FRAMEWORK LARAVEL

1. Introducción a las Bases de Datos.

La introducción a las bases de datos es fundamental para comprender cómo se organizan y gestionan los datos en un sistema de información. Aquí tienes una explicación básica sobre los elementos principales de las bases de datos y el modelo entidad-relación:

1.1 Base de Datos.

Una base de datos es un conjunto estructurado de datos organizados y almacenados electrónicamente en un sistema de computadora. Estas bases de datos pueden ser de diferentes tipos, como bases de datos relacionales (SQL) o bases de datos NoSQL (como MongoDB).

1.2 Elementos de una Base de Datos.

- **Tabla:** Es la estructura básica de una base de datos relacional. Una tabla consta de filas y columnas, donde cada fila representa un registro y cada columna representa un atributo.

- **Registro:** También conocido como fila, es una instancia individual de datos almacenados en una tabla.

- **Atributo:** También conocido como columna, es una característica o propiedad de un objeto o entidad que se almacena en la base de datos.

- **Clave primaria:** Es un campo o conjunto de campos que identifican de forma única cada registro en una tabla. No puede haber dos registros con la misma clave primaria.

- **Clave foránea:** Es un campo o conjunto de campos en una tabla que se relaciona con la clave primaria de otra tabla. Se utiliza para establecer relaciones entre tablas.

1.3 Modelo Entidad-Relación (ER).

El modelo entidad-relación es una herramienta poderosa para diseñar bases de datos relacionales de manera visual y comprensible. Ayuda a los diseñadores a entender las estructuras de datos necesarias para un sistema y a planificar la organización de los datos de manera eficiente.

El modelo entidad-relación se compone de tres componentes principales:

- **Entidades**: Son objetos o conceptos del mundo real que son importantes para el sistema que se está diseñando. Cada entidad tiene atributos que

describen sus características. Por ejemplo, en una base de datos para una tienda en línea, las entidades pueden incluir "Product", "Client" y "Order".

- **Atributos**: Son las propiedades o características que describen a una entidad. Por ejemplo, para la entidad "Client" en una tienda en línea, los atributos pueden incluir el name, el email, la addres, etc.

- **Relaciones**: Establecen las conexiones entre las entidades. Estas relaciones describen cómo las entidades se asocian entre sí y pueden ser de varios tipos, como uno a uno, uno a muchos o muchos a muchos. Por ejemplo, un cliente puede realizar varios pedidos, lo que representa una relación uno a muchos entre las entidades "Client" y "Order".

Ejemplo.

client: id, name, address, phone, email

product: id, name, description, amount, price

order: product_id, client_id, date_order, total_amount

En este diagrama:

- Client, Product y Prder son las entidades principales.

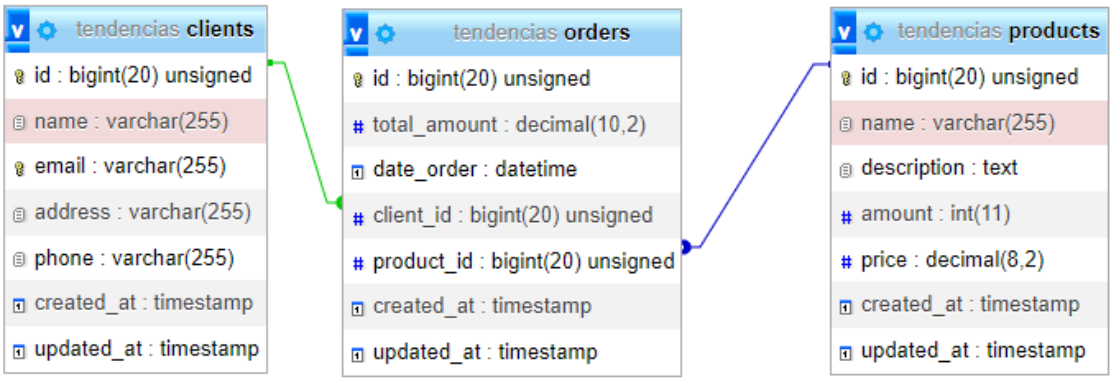
- Cada entidad tiene sus propios atributos. Por ejemplo, Client tiene: id, name, address, phone, email.

- La relación entre Client y Product se establece mediante las claves externas client_id y product_id en la entidad Order, que hacen referencia a las claves primarias de las entidades Client y Product, respectivamente.

- Un cliente puede realizar varios pedidos, lo que representa una relación uno a muchos entre Client y Order.

- Cada pedido está asociado a un cliente y a uno o más productos.

- Este diagrama representa una relación básica entre estas entidades. Dependiendo de los requisitos específicos del sistema, se podrían agregar más atributos y relaciones.



2. Base de Datos (Database) en Laravel

2.1 Configuración.

En Laravel, la configuración y gestión de la base de datos se realiza principalmente a través del archivo **config/database.php** y el archivo **.env**.

- **Configuración en .env**. En el archivo **.env**, puedes configurar las credenciales de la base de datos, como el nombre de la base de datos, el usuario y la contraseña. También puedes especificar el motor de base de datos que estás utilizando.

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=nombre_base_de_datos
DB_USERNAME=usuario_db
DB_PASSWORD=contraseña_db
```

- **Configuración en config/database.php**. Este archivo contiene la configuración de conexión a la base de datos, así como la configuración para múltiples conexiones de bases de datos si es necesario. Puedes especificar los detalles de conexión aquí o utilizar las variables de entorno definidas en **.env**.

```
'mysql' => [ 'driver' => 'mysql',
'host' => env('DB_HOST', '127.0.0.1'),
'port' => env('DB_PORT', '3306'),
'database' => env('DB_DATABASE', 'forge'),
'username' => env('DB_USERNAME', 'forge'),
'password' => env('DB_PASSWORD', ''), // Otros ajustes... ],
```

3. Migraciones.

Laravel utiliza migraciones para definir la estructura de la base de datos de manera programática. Las migraciones se encuentran en el directorio **database/migrations** y se utilizan para crear y modificar tablas de base de datos de manera controlada.

3.1 Migración Product

php artisan make:migration create_products_table

Esto generará un nuevo archivo de migración en el directorio **database/migrations**. Dentro de este archivo, puedes definir la estructura de la tabla de la siguiente manera:

```
<?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::create('products', function (Blueprint $table) {
            $table->id();
                        $table->string('name');
            $table->text('description')->nullable();
                        $table->integer('amount')->nullable();
            $table->decimal('price', 8, 2);
            $table->timestamps();
        });
    }

    public function down(): void
    {
        Schema::dropIfExists('products');
    }
};
```

Una vez que hayas definido la estructura de la tabla en la migración, puedes ejecutar la migración con el siguiente comando: **php artisan migrate**. Esto creará la tabla **products** en tu base de datos.

3.2 Migración Client

php artisan make:migration create_clients_table

Esto generará un nuevo archivo de migración en el directorio **database/migrations**. Dentro de este archivo, puedes definir la estructura de la tabla de la siguiente manera:

```
<?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::create('clients', function (Blueprint $table) {
            $table->id();
                        $table->string('name');
            $table->string('email')->unique();
                        $table->string('address');
                        $table->string('phone');
            $table->timestamps();
        });
    }

    public function down(): void
    {
        Schema::dropIfExists('clients');
    }
};
```

Una vez que hayas definido la estructura de la tabla en la migración, puedes ejecutar la migración con el siguiente comando: **php artisan migrate**. Esto creará la tabla **clients** en tu base de datos.

3.3 Migración Order

php artisan make:migration create_orders_table

Esto generará un nuevo archivo de migración en el directorio **database/migrations**. Dentro de este archivo, puedes definir la estructura de la tabla de la siguiente manera:

```php
<?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::create('orders', function (Blueprint $table) {
            $table->id();
            $table->decimal('total_amount', 10, 2);
                        $table->dateTime('date_order');
                        $table->unsignedBigInteger('client_id');
            $table->foreign('client_id')
                                ->references('id')
                                ->on('clients');
                        $table->unsignedBigInteger('product_id');
            $table->foreign('product_id')
                                ->references('id')
                                ->on('products');
            $table->timestamps();
        });
    }

    public function down(): void
    {
        Schema::dropIfExists('orders');
    }
};
```

Una vez que hayas definido la estructura de la tabla en la migración, puedes ejecutar la migración con el siguiente comando: **php artisan migrate**. Esto creará la tabla **orders** en tu base de datos.

Nota. Cada migración debe tener un método up() y un método down() . El propósito del método up() es realizar las operaciones requeridas para poner el esquema de la base de datos en su nuevo estado, y el propósito del método down() es revertir cualquier operación realizada por el método up() . Asegurarse de que el método down() invierta correctamente sus operaciones es fundamental para poder revertir los cambios del esquema de la base de datos.

3.1 Ejecución de migraciones

- Para ejecutar todas las migraciones pendientes, utiliza el comando **php artisan migrate**.

- Para ver el estado de las migraciones ejecutadas hasta el momento, usa el comando **php artisan migrate:status**.

- Para ver las sentencias SQL que ejecutarán las migraciones sin ejecutarlas realmente, añade la bandera **--pretend** al comando **php artisan migrate**.

- Para ejecutar las migraciones en un entorno de producción de manera aislada, utiliza la opción **--isolated** al invocar el comando **php artisan migrate**.

- Para forzar la ejecución de las migraciones sin confirmación, emplea la bandera **--force** con el comando **php artisan migrate**.

- Para revertir la última operación de migración, usa el comando **php artisan migrate:rollback**.

- Para revertir un número específico de migraciones, agrega la opción **--step** al comando **php artisan migrate:rollback**.

- Para revertir un "lote" específico de migraciones, proporciona la opción **--batch** al comando **php artisan migrate:rollback**.

- El comando **php artisan migrate:reset** revertirá todas las migraciones de tu aplicación.

- El comando **php artisan migrate:refresh** revertirá todas las migraciones y luego las ejecutará de nuevo.

- Para eliminar todas las tablas de la base de datos y migrar de nuevo, utiliza el comando **php artisan migrate:fresh**.

  Recuerda que algunos de estos comandos pueden ser destructivos, así que úsalos con precaución, especialmente en entornos de producción compartidos.

4. Modelos.

Los modelos Eloquent son clases PHP que representan tablas de la base de datos. Estos modelos se encuentran en el directorio **app/Models** por convención. Puedes definir relaciones entre modelos y utilizar métodos Eloquent para realizar consultas a la base de datos.

4.1 Creación de un Modelo.

Puedes crear un modelo en Laravel manualmente o utilizando el comando Artisan.

Por convención, los modelos se nombran en singular y en mayúscula la primera letra. Por ejemplo, si tienes una tabla llamada **productos**, el modelo correspondiente se llamaría **Producto**.

Puedes crear un modelo con el siguiente comando Artisan: **php artisan make:model Producto**. Esto generará un nuevo archivo de modelo en el directorio **app/Models**.

4.2 Definición de un Modelo.

El modelo define la estructura y la interacción con la tabla de la base de datos. Puedes definir la tabla asociada, los campos protegidos, las relaciones y otros métodos dentro del modelo.

4.2.1 Modelo Product.

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Product extends Model
{
    use HasFactory;

        protected $fillable = ['name', 'description', 'amount', 'price'];

        protected $guarded = ['id','created_at', 'updated_at'];

    public function orders()
    {
        return $this->hasMany(Order::class);
    }
}
```

4.2.2 Modelo Client.

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;
```

```
class Client extends Model
{
    use HasFactory;

        protected $fillable = ['name', 'email', 'address', 'phone'];

        protected $guarded = ['id','created_at', 'updated_at'];

    public function orders()
    {
        return $this->hasMany(Order::class);
    }
}
```

4.2.3 Modelo Order.

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Order extends Model
{
    use HasFactory;

        protected $fillable = ['product_id', 'client_id', 'date_order', 'total_amount'];

        protected $guarded = ['id','created_at', 'updated_at'];

    public function client()
    {
        return $this->belongsTo(Client::class);
    }

    public function product()
    {
        return $this->belongsTo(Product::class);
    }
}
```

En estos modelos:

- El modelo de "producto" (**Product**) tiene una relación de uno a muchos (**hasMany**) con el modelo de "pedido" (**Order**), indicando que un producto puede tener múltiples pedidos.

- El modelo de "cliente" (**Client**) también tiene una relación de uno a muchos (**hasMany**) con el modelo de "pedido" (**Order**), indicando que un cliente puede realizar múltiples pedidos.

- El modelo de "pedido" (**Order**) tiene una relación de pertenencia (**belongsTo**) con tanto el modelo de "cliente" (**Client**) como el modelo de "producto" (**Product**), indicando que un pedido pertenece a un cliente y a un producto específico.

En este ejemplo:

- La propiedad **$table** especifica el nombre de la tabla asociada al modelo.

- La propiedad **$fillable** especifica los campos que se pueden asignar masivamente.

- Puedes definir relaciones con otros modelos utilizando métodos como **hasMany**, **belongsTo**, **belongsToMany**, etc.

4.3 Uso del Modelo.

Una vez que has definido el modelo, puedes utilizarlo para interactuar con los datos de la base de datos en tu aplicación. Aquí tienes algunos ejemplos de cómo puedes usar el modelo **Producto**:

```
use App\Models\Producto;
// Crear un nuevo producto
Producto::create([
        'nombre' => 'Producto 1',
        'descripcion' => 'Descripción del producto 1',
        'precio' => 10.99,
]);
// Obtener todos los productos $productos = Producto::all();
// Obtener un producto por su ID $producto = Producto::find(1);
// Actualizar un producto $producto->nombre = 'Nuevo nombre';
$producto->save();
// Eliminar un producto $producto->delete();
```

5. Consultas SQL y Eloquent ORM.

Laravel proporciona un conjunto completo de métodos para realizar consultas a la base de datos utilizando el constructor de consultas, el Query Builder o el ORM Eloquent.

ORM Eloquent es el ORM (Object-Relational Mapping) integrado en Laravel que te permite interactuar con la base de datos utilizando objetos en lugar de consultas SQL directas. Proporciona una sintaxis expresiva y fácil de usar para realizar

operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en la base de datos. Aquí tienes una explicación detallada de cómo realizar consultas a la base de datos utilizando ORM Eloquent junto con ejemplos:

- **Obtener todos los registros**:
  use App\Models\Producto;
  $productos = Producto::all();
  Este código devuelve todos los registros de la tabla **productos**.

- **Obtener un registro por su ID**:
  $producto = Producto::find(1);
  Este código devuelve el registro de la tabla **productos** con el ID igual a 1.

- **Obtener registros que coincidan con un campo específico**:
  $productos = Producto::where('categoria', 'Electrónica')->get();
  Este código devuelve todos los registros de la tabla **productos** donde el campo **categoria** sea igual a "Electrónica".

- **Obtener el primer registro que coincida con un campo específico**:
  $producto = Producto::where('stock', '>', 0)->first();
  Este código devuelve el primer registro de la tabla **productos** donde el campo **stock** sea mayor que 0.

- **Crear un nuevo registro**:
  $producto = new Producto; $producto->nombre = 'Producto nuevo'; $producto->descripcion = 'Descripción del producto nuevo'; $producto->precio = 19.99; $producto->save();
  Este código crea un nuevo registro en la tabla **productos** con los valores proporcionados.

- **Actualizar un registro existente**:
  $producto = Producto::find(1); $producto->precio = 29.99; $producto->save();
  Este código actualiza el registro de la tabla **productos** con el ID igual a 1 cambiando el valor del campo **precio**.

- **Eliminar un registro existente**:
  $producto = Producto::find(1); $producto->delete();
  Este código elimina el registro de la tabla **productos** con el ID igual a 1.

6. Factories.

Las factories en Laravel se utilizan para generar datos ficticios para tus modelos. Estas factories se encuentran en el directorio database/factories y son archivos PHP que contienen la definición de cómo generar datos para un modelo específico.

6.1 Factory Product.

Para crear un Factory para el modelo Product se ejecuta el siguiente comando Artisan: php artisan make:factory ProductFactory --model=Product

Esto generará un archivo de Factory llamado **ProductFactory.php** en la carpeta **database/factories**. Luego se abre el archivo **ProductFactory.php** y define los atributos que deseas generar. Puedes hacerlo usando el método **define** de la clase Factory. Aquí tienes un ejemplo de cómo podría verse:

```
<?php
namespace Database\Factories;
use App\Models\Product;
use Illuminate\Database\Eloquent\Factories\Factory;

class ProductFactory extends Factory {
	protected $model = Product::class;
	public function definition() {
		return [
			'name' => $this->faker->word,
			'description' => $this->faker->sentence,
			'amount' => $this->faker->randomNumber(2),
			'price' => $this->faker->randomFloat(2, 0, 1000),
		];
	}
}
```

En este ejemplo, se están generando datos aleatorios para los campos 'name', 'description', 'amount' y 'price'. Puedes ajustar los valores predeterminados según tus necesidades.

Una vez que hayas definido tu Factory, puedes utilizarlo para generar datos de prueba utilizando el método **create()** en tus pruebas automatizadas o en cualquier otro lugar donde necesites datos de prueba para tu modelo **Product**.

6.2 Factory Client.

Para crear un Factory para el modelo Client se ejecuta php artisan make:factory ClientFactory --model=Client

Esto generará un archivo de Factory llamado **ClientFactory.php** en la carpeta **database/factories**. Luego se abre el archivo **ClientFactory.php** y define los atributos que deseas generar. Puedes hacerlo usando el método **define** de la clase Factory. Aquí tienes un ejemplo de cómo podría verse:

```
<?php
namespace Database\Factories;
use App\Models\Client;
use Illuminate\Database\Eloquent\Factories\Factory;
class ClientFactory extends Factory {
	protected $model = Client::class;
	public function definition() {
		return [
			'name' => $this->faker->name,
			'email' => $this->faker->unique()->safeEmail,
			'address' => $this->faker->address,
			'phone' => $this->faker->phoneNumber,
		];
	}
}
```

En este ejemplo, se están generando datos aleatorios para los campos 'name', 'email', 'address' y 'phone'. Puedes ajustar los valores predeterminados según tus necesidades
.
Una vez que hayas definido tu Factory, puedes utilizarlo para generar datos de prueba utilizando el método **create()** en tus pruebas automatizadas o en cualquier otro lugar donde necesites datos de prueba para tu modelo **Client**.

6.3 Factory Order.

Para crear un Factory para el modelo Order se ejecuta el siguiente comando Artisan:
php artisan make:factory OrderFactory --model=Order

Esto generará un archivo de Factory llamado **OrderFactory.php** en la carpeta **database/factories**.

Abre el archivo **OrderFactory.php** y define los atributos que deseas generar. Puedes hacerlo usando el método **define** de la clase Factory. Aquí tienes un ejemplo de cómo podría verse:

```
<?php
namespace Database\Factories;
use App\Models\Order;
use Illuminate\Database\Eloquent\Factories\Factory;
class OrderFactory extends Factory {
	protected $model = Order::class;
	public function definition() {
		return [
			'product_id' => \App\Models\Product::factory(),
			'client_id' => \App\Models\Client::factory(),
```

```
            'date_order' => $this->faker->dateTimeBetween('-1 year', 'now'),
            'total_amount' => $this->faker->randomFloat(2, 0, 1000),
        ];
    }
}
```

En este ejemplo, se están generando datos aleatorios para los campos 'product_id', 'client_id', 'date_order' y 'total_amount'. Para los campos 'product_id' y 'client_id', se están utilizando factories de los modelos **Product** y **Client** respectivamente para generar ids válidos.

Una vez que hayas definido tu Factory, puedes utilizarlo para generar datos de prueba utilizando el método **create()** en tus pruebas automatizadas o en cualquier otro lugar donde necesites datos de prueba para tu modelo **Order**.

7. Seeders.

Los seeders en Laravel se utilizan para poblar la base de datos con datos de prueba. Estos se encuentran en el directorio **database/seeders** y son archivos PHP que contienen instrucciones para insertar datos en la base de datos.

Para generar datos de prueba en tu base de datos utilizando un seeder en Laravel, se ejecuta el siguiente comando Artisan para crear un seeder: **php artisan make:seeder DatabaseSeeder**

Esto creará un archivo de seeder llamado **DatabaseSeeder.php** en la carpeta **database/seeders**. Luego se abre el archivo **DatabaseSeeder.php** y dentro del método **run**, utiliza los factories que has creado previamente para generar datos de prueba. Aquí tienes un ejemplo de cómo podría verse:

```
<?php
namespace Database\Seeders;
use Illuminate\Database\Seeder;
use App\Models\Product;
use App\Models\Client;
use App\Models\Order;
class DatabaseSeeder extends Seeder {
    public function run() {
        // Generar 10 productos Product::factory(10)->create();
        // Generar 5 clientes Client::factory(5)->create();
        // Generar 20 órdenes de compra Order::factory(20)->create();
    }
}
```

En este ejemplo, estamos utilizando los factories que hemos creado previamente para generar datos de prueba. Estamos generando 10 productos, 5 clientes y 20 órdenes de compra.

Ahora, para ejecutar el seeder y poblar la base de datos con los datos de prueba, ejecuta el siguiente comando: **php artisan db:seed**

Esto ejecutará el seeder **DatabaseSeeder** y generará los datos de prueba en tu base de datos. Asegúrate de que tus modelos y factories estén configurados correctamente antes de ejecutar el seeder.