa la sintaxis **@palabra_reservada**. Las directivas así enunciadas dicen al motor de Blade que debe ocurrir algo durante la ejecución. En ese momento Blade, tanto las llaves como la arroba se interpretan por lo que, si necesitamos crear ese texto en la página HTML devuelta (como parte de algún Framework cliente) no sería posible. Para incluir la sintaxis reservada sin

modificar, debemos añadir una arroba, por lo que las llaves se convertirían @{{...}} y las directivas en @@nombre, quedando reflejadas en la página de salida como {{...}} y @nombre respectivamente sin ser ejecutadas por el motor de Blade.

DIRECTIVAS MÁS COMUNES

- @if

```
@if (count($records) === 1)
    I have one record!
@elseif (count($records) > 1)
    I have multiple records!
@else
    I don't have any records!
@endif
```

- @isset y @empty

```
@isset($records)
    // $records is defined and is not null...
@endisset

@empty($records)
    // $records is "empty"...
@endempty
```

- @auth y @guest. Estar autorizado, no estar autorizado

```
@auth
    // The user is authenticated...
@endauth

@guest
    // The user is not authenticated...
@endguest
```

- @hasSection. Si existe la sección en la plantilla.

```
@hasSection('navigation')
<div class="pull-right">
    @yield('navigation')
</div>
<div class="clearfix"></div>
@endif
```

- @session se puede usar para determinar si un valor existe en la sesión actual.

```
@session('status')
<div class="p-4 bg-green-100">
    {{ $value }}
</div>
@endsession
```

- @switch

```
@switch($i)
  @case(1)
    First case...
    @break
  @case(2)
    Second case...
    @break
  @default
    Default case...
@endswitch
```

- Bucles

```
@for ($i = 0; $i < 10; $i++)
  The current value is {{ $i }}
@endfor
@foreach ($users as $user)
  <p>This is user {{ $user->id }}</p>
@endforeach
@forelse ($users as $user)
  <li>{{ $user->name }}</li>
@empty
  <p>No users</p>
@endforelse
@while (true)
  <p>I'm looping forever.</p>
@endwhile
```

- @php. Ejecutar código php arbitrario.

```
@php
  $isActive = false;
  $hasError = true;
@endphp
```

- @checked, @selected. Escribirá checked/selected si la condición es verdadera.

```
<input
  type="checkbox"
  name="active"
  value="active"
  @checked(old('active', $user->active))
/>

<select name="categorie_id" id="categorie_id" required>
  @foreach ($categories as $category)
    <option value="{{ $category->id }}"
      @selected( $post->categorie_id == $category->id )>{{ $category->categorie }}</option>
  @endforeach
</select>
```

- **@disabled, @readonly, @required.** Similar al anterior, pero con los correspondientes atributos HTML.
- **@csrf.** Incluye el token para evitar ataques. Todo formulario debe de añadirlo.

Herencia

El sistema de plantillas de Blade permite la herencia entre diferentes plantillas, así como la inclusión de unas dentro de otras. En la imagen siguiente vemos como se determina la herencia. Para la inclusión, solamente hay que utilizar **@include('nombre_vista',[parámetros_de_la_vista])** en la posición deseada.

```
php ventas.blade.php ×
1  @extends('layouts.base') <- directorio.fichero    plantilla a heredar
2  @section('contenido') <- sección a rellenar en plantilla base
3  <div class="row alert alert-primary">
4      <h1>Ventas</h1>
5  </div>
6
7  <div class="row">
8      <p>Esta es la página de ventas</p>
9  </div>
10 <div class="row">
11     <ul class="list-group">
12         <li class="list-group-item">Venta 1</li>
13         <li class="list-group-item">Venta 2</li>
14         <li class="list-group-item">Venta 3</li>
15     </ul>
16 </div>
17 @endsection <- fin de sección
```



```
php base.blade.php ×
1  <!doctype html>
2  <html lang="es">
3      <head>
4          <meta charset="UTF-8">
5          <meta name="viewport" content="width=device-width, initial-scale=1.0, shrink-to-fit=no">
6          <title>Ventas</title>
7          <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
8      </head>
9      <body>
10         <div class="container">
11             <div class="row">
12                 <div class="col-m9 justify-content-center">
13                     @yield('contenido') <- Sección a rellenar
14                 </div>
15             </div>
16         </div>
17         <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js">
18     </body>
19 </html>
```

Ilustración 16 - Herencia Blade

La herencia se establece marcando la sección en la que las plantillas hijas pueden incluir su contenido con el comando **@yield**, se delimitará en la plantilla base. Este comando acepta un nombre que será utilizado por las plantillas hijas, podemos definir tantas áreas diferentes como necesitemos con nombres siempre y cuando el nombre no coincida. A su vez, la plantilla que queremos hacer “hija” debe establecer **@extends('nombre_directorio.nombre_vista')** como primer comando y a continuación definir tantas secciones como desee implementar con el nombre usado en la plantilla padre, con las órdenes **@section...@endsection**, no es necesario que estén todas las secciones que define la plantilla padre. Ver imagen anterior.

Organización en carpetas

El directorio en el que deben residir todas las plantillas es como ya hemos visto **resources/views**. Pero a partir de ahí podemos crear la estructura de carpetas que deseemos, tan compleja como necesitemos. Hay que tener en cuenta que este directorio se marca como raíz de las plantillas por lo que, para acceder a una en concreto para extender, o visualizar, habrá que localizarla desde aquí, separando por puntos cada directorio. Así para una plantilla llamada **base.blade.php** situada en el directorio **resources/views/layouts/iniciales/**, la directiva **@extends** de otra plantilla tendría que ser **@extends('layouts.iniciales.base')** y la ruta desde una vista sería algo del estilo **view('layouts.iniciales.base');**.

Paso de parámetros

```
route.get(...){
    return view('mi_plantilla', ['nombre'=>$nombre];

    mi_plantilla.blade.php
    ...
<p>{{ $nombre }}</p>
```

Las plantillas admiten parámetros cuando son mostradas a través de la orden **view**. El segundo parámetro debe ser un Array asociativo, en el que cada clave se convertirá en una variable dentro de la plantilla, y el valor que tendrá será accesible a través de él.

ORDEN COMPACT

<https://www.php.net/manual/en/function.compact.php>

Crea un array que contiene variables con sus valores. Para cada una de ellas, **compact()** busca una variable con ese nombre declarada en el ámbito actual con un valor definido, la añade al array de salida de forma que el nombre de la variable se convierte en la clave y el contenido de la variable se convierte en el valor de esa clave. Esta función permite aligerar el peso de la conversión a la hora de pasar parámetros a las vistas.

```
$post = Post::paginate(2);
return view('...', ['post'=>$post]);
```

Es lo mismo que

```
$post = Post::paginate(2);
return view('...', compact('post'));
```

Práctica Laravel 5

Bases de datos

Migraciones

Las migraciones en Laravel son una forma de control de versiones para tu base de datos. Permiten sincronizar de manera ordenada los cambios en la estructura de las tablas (creación, modificación, eliminación) con el historial de tu proyecto. Gracias a esto, varios desarrolladores pueden trabajar sobre la misma base de datos sin generar conflictos y sin necesidad de compartir manualmente archivos SQL.

Principales ventajas:

- Facilitan el despliegue de cambios en producción.
- Permiten mantener un historial ordenado de modificaciones.
- Hacen que la colaboración entre equipos sea más fluida.
- Evitan tener que crear y compartir manualmente scripts de actualización de la base de datos.

Por defecto, Laravel incluye un conjunto básico de migraciones dentro de la carpeta **database/migrations**. Cada archivo de migración se nombra siguiendo la convención de fecha, hora y nombre, por ejemplo: `0001_01_01_000000_create_users_table.php`.

Dentro de estas migraciones por defecto, se encuentra:

- `users_table.php`: crea la tabla `users`, `password_reset_tokens` y `sessions`.
- `cache_table.php`: crea la tabla `cache` y `cache_locks`.
- `jobs_table.php`: crea la tabla `jobs`, `job_batches` y `failed_jobs`.

Si es la primera vez que lanzas las migraciones, asegúrate de configurar la conexión a la base de datos en el archivo: `.env` correctamente o en el directorio **config**, debería estarlo si has usado el instalador Laravel. Después, en tu terminal, ejecuta:

```
php artisan migrate
```

Este comando creará las tablas en la base de datos configurada si no lo están.

CREAR NUEVAS MIGRACIONES

Para crear una nueva migración (nuevas tablas, modificación de ellas, etc...), utilizamos el comando:

```
php artisan make:migration accion_nombretabla_table
```

Este comando creará un archivo de migración nuevo en la carpeta **database/migrations** con un nombre como: `2024_04_26_123456_accion_nombretabla_table.php`. La estructura del nombre es importante si queremos que Laravel accede a muchas funciones de bases de datos de forma automática, el nombre debe estar en plural. Un ejemplo de migración básica para la tabla `posts` podría ser:

```
<?php  
use Illuminate\Database\Migrations\Migration;  
use Illuminate\Database\Schema\Blueprint;  
use Illuminate\Support\Facades\Schema;  
  
return new class extends Migration {  
    public function up()  {  
        Schema::create('posts', function (Blueprint $table) {  
            $table->id();  
            $table->string('title');  
            $table->text('content');  
            $table->timestamps();  
        });  
    }  
    public function down()  {  
        Schema::dropIfExists('posts');  
    }  
}
```

```
};
```

El método **up()** describe lo que sucede cuando aplicamos la migración. El método **down()** marca lo que sucede cuando revertimos (o hacemos rollback) la migración. En este caso creamos cinco campos, id, title, content, y dos para los timestamps. Es obligatorio definir ambos métodos y en el caso del **down()** asegurarse que la Base de datos queda en el mismo estado que antes de la modificación de método **up()**.

TIPOS DE COLUMNAS Y OPCIONES COMUNES

Laravel proporciona un constructor que hace fácil definir los tipos de columnas y sus atributos:

- `string('title', 255)`: Crea un VARCHAR(255).
- `text('content')`: Crea un TEXT.
- `integer('votes')`: Crea un INT.
- `boolean('is_published')`: Crea un TINYINT(1).
- `date('published_date')`: Crea un DATE.
- `foreignId('user_id')->constrained()`: Crea una columna user_id como clave foránea apuntando a la tabla users. Si no seguimos la nomenclatura Laravel, dentro de método **constrained** se indicará la tabla a la que se referencia.
- Consultar la documentación oficial para ver todos los tipos de campos (<https://laravel.com/docs/12.x/migrations#available-column-types>).

Además, puedes añadir atributos como:

- `nullable()`: Permite valores NULL.
- `default('texto por defecto')`: Asigna un valor por defecto.
- `unique()`: Crea un índice único.
- `index()`: Crea un índice básico.

```
<?php  
Schema::create('posts', function (Blueprint $table) {  
    $table->id();  
    $table->string('title')->unique();  
    $table->string('slug')->nullable()->index();  
    $table->text('content')->nullable();  
    $table->integer('visits')->default(0);  
    $table->timestamp('published_at')->nullable();  
    $table->foreignId('user_id')->constrained()->onDelete('cascade');  
    $table->timestamps();  
});
```

MODIFICANDO TABLAS A TRAVÉS DE MIGRACIONES

Añadir columnas a una tabla existente en Laravel

Es muy necesario añadir columnas a tablas existentes. Para esta acción, añadir una columna a una tabla existente en Laravel, debemos hacer uso de migraciones. Primero se creará una nueva migración usando el comando **make:migration**.

```
php artisan make:migration add_status_column_on_users_table --table=users
```

El comando anterior habrá creado un nuevo archivo de migración en el directorio **database/migrations** de nuestro proyecto. Al haber utilizado el flag **--table=users** al generar nuestra migración, Laravel sabe que queremos interactuar con nuestra tabla users y ha creado ese pequeño código en los métodos up y down con lo que necesitamos. En este punto sólo queda ajustar los métodos up y down para añadir una nueva columna:

```
<?php  
use Illuminate\Database\Migrations\Migration;  
use Illuminate\Database\Schema\Blueprint;  
use Illuminate\Support\Facades\Schema;  
  
return new class extends Migration {  
    public function up(): void {  
        Schema::table('users', function (Blueprint $table) {  
            $table->string('status')->after('email')->default('active');  
        });  
    }  
    public function down(): void {  
        Schema::table('users', function (Blueprint $table) {  
            $table->dropColumn('status');  
        });  
    }  
};
```

El método up añadirá la columna status de tipo varchar a la tabla users después de la columna email utilizando por defecto el valor active. El método down eliminará la columna cuando hagamos un rollback.

Para que nuestra columna se refleje en la tabla de usuarios del servidor, hay que ejecutar el siguiente comando en nuestro proyecto:

```
php artisan migrate
```

Si ahora se revisa la tabla de usuarios encontraremos que se ha actualizado añadiendo la nueva columna. Para deshacer este cambio, es decir, que nuestra columna status sea eliminada de la tabla users, tendríamos que hacer un rollback de la siguiente forma:

```
php artisan migrate:rollback
```

El comando anterior eliminará la columna status de la tabla users, pero hay que tener cuidado, si se vuelve a ejecutar, también deshará todo lo que se había creado la última vez que se ejecutó el comando **php artisan migrate** y ya no será sólo nuestra nueva columna lo que habremos deshecho.

Modificar columnas en Laravel usando migraciones

Para modificar una columna existente en una tabla en Laravel usando migraciones, primero debemos crear una nueva migración usando el comando **make:migration**.

```
php artisan make:migration modify_status_column_on_users_table --table=users
```

Una vez creada la migración, podemos modificar la columna status haciendo que su longitud sea de 20 caracteres, en lugar de 255 que es el valor por defecto en MySQL utilizando el método **change()**:

```
<?php  
use Illuminate\Database\Migrations\Migration;  
use Illuminate\Database\Schema\Blueprint;  
use Illuminate\Support\Facades\Schema;  
  
return new class extends Migration {  
    public function up(): void {  
        Schema::table('users', function (Blueprint $table) {  
            $table->string('status', 20)->change();  
        });  
    }  
    public function down(): void {  
        Schema::table('users', function (Blueprint $table) {  
            $table->string('status')->change();  
        });  
    }  
};
```

El método up() cambia la longitud a 20. El método down() devolverá la longitud a 255.

Eliminar columnas en Laravel usando migraciones

El proceso para eliminar columnas utilizando migraciones en Laravel es el mismo, primero debemos crear una nueva migración usando el comando **make:migration**.

```
php artisan make:migration drop_status_column_on_users_table --table=users
```

Una vez tenemos nuestra migración, debemos definir los métodos up y down.

```
<?php  
use Illuminate\Database\Migrations\Migration;  
use Illuminate\Database\Schema\Blueprint;  
use Illuminate\Support\Facades\Schema;
```

```

return new class extends Migration {
    public function up(): void {
        Schema::table('users', function (Blueprint $table) {
            $table->dropColumn('status');
        });
    }

    public function down(): void {
        Schema::table('users', function (Blueprint $table) {
            $table->string('status')->after('email')->default('active');
        });
    }
};

```

El método `up()` elimina la columna `status`. El método `down()` vuelve a crear la columna `status`, es imprescindible que la tabla quede en el estado que anterior a esta modificación.

Migraciones avanzadas: renombrar tablas

```
php artisan make:migration rename_posts_to_articles --table=posts
```

Dentro de la migración:

```

<?php
public function up() {
    Schema::rename('posts', 'articles');
}
public function down() {
    Schema::rename('articles', 'posts');
}

```

Uso de Seeders junto a las migraciones

Es muy común querer insertar datos de prueba inmediatamente después de lanzar una migración. Para ello, se usan los Seeders. Para crea un seeder:

```
php artisan make:seeder ArticlesTableSeeder
```

En el archivo generado (por defecto en `database/seeders`), se escribe la lógica de inserción:

```

<?php
public function run()
{
    \DB::table('articles')->insert([
        [ 'title' => 'Mi primer post',

```

```
'content' => 'Contenido de ejemplo',
'user_id' => 1,
'created_at' => now(),
'updated_at' => now(),
],
// ...
]);
}
```

A continuación, se podrá ejecutar la carga de datos de prueba:

```
php artisan db:seed --class=PostsTableSeeder
```

Para un enfoque más escalable, lo ideal es utilizar Factorías (Laravel Factories) junto a Seeders para generar datos de prueba masivamente.

Rollbacks, refrescar y reiniciar migraciones

Durante el desarrollo, es muy útil revertir los cambios aplicados por una migración. Los comandos principales son:

- **php artisan migrate:rollback**: Revierte la última migración.
- **php artisan migrate:rollback --step=2**: Revierte las dos últimas migraciones.
- **php artisan migrate:refresh**: Revierte todas las migraciones y luego las ejecuta de nuevo.
- **php artisan migrate:fresh**: Elimina todas las tablas y ejecuta todas las migraciones desde cero.

BUENAS PRÁCTICAS PARA LAS MIGRACIONES

- Hacer una migración por cada cambio de esquema. Mantener las migraciones atómicas y pequeñas para facilitar las revisiones de código y la resolución de conflictos.
- No modificar migraciones ya publicadas.
- Una vez que una migración ha sido aplicada en producción, no se modifica. En su lugar, se crea una nueva migración para reflejar el cambio.
- Prestar atención a los índices. La creación de índices y claves foráneas puede afectar el rendimiento cuando las tablas ya están muy pobladas.
- Hacer uso de transacciones. En algunos sistemas de base de datos (p. ej., PostgreSQL), Laravel permite envolver las migraciones en transacciones (configurando DB::beginTransaction()) para garantizar atomicidad.
- Realizar Backups antes de migrar. Especialmente en producción, mantener una política de backups para evitar pérdidas irreversibles de datos.

Práctica Laravel 6

Acceso a los datos a través del ORM

```
https://laravel.com/docs/12.x/eloquent
```

RELACIONES

Las relaciones se definen como **métodos** en las clases del **modelo**. Dado que las relaciones también sirven como generadores de consultas, la definición de relaciones como métodos proporciona capacidades de encadenamiento de métodos y consultas.

Relaciones uno a uno (Has One)

Una relación uno a uno es un tipo muy básico de relación de base de datos. Por ejemplo, un modelo de usuario puede estar asociado a un modelo de teléfono. Para definir esta relación, colocaremos un método **telefono()** en el modelo de usuario. El método **telefono** debe llamar al método **hasOne()** y devolver su resultado.

```
<?php  
namespace App\Models;  
use Illuminate\Database\Eloquent\Model;  
use Illuminate\Database\Eloquent\Relations\HasOne;  
  
class User extends Model {  
    public function telefono(): HasOne {  
        return $this->hasOne(Phone::class);  
    }  
}
```

El primer y único argumento que se pasa al método **hasOne** es el nombre de la clase de modelo relacionada. Una vez definida la relación, podemos recuperar el registro relacionado llamando a dicho método como una propiedad.

```
$phone = User::find(1)->phone;
```

Eloquent determina la clave externa de la relación en función del nombre del modelo principal. En este caso, se supone automáticamente que el modelo de **Phone** tiene una clave externa llamada **user_id** definida. El ORM determina el nombre de la clave externa examinando el nombre del método de relación y sufijando el nombre del método con **_id**. Si el nombre que se ha dado al campo es otro, se puede pasar un segundo argumento al método **hasOne** indicándolo:

```
return $this->hasOne(Phone::class, 'nombre_foreign_key_en_tabla_Phone');
```

Definición de la relación inversa

Toda tabla relacionada tiene una relación inversa en la tabla con la que se relaciona. Es decir, podremos acceder al modelo de teléfono desde nuestro modelo de usuario (la relación inversa) si la creamos. A continuación, definimos una relación en el modelo **Phone** que nos permitirá acceder al usuario propietario del teléfono. Podemos definir la inversa de una relación **hasOne** usando el método **belongsTo**:

```
<?php
namespace App\Models;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;

class Phone extends Model
{
    public function user(): BelongsTo {
        return $this->belongsTo(User::class);
    }
}
```

Relación Uno a muchos (Has Many)

Una relación de uno a varios se utiliza para definir relaciones en las que un único modelo es el elemento primario de uno o varios modelos secundarios. Por ejemplo, una entrada de blog puede tener un número infinito de comentarios. Al igual que todas las demás relaciones, las relaciones de uno a varios se definen programando un método en el modelo que devuelve la llamada a **hasMany()**:

```
<?php
namespace App\Models;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\HasMany;

class Post extends Model {
    public function comments(): HasMany {
        return $this->hasMany(Comment::class);
    }
}
```

Es posible crear las relaciones inversas de las tablas relacionadas también con este método.

Valores por defecto

Las relaciones **belongsTo** y **hasOne**, permiten definir un modelo o valor predeterminado que se devolverá si la relación especificada es nula

```
public function user(): BelongsTo {
    return $this->belongsTo(User::class)->withDefault();
}

public function user(): BelongsTo {
    return $this->belongsTo(User::class)->withDefault([
        'name' => 'Guest Author', ]);
}
```

Acceso rápido a valores recientes

A veces, un modelo puede tener muchos modelos relacionados, pero se desea recuperar fácilmente el "más reciente" o "más antiguo" de la relación

```
public function latestOrder(): HasOne {  
    return $this->hasOne(Order::class)->latestOfMany(); }
```

Del mismo modo, se puede definir un método para recuperar el modelo "más antiguo" o el primero relacionado de una relación:

```
public function oldestOrder(): HasOne {  
    return $this->hasOne(Order::class)->oldestOfMany(); }
```

Relaciones muchos a muchos

Las relaciones de varios a varios son un poco más complicadas que las relaciones hasOne y hasMany. Se definen escribiendo un método que devuelve el resultado del método **belongsToMany**. Para definir esta relación, se necesitan tres tablas en la base de datos creadas, las dos relacionadas y la que implementa la relación. Para determinar el nombre de la tabla intermedia de la relación, el ORM unirá los dos nombres de modelo relacionados en orden alfabético. Sin embargo, se puede cambiar esta nomenclatura, pasando un segundo argumento al método **belongsToMany**:

```
return $this->belongsToMany(Role::class, 'role_user');
```

Además de personalizar el nombre de la tabla intermedia, también se pueden establecer los nombres de columna de las claves de la tabla, pasando argumentos adicionales al método **belongsToMany**. El tercer argumento es el nombre de clave externa del modelo en el que se está definiendo la relación, mientras que el cuarto argumento es el nombre de clave externa del modelo al que se está uniendo:

```
return $this->belongsToMany(Role::class, 'role_user', 'user_id', 'role_id');
```

Para definir la relación inversa de muchos a muchos, se debe definir un método en el modelo relacionado que también devuelva el resultado del método belongsToMany.

Trabajar con relaciones de muchos a muchos requiere la presencia de una tabla intermedia creada mediante una migración con al menos la una primary key y los dos campos de las tablas relacionadas con su correspondiente foreign key creada. Estos cambios también se reflejarán en el modelo correspondiente.

Eloquent proporciona algunas formas muy útiles de interactuar con esta tabla. Por ejemplo, supongamos que nuestro modelo de usuario tiene muchos modelos de roles con los que está relacionado. Después de acceder a esta relación, podemos acceder a la tabla intermedia utilizando el atributo **pivot** en los modelos:

```
use App\Models\User;  
  
$user = User::find(1);  
foreach ($user->roles as $role) {  
    echo $role->pivot->created_at;  
}
```

Hay que tener en cuenta que a cada modelo de rol que recuperamos se le asigna automáticamente un atributo de **pivot**. Este atributo contiene un modelo que representa la tabla intermedia. De forma predeterminada, solo las claves del modelo estarán presentes en el modelo dinámico (pivot). Si la tabla intermedia contiene atributos adicionales, hay que especificarlos al definir la relación:

```
return $this->belongsToMany(Role::class)->withPivot('active', 'created_by');
```

En las relaciones muchos a muchos, tenemos 3 métodos principales para poder administrar la tabla intermedia de la relación:

- **attach()**: Este método se utiliza para crear una nueva relación en la tabla pivote.

```
$post->tags()->attach(1),
```

en el ejemplo anterior, permite establecer la etiqueta con identificador 1 en el post.

- **detach()**: Este método elimina la relación entre los dos modelos, es decir, lo elimina de la tabla pivote, viene siendo el proceso inverso del método attach().
- **sync()**: Este método permite sincronizar el listado de identificadores proporcionados, eliminando de la relación los que no estén presentes en el listado y agregando aquellos que estén presentes en el listado.

A todos los métodos anteriores, se puedes pasar como parámetro un identificador, la instancia del modelo o un array con identificadores o instancias de modelos:

```
$post = Post::find(80)
$tag2 = Tag::find(2)
$tag = Tag::find(1)
$post->tags()->attach($tag)
$tag2->posts()->detach(790)
$tag2->posts()->sync(79)
$tag2->posts()->attach($post)
$tag2->posts()->attach([79,80])
```

CONSULTAS

Una vez que se haya creado un modelo y su tabla de base de datos asociada por las migraciones correspondientes, se estará listo para empezar a recuperar datos de la base de datos. Se puede pensar en cada modelo de Eloquent como un potente generador de consultas que permite consultar con fluidez la tabla de la base de datos asociada. Es posible utilizar el método **table** para iniciar una consulta. Este devuelve una instancia fluida, lo que le permite encadenar más restricciones en la consulta y finalmente, recuperar los resultados de la consulta mediante el método **get**:

```
$users = DB::table('users')->get(); // SELECT * FROM users
return view('user.index', ['users' => $users]);
```

El método **get** devuelve una instancia de Illuminate\Support\Collection que contiene los resultados de la consulta, donde cada resultado es una instancia del objeto stdClass de PHP del modelo correspondiente. Puede acceder al valor de cada columna accediendo como una propiedad del objeto:

```
foreach ($users as $user) {
    echo $user->name;
}
```

Recuperando valores

- **first**: devuelve un único objeto, el primero.
- **firstOrFail**. Si no se encuentra el objeto lanza una excepción que si no está capturada será devuelta al cliente.
- Para recuperar un campo también se puede usar el método **value** con el nombre del campo.
- **find**. Búsqueda de una fila por la clave primaria, generalmente id.
- **pluck**. Devuelve todos los elementos de una columna.
- Existen las funciones agregadas: **max**, **min**, **avg** y **sum**.
- **exists**. Si el campo existe.
- **doesntExist**. Si el campo no existe.

```
$price = DB::table('orders')->max('price');
```

Limitando las búsquedas

Es deseable limitar los resultados que se recogen desde el servidor. Para ello usaremos el método **where** del generador de consultas para añadir cláusulas "where SQL" a la consulta. La llamada más básica al método requiere tres argumentos. El primer el nombre de la columna. El segundo un operador, que puede ser cualquiera de los operadores admitidos en la base de datos. El tercero el valor que se va a comparar con el valor de la columna. Si anidamos llamadas a **where** el operador que se usará para unirlas será **and**.

```
$users = DB::table('users') // Se puede cambiar por el modelo Users::
    ->where('votes', '=', 100)
    ->where('age', '>', 35) ->get();
```

Opcionalmente se puede pasar un Array de condiciones a la función **where**. Cada elemento de la matriz debe ser una matriz que contenga los tres argumentos que normalmente se pasan al método:

```
$users = DB::table('users')->where([
    ['status', '=', '1'],
    ['subscribed', '<>', '1'],
])->get();
```

PDO no admite el enlace de nombres de columna. Por lo tanto, nunca se debe permitir que la entrada del usuario dicte los nombres de columna a los que hacen referencia las consultas, incluidas las columnas "ordenar by". Al encadenar llamadas al método **where** del generador de consultas, las cláusulas "where" se unirán mediante el operador **and**. Sin embargo, puede utilizar el método **orWhere** para unir una cláusula a la consulta mediante el operador **or**. El método **orWhere** acepta los mismos argumentos que el método **where**:

```
$users = Users::where('votes', '>', 100)
    ->orWhere('name', 'John') ->get();
```

Laravel también admite la consulta de tipos de columnas JSON en bases de datos que brindan soporte para este tipo de columnas. Actualmente, esto incluye MariaDB 10.3+, MySQL 8.0+, PostgreSQL 12.0+, SQL Server 2017+ y SQLite 3.39.0+. Para consultar una columna JSON, se usa el operador `->`:

```
$users = Users::where('preferences->dining->meal', 'salad')->get();
```

Resumen de métodos:

- `orWhereNot` (para negar la consulta), `whereAny` (cualquiera de los pasados) o `whereAll` (Or todos).
- `whereLike` / `orWhereLike` / `whereNotLike` / `orWhereNotLike`, para añadir cláusulas LIKE SQL.
- `whereIn` / `whereNotIn` / `orWhereIn` / `orWhereNotIn` para el operador IN SQL.
- `whereBetween` / `orWhereBetween` para el operador Between.
- `whereNull` / `whereNotNull` / `orWhereNull` / `orWhereNotNull` para determinar si un campo es nulo.
- `whereDate` / `whereMonth` / `whereDay` / `whereYear` / `whereTime` para preguntas de tiempo.
- `whereColumn` / `orWhereColumn`, para comparación que dos columnas sean iguales.

```
$users = User::whereBetween('age', [18, 25])->get();
$users = User::whereIn('role', ['admin', 'editor'])->get();
$users = User::whereNull('deleted_at')->get();
```

Ordenaciones

El método `orderBy` permite ordenar los resultados de la consulta por una columna determinada. El primer argumento aceptado por el método debe ser la columna por la que desea ordenar, mientras que el segundo argumento fija la dirección de la ordenación y puede ser `asc` o `desc`:

```
$users = DB::table('users')
    ->orderBy('name', 'desc')
    ->get();
```

Para ordenar por varias columnas, simplemente se puede invocar `orderBy` tantas veces como sea necesario:

```
$users = Users::orderBy('name', 'desc')
    ->orderBy('email', 'asc')
    ->get();
```

Group By

Como cabría esperar, los métodos `groupBy` y `having` se pueden utilizar para agrupar los resultados de la consulta con el mismo significado que en SQL. La signatura del método `having` es similar a la del método `where`:

```
$users = DB::table('users')
    ->groupBy('account_id')
    ->having('account_id', '>', 100)
    ->get();
```

Se puede usar el método `havingBetween` para filtrar los resultados dentro de un intervalo determinado:

```
$report = DB::table('orders')
    ->selectRaw('count(id) as number_of_orders, customer_id')
    ->groupBy('customer_id')
    ->havingBetween('number_of_orders', [5, 15])
    ->get();
```

Limit y Offset

Es posible utilizar los métodos **skip** y **take** para limitar el número de resultados devueltos por la consulta. Para omitir un número determinado de resultados en la consulta usaremos skip. Si es necesario recoger un número de elementos será take el elegido:

```
$users = Users::skip(10)->take(5)->get();
```

Como alternativa, se pueden utilizar los métodos **limit** y **offset**. Estos son funcionalmente equivalentes a los métodos take y skip, respectivamente:

```
$users = Users::offset(10)
    ->limit(5)
    ->get();
```

Si se tiene una instancia de un modelo Eloquent que se recuperó de la base de datos, se puede "actualizar" el modelo mediante los métodos **fresh** y **refresh**. El método fresh devuelve un nuevo modelo de la base de datos con los nuevos datos, la instancia del modelo existente no se verá afectada. Por otro lado, el método refresh volverá a cargar el mismo modelo utilizando datos nuevos de la base de datos. Además, también se actualizarán todas sus relaciones. En resumen, fresh crea un nuevo modelo de uno existente, mientras que refresh lo recarga con los datos actualizados.

Colecciones

Todos los métodos Eloquent que devuelvan más de un resultado del modelo devolverán instancias de la clase Illuminate\Database\Eloquent\Collection, incluidos los resultados recuperados a través del método **get** o a los que se tenga acceso a través de una relación. El objeto colección Eloquent amplía la colección base de Laravel, por lo que hereda docenas de métodos, utilizados para trabajar con fluidez con el Array subyacente. Todas las colecciones sirven como iteradores, lo que permiten recorrerlas en bucle como si fueran simples matrices PHP:

```
use App\Models\User;

$users = User::where('active', 1)->get();
foreach ($users as $user) {
    echo $user->name;
}
```

Sin embargo, como se mencionó anteriormente, las colecciones son mucho más poderosas que los Arrays y exponen una variedad de operaciones de mapeo / reducción que se pueden encadenar. Por ejemplo, podemos eliminar todos los modelos inactivos y, a continuación, recopilar el nombre de cada usuario:

```
$names = User::all()->reject(function (User $user) {
    return $user->active === false;
})->map(function (User $user) {
    return $user->name;
});
```

Métodos disponibles

Todas las colecciones de Eloquent amplían el objeto de colección base de Laravel; Por lo tanto, heredan todos los métodos proporcionados por la clase de colección base. Además, la clase Illuminate\Database\Eloquent\Collection proporciona un superconjunto de métodos para ayudar a administrar las colecciones de modelos. La mayoría de los métodos devuelven instancias de Illuminate\Database\Eloquent\Collection; sin embargo, algunos métodos, como **modelKeys**, devuelven una instancia de Illuminate\Support\Collection. Los métodos disponibles son:

| | | |
|-------------|-----------|-------------|
| append | contains | diff |
| except | find | findOrFail |
| fresh | intersect | load |
| loadMissing | modelKeys | makeVisible |
| makeHidden | only | setVisible |
| setHidden | toQuery | unique |

BORRADO

En Eloquent, creas modelos para ayudar a construir consultas. Sin embargo, a veces necesitas eliminar modelos para que una aplicación sea más eficiente. Para ello, se llama a **delete** en la instancia del modelo.

```
use App\Models\Stock;

$stock = Stock::find(1);
$stock->delete();
```

El código anterior elimina el modelo Stock de una aplicación. Se trata de una eliminación permanente que no puede deshacerse.

Borrado condicional

Por supuesto, se puede crear una consulta Eloquent para eliminar todos los modelos que coincidan con los criterios de búsqueda. En este ejemplo, eliminaremos todos los vuelos que estén marcados como inactivos.

```
$deleted = Flight::where('active', 0)->delete();
```

Eliminación suave (Soft delete)

Otra función que contiene Eloquent es la capacidad de borrado suave de modelos. Cuando borras suavemente un modelo, no lo eliminás de la base de datos, lo marcas como inactivo. Este borrado es contemplado en las consultas de recuperación si está configurado el modelo para ello.

El borrado se marca utilizando una columna **deleted_at** en la tabla correspondiente para indicar la hora y la fecha del borrado suave. Esto es importante cuando quieras excluir una parte de los registros de la base de datos, como los que están incompletos, sin eliminarlos permanentemente. Este tipo de borrado ayuda a limpiar los resultados de las consultas de Eloquent sin añadir condiciones adicionales. Se puede activar la eliminación suave añadiendo el trait **softDeletes** a un modelo y añadiendo una columna **deleted_at** en la tabla de base de datos relacionada.

Añadir el borrado suave a un modelo

Para activar el borrado suave de un modelo, se añade el trait **IlluminateDatabaseEloquentSoftDeletes**, como se muestra a continuación.

```
namespace App\Models;  
use Illuminate\Database\Eloquent\Model;  
use Illuminate\Database\Eloquent\SoftDeletes;  
  
class Flight extends Model {  
    use SoftDeletes;  
}
```

Antes de que se pueda empezar a utilizar el borrado suave, la tabla en la base de datos debe tener una columna **delete_at**. Se añade esta columna utilizando una migración, como se muestra a continuación:

```
use Illuminate\Database\Schema\Blueprint;  
use Illuminate\Support\Facades\Schema;  
...  
Schema::table('users', function (Blueprint $table) {  
    $table->softDeletes();});  
Schema::table('users', function (Blueprint $table) {  
    $table->dropSoftDeletes();});
```

Esto añade una columna **delete_at** que se actualiza con la fecha y la hora, en caso que una acción de borrado suave tenga éxito.

Cómo incluir modelos con borrado suave

Si quieras que los resultados de la consulta incluyan modelos con borrado suave, debes añadir el método **withTrashed()** a la consulta. A continuación, se muestra cómo:

```
$stocks = Stock::withTrashed()->where('stock_id', 20)->get();
```

La consulta anterior también incluirá los modelos con el atributo **deleted_at**, es decir, recuperará los borrados y los no borrados, para recoger los no borrados solo, se siguen utilizando los métodos tradicionales.

Cómo recuperar únicamente modelos con borrado suave

Eloquent también permite recuperar exclusivamente modelos no activos. Se puede hacer llamando al método **onlyTrashed()**, por ejemplo:

```
$Stock = Stock::onlyTrashed()->where('stock_id', 1)->get();
```

Cómo restaurar modelos con borrado suave

Podríamos restaurar los modelos borrados de esta manera llamando al método **restore()**.

```
$stocks = Stock::withTrashed()->where('stock_id', 20)->restore();
```

Esto cambia el campo **delete_at** a null. Si el modelo no ha sido borrado suavemente, deja el campo sin cambios.

INSERCIÓNES Y ACTUALIZACIONES

Al usar Eloquent, no solo necesitamos recuperar modelos de la base de datos, también necesitamos insertar nuevos registros. Para insertar un nuevo registro en la base de datos, se creará una instancia de un nuevo modelo y establecer los atributos en el modelo. A continuación, se llama al método **save** en la instancia del modelo.

El método **save** también se puede utilizar para actualizar los modelos que ya existen en la base de datos. Para actualizar un modelo, se tiene que recuperar y establecer los atributos que se deseen actualizar. A continuación, hay que llamar al método **save** del modelo. De nuevo, la marca de tiempo `updated_at` se actualizará automáticamente, por lo que no es necesario establecer manualmente su valor.

En ocasiones, es posible que haya que actualizar un modelo existente o crear uno nuevo si no existe ningún modelo coincidente. Para ese fin existen los métodos **firstOrCreate** y **updateOrCreate** en los que no es necesario llamar manualmente al método `save`.

También se puede utilizar el método **create** para "guardar" un nuevo modelo utilizando una sola declaración PHP. La instancia de modelo insertada se le devolverá mediante el método:

```
use App\Models\Flight;  
$flight = Flight::create([  
    'name' => 'London to Paris',  
]);
```

Sin embargo, antes de utilizar el método **create**, se deberá especificar una propiedad **\$fillable** protegida en la clase de modelo. Esta propiedad es necesaria porque todos los modelos de Eloquent están protegidos contra vulnerabilidades de asignación masiva de forma predeterminada.

El generador de consultas también proporciona un método que se puede utilizar para insertar registros en la tabla de la base de datos. El método **insert** acepta una matriz de nombres y valores de columna:

```
DB::table('users')->insert([  
    'email' => 'kayla@example.com',  
    'votes' => 0  
]);
```

Es posible insertar varios registros a la vez pasando una matriz de matrices. Cada Array representa un registro que se debe añadir a la tabla:

```
DB::table('users')->insert([
    ['email' => 'picard@example.com', 'votes' => 0],
    ['email' => 'janeway@example.com', 'votes' => 0],
]);
```

El método **upsert** insertará registros que no existen o actualizará los registros que ya existen con nuevos valores que se pueden especificar, en una única operación. Además de insertar registros en la base de datos, el generador de consultas también puede actualizar los registros existentes mediante el método de actualización.

El método **update**, al igual que el método `insert`, acepta un Array de pares de columna - valor que indican las columnas que se van a actualizar. El método **update** devuelve el número de filas afectadas. Es posible restringir la consulta de actualización mediante cláusulas **where**:

```
$affected = DB::table('users')
    ->where('id', 1)
    ->update(['votes' => 1]);
```

Al actualizar una columna JSON, se tiene que usar la sintaxis `->` para actualizar la clave adecuada en el objeto JSON. Esta operación es compatible con MariaDB 10.3+, MySQL 5.7+ y PostgreSQL 9.5+:

```
$affected = DB::table('users')
    ->where('id', 1)
    ->update(['options->enabled' => true]);
```

El generador de consultas también proporciona métodos para aumentar o disminuir el valor de una columna determinada. Ambos métodos aceptan al menos un argumento: la columna que se va a modificar. Se puede proporcionar un segundo argumento para especificar la cantidad en la que se debe incrementar o disminuir la columna:

```
DB::table('users')->increment('votes');
DB::table('users')->increment('votes', 5);
DB::table('users')->decrement('votes');
DB::table('users')->decrement('votes', 5);
```

SERIALIZACIÓN

Serialización a Arrays

Para convertir un modelo y sus relaciones cargadas en una matriz, podríamos usar el método **toArray**. Este método es recursivo, por lo que todos los atributos y todas las relaciones (incluidas las relaciones de relaciones) se convertirán en matrices:

```
use App\Models\User;

$user = User::with('roles')->first();
return $user->toArray();
```

El método **attributesToArray** se puede usar para convertir los atributos de un modelo en una matriz, pero no sus relaciones:

```
$user = User::first();
return $user->attributesToArray();
```

También se puede convertir colecciones completas de modelos en Arrays llamando al método **toArray** en la instancia de la colección:

```
$users = User::all();
return $users->toArray();
```

Serialización a JSON

Para convertir un modelo en JSON, existe el método **toJson**. Al igual que toArray, el método toJson es recursivo, por lo que todos los atributos y relaciones se convertirán en JSON. También puede especificar cualquier opción de codificación JSON que sea compatible con PHP:

```
use App\Models\User;

$user = User::find(1);
return $user->toJson();
return $user->toJson(JSON_PRETTY_PRINT);
```

Como alternativa, es posible convertir un modelo o una colección en una cadena, que llamará automáticamente al método **toJson** en el modelo o la colección de la siguiente manera:

```
return (string) User::find(1);
```

Dado que los modelos y las colecciones se transoforman a JSON cuando se convierten en una cadena, puede devolver objetos Eloquent directamente desde las rutas o los controladores de la aplicación. Laravel serializará automáticamente sus modelos y colecciones a JSON cuando se devuelvan:

```
Route::get('/users', function () {
    return User::all();
});
```

Cuando un modelo Eloquent se convierte en JSON, sus relaciones cargadas se incluirán automáticamente como atributos en el objeto JSON. Además, aunque los métodos de relación de Eloquent se definen utilizando nombres de método de "camelCase", el atributo JSON de una relación se transformará en "sanke_case".

PAGINACIÓN

Laravel proporciona un sistema de paginación robusto y fácil de usar. Cuando se combina con Bootstrap, puede crear listas paginadas limpias y con capacidad de respuesta.

Configuración de la paginación

La paginación de Laravel es sencilla. Se pueden paginar los resultados del generador de consultas o las colecciones de Eloquent.

```
<?php  
// Eloquent  
$posts = Post::paginate(15);  
// Query Builder  
$users = DB::table('users')->paginate(10);
```

El método **paginate** toma como argumento el número de elementos por página.

Uso de la paginación

Para usar los estilos de paginación de Bootstrap, se tiene que decir a Laravel que use la vista de Bootstrap.

```
<?php  
use Illuminate\Pagination\Paginator;  
  
Paginator::useBootstrap();
```

Normalmente, se colocará en el método de arranque (**boot**) de **AppServiceProvider** en la ruta app/Providers.

Imprimiendo los enlaces

En la plantilla, se puede mostrar los vínculos de paginación mediante el método **links** de la colección:

```
{{ $posts->links() }}  
This will generate the HTML for the pagination links using Bootstrap's CSS classes.
```

```
<?php  
// Controller  
public function index() {  
    $posts = Post::paginate(10);  
    return view('posts.index', compact('posts'));  
}
```

```
<h1>Posts</h1>  
<ul>  
    @foreach ($posts as $post)  
        <li>{{ $post->title }}</li>  
    @endforeach  
</ul>  
&nbsp;{{ $posts->links() }}
```

Personalización de los enlaces

Es posible personalizar los vínculos de paginación pasando opciones adicionales al método **links**.

```
 {{ $posts->links(['class' => 'pagination-sm']) }}
```

Esto renderizará los enlaces de paginación con la clase pagination-sm de Bootstrap.

Consejos adicionales

- **Personalización de la vista de paginación:** si se necesita más control sobre los enlaces de paginación, puede crear una vista personalizada.
- **Anexar parámetros de consulta:** se puede anexar parámetros de consulta adicionales a los vínculos de paginación mediante el método **appends** de la colección paginada.
- **Longitud de paginación:** puede personalizar el número de vínculos de página que se muestran mediante el método **lengthAwarePaginator**.

Controladores y Modelos

Introducción

El patrón MVC permite crear aplicaciones estructuradas, mantenibles y modificables a lo largo del tiempo. Hasta ahora solo hemos visto la parte de Vista a través de las plantillas Blade. Es hora de abordar los otros dos elementos, los modelos y los controladores.

Los controladores son elementos que van a gestionar las peticiones que vienen desde el cliente, desarrollar el trabajo necesario usando los modelos y devolviendo la vista correspondiente. Estos controladores se pueden implementar de varias formas: funciones, clases o componentes. En esta parte nos vamos a dedicar a crear clases.

Por otro lado, el modelo es una clase que se utiliza para manejar los datos de la base de datos. Si bien las migraciones modificaban la estructura, los modelos permitirán modificar y acceder a los datos de dicha estructura. Podemos crear modelos y controladores de forma independiente, pero es más cómodo hacerlo simultáneamente y siguiendo las normas de Laravel.

```
php artisan make:controller -r Dashboard/PostController -m Post
```

El comando anterior crea un controlador llamado **PostController** dentro del directorio `http/Controllers/Dashboard`. A la vez crea el modelo correspondiente, llamado **Post**, dentro del directorio `app/Models`. La nomenclatura de dichos archivos hará referencia a una table de la BBDD llamada Posts.

Generalmente, dentro del controlador haremos uso del Modelo de la forma **Post::** usando los métodos que hemos visto dentro de la sección ORM anterior.

Modelos

El modelo se utiliza para evitar la comunicación directa con la base de datos, debe definir al menos las columnas que maneja de la tabla y las relaciones de la misma con otras tablas. Para las columnas, en el caso base se establecerán todas las columnas menos aquellas autogeneradas, tales como autoincrementales o calculadas, pero no es obligatorio, siendo posible no añadir al modelo todos los campos de la correspondiente tabla. Para hacerlo nos iremos al fichero del modelo creado y añadiremos una variable protegida llamada **\$fillable**, cuyo contenido será un array con los nombres de los campos que queremos gestionar.

Como hemos mencionado arriba, es necesario incluir en el modelo las relaciones existentes. Para ello crearemos un método dentro del modelo para la relación, si esta es en ambas direcciones lo definiremos en ambos modelos como, por ejemplo:

```
9 class Post extends Model
10
11     public function categorie():BelongsTo{
12         return $this->belongsTo( related: Categorie::class);
13     }
14 }
```

```
class Categorie extends Model
{
    no usages
    protected $fillable = ['categorie', 'slug', 'created_at', 'updated_at'];
    public function posts()
    {
        return $this->hasMany( related: Post::class);
    }
}
```

En el que creamos las relaciones entre los **Post** y las **Categories** en ambas direcciones. Para la descripción de todas las relaciones ver el punto ORM anterior.

Muy importante

```
public function up(): void
{
    Schema::table( table: 'posts', function (Blueprint $table) {
        $table->foreignId( column: 'categorie_id')->nullable()->after( column: 'user_id')->constrained( table: 'categories')->onDelete( action: 'cascade');
    });
}
```

Las relaciones creadas en el modelo deben haberse reflejado anteriormente en la tabla con un campo nuevo a través de una migración, que añadiremos a la variable **\$fillable** del modelo correspondiente, en este caso Posts.

CONVENCIONES

Como hemos hablado anteriormente, hay que seguir unas convenciones a la hora de nombrar los campos clave, y los nombres de las tablas y modelos. Cuando se crea un modelo, este se sitúa dentro del directorio app/Models. Si el nombre lleva una ruta anterior, se creará dentro de dicha ruta y deberá usarse para utilizarlo. El nombre del modelo debe seguir la convención CamelCase y la tabla snake_case en plural. Así el modelo llamado **Flight** almacenará datos en la tabla **flights**, mientras que el modelo **AirTrafficController** lo hará en una tabla llamada **air_traffic_controllers**.

```
protected $fillable = [
    'title',
    'slug',
    'content',
    'visits',
    'published_at',
    'user_id',
    'created_at',
    'updated_at',
    'categorie_id',
];
];
```

Existe otra convención que tenemos que tener en cuenta, Larabel asume que toda tabla tiene una clave primaria llamada id (\$table->id()), pero es posible cambiar el nombre de la misma, si en la migración hemos definido un campo, mediante una variable protegida en el modelo de la siguiente forma **protected \$primaryKey = 'flight_id';**

Eloquent espera que existan las columnas **created_at** y **updated_at** en la tabla correspondiente de su modelo. El ORM establecerá automáticamente los valores de estas columnas cuando se creen o actualicen los modelos. Si no se desea que se administre automáticamente estas columnas, hay que definir una propiedad pública **\$timestamps** en el modelo con un valor de **false**.

Controladores

Hemos visto que el controlador recoge la petición del cliente, usa el modelo y general la respuesta al cliente. En este flujo, el controlador puede crear tres tipos de resultado. Una respuesta, una respuesta con redirección y una vista.

- **Response.** Devuelve tanto una vista como valores JSON.
- **RedirectResponse.** En este caso es una redirección a otra página, se suele usar para operaciones CRUD en las que no se devuelven datos.
- **View.** Se devuelve una vista.
- **Datos de un modelo.** Adicionalmente se puede devolver directamente un modelo dentro de un método o una lista de ellos, automáticamente se transformarán en datos JSON y serán enviados al cliente, en este caso el método no tiene que tener nada configurado en el método como tipo de retorno.

Además de utilizar la función correcta dentro del método del controlador, este debe estar definido adecuadamente a la devolución que hace, si no lo está, Laravel intentará inferirlo de la llamada de la función.

```
public function index():View
{
    return view( view: 'welcome');
}
```

La orden dd

Esta orden se utiliza en depuración para devolver código más explicativo que solo el resultado. Podemos cambiar lo retornado por cualquier método de un controlador y ponerlo dentro de esta orden para dar mucha más información.

```
Illuminate\Database\Eloquent\Collection {#273 ▼ // app\models\Post
  #items: array:2 [▼
    0 => App\Models\Post {#1241 ▶}
    1 => App\Models\Post {#1024 ▶}
  ]
  #escapeWhenCastingToString: false
}
```

Ilustración 17 - La orden dd

Las flechas son funcionales, extienden y contraen el contenido mostrado.

Se recomienda ver el punto de rutas para recordar la creación y uso junto con las rutas.

Práctica Laravel 7

Autenticación bajo Laravel

<https://laravel.com/docs/12.x/authentication>

Dentro de las consideraciones a tener en cuenta se encuentra que, si se va a acceder a la aplicación utilizando un navegador y está creando una aplicación Laravel monolítica, su aplicación utilizará los servicios de autenticación integrados de Laravel.

Si su aplicación ofrece una API que será consumida por terceros, tendremos que elegir configurar entre Passport o Sanctum para proporcionar autenticación de token de API. En general, es preferible Sanctum cuando sea posible, ya que es una solución simple y completa para la autenticación de API, la autenticación SPA y la autenticación móvil. Si se está creando una aplicación de una sola página (SPA) que funcionará con un backend de Laravel, debe usar Laravel Sanctum. Para desarrollar un sistema de autorización propio se visitará <https://laravel.com/docs/12.x/authorization>.

Muchas aplicaciones web proporcionan una forma para que sus usuarios se autentiquen con la aplicación e "inician sesión". Implementar esta función en aplicaciones web puede ser una tarea compleja y potencialmente arriesgada. Por esta razón, Laravel se esfuerza por brindar las herramientas que se necesitan para implementarla de forma rápida, segura y sencilla.

En esencia, las instalaciones de autenticación de Laravel están formadas por "guardias" y "proveedores". Los guardias definen cómo se autentican los usuarios para cada solicitud. Por ejemplo, Laravel se envía con una protección de sesión que mantiene el estado mediante el almacenamiento de sesión y las cookies.

Los proveedores definen cómo se recuperan los usuarios del almacenamiento persistente. Laravel incluye soporte para recuperar usuarios que utilizan Eloquent y el generador de consultas de base de datos. Sin embargo, puede definir proveedores adicionales según sea necesario para su aplicación.

El archivo de configuración de autenticación de la aplicación se encuentra en config/auth.php. Este archivo contiene varias opciones bien documentadas para ajustar el comportamiento de los servicios de autenticación de Laravel.

Starter Kits

Para empezar por algo rápido, se puede instalar un kit de inicio en una nueva aplicación Laravel. Después de migrar la base de datos, se navega por el explorador hasta `/register` o cualquier otra dirección URL asignada a la aplicación. Los kits de inicio se encargarán de generar todo su sistema de autenticación.

Autentificación básica bajo HTTP

La autenticación básica HTTP proporciona una forma rápida de autenticar a los usuarios de su aplicación sin configurar una página de "inicio de sesión" dedicada. Para empezar, adjunte el middleware auth.basic a una ruta. El middleware auth.basic está incluido con el framework Laravel, por lo que no es necesario definirlo:

```
Route::get('/profile', function () {  
    // Only authenticated users may access this route...  
})->middleware('auth.basic');
```

Una vez que el middleware se haya adjuntado a la ruta, se le solicitarán automáticamente las credenciales al acceder a la ruta en su navegador. De forma predeterminada, el middleware auth.basic asumirá que la columna de correo electrónico en la tabla de la base de datos de usuarios es el "nombre de usuario" del usuario.

Mediante este modelo, no podemos hacer un logout ya que el explorador mantiene las credenciales HTTP y las manda cuando lo cree necesario, solo funciona cerrando el navegador y recargando sin caché.

Recordar: En las plantillas se puede usar `@auth` para gestionar las partes accesibles de las que no.

Práctica Laravel 8

Autentificación manual

No es necesario que utilice el andamiaje de autenticación incluido con los kits de inicio de aplicaciones de Laravel. Si se decide no utilizar este andamiaje, deberá administrar la autenticación de usuario utilizando directamente las clases de autenticación. Accederemos a dichos servicios a través de la facade de `Auth`, por lo que tendremos que asegurarnos de importarlo en la parte superior de la clase. El método `attempt` se utiliza normalmente para gestionar los intentos de autenticación desde el formulario de "inicio de sesión". Si la autenticación se realiza correctamente, se debe volver a generar la sesión del usuario para evitar problemas de la sesión:

```
<?php  
namespace App\Http\Controllers;  
  
use Illuminate\Http\Request;  
use Illuminate\Http\RedirectResponse;  
use Illuminate\Support\Facades\Auth;  
  
class LoginController extends Controller  
{
```

```

/**
 * Handle an authentication attempt.
 */
public function authenticate(Request $request): RedirectResponse
{
    $credentials = $request->validate([
        'email' => ['required', 'email'],
        'password' => ['required'],
    ]);

    if (Auth::attempt($credentials)) {
        $request->session()->regenerate();

        return redirect()->intended('dashboard');
    }

    return back()->withErrors([
        'email' => 'The provided credentials do not match our records.',
    ])->onlyInput('email');
}
}

```

El método `attempt` acepta una matriz de pares clave/valor como primer argumento. Los valores de la matriz se utilizarán para encontrar al usuario en la tabla de la base de datos. Por lo tanto, en el ejemplo anterior, el usuario será recuperado por el valor de la columna de correo electrónico. Si se encuentra el usuario, la contraseña con hash almacenada en la base de datos se comparará con el valor de la contraseña pasada al método a través de la matriz. **No hay que aplicar un hash al valor de la contraseña de la solicitud entrante, ya que el marco lo hará automáticamente antes de compararlo con la contraseña en la base de datos.** Se iniciará una sesión autenticada para el usuario si las dos contraseñas hash coinciden.

Hay que recordar que los servicios de autenticación de Laravel recuperarán usuarios de su base de datos en función de la configuración del "proveedor" de autenticación. En el archivo de configuración `config/auth.php` predeterminado, se especifica que el proveedor de usuarios por defecto es Eloquent y se le indica que use el modelo `App\Models\User` para la gestión de la contraseña. Puede cambiar estos valores dentro de su archivo de configuración en función de las necesidades de su aplicación.

```

'providers' => [
    'users' => [
        'driver' => 'eloquent',
        'model' => env( key: 'AUTH_MODEL' , default: App\Models\User::class),
    ],
]

```

El método **attempt** devolverá true si la autenticación se realizó correctamente. De lo contrario, devolverá false.

El método proporcionado por el redirector de Laravel redirigirá al usuario a la URL a la que intentaba acceder antes de ser interceptado por el middleware de autenticación. Se puede proporcionar una URI más a este método para usarla en caso de que el destino previsto no esté disponible.

ESPECIFICANDO CONDICIONALES ADICIONALES

Si lo desea, también puede agregar condiciones de consulta adicionales a la consulta de autenticación además del correo electrónico y la contraseña del usuario. Para lograr esto, simplemente podemos agregar las condiciones de consulta a la matriz pasada al método attempt. Por ejemplo, podemos verificar que el usuario está marcado como "activo":

```
if (Auth::attempt(['email' => $email, 'password' => $password, 'active' => 1])) {  
    // Authentication was successful...  
}
```

Para casos de condiciones de consulta complejas, se puede proporcionar un closure en la matriz de credenciales. Esta se invocará con la instancia de consulta, lo que le permitirá personalizar la consulta en función de las necesidades de su aplicación:

```
use Illuminate\Database\Eloquent\Builder;  
  
if (Auth::attempt([  
    'email' => $email,  
    'password' => $password,  
    fn (Builder $query) => $query->has('activeSubscription'),  
])) {  
    // Authentication was successful...  
}
```

En estos ejemplos, el correo electrónico no es una opción obligatoria, simplemente se utiliza como ejemplo. Debe utilizar cualquier nombre de columna que corresponda a un "nombre de usuario" en la tabla de la base de datos.

El método **attemptWhen**, que recibe una closure como segundo argumento, se puede usar para realizar una inspección más exhaustiva del usuario potencial antes de autenticarlo. La closure recibirá al usuario potencial y debe devolver verdadero o falso para indicar si el usuario puede ser autenticado:

```
if (Auth::attemptWhen([  
    'email' => $email,  
    'password' => $password,  
], function (User $user) {  
    return $user->isNotBanned();  
})) {  
    // Authentication was successful... }
```

RECORDANDO USUARIOS

Muchas aplicaciones web proporcionan una casilla de verificación "recuérdame" en su formulario de inicio de sesión. Si desea proporcionar la funcionalidad "recuérdame" en la aplicación, puede pasar un valor booleano como segundo argumento al método **attempt**.

Cuando este valor es verdadero, Laravel mantendrá al usuario autenticado indefinidamente o hasta que cierre la sesión manualmente. La tabla de usuarios debe incluir el campo **remember_token** en la tabla, que se usará para almacenar el token "recordarme". La migración de la tabla de usuarios incluida con las nuevas aplicaciones de Laravel ya incluye esta columna y no hay que modificar nada:

```
use Illuminate\Support\Facades\Auth;

if (Auth::attempt(['email' => $email, 'password' => $password], $remember)) {
    // The user is being remembered...
}
```

Si la aplicación ofrece la funcionalidad "recordarme", se puede usar el método **viaRemember** para determinar si el usuario autenticado actualmente se autenticó mediante la cookie "recuérdame":

```
use Illuminate\Support\Facades\Auth;

if (Auth::viaRemember()) {
    // ...
}
```

LogOut

Para cerrar manualmente la sesión de los usuarios en la aplicación, se utiliza el método de cierre de sesión proporcionado por la facade Auth. Esto eliminará la información de autenticación de la sesión del usuario para que las solicitudes posteriores no se autentiquen.

Además de llamar al método de cierre de sesión, se recomienda invalidar la sesión del usuario y volver a generar su token CSRF. Después de cerrar la sesión del usuario, normalmente se redirigirá al usuario a la raíz de la aplicación:

```
use Illuminate\Http\Request;
use Illuminate\Http\RedirectResponse;
use Illuminate\Support\Facades\Auth;

/**
 * Log the user out of the application.
 */
public function logout(Request $request): RedirectResponse
{
    Auth::logout();
```

```
$request->session()->invalidate();  
  
$request->session()->regenerateToken();  
  
return redirect('/');  
}
```

Práctica Laravel 9

Confirmación de clave

Es posible que ocasionalmente tengamos acciones que requieran que el usuario confirme su contraseña antes de que se realice la acción o antes de que el usuario sea redirigido a un área confidencial de la aplicación. Laravel incluye middleware incorporado para que este proceso sea muy sencillo. La implementación de esta característica requerirá que defina dos rutas: una ruta para mostrar una vista que le pida al usuario que confirme su contraseña y otra ruta para confirmar que la contraseña es válida y redirigir al usuario a su destino previsto.

CONFIGURACIÓN

Después de confirmar su contraseña, no se le pedirá al usuario que vuelva a confirmar su contraseña durante tres horas. Sin embargo, puede configurar el período de tiempo antes de que se le vuelva a solicitar al usuario su contraseña cambiando el valor del valor de configuración de `password_timeout` dentro del archivo de configuración de configuración/auth.php de la aplicación.

ENRUTADO

El formulario de confirmación

En primer lugar, definiremos una ruta para mostrar una vista que solicite al usuario que confirme su contraseña:

```
Route::get('/confirm-password', function () {  
    return view('auth.confirm-password');  
})->middleware('auth')->name('password.confirm');
```

Como es de esperar, la vista que devuelve esta ruta debe tener un formulario que contenga un campo de contraseña. Además, no dude en incluir texto dentro de la vista que explique que el usuario está entrando en un área protegida de la aplicación y debe confirmar su contraseña.

Confirmación de la clave

A continuación, definiremos una ruta que gestionará la solicitud del formulario desde la vista de "confirmar contraseña". Esta ruta se encargará de validar la contraseña y redirigir al usuario a su destino previsto:

```
use Illuminate\Http\Request;  
use Illuminate\Support\Facades\Hash;
```

```
use Illuminate\Support\Facades\Redirect;

Route::post('/confirm-password', function (Request $request) {
    if (! Hash::check($request->password, $request->user()->password)) {
        return back()->withErrors([
            'password' => ['The provided password does not match our records.']
        ]);
    }

    $request->session()->passwordConfirmed();

    return redirect()->intended();
})->middleware(['auth', 'throttle:6,1']);
```

Antes de continuar, examinemos esta ruta con más detalle. En primer lugar, se determina que el campo de contraseña de la solicitud coincida realmente con la contraseña del usuario autenticado. Si la contraseña es válida, debemos informar a la sesión de Laravel de que el usuario ha confirmado su contraseña. El método `passwordConfirmed` establecerá una marca de tiempo en la sesión del usuario que Laravel puede usar para determinar cuándo fue la última vez que el usuario confirmó su contraseña. Por último, podemos redirigir al usuario a su destino previsto.

Protegiendo rutas

Debe asegurarse que a cualquier ruta que realice una acción que requiera la confirmación de una contraseña reciente se le asigne el middleware `password.confirm`. Este middleware se incluye con la instalación predeterminada de Laravel y almacenará automáticamente el destino previsto del usuario en la sesión para que el usuario pueda ser redirigido a esa ubicación después de confirmar su contraseña. Después de almacenar el destino previsto del usuario en la sesión, el middleware redirigirá al usuario a la ruta con nombre `password.confirm`:

```
Route::get('/settings', function () {
    // ...
})->middleware(['password.confirm']);

Route::post('/settings', function () {
    // ...
})->middleware(['password.confirm']);
```

GraphQL bajo Laravel

Existen varios Frameworks creados para desarrollar una API GraphQL bajo Laravel, nosotros nos hemos decantado por Lighthouse.

```
https://lighthouse-php.com/
```

Instalación vía Composer

La instalación más sencilla es haciendo uso Composer, como la mayoría de los paquetes bajo php.

```
composer require nuwave/lighthouse
```

Publicación del esquema por defecto

Lighthouse incluye un esquema predeterminado para ponernos en marcha de inmediato. Se publica con el siguiente comando **artisan**, el archivo creado se sitúa en el directorio raíz, carpeta **graphql**, con nombre **schema.graphql**, este archivo lo podremos modificar como necesitemos.

```
php artisan vendor:publish --tag=lighthouse-schema
```

Configuración

Para usar la configuración de lighthouse incluida, hay que copiarla en la carpeta de configuración.

```
copy vendor/nuwave/lighthouse/src/lighthouse.php config/
```

Un API de GraphQL se puede consumir desde varios clientes, que pueden o no residir en el mismo dominio que su servidor. Hay que asegurarse de habilitar el uso compartido de recursos de origen cruzado (CORS) para su punto de conexión de GraphQL en config/cors.php si es necesario, pero antes habrá que configurar Laravel para CORS (conf/cors.php), en nuestro entorno de desarrollo no es necesario.

```
return [  
    'paths' => ['api/*', 'graphql'],  
    ...];
```

Instalación de las herramientas GraphQL

GraphQL tiene herramientas de gestión muy potentes, para hacer uso de ellas hay que instalar GraphiQL.

```
composer require mll-lab/laravel-graphiql
```

Después de la instalación, accederemos a la url **/graphiql** para acceder a la herramienta. Es posible usar cualquier cliente de GraphQL con Lighthouse, solo hay que asegurarse de apuntar a la URL definida en la configuración. De forma predeterminada, el punto de conexión se encuentra en **/graphql**.

Ampliar el ejemplo

Añadimos una mutación a esquema en **/graphql/schema.graphql**

```
type Mutation {  
    createUser(name: String!, email: String!, password:String!): User  
}
```

Definimos un resolutor para dicha mutación en **/app/GraphQL/Mutations/CreateUser.php**, el archivo no existe, habrá que crearlo y debe coincidir con el nombre de la mutación. Para acceder a los parámetros que vengan desde la mutación será a través del segundo parámetro del método `__invoke`, este array asociativo tendrá por claves los nombres de parámetros definidos en la mutación (`$args`).

```
<?php
namespace App\GraphQL\Mutations;

use App\Models\User;
use Illuminate\Support\Facades\Auth;

class CreateUser
{
    public function __invoke($rootValue, array $args)
    {
        $user = User::create([
            'name' => $args['name'],
            'email' => $args['email'],
            'password' => bcrypt($args['password']),
        ]);

        return $user;
    }
}
```

Ya podemos hacer uso del API, desde <http://localhost:8000/graphiql> podemos hacer pruebas.

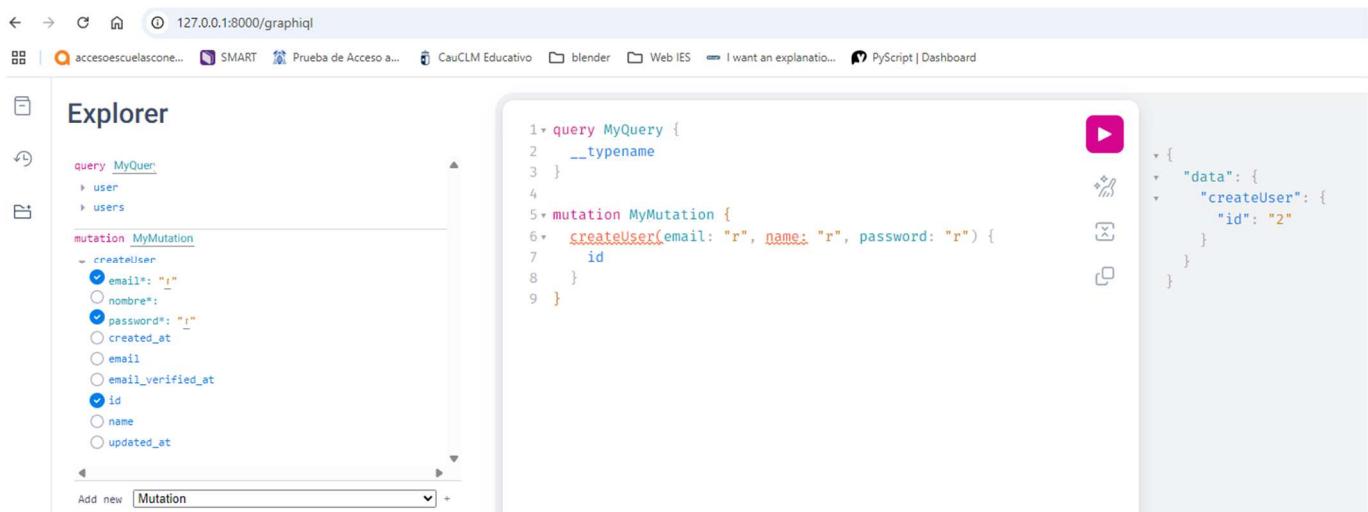


Ilustración 18 - Herramienta Grapiql

Directivas aplicables a un esquema bajo Lighthouse

- @create: Gestiona la creación de una nueva instancia de modelo.
- @update: Gestiona la actualización de una instancia de modelo existente.
- @delete: Gestiona la eliminación de una instancia de modelo.
- @upsert: controla la creación o actualización de una instancia de modelo en función de las condiciones proporcionadas.
- @find: Recupera un solo modelo por su clave principal.
- @all: Obtiene una lista de todas las instancias del modelo.
- @paginate: Devuelve una lista paginada de instancias de modelo.
- @belongsToMany: resuelve un campo como una conexión a una relación de varios a varios.
- @hasOne: Resuelve un campo como una conexión a una relación uno a uno.
- @hasMany: resuelve un campo como una conexión a una relación de uno a varios.
- @belongsTo: Resuelve un campo como una conexión a una relación inversa de uno a uno o de varios a uno.
- @search: Agrega capacidades de búsqueda de texto completo a un campo.
- @orderBy: Ordena una lista de resultados por uno o varios campos.
- @count: Devuelve el recuento de modelos relacionados.
- @groupCount: Devuelve el recuento de modelos relacionados, agrupados por un campo determinado.
- @date: Analiza y da formato a los campos de fecha.
- @rename: Cambia el nombre de un campo en el esquema.
- @guard: Protege las rutas comprobando la autenticación del usuario.
- @can: Comprueba la autorización del usuario mediante las políticas integradas de Laravel.
- @deprecated: Marca un campo o tipo como en desuso.
- @broadcast: Transmite suscripciones a través del sistema de transmisión de Laravel.
- @rules: Aplica reglas de validación a los valores de entrada.
- @rulesApply: Aplica reglas de validación a una lista de valores de entrada.
- @trim: Recorta la entrada de cadena.
- @spread: Extiende los valores de entrada al argumento o campo circundante.
- @eq: Filtra una lista de modelos comprobando la igualdad de un campo.
- @neq: Filtra una lista de modelos comprobando la falta de igualdad de un campo.
- @json: Maneja las columnas JSON de una base de datos.
- @paginate: Página los resultados.

Seguridad basada en tokens

Al igual que con un API REST, para dar seguridad a este API tendremos que hacer uso de las facilidades del paquete Sanctum para gestionar la autentificación y la autorización basada en tokens.

<https://laravel.com/docs/12.x/sanctum>

Desplegar a producción

Cuando la aplicación está terminada hay que desplegarla a producción, en este momento configuraremos toda la aplicación con la instrucción siguiente.

```
npm run build
```

Configuraciones en ficheros

base.blade.php

Para que los recursos funcionen correctamente, sería deseable incluir algo similar a siguiente código en la plantilla base.

```
@if (env('APP_ENV') == 'local')
    @vite(['resources/css/app.css', 'resources/js/app.js'])
@else
    @if (file_exists(public_path('build/manifest.json')))
        @php
            $str = file_get_contents(public_path('build/manifest.json'));
            $manifest = json_decode($str, true);
        @endphp
        <script type="module"
            src="/build/{{ $manifest['resources/js/app.js']['file'] }}"></script>
        <link href="/build/{{ $manifest['resources/sass/app.scss']['file'] }}"
            rel="stylesheet" />
    @endif
@endif
```

Otras consideraciones

- Debemos de publicar todo el proyecto salvo la carpeta de node_modules.
- Se debe evitar en la mayoría de lo posible emplear las variables de entorno en el fichero .env, en caso de que sea imprescindible, se puede emplearlo pero, se simplificará lo más posible y siempre cambiar APP_ENV=production.

Configurar el servidor web Apache

En el fichero .htaccess del directorio raíz del proyecto y al que debe apuntar el DocumentRoot correspondiente, establecerá la siguiente redirección, que hace que toda entrada sea llevada al directorio **public**.

```
<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteRule ^(.*)$ public/$1 [L]
</IfModule>
```

Capítulo VIII. Proyecto Laravel - Bootstrap: Web de Reservas de Turismo Espacial: Estrella Viajera

Tiempo desarrollo real 12 horas, estimado en clase 25 horas guiadas.

Requisitos Funcionales -

1. Gestión de usuarios

- RF1.1 – El sistema debe permitir el registro de nuevos usuarios.
- RF1.2 – El sistema debe permitir el inicio de sesión con correo y contraseña.
- RF1.3 – El sistema debe permitir a los usuarios modificar sus datos personales.

2. Exploración de viajes espaciales

- RF2.1 – El usuario debe poder ver un listado de destinos espaciales disponibles.
- RF2.2 – El usuario debe poder filtrar destinos por tipo (orbital, lunar, marciano, etc.) o nombre.
- RF2.3 – El usuario debe poder acceder al detalle de cada destino, incluyendo: Nombre, descripción, duración, precio.

3. Reservas

- RF3.1 – El sistema debe permitir al usuario realizar una reserva de un destino en una fecha concreta en tal caso se disminuirán las plazas.
- RF3.2 – El sistema debe verificar la disponibilidad de plazas antes de confirmar la reserva.
- RF3.3 – El usuario debe poder consultar sus reservas activas.
- RF3.4 – El usuario debe poder cancelar una reserva activa, si se cancela se aumentarán las plazas y podrá finalizar una, en este caso no aumentará plazas.
- RF3.5 – El sistema debe permitir valorar los destinos al finalizarla.

4. Panel de administración

- RF4.1 – El administrador debe poder gestionar destinos (crear, editar, eliminar).
- RF4.2 – El administrador debe poder ver y cancelar las reservas de los usuarios, se establecerán las plazas como corresponda.
- RF4.3 – El administrador debe poder ver estadísticas básicas: reservas por destino, ingresos, valoraciones.

Requisitos No Funcionales (RNF)

1. Usabilidad

- RNF1.1 – La interfaz debe ser intuitiva y fácil de usar para cualquier tipo de usuario.
- RNF1.2 – La web debe ser responsive, adaptándose a distintos tamaños de pantalla (ordenador, móvil, tablet).
- RNF1.3 – La web debe contar con una navegación clara (menú accesible, rutas bien definidas).
- RNF1.4 – Se debe usar un diseño coherente y atractivo, relacionado con la temática espacial.

2. Rendimiento

- RNF2.1 – El sistema debe cargar en un tiempo inferior a 3 segundos en condiciones normales.
- RNF2.2 – Las consultas a la base de datos deben estar optimizadas para evitar tiempos de respuesta largos.
- RNF2.3 – Las imágenes deben estar optimizadas para no afectar al rendimiento.

3. Seguridad

- RNF3.1 – Las contraseñas de los usuarios deben ser almacenadas cifradas (por ejemplo, con bcrypt).
- RNF3.2 – Las páginas privadas solo deben ser accesibles con autenticación de usuario.
- RNF3.3 – El sistema debe protegerse contra ataques típicos como:
 - Inyección SQL
 - XSS (cross-site scripting)
 - CSRF (cross-site request forgery)

4. Mantenibilidad y escalabilidad

- RNF4.1 – El código debe estar organizado por módulos, siguiendo el patrón MVC u otro estándar.
- RNF4.2 – La aplicación debe permitir añadir nuevos destinos, usuarios o funcionalidades sin modificar el núcleo del sistema.
- RNF4.3 – Se debe incluir una documentación básica para futuros desarrolladores (README, estructura del proyecto).
- RNF4.4 – Se utilizará un repositorio Git para el proyecto.

5. Calidad y pruebas

- RNF5.1 – El sistema debe pasar pruebas de validación de formularios en cliente y servidor.
- RNF5.2 – Deben realizarse pruebas funcionales para cada caso de uso principal.
- RNF5.3 – Se deben realizar pruebas de estrés básicas para verificar el comportamiento con múltiples usuarios (simulados).

6. Despliegue

- RNF6.1 – Despliegue en un servidor web Apache o NGix local o remoto.

Requisitos Software

- RNFS.1 – Uso de un servidor BBDD Relacional: Mysql o MariaDB
- RNFS.2 – Backend: Desarrollo bajo Laravel 12 MVC.
- RNFS.3 – Frontend. Desarrollo con Bootstrap y plantillas Blade Laravel
- RNFS.4 – Uso de un servidor web diferente del de desarrollo para despliegue.

Casos de Uso

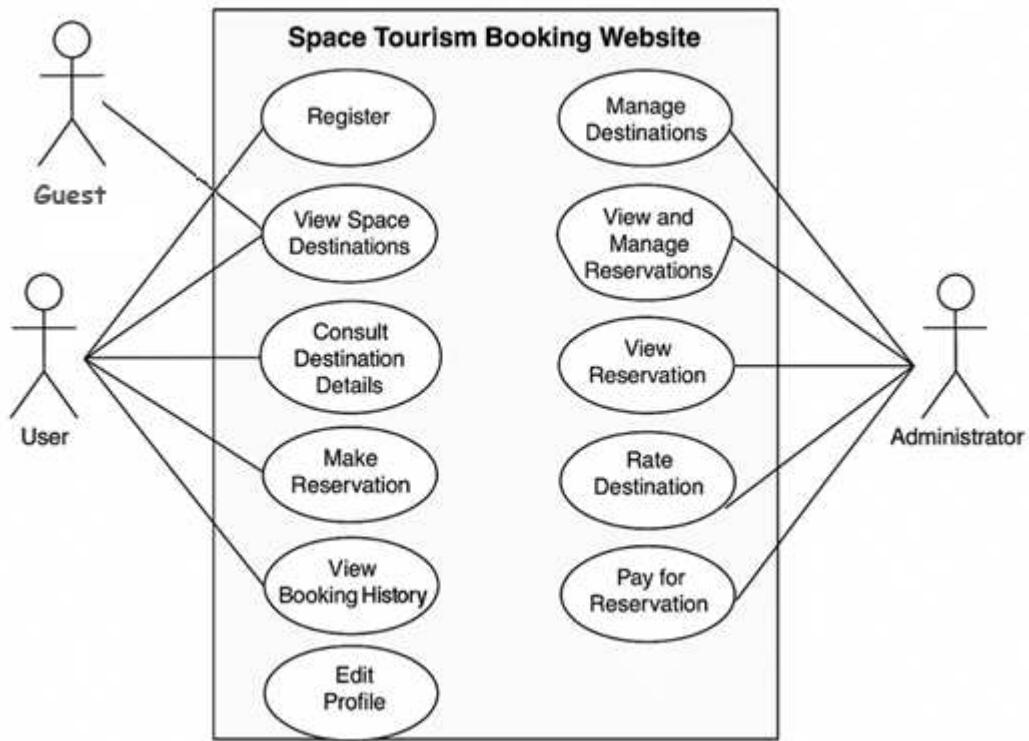
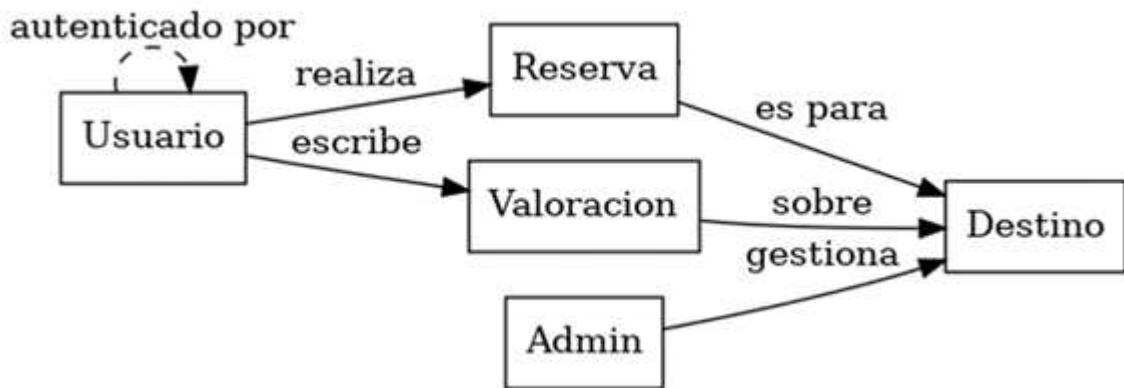


Diagrama E-R simplificado



Modelo Relacional

Usuarios

- id_usuario (PK)
- nombre
- email (único)
- contraseña
- fecha_registro
- rol (cliente / admin / guest)

Destinos

- id_destino (PK)

- nombre
- ubicacion
- descripcion
- tipo (orbital, lunar, marciano...)
- duracion_viaje
- precio
- plazas_disponibles
- imagen_url

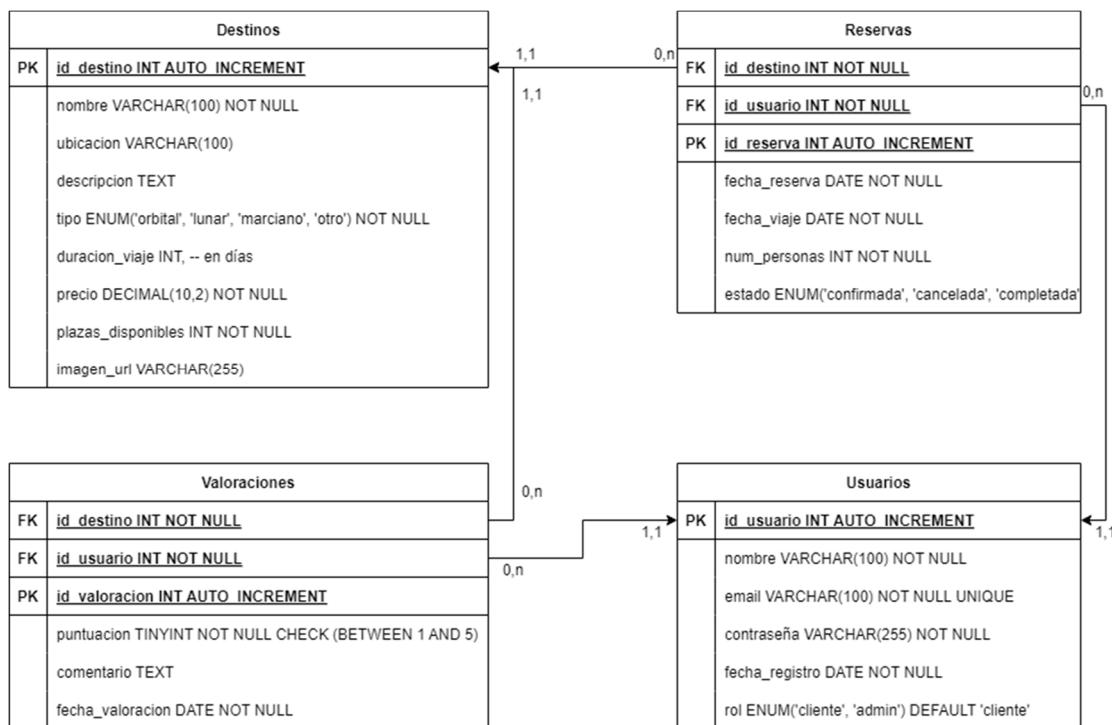
Reservas

- id_reserva (PK)
- id_usuario (FK → Usuarios.id_usuario)
- id_destino (FK → Destinos.id_destino)
- fecha_reserva
- fecha_viaje
- num_personas
- estado (confirmada / cancelada / completada)

Valoraciones

- id_valoracion (PK)
- id_usuario (FK → Usuarios.id_usuario)
- id_destino (FK → Destinos.id_destino)
- puntuacion (1 a 5)
- comentario
- fecha_valoracion

Diagrama de clases



Desarrollo

Creación del entorno y preparación del software

```
laravel new estrella_viajera
```

Configuración del fichero .env

Configurar iconos blade

```
composer require blade-ui-kit/blade-icons
php artisan vendor:publish --tag=blade-icons
composer require blade-ui-kit/blade-heroicons
php artisan vendor:publish --tag=blade-heroicons-config
```

Modificar config/blade-heroicons.php

Cambiar el tamaño al deseado

Creación de la plantilla base.blade

Creación de la vista index.blade

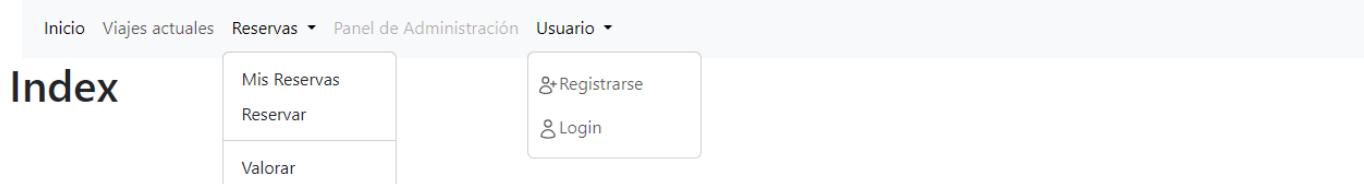
Borrado de la plantilla welcome.blade

Cambio de la ruta raíz a index.blade

Creación del menú y de la cabecera con el logo

Cambiar la plantilla para que admita un menú superior. Se definirá en otro fichero dentro de un directorio **partials**

Bienvenido a Estrella Viajera



Bienvenido a Estrella Viajera



ndex

Creación de la Base de datos

Creación de la migración de la tabla destinos y su Seeder

Creación de la migración de la tabla reservas y su Seeder

Creación de la migración de la tabla valoraciones y su Seeder

Creación Seeder de usuarios

Lanzar las migraciones y lanzar la carga de datos.

Nota: la primera tabla a crear es la de destinos si no dará problema el resto

Creación de la migración Usuarios para añadir campo tipo y su Seeder de usuarios.

Completar los datos que faltan en la tabla Users a mano y cambiar un tipo a **admin**

Crear los modelos de las tres tablas

Modificar el modelo User

Probar para cada modelo en la ruta index:

- all()
- find(1)
- find(1)->campo
- find(1)->relacion_inversa
- find(1)->relacion_inversa->campo

Creación del sistema de login/logout y control de acceso

Añadir el logo a la aplicación

Crear el directorio **images** dentro de **resources** y añadir el **logo.png**

Añadir en el fichero **resources/js/app.js**

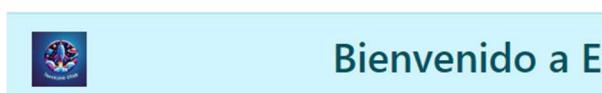
```
import.meta.glob([
  './images/**',
]);
```

Utilizar un comando similar al siguiente para establecerlo en la plantilla correspondiente

```
{{ Vite::asset('resources/images/logo.png') }}
```

Crear las rutas de login / logout

Hacer la lógica de validación



Bienvenido a Es

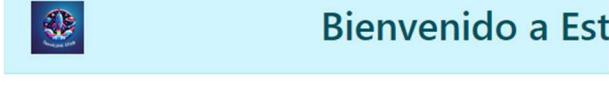
Logo

Viajar... Reservas ▾ Panel de Administración Usuario ▾

Email

Password

Sign in



Bienvenido a Est

Logo

Viajar... Reservas ▾ Panel de Administración Usuario ▾

Las credenciales no son correctas.

Email a@a

Password

Sign in



Bienvenido a Est

Logo

Viajar... Reservas ▾ Panel de Administración Usuario ▾

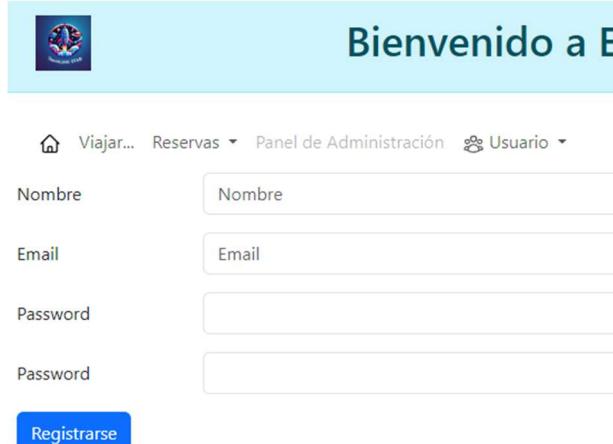
Index

Logout

Modificar la lógica para que solo se pueda acceder al panel de administrador un usuario con role admin.

Añadir un método al modelo **User:: isAdmin** que devuelva true si es el usuario es administrador

Usar en la plantilla del menú @if para controlar el acceso



Bienvenido a E

Logo

Viajar... Reservas ▾ Panel de Administración Usuario ▾

Nombre

Email

Password

Password

Registrarse

Crear la sección de registro

Crear la plantilla de registro y la ruta

Añadir la gestión del registro con validación de campos y textos de error personalizados

Modificar el menú para acceder a la ruta de registro

Crear la sección de mis datos

Crear la plantilla de mis datos y la ruta

Añadir la gestión del cambio de datos con validación de campos y textos de error personalizados

Modificar el menú para acceder a la ruta de mis datos

Controlar que no se pueda acceder directamente a ruta usuarios no autenticados

Generar los listados de destinos y el sistema de búsquedas

Crear un fichero auth_routes.php en /routes

Pasar todas las rutas del LoginController y añadir los use necesarios

Añadir `require __DIR__ . '/auth_routes.php'`; en el web.php

Crear un fichero list_routes.php en /routes

Crear toda la lógica para añadir aquí las rutas de los listados

Crear la sección de listados

Crear la plantilla de listados y la ruta

Crear el controlador para listados y datos filtrados

Solo se podrá reservar si se está autorizado

Modificar el modelo Destino añadiendo una función estática `get_tipos` que devuelva un Array con los tipos que hay en la Tabla

Generar la reserva de un destino

Crear la ruta para reservar

Crear la vista

Crear el controlador: Opcional: gestionar la grabación en una Transacción. Mostrar mensaje si se ha podido realizar.

Modificar el formulario de listados para llamar adecuadamente a la ruta

Gestionar las reservas existentes

Crear la ruta para ver mis reservas

Crear la vista

Añadir la lógica necesaria

Gestionar el cierre del viaje

Crear la ruta para ver los datos del viaje

Crear la vista

Añadir la lógica necesaria

The screenshot shows a user interface for managing reservations. At the top, there are navigation links: Reservas, Panel de Administración, and Usuario. A green success message box says "Reservado exitoso.". Below it, there's a search form with fields for "Nombre" (Name) and "tipo" (type), and a "Filtrar" (Filter) button. The main content area displays a travel trip named "Luna" with the following details:

- Descripción: viaje a la Luna orbital
- Tipo: ORBITAL
- Plazas: 3
- Precio: 500€
- Duración: 7

At the bottom, there are input fields for "Plazas" (Places) and a green "Reservar" (Reserve) button.

Refactorizar el código que se pueda según los principios SOLID – Código Limpio**Añadir al bootstrap/app.php**

```
use Illuminate\Http\Request;  
  
->withMiddleware(function (Middleware $middleware) {  
    $middleware->redirectGuestsTo(fn (Request $request) => route('login.login'));
```

Ya se puede usar en las rutas protegidas:

```
->middleware('auth')
```

En vez de al comienzo del controlador

```
if (!Auth::check()) {  
    return redirect()->route('login.login');  
}
```

Trabajo para el alumno

Tiempo desarrollo por parte del alumno estimado: 25 horas.

Implementar el RF4**Desplegar en un servidor real Apache o NGIX dentro de una Máquina Virtual o Docker y MariaDB**

Recordar migrar la BBDD también: **php artisan migrate**

Capítulo IX. Angular

<https://angular.dev/overview>

Introducción

Angular es un Framework creado y mantenido por Google. Está basado en TypeScript, un superconjunto de JavaScript y necesita de Nodejs para su correcto funcionamiento. Incluye una herramienta de consola, posibilidades de depuración y un amplio abanico de librerías instaladas por defecto, a un solo **import** de nuestro código. Es un Framework multiplataforma con dos herramientas básicas: El cliente de consola (Angular CLI) y las herramientas de depuración para el navegador (Angular DevTools) tanto para Chrome como para Firefox.

Establecer el entorno de desarrollo

Prerrequisitos

Muchos de los requisitos necesarios ya están satisfechos si se ha seguido el capítulo de Laravel, pero vamos a recopilarlos para los proyectos que estén solo basados en Angular. Primero instalación de Nodejs, se recomienda utilizar la última versión estable (LST 22.14) de <https://nodejs.org/en/download/>. Una vez instalado comprobaremos si todo está correctamente configurado con:

`node -v`

Con Nodejs se instala también el gestor de paquetes npm, también comprobaremos si está configurado correctamente mediante:

`npm -v`

El último paso es instalar el paquete de Angular en el sistema, recordemos que podemos hacer una instalación global (-g) en la misma localización para todos, pero con el inconveniente que necesitamos permisos de administrador, o por el contrario en el directorio local del proyecto, en nuestro caso. Como anteriormente, recomendamos una instalación global.

`npm install -g @angular/cli`

Si todo ha ido correctamente tendremos acceso al cliente de consola de Angular: **ng**.

`ng version`

Uso del CLI (Consola Angular)

La herramienta de consola es nuestra navaja suiza para el desarrollo bajo Angular, se utiliza para la creación del proyecto, creación de elementos software, desarrollo y test, etc. Al ser modo consola es muy fácil de usar con unos pocos comandos que debemos conocer: **new**, **add**, **build** y **generate**. Esta herramienta tiene una opción

```
C:\Users\arcipreste>node -v
v22.14.0

C:\Users\arcipreste>npm -v
10.9.2

C:\Users\arcipreste>
```

```
C:\Users\arcipreste>ng version

Angular CLI: 19.2.7
Node: 22.14.0
Package Manager: npm 10.9.2
OS: win32 x64

Angular:
...
Package          Version
@angular-devkit/architect    0.1902.7 (cli-only)
@angular-devkit/core         19.2.7 (cli-only)
@angular-devkit/schematics   19.2.7 (cli-only)
@schematics/angular          19.2.7 (cli-only)
```

help que describe su uso de forma muy rápida. Veremos con ejemplos a lo largo del capítulo su utilización (anteriormente habremos creado dicho directorio).

Creación de un proyecto

Una vez configurado el entorno, vamos a crear un proyecto de prueba. Nos desplazamos al directorio que queremos que **contenga** el proyecto y lanzamos la orden de creación

```
ng new ang_pr_01
```

Dejaremos todos los datos por defecto. Lo único remarcable es la posibilidad de usar preprocesadores para CSS, tales como SCSS o Less, pero en este curso estamos utilizando Bootstrap y es lo primero que vamos a configurar por lo que no cambiamos nada. Una vez terminado abrimos el proyecto desde nuestro IDE.

Al igual que en Laravel, se debe usar un IDE de desarrollo. Podemos utilizar el que nos parezca más adecuado, pero recomendamos VS Code o WebStorm. Después configuramos el IDE para que lance su servidor de desarrollo, monitorice todos los cambios del proyecto y se recargue, además de servir como servidor web sin necesidad de tener uno instalado en el IDE o lanzamos la orden desde el terminal:

```
ng server
```

Podremos acceder al proyecto vía un navegador en la url <http://localhost:4200>.

Añadir Bootstrap al proyecto Angular

Para comenzar, es necesario instalar Bootstrap en el proyecto. Se puede hacer utilizando npm:

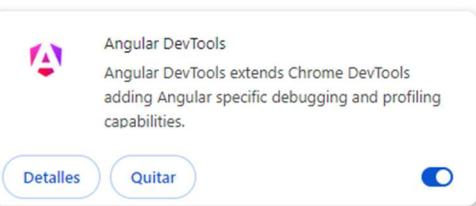
```
npm install bootstrap
```

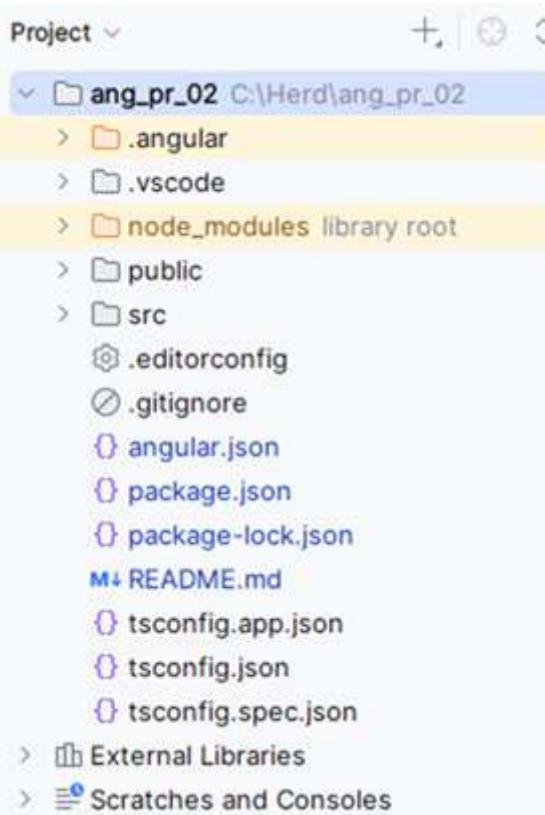
Una vez instalado, debemos importar Bootstrap en el archivo **angular.json** para que Angular lo reconozca. Se añaden las siguientes líneas en la sección de estilos y scripts:

```
"styles": [
  "node_modules/bootstrap/dist/css/bootstrap.min.css",
  "src/styles.css"
],
"scripts": [
  "node_modules/bootstrap/dist/js/bootstrap.bundle.js"
]
```

Añadir las herramientas DevTools al navegador

Para terminar esta sección, es muy recomendable que instalemos en el navegador que vayamos a usar para depuración, bien Chrome o Firefox, las herramientas de depuración de Angular, están presentes en sus respectivas tiendas de extensiones y se pueden buscar e instalar directamente.





Estructura de un proyecto Angular

Al igual que Laravel, la creación del proyecto Angular nos deja una estructura de directorios y ficheros que tenemos que conocer.

Los directorios `.angular` y `.vscode`, así como el fichero `.editorconfig` los podemos ignorar, están relacionados con la caché y el uso del IDE VS Code.

El directorio `node_modules`, es conocido de capítulos anteriores y contiene todos los ficheros que se instalan a través de la herramienta npm de NodeJs.

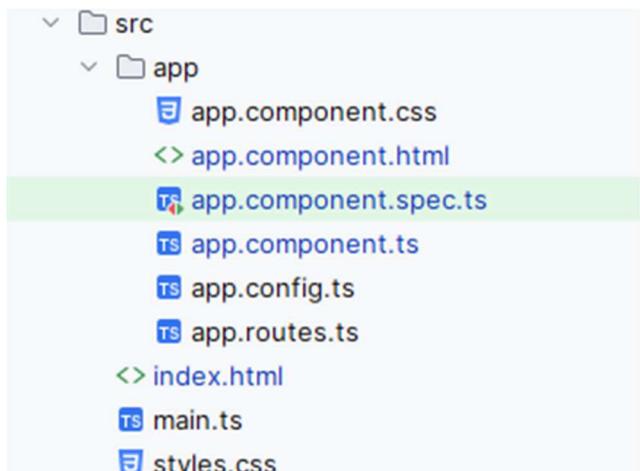
El directorio `public` tendrá la aplicación final una vez que demos la orden de despliegue al Framework.

Nuestro trabajo se centrará en el directorio `src`, que contiene toda la lógica de la aplicación.

Por último, hay que entender el fichero `angular.json`, en el que se establecen las configuraciones del proyecto; y el fichero `tsconfig.json` que se usa para fijar las configuraciones del compilador de TypeScript a JavaScript.

Estructura de una aplicación

El desarrollo de la aplicación se hará casi exclusivamente dentro del directorio `src`. Está compuesto a su vez por un directorio `app` que contendrá la mayoría de los ficheros Angular que nosotros creamos. Ya existe un **componente** (`app`) que se carga inicialmente. Hablaremos de los componentes más adelante en el capítulo, pero un componente es el bloque estructural que se utiliza bajo Angular para presentar el interfaz del usuario.



También se instala el punto de entrada a nuestra aplicación: `index.html`, su correspondiente hoja de estilos y el código asociado a la misma en TypeScript: `main.ts`. Con esto tenemos la estructura básica de la aplicación. Es importante tener desde el principio una idea básica de la estructura de la página Angular.

La página principal contiene la estructura HTML, se le añaden etiquetas propias como `app-root` (ver imagen en página siguiente). Como hemos dicho, Angular trata la estructura visual a través de componentes y la inclusión de estos dentro del código HTML se realiza a través de estas etiquetas. El

```

m  TS app.component.ts ×
import ...

@Component({
  selector: 'app-root',
  imports: [RouterOutlet]
})

```

nombre de la etiqueta lo elegimos nosotros en el fichero **.ts** del componente, pero siempre debería empezar por **app** (o un prefijo propio). De hecho si abrimos el fichero **app.component.ts** vemos el lugar en el que se define el nombre de la etiqueta (campo selector): **app-root** (ver imagen de página anterior).



```
<> index.html ×  
1  <!doctype html>  
2  <html lang="es">  
3  <head>  
4    <meta charset="utf-8">  
5    <title>AngPr01</title>  
6    <base href="/">  
7    <meta name="viewport" content="width=device-width, initial-scale=1">  
8    <link rel="icon" type="image/x-icon" href="favicon.ico">  
9  </head>  
10 <body>  
11   <app-root></app-root>  
12 </body>  
13 </html>  
14 |
```

Este componente no debería modificarse, solamente para configuraciones mínimas. Su plantilla (.html) sí podemos cambiarla según nuestras necesidades.

Las configuraciones generales de la aplicación se fijan en el fichero **app.conf.ts**. En este se definen los servicios (**services**). Son la otra pata que sostiene el Framework, se encargan de exponer la lógica de comunicaciones con elementos externos de forma unificada, veremos servicios más adelante.

El fichero **app.routes.ts** define las rutas (**routes**) que gestiona Angular. Las rutas son un concepto similar a las existentes en Laravel.

Introducción a TypeScript

TypeScript es un superconjunto de JavaScript, esto quiere decir que todo lo que se puede utilizar en este último lo aceptará el primero. Por tal vamos a hacer un rápido recorrido por las características de **JavaScript** más relevantes para poder utilizar adecuadamente Angular.

JavaScript: declaración de variables

```
function myfunc() {  
  let x = 0;  
  const precio = 20;  
  x = 10;  
  // precio = 30  
}
```

Definiremos variables con la palabra reservada **let** (aunque también existen otras maneras) y constantes mediante **const**. Para las constantes tendremos que recordar que es posible modificar los elementos de un objeto que se haya definido constante, pero no será posible cambiar el objeto completo (la referencia).

```
const producto = {id: 10, nombre:"Casa"};
producto.id = 20 // es válido
// producto = ..... //esto no es válido
```

El operador de extensión (...) permite expandir un Array o un objeto en sus elementos para ser tratados de forma individual.

```
const categoria = "Pisos";
const categorias = ["Casas", "Chalets"]
const todo = [...categorias, categoria] // ["Casas", "Chalet", "Pisos"]
```

Para objetos se podría utilizar

```
const producto = {
  nombre:= 'Zapatillas',
  precio: 234
};

const otroProducto = { //Esta estructura es muy común en Angular
  ...producto,
  categoria: 'Varios'
}
```

JavaScript: parámetros de funciones

Además de la definición de parámetros generales en una función, debemos considerar el uso de los parámetros por defecto. Estos se definirán al final de la cabecera de la función, los últimos, asignando un valor, para utilizarlos en caso que no se pasen a la función durante la llamada.

```
function sumar(ob_1, op_2=0){
  return ob_1 + op_2;
}
sumar(1,2); // 3
sumar(1); // 1
```

El último tipo que tenemos que mencionar, es aquel que nos permite recoger un número indeterminado de parámetros. Este tipo de parámetro, debe ser el último exactamente y utilizaremos el operador extensión para su definición.

```
function sumar(op_1, ...resto){
  return op_1 + resto.reduce((acc, curr) => acc + curr, 0);
}
sumar(1,2); // 3
sumar(1, 2, ,3, 4);
```

JavaScript: funciones flecha

Es un tipo de función anónima en la que no cambia el contexto actual, manteniendo el de la función que la llama. No solo es una reducción en la sintaxis a la hora de la definición. El caso más claro, es el uso de la palabra reservada **this**. Al utilizar dentro de un objeto esta palabra, y llamar a otra función, el valor de this apuntará ahora al nuevo objeto, siendo imposible acceder a los datos anteriores. Si esa función es de tipo flecha, al no cambiar el contexto, **this** no cambia y tendremos acceso a los mismos datos.

```
const mi_funcion = (precio) => { return (precio/100)*10;};
```

La definición de funciones flecha, permite eliminar los paréntesis si solo hay un parámetro, pero no si no hay ninguno o varios, y las llaves en caso que solo haya una única sentencia que sea **return**.

```
const mi_funcion = precio => (precio/100)*10;
```

JavaScript: acceso condicionado

Esta facilidad permite acceder a un elemento de un objeto solo si es no nulo, evitando que se generen errores de acceso.

```
console.log(mi_producto?.precio?);
```

En este caso tanto si la variable **mi_producto** es nula, como si no lo es, pero también es nulo el campo **precio**, no se producirá ningún error.

JavaScript: operador Fusión de Null (??)

Este operador permite devolver un valor por defecto al comprobar que la variable sea nula, en otro caso se devuelve el valor de la variable.

```
valor = precio ?? -1; // Si precio es nulo devuelve -1, si no devuelve precio
mi_producto.precio ??= -1; // Si queremos validaciones esta sintaxis es más rápida
console.log(mi_producto.precio);
```

JavaScript: clases

La creación de clases en JavaScript es muy similar al de otros lenguajes, lo más interesante respecto a lo que nos incumbe es la definición de propiedades privadas, que se realiza anteponiendo el carácter **#** delante del nombre correspondiente y posteriormente también durante su uso.

```
class MiClase{
    p_publica = 23;
    #p_privada = 45;
    constructor(){}
    datos(){
        return [this.p_publica,this.#p_privada];
    };
    get privada(){
        return this.#p_privada;
    }
}
mi_objeto = new MiClase();
```

```
console.log(mi_objeto.datos());
console.log(mi_objeto.p_publica);
//console.log(mi_objeto.#p_privada); // Error, privada
console.log(mi_objeto.privada);
```

También está permitido el uso de getters y setters a través de la sintaxis **get** y **set** tal y como vemos en el ejemplo anterior. Por supuesto es posible la herencia simple mediante **extends** y es necesario en el constructor de la clase hija llamar al constructor de la clase padre a través del método **super**.

JavaScript: módulos

Un fichero en cualquier directorio se puede convertir en un módulo de librería simplemente no añadiendo código que se ejecute directamente, encapsulándolo en clases o funciones y añadiendo delante de la definición de cada elemento que queramos exportar la palabra clave **export**. Evidentemente, el fichero que necesite hacer uso estos elementos tendrá que importarlos

```
import { MiClase } from 'rutacompleta/fichero';
```

Las rutas serán relativas al fichero que hace la importación. Por otro lado, si se necesitan varios elementos del mismo módulo, se pueden importar en la misma línea, separándolos por comas dentro de las llaves.

Qué es TypeScript

De la forma más simple, es un JavaScript con esteroides, en el que todas las características extra que se han añadido son opcionales, por lo que **todo código JavaScript** funcionará dentro de TypeScript. Este nuevo lenguaje es compilado, y generará al final código JavaScript ya que no existen todavía navegadores que interpreten directamente TypeScript. La instalación en el sistema se debe hacer de forma global

```
npm install -g typescript
```

Y usar la extensión **.ts** en los ficheros. Cada proyecto crea un fichero de configuración para el compilador llamado **tsconfig.json**. Este fichero se utilizará automáticamente, cuando lancemos el compilador desde la línea de comandos, situados en el mismo directorio que la raíz del proyecto.

```
tsc app.ts
```

Desde Angular este proceso se automatiza cuando lanzamos el servidor con ng. El proceso de compilado y pasar las fuentes a JavaScript se denomina **traspilación**.

Tipos en TypeScript

La primera y más notable mejora es la inclusión del tipado en la definición de las variables y parámetros del lenguaje. De esta manera, se comprobará en tiempo de compilación si corresponden los tipos y seremos avisados en caso aparezca algún error de tipo. Los tipos no se traspasarán a JavaScript ya que no están soportados. La definición de tipos es opcional, el compilador hace un buen trabajo infiriendo el tipo de la variable en función de su primera definición, y si no es capaz determinará uno especial llamado **Any** que alberga a todos. Los tipos básicos son: **string**, **boolean**, **number** (tanto decimales como no decimales).

```
const isActive: boolean = false;
const precio:number = 78;
const nombre:string = 'Zapatillas';
```

Por supuesto, también podemos definir Arrays de elementos mediante **Array** o los corchetes **[]**.

```
const isActive: number[] = [];
const nombres: Array<string> = [];
```

Solo indicar que, si usamos el tipo **Any** o se infiere en alguna variable, estaremos reduciendo al mínimo el sistema de comprobación de tipos por lo que es muy recomendable usarlos de forma correcta.

Para terminar dos aspectos, la existencia de los tipos personalizados y del tipo **unknown**. Este último es similar a Any pero no permite utilizarlo en código a menos que se haga una validación de tipo anterior. Para definir tipos personalizados utilizaremos la palabra reservada **type**.

Funciones

En TypeScript se puede anotar con tipos tanto los parámetros como el resultado a devolver en el momento de la definición.

```
function miFuncion(mi_parametro:number=0): boolean {
    return mi_parametro > 0;
}
```

En caso que la función no devuelva nada, utilizaremos como en otros leguajes **void** para indicarlo. TypeScript añade la posibilidad de indicar parámetros optativos a través del carácter: **?** en la definición, aunque yo prefiero la notación de JavaScript (=valor por defecto) ya que la anterior obliga a usar el operador expansión de nulo **(??)** antes de usar el parámetro.

```
function miFuncion_1(mi_parametro?:number): boolean {
    mi_parametro ??= 0;
    return mi_parametro > 0;
}

function miFuncion_2(mi_parametro:number=0): boolean {
    return mi_parametro > 0;
}
```

Modificaciones a las clases

En TypeScript se añade una nueva palabra reservada **private** para denotar las variables privadas, en vez de la notación de **#** de JavaScript. La sintaxis se hace mucho más clara.

```
class MiClase{
    p_publica = 23;
    private p_privada = 45;

    constructor(){}
    datos(){ return [this.p_publica,this.p_privada]; }
    get privada(){ return this.p_privada; }
}
```

Para determinar el tipo de una clase, al igual que en JavaScript, podemos usar el operador **instanceof** que devolverá verdadero o falso si un objeto es de una clase concreta.

Interfaces

Un interfaz es un conjunto de métodos y propiedades que una clase debe obligatoriamente implementar, es un contrato sobre un esquema definido. El concepto de interfaz está muy extendido dentro de los lenguajes POO con herencia simple. JavaScript no implementa este concepto, pero TypeScript lo añade.

```
interface Producto{
    nombre:string;
    precio: number;
    get_descuento: (desc:number) => number;
}
```

Una vez establecido, las clases deben implementar el interfaz, usando **implements** en su definición, añadiendo las propiedades del interfaz e implementando los métodos.

```
class ProductImpl implements Producto{
    nombre: string;
    precio: number;
    constructor(nombre:string, precio:number){
        this.nombre = nombre;
        this.precio = precio;
    }
    get_descuento(desc:number):number{
        return this.precio - (this.precio * desc);
    }
}
```

Genéricos

Otras de las herramientas de los lenguajes POO son los genéricos. Estas clases definen el uso sobre un tipo genérico en vez de uno concreto, pudiendo así trabajar sobre varios tipos. Por ejemplo, al crear una lista, en vez de hacerla sobre string solo, se puede hacer genérica y usarse tanto string como para number o cualquier otro.

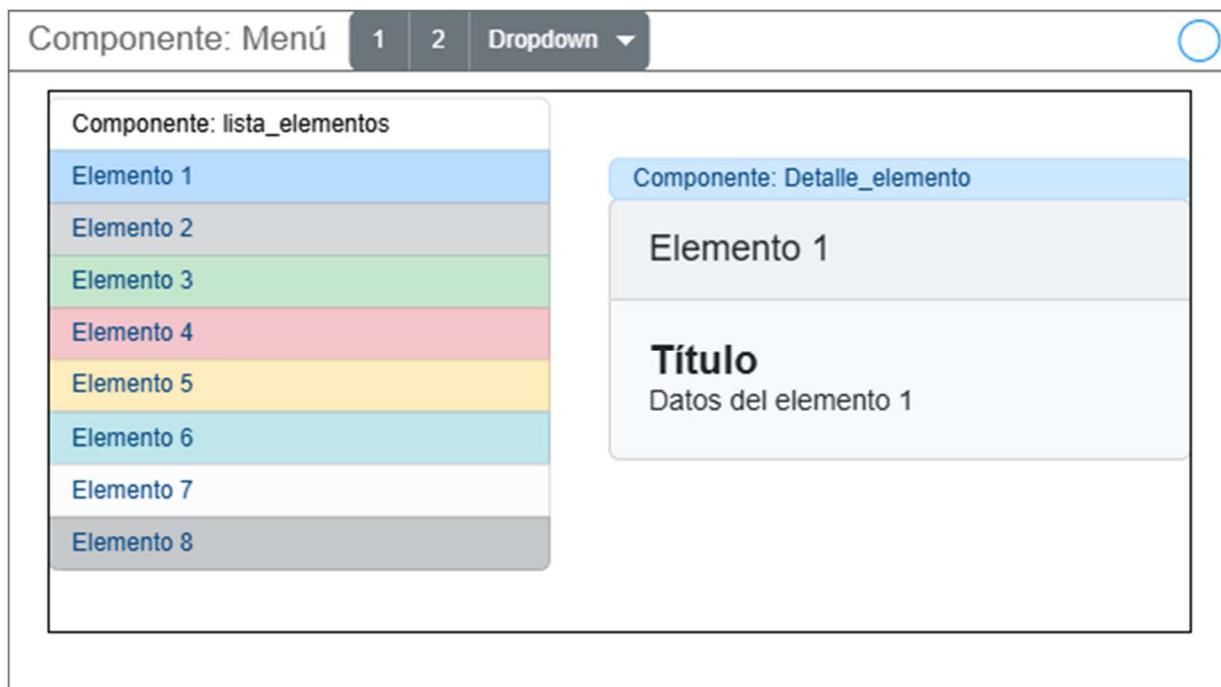
```
function save<T>(obj: T) {
    const json = JSON.stringify(obj);
    localStorage.setItem('appConfig', json);
    return obj;
}
save<string>("Hola")
save<number>(123)
save({nombre: "Juan", edad: 30})
save([1, 2, 3])
```

Tipos utilitarios

Para crear un subtipo a partir de otro ya existente, pero modificando cómo se heredan campos, podremos usar el concepto de **Partial** o el tipo **Pick**. Estos tipos permiten algunas funciones más. Para más información visitar:

<https://www.typescriptlang.org/docs/handbook/utility-types.html>

Componentes



Los componentes son el elemento organizativo del GUI de una aplicación Angular. Son los bloques básicos de cualquier aplicación. Controlan la vista (que son cada parte visual de una aplicación) y son los responsables de la lógica de presentación. Todos los componentes están organizados de forma jerárquica, siendo el elemento padre el componente **root**, generalmente definido en la creación del proyecto. Cada componente puede comunicarse con cualquier otro e interactuar con él, independientemente de su posición en el árbol jerárquico de componentes.

```
<body>
  <app-root></app-root>
</body>
```

Estructura de un componente

Durante mucho tiempo la forma de organizar la aplicación angular era a través de los módulos (NGModules). Los módulos definían en sí interior componentes y tenían que ser importados por otros módulos. Esta estructura añadía una capa innecesaria a la programación, por lo que se decidió eliminar y establecer por defecto la definición de los elementos sin los módulos. Si bien todavía es compatible con ellos, se recomienda no usarlos y utilizar la definición de componentes solos.

Un componente está formado por cuatro elementos: La lógica del componente y su definición; una plantilla de presentación; un conjunto de estilos CSS que se le aplicarán; y un fichero de pruebas. Estos se corresponden con cuatro ficheros por cada componente creado a través de la orden:

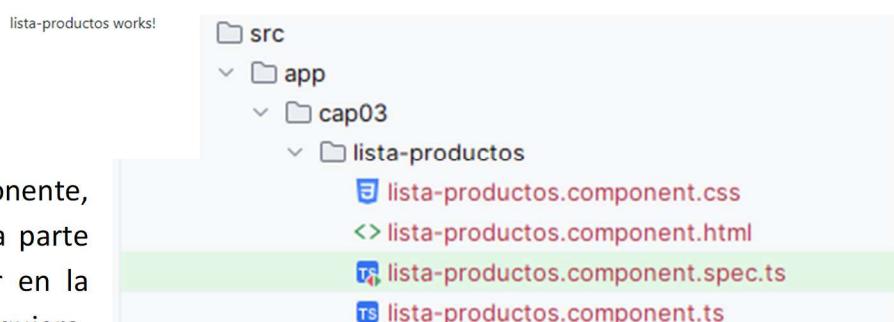
```
ng generate component ruta/nombre
```

```
ng generate component cap03/lista-productos
```

Hello, ang_pr_01

Congratulations! Your app is running. 🎉

Una vez creado el componente, tendremos que incluirlo en alguna parte de la página web, debe aparecer en la plantilla de otro componente cualquiera, vamos a hacerlo a modo de ejemplo en la plantilla del componente root o principal. Modificaremos el fichero **app.component.html** añadiendo la etiqueta **app-lista-productos**, importándolo mediante una ruta relativa al directorio del componente, dentro del fichero **app.component.ts** y ejecutaremos el proyecto.



La orden generación de componentes creará la ruta completa que se le indique antes del nombre más un

```
ts app.component.ts ×
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { ListaProductosComponent } from './cap03/lista-productos/lista-productos.component';

@Component({
  Show usages  csanjuanp-ies *
  selector: 'app-root',
  imports: [RouterOutlet, ListaProductosComponent],
```



```
<> app.component.html ×
<div class="divider" role="separator" aria-label="Divider"></div>
<app-lista-productos></app-lista-productos>
```

directorio dedicado para el componente con dicho nombre, en este es donde se inicializarán los cuatro ficheros del componente. Los ficheros creados son:

- **nombre_componente.html**. Este fichero es una plantilla html que admite directivas similares a las vistas ya en Laravel, se verán más adelante. Es la representación visual de una parte de la página, la que corresponde a este componente.
- **nombre_componente.css**. Conjuntos de estilos a aplicar al componente en la plantilla, en nuestro caso no los vamos a utilizar casi al configurar Bootstrap como Framework de CSS, pero nos permite que podamos establecer clases propias y aplicarlas localmente al componente sin interferir con el resto.
- **nombre_componente.spec.ts**. Fichero de test básico.

- **nombre_componente.ts.** Lógica de ejecución del componente, la clase que maneja la vista. A su vez este fichero tiene dos partes, la definición de componente y la de programación. La que vemos a la derecha es la definición del componente en la que se establece de arriba abajo, el nombre de la etiqueta (**selector**) que usaremos en las plantillas de otros componentes (podemos poner el que queramos e incluso cambiar el prefijo **app** predefinido por otro al crear el componente mediante **ng** y el parámetro **--prefix**); los nombres de otros componentes o servicios que usaremos, en la sentencia **imports**; el nombre que tendrá el fichero de plantilla HTML y el de CSS. Estas dos últimas propiedades admiten que utilicen por otras en la que el contenido se escribe a continuación, pero no se deben usar al ser esta estructura mucho más limpia. Este decorador admite más propiedades que no entraremos en detalle en este apartado.

Comunicación entre la lógica y la plantilla

Evidentemente, queremos tener sincronizados los datos del componente y la vista para que esta sea dinámica.

Para mostrar el valor de un propiedad del componente en la plantilla, simplemente utilizaremos el nombre de dicha propiedad encerrada entre llaves dobles (esta acción se llama interpolación): **{{ propiedad }}**. La asociación es dinámica y reflejará cambios posteriores del componente (componente → plantilla) de forma automática, se utiliza para establecer cadenas de texto posiciones concretas de la plantilla. Si necesitamos enlazar una propiedad del **DOM** de la etiqueta HTML correspondiente con una propiedad del componente se hace mediante la sintaxis: **[propiedadDOM] = "propiedad_componente"** (Ver imagen anterior). El efecto puede ser similar a la interpolación, pero se puede trabajar con objetos complejos que acepte la propiedad, no solo cadenas que genera la interpolación.

The screenshot shows two files side-by-side. On the left is `lista-productos.component.html` containing:

```

1 <p>lista-productos works!: <b>{{ title }}</b></p>
2 <h1 name="data" [innerText]="title"></h1>
3

```

On the right is `lista-productos.component.ts` containing:

```

T5 lista-productos.component.ts
export class ListaProductosComponent {
  title: string = 'Lista de Productos';
}

```

El mecanismo anterior nos permite unir una propiedad del nodo DOM a una propiedad nuestra. Pero si necesitamos enlazar atributos HTML a propiedades propias, la sintaxis es: **[attr.atributo] = "propiedad"**. En ambos casos, la propiedad o atributo entre los corchetes se denomina **propiedad objetivo**, la variable entre comillas, **expresión de plantilla**.

Nota: Hay que distinguir muy bien cuando hablamos de atributo HTML (class) de cuando nos referimos a una propiedad del DOM (innerHTML). No es lo mismo.

Usando las expresiones anteriores, podemos unir atributos de clase o de estilo a una propiedad del componente, pero es posible hacerlo de forma más rápida: **[class.clase] = "propiedad_booleana"**: la expresión anterior añadirá al elemento HTML la **clase** especificada si es verdadera la propiedad Booleana

del componente, o la eliminará en caso que sea falsa. Si usamos solo **[class]**: se deberá unir a una propiedad que contenga una lista de clases CSS separadas por espacios en blanco; o a un objeto en donde las claves serán el nombre de la clase y el valor verdadero o falso para incluirla o no.

Podemos hacer algo similar con la propiedad style de HTML **[style.atributo] = "propiedad_booleana"** o **[style] = "propiedad"**. Para usar los estilos, existe una sintaxis avanzada cuando el atributo tiene unidades, pudiéndose usar de la siguiente manera: **[style.width.px] = "propiedad"**.

Comunicación bidireccional

Para una comunicación bidireccional, por ejemplo en los formularios, se debe usar la sintaxis **[(ngModel)] = "persona.password"**. En donde **ngModel** es una directiva fija, y entre comillas la propiedad del componente que queremos actualizar, en este caso una propiedad de un objeto. Se estudiará con más detenimiento más adelante en el capítulo.

Directivas de plantilla

Al igual que en Laravel, existen directivas de plantilla para mostrar un código HTML u otro código en función de condiciones o repeticiones. En las versiones anteriores de Angular la sintaxis era distinta basada en directivas (@ngIf, @ngFor, @ngSwitch), pero se recomienda usar la presentada en esta sección.

```
<> lista-productos.component.html
<p>lista-productos works!: <b>{{ title }}</b></p>
<H1 name="data" [innerText] = "title"></H1>
@if (productos.length > 0){
  <h2>Lista de productos con {{ productos.length }}</h2>
  <ul class="list-group">
    @for (producto of productos; track producto.id; let i = $index) {
      <li class="list-group-item">
        <h3>{{i}} -> {{ producto.nombre }}</h3>
        <p>@switch (producto.id) {
          @case (1){ ☕ }
          @case (2){ ☕ }
          @default { ✨ }
        }
        Precio: {{ producto.precio }}</p>
      </li>
    }
  </ul>
}
@else {
  <p>No hay productos disponibles.</p>
}<img alt="Yellow lightbulb icon indicating a tip or note." data-bbox="237 825 255 842"/>
```

- **@if @else**. Se muestra o no un bloque en función de una condición.
- **@for @empty**. Se utiliza para iterar a través de una lista. Debemos usar la variable **track** para mejorar el rendimiento de la aplicación, hay que establecerla a un valor único para cada elemento (generalmente un identificador único: *producto.id*) que hace distinguir cada componente de otro.

Angular sincroniza los cambios en el DOM con los Arrays y sus correspondientes nodos, actualizando de forma automática la presentación con el contenido de la lista, incluso si se añaden o eliminan elementos de ella. Este procedimiento puede ser muy lento, por lo que se usa la propiedad **track** para acelerarlo.

La expresión **@empty** se utilizará para los casos en la que la lista esté vacía.

Es posible definir una variable en el **for** para que recoja un valor en cada vuelta y utilizarlo dentro del bucle. En el código ejemplo se puede ver la variable **i** definida para tal fin. Las propiedades que podemos usar son: **\$index**, **\$count**, **\$first**, **\$last**, **\$even** o **\$odd**.

- **@switch @case @default**. Con el mismo significado que en un lenguaje tradicional, determinará que código visualizar en función de una variable.

Eventos de plantilla

Hemos visto anteriormente cómo pasar datos desde el componente hasta la plantilla, pero vamos a abordar ahora el caso contrario. Para que el componente sea capaz de recoger acciones que el usuario haga tendremos que capturar los eventos correspondientes. Esta técnica se denomina **enlace por eventos**. Podemos utilizar dos mecanismos: enlazando con una propiedad o ejecutando un método del componente, vamos a explicar ambas soluciones.

Primero enlace: Añadimos la propiedad **productoSeleccionado** al controlador, podremos enlazarla y usarla en la plantilla tal y como se ve en la imagen inferior resaltado en azul. El evento va siempre entre paréntesis y se pueden utilizar todos los eventos del **DOM**. A continuación, está la expresión de variable en la que se asigna el cambio (**producto**). En este caso se guarda todo el producto, pero se puede almacenar un valor simple.

```

1  import { Component } from '@angular/core';
2  import { Producto } from '../producto';
3
4  @Component({
5      selector: 'app-lista-productos',
6      imports: [],
7      templateUrl: './lista-productos.component.html',
8      styleUrls: ['./lista-productos.component.css']
9  })
10 export class ListaProductosComponent {
11     title: string = 'Lista de Productos';
12     productos: Producto[] = [
13         { id: 1, nombre: 'Producto 1', precio: 10.99 },
14         { id: 2, nombre: 'Producto 2', precio: 20.99 },
15         { id: 3, nombre: 'Producto 3', precio: 30.99 },
16         { id: 4, nombre: 'Producto 4', precio: 40.99 },
17         { id: 5, nombre: 'Producto 5', precio: 50.99 }
18     ];
19 }

```

```

1  export interface Producto {
2      id: number,
3      nombre: string,
4      precio: number
5  }

```

productoSeleccionado: Producto | null = null;

```

<ul class="list-group">
    <ng-container *ngFor="let producto of productos; track producto.id; let i = $index">
        <li class="list-group-item" (click)="productoSeleccionado=producto">
            <ng-container *ngIf="productoSeleccionado?.id == producto.id">
                <h3>{{i}} - {{producto.nombre}} - Seleccionado </h3>
            </ng-container>
            <ng-container *ngElse>
                <h3>{{i}} - {{producto.nombre}}</h3>
            </ng-container>
        </li>
    </ng-container>
</ul>

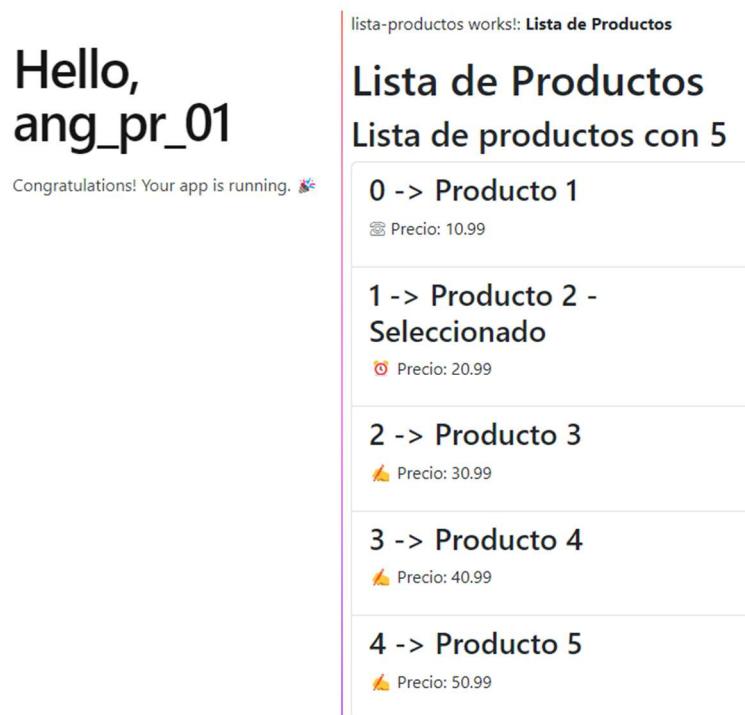
```

Segundo ejecutando: en vez de realizar una asignación, establecemos la llamada al método. El resultado es el mismo, pero a mi parecer un poco más claro.

En ambos casos, se puede utilizar la técnica de filtrado de eventos o pseudo-eventos para limitar a un tipo concreto de evento dentro de varios de una clase. Así para los eventos keypress se puede limitar solo a algunos usando: keyup.enter, o keydown.escape. Ver la documentación oficial para más información.

```
<li class="list-group-item" (click)="seleccionarProducto(producto)">
  @if (productoSeleccionado?.id == producto.id) {
```

En ambos casos es el resultado de la imagen siguiente:



Comunicación entre componentes

La comunicación entre diferentes componentes es esencial para la ejecución de la aplicación. Un componente lista tendrá que seleccionar el hijo correspondiente, un componente formulario deberá informar al padre de la acción a tomar, etc. Para ese fin, cada componente expone un API público de comunicaciones.

Primero, vamos a trasladar datos del padre al hijo. En este caso, en el **componente hijo** crea una señal de entrada (@input o @signal) que será lanzada automáticamente cada vez que cambie el valor de la **propiedad del padre**. Es más fácil pensar que el conjunto de señales @input es el API pública de un componente, que define cómo se puede configurar o qué datos puede recibir. Es posible ejecutar una función de transformación sobre los datos recibidos a la vez, esta funcionalidad no la vamos a ver en este punto.

Para que el padre ejecute la señal, la incluiremos dentro de la etiqueta del hijo en la plantilla del padre, como cualquier otra propiedad.

Por ejemplo, en la **plantilla del padre** (lista-producto.component.html) incluiremos:

```
<app-detalles Producto [producto]="producto"></app-detalles Producto>
```

La señal **producto del hijo** será llamada con el valor de la **propiedad producto del padre**. El hijo habrá hecho una definición de la forma:

```
producto = input.required<Producto>;  
o  
@Input({required: true}) producto: Producto | null = null;
```

Cualquiera de las dos sintaxis está aceptado. Si bien es recomendada la primera forma (input), en mi versión de Angular daba problemas, pero la segunda funcionaba perfectamente.

The screenshot shows a code editor with two tabs: 'lista-productos.component.ts' and 'lista-productos.component.html'. The 'lista-productos.component.ts' tab contains the following TypeScript code:

```
seleccionarProducto(producto: Producto): void { Show usages  
    if (this.productoSeleccionado !== producto && this.productoSeleccionado !== null) {  
        this.productoSeleccionado.seleccionado = false;  
    }  
    this.productoSeleccionado = producto;  
    this.productoSeleccionado.seleccionado = true;
```

The 'lista-productos.component.html' tab contains the following HTML template:

```
<p>lista-productos works!: <b>{{ title }}</b></p>  
<H1 name="data" [innerText]="'title'"></H1>  
@if (productos.length > 0){  
    <h2>Lista de productos con {{ productos.length }}</h2>  
    <ul class="list-group">  
        @for (producto of productos; track producto.id; let i = $index) {  
            <li class="list-group-item" (click)="seleccionarProducto(producto)">  
                <app-detalles Producto [producto]="producto"></app-detalles Producto>
```

```
<> detalles Producto.component.html </>

@if (producto?.seleccionado) {
  <h3>{{ producto?.nombre }} - Seleccionado </h3>
}
@else {
  <h3>{{ producto?.nombre }}</h3>
}
<p>@switch (producto?.id) {
  @case (1){ ☰ }
  @case (2){ ☱ }
  @default { ✎ }
}</p>
Precio: {{ producto?.precio }}</p>

detalles Producto.component.ts
```

```
export class DetallesProductoComponent {
  // producto = input.required<Producto>;
  @Input({required: true}) producto: Producto|null = null;
```

La segunda parte de la comunicación se da en el sentido contrario, del hijo al padre. Para este mecanismo se tiene que crear un evento en el hijo que será capturado por parte del padre. El hijo cuando sea necesario pasar información al padre lo emitirá.

El primer paso es crear una señal de salida y la función que lanzará dicha señal dentro del fichero .ts del hijo.

```
clic_buton = output<string>(); //Evento de salida
clicBoton(): void { //método ejecutado en la plantilla para comunicarse con el padre
  this.clic_buton.emit("Emitido por "+ this.producto?.id);
}
```

En la plantilla del hijo (.html) creamos la posibilidad de lanzar el evento, en el ejemplo en el clic de un botón:

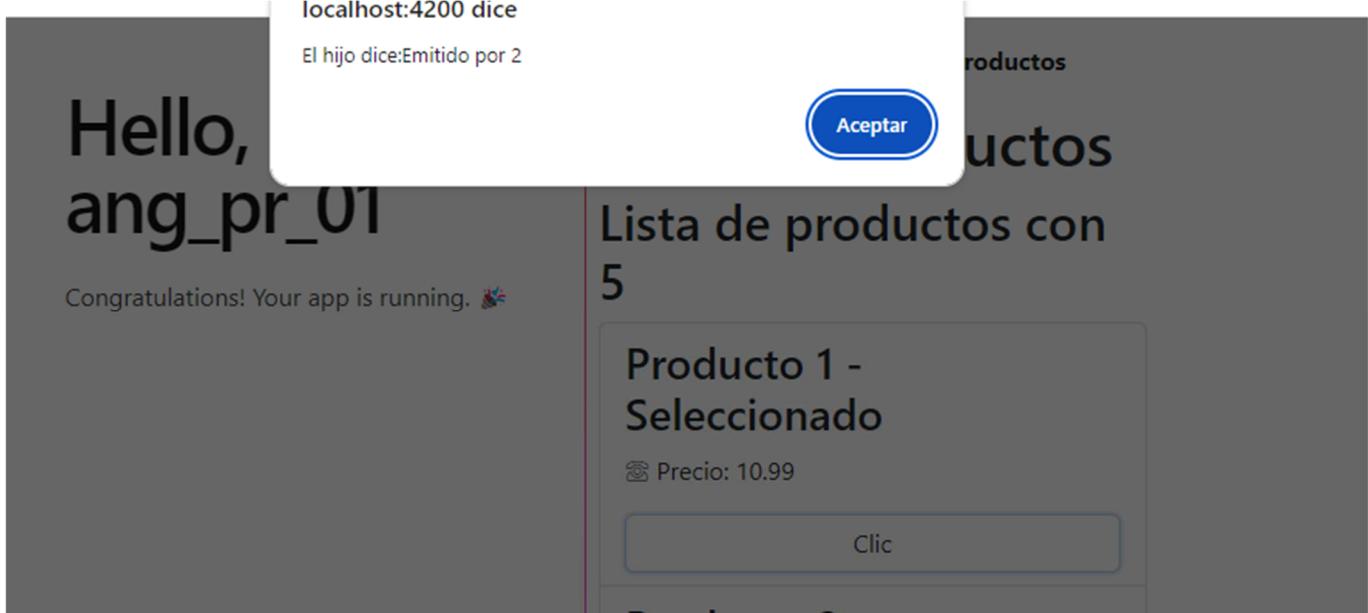
```
<button (click)="clicBoton()" class="form-control">Clic</button>
```

Después crearemos el controlador del evento en el padre que recibirá los datos y lo enlazamos en la plantilla del padre. En el fichero .ts:

```
onClicBoton(data:string): void {
  alert("El hijo dice:" + data); //gestión de datos
}
```

En la plantilla del padre (.html):

```
<app-detalles-producto [producto]="producto" (clic_buton)="onClicBoton($event)">
</app-detalles-producto>
```



El paso de información del hijo al padre es el caso más común, pero es totalmente optativo, no hace falta devolver ningún dato, por lo que podemos eliminar de la definición y en la emisión del evento la variable **data**. La variable **\$event** es una palabra reservada de Angular que recoge los datos transferidos en el evento. En este ejemplo hemos pasado una cadena, pero se podría devolver un objeto complejo.

Hemos explorado la comunicación bidireccional de los componentes, pero podemos añadir un último método en el que el padre tiene control total sobre las propiedades del hijo. La forma es definiendo una variable de plantilla (#hijo) que haga referencia al hijo en la etiqueta:

```
<app-detalles-producto #hijo ...>

<span>{{ hijo.producto?.precio }}</span>
```

Ciclo de vida de un componente

Todo componente tiene un ciclo de vida, desde su creación hasta su destrucción. Angular proporciona enganches a los que podemos asociar una función que será ejecutada en ese momento del ciclo de vida. El uso de estos enganches es opcional, pero algunos son muy prácticos. Los existentes son:

- **ngOnInit**: Se llama en la inicialización. En este momento todas propiedades de datos han sido definidas, pero no tienen los valores, no se han establecido todavía. Se suele utilizar mucho para cargar datos desde servicios.
- **ngOnDestroy**: Es llamado al eliminar el componente. En el momento que el componente sea eliminado del DOM se lanza este evento, bien por el uso de la directiva @if o por cambiar de ruta.
- **ngOnChanges**: Se ejecuta cuando cambia el valor en alguna propiedad enlazada con la plantilla. Este evento acepta un parámetro del tipo SimpleChanges que contiene un diccionario en el que aparecerán como claves todas las propiedades y podremos acceder a los valores anterior y actual de la misma.
- **ngAfterViewInit**: Es llamado una vez está inicializada la vista del componente y de todos sus hijos. Solo se deberá intentar acceder a los elementos de las consultas @ViewChild o @ViewChildren en este evento, no antes.

```
export class DetallesProductoComponent implements OnInit{
  ngOnInit(): void {
    if (this.producto) {
      console.log('Detalles del producto:', this.producto);
    } else {
      console.log('No se ha proporcionado un producto.');
    }
  }
}
```

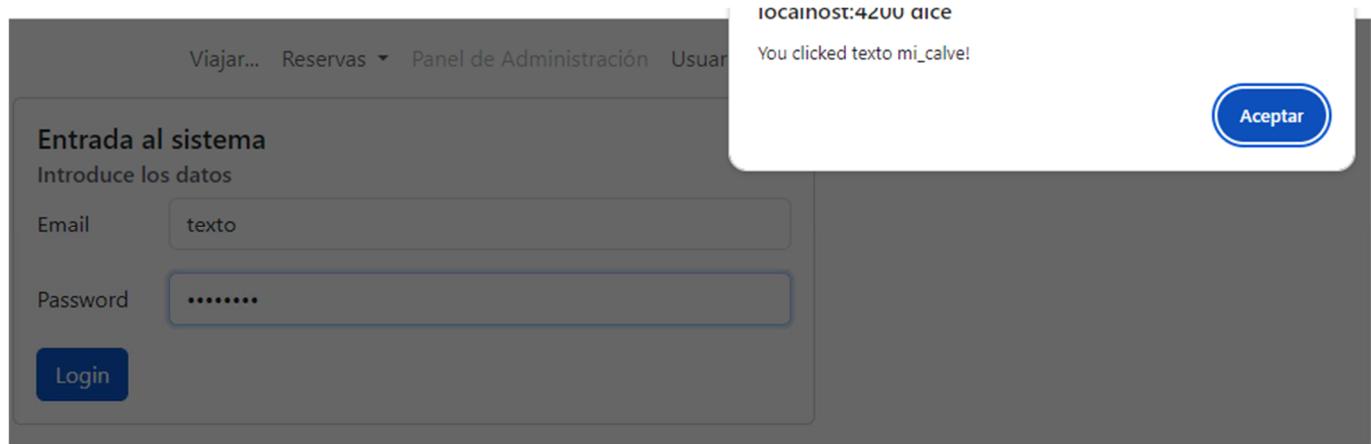
El código de ayuda anterior se puede ver en cap03.

Patrones de programación (Ver código de ayuda: cap03_patrones)

MENÚ



FORMULARIO DE VALIDACIÓN



Tuberías y directivas

Tuberías

Dos herramientas que debemos conocer son las tuberías y las directivas. Las primeras permiten modificar la visualización de una propiedad en la plantilla y la segunda permite realizar funciones más complejas como modificar el comportamiento de un nodo HTML o su apariencia.

En general una tubería (|) recoge una entrada y la transforma en el formato de salida deseado mostrándolo en la plantilla. Si la tubería necesita algún tipo de parámetro se pasará a continuación separándolo por dos puntos:

expresión|tubería: parámetro

```

1  export interface Producto { Show usages
2    id: number,
3    nombre: string,
4    precio: number,
5    seleccionado?: boolean
6    categorias: Record<number, string>
7  }
  
```

```

  <p>Precio: {{ producto?.precio | currency: currencyCode: 'EUR' }}</p>
  <div class="container">
    @for (cat of producto?.categorias| keyvalue; track cat.key) {
      <p class="text-primary m-0"> {{ cat.value | uppercase }}</p>
    }
  </div>
  <button (click)="clicBoton()" class="form-control">Clic</button>
  
```

Angular incluye un conjunto bastante amplio de tuberías predefinidas:

- **uppercase/lowercase**: Para transformación a mayúsculas o minúsculas.
- **date**: genera un formato de salida tipo fecha, se puede configurar según nuestras necesidades a través del parámetro. Para ver el formato de esta tubería visitar la url: <https://angular.dev/api/common/DatePipe#pre-defined-format-options>
- **currency**: permite mostrar un número en formato moneda dependiendo de la configuración local del sistema. Le podemos pasar como parámetro la moneda a usar en caso de necesidad `currency:'EUR'`.
- **json / keyvalue / slice**: permiten convertir al formato JSON, convierte en una colección de pares clave – valor y permite extraer partes de subcadenas.
- **async**: se usa cuando gestionamos datos asíncronos en nuestro componente y hay que asegurarse que las vistas cambian adecuadamente, por ejemplo, a recoger datos de un servidor externo.

Aunque no se encuentra dentro de la categoría de tuberías, es buen momento para introducir los alias de plantillas. Nos permiten dar un nombre más corto a un objeto al que hagamos referencia a lo largo de la plantilla a través de la directiva `@let`, como se puede ver en la imagen siguiente:

```

@let categorias : Record<number, string> | undefined = producto?.categorias;
@for (cat of categorias| keyvalue; track cat.key) {
  <p class="text-primary m-0"> {{ cat.value | uppercase }}</p>
}
  
```

Es factible crear nuestras propias tuberías para personalizar el funcionamiento de nuestra aplicación:

ng generate pipe sort

Deberemos implementar el método **transform** que recoge un valor a transformar (`value`) y un conjunto optativo de parámetros (`...arg`), se tiene que devolver un valor transformado del mismo tipo.

```

export class SortPipe implements PipeTransform {
  transform(value: Producto[], args: unknown = 'nombre'): Producto[] {
    if (!value || !Array.isArray(value)) { return value; }
    return value.sort(
      (a, b) => {
        if (a.nombre < b.nombre) { return -1; }
        if (b.nombre < a.nombre) { return 1; }
        return 0;
      }
    );
  }
}
  
```

```

    if (a.nombre > b.nombre) { return 1; }
    return 0;
}
);
}
}

```

Una vez definida, se importará en el componente correspondiente y se usará como cualquier tubería

```
@for (producto of productos|sort; track producto.id; let i = $index) {
```

Existe la posibilidad de añadir diferentes parámetros a nuestra tubería, se deja como actividad al lector.

Es importante tener en cuenta que las tuberías solo se ejecutan si se cambia la referencia del objeto global, es decir si se crea de nuevo. Si se modifica una propiedad, o se usan métodos, no se relanzará la tubería. Así para una lista, el código para que se aplique siempre la tubería sería algo similar a:

```
this.productos = [ ...this.productos,
{ defincion de nuevo producto }];
```

En vez de:

```
this.push({ definición de nuevo producto });
```

[El código de ayuda anterior se puede ver en cap04.](#)

Para terminar este punto hay que comentar, referente a la implementación de una tubería, dos clases: puras e impuras que tienen incidencia directa en el rendimiento de la aplicación. Las directivas puras su método de transformación solo se ejecuta cuando Angular detecta un cambio en un valor de entrada o de uno de los parámetros (Cuando cambia el valor o cuando cambia la referencia de un valor, pero no cuando cambiar la propiedad de un objeto sin cambiar la referencia al objeto). Por otro lado, las directivas impuras se ejecutan en cada ciclo de detección de cambio, es decir siempre, por lo que la ejecución de su método de transformación debe ser lo más rápido posible.

Directivas

<https://angular.dev/guide/directives>

Existen tres tipos de directivas: de componentes, de atributos y estructurales. Las primeras se han desarrollado en el punto de componentes.

Las directivas de atributos en Angular modifican el comportamiento o la apariencia de un elemento estándar HTML. Existen varias directivas predefinidas de este tipo que vamos a explicarlas:

- **NgClass.** Añade o elimina un conjunto de clases CSS a la etiqueta HTML.
- **NgStyle.** Añade o elimina un conjunto de estilos a la etiqueta HTML.
- **NgModel.** Añade comunicación bidireccional entre el componente y la etiqueta HTML, generalmente del tipo **input**. Esta directiva ya la hemos utilizado antes.

```
<div [ngClass]="isSpecial ? 'special' : "">This div is special</div>
<div [ngStyle]="currentStyles">
<input [(ngModel)]="currentItem.name" id="example-ngModel">
```

Existe la posibilidad de definir nuestras propias directivas para aumentar la funcionalidad y realizar un código más claro, se recomienda visitar la documentación de Angular para ampliar este punto.

<https://angular.dev/guide/directives/attribute-directives>.

Por último, las directivas estructurales se usan dentro de las plantillas y están siendo sustituidas por la notación @if, @for, @switch, por lo que no es recomendable su uso a no ser que sea por mantenimiento de la aplicación.

Práctica Ángular 2

Servicios

[El código de ayuda se puede ver en cap05.](#)

La arquitectura que presenta Angular de componentes hace que la comunicación entre ellos sea una tarea vital. Ya hemos abordado la comunicación Padre – hijo en ambas direcciones, pero falta adentrarse en la comunicación entre dos componentes no relacionados directamente y de un componente con proveedores de datos. También existe la comunicación de un componente con recursos externos, pero este supuesto se deja para más adelante.

El mecanismo que crea Angular para este tipo de comunicación se llama servicio, y la manera de uso dentro de un componente es a través del patrón de programación de Inyección de dependencias (DI) (¿Nos acordamos de los principios SOLID?).

Cuando una aplicación crece y evoluciona, el código interno necesita que unos componentes usen otros sin estar estos relacionados directamente, esto es lo que se conoce en el mundo de la programación como una **dependencia**. El paso esta dependencia al objeto que la necesita, al consumidor, a través del constructor o cualquier otro método es lo que se llama **inyección**, que conlleva la creación de clases de código que generan las dependencias y la pasan al consumidor en el momento de su creación, el **injectedor**. Por tanto, un inyector es responsable de crear e inicializar las dependencias necesarias para que estén disponibles para ser insertadas en los consumidores. El objeto consumidor no tiene que saber nada de cómo crearlas, simplemente debe conocer el interfaz de la dependencia para usarlo cuando sea necesario. En Angular, el concepto de dependencia se implementa bajo servicios. Y cada servicio debe ser responsable, tal y como dictan las reglas de código limpio, de un único dominio o contexto empresarial, si nos encontramos que abarca más, deberemos pensar en dividirlo en varios servicios.

`ng generate service productos`

Para verlo claro, vamos a crear un servicio que se encargue de crear los datos iniciales de un controlador. Generalmente estos datos se cargarán desde un servidor de Bases de Datos, pero en nuestro ejemplo se generarán de forma estática.

Un servicio es una clase que está marcada como `@Injectable`. Además, se debe decidir qué inyector usar, por defecto se utiliza el global de la aplicación (`root`), que usa un patrón de programación Singleton sobre cada servicio, lo que hará este servicio disponible para cualquier componente en cualquier lugar y de forma única, simplemente inyectándolo en su código (ver más adelante), además en el proceso de llevar a producción, puede detectar servicios no usados y eliminarlos de código final.

Cada **servicio** debe declarar el interfaz que se puede usar con él, creando tantos métodos como necesitemos, nosotros definiremos solo uno para recoger los productos iniciales que antes se creaba en el constructor del componente: `getProductos`.

```
@Injectable({ Show usages
  providedIn: 'root'
})
export class ProductosService {
  constructor() { } no usages
  getProductos(): Producto[] { Show usages
    return [
      { id: 1, nombre: 'Producto Gen 11', precio: 10.99, categorias: {1: 'Cat 1' , 2:'Cat 2'} },
      { id: 2, nombre: 'Producto Gen 2', precio: 20.99, categorias: {1: 'Cat 1' , 2:'Cat 2'} },
      { id: 3, nombre: 'Producto Gen 13', precio: 30.99, categorias: {2: 'Cat 2' , 3:'Cat 3'} },
      { id: 4, nombre: 'Producto Gen 4', precio: 40.99, categorias: {2: 'Cat 3' , 3:'Cat 3'} },
      { id: 5, nombre: 'Producto Gen 5', precio: 50.99, categorias: {2:'Cat 2'} }
    ];
  }
}
```

El servicio lo primero que define es la clase que actuará como inyector, en este caso `root` (recordar que este inyector crea un único servicio global a todos los componentes), definido en la propiedad `providedIn` del decorador `@Injectable`, para a continuación definir el interfaz.

El siguiente paso, es hacer uso del servicio desde el componente y para ello el consumidor tiene que tener conocimiento del servicio, es decir se le tiene que inyectar el servicio en él. Se puede hacer de dos maneras, vamos a utilizar la más simple a través de su constructor.

```
productos: Producto[] = [];

constructor(private servicioProductos: ProductosService ) {} no usages

ngOnInit(): void { no usages
  this.productos = this.servicioProductos.getProductos();
}
```

En el componente se inyecta automáticamente a través de su constructor por medio del inyector root de la aplicación Angular que es global a todos, cuando se crea. Una vez inyectado, se define la propiedad

privada que lo albergará (**servicioProductos**). Cuando este componente es creado, pero todavía no se ha visualizado, cargamos los datos de la lista (evento **ngOnInit**).

De esta manera podemos usar un servicio de datos desde un componente.

Existe otra manera de injectar el servicio en el componente, es mediante la palabra **inject**. Se definiría una propiedad privada de la siguiente manera y se eliminaría del constructor:

```
private servicioProductos : ProductosService = inject(ProductosService);
// constructor(private servicioProductos: ProductosService ) {}
```

Ambas aproximaciones son similares y queda a elección del programador elegir la que más le conviene.

Tratamiento de las transmisiones asíncronas

El código de ayuda se puede ver en cap06.

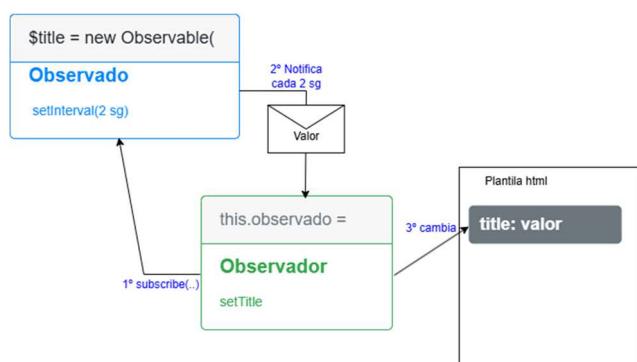
El tratamiento de la información asíncrona en una aplicación JavaScript es una tarea muy común. El paradigma de programación reactiva nos ayuda a consumir, gestionar y transformar información asíncrona a través de flujos de datos. La gestión se realiza a través del patrón Observador/Observado.

Anteriormente a este mecanismo, se han utilizado las Callbacks de eventos y las promesas, pero ambas soluciones tienen varios problemas de gestión importantes, entre los que se encuentran la imposibilidad de parar una petición lanzada o hacer una carga diferida de los datos, por lo que se usa el patrón comentado.

Angular, utiliza el patrón Observador/Observado mediante el mecanismo de **Observable/ observer**. Un Observable es un objeto que mantiene un conjunto de dependencias, denominadas observadores (**observer**) a los que informará de cualquier cambio en el flujo controlado por él, mediante eventos que pueden ser lanzados o cancelados en cualquier momento independientemente de que se hayan emitido ya los datos o no. Es obligatorio que el observador (**observer**) se suscriba (**subscribe**) al observado para poder reaccionar a la notificación.

`<p [innerText] = "title" > observador works! </p>`

En el ejemplo de la imagen siguiente vemos que dentro del mismo componente creamos un Observado



(**\$title** de tipo Observable) que cada dos segundos (setInterval) notificará a todos sus suscriptores (observadores) el cambio y le pasará el valor actual. El observador, en este caso el mismo componente, en el constructor se suscribe, lanzando **setTitle** cada vez que el observador lo notifica.

```

title:string = 'Hello';
private observado: Subscription;

title$ : Observable<unknown> = new Observable(
  observador : Subscriber<unknown> => {
    let i : number = 0;
    setInterval(() : void => {
      i++;
      observador.next("Hello : " + String(i));
    },
    delay: 2000);
  }
);

constructor(){ no usages new *
  this.observado = this.title$.subscribe(params : unknown => {
    this.setTitle(String(params));
  });
}

```

Los observadores seguirán mandando eventos mientras tengan registrado algún observador por lo que, si no eliminamos dicha asociación, aunque el componente sea destruido seguirá ocupando recursos del sistema. Existen varios mecanismos para eliminar la suscripción, pero vamos a utilizar el más básico, podemos obtener más información en la documentación oficial.

```

ngOnDestroy(): void { no usages new *
  this.observado.unsubscribe();
}

```

Para la conclusión de la suscripción, en el destructor del componente llamaremos al método **unsubscribe** sobre la suscripción. Evidentemente, antes deberemos guardarnos la referencia de dicha suscripción en una variable, en este caso **observado** durante la suscripción en el constructor.

Comunicación entre componentes no relacionados

En la práctica dos, no pudimos realizar varios aspectos de la aplicación porque estaban involucrados componentes que no tenían una relación de herencia directa. Para solventar este caso vamos a crear un servicio, usando el patrón de programación Observador/Observado anterior.

PRIMERO CREAMOS EL SERVICIO (OBSERVABLE)

El servicio se creará en la parte global de la aplicación, creándose solo una vez y manteniendo los datos sincronizados entre todos los componentes. Se implementa como una clase observable (Observado del patrón) que el resto de componentes usarán mediante una dependencia inyectada en el constructor de cada componente. Este servicio añade a su interfaz el método **push_carrito** que se utilizará para mandar los datos desde el componente origen al destino, en este caso no se implementa un almacén en el servicio.

```

@Injectable({ Show usages
  providedIn: 'root',
})
export class CarritoService {
  private addCarritoSubject : Subject<Producto> = new Subject<Producto>()
  addCarritoObservable : Observable<Producto> = this.addCarritoSubject.asObservable();
  constructor() { } no usages
  push_carrito(producto: Producto) : void { Show usages
    this.addCarritoSubject.next(producto);
  }
}

```

SEGUNDO DEFINIMOS EL OBSERVADOR (RECEPTOR)

El componente destino de los datos, se suscribe (`subscribe(..=>)`) al servicio en la inicialización del mismo (`ngOnInit`), previamente se ha inyectado en el constructor el servicio (`dataService: CarritoService`). Cada vez que sea notificado por el observad (el servidor) se añaden los datos a array correspondiente mediante la llamada `add_producto`.

```
constructor(private dataService: CarritoService) { } no usages
private add_producto(producto: Producto) : void { Show usages
  this.elementosDelCarrito.push(producto);
}
ngOnInit() : void { no usages
  this.dataService.addCarritoObservable.subscribe(
    producto : Producto => {
      let producto_copia: Producto = {...producto};
      producto_copia.esta_en_cesta = true;
      this.add_producto(producto_copia);
    }
);}
```

ULTIMO MODIFICAMOS EL COMPONENTE GENERADOR DEL EVENTO

El componente generador de todo el proceso debe implementar la llamada al interfaz del servidor (`push_carrito`), previa inyección del servicio, en este caso en el constructor también (`dataService_carrito: CarritoService`).

```
constructor(private dataService_carrito: CarritoService) { } no usages

clicEnviar(producto: Producto) : void { Show usages
  const maxId : number = Math.max(...this.elementos.map(e : Producto => e.id));
  //this.on_enviar.emit(producto);
  this.dataService_carrito.push_carrito(producto);
}
```

El evento generador (`clicEnviar`) se genera al pulsar en un elemento de la lista tal y como se ve en la plantilla generada por el código siguiente:

```
@for (elemento of elementos; track elemento.id; let i = $index) {
<li class="list-group-item">
  <app-elemento [producto]="elemento" (addtocart)="clicEnviar($event)"></app-elemento>
</li>
```

En resumen, la comunicación que se da entre dos componentes a través de un servicio, anteriormente implementada, sería la que vemos en la siguiente imagen.

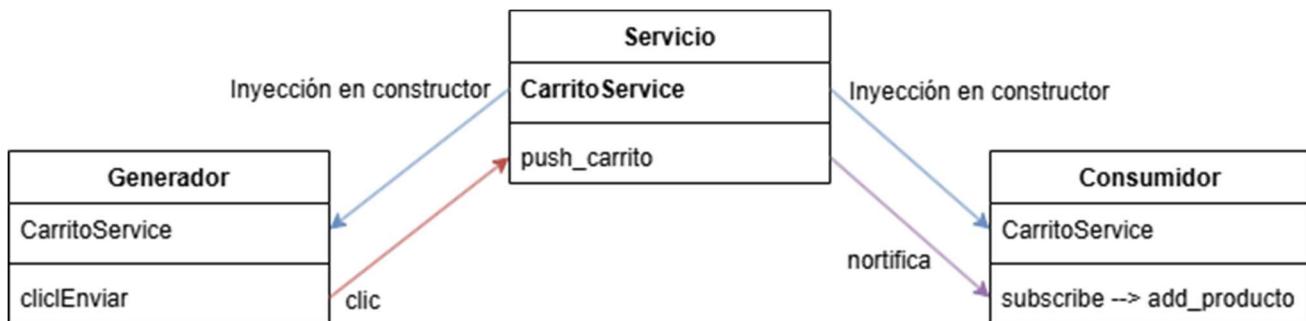


Ilustración 19 - Comunicación entre componentes

Práctica Ángular 3

Operadores para la comunicación asíncrona

Angular nos proporciona varios operadores para realizar de forma más rápida algunas tareas:

- **fromEvent**. Crea un observable a partir de un elemento del DOM. Para esta acción hay que saber que, para conseguir acceso a un elemento del DOM desde el controlador, hay que crear una variable de plantilla (#nombre) y a través de la propiedad **nativeElement** podremos enlazarlo. Para recoger desde el controlador la variable de plantilla se usa la función **viewChild** que devuelve un objeto de la clase **ElementRef** (Este servicio proporciona una referencia directa al elemento contenedor del DOM).
- **pipe**. Este operador permite que se realicen en el mismo observador múltiples operaciones, una a continuación de otra.
- **tap**. Se utiliza cuando queremos hacer alguna tarea con los datos sin modificar los originales.
- **map**. Es similar al método map de los Arrays y devuelve un observable con una versión modificada de los datos recibidos.
- **toSignal**. No es un operador asíncrono, pero permite en una comunicación asíncrona emitir un valor como una señal.

Realizar un trabajo que investigue cómo aplicar estos operadores

Señales

Las señales en Angular es una aproximación síncrona a la programación reactiva. Vimos en los puntos anteriores que el patrón Observador/ Observado eran asíncronos. Angular hace uso muy intensamente de una librería para detectar cambios en los controladores y HTML para enviar eventos de sincronización al núcleo y mantenerlos sincronizados. Esta tarea es computacionalmente muy pesada y los programadores tratan de aminorar su efecto todo lo que pueden. Las señales, es un intento de mitigar es carga todo lo posible, de minimizar el proceso de detección de cambios en la aplicación. Las señales vigilan cuando el estado de la aplicación cambia y permiten al Framework reaccionar ante él solo en aquellas partes en las que es necesario, no en toda la aplicación. Las señales actúan como contenedores de datos en los que hay que detectar un cambio, notificándolo en caso que se produzca. El avance es significativo, en vez de rastrear los cambios en toda la aplicación, se hace exclusivamente en aquellas partes en las que el programador espera que se realicen cambios.

El código de ayuda se puede ver en cap07.

Creación de señales

```
seniales.component.ts
 currentDate: WritableSignal<Date> = signal(new Date());
 
setTitle(title: string) : void { Show usages
  this.currentDate.set(new Date());
}

<p>seniales works!: {{ currentDate() }}</p>
```

anterior, para actualizar la señal cada dos segundos mediante **set**, pero se podría hacer también desde cualquier otro evento de la plantilla o cualquier mecanismo válido. Como podemos ver en el ejemplo, el mayor cambio es la necesidad de paréntesis en el uso en la plantilla.

Este mecanismo lo hemos ya utilizado cuando vimos los mecanismos de comunicación entre padres – hijo, en concreto las funciones **input** y **output** definen dos señales, una de entrada y otra de salida.

Si deseamos convertir una señal en un Observable, tendremos que usar la función **toObservable()**.

Es recomendable usar señales en vez de propiedades en los componentes para mantener el estado del mismo.

Señales calculadas

Las señales pueden depender de otras señales, tanto generales como computadas, pero las señales creadas de este modo son de solo lectura, no se pueden modificar directamente, solo a través de las otras señales. La definición usa el tipo **Signal**, a la vez de la función **signal** y la función **computed** en el momento de la definición.

```
title: Signal<string> = signal();
...
this.title = computed( () =>{ return "Fecha: " + this.currentDate() } );
```

Comunicación con servicios externos

La comunicación de Angular con los clientes HTTP implementados se realiza como hemos visto en el punto de servicios, a través de flujos observables. Si bien, podemos utilizar el API existente en JavaScript como es `fetch` o `HTTPRequest`, ya vimos que tenían varios problemas graves que no incluye el API nativo de Angular. Si necesitamos usar el API `fetch`, podemos incluirlo dentro de un Observable y minimizar los problemas que existen.

[El código de ayuda se puede ver en cap08.](#)

```
const request$ = new Observable(observer => {
```

```

fetch(url)
  .then(
    result => {
      if (result) {
        observer.next(result);
        observer.complete();
      }
      else{
        observer.error(' Error de conexión');
      }
    }
  );
});

```

Como hemos dicho el cliente http propio de Angular es la mejor opción. Es una librería separada que se instalada por defecto, con todas las utilidades necesarias para realizar y gestionar una conexión con el servidor externo.

Para poder utilizar el cliente http deberemos importar en nuestro código la clase **provideHttpClient**. Esta clase expone diferentes servicios para manejar llamadas asíncronas a servidores externos, en concreto existen: **get**, **post**, **put**, **patch** y **delete** para cada uno de los verbos HTTP. Todos estos métodos devuelven un flujo observable al que nos podemos suscribir como hemos visto en puntos anteriores (de momento el cliente no soporta señales para la gestión). Veamos un ejemplo práctico que se ejecuta sobre un servidor Graphql situado en la url: <http://localhost/public/graphql>.

Creación del componente e inclusión en el componente principal

```
C:\Herd\ang_pr_01>ng generate component cap08/http_read
```

```
<div class="right-side">
  <app-http-read></app-http-read>
</div>
```

Es necesario configurar el fichero **app.config.ts** para indicar que use el **provideHttpClient**.

```
export const appConfig: ApplicationConfig = { Show usages csanjuap-ies *
  providers: [provideZoneChangeDetection({ eventCoalescing: true }), provideRouter(routes), provideHttpClient()]
}
```

Programación del componente

A continuación, escribiremos la lógica del componente, en este caso una única propiedad (**title**) será la que recibirá los datos y posteriormente se muestre en la plantilla.

Creamos un observador a partir del método **getData** en el que se configura el cliente HTTP con las opciones correctas, la Query GraphQL y la llamada el servidor con el método **post**. Este método **getData** devuelve el observador que es creado en el constructor y suscrito como en ejemplos anteriores, llamando al método **setTitle** cuando se complete la transferencia de datos con el servidor.

```

export class HttpReadComponent implements OnDestroy {
  URL : string = 'http://localhost/public/graphql';
  private observado: Subscription;
  title: string = '';

  constructor(private http: HttpClient) { no usages
    this.observado = this.getData().subscribe(params : any => {
      this.setTitle(JSON.stringify(params));
    });
  }

  getData(): Observable<any> { Show usages
    const headers : HttpHeaders = new HttpHeaders().set( name: 'Authorization', value: 'Bearer token_value');
    // const post_params = { query : "query { login(email:\"abc@abc\", password:\"abc\")}" };
    const post_params = { query : "query { logout }" };
    return this.http.post(this.URL, post_params, {headers: headers});
  }

  setTitle(title: string) : void { Show usages
    this.title = title;
  }

  ngOnDestroy() : void { ... }
}

```

Este ejemplo carga los datos solo en el proceso de creación, se puede modificar para que se gestione cuando la aplicación lo necesite.

Modificación de la plantilla

```
<p [innerText]="title">http-read works!</p>
```

Cabeceras y paso de parámetros HTTP

Los métodos mostrados utilizan para realizar su función parámetros HTTP y cabeceras HTTP. Ambos se establecen en la llamada al método correspondiente, en este ejemplo **post**. En la forma más simple, los parámetros a enviar se pasarán en el segundo parámetro como cadena o como objeto JSON, el tipo MIME se establece de forma automática en función del tipo de este parámetro (`post_params`).

Las cabeceras se definir establecer en el tercer parámetro, con otro objeto cuya clave sea **headers** y el valor un objeto del tipo `HttpHeaders`. Este admite el método **set** para fijar todas las cabeceras necesarias, ver ejemplo. Este tipo es inmutable, lo que hace que debamos encadenar una secuencia de varias llamadas al método `set` para establecer varias cabeceras, una llamada por cabecera.

```

const headers= new HttpHeaders()
  .set('content-type', 'application/json')
  .set('Access-Control-Allow-Origin', '*');

```

Hemos comentado en el párrafo anterior que también hay que establecer los parámetros y hemos hablado de una cadena de texto o un objeto JSON, pero también existe la posibilidad de crear un objeto

del tipo `HttpParams` de forma similar a las cabeceras y pasarlo mediante un diccionario cuya clave sea `params`.

```
const params= new HttpParams()
.set('limit', '10')
.set('password', 'clave');
```

Interceptores

Un interceptor HTTP es un servicio Angular que intercepta una petición HTTP, realiza alguna tarea sobre ella y la pasa al cliente HTTP para que la gestione. Por ejemplo, supongamos que hay que mandar siempre una cabecera de autentificación con un token conseguido por la aplicación. En este caso, podríamos crear un interceptor para que inyectara en cada petición del cliente la cabecera de autorización con el correspondiente token. Creamos el interceptor con:

```
ng generate interceptor nombre_int
```

configuramos la aplicación para que lo use en el fichero `app.config.ts`, en la sección que hemos añadido el proveedor del cliente `http`, ahora le pasamos un parámetro, tal y como se ve a continuación:

```
provideHttpClient(withInterceptors([nombre_int])),
```

Creamos el código del interceptor, para que añada la cabecera de autentificación

```
export const nombre_intInterceptor: HttpInterceptorFn = (req, next) => {
  const authReq = req.clone({setHeaders: {Authorization: 'token_almacenado'}});
  return next(authReq);
};
```

Los interceptores, pueden hacer uso de dependencias externas mediante `inject`, con lo que, si hemos creado un servicio de autorización que se conecta a un servidor, y guarda el token, podría capturarlo de este servicio.

```
const authSevice = inject(AuthService);
const authReq = req.clone({setHeaders: {Authorization: authService.getToken()}});
```

Rutas

Desarrollar una aplicación de página única, no significa que no tenga diferentes áreas de trabajo. Para proporcionar una experiencia de navegación mejorada, en Angular se utilizan las rutas. Son un mecanismo de cambiar de forma intuitiva la vista de la aplicación. similar al utilizado en el lado servidor.

[El código de ayuda se puede ver en cap09.](#)

Las aplicaciones web modernas usan JavaScript para realizar los cambios entre vistas o componentes minimizando la comunicación con el servidor. En el primer contacto se trae el esqueleto de la aplicación con la mayoría del HTML, el router de JavaScript interceptará el resto de llamadas del usuario para realizar los cambios en el interfaz subyacente con la mínima intervención del servidor, se le solicitarán exclusivamente los datos necesarios y generalmente en formato JSON sin necesidad el HTML. Será JavaScript el que, con esos datos, acomodará el interfaz como sea necesario. Este tipo de metodología se

llama aplicación de página única o SPA. El enrutado en Angular está habilitado por defecto, solo hay que configurarlo. El router es capaz de navegar entre diferentes componentes, en concreto se encarga de:

- Especificar la raíz de las rutas, o ruta base, generalmente: /.
- Usar las directivas necesarias para el funcionamiento.
- Configurar las rutas necesarias para la visualización.
- Decidir cuáles y dónde presentar los componentes en función de la ruta.

Los navegadores actuales son capaces de manejar el cambio de una URL y su historial cuando se navega sin necesidad de mandar peticiones a un servidor, esta técnica se denomina pushState y se implementó bajo el estándar HTML 5. Angular necesita una directiva en toda página HTML para poder usarlo, se incorpora en la cabecera de la página:

```
<base href="/">
```

Podemos abrir el código de cualquier aplicación Angular y comprobar que está allí. Solo es importante tenerla en cuenta si nuestra aplicación **no va a residir en el directorio raíz**, ya que deberemos cambiar a la ruta en la que esté la aplicación en el momento del despliegue.

Configuración de rutas

El fichero **app.routes.ts** contiene la lista de rutas y componentes que serán mostrados cuando se navegue hasta ella. Cada línea marca una ruta diferente, indicando en la propiedad **path** la ruta relativa desde la base, y la propiedad **component** el componente a mostrar, también es posible añadir una propiedad **title** que indique la sección de la aplicación.

Nota: La navegación web se puede producir de forma automática a través de etiqueta y sus propiedades HTML o de forma manual porque el usuario la introduzca en la barra de navegación.

Una vez definidas las rutas, hay que establecer en qué parte de la página principal aparecerá el componente indicado. Tendremos que determinar dentro de la plantilla HTML de la aplicación (**app.component.html**) una etiqueta que indique dicho lugar. La manera que tenemos en Angular de establecer esa posición es a través de la etiqueta **<router-outlet>**.

Pasemos a crear un pequeño ejemplo. Primero configuraremos las rutas en el fichero **app.routes.ts**, evidentemente antes habremos creado los componentes correspondientes.

```
<div class="right-side">
  <router-outlet></router-outlet>
</div>
```

```
import { Routes } from '@angular/router';
import { Ruta01Component } from './cap09/ruta-01/ruta-01.component';
import { Ruta02Component } from './cap09/ruta-02/ruta-02.component';

export const routes: Routes = [ Show usages new *
  {path: 'ruta_1', component: Ruta01Component },
  {path: 'ruta_2', component: Ruta02Component },
]
```

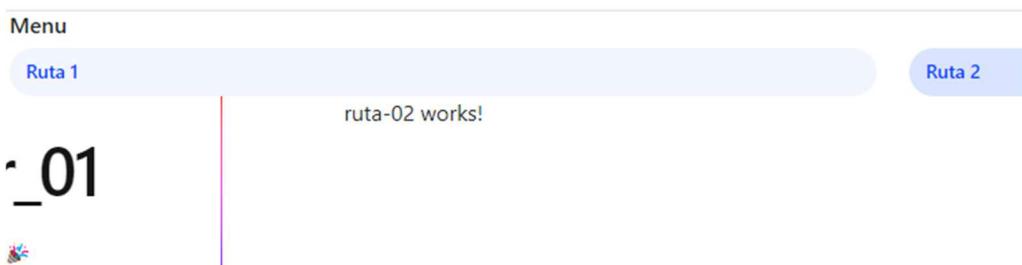
A continuación, habilitamos la

aplicación para que enrute en el fichero **app.component.ts** añadiendo las dependencias necesarias a la sección **imports** (RouterLink, RouterOutlet).

Para terminar, hay que definir la posición en la que se insertará el correspondiente componente en la plantilla y los enlaces para el usuario, en este caso con un menú en la parte superior (app.component.html).

| | |
|---|--|
| app.component.ts <pre>import { Component } from '@angular/core'; import {RouterLink, RouterOutlet} from '@angular/router'; @Component({ Show usages ↳ csanjuap-ies * selector: 'app-root', imports: [RouterOutlet,RouterLink], templateUrl: './app.component.html', styleUrls: ['./app.component.css' }) export class AppComponent {</pre> | app.component.html <pre><div class="container"> <div class="row pill-group"><h6>Menu</h6></div> <div class="row"> <div class="col col-6"> Ruta 1 </div> <div class="col col-6"> Ruta 2 </div> </div> <div class="content"> <div class="left-side"> <h1>Hello, {{ title }}</h1> <p>Congratulations! Your app is running. 🎉</p> </div> <div class="divider" role="separator" aria-label="Divider"></div> <div class="right-side"> <router-outlet /> </div> </div></pre> |
|---|--|

El resultado que obtendremos será el siguiente:



El valor de la directiva **routerLink** utilizada en la etiqueta **<a>** de la plantilla genera el atributo **href** a la ruta de forma correcta, en función de la jerarquía establecida en los componentes de la aplicación (en la plantilla del componente app no tiene un efecto evidente). La ruta es relativa al componente en el que se encuentra definido, así si en una plantilla de un componente hijo la definimos a *ruta_hijo*, la ruta real que genera esta directiva será: *ruta_padre/ruta_hijo*. Para rutas más complejas, especialmente con parámetros, la directiva puede aceptar un Array de valores, en el que el primer elemento es la ruta y el resto los parámetros.

Ejercicio: Añadir dos nuevas rutas para dos hijos del componente 2 y su lógica

Podemos redirigir la aplicación en cualquier momento a otra ruta con el método **navigate** del router, pasándole la url a la que queremos dirigir la aplicación y un parámetro opcional de extras (ver documentación). Similar a navigate existe el método **navigateByUrl** que recoge una url completa a la que dirigirnos como primer parámetro. Vamos hacer que los dos hijos se puedan enlazar entre sí con un botón.

Primero crearemos dos eventos **clic** en el controlador de cada hijo para que desplaza a la url del otro. Inyectaremos el router en el constructor a través de una propiedad privada (**router**) para poder hacer uso.

El siguiente paso, es modificar la plantilla con un botón que llame a dicho evento. Se deja como trabajo para el alumno. El resultado es que al pulsar en el botón se mostrará el componente deseado.

```
export class Hijo1Component {
  constructor(private router: Router) {} no usages
  onClick(): void { no usages
    this.router.navigate(['/ruta_2/hijo_2']);
  }
}
```

hijo-2 works!

[Ir hermano](#)

[Hijo 1](#)

[Hijo 2](#)

No se abordado el tema de las rutas auxiliares por falta de tiempo, se recomienda al lector su consulta en la documentación oficial.

Orden de las rutas

Las rutas definen los puntos de entrada, pero al ver la definición, es fácil darse cuenta que su orden es muy importante, ya que se buscará la coincidencia de arriba abajo. Una vez considerado esto, es muy importante definir una ruta por defecto, además de estar situada la última, por la que entrarán todas las URL que no coincidan, evitando mensajes de ruta no existente.

La ruta por defecto se configura con una ruta más en el fichero **app.routes.ts** en la que la componente **path** está vacía, se establece la propiedad **pathMatch** a **full** y una redirección a la ruta que nos interese. Si bien no es obligatoria dicha redirección, pudiendo utilizar un componente propio, es el comportamiento por defecto.

Hay que saber que, de forma predeterminada, el router verifica los elementos de URL desde la izquierda para ver si la URL coincide con una ruta determinada y se detiene cuando hay una coincidencia. La estrategia de coincidencia de ruta "**full**" hace coincidir con toda la URL. Es importante hacer esto cuando se redirigen desde rutas vacías. De lo contrario, debido a que una ruta vacía es un prefijo de cualquier URL, el router aplicaría la redirección incluso cuando se navegue hacia el destino de la redirección, creando un bucle sin fin.

La propiedad **pathMatch** configura cómo controlar la coincidencia en la ruta. En el ejemplo, le indica al router que solo navegue a la **ruta_1** cuando corresponda con la ruta raíz de la aplicación (""), no en el resto de rutas que se desplazará a la dirección normal.

```
{path: 'ruta_2/hijo_1', component: Hijo1Component },
{path: 'ruta_2/hijo_2', component: Hijo2Component },
{path: '', redirectTo: '/ruta_1', pathMatch: 'full' },
{path: '**', redirectTo: '/ruta_2'},
```

También podemos hacer que todas las rutas excepto las anteriores definidas se dirijan o enruten a otro sitio con el patrón ****** en la propiedad **path** de la definición, como se puede ver en la última línea del ejemplo.

Estilo en las rutas

Podemos determinar qué ruta es la activa y aplicarle un estilo propio mediante la propiedad **routerLinkActive** de HTML en la plantilla que definamos el menú. Antes hay que importarlo en el fichero .ts correspondiente.

```
<a routerLink="/ruta_1" class="pill" routerLinkActive="bg-black">Ruta 1</a>
</div>
<div class="col col-6">
  <a routerLink="/ruta_2" class="pill" routerLinkActive="bg-black">Ruta 2</a>
```



Aceptará un conjunto de clases separadas por blancos que queramos aplicar, también de Bootstrap.

Paso de parámetros en las rutas

El diseño de URLs en las que se pasan los parámetros es similar a los Frameworks servidor y a lo visto anteriormente, hay que introducir unos pequeños ajustes para que al pulsar en cada URL se refleje el cambio, a través de un Observable.

Crearemos el ejemplo anterior, en el que una Ruta_3 mostrará un componente lista, que a su vez tiene componentes detalle. Al pulsar en cada enlace, se cambiará a su propia ruta del estilo ruta_3/numero, siendo este número el id del producto a mostrar en la parte inferior.

ruta-03 works!!

Lista de Productos

Lista de productos con 5

| |
|------------|
| Producto 1 |
| Producto 2 |
| Producto 3 |
| Producto 4 |
| Producto 5 |

Producto 5

Cada uno es su propia plantilla de detalles

⚠ Precio: 50.99

Plantilla de la Ruta_3

Plantilla del componente: lista

Primero establecemos las rutas con el patrón `:id` al final para indicar que es un parámetro (los dos puntos indican que es un parámetro, el nombre se puede elegir el que deseemos). La ruta para este ejercicio se puede crear de forma directa como hasta ahora, pero vamos a mostrar cómo utilizar rutas hijas (muy útil cuando el número de ruta crece y están relacionadas). Al definir una ruta como hija evitamos el prefijo del parent, que se añadirá de forma automática. En el fichero `routes.app.ts` (vemos en la imagen siguiente cómo sería de forma simple en el comentario):

```
// {path: 'ruta_3/:id', component: DetallesProductoComponent },
{
  path: 'ruta_3',
  component: Ruta03Component,
  children: [
    {path: ':id', component: DetallesProductoComponent },
  ]
},
```

Añadimos el enlace en el menú, `app.component.html`:

```
<div class="col col-4">
  <a routerLink="/ruta_3" class="pill" routerLinkActive="bg-success">Ruta 3</a>
</div>
```

En la plantilla del componente `Ruta_3 (ruta-03.component.html)` Añadimos el selector de la lista: `app-lista-productos`.

```
<app-lista-productos></app-lista-productos>
```

Modificamos el componente de la lista para que incluya elementos `<a>` y `` (`lista-productos.component.html`) en vez de la etiqueta del componente, también añadimos el parámetro para que cree la ruta correctamente con `[routerLink]="[producto.id]"`. El fichero `lista-producto.component.html`:

```
<ul class="list-group">
  @for (producto of productos; track producto.id; let i = $index) {
    <li class="list-group-item" (click)="seleccionarProducto(producto)">
      <a [routerLink]=[producto.id]><span>{{ producto?.nombre }}</span></a>
    </li>
  }
</ul>
}
@else {
  <p>No hay productos disponibles.</p>
}
<outer-outlet />
```

Esto hará que el atributo `href` de la etiqueta `<a>` apunte a `ruta_03/un_numero_de_id`, parámetro que posteriormente recogeremos en el componente hijo cada vez que pulsemos. Esta plantilla, con este

método, también debe incluir una etiqueta `<router-outlet>` en donde aparecerá la parte del hijo, al igual que el componente root en su plantilla.

Añadimos un método al componente de lista, que al pasarle un id, nos devuelva el correspondiente producto. Esto lo hacemos porque no hemos creado un servicio de carga que sería lo suyo y los datos están generados en HardCode en este componente, no se debe usar nunca en una aplicación este método, solo por rapidez (El método `findProducto` así como los datos deberían formar parte de un servicio y usarse inyectado). Este método lo usará el componente de detalle para conseguir los datos.

```
findProducto(id: number): Producto|undefined { Show usages
  return this.productos.find(p : Producto => p.id === id) ;
}
```

Con esto hemos terminado la parte de la lista, se configurará ahora el hijo (el detalle del producto). Lo primero es simplificar la plantilla para que muestre los datos que queremos en `detalles-productos.component.html`:

```
<h3>{{ producto?.nombre }}</h3>
<p>@switch (producto?.id) {
  @case (1){ ⏺ }
  @case (2){ ⏲ }
  @default { ⏴ }
}
Precio: {{ producto?.precio }}</p>
```

Para terminar el componente de detalles, como se crea un solo componente y los datos van a cambiar cada vez que demos en un producto, es necesario crear un observable que notifique este cambio de ruta al componente detalles. Lo inicializaremos en el método `ngOnInit` de Angular. También nos suscribimos a este observable desde el mismo componente, para que podamos establecer el valor de la propiedad que usamos en la plantilla.

El constructor inserta tres valores, la lista de datos para poder conseguir el producto, esto está mal y se hace por rapidez, debería ser un servicio diferenciado. La ruta actual, de la que podremos conseguir el identificador que nos han pasado y el router completo por si lo necesitamos. Para conseguir el identificador que viene en la ruta se utiliza el `params.get`, y como estamos tratando con observables, no se puede devolver un producto, hay que convertirlo a ello con `of`, esta instrucción convierte cualquier valor en un observable, que es lo que nos interesa (`producto$`).

A continuación, nos suscribimos para recibir los cambios y modificamos la propiedad cuando este se produzca, en el momento que seleccionemos otra url y la aplicación cambie.

```
export class DetallesProductoComponent implements OnInit {
  producto$: Observable<Producto>|undefined;
  producto: Producto | null = null;

  constructor(private lista: ListaProductosComponent, private route: ActivatedRoute, private router: Router) {}

  ngOnInit(): void { no usages
    this.producto$ = this.route.paramMap.pipe(
      switchMap(params : ParamMap => {
        const producto : Producto = this.lista.findProducto(Number(params.get( name: 'id')))||{} as Producto;
        return of(producto);
      })
    );
    this.producto$.subscribe(params : Producto => {
      this.producto = params;
    });
  }
}
```

Hay que entender el porqué de todo este artificio. Cuando se crea un componente se hace una vez y se queda en el DOM, si no se destruye hay que cambiar los datos de sus propiedades para que se refleje en la plantilla. En este caso, el componente de Detalles solo se crea una vez, desde la plantilla de la lista, por eso la necesidad de un Observable y de modificar los valores de la propiedad a mano nosotros, pero los enlaces se activan cada vez que el enrutado cambia, momento que el observador notifica en el hijo la transición de ruta para actualizar los datos. Si en vez de rutas hijas, tal y como hemos hecho en este ejemplo, fuesen las rutas tradicionales, el componente se crearía y destruiría en cada pulsación por lo que no sería necesario el observable.

Control de acceso a una ruta

Para controlar el acceso a una ruta deberemos crear un guardián de la misma. Será el encargado de determinar si se accede o no a la ruta en un momento dado.

`ng generate guard auth`

El comando anterior no solo creará la estructura necesaria de ficheros, también nos permite elegir cuatro opciones: Controlar cuando una ruta será activada, cuando una ruta hijo se podrá activar, cuando una ruta se podrá desactivar o cuando una ruta se podrá acceder completamente. En este caso seleccionamos la primera opción y abrimos el fichero `.ts` generado.

? Which type of guard would you like

CanActivate

CanActivateChild

CanDeactivate

CanMatch

```
export const authGuard: CanActivateFn = (route : ActivatedRouteSnapshot , state : RouterStateSnapshot ) : UrlTree => {
  // modificar esto con un servicio de autorización
  const router = inject(Router);
  // return true;
  return router.parseUrl( url: '/ruta_2'); //esta la forma más recomendada para la redirección
}
```

Vemos que ha creado una función **CanActivateFn** que deberemos programar. Esta función recoge dos parámetros. El primero es el nombre de la ruta a comprobar, el segundo contiene el estado de la ruta si se produce una navegación correcta. La función devuelve un tipo boolean (verdadero si se puede acceder y falso en caso contrario). También se puede devolver una redirección a la que se navegará de forma incondicional) independientemente la llamada sea síncrona o no. En caso que sea asíncrona, el router espera un Observable o una promesa para resolver. El último paso es configurar las rutas para que lo usen, en el fichero **app.routes.ts**

```
{path: 'ruta_4', component: Ruta02Component, canActivate: [authGuard]},
```

Podemos implementar del mismo modo el resto de las tres opciones, se deja como ejercicio para el lector, sobre todo **CanDeactivate** que permite controlar la salida de una página.

Carga diferida de partes de la aplicación.

Según va creciendo la aplicación es una buena práctica que no se cargue completamente todo el GUI y se retrase hasta que sea necesaria dicha parte. Podemos hacer una carga diferida de una ruta completa tan solo creando un nuevo fichero de rutas para esa parte y usando la función **loadChildren**, veamos cómo. En el fichero **app.routes.ts** apuntamos al nuevo fichero de rutas a incluir:

```
{path: 'ruta_5', loadChildren: () : Promise<typeof import('C:/Herd/ang_pr_01...') => import('./routes/ruta-05.routes') },
```

En el fichero de rutas nuevo, en este caso **routes/ruta-05.routes.ts** exportamos las rutas deseadas:

```
import { Ruta03Component } from '../cap09/ruta-03/ruta-03.component';
export default [{path: '', component: Ruta03Component}]; Show usages
```

Ya estamos cargando todas las rutas de forma diferida, solo cuando se necesiten. La carga diferida no solo funciona con rutas, también lo hace con componentes a través de la función **loadComponent**:

```
{path: 'ruta_5', loadComponent: () => import('../cap09/ruta-03/ruta-03.component').then(c => c.Ruta03Component) },
```

La carga diferida es una técnica imprescindible cuando la aplicación crece en tamaño, por lo que Angular ha introducido lo que se denomina vistas aplazables (deferrable views), para más información ver la documentación oficial.

Formularios

Los formularios web son la tecnología para recoger datos de usuario, consisten en una etiqueta `<form>` de HTML y uno o varios elementos `<input>` de entrada. EL Framework de Angular proporciona dos mecanismos para manejarlos, formularios dirigidos por plantillas y formularios reactivos. Ninguna de las dos aproximaciones la podemos considerar mejor, dependerá del problema que estemos tratando el que nos hará decidir por una o por otra. La principal diferencia entre ambos es cómo se manejan los datos.

- **Formularios dirigidos por plantillas.** Son la manera más fácil de crear un formulario. Operan exclusivamente a nivel de plantilla para crear elementos de entrada y sus reglas de validación, dependen del mecanismo de cambio del Framework para su buen funcionamiento.
- **Formularios reactivos.** Son más robustos para la realización de test y escalado. Operan al nivel de componente para manejar los campos de entrada y la validación.

El código de ayuda se puede ver en cap10.

Dirigidos por plantillas

Este mecanismo es muy poderoso si queremos crear formularios simples en nuestra aplicación basados en HTML con reglas de validación simples. Podemos usar la sincronización bidireccional entre campos y propiedades del componente (`ngModel`) para realizar el trabajo. Crearemos de ejemplo un formulario de validación.



Tras la creación del componente añadimos a su plantilla el formulario HTML con las reglas de validación HTML necesarias.

```

<h2> Login Form</h2>
<form #frmLogin : HTMLFormElement >
  <div class="form-group">
    <label for="username">Username:</label>
    <input type="text" class="form-control border-dark" id="username" [(ngModel)]="username" name="username" required>
  </div>
  <div class="form-group">
    <label for="password">Password:</label>
    <input type="password" class="form-control border-dark" id="password" [(ngModel)]="password" name="password" required>
  </div>
  <div class="form-group mt-1">
    <button type="submit" class="btn btn-success" (click)="Login()">Login</button>
    <button type="button" (click)="reset()" class="btn btn-danger">Reset</button>
  </div>
</form>

```

Vemos el uso de la sincronización bidireccional en los campos input de clave y usuario (`[(ngModel)]`) que utilizarán las propiedades correspondientes del componente. En el momento de pulsar alguno de los botones presentes, se lanzarán los métodos de `login` o `reset`. Establecemos las reglas de validación HTML (`required`). El código del componente es el siguiente:

```
export class LoginComponent {
  password: string = '';
  username: string = '';
  @ViewChild( selector: 'frmLogin', { static: true } ) frmLogin!: ElementRef;

  login(): void { Show usages
    alert( message: `Username: ${this.username}, Password: ${this.password}` );
  }

  reset(): void { Show usages
    this.frmLogin.nativeElement.reset();
  }
}
```

Introducimos el uso del decorador `@ViewChild` para tener acceso al formulario de la plantilla y poder resetearlo, para ello se crea una variable de plantilla en la etiqueta `form` como `#frmLogin` (ver plantilla), en el componente definimos una propiedad llamada igual. Para el acceso a la etiqueta se usa la propiedad `nativeElement`, haciendo uso del método `reset` en este caso.

Este ejemplo se ha capturado el evento `clic` en el botón de envío que ejecuta `login()`, pero se podría definir dicho botón del tipo `submit` exclusivamente y capturar el evento `submit` del formulario.

```
<form (ngSubmit)="onSubmit()"> <-- En template -->
...
onSubmit(){ // En componente
  ...
}
```

Para un formulario completo.

- **markAllAsTouched**. Marca todos los campos como modificados, generalmente usado para mostrar todos los errores.

Para cada campo

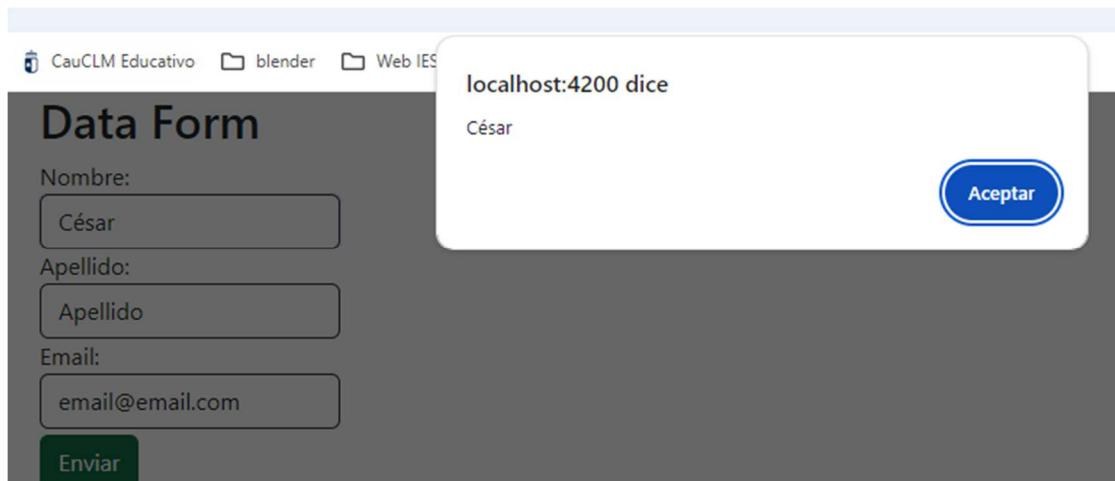
- **markAsTouched**. Marca el campo actual como modificado.

Formularios reactivos

En esta aproximación, los controles y sus valores se manipulan desde flujos observables, manteniendo en todo momento el estado del formulario de forma inmutable. Los elementos se crean de forma

programática y se enlazan con los de la plantilla para la sincronización. Angular tiene varias clases para realizar la gestión de este tipo de formularios:

- **FormControl**. Representa un control de entrada simple.
- **FormGroup**. Es un conjunto de controles de entrada.
- **FormArray**. Similar al anterior pero su contenido puede variar de forma dinámica durante la ejecución, no como el anterior que es estático.



```
<h2> Data Form</h2>
<form [formGroup]="formulario" (ngSubmit)="onSubmit()">
  <div class="form-group">
    <label for="nombre">Nombre:</label>
    <input id="nombre" class="form-control border-dark" formControlName="nombre" />
  </div>
  <div class="form-group">
    <label for="apellido">Apellido:</label>
    <input id="apellido" class="form-control border-dark" formControlName="apellido" />
  </div>
  <div class="form-group">
    <label for="email">Email:</label>
    <input id="email" class="form-control border-dark" formControlName="email" type="email"/>
  </div>
  <button type="submit" class="btn btn-success mt-1" (click)="onEnviar()>Enviar</button>
</form>
```

```

export class FrmDatosComponent {
  formulario : FormGroup<{ nombre: FormControl<string>; ... }> = new FormGroup({
    nombre: new FormControl( value: 'Nombre', {nonNullable: true} ),
    apellido: new FormControl( value: 'Apellido', {nonNullable: true} ),
    email: new FormControl( value: 'email@email.com', {nonNullable: true} ),
  });
}

onSubmit() : void { Show usages
  alert(this.formulario.controls.nombre.value);
}

onEnviar() : void { Show usages
  alert( message: "On enviar");
}
}

```

Como hemos dicho, la creación de este tipo de formularios implica la creación de la estructura en el componente que podemos apreciar en la variable **formulario**, en la que se define la misma estructura que en la plantilla. A continuación, se enlaza esta variable con la plantilla a través de la directiva **[FormGroup]** de la etiqueta HTML **form**. Para que todo funcione correctamente, la definición del formulario en el componente tendrá tantos campos como los de la plantilla y el nombre que usemos como clave (**nombre**, **apellido**, **email**) en el mismo, se enlazará a la plantilla a través de la directiva **formControlName** en cada campo. Con esto tenemos enlazado los campos con las propiedades.

Las reglas de validación, se establecen en el componente (fichero .ts) a través de un diccionario en el segundo parámetro de la creación del control (**FormControl**).

Para acceder al campo HTML desde el componente, a través del formulario, la propiedad **controls** es un objeto con todos los controles accesibles por nombre (**formulario.controls.nombre_del_campo**) y la propiedad **value** nos proporciona el valor actual del campo (ver **onSubmit** del ejemplo). Esta propiedad no incluirá valores si el campo está deshabilitado, en tal caso usaremos el método **getRawValue**. Para establecer desde el componente el valor de un campo usaremos el método **setValue** del mismo.

Angular nos proporciona una clase generadora de formularios (**FormBuilder**) que simplifica la creación repetitiva.

Visitar <https://angular.dev/api/forms/FormBuilder>

A través del observable **valueChanges** de cada campo o del formulario, se puede escuchar los cambios de la plantilla mediante el método **subscribe()** utilizando la propiedad **value**, que proporciona una instantánea del valor actual. En el ejemplo siguiente vemos los cambios del campo **apellidos**, se configuran dos botones para poner y quitar la suscripción y ver el funcionamiento.

```

<button type="submit" class="btn btn-success mt-1" (click)="onSuscribir()">Suscribir a Apellido</button>
<button type="submit" class="btn btn-success mt-1" (click)="onQuitar()">Quitar suscripción</button>

```

```

onSuscribir() : void { Show usages
  this.observable_suscripcion = this.formulario.controls.apellido.valueChanges.subscribe(
    (apellido : string) : void => {
      alert(apellido);
    }
  );
}

onQuitar() : void { Show usages
  this.observable_suscripcion.unsubscribe();
}

```

Una vez completados los cambios anteriores, si entramos en el campo Apellido y cambiamos su valor veremos que se muestra un mensaje si estamos suscritos.

Validación CSS

Uno de los aspectos más importantes de la gestión de un formulario no solo es que se lleve a cabo una validación, si no el mostrarle el resultado al usuario de dicha acción de forma adecuada. Para esta tarea Angular crea un conjunto de clases CSS que podemos utilizar para proporcionar al usuario feedback:

- **ng-untouched**. Indica que no se ha interactuado con el formulario.
- **ng-touched**. Indica que se ha interactuado con el formulario.
- **ng-dirty**. Indica que se ha establecido un valor en el formulario.
- **ng-pristine**. Indica que todavía no se ha modificado el formulario.
- **ng-valid**. Indica que el campo es válido.
- **ng-invalid**. Indica que el campo es invalido.

Estas clases se establecen de forma automática con la validación del formulario, con lo que simplemente tendremos que dar un estilo global CSS, generalmente en el fichero style.css, a las mismas para que se aplique.

Cada una de las clases anteriores, se corresponde con una propiedad booleana en cualquiera de los dos tipos de formularios. El nombre de estas propiedades es el visto en la lista anterior, eliminado el prefijo **ng-**. Veamos como acceder a ellas a través del modelo de plantilla.

```

<form #frmLogin :NgForm ="ngForm" (ngSubmit)="login()">
  <div class="form-group">
    <label for="username">Username:</label>
    <input type="text" #usernameCtrl :NgModel = "ngModel" required
      class="form-control border-dark" id="username" [(ngModel)]="username" name="username" >
    @if (usernameCtrl.dirty && (usernameCtrl.invalid || usernameCtrl.hasError( errorCode: 'required'))) {
      <div class="alert alert-danger">
        <strong>Invalid Username</strong>
        <p>Username required.</p>
      </div>
    }
  </div>

```

```
<button type="submit" [disabled]="frmLogin.invalid"
        class="btn btn-success" (click)="login()>Login</button>
```

Para acceder al control creamos una variable de plantilla (#usernameCtrl) uniéndola con el modelo a través de **ngModel**. Esta configuración nos permite acceder a través de directivas de plantilla (@if) a la etiqueta y su estado, pudiéndolo usar como nos interese accediendo a los errores de validación existentes mediante el método **hasError** (ver condición de la directiva @if).

Podemos extender es método para un formulario mediante **ngForm** asignado a la variable de plantilla y utilizarlo como anteriormente o enlazándolo a una propiedad de la etiqueta, como vemos en ejemplo del botón.

Con este mecanismo se puede comprobar cualquier atributo HTML de validación: min, max, minLength, maxLength, etc.

Hemos visto que el mecanismo orientado a plantillas es eficiente cuando el formulario es pequeño, veamos ahora con cómo se implementaría visualmente el control de errores en una aproximación de formularios reactivos.

En los formularios reactivos, la fuente de toda verdad es el modelo implementado en las propiedades del componente, por lo que hay que implementar las reglas de validación cuando se crea el mismo mediante la llamada a **FormGroup**.

En un primer paso, definimos los validadores que usaremos en cada campo en el momento que definimos el grupo, vemos en el ejemplo que se puede usar más de uno si los pasamos como un Array. Una vez establecidos, usamos una técnica similar a la de plantillas para presentar la información, la diferencia es que el acceso a la etiqueta se hace a través de la colección **controls** del formulario y el **name** del campo en vez de una variable de plantilla, siendo también posible el enlazar propiedades de una etiqueta al valor del formulario como antes mediante la propiedad correspondiente.

```
formulario : FormGroup<{ nombre: FormControl<string>; ... }> = new FormGroup({
  nombre: new FormControl( value: 'Nombre', {nonNullable: true, validators: Validators.required} ),
  apellido: new FormControl( value: 'Apellido', {nonNullable: true, validators: Validators.required} ),
  email: new FormControl( value: 'email@email.com', {nonNullable: true, validators: [Validators.required, Validators.email] } ),
});

<input id="email" class="form-control border-dark" formControlName="email" type="email"/>
@if (formulario.controls.email.dirty && (formulario.controls.email.invalid || formulario.controls.email.hasError( errorCode: 'email')) ) {
  <div class="alert alert-danger">
    <strong>Invalid email</strong>
    <p>Must be a valid email.</p>
  </div>
}
```

login Form

The screenshot shows a login form with two fields: 'Username' and 'Password'. The 'Username' field has a red error box with the message 'Invalid Username' and 'Username required.'. The 'Password' field also has a red error box with the message 'Invalid Password' and 'Password must be at least 4 characters long.' Below the fields are 'Login' and 'Reset' buttons.

The screenshot shows a reactive form with three fields: 'Nombre', 'Apellido', and 'Email'. The 'Email' field has a red error box with the message 'Invalid email' and 'Must be a valid email.'. Below the fields are 'Enviar', 'Suscribir a Apellido', and 'Quitar suscripción' buttons.

```
<button type="submit" [disabled]="formulario.invalid"
        class="btn btn-success mt-1" (click)="onEnviar()>Enviar</button>
```

Angular proporciona un conjunto lo suficientemente amplio de validaciones predefinidas para que no tengamos que preocuparnos por crear los propios, siendo esto último también posible. Los validadores preestablecidos los podemos ver en:

<https://angular.dev/api/forms/Validators>

La creación de validadores propios se puede visitar en:

<https://angular.dev/guide/forms/form-validation#defineing-custom-validators>

Manipular el estado del formulario

El último aspecto a tratar es la gestión de los datos de un formulario, que difiere estemos usando el modelo de plantillas o el reactivo. Para el acceso, en el primer caso utilizaremos la sincronización de los datos a través de **ngModel** (ver ejemplos anteriores) mientras que, en el segundo caso, será a través de la propiedad **value** de la clase FormControl (ver ejemplos anteriores).

Para establecer los valores, en el modelo de plantillas, será necesario crear una variable de plantilla y recogerla en el componente con el decorador @ViewChild (ver ejemplos anteriores). En el caso de formularios reactivos, haremos uso de los siguientes métodos de la clase FormGroup:

- **setValue**. Cambia el valor de todos los campos de un formulario, acepta un objeto con pares clave-valor en los que aparecerán todos los campos del formulario. La clave debe ser el **name** el control.
- **patchValue**. Actualiza los valores de uno o varios campos exclusivamente, NO de todos. Acepta como parámetro un objeto similar al método anterior, pero en el que aparecerán solo los pares clave-valor que deseamos cambiar.

REACCIONAR A LOS CAMBIOS

Para reaccionar a los cambios que se van produciendo en el formulario se utiliza la sincronización de propiedades mediante **ngModel** (ver ejemplo anterior) y también podemos utilizar el evento **ngModelChange** que Angular añade a cada etiqueta INPUT, pudiendo ser capturado como cualquier otro evento. Este evento es lanzado por Angular de forma automática ante cualquier cambio en el formulario.

```
<input type="password" #passwordCtrl : NgModel = "ngModel" minlength="4" required
       class="form-control border-dark" id="password" [(ngModel)]="password" name="password"
       (ngModelChange)="onChange($event)">
```

En formularios reactivos, existe un Observable al que nos podremos suscribir para recibir las notificaciones de cambio. Este se llama **valueChange** y está presente tanto en la clase FormControl como en FormGroup (Ver ejemplo anterior).

Llevar a Producción

Una vez creada y testeada la aplicación, hay que llevarla a producción, hacerla accesible a los usuarios. El proceso consta de dos partes, compilación de las fuentes y publicación en el servidor. Vamos a detallar cada parte.

El primer paso es la creación de las fuentes. El compilador debe recoger todos los ficheros fuente de TypeScript y transcribirlos a JavaScript, copiar todos los elementos estáticos, así como los ficheros CSS. El proceso de generación también realiza la optimización y el renderizado del GUI. El comando de generación es:

```
ng build
```

Este comando utilizará las opciones configuradas en los ficheros **tsconfig.app.json** y **angular.json**, dejará todo lo generado en el directorio **dist\appNombre\browser**. La salida que hay en este directorio se puede llevar directamente a la raíz de cualquier servidor web. En caso de querer que la aplicación se despliegue en otro directorio diferente al raíz en el servidor tendremos que establecer la opción **baseHref** en el fichero configuración la ruta correspondiente, para este cambio se usa el comando siguiente:

```
ng build --base-href /ruta/
```

Para terminar esta sección comentaremos una última configuración necesaria para aquellos casos en los que queramos añadir otros Frameworks (Bootstrap, JQuery, etc) o ficheros de estilo o JavaScript propios, correctamente. En este caso en el fichero de configuración, buscaremos las secciones **styles** y **scripts**. Y nos aseguraremos que estén establecidos aquí correctamente.

```
"options": {  
  ...  
  "styles": [  
    "node_modules/bootstrap/dist/css/bootstrap.min.css",  
    "src/styles.css"  
  ],  
  "scripts": [  
    "node_modules/bootstrap/dist/js/bootstrap.bundle.js"  
  ]  
}
```

Capítulo X. Proyecto Laravel (GraphQL) - Angular

Servidor Laravel

Crear un API GraphQL para que del proyecto del Capítulo VIII un administrador pueda gestionar las reservas (RF4.2). Hará falta que se esté validado antes de ello.

Crear un proyecto nuevo y configurar

Copiar del cap_08: Migrations (NO las del sistema, las que empiezan con 001), Seeders y Models

Hacer la migración y carga de la Base de datos

Instalar el software para GraphQL

composer require nuwave/lighthouse

php artisan vendor:publish --tag=lighthouse-schema

copy vendor\nuwave\lighthouse\src\lighthouse.php config\

composer require mll-lab/laravel-graphiql

php artisan install:api

Instalar el software de gestión de tokens

composer require laravel/sanctum

php artisan vendor:publish --provider="Laravel\Sanctum\SanctumServiceProvider"

php artisan migrate

Modificar el sistema para autorización

En el fichero models/user.php

use Laravel\Sanctum\HasApiTokens;

---> use **HasApiTokens**..., HasFactory, Notifiable;

Crear el sistema de autorización

En el fichero config/lighthouse.php

'guards' => ['**graphql_auth**'],

En el fichero config/auth.php

En guards: añadir

```
'graphql_auth' => [
    'driver' => 'sanctum',
    'provider' => 'users',
],
```

Crear el API para gestionar sus reservas: Login / Logout / Me

Modificar el esquema para login: añadir las tres Querys

Crear los tipos necesarios

logout: Boolean! @guard(with: ["graphql_auth"])

Añadir los resolutores para cada query en http/GraphQL/Queries: Login.php Logout.Php Me.php

Probar con Graphiql

Crear el API para gestionar sus reservas: Ver reservas

Crear el tipo con los datos

Crear la Query en el esquema

Crear el resolutor

Crear el API para gestionar sus reservas: Ver datos de una reserva

Crear el tipo con los datos

Crear la Query en el esquema

Crear el resolutor

Crear el API para gestionar sus reservas: Cancelar una reserva

Crear el tipo con los datos

Crear la Query en el esquema

Crear el resolutor: asegurarse que solo se cancela si es posible: CONFIRMADA

Crear el API para gestionar sus reservas: Valorar y Finalizar una reserva

Crear el tipo con los datos

Crear la Query en el esquema

Crear el resolutor: asegurarse que solo se finaliza si es posible: CONFIRMADA y con los datos necesarios en rango correcto

```
curl http://localhost//graphql -L -H "Content-Type: application/json" -H "Accept: application/json" --data-binary "{\"query\":\"\\\"query {login(email:\\\"abc@abc\\\\\",password:\\\"abc\\\\\")}\\\""}"
```

```
curl http://localhost//graphql -L -H "Content-Type: application/json" -H "Accept: application/json" --data-binary "{\"query\":\"\\\"{logout}\\\"\"}" -H "Authorization: Bearer token_tesultado_login"
```

Frontend Angular

Crear un Frontend basado en Angular para gestionar las reservas del sistema por un administrador, ser capaz de verlas, filtralas y cancelarlas (RF4.2).

Crear el API en php para gestionar sus reservas: Ver destinos: modificar el esquema GraphQL

Crear el tipo con los datos

Crear las Querys en el esquema

Crear los resolutores

Al mandar desde cliente hay que añadir la siguiente cabecera en los posts:

```
{"Authorization":"Bearer token_resultado_login"}
```

Crear un componente para el menú

Similar al del proyecto anterior, la sección de reservas solo se activará si se ha validado antes

Modificar la plantilla de la aplicación para incluir el menú y la estructura

Crear un componente para un formulario de validación

Crear un servidor de comunicaciones con el API GraphQL

Crear un servidor de autorizaciones

Crear un servicio de autorización que al validar guarde el token recibido y al hacer logout lo borre, y sea utilizado cliente http en toda petición.

Crear el componente de gestión de reservas

Crear un componente para la reserva

Crear un componente para la lista de reservas

Definir la lógica de gestión

Trabajo para el alumno

Modificar el API GraphQL creado anteriormente para la gestión de panel administrativo, Requisitos funcionales RF4.1 y RF4.3

Modificar el Frontend para modificar el panel administrativo RF4.1 y RF4.3

Desplegar en un servidor real Apache o NGIX dentro de una Máquina Virtual o Docker y MariaDB

Hay que migrar la BBDD con **php artisan migrate**

Tiempo desarrollo por parte del alumno estimado: 25 horas.

Solucionar CORS en Laravel Lighthouse

Cuando hacemos peticiones a nuestro Backend desde otro servidor (como, por ejemplo, desde nuestro Framework Frontend, alojado en otra dirección), nos dará un error de acceso no permitido CORS (Cross-Origin Request Blocked). Este error no es específico de Laravel, sino que es un método de seguridad que se da en las peticiones que se hagan a otro servidor y no se tenga acceso. Para solucionarlo, manteniendo la seguridad. Primero publicamos el fichero de configuración de cors.

`php artisan config:publish cors`

en el fichero config/cors.php añadimos:

`'paths' => ['api/*', 'graphql', 'sanctum/csrf-cookie'],`

SEGUNDA SOLUCIÓN SOLO PARA DESARROLLO

Sin hacer lo anterior, podemos hacer que Chrome no lance las peticiones CORS de la siguiente manera y nos evitamos todo el problema hasta el despliegue que sí será obligatorio. Creamos el directorio **chrome_dev** y nos posicionamos en él:

```
mkdir c:\chrome_dev  
cd c:\ chrome_dev
```

Lanzamos el Chrome en modo desarrollo

```
"C:\Program Files\Google\Chrome\Application\chrome.exe" --disable-web-security --user-data-dir="C:\chrome_dev"
```

Ejemplo de despliegue

- Servidor API a: C:\xampp\htdocs\estrella_viajera_api
 - npm build desde el directorio raíz del proyecto
 - copiar todo el proyecto excepto el nodes_modules
- Configurar el directorio raíz de Apache a este directorio en httpd.conf
 - DocumentRoot "C:\xampp\htdocs\estrella_viajera_api"
 - <Directory "C:\xampp\htdocs\estrella_viajera_api">
- FrontEnd deploy a: \estrella_viajera_api_fe\
 - ng build --base-href /estrella_viajera_api_fe/
- FrontEnd copiado en C:\xampp\htdocs\estrella_viajera_api\public\estrella_viajera_api_fe
 - Copiar el contenido de directorio dir_Proyecto\dist\estrella_viajera_api_fe\browser
- Acceso: http://localhost/estrella_viajera_api_fe

Anexo I. Recursos

Fuente de datos JSON

<https://jsonplaceholder.typicode.com/>
<https://jsonplaceholder.typicode.com/guide/>

```
fetch('https://jsonplaceholder.typicode.com/todos/1')
  .then(response => response.json())
  .then(json => console.log(json))
```

JSONPlaceholder is a free online REST API that you can use **whenever you need some fake data**. It can be in a README on GitHub, for a demo on CodeSandbox, in code examples on Stack Overflow, ...or simply to test things locally.

Resources

JSONPlaceholder comes with a set of 6 common resources:

| | |
|---------------------------|--------------|
| /posts | 100 posts |
| /comments | 500 comments |
| /albums | 100 albums |
| /photos | 5000 photos |
| /todos | 200 todos |
| /users | 10 users |

Routes

All HTTP methods are supported. You can use http or https for your requests.

| | |
|--------|------------------------------------|
| GET | /posts |
| GET | /posts/1 |
| GET | /posts/1/comments |
| GET | /comments?postId=1 |
| POST | /posts |
| PUT | /posts/1 |
| PATCH | /posts/1 |
| DELETE | /posts/1 |

Anexo II. Índice completo

| | |
|---|-----------|
| Capítulo I. Introducción | 6 |
| Historia de las Apis..... | 6 |
| Problema que resuelve..... | 7 |
| Diseño Rest vs GraphQL..... | 7 |
| Diferencias en la solicitud del cliente | 8 |
| Resumen de las diferencias | 9 |
| Por qué GraphQL | 9 |
| Preparando el entorno para PHP..... | 10 |
| Servidor web..... | 11 |
| Instalación de composer..... | 13 |
| Consideraciones sobre la instalación del resto de elementos | 13 |
| Capítulo II. GraphQL: Introducción | 14 |
| Qué es GraphQL..... | 14 |
| Qué es un lenguaje de consultas | 14 |
| Conceptos clave | 14 |
| Características..... | 15 |
| Arquitectura de GraphQL | 16 |
| Primer ejemplo | 16 |
| Expliquemos lo que ha pasado | 18 |
| Introspección | 18 |
| Uso de la IA | 19 |
| Generación de Consultas..... | 19 |
| Generación de documentación | 20 |
| Generación de datos de prueba | 21 |
| Generación de esquemas | 21 |
| Generación de un cliente html | 22 |
| Capítulo III. GraphQL: Esquemas y tipos | 24 |
| Esquema | 24 |
| Qué es | 24 |
| Tipos..... | 25 |
| Tipos escalares predefinidos | 25 |
| Tipos lista | 26 |

| | |
|--|-----------|
| Tipos de objetos..... | 26 |
| Componentes..... | 26 |
| Argumentos | 27 |
| Enumeraciones | 27 |
| Los tipos Query, Mutation y Subscription | 27 |
| Interfaces | 29 |
| Uniones..... | 30 |
| Inputs | 31 |
| Directivas | 32 |
| Documentación y comentarios..... | 32 |
| Diseño de los esquemas con POO en vez de SDL | 32 |
| SDL de partida | 33 |
| Creación bajo POO | 33 |
| Capítulo IV. GraphQL: Consultas | 39 |
| Qué es el lenguaje de consultas | 39 |
| Estructura de una consulta desde el cliente..... | 39 |
| Componentes de una consulta GraphQL..... | 40 |
| Campos | 40 |
| Aliases | 41 |
| Variables | 42 |
| Fragmentos..... | 44 |
| Directivas | 44 |
| Ejemplos de consultas GraphQL de introspección | 45 |
| Ejemplos de consultas GraphQL mutaciones | 47 |
| Ejemplos de consultas GraphQL suscripciones | 48 |
| Validaciones..... | 49 |
| Ejecución de las consultas | 49 |
| Resolutores de campos..... | 49 |
| Resolutores de los campos raíz | 50 |
| Resolutores por defecto | 50 |
| Producción del resultado..... | 50 |
| Capítulo V. Bootstrap | 51 |
| Configurar PhpStorm para Bootstrap | 51 |
| Configurar Bootstrap en mis páginas | 51 |

| | |
|--|-----------|
| Configuración inicial de una pagina..... | 52 |
| Disposición de la página | 53 |
| Puntos de ruptura..... | 53 |
| Contenedores | 53 |
| Grid | 54 |
| Alineamiento de las columnas | 57 |
| Desplazamiento de columnas | 57 |
| Espaciado (gutters)..... | 58 |
| Orden de profundidad | 58 |
| Formularios | 59 |
| Checkboxes y options | 60 |
| Rangos..... | 61 |
| Grupos de controles | 61 |
| Etiquetas en las cabeceras..... | 64 |
| Componentes Bootstrap..... | 65 |
| Clases de ayuda | 66 |
| Utilidades | 67 |
| Espaciado | 68 |
| Utilidades de texto..... | 69 |
| Alineación vertical | 69 |
| Visibilidad..... | 70 |
| Orden de profundidad | 70 |
| Capítulo VI. Composer PHP | 71 |
| Uso de Composer..... | 71 |
| Instalación en local bajo Windows | 71 |
| Instalación Windows Global | 71 |
| Instalación local | 71 |
| Generando y comprendiendo composer.json..... | 71 |
| Usando el Script Autoload | 72 |
| Actualización de las dependencias de tu proyecto | 73 |
| Configurar el sistema de autocarga de clases | 73 |
| Ejemplo | 74 |
| Capítulo VII. Laravel..... | 76 |
| Preparando un entorno de desarrollo..... | 76 |

| | |
|---|----|
| Inicializando Laravel..... | 76 |
| El patrón Modelo-Vista-Controlador (MVC)..... | 77 |
| Tipos de programación | 78 |
| Desarrollo completo | 78 |
| Servidor API | 79 |
| Estructura de directorios de la aplicación Laravel..... | 79 |
| Opciones de configuración | 80 |
| Configuración del entorno..... | 80 |
| Configurar Bootstrap para Laravel | 81 |
| Instalar los ficheros necesarios | 81 |
| Modificar los siguientes archivos a través del IDE..... | 81 |
| Rutas | 82 |
| Cómo definir rutas en Laravel | 82 |
| Redirección a otra URL | 84 |
| Devolviendo vistas | 84 |
| Cómo pasar parámetros en las rutas..... | 84 |
| Parámetros opcionales | 84 |
| Cómo utilizar controladores en Laravel..... | 85 |
| Cómo utilizar grupos de rutas en Laravel..... | 85 |
| Cómo definir rutas con Route::resource | 86 |
| Cómo utilizar rutas con nombres en Laravel..... | 87 |
| Cómo generar URLs en Laravel..... | 88 |
| Validación..... | 88 |
| Entendiendo la utilidad Artisan | 89 |
| Blade | 90 |
| Introducción..... | 90 |
| Directivas | 90 |
| Directivas más comunes | 91 |
| Herencia..... | 93 |
| Organización en carpetas | 93 |
| Paso de parámetros..... | 94 |
| orden compact..... | 94 |
| Bases de datos | 94 |
| Migraciones | 94 |

| | |
|---|-----|
| Crear nuevas migraciones | 95 |
| Tipos de columnas y opciones comunes..... | 96 |
| Modificando tablas a través de migraciones..... | 96 |
| Añadir columnas a una tabla existente en Laravel..... | 96 |
| Modificar columnas en Laravel usando migraciones | 98 |
| Eliminar columnas en Laravel usando migraciones | 98 |
| Migraciones avanzadas: renombrar tablas | 99 |
| Uso de Seeders junto a las migraciones..... | 99 |
| Rollbacks, refrescar y reiniciar migraciones..... | 100 |
| Buenas prácticas para las migraciones | 100 |
| Acceso a los datos a través del ORM | 100 |
| Relaciones..... | 101 |
| Relaciones uno a uno (Has One)..... | 101 |
| Definición de la relación inversa | 101 |
| Relación Uno a muchos (Has Many) | 102 |
| Valores por defecto | 102 |
| Acceso rápido a valores recientes..... | 103 |
| Relaciones muchos a muchos..... | 103 |
| Consultas..... | 104 |
| Recuperando valores | 105 |
| Limitando las búsquedas..... | 105 |
| Ordenaciones..... | 106 |
| Group By | 106 |
| Limit y Offset | 107 |
| Colecciones..... | 107 |
| Métodos disponibles..... | 108 |
| Borrado..... | 108 |
| Borrado condicional..... | 108 |
| Eliminación suave (Soft delete) | 108 |
| Añadir el borrado suave a un modelo..... | 109 |
| Cómo incluir modelos con borrado suave | 109 |
| Cómo recuperar únicamente modelos con borrado suave | 109 |
| Cómo restaurar modelos con borrado suave | 110 |
| INSERCIONES y actualizaciones | 110 |
| Serialización | 111 |
| Serialización a Arrays..... | 111 |
| Serialización a JSON | 112 |
| Paginación..... | 112 |

| | |
|--|-----|
| Configuración de la paginación | 112 |
| Uso de la paginación | 113 |
| Imprimiendo los enlaces | 113 |
| Personalización de los enlaces | 114 |
| Consejos adicionales | 114 |
| Controladores y Modelos | 114 |
| Introducción..... | 114 |
| Modelos | 115 |
| Convenciones | 116 |
| Controladores | 116 |
| Autentificación bajo Laravel | 117 |
| Starter Kits | 118 |
| Autentificación básica bajo HTTP | 118 |
| Autentificación manual..... | 118 |
| Especificando condicionales adicionales | 120 |
| Recordando usuarios..... | 121 |
| LogOut..... | 121 |
| Confirmación de clave | 122 |
| Configuración | 122 |
| Enrutado | 122 |
| El formulario de confirmación | 122 |
| Confirmación de la clave | 122 |
| Protegiendo rutas | 123 |
| GraphQL bajo Laravel | 124 |
| Instalación vía Composer..... | 124 |
| Publicación del esquema por defecto | 124 |
| Configuración..... | 124 |
| Instalación de las herramientas GraphQL | 124 |
| Ampliar el ejemplo..... | 124 |
| Directivas aplicables a un esquema bajo Lighthouse | 126 |
| Seguridad basada en tokens..... | 126 |
| Desplegar a producción | 127 |
| Configuraciones en ficheros | 127 |
| base.blade.php | 127 |
| Otras consideraciones | 127 |

| | |
|---|------------|
| Configurar el servidor web Apache | 127 |
| Capítulo VIII. Proyecto Laravel – Bootstrap: Web de Reservas de Turismo Espacial: Estrella Viajera . | 128 |
| Requisitos Funcionales - | 128 |
| 1. Gestión de usuarios | 128 |
| 2. Exploración de viajes espaciales | 128 |
| 3. Reservas | 128 |
| 4. Panel de administración | 128 |
| Requisitos No Funcionales (RNF) | 128 |
| 1. Usabilidad | 128 |
| 2. Rendimiento | 128 |
| 3. Seguridad | 129 |
| 4. Mantenibilidad y escalabilidad | 129 |
| 5. Calidad y pruebas..... | 129 |
| 6. Despliegue | 129 |
| Requisitos Software | 129 |
| Casos de Uso | 130 |
| Diagrama E-R simplificado | 130 |
| Modelo Relacional | 130 |
| Usuarios | 130 |
| Destinos | 130 |
| Reservas | 131 |
| Valoraciones | 131 |
| Diagrama de clases | 131 |
| Desarrollo..... | 132 |
| Creación del entorno y preparación del software..... | 132 |
| Creación del menú y de la cabecera con el logo | 132 |
| Creación de la Base de datos..... | 133 |
| Creación del sistema de login/logout y control de acceso..... | 134 |
| Generar los listados de destinos y el sistema de búsquedas | 135 |
| Generar la reserva de un destino | 135 |
| Gestionar las reservas existentes | 135 |
| Gestionar el cierre del viaje | 135 |
| Trabajo para el alumno..... | 136 |

| | |
|---|------------|
| Capítulo IX. Angular..... | 137 |
| Introducción..... | 137 |
| Establecer el entorno de desarrollo | 137 |
| Prerrequisitos | 137 |
| Uso del CLI (Consola Angular)..... | 137 |
| Creación de un proyecto..... | 138 |
| Añadir Bootstrap al proyecto Angular | 138 |
| Añadir las herramientas DevTools al navegador | 138 |
| Estructura de un proyecto Angular | 139 |
| Estructura de una aplicación | 139 |
| Introducción a TypeScript..... | 140 |
| JavaScript: declaración de variables | 140 |
| JavaScript: parámetros de funciones..... | 141 |
| JavaScript: funciones flecha..... | 142 |
| JavaScript: acceso condicionado | 142 |
| JavaScript: operador Fusión de Null (??) | 142 |
| JavaScript: clases..... | 142 |
| JavaScript: módulos | 143 |
| Qué es TypeScript | 143 |
| Tipos en TypeScript..... | 143 |
| Funciones | 144 |
| Modificaciones a las clases | 144 |
| Interfaces | 145 |
| Genéricos | 145 |
| Tipos utilitarios | 145 |
| Componentes..... | 146 |
| Estructura de un componente | 146 |
| Comunicación entre la lógica y la plantilla | 148 |
| Comunicación bidireccional..... | 149 |
| Directivas de plantilla | 149 |
| Eventos de plantilla..... | 150 |
| Comunicación entre componentes | 151 |
| Ciclo de vida de un componente | 154 |

| | |
|--|-----|
| Patrones de programación (Ver código de ayuda: cap03_patrones) | 155 |
| Menú | 155 |
| Formulario de validación..... | 155 |
| Tuberías y directivas | 155 |
| Tuberías | 155 |
| Directivas | 157 |
| Servicios | 158 |
| Tratamiento de las transmisiones asíncronas | 160 |
| Comunicación entre componentes no relacionados..... | 161 |
| Primero creamos El servicio (Observable) | 161 |
| Segundo definimos el Observador (receptor)..... | 162 |
| Ultimo modificamos el componente generador del evento | 162 |
| Operadores para la comunicación asíncrona | 163 |
| Señales | 163 |
| Creación de señales | 164 |
| Señales calculadas | 164 |
| Comunicación con servicios externos..... | 164 |
| Creación del componente e inclusión en el componente principal..... | 165 |
| Programación del componente | 165 |
| Modificación de la plantilla..... | 166 |
| Cabeceras y paso de parámetros HTTP | 166 |
| Interceptores | 167 |
| Rutas | 167 |
| Configuración de rutas..... | 168 |
| Orden de las rutas..... | 170 |
| Estilo en las rutas | 171 |
| Paso de parámetros en las rutas | 171 |
| Control de acceso a una ruta..... | 174 |
| Carga diferida de partes de la aplicación. | 175 |
| Formularios..... | 176 |
| Dirigidos por plantillas | 176 |
| Formularios reactivos | 177 |
| Validación CSS..... | 180 |
| Manipular el estado del formulario..... | 182 |

| | |
|--|------------|
| Reaccionar a los cambios | 182 |
| Llevar a Producción..... | 183 |
| Capítulo X. Proyecto Laravel (GraphQL) - Angular..... | 184 |
| Servidor Laravel | 184 |
| Frontend Angular..... | 186 |
| Trabajo para el alumno..... | 187 |
| Solucionar CORS en Laravel Lighthouse | 187 |
| Segunda Solución solo para desarrollo..... | 187 |
| Ejemplo de despliegue..... | 188 |
| Anexo I. Recursos | 189 |
| Fuente de datos JSON..... | 189 |
| Anexo II. Índice completo | 190 |