


The Fancy Gallery

Trabajo Final

2º Desarrollo Aplicaciones Multiplataforma, Dual

Sergio Gómez Santos, DAM2D

<https://github.com/SergioNodriza/gallery>



Índice

1. Análisis previo	2
1.1 Descripción del producto o proyecto a realizar.	2
1.2 Descripción de la empresa.	3
1.3 Justificación del proyecto.	3
1.4 Planificación del proyecto.	4
2. Modelado: Análisis y diseño.	5
2.1 Diagramas de casos de uso	5
2.2 Diagramas de ER y Relacional y scripts	8
2.3 Diagrama de Clases	11
2.4 Diseño de Interfaces	12
3. Implementación, despliegue y pruebas realizadas	13
3.1 Packages Utilizados	13
3.2 Ejemplo Testing	13
3.3 Ejemplo Fixtures	16
3.4 Ejemplo Permisos por Roles	16
3.5 Ejemplo Implementación de una Llamada	17
4. Conclusiones	19
5. Bibliografía	19

1. Análisis previo

1.1. Descripción del producto o proyecto a realizar.

TFG, The Fancy Gallery, es un proyecto que consiste en una API para una galería de fotos.

Las distintas funcionalidades son:

- Sistema de Usuarios
 - Registrar/Editar
 - Login
 - Roles
 - Permisos
- Fotografías
 - Subir/Editar/Eliminar
 - Interactuar
- Grupos
 - Crear/Editar/Eliminar
 - Añadir/Quitar fotos

El objetivo es crear un API funcional con estas ideas para poder empezar a utilizarla y dejar igualmente abierta la posibilidad de expandirse y abarcar nuevas funcionalidades.

1.2. Descripción de la empresa.

Como en el inicio de TFG, el alcance no sería demasiado grande, se empezaría dándome de alta como autónomo trabajando yo solo, y si el proyecto escalase, se plantearían otras opciones.

Para que TFG empiece a ser rentable económicamente, la parte funcional de los grupos es de pago. Solo los usuarios “premium” podrán acceder a esas funcionalidades.

1.3. Justificación del proyecto.

En el mercado ya existente para este tipo de aplicaciones hay 2 grandes nombres: Instagram y Pinterest. TFG se distingue de ambas en su objetivo.

Mientras que otras marcas apuntan a un público más generalizado, TFG va dirigido a un nicho más específico, se centraría en su uso para profesionales de la fotografía, dándoles un lugar para exponer su trabajo en un círculo más relacionado con este mundo.

Así, la idea de interactuar con las fotos o las opciones de grupos, se pueden asociar a formas de buscar inspiración para su trabajo.

1.4. Planificación del proyecto.

TFG se ha desarrollado con la metodología Incremental. Las funcionalidades se han ido desarrollando por grupos de la siguiente forma:

- Planteamiento
- Desarrollo
- Testing

Diagrama de Gantt de TFG:

	May 31 – Jun 06	Jun 07 – Jun 13	Jun 14 – Jun 20	Jun 21 – Jun 27	Jun 28 – Jul 04
Configuración Proyecto	2h				
Configuración API	2h				
Base de Datos	2h				
Fixtures		1h	1h	1h	
User functions		4h 30 min			
Photo functions			4h		
Group functions				4h	
Voters					4h 30 min
Testing		2h	2h	2h	2h
Documentación		1h 30 min	1h 30 min	1h 30 min	1h 30 min

En el planteamiento previo al proyecto se especificó el uso de Docker, Symfony y PostgreSQL, por lo que la primera fase del proyecto fue la creación y configuración de los mismos.

Después, se empezó por orden con los grupos de funcionalidades de la siguiente manera:

- Primero se crean los fixtures relacionados a esa funcionalidad tanto para poder utilizarlos como ejemplos como para comprobar si había algo que modificar respecto a la base de datos
- Segundo, se desarrollan todas las funcionalidades del grupo.
- Tercero y último, se realizan los tests para estas funcionalidades. Si los tests son correctos, se pasa al siguiente grupo de funcionalidades, y en caso contrario, se arreglan los fallos.

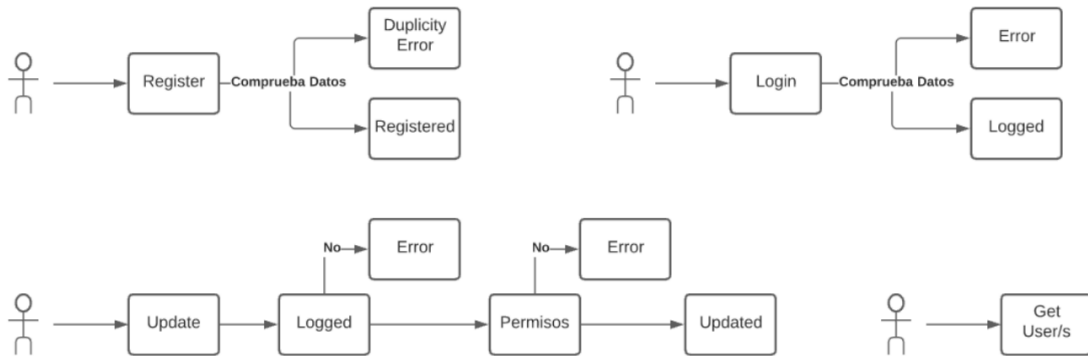
La idea era ir poco a poco sabiendo que cada parte seguía funcionando. Cuando se acababa un grupo de funcionalidades y el testeo era correcto, se comprobaba también el testeo anterior.

Para el testing se ha utilizado Behat, un sistema que permite realizar los tests tanto de uno en uno, como en grupos o todos a la vez, por lo que facilitaba esta idea.

2. Modelado: Análisis y diseño.

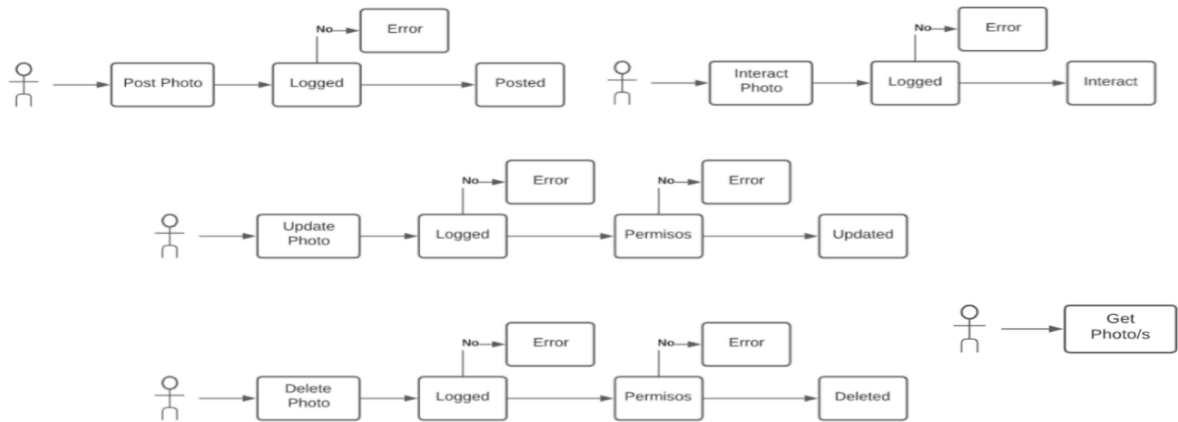
2.1 Diagramas de casos de uso

Casos de Uso para Usuarios



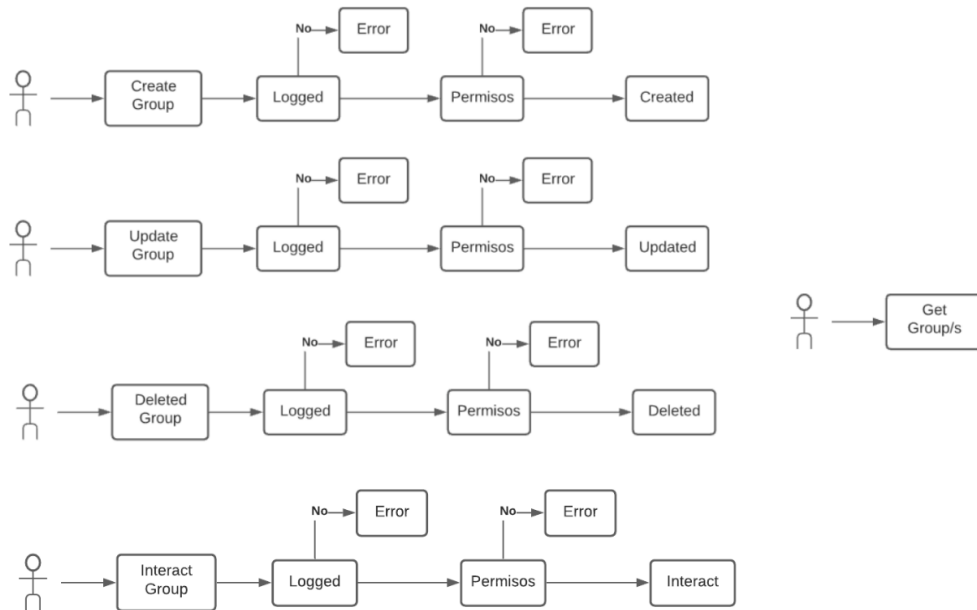
- Register
 - Usuario Anónimo -> Error de duplicidad
 - Usuario Anónimo -> Usuario registrado
- Login
 - Usuario Anónimo -> Error en los datos
 - Usuario Anónimo -> Usuario logueado
- Editar Usuario
 - Usuario Anónimo -> Error
 - Usuario Logueado -> No tiene permisos -> Error
 - Usuario Logueado -> Tiene permisos -> Usuario editado
- Ver Usuarios
 - Usuario Anónimo -> Ve los usuarios
 - Usuario Logueado -> Ve los usuarios

Casos de Uso para Fotos



- Subir Foto
 - Usuario Anónimo -> Error
 - Usuario Logueado -> Foto subida
- Editar/Eliminar Foto
 - Usuario Anónimo -> Error
 - Usuario Logueado -> No tiene permisos -> Error
 - Usuario Logueado -> Tiene permisos -> Foto editada/eliminada
- Interactuar con Foto
 - Usuario Anónimo -> Error
 - Usuario Logueado -> Foto interactuada
- Ver Fotos
 - Usuario Anónimo -> Ve las fotos
 - Usuario Logueado -> Ve las fotos

Casos de Uso para Grupos



- Crear/Editar/Eliminar/Interactuar/Ver Grupo
 - Usuario Anónimo -> Error
 - Usuario Logueado -> No tiene permisos -> Error
 - Usuario Logueado -> Tiene permisos -> Correcto

2.2 Diagramas de ER y Relacional y scripts

Primer Diagrama E/R de TFG:

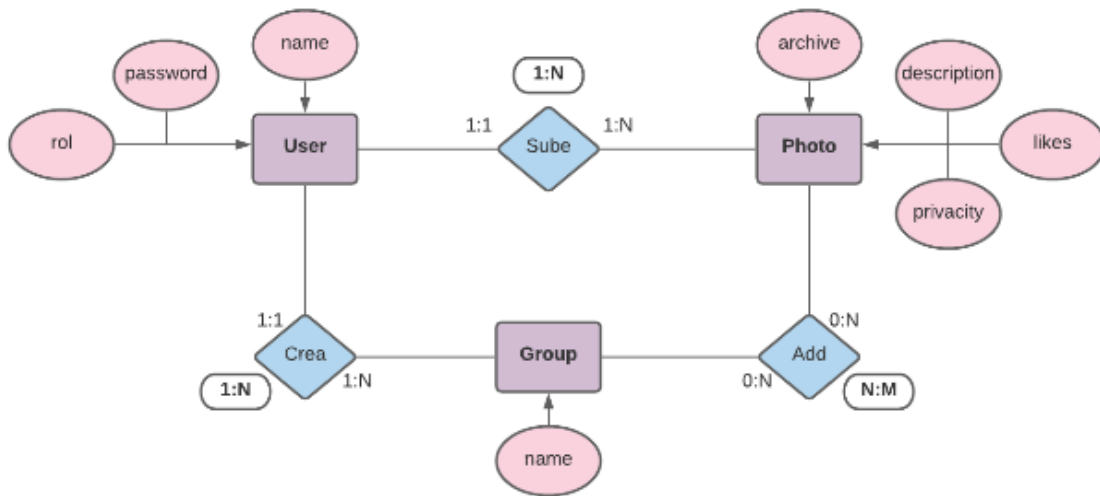
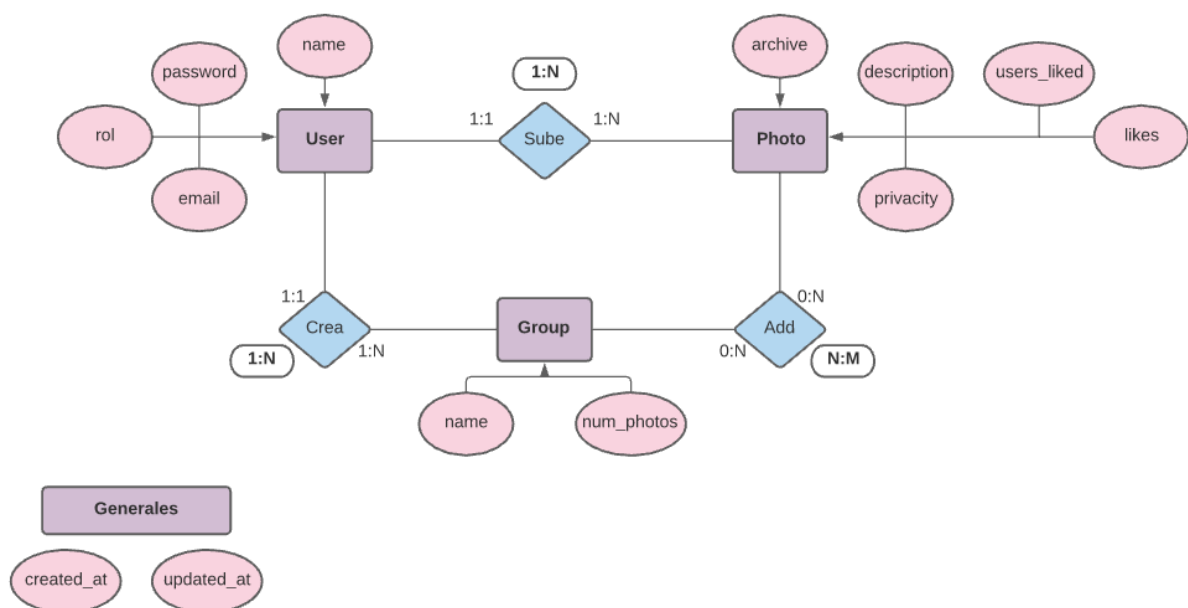


Diagrama Final E/R de TFG:



Scripts tablas finales de TFG:

- Tabla Usuario

```
create table user
(
  id          varchar(255) not null
    primary key,
  name        varchar(255) not null,
  email       varchar(255) not null,
  password    varchar(255) not null,
  roles       longtext     not null comment '(DC2Type:array)',
  created_at  datetime     not null,
  updated_at  datetime     not null,
  constraint UNIQ_8D93D649E7927C74
    unique (email)
)
collate = utf8_unicode_ci;
```

- Tabla Foto

```
create table photo
(
  id          varchar(255) not null
    primary key,
  archive     varchar(255) not null,
  description varchar(255) null,
  likes       int          not null,
  private     tinyint(1)   not null,
  owner_id    varchar(255) null,
  users_liked longtext     null comment '(DC2Type:array)',
  created_at  datetime     not null,
  updated_at  datetime     not null,
  constraint FK_14B784187E3C61F9
    foreign key (owner_id) references user (id)
)
collate = utf8_unicode_ci;

create index IDX_14B784187E3C61F9
  on photo (owner_id);
```

- Tabla Grupo

```
create table `group`
(
    id          varchar(255) not null
        primary key,
    owner_id    varchar(255) null,
    name        varchar(255) not null,
    created_at  datetime      not null,
    updated_at  datetime      not null,
    num_photos  int           not null,
    constraint FK_6DC044C57E3C61F9
        foreign key (owner_id) references user (id)
)
collate = utf8_unicode_ci;

create index IDX_6DC044C57E3C61F9
on `group` (owner_id);
```

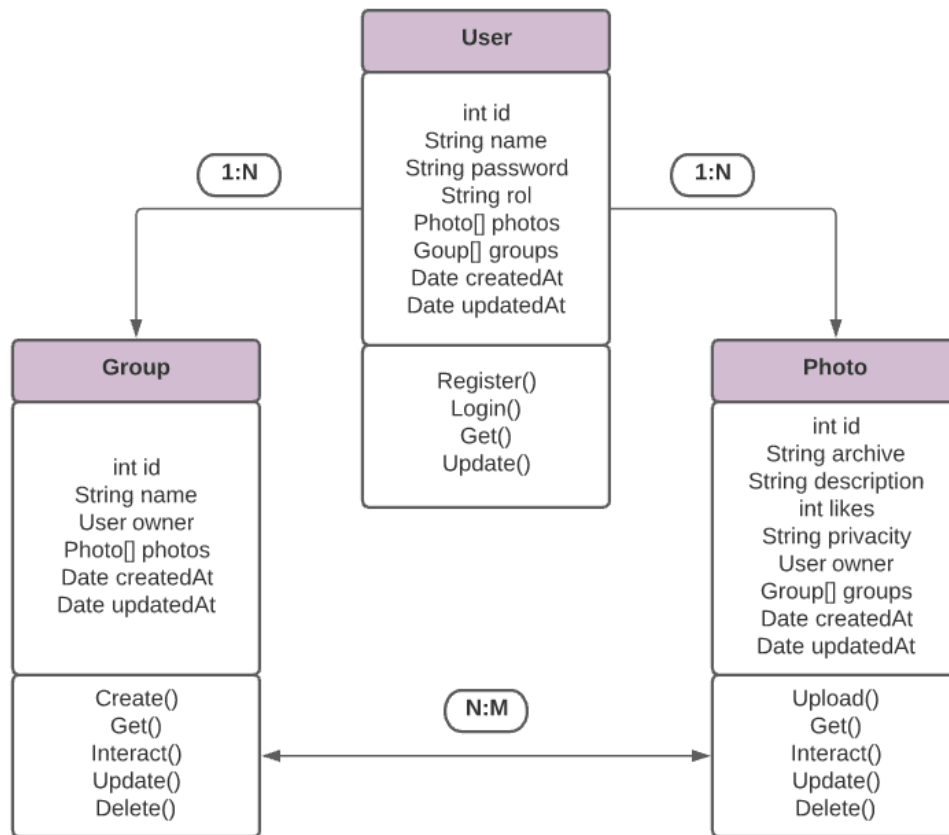
- Tabla Foto-Grupo

```
create table group_photo
(
    group_id varchar(255) not null,
    photo_id varchar(255) not null,
    primary key (group_id, photo_id),
    constraint FK_449BAF7F7E9E4C8C
        foreign key (photo_id) references photo (id)
        on delete cascade,
    constraint FK_449BAF7FFE54D947
        foreign key (group_id) references `group` (id)
        on delete cascade
)
collate = utf8_unicode_ci;

create index IDX_449BAF7F7E9E4C8C
on group_photo (photo_id);

create index IDX_449BAF7FFE54D947
on group_photo (group_id);
```

2.3 Diagrama de Clases



2.4 Diseño de Interfaces

TFG es una API, por lo que no tiene una interfaz al uso. Pese a eso, al utilizar API Platform como un recurso dentro del proyecto, este te permite configurar un Swagger con OAS3 para ver de forma visual la API

User				^
GET	/api/groups/{id}/owner	Retrieves a Group resource.	✓	🔒
GET	/api/photos/{id}/owner	Retrieves a Photo resource.	✓	🔒
GET	/api/users	Retrieves the collection of User resources.	✓	🔒
POST	/api/users/login_check	Login a User	✓	🔒
POST	/api/users/register	Register a new user	✓	🔒
GET	/api/users/{id}	Retrieves a User resource.	✓	🔒
PUT	/api/users/{id}	Replaces the User resource.	✓	🔒
GET	/api/users/{id}/groups	Retrieves a User resource.	✓	🔒
GET	/api/users/{id}/photos	Retrieves a User resource.	✓	🔒

Photo				^
GET	/api/groups/{id}/photos	Retrieves a Group resource.	✓	🔒
GET	/api/photos	Retrieves the collection of Photo resources.	✓	🔒
POST	/api/photos/upload	Upload a new Photo	✓	🔒
GET	/api/photos/{id}	Retrieves a Photo resource.	✓	🔒
PUT	/api/photos/{id}	Replaces the Photo resource.	✓	🔒
DELETE	/api/photos/{id}	Removes the Photo resource.	✓	🔒
POST	/api/photos/{id}/interact	Like/Dislike a Photo	✓	🔒
GET	/api/photos/{id}/owner	Retrieves a Photo resource.	✓	🔒
GET	/api/users/{id}/photos	Retrieves a User resource.	✓	🔒

Group				^
POST	/api/groups	Create a new Group	✓	🔒
GET	/api/groups/{id}	Retrieves a Group resource.	✓	🔒
PUT	/api/groups/{id}	Replaces the Group resource.	✓	🔒
DELETE	/api/groups/{id}	Removes the Group resource.	✓	🔒
GET	/api/groups/{id}/owner	Retrieves a Group resource.	✓	🔒
POST	/api/groups/{id}/photo	Add/Remove a Photo to a Group	✓	🔒
GET	/api/groups/{id}/photos	Retrieves a Group resource.	✓	🔒
GET	/api/users/{id}/groups	Retrieves a User resource.	✓	🔒

3. Implementación, despliegue y pruebas realizadas

3.1 Packages Utilizados

- **Symfony Flex Package**
 - Paquete básico instalado con Symfony
- **Doctrine**
 - Para trabajar con base de datos
- **Hautelook bundle**
 - Para crear fixtures
- **Api Platform**
 - Herramientas para una API
- **Lexik JWT**
 - Sistema de Login con Json Web Token
- **Behat**
 - Para testing

3.2 Ejemplo Testing

El comando lanza todos los tests relacionados con users y después reinicia los fixtures

```

appuser@92fb8d9c47ae:/appdata/www$ vendor/bin/behat -s users
@USER @USER-GET
Feature: GET /users
  In order to get the Users
  I need to check the response

  @USER @USER-GET-ALL
  Scenario: GET All                                     # tests/features/User/GetUse
    When I do a "GET" to "users"                         # App\Tests\Behat\RequestCor
      | http://localhost:250/api/users
    Then I should check the GET All Users Response # App\Tests\Behat\User\Check

  @USER @USER-GET-ID
  Scenario: GET All                                     # tests/t
    When I do a "GET" to "users/113ac47c-73ff-4f51-b4b9-ccae1e3c3d4" # App\Tes
      | http://localhost:250/api/users/113ac47c-73ff-4f51-b4b9-ccae1e3c3d4
    Then I should check the GET User By Id Response # App\Tes

  @USER @USER-LOGIN
  Feature: POST /users/login_check
    In order to login as a User
    I need to check the response

    @USER @USER-LOGIN @LOGIN
    Scenario: Post Login                               # tests/features/User/Login.featu
      When I do a "POST" to "users/login_check" # App\Tests\Behat\RequestContext:
        """
        {
          "username": "user@premium.com",
          "password": "password"
        }
        """
      | http://localhost:250/api/users/login_check
    Then I should get 200 # App\Tests\Behat\RequestContext:
    And I should check the Login Response # App\Tests\Behat\User\CheckUser
  
```

```

@USER @USER-LOGIN @LOGIN-FAILED
Scenario: Post Login # tests/features/User/Login.feature:19
  When I do a "POST" to "users/login_check" # App\Tests\Behat\RequestContext::doRequest()
    """
    {
      "username": "user@premium.com",
      "password": "p"
    }
    """
  | http://localhost:250/api/users/login_check
  Then I should get 401 # App\Tests\Behat\RequestContext::checkCode()

@USER @USER-PUT
Feature: PUT /users/id
  In order to update a User
  I need to check the response

@USER @USER-PUT @PUT
Scenario: Put User
  When I do a "PUT" to "users/113ac47c-73ff-4f51-b4b9-ccae1e3c3d4" as "113ac47c-73ff-4f51-b4b9-ccae1e3c3d4" #
    """
    {
      "name": "EditedUser"
    }
    """
  | http://localhost:250/api/users/113ac47c-73ff-4f51-b4b9-ccae1e3c3d4
  Then I should get 200
  And I should check the PUT User Response

@USER @USER-PUT @PUT
Scenario: Put Another User
  When I do a "PUT" to "users/87c63d28-a358-4f7f-b456-edaa097a2c68" as "113ac47c-73ff-4f51-b4b9-ccae1e3c3d4" #
    """
    {
      "name": "EditedUser"
    }
    """
  | http://localhost:250/api/users/87c63d28-a358-4f7f-b456-edaa097a2c68
  Then I should get 403

@USER @USER-PUT @PUT
Scenario: Put User As Anonymous # tests/features/User/PutUser.feature:28
  When I do a "PUT" to "users/113ac47c-73ff-4f51-b4b9-ccae1e3c3d4" # App\Tests\Behat\RequestContext::doRequest()
    """
    {
      "name": "EditedUser"
    }
    """
  | http://localhost:250/api/users/113ac47c-73ff-4f51-b4b9-ccae1e3c3d4
  Then I should get 401 # App\Tests\Behat\RequestContext::checkCode()

@USER @USER-REGISTER
Feature: POST /users/register
  In order to register a User
  I need to check the response

@USER @USER-REGISTER @REGISTER
Scenario: Post Register # tests/features/User/Register.feature:7
  When I do a "POST" to "users/register" # App\Tests\Behat\RequestContext::doRequest()
    """
    {
      "name": "User",
      "email": "user@email.com",
      "password": "User@01"
    }
    """
  | http://localhost:250/api/users/register
  Then I should get 201 # App\Tests\Behat\RequestContext::checkCode()

```

```

@USER @USER-REGISTER
Feature: POST /users/register
  In order to register a User
  I need to check the response

@USER @USER-REGISTER @REGISTER
Scenario: Post Register # test
  When I do a "POST" to "users/register" # App\T
    """
    {
      "name": "User",
      "email": "user@email.com",
      "password": "User@01"
    }
    """
    | http://localhost:250/api/users/register
  Then I should get 201 # App\T
  And I should check the Register Response # App\T

@USER @USER-REGISTER @REGISTER-DUPLICITY
Scenario: Post Duplicity # tests,
  When I do a "POST" to "users/register" # App\T
    """
    {
      "name": "User",
      "email": "user@email.com",
      "password": "User@01"
    }
    """
    | http://localhost:250/api/users/register
  Then I should get 409 # App\T

```

```

@RESET
Feature: Reset Fixtures
  In order to reset the DataBase
  I should run the commands

@RESET
Scenario: Reset Fixtures # tests/fe
  Then Reset # App\Test

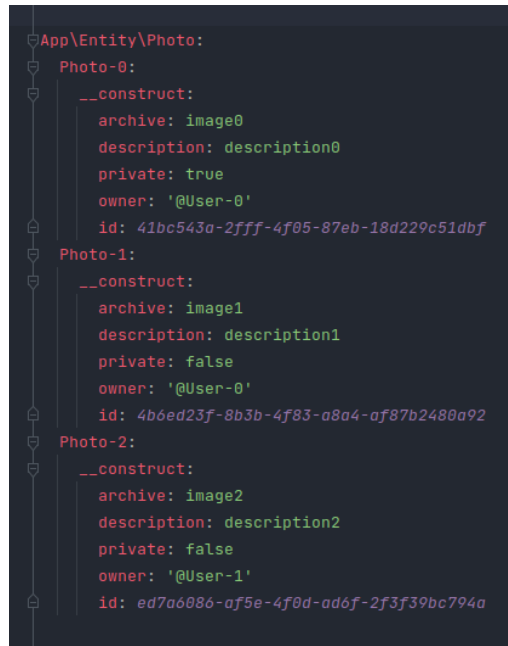
10 scenarios (10 passed)
22 steps (22 passed)
0m2.17s (50.61Mb)
★ appuser@92fb8d9c47ae:/appdata/www$

```

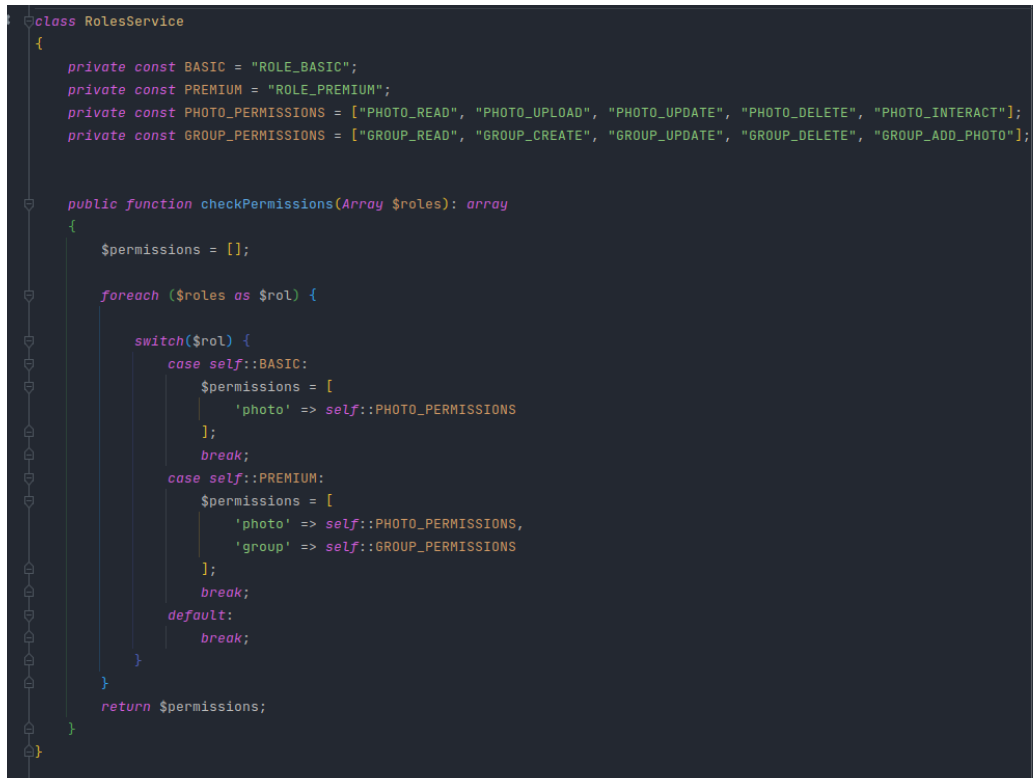
Como se puede ver, todos los tests han pasado correctamente en verde.

Todos los test en conjunto hacen 37 escenarios con 76 pasos, y todos dan resultado correcto.

3.3 Ejemplo Fixtures



3.4 Ejemplo Permisos por Roles



3.5 Ejemplo Implementación de una Llamada

1. Configuración de la operación para API Platform - Swagger.
2. Configuración de la serialización de los campos de la entidad de Usuario.

```

register:
  method: POST
  path: /users/register
  controller: App\Api\Action\User\Register
  denormalization_context:
    groups: [ user_register ]
  openapi_context:
    summary: Register a new user

```

```

App\Entity\User:
  attributes:
    id:
      groups: [ user_read ]
    name:
      groups: [ user_read, user_register, user_update ]
    email:
      groups: [ user_read, user_register ]
    password:
      groups: [ user_register ]
    roles:
      groups: [ user_read ]
    photos:
      groups: [ user_read ]
    groups:
      groups: [ user_read ]
    createdAt:
      groups: [ user_read ]
    updatedAt:
      groups: [ user_read ]

```

Con estos 2 archivos, se indica que solo los campos de “user_register” son los necesarios para la operación.

3. Controlador. Valida y extrae el contenido del Request y ejecuta el servicio.

```

class Register
{
    private UserRegisterService $userRegisterService;

    public function __construct(UserRegisterService $userRegisterService)
    {
        $this->userRegisterService = $userRegisterService;
    }

    public function __invoke(Request $request): User
    {
        $requestArray = $request->toArray();

        try {
            $name = $requestArray['name'];
            $email = $requestArray['email'];
            $password = $requestArray['password'];
        } catch (Exception $exception) {
            throw new BadRequestHttpException( message: 'Wrong Body Format');
        }

        return $this->userRegisterService->create($name, $email, $password);
    }
}

```

4. Servicio. Crea el usuario, encodea la contraseña e intenta guardarlo y devolverlo.

```

class UserRegisterService
{
    private EncoderService $encoderService;
    private EntityManagerInterface $entityManager;

    public function __construct(EntityManagerInterface $entityManager, EncoderService $encoderService)
    {
        $this->encoderService = $encoderService;
        $this->entityManager = $entityManager;
    }

    public function create($name, $email, $password): User
    {
        $user = new User($name, $email);
        $user->setPassword($this->encoderService->generateEncodedPassword($user, $password));

        try {
            $this->entityManager->persist($user);
            $this->entityManager->flush();
        } catch (Exception $exception) {
            throw AlreadyExistsException::fromEmail($email);
        }

        return $user;
    }
}

```

5. Definición Testing

```

@USER @USER-REGISTER
Feature: POST /users/register
    In order to register a User
    I need to check the response

@USER @USER-REGISTER @REGISTER
Scenario: Post Register
    When I do a "POST" to "users/register"
    """
    {
        "name": "User",
        "email": "user@email.com",
        "password": "User@01"
    }
    """
    Then I should get 201
    And I should check the Register Response

@USER @USER-REGISTER @REGISTER-DUPLICITY
Scenario: Post Duplicity
    When I do a "POST" to "users/register"
    """
    {
        "name": "User",
        "email": "user@email.com",
        "password": "User@01"
    }
    """
    Then I should get 409

```

6. Ejecutar Llamada

```

3
4 ▶ POST http://localhost:250/api/users/register
5   Content-Type: application/json
6
7   {
8     "name": "UserEjemplo",
9     "email": "userejemplo@email.com",
10    "password": "User@01"
11  }
12

```

```

X-Content-Type-Options: nosniff
X-Frame-Options: deny
Content-Location: /api/users/e4c2f2a5-758a-4f3a-b4a9-1d20b5cb09da
Location: /api/users/e4c2f2a5-758a-4f3a-b4a9-1d20b5cb09da
Cache-Control: no-cache, private
Date: Thu, 26 Aug 2021 11:05:22 GMT
Link: <http://localhost:250/api/docs.jsonld>; rel="http://www.w3.org/ns/hydra/core#apiDocumentation"
X-Robots-Tag: noindex

{
  "@context": "\/api\/contexts\/User",
  "@id": "\/api\/users\/e4c2f2a5-758a-4f3a-b4a9-1d20b5cb09da",
  "@type": "User",
  "id": "e4c2f2a5-758a-4f3a-b4a9-1d20b5cb09da",
  "name": "UserEjemplo",
  "email": "userejemplo@email.com",
  "roles": [
    "ROLE_BASIC"
  ],
  "photos": [],
  "groups": [],
  "createdAt": "2021-08-26T11:05:22+00:00",
  "updatedAt": "2021-08-26T11:05:22+00:00"
}

Response code: 201 (Created); Time: 735ms; Content length: 334 bytes

```

4. Conclusiones

TFG empezaría como una pequeña aplicación, centrándose en sus puntos fuertes para diferenciarla de la competencia y un pequeño sistema de monetización para su rentabilidad.

Se desarrollaría primero trabajando como autónomo e invirtiendo una cantidad de tiempo y dinero ajustada al equivalente de horas invertidas en el proyecto y más adelante se plantearían más opciones.

De la misma manera, las funcionalidades están pensadas para poder ser fácilmente ampliadas con nuevas ideas, lo que daría una expansión a futuras

5. Bibliografía

Páginas de documentación principal:

- <https://symfony.com>
- <https://www.postgresql.org>
- <https://docs.behat.org/en/latest/>
- <https://api-platform.com/docs/core/>

Extras necesarios:

- <https://github.com/hautelook/AliceBundle#basic-usage>
- <https://github.com/lexik/LexikJWTAuthenticationBundle/blob/2.x/Resources/doc/index.md#getting-started>
- https://behat.org/en/latest/cookbooks/integrating_symfony2_with_behat.html

Páginas concretas de Symfony:

- <https://symfony.com/bundles/SymfonyMakerBundle/current/index.html>
- <https://symfony.com/doc/current/doctrine.html>
- <https://symfony.com/doc/current/security/voters.html>