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 Oferta -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_ofertas_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('ofertas', 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

```php
public function up(): void
{
    Schema::create('ofertas', 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, `OfertaFactory.php`. Este nos permite definir cómo crear datos ficticios de **Oferta**. 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\Oferta::factory(5)->create()
```

Esto creará 5 entradas en la BBDD.

Si nos convence lo que hemos hecho, podemos configurar el número de `Oferta` 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
{
    Oferta::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 $Oferta` (Laravel lo convierte automáticamente a JSON)
- Códigos HTTP: `return response()->json($Oferta, 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 Oferta::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

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, `Oferta`).

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
`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)

Vista `index`

En Laravel, los modelos (como `Oferta`) 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: `Oferta::all()`
2. **Devolver una vista** (`index`), pasándole los datos cargados (¡MVC, bitch!)

A continuación haremos 2 cosas:

- Crearemos la vista:
    a. Dentro de `resources/views` crearemos un directorio llamado `ofertas`
    b. 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

```php
Route::get('/ofertas', function() {
    $ofertas = Oferta::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:

```php
<!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.

Vista `show`

El objetivo de la vista **show** es el de mostrar información sobre un único elemento. Para ello tendremos que:

1. **Cargar los datos** de un elemento usando métodos que facilita el propio modelo: `Oferta::find($id)`
2. **Devolver una vista** (`show`), pasándole los datos cargados

Siguiendo lo indicado en la tabla anterior, definiremos una ruta GET con URI `/ofertas/{id}` en `routes/web.php`:

```
Route::get('/ofertas/{id}', function($id){
    return Oferta::find($id);
});
```

Al igual que con el ejemplo anterior, apreciaremos que el servidor devuelve un fichero JSON, pero esos datos se los podríamos pasar a una vista:

- Dentro de `resources/views/ofertas` crearemos una vista llamada `show.blade.php`
- Dentro de `routes/web.php`, modifico la ruta definida previamente:

  ```
  Route::get('/ofertas/{id}', function($id){
      return view('ofertas.show', [
          'oferta' => Oferta::find($id)
      ]);
  });
  ```

La vista podríamos definirla tal que así:

```
<!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>
        <tr>
            <td>{{ $oferta->titulo }}</td>
            <td>{{ $oferta->empresa }}</td>
            <td>{{ $oferta->descripcion }}</td>
        </tr>
    </table>
</body>
</html>
```

Podríamos modificar la vista `index` para que añada un enlace a la vista `show` en cada oferta de trabajo:

```
<!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>
            <th>Detalles</th>
        </tr>
        @foreach($ofertas as $oferta)
        <tr>
            <td>{{ $oferta->titulo }}</td>
            <td>{{ $oferta->empresa }}</td>
            <td>{{ $oferta->descripcion }}</td>
            <td><a href="/ofertas/{{ $oferta->id }}">Ver
detalles</a></td>
        </tr>
        @endforeach
    </table>
</body>
</html>
```

Vista `create`

El objetivo de la vista **create** es el de ofrecer una interfaz que permita asignar valores a un nuevo elemento. Deberemos diseñar un formulario a través del cual rellenemos los campos del elemento que cpreasignonsideremos "asignables". Hay algunos campos **autoasignables** y otros **preasignados**. Ejemplos de campos **preasignados** pueden ser:

- Rol de un usuario que se registra
- Estado inicial de un pedido tras realizar una compra

Por otro lado, los campos **autoasignados** típicos son:

- Identificador único: indispensable para cumplir con la restricción UNIQUE NOT NULL de la clave primaria
- Fecha y hora de creación

Podemos consultar el fichero de migraciones cuáles son los campos de nuestro modelo si no los recordamos. Apreciaremos que claramente hay 2 campos **autoasignados**, que son `id` y `timestamps` (que ya dijimos que se traduce en realidad en 2 campos: `created_at` y `updated_at`). Los otros 3 campos (`titulo`, `descripcion` y `empresa`) son perfectos candidatos para formar parte del formulario. Idealmente, deberemos elegir elementos gráficos adecuados al tipo de dato a rellenar. Por ejemplo:

- `titulo` y `empresa`: varchar, podemos elegir `input:text`
- `descripcion`: text, podemos elegir `textarea`

Si tuviéramos algún campo de fecha (por ejemplo: fecha de fundación de la empresa), podríamos utilizar `input:date`.

Dentro de `resources/views/ofertas` crearemos una vista llamada `create.blade.php` que, en nuestro caso, podría ser:

```
<!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>Creación de Oferta de Trabajo</title>
</head>
<body>
    <form action="" method="post">
        <label for="titulo">Título del Trabajo:</label>
        <input type="text" id="titulo" name="titulo" required><br>
<br>

        <label for="empresa">Empresa:</label>
        <input type="text" id="empresa" name="empresa" required>
<br><br>

        <label for="descripcion">Descripción:</label><br>
        <textarea id="descripcion" name="descripcion" rows="4"
cols="50" required></textarea><br><br>

        <input type="submit" value="Crear Oferta de Trabajo">
    </form>
</body>
</html>
```

De momento no vamos a rellenar el atributo `action`, ya que estamos definiendo solamente las vistas.

Deberemos devolver esta vista cuando accedan a la ruta indicada en la tabla anterior (método `GET` con URI `/ofertas/create`). Esto lo definimos en `routes/web.php`

```
Route::get('/ofertas/create', function(){
    return view('ofertas.create');
});
```

Vista `edit`

La vista `edit` es la que nos permite modificar un elemento ya creado. Podemos aprovechar la **vista create**, con las siguientes diferencias:

- El nombre de la vista
- El valor del atributo `action` del formulario (ya que será otro método diferente al de creación el que se encargue de realizar la actualización)
- Los campos del formulario deberán estar **pre-rellenados** con los valores del elemento ya creado
- El método HTTP: no será POST, sino PUT/PATCH

Por tanto, crearemos una nueva vista dentro de `resources/views/ofertas` con nombre `edit.blade.php` que, en nuestro caso, podría ser:

```
<!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>Creación de Oferta de Trabajo</title>
</head>
<body>
    <form action="" method="post">
        @method('PATCH')
        <label for="titulo">Título del Trabajo:</label>
        <input type="text" id="titulo" name="titulo" value="{{
$oferta->titulo }}" required><br><br>
```

```
        <label for="empresa">Empresa:</label>
        <input type="text" id="empresa" name="empresa" value="{{
$oferta->empresa }}" required><br><br>

        <label for="descripcion">Descripción:</label><br>
        <textarea id="descripcion" name="descripcion" rows="4"
cols="50" required>{{ $oferta->descripcion }}</textarea><br><br>

        <input type="submit" value="Crear Oferta de Trabajo">
    </form>
</body>
</html>
```

Según el estándar HTML, sólo existen 3 valores posibles para el atributo `method`: `GET`, `POST` y `DIALOG`. Sin embargo, necesitamos mandar un `PATCH` al servidor. Esto se puede hacer con Laravel mediante `@method('PATCH')`. También utilizaremos esto con el `DELETE`

Deberemos devolver esta vista cuando accedan a la ruta indicada en la tabla anterior (método `GET` con URI `/ofertas/{id}/edit`). Esto lo definimos en `routes/web.php`

```
Route::get('/ofertas/{id}/create', function($id){
    return view('ofertas.edit', [
        'oferta' => Oferta::find($id)
    ]);
});
```

Puesto que estamos modificando una oferta ya existente, debemos buscarla primero y pasársela a la vista

Acción `store`

Esta acción no tiene una vista asociada, pero conecta conceptualmente con la vista `create`. ¿Recuerdas que dijimos que no íbamos a rellenar el atributo `action` del formulario. Ahora es el momento. Primero definiremos en `routes/web.php` una nueva ruta, recordemos la tabla de nomenclatura estándar, para store se espera un `POST` a `/ofertas`:

```
Route::post('/ofertas', function(){

});
```

Aquí procesaremos los datos enviados por el formulario de la vista `create`. A continuación, modificamos el atributo `action` de esa vista:

```
...
</head>
<body>
    <form action="/ofertas" method="post">
        <label for="titulo">Título del Trabajo:</label>
...
```

Si probamos nuestro formulario de creación, nos dará **Error 419: Page Expired**. Esto es debido a que Laravel, por seguridad, implementa por defecto la protección **CSRF**. Para enviar el token anti-CSRF, tendremos que añadir la sentencia `@csrf` dentro del formulario

```
...
</head>
<body>
    <form action="/ofertas" method="post">
        @csrf
        <label for="titulo">Título del Trabajo:</label>
...
```

Ahora debería funcionar, pero no hemos definido nada dentro de nuestro `Route::post`.

Request

Laravel envía los datos mediante una instancia de `Request`. Puedes aprender más sobre esta clase aquí, pero, de momento, nos conviene conocer lo siguiente:

- `request()`: obtiene la instancia de `Request` en cualquier parte del programa
- `request('campo')`: obtiene el valor de la variable "campo" contenida en la

petición HTTP, independientemente de si se trata de un `GET` o un `POST`. Es el equivalente a `$_GET['campo']` y `$_POST['campo']` de PHP puro
- `request->all()`: obtiene todos los datos enviados en la petición HTTP

Haremos uso de `request()` para recibir los datos del formulario. Podemos hacer esto:

```
Route::post('/ofertas', function(){
    Oferta::create([
        'titulo' => request('titulo'),
        'empresa' => request('empresa'),
        'descripcion' => request('descripcion')
    ]);
    return redirect('/ofertas');
});
```

Nota: es especialmente importante que hayas asignado adecuadamente los atributos `name` de los `input` del formulario de la vista `create`

También podremos hacer esto:

```
Route::post('/ofertas', function(){
    Oferta::create(request()->all());
    return redirect('/ofertas');
});
```

Pero nos dará una excepción `MassAssignmentException`. Esto se debe a otra medida de seguridad para evitar cambios en los campos **preasignados** (consulta la vista `create` si no recuerdas lo que es). Imagina que un usuario avispado modificase los datos enviados por un formulario de registro para modificar el valor **preasignado** en "rol" de "rol de usuario" a "rol de administrador". Si ejecutamos el último bloque de código, estaríamos aceptando a ciegas los valores que nos llegan. Para decidir qué valores aceptamos a ciegas, debemos hacer un pequeño cambio en `app/Http/Models/Oferta.php` y añadiremos la siguiente línea:

```
protected $fillable = ['titulo', 'empresa', 'descripcion'];
```

Ahora debería funcionar correctamente

Acción `update`

Esta acción no tiene una vista asociada, pero conecta conceptualmente con la vista `edit`. Al igual que con la acción `store`, empezaremos añadiendo la ruta según lo establecido en la tabla de nomenclatura estándar. Se espera un `PUT/PATCH` en `/ofertas/{id}`. La diferencia entre `PUT` y `PATCH` reside en "cuánto actualizamos el recurso". Si se actualiza completamente (se sustituye el recurso entero por otro diferente) se utilizaría la petición `PUT`. Si solamente quisiéramos actualizar algunos campos, usaríamos `PATCH`. Nosotros usaremos `PATCH`

```
Route::patch('/ofertas/{id}', function($id){
    $oferta = Oferta::find($id);
    $oferta->update(request()->all());
    return redirect('/ofertas');
});
```

Debemos actualizar la vista `edit` para que el atributo `action` del formulario apunte a nuestra ruta de actualización:

```
...
</head>
<body>
    <form action="/ofertas/{{ $oferta->id }}" method="post">
        @csrf
        @method('PATCH')
        <label for="titulo">Título del Trabajo:</label>
...
```

Acción `delete`

Esta acción no tiene una vista asociada. Para poder realizarla necesitamos un elemento de interfaz gráfica que nos permita realizar la acción HTTP descrita en la tabla de nomenclatura estándar. Con respecto a la acción, se espera una petición HTTP de `DELETE` sobre `/ofertas/{id}`. Dónde colocamos y utilizamos ese elemento de interfaz gráfica dependerá del diseño de nuestra aplicación. Podríamos querer tener la posibilidad de eliminar ofertas de trabajo desde:

- La vista `index`, uno para cada oferta de trabajo

- La vista `show`, para esa oferta en concreto
- La vista `edit`, para esa oferta en concreto, como parte de sus posibilidades de actualización

En cualquier caso, debe existir dicha acción dentro de nuestro fichero de rutas `routes/web.php`

```
Route::delete('/ofertas/{id}', function($id){
    Oferta::destroy($id);
    return redirect('/ofertas');
});
```

Para este ejemplo, modificaré la vista `index`, añadiendo un pequeño formulario con un único botón y los atributos `method` y `action` correspondientes para cada oferta de trabajo listada, quedando tal que así:

```
<!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>
            <th>Detalles</th>
            <th>Eliminar</th>
        </tr>
        @foreach($ofertas as $oferta)
        <tr>
            <td>{{ $oferta->titulo }}</td>
            <td>{{ $oferta->empresa }}</td>
            <td>{{ $oferta->descripcion }}</td>
```

```
            <td><a href="/ofertas/{{ $oferta->id }}">Ver
detalles</a></td>
            <td>
                <form method="POST" action="/ofertas/{{ $oferta->id
}}">
                    @method('DELETE')
                    @csrf
                    <button>Eliminar</button>
                </form>
            </td>
        </tr>
        @endforeach
    </table>
</body>
</html>
```

Recuerda que, puesto que se trata de formularios, deberemos de aplicar la protección anti-CSRF

Controlador de recursos

Para una única clase de nuestro negocio hemos creado en el fichero `routes/web.php` un total de 7 rutas. Imagina si tuviéramos una aplicación más grande, con 10 clases... Además de otras rutas no necesariamente relacionadas con operaciones CRUD de estas clases, claro. Ese fichero crecerá bastante y será cada vez más lento querer buscar un método concreto para modificarlo. Por tanto, debemos hacer una pequeña refactorización. Nos llevaremos los métodos CRUD a un controlador de recursos

Un controlador de recursos es una clase que agrupa la lógica de manejo de rutas CRUD para un único recurso (en nuestro caso, Oferta). La función es estandarizar la forma en que se manejan las operaciones HTTP comunes. Puesto que hemos tenido la precaución de seguir la nomenclatura estándar, esto será rápido e indoloro:

1. Crearemos un controlador haciendo uso de **artisan**:

   ```
   php artisan make:controller OfertaController -r
   ```

   El argumento `-r` es importante para que cree automáticamente el controlador con los métodos

2. Por laravarte de magia, se habrá creado una clase nueva en `app/Http/Controllers` con nombre `OfertaController`, y los métodos estarán nombrados acorde a la nomenclatura que no hacemos más que repetir, ¡yuju!

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class OfertaController extends Controller
{
    /**
     * Display a listing of the resource.
     */
    public function index()
    {
        //
    }

    /**
     * Show the form for creating a new resource.
     */
    public function create()
    {
        //
    }

    /**
     * Store a newly created resource in storage.
     */
    public function store(Request $request)
    {
        //
    }

    /**
     * Display the specified resource.
     */
    public function show(string $id)
    {
```

```
        //
    }

    /**
     * Show the form for editing the specified resource.
     */
    public function edit(string $id)
    {
        //
    }

    /**
     * Update the specified resource in storage.
     */
    public function update(Request $request, string $id)
    {
        //
    }

    /**
     * Remove the specified resource from storage.
     */
    public function destroy(string $id)
    {
        //
    }
}
```

3. Lo siguiente que tendremos que hacer es copiar la funcionalidad de cada uno de los métodos de nuestro fichero de rutas a los métodos correspondientes del controlador

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class OfertaController extends Controller
{
    /**
```

```php
     * Display a listing of the resource.
     */
    public function index()
    {
        $ofertas = Oferta::all();
        return view('ofertas.index', [
            'ofertas' => $ofertas
        ]);
    }

    /**
     * Show the form for creating a new resource.
     */
    public function create()
    {
        return view('ofertas.create');
    }

    /**
     * Store a newly created resource in storage.
     */
    public function store(Request $request)
    {
        Oferta::create($request->all());
        return redirect('/ofertas');
    }

    /**
     * Display the specified resource.
     */
    public function show(string $id)
    {
        return view('ofertas.show', [
            'oferta' => Oferta::find($id)
        ]);
    }

    /**
     * Show the form for editing the specified resource.
     */
```

```
    public function edit(string $id)
    {
        return view('ofertas.edit', [
            'oferta' => Oferta::find($id)
        ]);
    }

    /**
     * Update the specified resource in storage.
     */
    public function update(Request $request, string $id)
    {
        $oferta = Oferta::find($id);
        $oferta->update($request->all());
        return redirect('/ofertas');
    }

    /**
     * Remove the specified resource from storage.
     */
    public function destroy(string $id)
    {
        Oferta::destroy($id);
        return redirect('/ofertas');
    }
}
```

Apreciarás que hay un pequeño cambio, ya no vamos a usar `request()`, puesto que el propio método del controlador ya recibirá una instancia de `Request`, y esto está muy relacionado con el siguiente paso

Inyección de dependencias

La inyección de dependencias es un patrón de diseño basado en que una clase, cuando tiene que crear, modificar o utilizar instancias de otra de las que depende, en lugar de crear/cargar dichas instancias dentro de su propio código, se le "inyectan" (se le pasan) directamente para que cree las relaciones o haga las modificaciones adecuadas.

En nuestro controlador, tenemos varias acciones que reciben un `$id` de una oferta de trabajo que utilizamos para cargar dicha oferta y hacer las modificaciones adecuadas. Laravel facilita la inyección de dependencias simplemente cambiando el tipo de dato que recibe la función por parámetro. Como sabes, en PHP hay tipado dinámico y el intérprete hacer type-juggling al operar con los datos, pero eso no significa que no podamos hacer explícito el tipo que queremos que reciba una función y, aunque sea simplemente "informativo", es un perfecto indicador para que Laravel haga la inyección de dependencias.

Vamos a modificar los métodos que reciben un `$id` en nuestro controlador y que lo utilizan para cargar una `Oferta` para que reciban directamente una `Oferta`, que son: `show`, `edit` y `update`

```
...

/**
 * Display the specified resource.
 */
public function show(Oferta $oferta)
{
    return view('ofertas.show', [
        'oferta' => $oferta
    ]);
}

...

/**
 * Show the form for editing the specified resource.
 */
public function edit(Oferta $oferta)
{
    return view('ofertas.edit', [
        'oferta' => $oferta
    ]);
}

...

/**
```

```
     * Update the specified resource in storage.
     */
    public function update(Request $request, Oferta $oferta)
    {
        $oferta->update($request->all());
        return redirect('/ofertas');
    }

    ...
```

No es necesario cambiar el método `destroy` puesto que no cargamos la oferta, sino que la destruimos directamente, o bien podemos utilizar inyección de dependencias y usar el método `delete` de la instancia

```
    /**
     * Remove the specified resource from storage.
     */
    public function destroy(Oferta $oferta)
    {
        $oferta->delete();
        return redirect('/ofertas');
    }
```

Definición de la ruta de recurso

Ya hemos refactorizado el CRUD en el controlador de recursos, pero no lo hemos "conectado" con `routes/web.php` todavía. Si no lo hemos hecho ya, borraremos todas las rutas relacionadas con `Oferta`. Podemos definir las rutas hacia el controlador de 2 formas:

- Método a método: es útil, especialmente si no cumples con el nombrado de la nomenclatura estándar, y lo usaremos para definir otras rutas y funciones diferentes a CRUD en el controlador:

```
Route::controller(OfertaController::class) -
>group(function() {
    Route::get('/ofertas', 'index');
    Route::get('/ofertas/create', 'create');
    Route::get('/ofertas/{oferta}', 'show');
    Route::get('/ofertas/{oferta}/edit', 'edit');
    Route::post('/ofertas', 'store');
    Route::patch('/ofertas/{oferta}', 'update');
    Route::delete('/ofertas/{oferta}', 'destroy');
});
```

- Mediante la definición de la ruta de recurso, que es una única línea que registra automáticamente las siete rutas que mapean los verbos HTTP a los métodos correspondientes dentro de `OfertaController`. La sintaxis sigue el siguiente patrón:

  ```
  Route::resource('URI_base', Controlador::class);
  ```

  Por tanto, en nuestro caso sería:

  ```
  Route::resource('/ofertas', OfertaController::class);
  ```

  De nuevo, esto es posible hacerlo gracias a que nos hemos ajustado al nombrado de la nomenclatura estándar

Relaciones entre clases

Hasta ahora, nuestro modelo solamente cuenta con atributos que son datos escalares (cadenas, números, fechas.... Sin embargo, las aplicaciones del mundo real tienen datos que se conectan entre sí. Por ejemplo, un `Usuario` crea muchos `Posts`, y cada `Post` pertenece a un solo `Usuario`

Laravel simplifica la gestión de estas conexiones mediante **Eloquent ORM** (Object-Relational Mapper) y su sistema de **Relaciones**

¿Por qué necesitamos relaciones? Estas permiten interactuar con las tablas de una base de datos relacional simulando que se trata de una base de datos orientada a objetos. Para el programador, acceder a los posts de un usuario es tan sencillo como acceder a una propiedad pero, por detrás, el ORM lo traduce a una consulta SQL con `JOIN`.

Tipos fundamentales de relaciones

Uno a muchos (one to many)

Es la más común. Un modelo es el "padre", y tiene muchos modelos "hijos". Por ejemplo: un `Usuario` puede tener muchos `Posts`. **Implementación**: mediante los métodos `hasMany()` y `belongsTo()`:

- `Usuario`: puede tener muchos `Posts`: `hasMany()`
- `Post`: pertenece a un `Usuario`: `belongsTo()`

Uno a uno (one to one)

Un modelo solo puede estar asociado con una instancia de otro modelo. Por ejemplo: un `Usuario` solo puede tener un `Perfil`. **Implementación**: mediante los métodos `hasOne()` y `belongsTo()`:

- `Usuario`: tiene 1 `Perfil`: `hasOne()`
- `Perfil`: pertenece a un `Usuario`: `belongsTo()`

Muchos a muchos (many to many)

Un modelo puede estar relacionado con muchos otros modelos y viceversa. Requiere una **tabla pivote (intermedia)** en la base de datos. Por ejemplo: un `Post` puede tener muchas `Etiquetas` y una `Etiqueta` puede estar en muchos `Posts`. **Implementación**:

- `Post`: puede tener muchas `Etiquetas`: `belongsToMany()`
- `Etiquetas`: puede estar en muchos `Posts`: `belongsToMany()`

Aunque en lenguaje natural diríamos que "un post tiene varias etiquetas", es importante utilizar la relación `belongsToMany()` en lugar de la `hasMany()` para que Laravel busque la clave foránea a `Post` en otra tabla. ¿En cuál? La **tabla pivote**. Por defecto, tratará de buscar una tabla que siga la siguiente regla de nombrado:

- Se toman los nombres de las clases de los modelos involucrados y se convierten a

minúsculas y **singular**

    - `Post` se convierte en `post`
    - `Etiqueta` se convierte en `etiqueta`
- Se unen con un guión bajo `_` y se ordenan **alfabéticamente**, quedando tal que así:
    - `etiqueta_post`

Sin embargo, es posible especificar explícitamente el nombre de esta tabla, y esto es especialmente útil cuando exista más de 1 relación many to many entre 2 modelos

Por supuesto, tendremos que modificar nuestra migración para añadirle las claves foráneas, así como crear las tablas pivote necesarias

Tablas y claves

Primero vamos a crear un modelo nuevo al que llamaremos `Empresa`. ¿Cómo se relaciona esta con `Oferta`? Se trataría de una relación **uno a muchos**, puesto que una empresa puede publicar muchas ofertas de trabajo, pero una oferta de trabajo sólo puede estar publicada por una empresa

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

Modificamos `database/migrations/xxxx_create_empresas_table.php` para añadir los atributos que consideremos. Yo tengo lo siguiente:

```
Schema::create('empresas', function (Blueprint $table) {
    $table->id();
    $table->string('nombre');
    $table->string('ubicacion');
    $table->timestamps();
});
```

A continuación modificamos `database/migrations/xxxx_create_ofertas_table.php` para que el atributo `empresa`, que teníamos como `string`, se convierta en una clave foránea a `Empresa`

```
Schema::create('ofertas', function (Blueprint $table) {
    $table->id();
    $table->string('titulo');
    $table->text('descripcion');
    $table->foreignIdFor(Empresa::class);
    $table->timestamps();
});
```

Vamos a realizar la nueva migración y ver qué tenemos en base de datos

```
php artisan migrate:refresh
```

- Tabla `empresas` con los campos "nombre" y "ubicacion" de tipo `VARCHAR` cada uno
- Tabla `ofertas` con el campo "empresa" modificado. Este ha pasado a llamarse `empresa_id` y ya no es de tipo `VARCHAR` como antes, sino de tipo `INTEGER`

Sin embargo, si consultamos las claves foráneas en la tabla `ofertas`, apreciaremos que no existe una conexión con la tabla `empresas`. ¿Por qué? **Eloquent ORM** va a ser capaz de manejar las relaciones entre las clases por sí solo simplemente con las configuraciones que hagamos en las migraciones y los modelos, por lo que no necesita que esas restricciones existan en BBDD, por lo que no las crea a nivel de BBDD por defecto. Estas restricciones son fundamentales para mantener la integridad referencial, y no se me ocurre una buena razón para que Laravel no las cree por defecto. Para forzar a que las cree, hay que hacer que la relación sea restringida (`constrained`). Volvemos a modificar la migración de `ofertas`:

```
$table->foreignIdFor(Empresa::class)->constrained();
```

Hacemos de nuevo la migración

```
php artisan migrate:refresh
```

Con esto, ya deberíamos apreciar la restricción a nivel de BBDD.

Vamos a crear la factoría de `Empresa`, modificar la de `Oferta` y realizar la migración y poblado de base de datos de nuevo:

- Factoría de `Empresa`

```
return [
    'nombre' => fake()->company(),
    'ubicacion' => fake()->city(),
];
```

- Factoría de `Oferta`: tendremos que modificar el campo `empresa` por `empresa_id`, y aquí tenemos varias opciones:
    - Podemos decir que `empresa_id` apunte directamente a `Empresa::factory()` pero, si bien sí se respetará que el tipo de relación sea "uno a muchos", se percibirá como "uno a uno" porque creará una empresa aleatoria por cada oferta de trabajo
    - Idealmente tendremos un conjunto reducido de empresas que publican muchas ofertas de trabajo cada una. En la factoría, elegiríamos una aleatoria y crearemos la clave foránea

```
return [
    'titulo' => fake()->jobTitle(),
    'descripcion' => fake()->paragraph(),
    'empresa_id' => Empresa::inRandomOrder()->first()->id ??
Empresa::factory(),
];
```

  Si no existen empresas creadas, creará una. Para garantizar que sí que las hay, tendremos que llamar a las factorías en orden

A continuación vamos a `database/seeders/DatabaseSeeder.php` y la modificamos para que quede así:

```
Empresa::factory(10)->create();
Oferta::factory(100)->create();
```

Es muy importante el orden en este caso. Necesitamos crear las empresas antes que las ofertas para que las ofertas ya tengan empresas disponibles para rellenar la clave foránea

En base de datos apreciaremos que se han creado y conectado correctamente los datos

Métodos en Model

¿Qué tocaría ahora? Tenemos que modificar el modelo para indicarle a **Eloquent** a través de qué métodos se "conectarán" `Empresa` y `Oferta`. Para ello, iremos a `app/Http/Models` y haremos los siguientes cambios según lo que hemos indicado en el apartado de tipos fundamentales de relaciones

- `Empresa`: puede tener muchas `Ofertas`

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

- `Oferta`: pertenece a una `Empresa`

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

Vamos a probarlo con **artisan tinker**, que hace mucho que no lo usamos:

```
php artisan tinker
> $oferta = App\Models\Oferta::find(1);
> $oferta->empresa
```

Constatarás que, a pesar de que hemos definido la relación con un método, Laravel ha hecho de su magia y nos permite acceder a los datos como si fuera un atributo. De hecho, si intentamos llamar al método `empresa()`, simplemente nos dirá que es una instancia de relación `BelongsTo`. ¿Y si quisiéramos obtener todas las ofertas de esa empresa?

```
> $oferta->empresa->ofertas
```

Facilito

Reglas de integridad referencial

Vamos a intentar hacer una nueva miración:

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

Fallará, ¿por qué? Bueno, hemos definido una clave foránea desde `Ofertas` hasta `Empresas`, pero el último modelo que hemos creado es el de `Empresa`, por lo que Laravel, durante la migración, intentará eliminar primero las tablas más nuevas y luego las más viejas. Al eliminar datos de `Empresa`, se producirán efectos inesperados que no hemos considerado: una clave foránea con restricción `NOT NULL` en la tabla `ofertas` se quedará con valor `null` y esto, por cuestiones de integridad referencial, es incorrecto y muy incorrecto. Debemos definir qué se hace en estos casos, y para eso modificaremos de nuevo el fichero de migraciones de `Oferta`:

```
$table->foreignIdFor(Empresa::class)->constrained()-
>onDelete('cascade');
```

Que significa: cuando eliminen una `Empresa` a la que yo apunto con mi clave foránea, elimíname a mí también. Sin embargo, aunque hayamos añadido esta línea, recuerda que este método (`up()`) se ejecuta en el momento de creación de la tabla, y la BBDD ya está creada... ¿Cómo lo podemos apañar? Podríamos añadir la regla manualmente en BBDD para superar este bache, o podemos eliminar manualmente las tablas en otro orden para ir tirando. Desde el cliente de BBDD, eliminamos primero la tabla `ofertas` y luego la tabla `empresas`

Podemos comprobar que esto funciona adecuadamente al ejecutar 2 veces

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

La segunda vez es importante porque en la primera no borrará nada puesto que ya lo hemos borrado nosotros. En la segunda podremos apreciar cómo ya no se produce ese error

A partir de ahora tendremos que definir adecuadamente estas restricciones desde el principio para que no nos vuelva a pasar

Completar las relaciones

Uno a uno

Vamos a hacer rápidamente la relación uno a uno. Podríamos decir que una empresa tiene una `InformacionFiscal`. Así que haremos rápidamente:

1. Crear Model + Migration + Factory

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

   La primera en la frente... Esto creará una tabla llamada `informacion_fiscals`, lo cual no tiene mucho sentido en español. Tendremos que hacer cambios, no te preocupes

2. Modificar Migration para añadir los campos y restricciones que consideremos. Renombramos `xxxx_create_informacion_fiscals_table.php` a `xxxx_create_informaciones_fiscales_table.php`. Modificamos también el nombre en `Schema::create` y el `Schema:dropIfExists`. Método `up()`:

   ```
   Schema::create('informaciones_fiscales', function (Blueprint
   $table) {
       $table->id();

       $table->string('cif');
       $table->string('direccion');
       $table->foreignIdFor(Empresa::class)
                   ->unique()
                   ->constrained()
                   ->onDelete('cascade');

       $table->timestamps();
   });
   ```

   La restricción `unique()` no sería estrictamente necesaria, pero si queremos **garantizar** a nivel de BBDD una relación uno a uno, es importante ponerla

3. Modificar Factory para crear datos sin necesidad de definir interfaces gráficas

```
return [
    'cif' => fake()->numerify('##########'),
    'direccion' => fake()->address(),
    'empresa_id' => Empresa::factory()
];
```

4. Modificar Seeder para crear instancias de información fiscal. Importante: eliminamos la creación de empresas ya que la vamos a crear a través del Factory de InformacionFiscal

```
InformacionFiscal::factory(10)->create();
```

5. Modificar Model para indicarle a **Eloquent** cómo se reflejan esas relaciones. Primero vamos a arreglar lo del renombrado a "informaciones fiscales". En `app/Models/InformacionFiscal.php` añadiremos el siguiente atributo:

```
protected $table = 'informaciones_fiscales';
```

   Y, ya que estamos, añadimos la relación con `Empresa`

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

   Ahora modificamos `Empresa` para añadir la relación con `InformacionFiscal`

```
public function informacionFiscal() {
    return $this->hasOne(InformacionFiscal::class);
}
```

6. Ejecutar la migración cruzando los dedos por si la hemos liado con algo en el camino:

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

Vamos a tomarnos un descanso, aquí tienes un vídeo: https://www.youtube.com/watch?v=A-gCaAMGd-w

Muchos a muchos

A continuación toca plantear el tercer tipo de relación: muchos a muchos.

Un `Usuario` puede aplicar a varias `Ofertas`, y una oferta puede recibir aplicaciones de varios `Usuarios` (no me gusta nada lo de usar "aplicar" para las ofertas de trabajo, es un anglicismo). Por tanto, se trata de una relación many to many. Laravel trae por defecto una clase `User` que nos viene de perlas para este caso. Solo tiene un problema, y es que el nombre está en inglés. Podríamos renombrarlo, pero nos podríamos encontrar con efectos colaterales que nos causen quebraderos de cabeza debido a que es la clase que se utiliza para autenticación, por lo que no considero que merezca la pena. Trabajaremos en 2 idiomas: **español** e inglés

Como ya tenemos el modelo creado, estaría prácticamente hecho, sólo nos falta el "pegamento". En las relaciones muchos a muchos, como dijimos aquí, hace falta una **tabla pivote**. La convención para el nombrado de la tabla también está ahí especificada, y sería `oferta_user` (ordenado alfabéticamente y en snake_case). No obstante, cambiaremos el nombre de la tabla pivote a `candidaturas`. Esto nos obliga a ser más explícitos en el código, al igual que lo hemos sido para la tabla `informaciones_fiscales`

Hasta ahora hemos creado ficheros de migraciones gracias a `make:model -mf`. Sin embargo, ahora no queremos crear un modelo, sino simplemente una tabla que no va a tener un modelo asociado. Podemos hacer 2 cosas:

- O bien creamos la tabla en el fichero de migración de `xxxx_create_ofertas_table` (esto se puede hacer, fíjate dentro del fichero de migración de `xxxx_create_users_table`)
- O bien creamos una migración específica

La mejor práctica es la creación de una migración específica, ya que el orden importa muchísimo cuando se trata de integridad referencial. Tener un fichero de migración propio nos permite controlar mejor en qué orden se ejecutan las migraciones

Para crear solamente el fichero de migración haremos lo siguiente:

```
php artisan make:migration create_candidaturas_table
```

Esto creará el fichero `xxxx_create_candidaturas_table`, el cual procederemos a completar como sigue:

```
Schema::create('candidaturas', function (Blueprint $table) {
    $table->id();

    $table->foreignIdFor(Oferta::class)->constrained()-
>onDelete('cascade');
    $table->foreignIdFor(User::class)->constrained()-
>onDelete('cascade');
    $table->unique(['oferta_id', 'user_id']);

    $table->timestamps();
});
```

La línea `$table->unique(['oferta_id', 'user_id'])` evita que un usuario pueda aplicar 2 veces a la misma oferta de trabajo

Ejecutamos el comando de migración y nos aseguramos de que se crea la tabla en BBDD

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

A continuación modificaremos el modelo, tendremos que utilizar el `belongsToMany` tanto en `Oferta` como en `User` e, **importante**, tendremos que definir explícitamente el nombre de la tabla pivote, ya que no hemos seguido la regla de nombrado que Laravel reconoce por defecto

En `app/Models/Oferta.php` añadiremos:

```
public function candidatos() {
    return $this->belongsToMany(User::class, 'candidaturas');
}
```

En `app/Models/User.php` añadiremos:

```
public function ofertas() {
    return $this->belongsToMany(Oferta::class, 'candidaturas');
}
```

Y nos faltaría probarlo, pero nos faltan datos... ¿Cómo podemos generar automáticamente datos para probar esta relación? No tenemos un fichero de factoría... Pero no pasa nada, en realidad podemos hacerlo como queramos, por algo somos diose-digoooo informáticos

Modificaremos `database/seeders/DatabaseSeeder.php`:

```
InformacionFiscal::factory(10)->create();
$ofertas = Oferta::factory(100)->create();
$users = User::factory(20)->create();

foreach ($ofertas as $oferta) {
    $candidatosAleatorios = $users->random(5);
    $oferta->candidatos()->attach($candidatosAleatorios);
}
```

Este código asignará a cada oferta de trabajo 5 usuarios aleatorios

Volvemos a ejecutar

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

En principio ya estaría, pero en BBDD apreciaremos que las columnas de `created_at` y `updated_at` están a `NULL`. ¡Demonios! ¿Es que esta pesadilla no acaba nunca? ¿Qué ha pasado? Una tabla pivote, por lo general, no suele necesitar de esas 2 columnas. Las tenemos porque Laravel las ha metido por defecto en el fichero de migrations pero, al no tener un modelo asociado, **Eloquent** lo va a tratar de una forma distinta. Podemos hacer 2 cosas:

- Podemos eliminar el `$table->timestamps()` del fichero de migraciones si no nos interesa tener esa información para quedarnos con una tabla pivote clásica y aburrida. Si optas por esta opción, ya sabes qué fichero tocar y dónde está la puerta de salida
- O bien podemos indicarle a **Eloquent** explícitamente que sí rellene esas columnas al crear la relación. Si optas por esta opción, que nos permitiría ordenar cronológicamente las candidaturas, nos vamos de viaje a los modelos de `Oferta` y `User`

  En `Oferta`

```
public function candidatos() {
    return $this->belongsToMany(User::class,
'candidaturas')->withTimestamps();
}
```

En `User`

```
public function ofertas() {
    return $this->belongsToMany(Oferta::class,
'candidaturas')->withTimestamps();
}
```

Decidas lo que decidas, repite de nuevo el

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

Ahora sí lo tenemos. ¿No me crees? Compruébalo en la base de datos, yo mientras voy comprobando cómo se comporta usando los métodos del modelo:

```
php artisan tinker
> $oferta = App\Models\Oferta::find(1)
> $oferta->candidatos
> $user = App\Models\User::find(1)
> $user->ofertas
```

Con esto, hemos cubierto las relaciones más importantes entre clases en Laravel. Lo siguiente sería... *redoble de tambor* ¡Autenticación de usuarios! Podrán registrarse, hacer login, y gestionar ofertas de trabajo a las que apliquen. ¿No es excitante? Yo me voy a dormir, buenas noches

Autenticación y autorización

Toda aplicación que pretenda ofrecer un servicio personalizado a un usuario debe contar con una manera de permitir al usuario que se registre, se autentique y haga uso de tu aplicación desde su perfil personal. Ya tenemos unos cuantos usuarios creados, así que vamos a ir al grano

Inicio de sesión

Primero necesitaremos un formulario de login. Para eso crearemos un nuevo controlador. Puedes llamarlo como quieras, pero una buena práctica es llamarlo `SessionController` puesto que será el encargado de crear y destruir sesiones. Un formulario de login puede interpretarse como una vista de creación de sesión. ¿Recuerdas cómo se hacía?

```
public function create() {
    return view('auth.create');
}
```

Para ello, por supuesto, tendrás que crear un directorio `auth` dentro de `resources/views` y, dentro de ese directorio, un fichero `create.blade.php`. El nombrado de la vista no tiene por qué seguir exactamente el de los verbos HTTP, podría ser `login.blade.php` y la función `create` devolvería `auth.login`, sin problema. Ten en cuenta que una sesión no va a ser un modelo de nuestra aplicación, así que no tiene por qué seguir la nomenclatura estándar. Una buena práctica, en este caso sería registrar en `routes/web.php` las peticiones `GET` y `POST` a `/login` para la vista `create` y la acción `store`, respectivamente:

```
Route::get('/login', [SessionController::class, 'create'])
Route::post('/login', [SessionController::class, 'store'])
```

Por tanto, el formulario de login deberá hacer un `POST` a la ruta de login. De momento vamos a hacer lo siguiente:

```
public function store() {
    $attributes = request()->validate([
        'email' => ['required', 'email'],
        'password' => ['required']
    ]);
}
```

Este es un buen momento para hablar de la función `validate`. Nos permite aplicar una serie de restricciones sobre los datos que nos llegan desde un formulario y comunicar errores de forma muy sencilla. Para poder mostrar esos errores, simplemente tendremos que añadir a nuestra vista Blade la sintaxis `@error('nombre_campo')`. Por ejemplo:

```
...
<label for="email">Email: </label><br>
<input id="email" type="email" name="email">
@error('email')
    <p style="color:red">{{ $message }}</p>
@enderror
...
```

Este mensaje estará en inglés. Puede ponerse en español cambiando la configuración de la localización de Laravel o podemos poner mensajes personalizados.

```
public function store() {
    $attributes = request()->validate([
        'email' => ['required', 'email'],
        'password' => ['required']
    ],
    [
        'email.required' => 'El campo email es obligatorio.',
        'email.email' => 'El campo email debe ser un correo
electrónico válido.',
        'password.required' => 'El campo contraseña es
obligatorio.'
    ]);
}
```

Esto es muy mejorable, pero no es lo más importante ahora mismo, vamos a seguir por el "happy path" de momento para ver cómo podemos terminar de iniciar sesión. Laravel trae una clase llamada `Auth` que nos permite hacer fácilmente todo lo que con PHP puro tendríamos que hacer "a pelo":

- Verificación de credenciales: obtener la contraseña hasheada de la base de datos y comparar con la contraseña introducida por el usuario
- Iniciar sesión: llamar a `session_start` e introducir en `$_SESSION['user_id']` el id del usuario
- Verificación en cada página si existe `$_SESSION['user_id']` para tomar decisiones sobre lo que el usuario puede o no hacer

Usaremos esa clase de la siguiente forma:

```
public function store() {
    $attributes = request()->validate([
        'email' => ['required', 'email'],
        'password' => ['required']
    ],
    [
        'email.required' => 'El campo email es obligatorio.',
        'email.email' => 'El campo email debe ser un correo
electrónico válido.',
        'password.required' => 'El campo contraseña es
obligatorio.'
    ]);

    if (!Auth::attempt($attributes)) {
        throw ValidationException::withMessages([
            'email' => 'Lo siento, me he equivocado y no volverá a
ocurrir'
        ])
    }

    request()->session()->regenerate(); // Previene token hijacking

    return rediret ('/ofertas');
}
```

Un pequeño aperitivo para comprobar si funciona es ocultar una parte de la aplicación a menos que hayas iniciado sesión. Para facilitarlo, **Blade** ofrece una sintaxis muy sencilla: `@guest` y `@auth`. Se me ocurre lo siguiente:

- Añade un botón de `Iniciar sesión` que sólo se vea cuando no hayas iniciado sesión (`@guest`)
- Añade un botón de `Cerrar sesión` que sólo se vea cuando hayas iniciado sesión (`@auth`)

Pequeña refactorización a `SessionRequest`

En lugar de tener la lógica de validación de los datos del formulario en el controlador, podemos llevarla a una clase que se encargue solamente de la validación de estos datos y los mensajes de error. Primero crearemos la clase con Artisan:

```
php artisan make:request SessionRequest
```

Esto creará un fichero `SessionRequest` dentro de `app/Http/Requests`. Esta cuenta con 3 métodos:

- `authorize()`: qué debe cumplirse para que un usuario pueda realizar esta petición. No se trata de una autenticación o una autorización, que supondría validar los datos del formulario y comprobar los permisos del usuario, sino de si puede iniciar la petición. Al tratarse de un inicio de sesión, lo normal es que sea que `true`, ya que no tenemos muchos más datos que podamos comprobar para decidir si el usuario puede hacer la petición o no, pero podría no ser el caso. A lo mejor podrías restringir estas peticiones a los usuarios que no superen un captcha, sean menores de edad o que su IP pertenezca a un rango determinado (por ejemplo, IPs reconocidas de servidores VPN, proxies, bots...)
- `rules()`: reglas de validación del formulario
- `messages()`: mensajes de feedback para cuando falla alguna de las reglas de validación

Podemos realizar la siguiente refactorización:

```
public function rules(): array
{
    return [
        'email' => ['required', 'email'],
        'password' => ['required']
    ];
}
```

```
public function messages()
{
    return [
        'email.required' => 'El campo email es obligatorio',
        'email.email' => 'El campo email debe ser un correo
electrónico válido',
        'password.required' => 'La contraseña es obligatoria',
    ];
}
```

Estos 2 métodos se reparten toda la responsabilidad que antes tenía el `request()->validate(...)`

Sólo hay que hacer 2 cambios más:

- Por un lado, el método `store` de `SessionController` recibe un `SessionRequest`
- Por otro lado, ya no llamamos al método `validate(...)`, sino al método `validated()`, que devuelve los atributos que superan la validación del formulario y sus valores

Quedaría tal que así:

```
public function store(SessionRequest $request) {
    $attributes = $request->validated();

    if (!Auth::attempt($attributes)) {
        throw ValidationException::withMessages([
            'email' => 'Lo siento, me he equivocado y no volverá a
ocurrir'
        ])
    }

    request()->session()->regenerate(); // Previene token hijacking

    return rediret ('/ofertas');
}
```

Cierre de sesión

Para cerrar sesión se usa `POST` a `/logout`

```
public function destroy() {
    Auth::logout()
}
```

Autorización: ¡no puedes pasar!

Vamos a trabajar la autorización de forma iterativa. Primero prohibiremos acciones que el usuario no puede hacer a menos que haya hecho login. Por ejemplo: editar una oferta de trabajo. Modificamos `OfertaController.php`

```
public function edit(Oferta $oferta)
{
    if(Auth::guest()) {
        return redirect('/ofertas');
    }
    return view('ofertas.edit', [
        'oferta' => $oferta
    ]);
}
```

Esto es muy mejorable. Vamos a crear una nueva relación. Podríamos suponer que nuestra aplicación la van a usar tanto usuarios que buscan trabajo como usuarios que ofrecen trabajo a través de sus empresas. Un usuario podría querer registrar su empresa y publicar ofertas para esa empresa. ¿Tiene sentido? Podríamos decir que existe una relación one-to-many, porque alguien puede ser muy emprendedor y tener muchas empresas: una empresa **pertenece** a un usuario y un usuario **puede tener** varias empresas:

1. Modificamos migration de empresas

```
Schema::create('empresas', function (Blueprint $table) {
    $table->id();
    $table->foreignIdFor(User::class)->constrained()-
>cascadeOnDelete();
    $table->string('nombre');
    $table->string('ubicacion');
    $table->timestamps();
});
```

2. Modificamos el factory de empresas

```
return [
    'nombre' => fake()->company(),
    'ubicacion' => fake()->city(),
    'user_id' => User::factory(),
];
```

3. Modificamos el model de Empresa

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

4. Modificamos el model de User

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

Le pediremos a nuestro amigo Artisan que nos cree de nuevo la BBDD y seguimos:

```
php artisan migrate:refresh
```

Y ahora volvemos a `OfertaController` y modificamos los permisos para el método `edit`

```
public function edit(Oferta $oferta)
{
    if(Auth::guest()) {
        return redirect('/ofertas');
    }
```

```
    if($oferta->empresa->user->isNot(Auth::user())){
        abort(403);
    }

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

Gate

Esta lógica de acceso podría refactorizarse haciendo uso de una clase de Laravel llamada `Gate`. Estas se definen en el método `boot` de `app/Http/Providers/AppServiceProvider.php`

```
public function boot(): void
{
    Gate::define('edit-oferta', function (User $user, Oferta
$oferta){
        return $oferta->empresa->user->is($user);
    });
}
```

Es **muy importante** que importes `Gate` desde `Illuminate\Support\Facades`

Volvemos al controlador de Oferta y...

```
public function edit(Oferta $oferta)
{
    Gate::authorize('edit-oferta', $oferta);

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

Siempre hacemos que la función crezca para luego hacerla decrecer, ¿no es guay?

Fíjate que solamente hemos llamado a `authorize` con `$oferta` como argumento. Esto es porque Laravel llamará a todos los métodos de `Gate` pasándole el usuario de `Auth` automáticamente. Una vez definimos un `Gate` podemos, además, llamar a métodos como:

```
Auth::user()->can('edit-oferta', $oferta);
Auth::user()->cannot('edit-oferta', $oferta);
```

Por si nos hiciera falta en algún momento, pero... **pero**... **PERO** hay algo mucho más guay, que es la directiva `@can` en **Blade**. Vamos a modificar el código de `ofertas.index` para que solamente muestre el enlace a `Editar` si el usuario puede editar esa oferta:

```
...
<td><a href="/ofertas/{{ $oferta->id }}">Ver detalles</a></td>
<td>
    @can('edit-oferta', $oferta)
        <a href="/ofertas/{{ $oferta->id }}/edit">Editar</a>
    @endcan
</td>
<td>
    <form method="POST" action="/ofertas/{{ $oferta->id }}">
...
```

Ahora que tenemos toda la motivación por haber ocultado un botón no autorizado vamos a volver a `OfertaController`. Queremos proteger el resto de métodos y nos daremos cuenta de una cosa: tenemos que repetir `Gate::authorize` en todos lados, qué fastidio...

Middleware

Para mejorar esto, Laravel propone un mecanismo llamado "middleware", que es como una serie de "barreras" que el usuario tiene que pasar antes de poder llegar a ejecutar un método. Hay un middleware para comprobar si un usuario ha hecho login, que es `auth`, y hay otro middleware por cada `Gate` que hayamos definido, vamos a probarlo en `routes/web.php`

```
Route::resource('ofertas', OfertaController::class);
```

Esa es la declaración que define todas las rutas de nomenclatura estándar para nuestras ofertas de trabajo. ¿Cuáles queremos proteger frente a usuarios sin autorización? A mí se me ocurren: vista `edit`, acción `update` y acción `destroy`. Tendríamos que definir por un lado aquellas rutas aceptadas siempre mediante `only` y aquellas rutas aceptadas sólo cuando el usuario cumple las condiciones con `except` y `middleware`. Sería algo como

```
Route::resource('ofertas', OfertaController::class)->only(['index',
'create', 'show', 'store']);
Route::resource('ofertas', OfertaController::class)-
>except(['edit', 'update', 'destroy'])->middleware(['auth', 'edit-
oferta']); // 'edit-oferta' está mal, esto es ilustrativo
```

Sin embargo, podemos pensar que para que un usuario pueda crear y guardar una oferta (create, store), debería hacer login, así que tendríamos que añadir algo como:

```
Route::resource('ofertas', OfertaController::class)->only(['index',
'show']);
Route::resource('ofertas', OfertaController::class)-
>except(['create', 'store'])->middleware('auth');
Route::resource('ofertas', OfertaController::class)-
>except(['edit', 'update', 'destroy'])->middleware(['auth', 'edit-
oferta']); // 'edit-oferta' está mal, esto es ilustrativo
```

Hemos llegado a un punto en el que se complica bastante la lectura, así que vamos a hacer regresión a un estado anterior que nos va a facilitar un poco la vida. Supongo que te acuerdas de:

```
Route::controller(OfertaController::class) ->group(function() {
    Route::get('/ofertas', 'index');
    Route::get('/ofertas/create', 'create');
    Route::get('/ofertas/{oferta}', 'show');
    Route::get('/ofertas/{oferta}/edit', 'edit');
    Route::post('/ofertas', 'store');
    Route::patch('/ofertas/{oferta}', 'update');
    Route::delete('/ofertas/{oferta}', 'destroy');
});
```

Vamos a migrarlo línea a línea y apuntando al método de su controlador:

```
Route::get('/ofertas', [OfertaController::class, 'index']);
Route::get('/ofertas/create', [OfertaController::class, 'create']);
Route::get('/ofertas/{oferta}', [OfertaController::class, 'show']);
Route::get('/ofertas/{oferta}/edit', [OfertaController::class,
'edit']);
Route::post('/ofertas', [OfertaController::class, 'store']);
Route::patch('/ofertas/{oferta}', [OfertaController::class,
'update']);
Route::delete('/ofertas/{oferta}', [OfertaController::class,
'destroy']);
```

Ahora tenemos una visión más clara tanto de las URIs como capacidad para añadir con mayor nivel de detalle las autorizaciones específicas en cada acción:

```
Route::get('/ofertas', [OfertaController::class, 'index']);
Route::get('/ofertas/create', [OfertaController::class, 'create']);
Route::get('/ofertas/{oferta}', [OfertaController::class, 'show']);
Route::get('/ofertas/{oferta}/edit', [OfertaController::class,
'edit'])
    ->middleware(['auth', 'can:edit-oferta,oferta']);
Route::post('/ofertas', [OfertaController::class, 'store'])
    ->middleware('auth');
Route::patch('/ofertas/{oferta}', [OfertaController::class,
'update'])
     ->middleware(['auth', 'can:edit-oferta,oferta']);
Route::delete('/ofertas/{oferta}', [OfertaController::class,
'destroy'])
     ->middleware(['auth', 'can:edit-oferta,oferta']);
```

La sintaxis middleware para aplicar un `Gate` a una ruta es un poco fea. Es necesario que sepas cómo se hace, pero también se puede dejar más bonita así:

```
...
Route::get('/ofertas/{oferta}/edit', [OfertaController::class,
'edit'])
    ->middleware('auth')
    ->can('edit-oferta', 'oferta']);
Route::post('/ofertas', [OfertaController::class, 'store'])
    ->middleware('auth'];
Route::patch('/ofertas/{oferta}', [OfertaController::class,
'update'])
    ->middleware('auth')
    ->can('edit-oferta', 'oferta']);
Route::delete('/ofertas/{oferta}', [OfertaController::class,
'destroy'])
    ->middleware('auth')
    ->can('edit-oferta', 'oferta']);
...
```

Una vez hemos definido esto, podemos ir a `OfertaController` y dejar el método `edit` como estaba:

```
public function edit(Oferta $oferta)
{
    return view('ofertas.edit', [
        'oferta' => $oferta
    ]);
}
```

Ya que el `Gate` se aplicará a través del `Middleware`

Políticas

Si bien `Gate` es útil para definir restricciones genéricas, como el manejo de permisos simple y de manera global (acceder a secciones de una página según un rol de usuario, por ejemplo), hay una forma de agrupar la lógica de autorización relativa a las operaciones CRUD sobre el modelo. A esto se le llama `Policy`

```
php artisan make:policy
```

Nos pedirá varios datos:

```
> OfertaPolicy
> Oferta
```

Esto creará un nuevo directorio `app/Policy` con el fichero `OfertaPolicy` y un montón de métodos. Vamos a borrar todos y añadir 1, de momento:

```
public function edit(User $user, Oferta $oferta) {

}
```

Aquí copiaremos el código que pusimos en nuestro `Gate`, que te recuerdo que está en `app/Providers/AppServiceProvider.php`:

```
public function edit(User $user, Oferta $oferta) {
    return $oferta->empresa->user->is($user);
}
```

Con esto podemos comentar (o eliminar) ese Gate y modificar las referencias que hubiera, que te recuerdo que están en:

1. `routes/web.php`: cambiamos `edit-oferta` por `edit`

```
Route::get('/ofertas/{oferta}/edit',
[OfertaController::class, 'edit'])
    ->middleware('auth')
    ->can('edit', 'oferta']);
Route::post('/ofertas', [OfertaController::class, 'store'])
    ->middleware('auth'];
Route::patch('/ofertas/{oferta}', [OfertaController::class,
'update'])
    ->middleware('auth')
    ->can('edit', 'oferta']);
Route::delete('/ofertas/{oferta}', [OfertaController::class,
'destroy'])
    ->middleware('auth')
    ->can('edit', 'oferta']);
```

2. Vista `index`: cambiamos `edit-oferta` por `edit`

```
...
<td><a href="/ofertas/{{ $oferta->id }}">Ver detalles</a>
</td>
<td>
    @can('edit', $oferta)
        <a href="/ofertas/{{ $oferta->id }}/edit">Editar</a>
    @endcan
</td>
<td>
    <form method="POST" action="/ofertas/{{ $oferta->id }}">
...
```

Creación de formulario de registro

Ya hemos hecho login y manejado la autorización, pero ahora falta el registro. Observa el fichero de migración de `User`:

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

¿Cuál de estos campos crees que tendremos que rellenar durante un registro?

- **Name**: representa el nombre del usuario
- **Email**: representa el email del usuario
- **Email verified at**: representa la fecha en que el usuario confirmó su email
- **Password**: representa la contraseña del usuario

Creo que coincidimos en que deberá ser "name", "email" y "password". Si quisiéramos añadir algún campo más a nuestro usuario, este sería el mejor momento. ¿Queremos saber su fecha de nacimiento, su color favorito, su tarjeta de crédito?

Un formulario de registro no es otra cosa que una vista `create` de `User`

Respira: Laravel Breeze

Construir desde 0 todo un sistema de autenticación es muy difícil pero, por suerte, Laravel nos facilita de nuevo la vida con su magia

Vamos a crear un nuevo proyecto de Laravel:

```
laravel new test_auth
```

Lo haremos siguiendo los pasos de creación de un proyecto. Una vez termine, haremos lo siguiente:

```
cd test_auth
composer require laravel/breeze
php artisan breeze:install
php artisan migrate
npm install
```

Esto descargará un "starter kit" de Laravel específico para autenticación y lo instalará en nuestro proyecto.

Importante: este proceso de instalación puede que sobreescriba algunos ficheros que hayamos modificado, como `routes/web.php`. Yo aconsejo hacer todo esto en un proyecto nuevo y mover tu programa a un nuevo proyecto, pero si tienes clarísimo cuáles son los cambios que has hecho para arreglar en caso de estropicio, adelante, hazlo sobre tu propio proyecto (haz antes un backup)

Depuración

Ya sabemos cómo crear el modelo y las relaciones entre las distintas entidades, así como obtener datos para mostrarlos en la vista a través del controlador pero, ¿qué está pasando por detrás? No hay magia, por detrás lo que sucede es PHP puro y consultas a bases de datos. Vamos a instalar una extensión en nuestro proyecto que nos permita visualizar las consultas que se ejecutan:

```
composer require barryvdh/laravel-debugbar --dev
```

Tras ello, arrancamos la aplicación y entramos en `/ofertas`. Apreciaremos una barra inferior que antes no teníamos y que nos permite explorar distinta información de interés pero, sobre todo, nos interesa la pestaña de `Queries`. Para poder cargar esta vista, se han realizado, para 100 ofertas, 201 consultas a base de datos:

- 1 para obtener la sesión actual
- 1 para obtener los datos del usuario que ha iniciado sesión
- 1 para cargar todas las ofertas
- 99 para cargar la información específica de la empresa que gestiona dicha oferta
- 99 para cargar los usuarios que gestionan la empresa que gestionan las ofertas (para poder aplicar la `Policy`)

Esto sucede por una técnica de recuperación de datos llamada **lazy loading** (carga perezosa). Esto es: solo hago la consulta a BBDD cuando necesito el dato. La técnica opuesta se llama **eager loading** (carga ansiosa). Es decir: cargo todo lo que pueda necesitar de antemano

Por cierto, antes de seguir, este complemento que hemos instalado puede exponer demasiada información. Es útil para desarrollo y depuración, pero no debe ejecutarse **nunca** en producción. Qué suerte que cambiando lo siguiente en `.env` desactive dicho complemento:

```
APP_DEBUG=false
```

Al indicar que la aplicación no está en modo depuración, comprobarás que la barra se desactiva. Aún así, si quieres eliminar el complemento del todo, puedes hacer la operación opuesta al composer require, que es:

```
composer remove barryvdh/laravel-debugbar
```

Carga perezosa VS carga ansiosa

Es importante estudiar en qué casos conviene aplicar la carga perezosa y en qué casos conviene aplicar la carga ansiosa: ¿qué datos va a necesitar esta vista?

Vamos a optimizar nuestro código, empezando por la carga de datos de las empresas. Modificamos `index()` de `OfertaController`. Recuerdo lo que tenemos hasta ahora:

```
public function index()
{
    return view('ofertas.index', [
        'ofertas' => Oferta::all()
    ]);
}
```

Aquí estamos aplicando la carga perezosa: dame todas las ofertas. ¿Qué datos de las ofertas? Pues todo lo que no sean relaciones. Para cargar las relaciones, haremos uso de `with`. La sintaxis es la siguiente:

```
Model::with('nombreRelacion')->get()
```

En nuestro caso:

```
public function index()
{
    return view('ofertas.index', [
        'ofertas' => Oferta::with('empresa')->get()
    ]);
}
```

Con esto, reduciremos **drásticamente** el número de consultas a bases de datos, hemos pasado de 201 consultas a solamente 14. Estas 14 vienen de los usuarios que necesitamos cargar para aplicar las `Policies`. Observa, además, que se ha reducido el tiempo de la consulta en, aproximadamente, un 30% (en mi caso, he pasado de 120ms a 80ms)

Pero hay un problema: `user` es una relación que está en `empresa`, no en `oferta`. ¿Cómo cargamos de forma ansiosa esa relación con el `with`? Con las relaciones anidadas. Estas se anidan con `.`, y la sintaxis sería algo como:

```
Model::with('nombreRelacion.nombreRelacion2')->get()
```

Ten en cuenta que `nombreRelacion2` "cuelga" de `nombreRelacion`, por lo que tendremos que cargar ansiosamente los datos de `nombreRelacion` primero:

```
Model::with(['nombreRelacion', 'nombreRelacion.nombreRelacion2'])-
>get()
```

En nuestro caso:

```
public function index()
{
    return view('ofertas.index', [
        'ofertas' => Oferta::with(['empresa', 'empresa.user'])-
>get()
    ]);
}
```

Y ya hemos pasado de 14 consultas a solamente 5

Ahora vamos a intentar reducir los datos que traemos de BBDD. ¿Realmente necesitamos todo? En la vista index necesitamos:

- `id`, `nombre` y `user_id` de Empresa. El `id` para parametrizar las URLs de `show` y `destroy`. El `nombre` para rellenar la columna **Empresa** y `user_id` para poder cargar el usuario
- `id` de User. El `id` para poder llamar a la `Policy`

Para ello, haremos uso de la selección de campos, con `:`. La sintaxis es la que sigue:

```
Model::with('nombreRelacion:campo1,campo2,campo3')->get()
```

En nuestro caso:

```
public function index()
{
    return view('ofertas.index', [
        'ofertas' => Oferta::with([
            'empresa:id,nombre,user_id',
            'empresa.user:id'
        ])->get()
    ]);
}
```

En realidad no necesitaríamos cargar el `User`, ya que nuestra `Policy` comprueba el `id`, que ya tenemos en `user_id` de `Empresa`, pero es preferible dejarlo como está, ya que la `Policy` podría complicarse más en el futuro y necesitar de más datos de `User`

Filtros sobre colecciones

Siguiente pregunta. ¿Una empresa debe poder ver todas las ofertas o solamente aquellas que ha publicado? Como siempre: depende del diseño que elijas para tu aplicación. Si optas por que una empresa puede ver todo y solamente gestionar sus ofertas, pues ya hemos terminado. Si quieres quedarte y seguir aprendiendo, vamos a utilizar filtros sobre colecciones

has()

Este método nos permite filtrar los datos a cargar en función de si está presente una relación o no. Supongamos que queremos cargar solamente aquellas ofertas de trabajo que tienen **al menos** un candidato

```
public function index()
{
    return view('ofertas.index', [
        'ofertas' => Oferta::with([
            'empresa:id,nombre,user_id',
            'empresa.user:id'
        ])
        ->has('candidatos')
        ->get()
    ]);
}
```

Para poder probarlo adecuadamente quizás tengas que borrar datos en BBDD. Comprueba cuántas filas recupera antes y después de añadir `has()`. Eso puedes comprobarlo en la pestaña "Models" de la barra de depuración

where()

Este método nos permite aplicar cualquier filtro de consulta de BBDD que se nos ocurra, es realmente potente. **Y recuerda**: es necesario hacerlo en el **controlador**. Dentro de una vista hay código PHP, y nada nos impide aplicar todos estos filtros dentro de la propia vista haciendo uso de la sintaxis de PHP en lugar de la de SQL, pero esto **viola** el patrón de diseño de MVC. La vista solamente debe recibir los datos y, como mucho, recorrer las relaciones, pero toda la carga debe realizarse en el controlador

La sintaxis del método `where` es la siguiente:

```
Model::where('nombre_atributo', 'operador', 'valor');
```

Por ejemplo, podríamos filtrar aquellas ofertas en cuyo título se incluya una palabra concreta, como "Administrative", de la siguiente manera:

```
Oferta::where('titulo', 'like', '%Administrative%')
```

Vamos a mostrar temporalmente cómo se vería, porque esto nos interesa para un poco más adelante:

```
public function index()
{
    return view('ofertas.index', [
        'ofertas' => Oferta::with([
            'empresa:id,nombre,user_id',
            'empresa.user:id'
        ])
        ->has('candidatos')
        ->where('titulo', 'like', '%Administrative%')
        ->get()
    ]);
}
```

Como he mencionado, vamos a quitar este `where` de momento para aplicarlo un poco más adelante

whereHas()

Funciona como una mezcla entre `has()` y `where`. Es decir, nos permite especificar la condición que debe cumplir una relación para incluirse

La sintaxis del método `whereHas` es la siguiente:

```
Model::whereHas('nombreRelacion', function($query) {
    $query->where();
})->get()
```

En nuestro caso:

```
whereHas('empresa', function($query) {
    $query->where('user_id', '=', Auth::id());
})
```

Vamos a aplicarlo en su conjunto:

```
public function index()
{
```

```
    return view('ofertas.index', [
        'ofertas' => Oferta::with([
            'empresa:id,nombre,user_id',
            'empresa.user:id'
        ])
        ->has('candidatos')
        ->whereHas('empresa', function($query) {
            $query->where('user_id', '=', Auth::id());
        })
        ->get()
    ]);
}
```

¡Enhorabuena! Hemos pasado de la siguiente consulta:

```
select * from "ofertas"
```

que luego derivaba en chorrocientas consultas más para cargar datos de empresa y usuario a la siguiente consulta:

```
select * from "ofertas" where exists (select * from "users" inner
join "candidaturas" on "users"."id" = "candidaturas"."user_id"
where "ofertas"."id" = "candidaturas"."oferta_id") and exists
(select * from "empresas" where "ofertas"."empresa_id" =
"empresas"."id" and "user_id" = 14)
```

que carga y filtra todos los datos de antemano

when()

Imagina que el usuario quiere utilizar un formulario de búsqueda para filtrar por ofertas de trabajo concretas, según una serie de criterios como el título o su descripción. Podemos hacer uso de `request()` para esto. El problema es que no siempre el usuario nos va a pedir ese filtro. Podríamos comprobar con una estructura condicional los casos en los que me llega el `request('titulo')` o el `request('descripcion')`, o lo que sea, pero Laravel nos proporciona un método `when()` para estos casos. Este método viene a decir algo "aplica esta restricción sólo cuando se cumpla esta condición". Esta condición puede ser

que un usuario interactúe con un formulario de búsqueda o puede ser cualquier otra. Por ejemplo, "si el usuario que ha iniciado sesión **no es** superadministrador" aplico todos los filtros anteriores, de manera que un usuario que sea superadministrador tendría acceso a todo el listado de ofertas independientemente de si tiene una empresa asociada o no

La sintaxis de `when` es la siguiente:

```
Model::when(condicion, function($query) {
    ...
})
```

Vamos a volver al caso del formulario de búsqueda. Quiero filtrar por `titulo` cuando el usuario introduzca un valor de búsqueda para el título, y por `descripcion` cuando el usuario introduzca un valor de búsqueda para la descripción, y por `titulo` y `descripcion` cuando el usuario introduzca valores de búsqueda para título y descripción:

```
public function index()
{
    return view('ofertas.index', [
        'ofertas' => Oferta::with([
            'empresa:id,nombre,user_id',
            'empresa.user:id'
        ])
        ->has('candidatos')
        ->whereHas('empresa', function($query) {
            $query->where('user_id', '=', Auth::id());
        })
        ->when(request('titulo'), function ($query) {
            $query->where('titulo', 'like', '%' . request('titulo')
. '%');
        })
        ->when(request('descripcion'), function ($query) {
            $query->where('descripcion', 'like', '%' .
request('descripcion') . '%');
        })
        ->get()
    ]);
}
```

Faltaría añadir un formulario que nos permita mandar los criterios de búsqueda al servidor, o bien podemos hacerlo manualmente en la barra de direcciones:

```
http://localhost:8000/ofertas?titulo=Administrative
```

```
http://localhost:8000/ofertas?descripcion=ullam
```

```
http://localhost:8000/ofertas?
titulo=Administrative&descripcion=ullam
```

Y, aplicando el caso descrito anteriormente, suponiendo que tuviéramos un sistema basado en roles adecuado, que no tenemos, esto es **simplemente orientativo**, podríamos hacer algo como:

```
public function index()
{
    return view('ofertas.index', [
        'ofertas' => Oferta::with([
            'empresa:id,nombre,user_id',
            'empresa.user:id'
        ])->when(!Auth::user()->hasRole('superadministrador'),
function ($query) {
            $query->has('candidatos')
                ->whereHas('empresa', function ($subQuery) {
                    $subQuery->where('user_id', '=', Auth::id());
                });
        })
        ->get()
    ]);
}
```