UA . MASTER MOVILES

MÁSTER UNIVERSITARIO EN DESARROLLO DE SOFTWARE PARA DISPOSITIVOS MÓVILES

PROGRAMACIÓN HIPERMEDIA PARA DISPOSITIVOS MÓVILES

Laravel 3 – Base de datos

CONTENIDO

1. Configuración de la base de datos
2. Migraciones
3. *Schema Builder*
4. *Database Seeding*
5. *Query Builder*
6. *Eloquent*

CONFIGURACIÓN DE LA BASE DE DATOS

- Laravel permite utilizar MySQL, Postgres, SQLite y SQL Server.
- La configuración de la BD a utilizar se establece en el fichero "`config/database.php`".
- Por ejemplo, para "`mysql`":

```
'mysql' => [
    'driver'    => 'mysql',
    'host'      => env('DB_HOST', '127.0.0.1'),  // ¿env?
    'database'  => env('DB_DATABASE', 'forge'),
    'username'  => env('DB_USERNAME', 'forge'),
    'password'  => env('DB_PASSWORD', ''),
    'charset'   => 'utf8mb4',
    'collation' => 'utf8mb4_unicode_ci',
    'prefix'    => '',
    'strict'    => true,
],
```

CONFIGURACIÓN DE LA BASE DE DATOS

- Además, en este mismo fichero se establece la base de datos principal o **por defecto**:

```
'default' => env('DB_CONNECTION', 'mysql'),
```

- Como se puede ver, las opciones de configuración se obtienen usando el método "**env**" de **variables de entorno**.
- Para indicar nuestra configuración de entorno tenemos que modificar el **fichero "`.env`"** (que está en la carpeta raíz del proyecto):

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_DATABASE=nombre-base-de-datos
DB_USERNAME=nombre-de-usuario
DB_PASSWORD=contraseña-de-acceso
```

CONFIGURACIÓN DE LA BASE DE DATOS

- Tenemos que crear la base de datos que vamos a usar manualmente.
- Para esto podemos usar PHPMyAdmin (incluido con XAMPP):

http://localhost/phpmyadmin

- En la pestaña “Bases de datos” usamos la opción para crear una nueva base de datos y le damos el nombre que hayamos indicado en la configuración.
- Para comprobar que todo funciona bien ejecutamos el comando de Artisan:

```
$ php artisan migrate:install
```

- Esto creará la **tabla de migraciones** en la base de datos, lo podemos comprobar actualizando PHPMyAdmin.

MIGRACIONES

MIGRACIONES

- Sistema de control de versiones para bases de datos.
- Guardan un histórico de cambios y estado actual de la BD.
- Se guardan en la carpeta "`database/migrations`" como ficheros PHP.
- Para cada tabla o cambio que queramos hacer en la BD creamos una migración.
  - De esta forma se irá guardando un histórico.
  - Además podremos deshacer los cambios (*rollback*).
- Para crear migraciones, añadirlas a la base de datos o deshacerlas utilizaremos comandos de Artisan.

MIGRACIONES

- Para crear una nueva migración se utiliza la opción de Artisan "`make:migration`", por ejemplo, para una nueva tabla "`users`":

```
php artisan make:migration create_users_table
```

Esto creará el fichero:
"`database/migrations/<TIMESTAMP>_create_users_table.php`"

- Estructura de una migración:

```
class CreateUsersTable extends Migration
{
      // Lanzar la migración
      public function up() {   /* … */   }

      // Deshacer la migración
      public function down() {   /* … */   }
}
```

MIGRACIONES. CONTROL DESDE ARTISAN

- Para lanzar ejecutar las últimas migraciones utilizamos:

```
$ php artisan migrate
```

- Para deshacer la última migración:

```
$ php artisan migrate:rollback
# O para deshacer todas las migraciones:
$ php artisan migrate:reset
```

- O para deshacer todas las migraciones y volver a lanzarlas:

```
$ php artisan migrate:refresh    (o migrate:fresh)
```

- También podemos comprobar el estado actual de las migraciones:

```
$ php artisan migrate:status
```

SCHEMA BUILDER

SCHEMA BUILDER

- *Schema* se utiliza de forma conjunta con las migraciones.
- Permite crear las tablas en el método "`up`" de la migración y eliminarlas en el "`down`", por ejemplo para crear la tabla "`users`":

```
Schema::create('users', function($table) {
  $table->id();
  $table->string('name');
  $table->string('email')->unique();
  $table->timestamp('email_verified_at')->nullable();
  $table->string('password');
  $table->rememberToken();
  $table->timestamps();
});
```

- Y para eliminar la tabla "`users`" en el método "`down`":

```
Schema::drop('users');
```

SCHEMA BUILDER, TIPOS DE CAMPOS

Comando	Tipo de campo
$table->boolean('confirmed');	BOOLEAN
$table->enum('choices', array('foo', 'bar'));	ENUM
$table->float('amount');	FLOAT
$table->increments('id');	Clave principal tipo INTEGER con Auto-Increment
$table->integer('votes');	INTEGER
$table->mediumInteger('numbers');	MEDIUMINT
$table->smallInteger('votes');	SMALLINT
$table->tinyInteger('numbers');	TINYINT
$table->string('email');	VARCHAR
$table->string('name', 100);	VARCHAR con la longitud indicada
$table->text('description');	TEXT
$table->timestamp('added_on');	TIMESTAMP
$table->timestamps();	Añade los *timestamps* "created_at" y "updated_at"
->nullable()	Indicar que la columna permite valores NULL
->default($value)	Declare a default value for a column
->unsigned()	Añade UNSIGNED a las columnas tipo INTEGER

Lista completa en: https://laravel.com/docs/master/migrations#columns

SCHEMA BUILDER, ÍNDICES

- También permite añadir **índices** a los campos de una tabla.
- Podemos crearlos **después** de definir un campo, por ejemplo con:

```
$table->primary('id');      // Añadir una clave primaria
$table->primary(['first', 'last']);       // Primaria compuesta
$table->unique('email');    // Definir el campo como UNIQUE
$table->index('state');     // Añadir un índice a una columna
```

- O añadirlos **a la vez** que se crea el campo, por ejemplo:

```
$table->string('email')->unique();
```

- Importante: al usar "`$table->id()`" ya se crea una clave principal tipo INTEGER auto-incremental.

SCHEMA BUILDER, CLAVES AJENAS

- Para crear una clave ajenas utilizamos "`foreign() -> references() -> on()`", de la forma:

```
$table->unsignedBigInteger('user_id');
$table->foreign('user_id')->references('id')->on('users');
```

- Es importante crear primero el campo de la referencia.
- Opción equivalente:

```
$table->foreignId('user_id')->constrained();
```

Esta instrucción es equivalente a las dos anteriores, ya que además de la clave ajena primero creará la columna "`user_id`".

- Para **eliminar** una clave ajena en el método "`down`" hacemos:

```
$table->dropForeign(['user_id']);
```

SCHEMA BUILDER, CLAVES AJENAS

- Podemos indicar qué hacer en "*on delete*" o en "*on update*":

```
$table->foreignId('user_id')-> constrained()
          ->onDelete('cascade');
```

- También disponemos de los siguientes métodos:

  - `cascadeOnUpdate()`
  - `restrictOnUpdate()`
  - `cascadeOnDelete()`
  - `restrictOnDelete()`
  - `nullOnDelete()`

DATABASE SEEDING

DATABASE SEEDING

- Permite la inserción de **datos iniciales** en la base de datos.
- Muy útil para realizar pruebas en desarrollo o para rellenar tablas que ya tengan que contener datos inicialmente.
- Los ficheros de semillas se encuentra en la carpeta "`database/seeders`".
- El método "`run`" de la clase "`DatabaseSeeder`" es el primero que se llama, y desde el cual podemos:
  - Ejecutar métodos privados de esta clase.
  - Llamar a otros ficheros/clases de semillas separados.

DATABASE SEEDING. CONTROL DESDE ARTISAN

- Para crear un nuevo fichero semilla podemos usar el siguiente comando de Artisan:

```
$ php artisan make:seeder UsersTableSeeder
```

- Una vez definidos los ficheros de semillas, para insertar esos datos en la BD usamos el comando de Artisan:

```
$ php artisan db:seed
```

- En desarrollo es probable que queramos restaurar la base de datos completamente, incluyendo las migraciones y las semillas:

```
$ php artisan migrate:refresh --seed
```

DATABASE SEEDING, EJEMPLO

```php
class DatabaseSeeder extends Seeder
{
    public function run()
    {
        // Llamamos a otro fichero de semillas
        $this->call( UserTableSeeder::class );

        // Mostramos información por consola
        $this->command->info('User table seeded!');
    }
}
```

Desde la clase principal podemos cargar otra clase externa de semillas o llamar a un método privado.

```php
class UserTableSeeder extends Seeder
{
    public function run()
    {
        // Borramos los datos de la tabla
        DB::table('users')->delete();

        // Añadimos una entrada a esta tabla
        User::create(['email' => 'foo@bar.com']);
    }
}
```

Primero eliminamos los datos de la tabla y después añadimos los datos que queramos.

Para usar este código añade: `use Illuminate\Support\Facades\DB;`

QUERY BUILDER

QUERY BUILDER

- Laravel incluye una serie de clases que nos facilita la construcción de consultas y otro tipo de operaciones con la base de datos.
- Al utilizar estas clases obtenemos varias ventajas:
  - Es compatible con todos los tipos de bases de datos soportados por Laravel.
  - Creamos una notación mucho más legible.
  - Nos previene de cometer errores o de ataques por inyección de código SQL.
- Por ejemplo, para realizar una consulta a la tabla "`users`" hacemos:

```
$users = DB::table('users')->get(); // select * from users

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

- **Nota**: para usar la clase DB tenememos que añadir:
  `use Illuminate\Support\Facades\DB;`

QUERY BUILDER

- En la construcción de la *query* podemos usar:
  - `where()`: permite filtar los valores. Si usamos varias clausulas se irán añadiendo con AND.
  - `orWhere()`: igual que "`where`" pero se añadirán con OR.
  - `get()`: para obtener todos los datos.
  - `first()`: para obtener el primero (equivalente a "`limit 1`").
- Ejemplos:

```
$user = DB::table('users')->where('name', 'Pedro')->first();
$users = DB::table('users')->where('votes', '>', 100)->get();
$users = DB::table('users')
            ->where('votes', '>', 100)
            ->orWhere('name', 'Pedro')
            ->get();
```

QUERY BUILDER

- También podemos utilizar los métodos `orderBy`, `groupBy` y `having` en las consultas:

```
$users = DB::table('users')
            ->orderBy('name', 'desc')
            ->groupBy('count')
            ->having('count', '>', 100)
            ->get();
```

- Para realizar el paginado utilizamos los métodos `skip` (para el *offset*) y `take` (para *limit*), por ejemplo:

```
$users = DB::table('users')->skip(10)->take(5)->get();
```

- Para más información (*join, insert, update, delete*, agregados, etc.)
https://laravel.com/docs/master/queries

QUERY BUILDER, TRANSACCIONES

- Para crear transacciones sobre un conjunto de operaciones tenemos que hacer:

```
DB::transaction( function()
{
    DB::table('users')->update(array('votes' => 1));

    DB::table('posts')->delete();
});
```

- Si se produce excepción en las operaciones que se realizan en la transacción se desharían todos los cambios aplicados hasta ese momento de forma automática.

ELOQUENT

ELOQUENT. MODELOS DE DATOS MEDIANTE ORM

- **ORM** (*Object-Relational mapping* o mapeo objeto-relacional) es una técnica de programación para convertir datos entre un lenguaje orientado a objetos y una BD relacional como motor de persistencia.
- Laravel incluye su propio sistema de ORM llamado ***Eloquent.***
- *Eloquent* proporciona una manera elegante y fácil de interactuar con la BD a través de PHP.
- Cada tabla en la BD debe tener su correspondiente modelo en la carpeta "`app/Models`".
- Para crear un nuevo modelo de datos podemos usar el comando de Artisan:

```
$ php artisan make:model Movie
```

- Este comando creará el fichero "`Movie.php`" en la carpeta "`app/Models`".

ELOQUENT

- Al crear un nuevo modelo con Artisan se incluirá el contenido básico del mismo.
- Por ejemplo, la clase o modelo "`Movie.php`" sería:

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

class Movie extends Model    // Heredamos de Model!
{
}
```

- Solamente con este código y sin escribir nada más podemos utilizarlo para realizar todo tipo de *queries* sobre la tabla "`movies`".

ELOQUENT

- Eloquent automáticamente enlaza el modelo con la tabla a partir del nombre de la clase, transformándolo al **plural en minúsculas**:
  - "User" ----> "users" `// Usará el plural en inglés`
  - "Movie" ----> "Movies"
  - Para cambiar el nombre usamos la propiedad "`$table`" del modelo.
- Eloquent asume que la tabla tendrá una **clave primaria** llamada "`id`".
  - Para cambiar el nombre usamos la propiedad "`$primaryKey`".
- Además actualiza automáticamente los *timestamps* de la tabla (`updated_at` y `created_at`).
  - Para desactivarlo usamos la propiedad "`$timestamps`".

```
class User extends Model
{
    protected $table = 'my_users';
    protected $primaryKey = 'my_id'
    public $timestamps = false;
}
```

ELOQUENT, CONSULTAR DATOS

- Desde un controlador, para obtener todas las filas de la tabla asociada al modelo "`User`" utilizaremos:

```
use App\Models\User;  // Indicamos su espacio de nombres!

$users = User::all(); // select * from users

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

- Este método nos devolverá un array de resultados, donde cada item del array es una instancia del modelo `` `User` ``.

- Esto nos permite acceder a los valores de cada elemento como si fuera un objeto: "`$user->name`"

ELOQUENT, CONSULTAR DATOS

- Todos los métodos de "*Query Builder*" se pueden utilizar con Eloquent. Además Eloquent incorpora algunos más.
- Buscar un elemento por su identificador (que por defecto será "`id`"):

```
$user = User::find(1);
var_dump($user->name);
```

- Si queremos que se lance una excepción cuando no se encuentre un modelo:

```
$model = User::findOrFail(1);
$model = User::where('votes', '>', 100)->firstOrFail();
```

- Esto nos permite capturar las excepciones y mostrar un error 404 cuando sucedan.

ELOQUENT, EJEMPLOS

Algunos otros ejemplos de uso:

```
$users = User::where('votes', '>', 100)->take(10)->get();
foreach ($users as $user)
{
    var_dump($user->name);
}

// O para obtener el primer usuario de la lista
$user = User::where('votes', '>', 100)->first();

$count = User::where('votes', '>', 100)->count();
$price = Orders::max('price');
$price = Orders::min('price');
$price = Orders::avg('price');
$total = User::sum('votes');
```

ELOQUENT, INSERTAR DATOS

- Para **añadir** una entrada en la tabla de la base de datos asociada con un modelo tenemos que hacer:

```
$user = new User;
$user->name = 'Juan';
$user->save();
```

- Para obtener el identificador asignado en la base de datos después de guardar:

```
$insertedId = $user->id;
```

ELOQUENT, ACTUALIZAR DATOS

- Para **actualizar** un registro de un modelo solo tendremos que recuperar en primer lugar la instancia, modificarla y por último guardar los datos:

```
$user = User::find(1);
$user->email = 'juan@gmail.com';
$user->save();
```

- Recuerda que también puedes usar "**`findOrFail`**".

ELOQUENT, BORRAR DATOS

- Para borrar una instancia de un modelo en la base de datos usamos el método `delete()`:

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

- También podemos borrar un conjunto de resultados:

```
$affectedRows = User::where('votes','>',100)->delete();
```

¿PREGUNTAS?