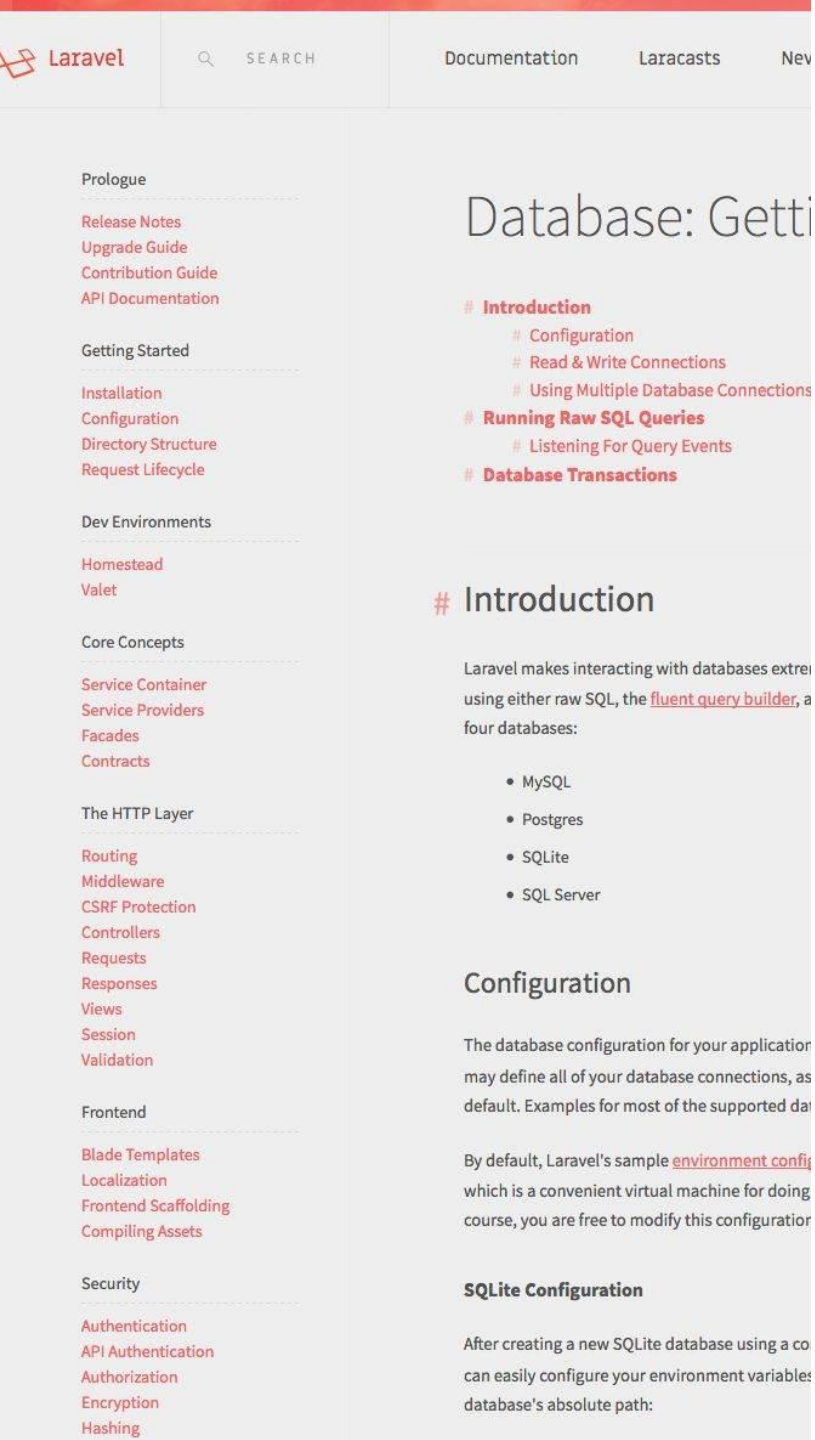


LARAVEL: ACCESO A DATOS

DISEÑO DE SISTEMAS SOFTWARE

Contenido

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

Configuración de un proyecto

- Laravel obtiene la mayor parte de su configuración de variables de entorno
- La carpeta `config/` contiene los scripts que cargan la configuración en memoria a partir de las variables de entorno cuando se inicia la aplicación, proporcionando un valor por defecto cuando no se encuentran:

```
'debug' => env('APP_DEBUG', false)
```

- Para facilitar la configuración y unificar entornos de desarrollo se pueden definir las variables de entorno en el archivo `.env` en la carpeta raíz:

```
APP_DEBUG=true
```

Configuración de la base de datos

- Laravel permite utilizar MySQL, Postgres, SQLite y SQL Server
- Para configurar la BBDD a utilizar tenemos que modificar el fichero de configuración `config/database.php`
- Por ejemplo, para configurar `sqlite`:

```
'sqlite' => [
    'driver' => 'sqlite',
    'database'=> env('DB_DATABASE',
                     database_path('database.sqlite')),
    'prefix' => '',
],
```

Configuración de la base de datos

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

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

- Para indicar que queremos usar **sqlite** tenemos que modificar el fichero "`.env`" en la carpeta raíz del proyecto, especificando el valor de `DB_CONNECTION`, el resto de opciones que empiezan por `DB_` se deben comentar:

```
DB_CONNECTION=sqlite
#DB_HOST=no usado por sqlite
#DB_PORT=no usado por sqlite
#DB_DATABASE=usa la configuración por defecto
#DB_USERNAME=no usado por sqlite
#DB_PASSWORD=no usado por sqlite
```

Creación de la base de datos

- Para crear el archivo de base de datos ejecutamos el siguiente comando en la carpeta del proyecto:

```
$ touch database/database.sqlite
```

- 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 usando `sqlitebrowser`:

```
$ sqlitebrowser database/database.sqlite
```

Creación de la base de datos

- El archivo `database.sqlite` **no debe subirse** al repositorio Git
- Basta con añadirlo al archivo `.gitignore` que ya tiene el proyecto

Laravel: Acceso a datos

MIGRACIONES

Migraciones

- Sistema de control de versiones para la estructura de la BBDD
- Guardan un histórico de cambios y estado actual de la estructura
- Son archivos PHP guardados en la carpeta `database/migrations`
- Para cada tabla o cambio que queramos hacer en la BBDD creamos una migración, de esta forma se irá guardando un histórico
- Además podremos deshacer los cambios (rollback)

https://laravel.com/docs/5.5/migrations

Crear migraciones

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

```
$ php artisan make:migration create_users_table --create=users
```

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

- Para crear una migración que modifica una tabla ya existente se usa la opción `--table=nombre_tabla`

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

Crear migraciones

- Los comandos anteriores crean un archivo de migración con una estructura preparada para trabajar con la tabla indicada

```php
class CreateUsersTable extends Migration
{
    // Lanza la migración
    public function up() {
        Schema::create('users', function (Blueprint $table) {
            // Define la estructura de la tabla con Schema Builder
        });
    }
    // Deshace la migración
    public function down() {
        Schema::dropIfExists('users');  // Elimina la tabla
    }
}
```

Lanzar y deshacer migraciones

- 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
```

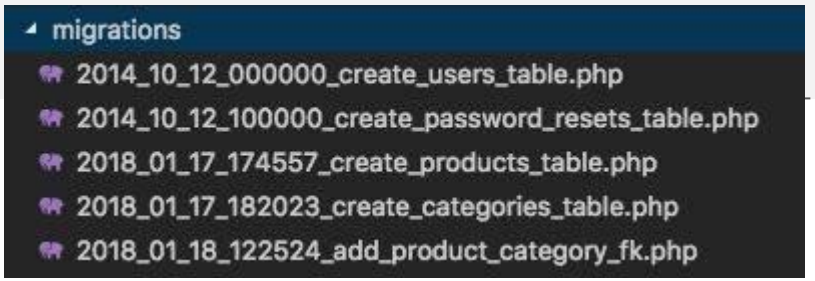
- También podemos comprobar el estado actual de las migraciones:

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

Orden de las migraciones

- Es importante no modificar migraciones anteriores para modificar la estructura de una tabla, de lo contrario puede haber errores al hacer un `rollback` de la base de datos
- Si se quiere añadir una clave ajena a otra tabla que se ha creado posteriormente, se debe crear una nueva migración que modifique la tabla para añadir la clave ajena

```php
public function up() {
  Schema::table('products', function (Blueprint $table) {
    $table->integer('category_id')->unsigned();
    $table->foreign('category_id')->references('id')->on('categories');
  });
}
```



Eliminar migraciones

- Si necesitas eliminar un archivo de migración creado por error hay que seguir estos pasos
  - Eliminar el archivo en la carpeta `database/migrations`
  - Ejecutar el comando

```
$ composer dump-autoload
```

Laravel: Acceso a datos

SCHEMA BUILDER

Crear tablas

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

```
public function up()
{
    Schema::create('users', function (Blueprint $table) {
        $table->increments('id');
        $table->string('name');
        $table->string('email')->unique();
        $table->string('password');
        $table->rememberToken();
        $table->timestamps();
    });
}
```

Eliminar tablas

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

```
    // Deshace la migración
    public function down() {
        Schema::dropIfExists('users');  // Elimina la tabla
    }
```

Modificar tablas

- Con las migraciones también se puede modificar la estructura de una tabla existente:

```
    public function up()
    {
            Schema::table('users', function (Blueprint $table) {
                $table->string('address');
            });
    }

    public function down()
    {
            Schema::table('users', function (Blueprint $table) {
                $table->dropColumn('address');
            });
    }
```

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

https://laravel.com/docs/5.5/migrations#columns

Tipos de datos

- Schema Builder se encarga de adaptar los tipos de datos al motor de BBDD que se esté utilizando

- Tipos de datos que utiliza SQLite3: https://www.sqlite.org/datatype3.html

- Aunque estemos usando SQLite3, el diseño de la BBDD se debe hacer sin tener en cuenta sus limitaciones, Schema Builder y Query Builder se encargarán de las transformaciones necesarias

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->increments('id')` ya se crea una **clave principal** tipo INTEGER auto-incremental.

Claves ajenas

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

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

- Es importante crear primero el campo de la referencia
- Podemos indicar que hacer en el `onDelete` on en `onUpdate`:

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

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

```
$table->dropForeign('posts_user_id_foreign');
// Siguiendo el patrón de nombre: <tabla>_<columna>_foreign
```

Claves ajenas

- El driver SQLite para PHP está configurado por defecto para no comprobar la integridad referencial de las claves ajenas
- Para activarla debes añadir el siguiente código en el método `boot()` del archivo `app/Providers/AppServiceProvider.php`

```php
public function boot(){
    if (config('database.default') == 'sqlite') {
        $db = app()->make('db');
        $db->connection()->getPdo()->exec("PRAGMA foreign_keys = ON");
    }
}
```

Laravel: Acceso a datos

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/seeds`
- 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

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
        DB::table('users')->insert([
            'name' => 'Username',
            'email' => 'name@domain.com',
            'password' => 'strongpassword' ]);
    }
}
```

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

Laravel: Acceso a datos

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

Query Builder

- Por ejemplo, para realizar una consulta a la tabla `users` hacemos:

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

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

Query Builder

- Se puede recuperar un conjunto de datos con `get()` o un único valor con `first()`

```php
// Recupera todos los objetos de la tabla
$users = DB::table('users')->get();
// $users es de tipo Illuminate\Support\Collection
foreach ($users as $user) {
    // Los objetos son de tipo StdClass
    // y se puede acceder a sus propiedades
    echo $user->name;
}

// Recupera un único objeto
$user = DB::table('users')->where('name', 'John')->first();
echo $user->name;
```

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 más información (join, insert, update, delete, etc.) https://laravel.com/docs/5.5/queries

Query Builder

- Al recuperar datos de una base de datos SQLite3 con Query Builder, las propiedades de los objetos que devuelve son siempre de tipo `string`
- PHP permite realizar operaciones con strings sin necesidad de convertirlos explícitamente

```
echo $user->age // “57”
if ($u->age < 67) {  // Evalúa a verdadero
  // …
}
```

- Más adelante veremos cómo realizar transformaciones y validaciones de tipos de datos

Laravel: Acceso a datos

EJERCICIO

Ejercicio

1. Configura tu proyecto para trabajar con una base de datos SQLite almacenada en el archivo `database/database.sqlite`
2. Crea una migración para crear una tabla `categories` con los siguientes campos

Nombre	**Tipo de datos**
id	Clave principal, autoincremental
name	VARCHAR

3. Crea un *seeder* para insertar las categorías “Procesadores” y “Discos duros”. Modifica `database/seeds/DatabaseSeeder.php` para que ejecute el nuevo seeder

Ejercicio

5. Ejecuta la migración y el seeder
6. Abre el archivo `database/database.sqlite` con `sqlitebrowser` para comprobar que los datos se han introducido
7. Crea una migración para crear una tabla `products` con los siguientes campos

Nombre	**Tipo de datos**
id	Clave principal, autoincremental
name	VARCHAR
price	Float
category_id	Entero, clave ajena a categorías

8. Ejecuta la nueva migración

Ejercicio

7. Crea un nuevo seeder que introduzca varios productos para cada categoría, necesitarás recuperar el identificador de cada categoría antes de realizar las inserciones con `insert()` para pasar el valor correcto de la clave ajena
8. Vuelve a poblar la base de datos
9. Usando `tinker`, prueba las siguientes consultas
   a. Recupera todas las categorías
   b. Recupera todos los productos de la categoría “Procesadores”
   c. Recupera el precio del producto más caro de la categoría “Procesadores”
   d. Elimina todos los productos cuyo nombre empiece por una letra a tu elección