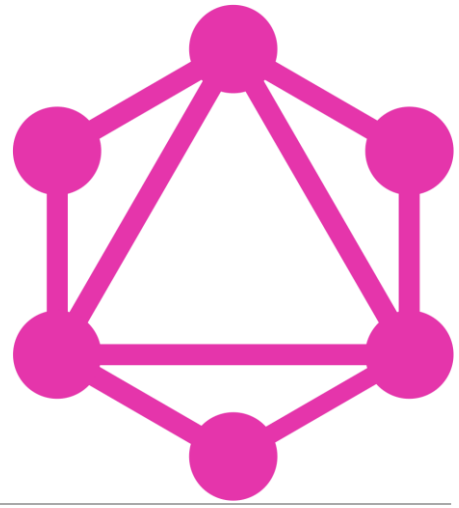
# GraphQL



PROGRAMACIÓN DE APLICACIONES UTILIZANDO *FRAMEWORKS*. UNIDAD 1

2º DESARROLLO DE APLICACIONES WEB. ARCIPRESTE DE HITA. 2025/2026

AARÓN MONTALVO

# 1. GraphQL.
# Criterios de evaluación

c) Se han utilizado distintos elementos middleware

e) Se han realizado consultas a bases de datos

h) Se ha implementado una API GraphQL

i) Se han realizado consultas utilizando una API GraphQL

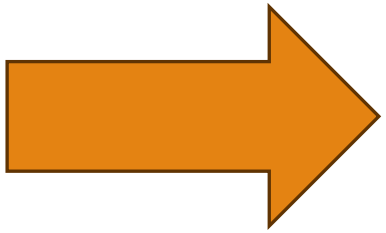
# 1. GraphQL. Contenidos

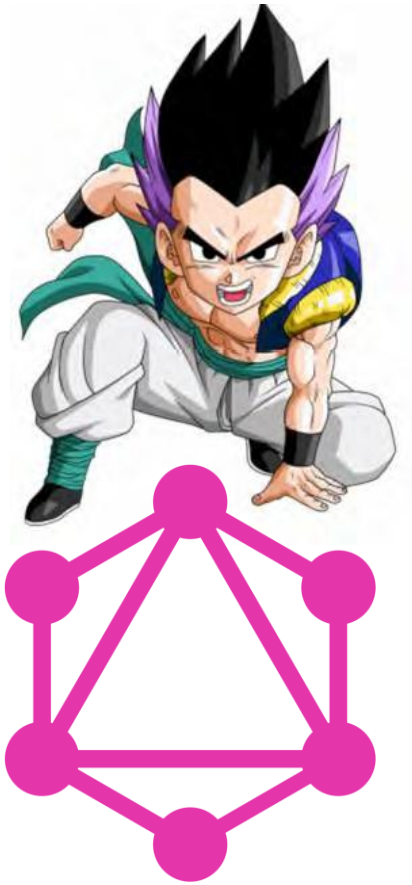
# 1.1. Introducción a GraphQL

GraphQL es un lenguaje de consulta para su API y un entorno de ejecución ***del lado del servidor*** para ejecutar consultas

- utiliza un sistema de tipos estrictamente definido para los datos, de código abierto, implementado en una variedad de lenguajes de programación.
- no está vinculado a ninguna base de datos o motor de almacenamiento específico.

# 1.1. Introducción a GraphQL



# 1.1. Introducción a GraphQL

GraphQL está formado por dos lenguajes:

- Lenguaje de esquema.
  - Definición de los tipos y campos.
- Lenguaje de consultas.
  - Solicitudes y extracción de información de un repositorio de datos.
    - Comandos o expresiones para especificar los campos o atributos deseados
    - Criterios de filtrado para seleccionar registros específicos
    - Condiciones de ordenamiento
    - Operaciones de agregación o combinación de datos

# 1.1. Introducción a GraphQL. Conceptos clave

## Esquema (*Schema*)

- Composición de diferentes definiciones que describen cómo interactuar con la API GraphQL, proporcionando un marco claro para la consulta y manipulación de datos.
- Plan estructurado que especifica los tipos de datos y las operaciones disponibles en una API GraphQL.
- En el esquema se definen objetos, interfaces y enumeraciones.

```
schema {
  query: Query
  mutation: Mutation
}
type Mutation {
  createOffice(input: OfficeInput!): Office
}
type Query {
  company(id: ID!): Company
}
type Company {
  id: ID!
  name: String
  address: String
  age: Int @deprecated(reason: "No longer relevant.")
  offices(limit: Int!, after: ID): OfficeConnection
}
```

```
type OfficeConnection {
  totalCount: Int
  nodes: [Office]
  edges: [OfficeEdge]
}
type OfficeEdge {
  node: Office
  cursor: ID
}
type Office {
  id: ID!
  name: String
}
input OfficeInput {
  name: String!
}
```

# 1.1. Introducción a GraphQL. Conceptos clave

**Tipo de datos (*Type*)**

- Estructura que manifiestan qué información se puede pedir y cómo pedirla en una API de GraphQL.
- La palabra 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.

# 1.1. Introducción a GraphQL. Conceptos clave

**Consulta (*Query*)**

- Solicitud enviada por los clientes para obtener datos específicos de la API de GraphQL.
- Las consultas permiten a los clientes obtener datos de manera precisa y eficiente.
- Los clientes utilizan la sintaxis de consulta de GraphQL para especificar los campos y relaciones que desean obtener.
- Las consultas de GraphQL son similares a peticiones REST del tipo SELECT.

# 1.1. Introducción a GraphQL. Conceptos clave

**Mutación (Mutations)**

- Operación 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 similares a peticiones REST de los tipos PUT, POST, PATCH y DELETE.

# 1.1. Introducción a GraphQL. Conceptos clave

**Resolutor (*Resolver*)**

- Función o método que se utiliza para resolver los campos solicitados en una consulta: devolver los valores asociados.
- Todo campo de un esquema de GraphQL está asociado a un *resolver* que se encarga de obtener los datos correspondientes.
  - Si no se asigna, se llamará al resolutor por defecto para el campo.
- Los *resolvers* pueden interactuar con bases de datos, servicios externos u otras fuentes de datos para obtener la información solicitada.

# 1.1. Introducción a GraphQL. Conceptos clave

**Directiva**

- Elemento opcional que se utiliza para modificar el comportamiento de las consultas y mutaciones.
- Permite realizar acciones como condicionales, fragmentación y cacheo de resultados.

**Introspección**

- Consulta del propio esquema para obtener información sobre los tipos de datos y las capacidades de la API.
  - Los clientes pueden descubrir las capacidades de la API de forma dinámica.

# 1.1. Introducción a GraphQL. Características

**Consulta precisa**: Permite a los clientes solicitar solo los datos específicos que necesitan, evitando la sobrecarga.

**Tipado fuerte**: Utiliza un sistema de tipos para definir la estructura y los campos, 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.

# 1.1. Introducción a GraphQL. Características

**Realización eficiente**: Mejora la eficiencia de la red al poder realizar múltiples solicitudes de datos en paralelo y devolver solo los resultados necesarios.

**Documentación automática**: El esquema se puede utilizar para generar automáticamente documentación precisa y actualizada.

**Introspección**: Permite a los clientes obtener información sobre el esquema y las capacidades de la API en tiempo de ejecución.

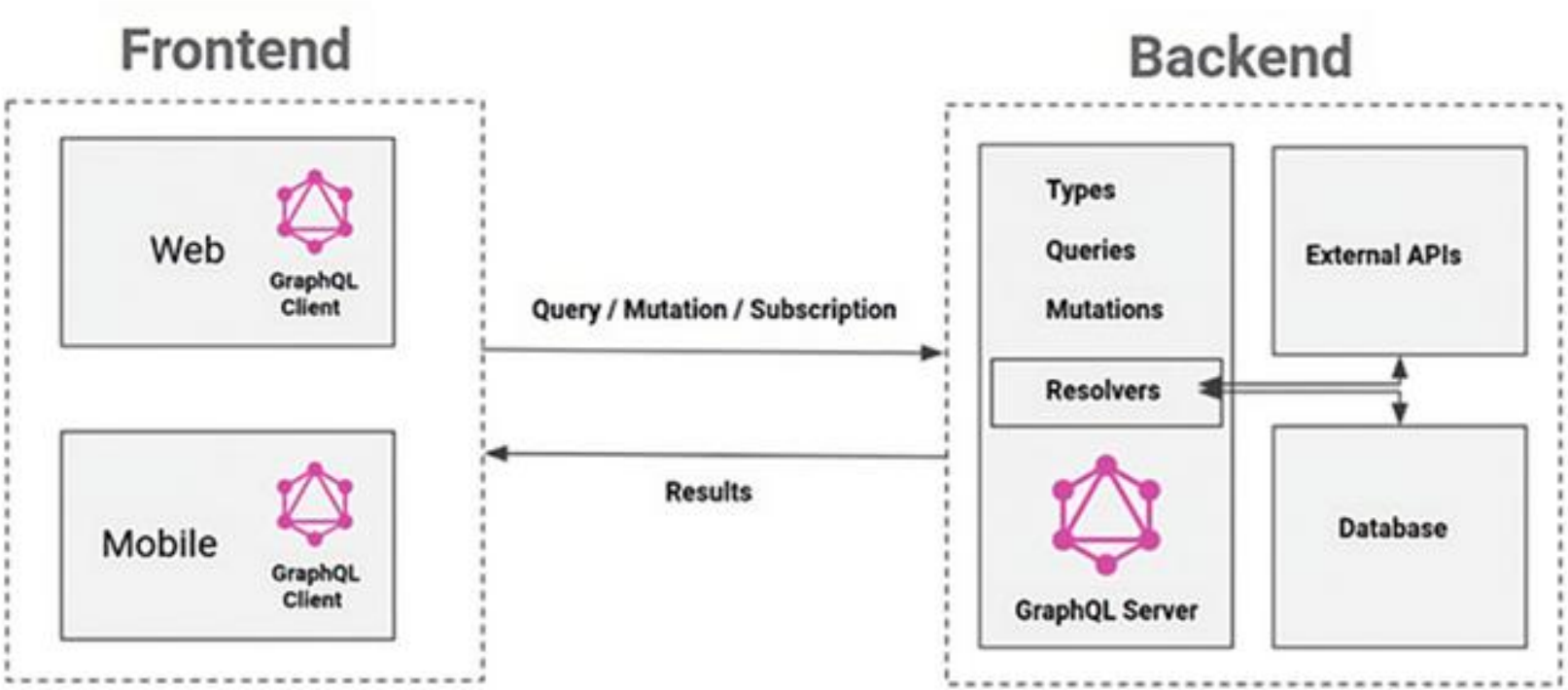
# 1.1. Introducción a GraphQL. Características

**Validación de consultas**: Permite validar las consultas en tiempo de compilación para asegurarse de que cumplan con el esquema.

**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.

# 1.1. Introducción a GraphQL. Arquitectura



# 1.1. Introducción a GraphQL. Arquitectura

**Servidor**

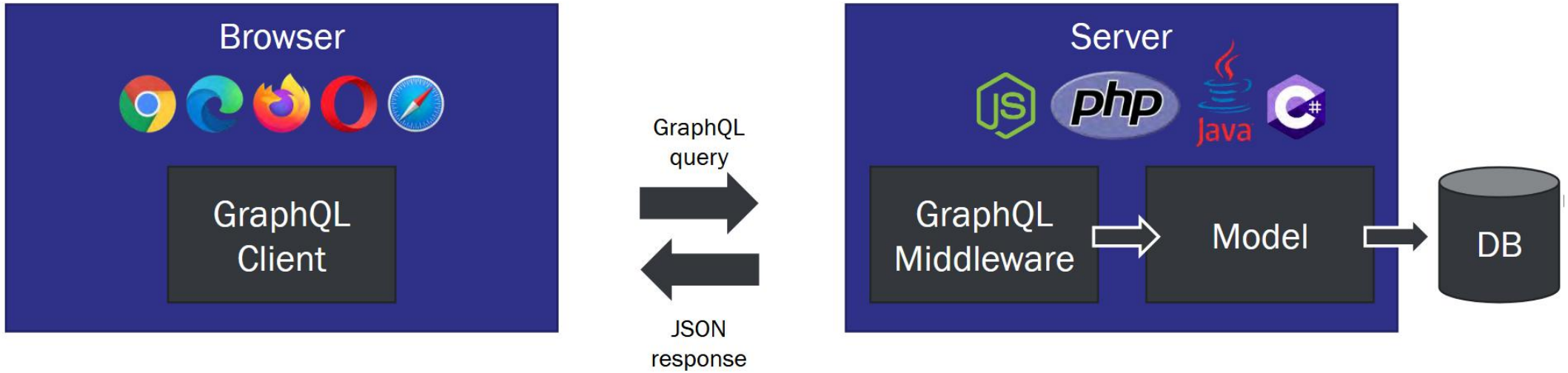
- Componente o aplicación que implementa la funcionalidad API GraphQL.
  - Puede ser construido utilizando diferentes lenguajes de programación y *frameworks*.
- Permite a los clientes realizar consultas, mutaciones y suscripciones a través de una API GraphQL.
- Se encarga de recibir las solicitudes GraphQL, interpretarlas y proporcionar las respuestas correspondientes.
- **Todo servidor GraphQL debe implementar al menos un esquema, usando el lenguaje de esquemas o mediante programación, y los resolutores asociados.**

# 1.1. Introducción a GraphQL. Arquitectura

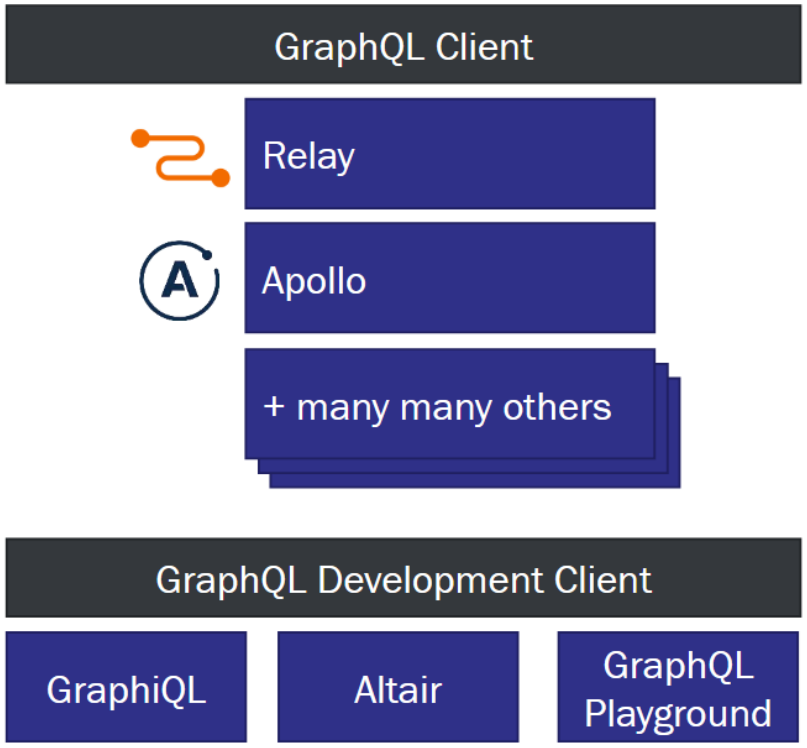
**Cliente**

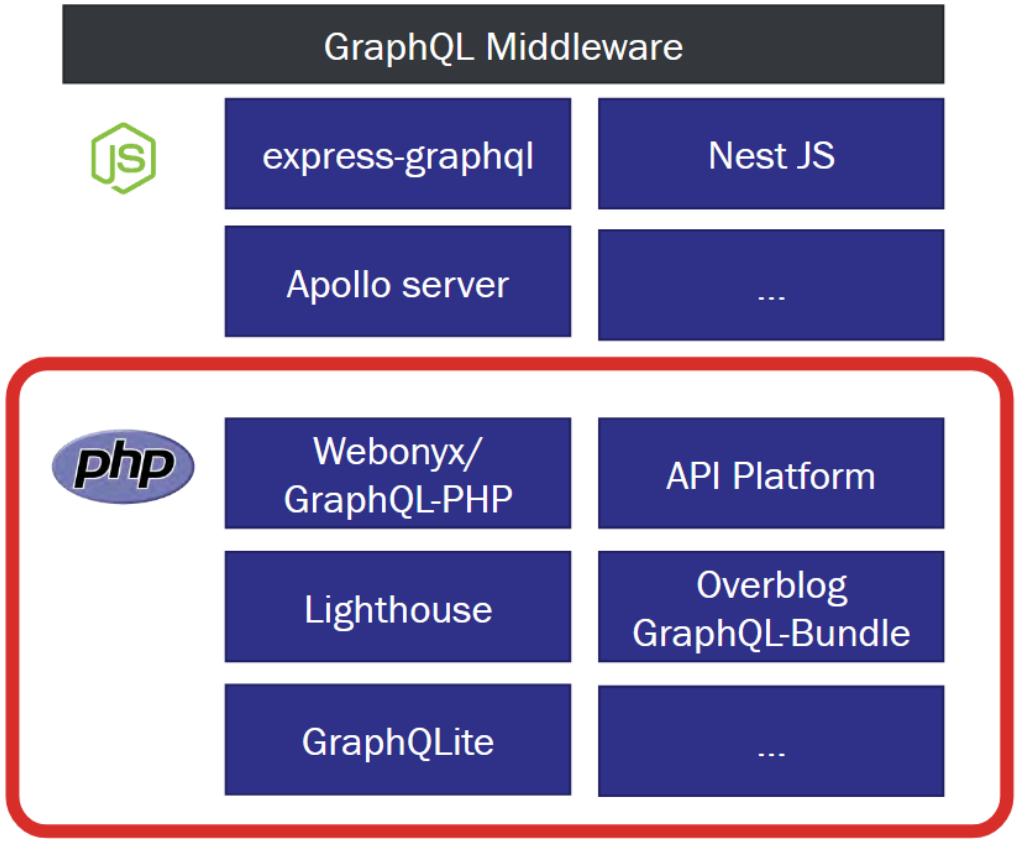
- Envía las solicitudes a través de HTTP o cualquier otro protocolo compatible con GraphQL.
- Puede ser una aplicación web, aplicación móvil u otros sistemas que puedan consumir datos de la API GraphQL directamente.
- Es necesario utilizar una biblioteca o cliente GraphQL para interactuar con una API GraphQL
- **Para realizar una consulta es obligatorio incluir en la consulta el campo `query` con el contenido que se quiere consultar, utilizando el lenguaje de consultas.**

# 1.2. Configuración del entorno

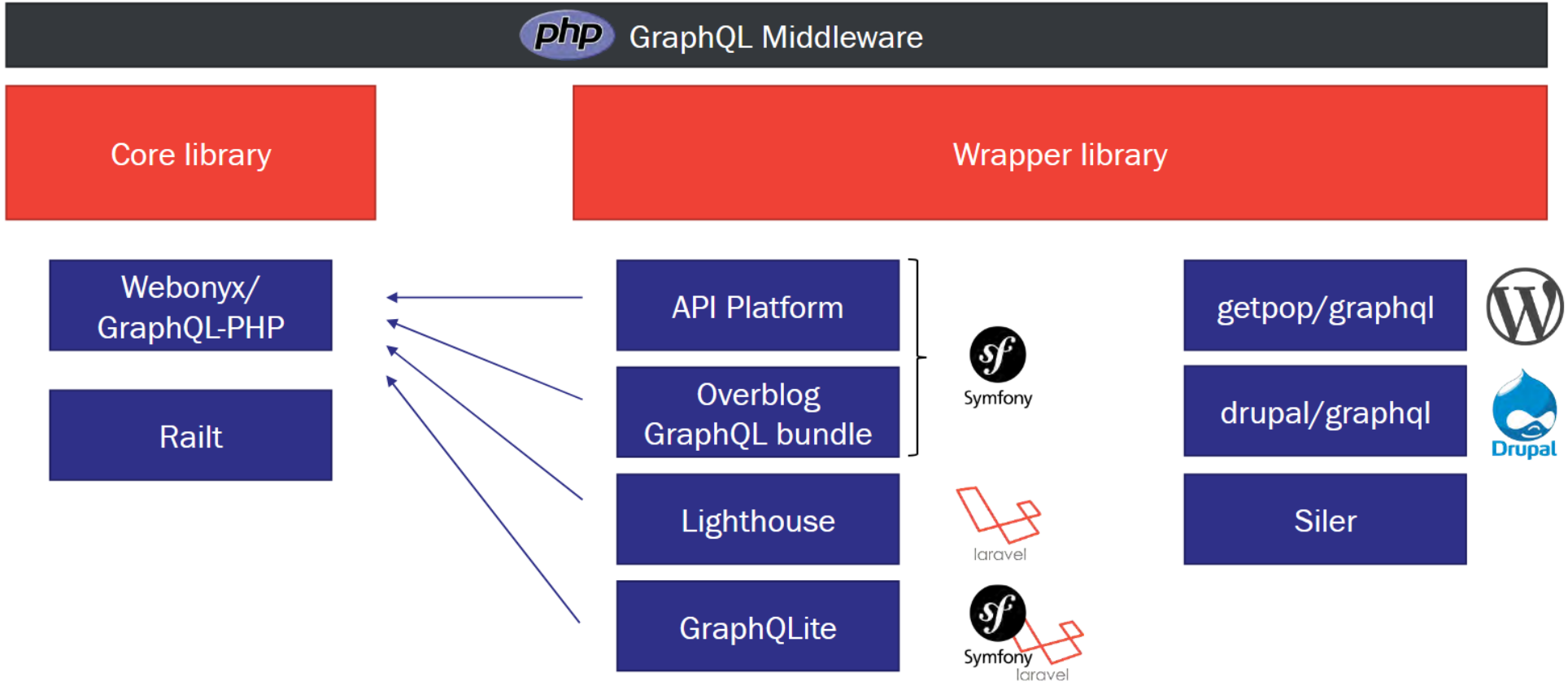


# 1.2. Configuración del entorno



# 1.2. Configuración del entorno



# 1.2. Configuración del entorno

La primera elección a realizar es si usar la librería base o una librería que encapsula la principal, un *wrapper*.

Entorno de desarrollo:

- Servidor web Apache con PHP 8.2 y Composer.
- Servidor MySQL o MariaDB.
- IDE de desarrollo: VSCode o similar.
  - Extensión **GraphQL: Language Feature Support**
- cURL
  - https://curl.se/windows/

# 1.2. Configuración del entorno. Uso de Composer

**Composer** es una aplicación que se lanza desde el terminal, bien con un fichero llamado `composer.json` o bien nombrando el paquete.

Al lanzarlo creará una carpeta llamada `vendor` que almacenará todas librerías PHP instaladas en el directorio actual.

- Esta ruta se puede utilizar para instalar todas las librerías necesarias.

# 1.2. Configuración del entorno. Uso de Composer

En XAMPP, la forma más sencilla de instalar una librería PHP con Composer es tenerlo dentro del *path* y lanzar su comando por terminal dentro de la carpeta donde se quiere guardar la librería.

```
composer require webonyx/graphql-php
```

Si usamos un contenedor Docker, hay que ejecutar el terminal del contenedor para poder lanzar la consulta

```
docker exec -it apache_php bash
composer require webonyx/graphql-php
```

# 1.2. Configuración del entorno. Uso de Composer

Una vez instalada la librería, se crea una carpeta llamada `vendor`.

- **Para GitHub o similar, es imprescindible crear un fichero de exclusión (.gitignore) que no sincronice el directorio `vendor`, los ficheros `composer.json` y `composer.lock`.**

En los scripts donde se quiera usar la librería habrá que incluir el fichero de carga de composer y el *namespace* adecuado.

- El fichero es el mismo para todas las librerías instaladas con Composer.

```
require_once 'vendor/autoload.php';

use GraphQL\Utils\BuildSchema;
```

# 1.2. Configuración del entorno. cURL

cURL (Client URL) es una herramienta de línea de comandos que permite transferir datos hacia o desde un servidor sin interacción del usuario utilizando la biblioteca libcurl.

```
curl <options> url
```

Opciones de línea de comandos:

- **-O, --remote-name** guarda el archivo en la carpeta actual con mismo nombre del remoto.
- **-o, --output** guarda el archivo permitiendo especificar un nombre o ubicación diferente.
- **-d**, **--data** envía datos en formato `'clave=valor'` como un *submit* de POST
- **--data-binary** envía datos tal y como se introducen sin procesarlos
- **-H, --header** incluye una cabecera extra

# 1.3. Hola mundo

Para tener un acceso a la API debemos crear una carpeta en nuestro directorio `www` donde se realizarán las llamadas.

- Si la carpeta se llama `graphql`, las llamadas se harán a la URL `http://localhost:8080/graphql/`
- También se puede crear un nuevo VirtualHost en la configuración de Apache en un puerto libre y lanzar allí las peticiones.

# 1.3. Hola mundo. Esquema

Lo primero que hay que crear es el esquema GraphQL.

Para ello, en el directorio `graphql` debemos crear un fichero llamado **`schema.graphql`** que contenga en esquema más sencillo posible: una consulta con una función llamada `hello` que devuelva una cadena de texto.

```
type Query {
    hello: String!
}
```

# 1.3. Hola mundo. Resolver

En el mismo directorio habrá que crear un fichero llamado `rootresolver.php`.

- Simplemente mostrará la cadena de texto `Hola mundo` al llamar al campo `hello`.

```
<?php

$root_fields_Resolver = [
    'hello' => 'Hola mundo',
];
```

# 1.3. Hola mundo. Punto de entrada GraphQL (1/2)

En el mismo directorio habrá que crear el fichero `index.php`, que será el punto de entrada a todas las llamadas del servicio GraphQL.

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

require_once __DIR__ . '/../vendor/autoload.php';
include_once __DIR__ . '/rootresolver.php';
use GraphQL\Utils\BuildSchema;
use GraphQL\Server\StandardServer;
```

# 1.3. Hola mundo. Punto de entrada GraphQL (2/2)

```php
$contents = file_get_contents(__DIR__. '/schema.graphql');
$schema = BuildSchema::build($contents);
$server = new StandardServer([
    'schema' => $schema,
    'rootValue' => $root_fields_Resolver,
]);
try{
    $server->handleRequest();
}catch (Exception $e){
    echo json_encode(['error' => $e->getMessage()]);
}
```

# 1.3. Hola mundo. Prueba

Para probar la aplicación, como aún no tenemos montado el cliente, podemos utilizar cURL con las cabeceras adecuadas.

- Es importante que la URL acabe en /

```
curl http://localhost:8080/graphql/ -H "Content-Type: application/json" -H "Accept: application/graphql-response+json" --data-binary "{\"query\":\"query {hello}\"}"
```

```
D:\Users\Aaron>curl http://localhost:8080/graphql/ -H "Content-Type: application/json" -H
"Accept: application/graphql-response+json" --data-binary "{\"query\":\"query {hello}\"}"
{"data":{"hello":"Hola mundo"}}
```

# 1.3. Hola mundo. Explicación

# 1.3. Hola mundo. Explicación. Petición

Desde el cliente HTTP cURL se lanzado una petición GET al servidor en formato JSON, utilizando el lenguaje de consulta GraphQL.

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

El único campo obligatorio en una consulta GraphQL es `query`, que debe especificar qué se quiere realizar: una `query` llamada `hello`.

Hay que distinguir entre:

- la palabra clave `query` que pertenece a la estructura de la consulta.
- el valor `query` del campo de la ejecución de la solicitud, que puede ser también `__schema`, `mutation` o `subscription`.

# 1.3. Hola mundo. Explicación. Respuesta

A dicha petición, el servidor ha respondido también en formato JSON con los datos programados en el servicio: la frase “Hola Mundo”.

```
{
    "data": {
        "hello": "Hola mundo"
    }
}
```

La respuesta incluye un campo `data` con la correspondiente respuesta:

- el nombre la acción requerida `hello`.
- los datos `Hola mundo`.

# 1.3. Hola mundo. Explicación. Respuesta

En caso que existieran errores en la ejecución, aparecería un campo con la información en el nivel superior.

```
{
    "errors": [
        {"message":"GraphQL Request must include at least
 one of those two parameters: \"query\" or \"queryId\""}
    ]
}
```

Se puede probar este comportamiento conectándose directamente desde un navegador a la API a la URL http://localhost:8080/graphql/.

# 1.3. Hola mundo. 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`.

Sobre el servicio de ejemplo vamos a montar una consulta de introspección que devuelva los nombres de los campos existentes.

```
curl http://localhost:8080/graphql/ -H "Content-Type:
application/json" -H "Accept: application/graphql-response+
json" --data-binary "{\"query\":\"{ __schema {queryType
{fields {name}}}}\"}"
```

# 1.3. Hola mundo. Introspección

```
{
    "query":"{__schema {queryType {fields {name}}}}"
}
```

```
D:\Users\Aaron>curl http://localhost:8080/graphql/ -H "Content-Type: application/json"
-H "Accept: application/graphql-response+ json" --data-binary "{\"query\":\"{ __schema
{queryType {fields {name}}}}\"}"
{"data":{"__schema":{"queryType":{"fields":[{"name":"hello"}]}}}}
```

- La respuesta muestra un campo que se puede solicitar que se llama `hello`.
- La capacidad de introspección es muy útil como fuente de documentación para los desarrolladores.

# 1.3. Hola mundo. Introspección


```
{
    "data": {
        "__schema": {
            "queryType": {
                "fields": [
                    {"name":"hello"}
                ]
            }
        }
    }
}
```

# 1.3. Hola mundo. Cliente

# 1.3. Hola mundo. Cliente

El cambio principal de GraphQL respecto de a una API REST es que no se puede hacer una petición desde el navegador directamente, ya que se necesita enviar un fichero JSON.

Por ello, es buena idea tener un cliente genérico para poder depurar el servidor GraphQL.

Lo más sencillo es una página HTML sencilla con una pequeña función JavaScript.

# 1.3. Hola mundo. Cliente. HTML

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>GraphQL Client</title>
  <script src="cliente_graphql.js"></script>
</head>
<body>
  <p><label for="send">query</label></p>
  <p><textarea id="send" rows="5"
    cols="80">query { hello }</textarea></p>
  <p><label for="vars">variables</label></p>
  <p><textarea id="vars" rows="5"
    cols="80"></textarea></p>
  <p><button onclick="getQueryString()"
    type="button">¡Vamos!</button></p>
  <p><label for="resp">respuesta</label></p>
  <p><textarea id="resp" rows="10"
    cols="80" readonly></textarea></p>
</body>
</html>
```

# 1.3. Hola mundo. Cliente. JavaScript

```
function getQueryString() {
  const url =
    'http://localhost:8080/graphql/';
  const resp =
    document.getElementById('resp');
  const query =
    document.getElementById('send').value;
  const vars =
    document.getElementById('vars').value;
  fetch(url, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({query, vars})
  })
  .then(result => result.json())
  .then(json => {
    resp.value =
      JSON.stringify(json,null,'  ');
  })
  .catch(error => {
    resp.value = error.stack;
  });
}
```

# Actividad 1.1

Utiliza el cliente HTML creado para tu servicio *Hola mundo* para hacer consultas en la PokeApi GraphSQL.

https://graphql.pokeapi.co/v1beta2

curl https://graphql.pokeapi.co/v1beta2 -H "Content-Type: application/json" -H "Accept: */*" --data-binary " { \"query\": \"query getItems{item{name,cost}}\", \"variables\": null, \"operationName\":\"getItems\" }"

# 1.4. Lenguaje de esquemas

El esquema de GraphQL es la base de cualquier implementación de servidor GraphQL. Los esquemas de GraphQL se escriben en el lenguaje de definición de esquema (SDL).

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.

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.

# 1.4. Lenguaje de esquemas

**Tipos**

- Definen de **la forma y el comportamiento** de los datos.
- Cada tipo que se defina en su esquema tendrá su propio ámbito.
  - Dentro del ámbito puede haber 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.
- Los tipos más comunes son los objetos o los escalares (tipos de valores primitivos).

# 1.4. Lenguaje de esquemas

**Campos**

- Contienen el **valor** que se solicita al servicio GraphQL.
  - Se parecen mucho a las variables de otros lenguajes de programación.
- Existen dentro del ámbito de un tipo.
- La definición de los campos de datos determinará cómo se estructuran 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.

# 1.4. Lenguaje de esquemas. Categorías de tipos

Los esquemas más simples contienen tres categorías de datos diferentes:

- **Raíz del esquema**
  - **Palabra clave `schema`**.
  - Definen los puntos de entrada de su esquema.
  - Señalan los campos que realizarán alguna operación con los datos:
    - **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.

# 1.4. Lenguaje de esquemas. Categorías de tipos

- **Tipos de objetos especiales**
  - Definen el comportamiento de las operaciones del esquema.
    - Cada campo de cada tipo de objeto especial define una sola operación a implementar.
  - Se nombran en la raíz del esquema, y se define cada tipo una vez en el cuerpo del esquema.
    - El único obligatorio es `Query`, mientras que `Mutation` y `Subscription` son opcionales.
- **Tipos**
  - Se utilizan para representar la forma de los datos.
  - Son similares a los objetos o representaciones abstractas con características definidas.
    - Un objeto *Persona* puede representar a una persona en una base de datos.
    - Las características de cada persona se definirán dentro de *Persona* como campos.
    - Los campos pueden ser cualquier cosa: el nombre, la edad, el trabajo, la dirección, etc.

# 1.4. Lenguaje de esquemas

```
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
}
```

# 1.4. Lenguaje de esquemas. Tipos especiales

Un esquema GraphQL soporta al menos tres clases de *root operations*, operaciones al más alto nivel, o formas de ejecutar acciones sobre el API.

- Query
  - Indica al sistema que se desea un conjunto de datos
  - La petición vendrá en formato JSON indicando exactamente los datos a devolver, que también se devolverán en formato JSON.
  - Es obligatorio al menos definir un tipo Query.
- Mutation
  - Operaciones de modificación sobre algún elemento del sistema (inserción, modificación)
- Subcription
  - El servidor se compromete a enviar un flujo de datos futuro al cliente.

# 1.4. Lenguaje de esquemas. Tipos especiales

```
type MyQuery {
    hello: String
    user(id: ID!): user
    users: [User]
    getRole(name:ID!): Role!
}

type MyMutation {
    createUser(name: String!,
      email: String!): User
}
```

```
type MySubscription {
    echo(name:ID!): Episode!
}
schema {
    query: MyQuery,
    mutation: MyMutation,
    subscription: MySubscription
}
```

# 1.4. Lenguaje de esquemas. Tipos escalares predefinidos

GraphQL incorpora un conjunto de tipos predefinidos:

- Int: entero de 32 bits como signo
- Float: valor en coma flotante de doble precisión con signo
- String: cadena UTF-8 de caracteres
- Boolean: un valor true o false.
- ID: identificador único, generalmente unido a un objeto como clave primaria
  - en implementación es similar a un String, pero sin significado para la lectura.

# 1.4. Lenguaje de esquemas. Tipos lista y objetos

**Lista**

Es posible definir el tipo de un campo como una lista de objetos utilizando corchetes [].

- El campo **`libros: [Libro]`**, indica que `libros` es una lista de tipo `Libro`.

**Objetos**

Son los componentes más básicos de un esquema

Representan un tipo de dato que se puede obtener

- En su cuerpo aparecerán todos los campos constitutivos con el tipo correspondiente
- Pueden usar otros tipos.

# 1.4. Lenguaje de esquemas. Tipos lista y objetos

**Modificadores de objetos**

La existencia de un cierre de exclamación tras un tipo indicará que el valor correspondiente no puede ser nulo, siempre devolverá algún valor

- **`String!`** indica que será del tipo `String` y no puede ser nulo.
- **`[Usuario]!`** obliga a que se devuelva una lista, vacía o con elementos del tipo `Usuario`.
- **`[Usuario!]!`** no permite devolver un elemento nulo, obliga a devolver una lista con al menos un elemento

# 1.4. Lenguaje de esquemas. Enumeraciones

Las enumeraciones permiten restringir el valor de un campo a los elementos descritos en dicha enumeración.

- Conceden un significado más concreto y expresivo a los campos.

```
enum Role {
    TRAINER
    RESEARCHER
}
type Query {
    getRole(name:ID!): Role!
}
```

# 1.4. Lenguaje de esquemas. Interfaces

En un lenguaje de programación, un interfaz es un conjunto de métodos que deben ser implementados obligatoriamente por la clase que los usa.

- El compartir especificación facilita la implementación de protocolos.

En GraphQL 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 no pueden implementarse así mismos de forma cíclica.

# 1.4. Lenguaje de esquemas. Interfaces. Ejemplo

```
type Author {
    firstName: String!
    lastName: String!
}
type Book
 implements LibItem {
    id: ID!
    title: String!
    authors: [Author!]
}
```

```
type BoardGame
 implements LibItem {
    id: ID!
    title: String!
    pub: Publisher!
}
interface LibItem {
    id: ID!
    title: String!
}
```

```
type Publisher {
    name: String!
}
type Query {
    libItems: [LibItem!]
}
schema {
    query: Query
}
```

# 1.4. Lenguaje de esquemas. Interfaces. Ejemplo

En el ejemplo se definen

- los tipos necesarios para crear el esquema `Publisher`, `Author` .
- el interfaz **`LibItem`** que es implementado por Book y por `BoardGame`.
- un punto de entrada **`libItems`** para las consultas, que devolverá una lista de `LibItem`
  - Esta consulta puede retornar tanto Book como `BoardGame`.

# 1.4. Lenguaje de esquemas. Uniones

Consiste en la posibilidad de utilizar dos tipos diferenciados a la vez sin compartir información.

- Es posible devolver cualquier de ellos.
- Los tipos definición en una unión tienen que ser concretos, no pueden ser interfaces u otras uniones.
- El principal cambio de la definición en el esquema respecto a los interfaces es que en las uniones es obligatorio repetir cada campo en ambos tipos, mientras que en los interfaces el tipo aparece en la parte común.

# 1.4. Lenguaje de esquemas. Uniones. Ejemplo

```
type Author {
    firstName: String!
    lastName: String!
}
type Book {
    id: ID!
    title: String!
    authors: [Author!]
}
```

```
type BoardGame {
    id: ID!
    title: String!
    pub: Publisher!
}

union LibItem =
  Book | BoardGame
```

```
type Publisher {
    name: String!
}
type Query {
    libItems: [LibItem!]
}
schema {
    query: Query
}
```

# 1.4. Lenguaje de esquemas. Inputs

Un tipo input se define con los campos que deberán pasarse al parámetro correspondiente y se usará en la definición del campo.

- Esta funcionalidad sirve para soportar el paso de argumentos complejos.
- Esta característica es muy útil sobre todo en las mutaciones, en la que hay que pasar gran información a la API para que realice su trabajo.
- **Los inputs no pueden tener argumentos definidos**.

# 1.4. Lenguaje de esquemas. Inputs. Ejemplo

```
enum Episode {
    NEWHOPE
    EMPIRE
    JEDI
}

input EpisodeInput {
    firstName: String!
    date: String!
}
```

```
type Query {
    echo(name:EpisodeInput!):
 Episode!
}

schema {
    query: Query,
}
```

# 1.4. Lenguaje de esquemas. 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.
- Existe la posibilidad también de definir nuestras propias directivas.

Una directiva importante es `@deprecated`, que se puede utilizar previamente a eliminar un campo o tipo que vaya a ser suprimido del sistema.

- Así se les ayuda a los clientes a que se adapten a los cambios de forma eficiente.

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

# 1.4. Lenguaje de esquemas. Documentación

GraphQL permite agregar documentación a los tipos, campos y argumentos de un esquema.

- De hecho, la especificación GraphQL incita 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.

También se pueden agregar comentarios de una sola línea al SDL anteponiendo un # carácter al texto a comentar.

- También se pueden añadir comentarios a las consultas cliente.

# 1.4. Lenguaje de esquemas. Ejemplo. SDL

```
type Alumno {
  id: ID!
  nombre: String!
  nota: Int!
}


type Query {
  alumnos: [Alumno!]!
  alumnoPorId(id: ID!): Alumno
}
```

# 1.4. Lenguaje de esquemas. Ejemplo. PHP (1/5)

```php
<?php
require_once __DIR__ . '/../../../vendor/autoload.php';
use GraphQL\Type\Definition\Type;
use GraphQL\Type\Definition\ObjectType;
use GraphQL\Server\StandardServer;
use GraphQL\Utils\BuildSchema;

$alumnos = [
    ["id" => 1, "nombre" => "Ana", "nota" => 8],
    ["id" => 2, "nombre" => "Luis", "nota" => 6],
    ["id" => 3, "nombre" => "Marta", "nota" => 9],
];
```

# 1.4. Lenguaje de esquemas. Ejemplo. PHP (2/5)

```
class AlumnoType extends ObjectType {
    public function __construct() {
        parent::__construct([
            'name' => 'Alumno',
            'fields' => [
                'id' => Type::nonNull(Type::id()),
                'nombre' => Type::nonNull(Type::string()),
                'nota' => Type::nonNull(Type::int())
            ]
        ]);
    }
}
```

# 1.4. Lenguaje de esquemas. Ejemplo. PHP (3/5)

```php
class MyTypes {
    private static ?AlumnoType $alumno = null;


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

# 1.4. Lenguaje de esquemas. Ejemplo. PHP (4/5)

```php
$rValue = [
    'alumnos' => function() use ($alumnos) { return $alumnos; },
    'alumnoPorId' => function($root, $args) use ($alumnos) {
        foreach ($alumnos as $alumno) {
            if ($alumno['id'] == $args['id']) {
                return $alumno;
            }
        }
        return null;
    },
];
```

# 1.4. Lenguaje de esquemas. Ejemplo. PHP (5/5)

```php
$schemaSDL = file_get_contents(__DIR__ . '/schema.graphql');
$schema = BuildSchema::build($schemaSDL);
$server = new StandardServer([
    'schema' => $schema,
    'rootValue' => $rValue,
]);
try{
    $server->handleRequest();
} catch (Exception $e){
    echo json_encode(['error' => $e->getMessage()]);
}
```

# Actividad 1.2

Realiza las siguientes consultas sobre el esquema de alumnos:

- ```
  curl http://localhost:8080/pdauf/ut01/act02_alumnos/ -H "Content-Type: application/json" -H "Accept: application/graphql-response+json" --data-binary "{\"query\":\"query { alumnos { id nombre nota } } \"}"
  ```

- ```
  curl http://localhost:8080/pdauf/ut01/act02_alumnos/ -H "Content-Type: application/json" -H "Accept: application/graphql-response+json" --data-binary "{\"query\":\"query { alumnoPorId(id: 2) { nombre nota } } \"}"
  ```

# 1.5. Lenguaje de consultas

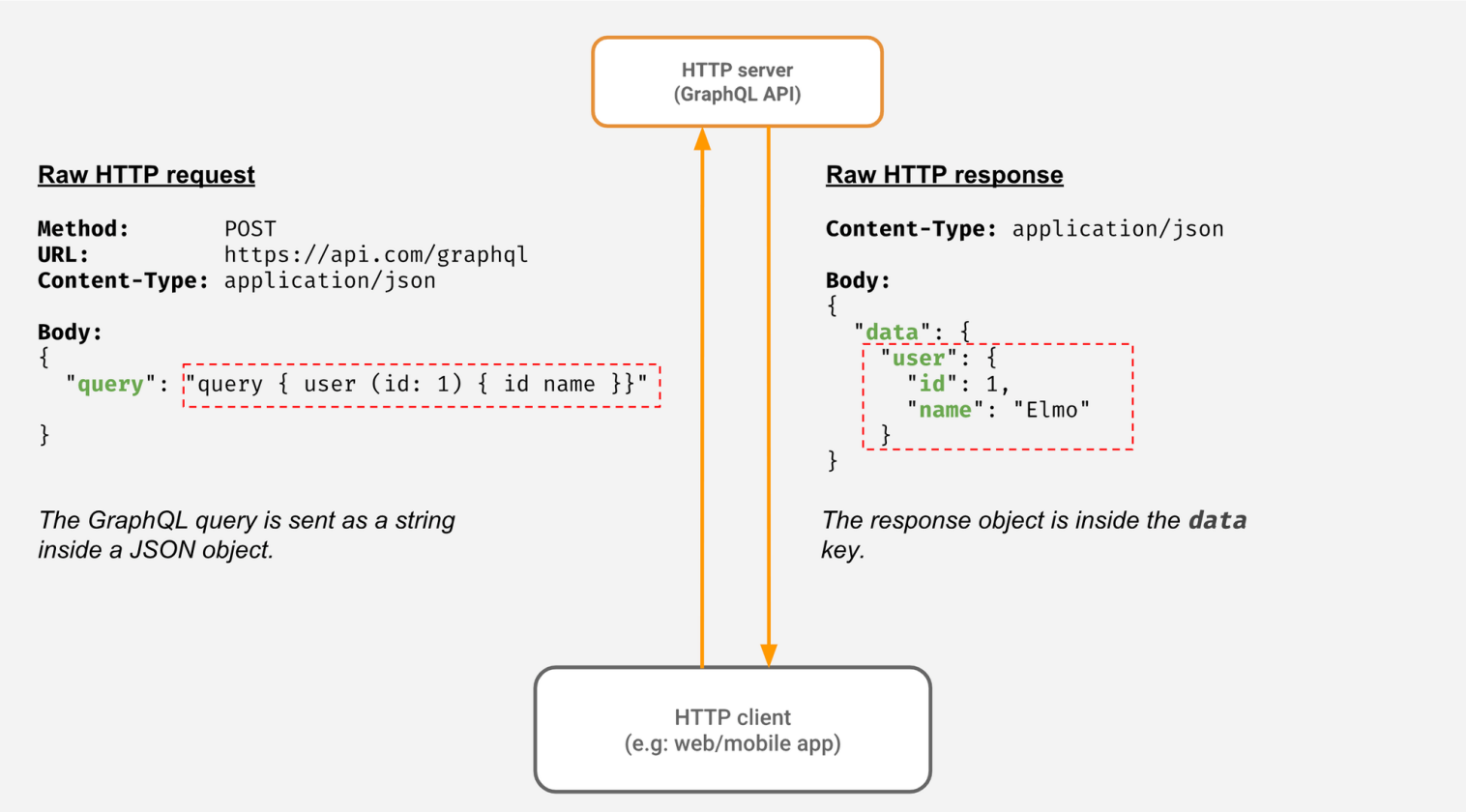
El lenguaje de consultas es un sistema formal utilizado para realizar solicitudes y extraer información de una fuente de datos.

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.**

- Si dentro de algún campo hay que usar las comillas dobles, habrá que anteponer la barra invertida \.

# 1.5. Lenguaje de consultas. Estructura de una consulta



# 1.5. Lenguaje de consultas. Estructura de una consulta

Para poder realizar consultas a través de un cliente hay que utilizar la estructura concreta que está definida en el estándar para que se entienda por parte del servidor.

El objeto en formato JSON a crear debe estar formado por dos campos

- query, obligatorio
- variables, opcional

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

# 1.5. Lenguaje de consultas. Estructura de una consulta

Los clientes y servidores GraphQL deben admitir JSON para la serialización.

- Los clientes también deben indicar que aceptan la respuesta GraphQL en el encabezado `Accept` del protocolo HTTP, indicado el valor `application/graphql-response+json`.

Respecto a la codificación, si no se incluye información se asumirá que es utf-8

- El servidor puede responder con un error si un cliente no incluye el encabezado `Accept` en la petición, como `charset=utf-8`.

La etiqueta `Content-Type` de la petición debe ser `application/json`.

El servidor HTTP de GraphQL debe poder procesar las peticiones HTTP con el método GET y POST para las operaciones de consulta, y solo las peticiones POST para las operaciones de mutación.

# 1.5. Lenguaje de consultas. Elementos. Consulta

La estructura de una consulta GraphQL es siempre la misma

- Primero el tipo de operación raíz (tipo de consulta)
  - `query`, `mutation` o `subscription`.
- A continuación, el cuerpo de la consulta
  - Donde se definirán los campos que se quiere recoger.
  - Si el tipo de consulta es `query` se puede omitir.

# 1.5. Lenguaje de consultas. Elementos. Consulta

En su forma más simple, GraphQL consiste en solicitar campos específicos de objetos.

- Si los campos tienen argumentos obligatorios, deberán aparecer a continuación del nombre del campo, entre paréntesis, con el nombre del argumento.

En un sistema REST, solo se puede pasar un único conjunto de argumentos (los parámetros de consulta y los segmentos de URL de la solicitud).

En GraphQL, cada campo y objeto anidado puede obtener su propio conjunto de argumentos

- GraphQL puede realizar búsquedas múltiples.

# 1.5. Lenguaje de consultas. Elementos. Respuesta

El resultado de una solicitud se devolverá en una clave de nivel superior.

- `data`, si la solicitud fue correcta
- `error`, si la solicitud generó algún error.
  - La especificación GraphQL gestiona los errores de resolución, pero el cliente tendrá que controlar todos aquellos errores relacionados con el protocolo que sean diferentes al OK/200, o posibles problemas de red.

# 1.5. Lenguaje de consultas. cURL

Hay que ser cuidadoso al utilizar cURL y anteponer la palabra clave "query" y las comillas escapadas correctamente.

```
query {
  libraryItems {
      title
  }
}
```

- `curl http://localhost:8080/pdauf/ut01/act02_library_interface/ -H "Content-Type: application/json" -H "Accept: application/graphql-response+json" --data-binary "{\"query\": \"query { libraryItems { title } } \" }"`

# 1.5. Lenguaje de consultas. cURL

```
query {
  libraryItems {
    ... on Book {
      id
      title
      authors {
        firstName
        lastName
      }
    }
  }
}
```

- curl http://localhost:8080/pdauf/ut01/act02_library_interface/ -H "Content-Type: application/json" -H "Accept: application/graphql-response+json" --data-binary "{\"query\": \"query { libraryItems { id title ... on Book { authors { firstName lastName } } } } \" }"

# 1.5. Lenguaje de consultas. Directivas

- `@include(if: Boolean)`. Solo incluye este campo o fragmento en el resultado si el argumento es verdadero.
- `@skip(if: Boolean).` Salta el campo o fragmento si la condición es verdadera.

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

# 1.6. Otras operaciones. Introspección

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.

# 1.6. Otras operaciones. Mutaciones

Operaciones que presenta el API para realizar cambios en el sistema

Solo este tipo es el que puede realizar cualquier tipo modificaciones

- la resolución de campos distintos de los campos de mutación (`mutation`) de nivel superior debe estar libre de efectos secundarios.
- el servidor GraphQL por sí solo no realiza ninguna acción de la `mutation`, debe ser en el correspondiente *resolver* el que realice la acción y devuelva un resultado según el tipo indicado en el esquema.

# 1.6. Otras operaciones. Mutaciones

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, como las consultas, se pueden agrupar en una única petición, nombrándolas con un alias-

- Hay que tener en cuenta que estas peticiones se ejecutarán de forma síncrona según el orden en el que se haya creado la mutación, de arriba abajo, pero cada una de forma independiente.

# 1.6. Otras operaciones. Mutaciones: SDL

```
enum Episode {
    NEWHOPE
    EMPIRE
    JEDI
}
input ReviewInput {
    stars: Int!
    commentary: String
}
type Mutation {
    createReview(episode: Episode, review: ReviewInput!): review
}
```

# 1.6. Otras operaciones. Mutaciones: Consulta

```
mutation createReviewForEpisode($ep: Episode!, $review: ReviewInput!){
    createReview(episode: $ep, review: $review!) {
        starts
        commentary
    }
}
```

# 1.6. Otras operaciones. Suscripciones

Las suscripciones de GraphQL están respaldadas por un sistema de publicación independiente.

GraphQL no especifica qué protocolo de transporte utilizar

- En la práctica se utiliza WebSockets o eventos enviados por el servidor.
  - Existen especificaciones mantenidas por la comunidad para implementar suscripciones GraphQL con WebSockets y eventos enviados por el servidor.
- Los clientes que desean enviar operaciones de suscripción también deberán admitir el protocolo elegido.

# 1.6. Otras operaciones. Suscripciones

Las operaciones de suscripción son una característica poderosa de GraphQL, pero son muy complicadas de implementar.

- 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 y el protocolo de transporte.

# 1.6. Otras operaciones. Suscripciones

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 antes posible.

- 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 mejores.

# 1.6. Otras operaciones. Suscripciones

```
#SDL
type Subscription {
    reviewCreated: Review
}
#Query
subscription NewReviewCreated {
    reviewCreated {
        rating
        commentary
    }
}
```

# 1.7. Ejecución. Validaciones

Una solicitud debe ser sintácticamente correcta para ser ejecutada, pero también debe ser válida cuando se la confronta al esquema.

- En la práctica, cuando una operación GraphQL llega al servidor, primero se analiza el documento y luego se valida mediante el sistema de tipos. 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.

# 1.7. Ejecución. Validaciones. Errores comunes

- 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.

# 1.7. Ejecución. Resolutores. 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.

# 1.7. Ejecución. Resolutores. Campos

Si un campo produce

- un valor escalar, como una cadena o un número, la ejecución se completa.
- un valor de objeto, la consulta contendrá otra selección de campos a aplicar a ese objeto.
  - Esto continúa hasta que se alcanzan tipos escalares o enumerados.

# 1.7. Ejecución. Resolutores. Campos raíz

El llamado *tipo query de operación raíz* o simplemente *tipo query* es un tipo de objeto en el nivel superior de un servidor GraphQL hay que representa los posibles puntos de entrada a la API.

Si la API también admite mutaciones para escribir datos y suscripciones para obtener datos en tiempo real, tendrá también tipos *mutation* y *subscription* que expongan campos sobre los que realizar este tipo de operaciones.

# 1.7. Ejecución. Resolutores. Campos raíz. Argumentos

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

# 1.7. Ejecución. Resolutores

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

# 1.7. Ejecución. Resolutores. Por defecto

**Todo campo tiene que tener su resolutor**

En la mayoría de las implementaciones, si la resolución es trivial, no hace falta que se proporcione.

En los casos en donde el valor a devolver es el nombre de una propiedad del objeto, el propio sistema es capaz de hacerlo.

- Si el objeto se llama `$obj`, y contiene una propiedad llamada `$nombre`, se puede acceder a ella como `$obj->nombre`.

# 1.7. Ejecución. Resolutores. Por defecto

La implementación sobre PHP también proporciona un resolutor por defecto para los casos en lo que el objeto es un a*rray* asociativo y la clave es igual al nombre a resolver: `$obj['nombre']`.

# 1.7. Ejecución. Resultado

A medida que se resuelve cada campo, el valor resultante se coloca en un array asociativo clave-valor (similar a un diccionario) con el nombre del campo (o alias) como clave, y el valor resuelto como valor.

Este proceso continúa desde los campos hoja inferiores de la consulta hasta el campo original en el tipo de consulta raíz.

Se produce una estructura jerárquica que refleja la consulta original que luego se puede enviar, normalmente como JSON, al cliente que hizo la petición.

# Actividad 1.3

Realiza 3 consultas distintas sobre

1. La API, adjunta en la plataforma, que contiene una unión.
2. La API de Rick y Morty disponible en este enlace
   - https://rickandmortyapi.com/graphql
     - Documentación: https://studio.apollographql.com/public/rick-and-morty-a3b90u/variant/current/schema/reference