Guía de Laravel

Estructura de directorios

- **app**: código principal
- **config**: archivos de configuración: bbdd, caché, correos, sesiones...
- **database**: definición de la capa DAO
- **public**: única carpeta accesible desde el exterior. Contiene index, css, js...
- **resources**: contenido estático de la aplicación. Se puede considerar como el "taller". Aquí tendremos vistas, CSS y JS. Estos recursos se compilan y pasan al directorio public
- **bootstrap**: procesa las llamadas a nuestro proyecto. ¡No tocar!
- **routes**: rutas de la aplicación. Es el Front Controller
- **storage**: información interna para la ejecución de la web: archivos de sesión, caché, logs...
- **tests**: ficheros para pruebas automatizadas
- **vendor**: librerías y dependencias que conforman Laravel

Ficheros

Es **muy importante** que ninguno de estos ficheros se suban a un repositorio de GIT

- **.env**: valores de configuración
- **composer.json**: dependencias PHP
- **package.json**: dependencias para la parte del cliente

APP

Nos interesa sobre todo:

- **Http -> Controllers**: reciben peticiones desde el Front Controller, interactúan con Model para pedir o guardar datos y envían una respuesta (View)
- **Models**: representan tus datos. Cada modelo se suele corresponder directamente con una tabla de BBDD. Usan Eloquent. Aquí se definen las relaciones: un Usuario tiene muchos Posts

Database

Nos interesa todo

- **migrations**: scripts de PHP que definen la estructura de la BBDD
- **seeders**: permiten poblar la BBDD con datos iniciales para probar la aplicación
- **factories**: plantillas para generar datos falsos de forma masiva. Muy usado junto con los seeders

Resources

Nos interesa sobre todo:

- **views**: plantillas que definen la vista que verá el usuario final. Se pueden usar como PHP básico o como plantilla **Blade**, **muy recomendable**, esto ayuda a simplificar muchísimo la integración de PHP con HTML.
    - **components**: por defecto no existe, pero lo crearemos para reutilizar componentes entre diferentes vistas

Routes

Nos interesa sobre todo:

- **web.php**: archivo principal para las rutas de la aplicación. Usan sesiones, cookies y protección CSRF por defecto. Aquí se mapean las peticiones web

Laravel paso a paso

Creación de un proyecto

```
laravel new quacker
...
Starter kit: None
Testing framework: Pest
Database: SQLite
Run npm install: Yes
...
cd nombre_proecto
composer run dev
# Solo si lo anterior falla:
php artisan serve
```

Creación de modelo, migraciones y factorías

```
php artisan make:model OfertaTrabajo -mf
```

Parámetros:

- -m: creará el fichero de migración: `database/migrations`
- -f: creará el fichero de factoría: `database/factories`

Modificando la migración

Dentro de `database/migrations` encontraremos un fichero `YYYY_MM_DD_HHmmSS_create_oferta_trabajos_table`

La función `up()` se ejecutará cuando hagamos una migración y creará la tabla en la BBDD

Por defecto tenemos lo siguiente:

```
public function up(): void
{
    Schema::create('oferta_trabajos', function (Blueprint $table) {
        $table->id();
        $table->timestamps();
    });
}
```

Aquí podemos hacer varios cambios pero, de momento, simplemente vamos a añadir tres columnas: título, descripción y empresa

```
public function up(): void
{
    Schema::create('oferta_trabajos', function (Blueprint $table) {
        $table->id();
        $table->string('titulo');
        $table->text('descripcion');
        $table->string('empresa');
        $table->timestamps();
    });
}
```

Importante:

- `id()`: representa la PK. Se creará en BBDD con el nombre `id` y la restricción `PK NOT NULL`
- `timestamps()`: representa 2 columnas: `created_at` y `updated_at`. Se crearán en BBDD con esos nombres. Se añaden por defecto, pero pueden desactivarse. Lo haremos más adelante
- `string('titulo')`: creará una columna en BBDD con el nombre `titulo` de tipo VARCHAR
- `text('descripcion')`: creará una columna en BBDD con el nombre `descripcion` de tipo TEXT

Para crear las tablas, haremos uso de **artisan**:

```
php artisan migrate
```

Podremos comprobar si se ha creado correctamente desde nuestro SGDB

Otros comandos útiles de **artisan migrate**:

- `php artisan migrate:rollback`: deshace la última migración ejecutada. Útil si haces un cambio y no quieres revertirlo todo
- `php artisan migrate:refresh`: crea la BBDD desde 0: borra todas las tablas y las vuelve a crear con sus respectivos ficheros de migración
    - `php artisan migrate:refresh --seed`: lo mismo pero, tras ello, ejecuta una "semilla" de población de BBDD (necesario explicar primero las factorías)

Poblando la BBDD

Dentro de `database/factories` encontraremos, entre otros ficheros, `OfertaTrabajoFactory.php`. Este nos permite definir cómo crear datos ficticios de **OfertaTrabajo**. Esto es muy útil para probar nuestra aplicación sin tener que estar continuamente creando datos manualmente. Para ello, hace uso de una librería especializada en la creación de datos ficticios: **Faker**

De primeras nos encontramos esto:

```
public function definition(): array
{
    return [
        //
    ];
}
```

Si nos fijamos en una factoría que ya esté hecha (por ejemplo, `UserFactory`), podemos hacernos una idea de cómo crear la nuestra:

```
public function definition(): array
{
    return [
        'name' => fake()->name(),
        'email' => fake()->unique()->safeEmail(),
        'email_verified_at' => now(),
        'password' => static::$password ??= Hash::make('password'),
        'remember_token' => Str::random(10),
    ];
}
```

Las claves definidas en ese array coinciden con los nombres de los campos en BBDD de la tabla `users`, los cuales vienen especificados en su propio fichero de migración. Por tanto, usaremos los campos definidos en nuestro fichero de migración para rellenar este array:

```
public function definition(): array
{
    return [
        'titulo' => fake()->jobTitle(),
        'descripcion' => fake()->paragraph(),
        'empresa' => fake()->company(),
    ];
}
```

Esos métodos: `jobTitle()`, `paragraph()` y `company()`, son propios de la librería **Faker**. No pretendo que los sepamos todos, ni tiene sentido hacerlo, simplemente exploraremos la lista cuando necesitemos algo concreto o, si no encontramos nada que nos convenza, haremos uso de algún método genérico para crear texto aleatorio (por ejemplo, `paragraph()`, o `text()`) o números aleatorios (`numberBetween`).

Podemos probar nuestra factoría de datos aleatorios con otra herramienta de **artisan**: **artisan tinker**. Es el CLI propio de Laravel, desde el cual cargamos todo el código de la aplicación para probarlo manualmente

```
php artisan tinker
> App\Models\OfertaTrabajo::factory(5)->create()
```

Esto creará 5 entradas en la BBDD.

Si nos convence lo que hemos hecho, podemos configurar el número de `OfertaTrabajo` que se crearán aleatoriamente cada vez que hagamos un `php artisan migrate:refresh --seed` desde `database/seeders/DatabaseSeeder.php`. Este cuenta con un método `run()` que contiene lo siguiente:

```
public function run(): void
{
    // User::factory(10)->create();

    User::factory()->create([
        'name' => 'Test User',
        'email' => 'test@example.com',
    ]);
}
```

Hay una línea comentada que ya nos da una pista de cómo podemos usarlo. Modificamos el método para que cree 100 ofertas de trabajo:

```
public function run(): void
{
    OfertaTrabajo::factory(100)->create();

    User::factory()->create([
        'name' => 'Test User',
        'email' => 'test@example.com',
    ]);
}
```

Ahora podemos ejecutar nuevamente una migración desde 0 con `php artisan migrate:refresh --seed` y debería crearse nuestro esquema de BBDD y poblarse con datos. Lo siguiente que veremos son las rutas para recuperar esos datos y las vistas para mostrarlos

Definiendo las rutas

Vale, ya hemos poblado nuestra aplicación con datos, pero ahora tenemos que darles uso. Para ello vamos a `routes/web.php`. Esto es nuestro **Front Controller**. Es decir: el primer punto de interacción del usuario con nuestra aplicación

De primeras nos encontraremos solamente esto:

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

Su comportamiento es el siguiente:

1. Cuando el usuario accede a nuestro sitio web con una **petición GET** sobre `http://url/`, se "llamará" a una función que:
2. Devuelve una vista llamada **welcome**. Esta vista podemos encontrarla en `resources/views/welcome.blade.php`. Apreciarás que tiene una extensión `.blade.php`. Esto es bueno, ya que nos permite simplificar la integración de PHP con HTML, ya lo veremos

Aquí podemos definir varios tipos de acciones, como son las siguientes:

- Devolver una vista: `return view('vista.blade.php')`
- Devolver una vista pasándole datos (indispensable en páginas dinámicas): `return view('vista.blade.php', ['datos' => 'Lo que sea'])`
- Redirigir: `return redirect('/')`
- Devolver un JSON: `return response()->json(['datos' => 'Lo que sea'])`
- Devolver texto plano: `return 'Hola Mundo'`
- Devolver un objeto: `return $ofertaTrabajo` (Laravel lo convierte automáticamente a JSON)
- Códigos HTTP: `return response()->json($ofertaTrabajo, 201)`
- Respuestas vacías: `return response()->noContent()`

Estas acciones sucederán cuando se "llame" a las rutas que definamos nosotros, y estas rutas están definidas por:

- El método HTTP: `GET`, `POST`, `DELETE`, `PUT/PATCH`
- La URL: `/ofertas`, `/ofertas/...`

Como primera ruta podríamos hacer simplemente lo siguiente:

```
Route::get('/ofertas', function() {
    return OfertaTrabajo::all();
});
```

Cuando entremos en `localhost:8000/ofertas`, veremos un fichero JSON con los datos de todas las ofertas de trabajo. Vamos ahora a definir una vista para mostrar esto más bonito

Vista index

En Laravel, los modelos (como `OfertaTrabajo`) son la capa que interactúa con la base de datos para manejar el **ciclo de vida de los recursos**. Este ciclo de vida se resume en las 4 acciones principales conocidas como **CRUD**:

- Crear (**C**reate): añadir nuevos registros
- Leer (**R**ead): obtener registros existentes (individuales o listas)
- Actualizar (**U**pdate): modificar registros existentes
- Borrar (**D**elete): eliminar registros

La **vista index** está directamente asociada con la acción **leer** de un conjunto de recursos. Su objetivo es mostrar la lista completa de todas las ofertas de trabajo disponibles. Para lograr esto necesitamos:

1. **Cargar los datos** de todas las ofertas de trabajo utilizando métodos que facilita el modelo: `OfertaTrabajo::all()`
2. **Devolver una vista** (`index`), pasándole los datos cargados (¡MVC, bitch!)

A continuación haremos 2 cosas:

- Crearemos la vista:
    1. Dentro de `resources/views` crearemos un directorio llamado `ofertas`
    2. Dentro de `resources/views/ofertas` crearemos una vista llamada `index.blade.php`
- Modificaremos la ruta que hemos definido antes en `routes/web.php` para que devuelva la vista

```
Route::get('/ofertas', function() {
    $ofertas = OfertaTrabajo::all();

    return view('ofertas.index', [
        'ofertas' => $ofertas
    ]);
});
```

Ahora tendremos que definir la vista para que muestre unos datos recibidos por parámetro. Yo voy a crear una tabla dinámica:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Ofertas de Trabajo</title>
</head>
<body>
```

```
    <table>
        <tr>
            <th>Título</th>
            <th>Empresa</th>
            <th>Descripción</th>
        </tr>
        @foreach($ofertas as $oferta)
        <tr>
            <td>{{ $oferta->titulo }}</td>
            <td>{{ $oferta->empresa }}</td>
            <td>{{ $oferta->descripcion }}</td>
        </tr>
        @endforeach
    </table>
</body>
</html>
```

¿Ves el `@foreach` y el `@endforeach`? ¿Ves `{{ $oferta->titulo }}` y demás? Esto es la magia del motor de plantillas **Blade**. Esto es lo mínimo que puede hacer, ya iremos viendo más magia.

Nomenclatura Estándar (Convención RESTful)

Para ahorrarnos escribir código, es crucial entender la **convención de nombrado** que Laravel promueve, conocida como **RESTful**. Esta convención asocia verbos HTTP con acciones estandarizadas sobre un recurso (en nuestro caso, `OfertaTrabajo`).

Adoptar esta nomenclatura es clave, ya que no solo hace que el código sea más legible, sino que nos prepara para usar **Controladores de Recursos** más adelante. Además, nos permite **nombrar** las rutas para referenciarlas fácilmente sin escribir la URL completa. La convención estándar de acciones para un recurso es la siguiente:

Petición HTTP	URI	Acción (nombre de la ruta)	Propósito
`GET`	/ofertas	index	**Leer**: muestra una lista con todas las ofertas
`GET`	/ofertas/create	create	Muestra un formulario para crear una nueva oferta
`POST`	/ofertas	store	**Crear**: guarda una nueva oferta enviada por un formulario
`GET`	/ofertas/{id}	show	**Leer**: muestra el detalle de una oferta específica
`GET`	/ofertas/{id}/edit	edit	Muestra el formulario para editar una oferta específica

Petición HTTP	URI	Acción (nombre de la ruta)	Propósito
`PUT/PATCH`	/ofertas/{id}	update	**Actualizar**: modifica una oferta específica
`DELETE`	/ofertas/{id}	destroy	**Borrar**: elimina una oferta específica

¿En qué se traduce ajustarse a la nomenclatura estándar?

- En crear vistas con nombres como `index.blade.php`, `show.blade.php`, `create.blade.php`, `edit.blade.php`
- En crear rutas con las peticiones y URIs especificadas en la tabla (¿o no? ya veremos algo bastante guay con respecto a esto)