irontec
Internet y Sistemas sobre GNU/Linux

Rest API, JWT, Repository Pattern con TDD.

Por: Koldo Picaza

@kpikaza

Rest API, JWT, Repository Pattern con TDD.

Aplicación de práctica:

Lo importante en este caso no es desarrollar una lógica extraordinaria, el objetivo reside en comprender las diferentes secciones que hemos analizado durante la parte teórica.

La aplicación consiste en un sistema de registro/login con autenticación JWT vía Rest, siguiendo el patrón repositorio, aplicando el principio de inversión de dependencias y programando dirigidos por tests.

Primero definiremos los user storys.

User stories

1. Como usuario sin autenticar puedo puedo acceder al login.
2. Como usuario sin autenticar puedo registrarme en el site.
3. Como usuario autenticado puedo editar mi perfil.
4. Como usuario autenticado puedo obtener la información de mi perfil.
5. Como usuario autenticado puedo darme de baja.

Al final tendremos un sistema de usuarios independiente del motor de base de datos, en nuestro caso implementaremos el servicio rest, que podría alimentar cuarquier aplicación creada con frameworks de frontend como angular o ember

Una vez definidos los `user stories` , pasamos a diferenciar la distintas parte de la aplicación en su conjunto:

Entidades

- User
  - FOSUserBundle

ROLES

- ROLE_USER
- ROLE_ADMIN
- ROLE_SUPER_ADMIN

Rest API:

API:

- **FOSRest**
  - FOSRestBundle

```
        "jms/serializer-bundle": "^1.1",
        "friendsofsymfony/rest-bundle": "^1.7",
```

- **CORS**
  - NelmioCorsBundle

```
        "nelmio/cors-bundle": "^1.4",
```

Usuarios

- **FOSUser:**
  - FOSOUserBundle

```
        "friendsofsymfony/user-bundle": "^1.3",
```

Autenticación:

- **OAuth2:**
  - FOSOAuthServerBundle

```
        "friendsofsymfony/rest-bundle": "^1.7",
        "friendsofsymfony/oauth-server-bundle": "1.4.*@dev",
```

- **JWT**
  - LexicJWTAuthenticationBundle

```
        "lexik/jwt-authentication-bundle": "^1.3",
```

Documentación y cliente apis:

- **ApiDoc**
  - nelmioApiDocBundle

```
        "nelmio/api-doc-bundle": "^2.11"
```

Parte 1, Bundles contribuidos:

Instalar Symfony 2.8.* standard edition

```
composer create-project symfony/framework-standard-edition practica "2.8.*"
```

podemos comprobar la versión que hemos descargado

```
php app/console --version
Symfony version 2.8.3 - app/dev/debug
```

Iremos instalando y configurando, uno por uno, los módulos que necesitaremos. Para instalar el serializer necesitamos subir la versión de php en el composer.json antes de instalar el nuevo bundle.

```
    // composer.json
        ...
        "config": {
            "bin-dir": "bin",
            "platform": {
                "php": "5.6.18"
            }
        },
        ...
```

Ahora si, podemos instalar el bundle.

```
composer require "jms/serializer-bundle" "^1.1"
```

Lo actvamos en el kernel

```
    // app/AppKernel.php
    class AppKernel extends Kernel
    {

        public function registerBundles()
        {
            $bundles = array(
              ...
              new JMS\SerializerBundle\JMSSerializerBundle(),
```

JmsSerializerBundle no necesita ninguna configuración inicial para funcionar, para más detalle la documentación del bundle es bastante detallada.

El siguiente bundle que instalaremos es el Rest Bundle de Friends of symfony

```
composer require friendsofsymfony/rest-bundle:^1.7
```

Lo activamos en el kernel y pasamos a la configuración

```
    # app/config/config.ymls
    ...
    fos_rest:
        param_fetcher_listener: true
        disable_csrf_role: ROLE_USER
        routing_loader:
            default_format:       json
            include_format:       false
```

Con esta configuración básica nos servirá, para más detalle sobre las configuraciones podemos mira la documentación del bundle.

Para evitar problemas con HTTP access control, CORS(cross-origin HTTP request), los chicos de Nelmio han creado un bundle que nos soluciona perfectamente este problema.

Lo instalamos del modo habitual:

```
composer require nelmio/cors-bundle:^1.4
```

Y pasamos a configurarlo, es conveniente repasar el README del bundle para revisar los parámetros de configuración.

```
    # app/config/config.yml
    nelmio_cors:
        defaults:
            allow_credentials: false
            allow_origin: []
            allow_headers: []
            allow_methods: []
            expose_headers: []
            max_age: 0
            hosts: []
            origin_regex: false
        paths:
            '^/api/':
                allow_origin: ['*']
                allow_headers: ['*']
                allow_methods: ['POST', 'PUT', 'GET', 'DELETE']
                max_age: 3600
```

Vamos con otro bundle de friends of symfony, este es uno de los bundle más utilizados por la comunidad de symfony, sin duda el principal en gestión de usuarios.

```
composer require friendsofsymfony/user-bundle:^1.3
```

Este módulo no nos da todo echo, pero nos facilita mucha lógica en la gestión de usuarios, según el caso puede ser muy útil. De momento dejaremos la configuración básica con Doctrine ORM, pero siguiendo el patrón repositorio, abstraeremos casi completamente nuestra aplicación del motor de base de dato que vayamos a utilizar, el objetivo es no depender de ningún motor de bbdd en particular.

Activamos el bundle en el kernel y vamos a las configuraciones, primero activamos las traducciones

```
# app/config/config.yml
...
framework:
    translator: ~
```

Ahora necesitamos crear la Entidad Usuario, en el primer caso la crearemos en Mysql con Doctrine ORM, el bundle nos permite diferentes tipos como veremos más adelante.

```
<?php

// src/AppBundle/Entity/User.php

namespace AppBundle\Entity;

use FOS\UserBundle\Model\User as BaseUser;
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name = "user")
 */
class User extends BaseUser
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    protected $id;

    /**
     * @var string
     * @ORM\Column(type="string", length=255)
     */
    protected $username;

    /**
     * @var string
     * @ORM\Column(type="string", length=255, unique=true )
     */
    protected $usernameCanonical;

    /**
     * @var string
     * @ORM\Column(type="string", length=255)
     */
    protected $email;

    /**
     * @var string
     * @ORM\Column(type="string", length=255, unique=true )
     */
    protected $emailCanonical;

    /**
     * @var bool
     * @ORM\Column(type="boolean")
     */
    protected $enabled;
```

```php
    /**
     * The salt to use for hashing.
     *
     * @ORM\Column(type="string")
     *
     * @var string
     */
    protected $salt;

    /**
     * Encrypted password. Must be persisted.
     *
     * @ORM\Column(type="string")
     *
     * @var string
     */
    protected $password;

    /**
     * User description.
     *
     * @ORM\Column(type="text", nullable=true)
     *
     * @var string
     */
    protected $description = null;

    /**
     * Plain password. Used for model validation. Must not be persisted.
     *
     * @var string
     */
    protected $plainPassword;

    /**
     * @ORM\Column(type="datetime", nullable=true)
     *
     * @var \DateTime
     */
    protected $lastLogin;

    /**
     * Random string sent to the user email address in order to verify it.
     *
     * @ORM\Column(type="string", nullable=true)
     *
     * @var string
     */
    protected $confirmationToken;

    /**
     * @ORM\Column(type="datetime", nullable=true)
     *
     * @var \DateTime
     */
    protected $passwordRequestedAt;

    /**
     * @ORM\Column(type="boolean")
     *
     * @var bool
     */
    protected $locked = false;

    /**
     * @ORM\Column(type="boolean")
     *
     * @var bool
     */
    protected $expired = false;

    /**
```

```
         * @ORM\Column(type="datetime", nullable=true)
         *
         * @var \DateTime
         */
        protected $expiresAt;

        /**
         * @ORM\Column(type="array")
         *
         * @var array
         */
        protected $roles;

        /**
         * @ORM\Column(type="boolean")
         *
         * @var bool
         */
        protected $credentialsExpired = false;

        /**
         * @ORM\Column(type="datetime", nullable=true)
         *
         * @var \DateTime
         */
        protected $credentialsExpireAt;
```

Como podéis ver hemos llamado a la clase `User` esto es porque a nuestra aplicación no tiene porque importarle de donde vengan los usuarios. Hemos mapeado los campos de la tabla user mediante las anotaciones de doctrine.

Actualizamos nuestro security.yml

```
    # app/config/security.yml
    security:
        encoders:
            FOS\UserBundle\Model\UserInterface: bcrypt

        role_hierarchy:
            ROLE_ADMIN:       ROLE_USER
            ROLE_SUPER_ADMIN: ROLE_ADMIN

        providers:
            fos_userbundle:
                id: fos_user.user_provider.username

        firewalls:
            main:
                pattern: ^/
                form_login:
                    provider: fos_userbundle
                    csrf_provider: security.csrf.token_manager # Use form.csrf_provider instead for Symfo
ny <2.4
                logout:       true
                anonymous:    true

        access_control:
            - { path: ^/login$, role: IS_AUTHENTICATED_ANONYMOUSLY }
            - { path: ^/register, role: IS_AUTHENTICATED_ANONYMOUSLY }
            - { path: ^/resetting, role: IS_AUTHENTICATED_ANONYMOUSLY }
            - { path: ^/admin/, role: ROLE_ADMIN }
```

y en el config seleccionamos el tipo de base de datos, le asignamos la clase usuario, el firewall y activamos la confirmación por email(opcional).

```
    # app/config/config.yml
    fos_user:
        db_driver: orm
        # other valid values are 'mongodb', 'couchdb' and 'propel'
        firewall_name: main
        user_class: AppBundle\Entity\User
        registration:
            confirmation:
                enabled: true
```

En el config_dev haremos que todos los email nos lleguen a nosotros mismos

```
    # app/config/config_dev.yml
    swiftmailer:
        delivery_address: 'tu@email.com'
```

Importamos las rutas

```
    # app/config/routing.yml
    fos_user_security:
        resource: "@FOSUserBundle/Resources/config/routing/security.xml"

    fos_user_profile:
        resource: "@FOSUserBundle/Resources/config/routing/profile.xml"
        prefix: /profile

    fos_user_register:
        resource: "@FOSUserBundle/Resources/config/routing/registration.xml"
        prefix: /register

    fos_user_resetting:
        resource: "@FOSUserBundle/Resources/config/routing/resetting.xml"
        prefix: /resetting

    fos_user_change_password:
        resource: "@FOSUserBundle/Resources/config/routing/change_password.xml"
        prefix: /profile
```

En este caso necesitaremos una bbdd para persistir nuestras tablas.

Primero generamos los `getters` y `setters` de nuestras entidades.

```
php app/console doctrine:generate:entities AppBundle
```

creamos la bbdd

```
php app/console doctrine:database:create
```

Y generamos el schema

```
php app/console doctrine:schema:create
```

Ahora generamos un usuario administrador con los comandos que nos provee fos user bundle

```
php app/console fos:user:create admin admin@admin.mail --super-admin ROLE_SUPER_ADMIN
php app/console fos:user:change-password admin Demo1234
```

podemos comprobar las rutas que nos ha generado

```
php app/console debug:router | grep fos_user
fos_user_security_login             ANY        ANY      ANY    /login
fos_user_security_check             POST       ANY      ANY    /login_check
fos_user_security_logout            ANY        ANY      ANY    /logout
fos_user_profile_show               GET        ANY      ANY    /profile/
fos_user_profile_edit               ANY        ANY      ANY    /profile/edit
fos_user_registration_register      ANY        ANY      ANY    /register/
fos_user_registration_check_email   GET        ANY      ANY    /register/check-email
fos_user_registration_confirm       GET        ANY      ANY    /register/confirm/{token}
fos_user_registration_confirmed     GET        ANY      ANY    /register/confirmed
fos_user_resetting_request          GET        ANY      ANY    /resetting/request
fos_user_resetting_send_email       POST       ANY      ANY    /resetting/send-email
fos_user_resetting_check_email      GET        ANY      ANY    /resetting/check-email
fos_user_resetting_reset            GET|POST   ANY      ANY    /resetting/reset/{token}
fos_user_change_password            GET|POST   ANY      ANY    /profile/change-password
```

Ahora descargaremos el bundle para implementar la autorización JWT

```
composer require lexik/jwt-authentication-bundle:^1.3
```

Lo activamos en el kernel y generamos las claves ssh

```
mkdir -p app/var/jwt
openssl genrsa -out app/var/jwt/private.pem -aes256 4096
openssl rsa -pubout -in app/var/jwt/private.pem -out app/var/jwt/public.pem
```

También las de test

```
openssl genrsa -out app/var/jwt/private-test.pem -aes256 4096
openssl rsa -pubout -in app/var/jwt/private-test.pem -out app/var/jwt/public-test.pem
```

Ahora deberíamos añadir las rutas al par de claves, como variables de entorno. Si estamos utilizando el server que monta symfony con el comando server:run los meteremos en el `parameters.yml`

Primero definimos los parámetros necesarios en el `parameters.yml`

```
    # app/config/parameters.yml.dist
    jwt_private_key_path: %kernel.root_dir%/var/jwt/private.pem   # ssh private key path
    jwt_public_key_path:  %kernel.root_dir%/var/jwt/public.pem    # ssh public key path
    jwt_key_pass_phrase:  ''                                      # ssh key pass phrase
    jwt_token_ttl:        86400
```

```
    # app/config/parameters.yml
    jwt_private_key_path: '%kernel.root_dir%/var/jwt/private.pem'
    jwt_public_key_path: '%kernel.root_dir%/var/jwt/public.pem'
    jwt_key_pass_phrase: demo
    jwt_token_ttl: ~
```

Ponemos los parámetros definidos en sus respectivas configuraciones, primero en test

```
    # app/config/config_test.yml
    lexik_jwt_authentication:
        private_key_path:   %kernel.root_dir%/var/jwt/private-test.pem
        public_key_path:    %kernel.root_dir%/var/jwt/public-test.pem
```

Despues en el config general.

```
# app/config/config.yml
lexik_jwt_authentication:
    private_key_path: %jwt_private_key_path%
    public_key_path:  %jwt_public_key_path%
    pass_phrase:      %jwt_key_pass_phrase%
    token_ttl:        %jwt_token_ttl%
```

Pasamos a actualizar el `security.yml`

```
# app/config/security.yml
...
    firewalls:
        login:
            pattern:  ^/api/login
            stateless: true
            anonymous: true
            form_login:
                check_path:               /api/login_check
                success_handler:          lexik_jwt_authentication.handler.authentication_success
                failure_handler:          lexik_jwt_authentication.handler.authentication_failure
                require_previous_session: false

        api:
            pattern:   ^/api
            stateless: true
            lexik_jwt: ~
        ...

    access_control:
        ...
        - { path: ^/api/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/api, roles: IS_AUTHENTICATED_FULLY }
```

Importamos las rutas en el `routing.yml`

```
# app/config/routing.yml
api_login_check:
    path: /api/login_check
```

Actualizamos el `parameters.yml`

```
# app/config/parameters.yml.dist
jwt_private_key_path: %kernel.root_dir%%private.key.pair%   # ssh private key path
jwt_public_key_path:  %kernel.root_dir%%public.key.pair%    # ssh public key path
```

Comprobamos que todo funciona, primero generamos el token JWT

```
curl -X POST http://127.0.0.1:8000/api/login_check -d _username=admin -d _password=Demo1234
{"token":"eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXUyJ9.eyJleHAiOjE0NTc0NzQwMTAsInVzZXJuYW1lIj....."}
```

Nos tiene que devolver el token JWT.

Nos falta nelmio api doc bundle para dejar preparado todo el stack de nuestra aplicación, después de instalarlo, empezaremos a aplicar tdd para implementar el el patrón repositorio y abstraer completamente el motor de bbdd de nuestra aplicación.

```
composer require nelmio/api-doc-bundle:^2.11
```

Lo activamos en el kernel y vamos a las configs

```
# app/config/config.yml
nelmio_api_doc:
    name: 'Educaedu práctica API docs'
#    exclude_sections: ["Some section"]
    default_sections_opened:  true
#Api Docs template
#    motd:
#        template: some/template.html.twig
    sandbox:
#        enabled: false
```

Importamos en el `routing.yml`

```
# app/config/routing.yml
NelmioApiDocBundle:
    resource: "@NelmioApiDocBundle/Resources/config/routing.yml"
    prefix:   /api/doc
```

Añadimos un nuevo firewall para la documentación, justo antes de `api`, para evitar colisiones de firewall

```
# app/config/security.yml
    ...
    firewalls:
        ...
        docs:
            pattern:  ^/api/doc
            stateless: true
            anonymous: true
        ...
    access_control:
        ...
        - { path: ^/api/doc, roles: IS_AUTHENTICATED_ANONYMOUSLY }
        ...
```

Y accedemos a nuestra recién creada documentación en 127.0.0.1:8000/app_dev.php/api/doc. Ya tenemos todo lo necesario, ahora podemos empezar a desarrollar nuestra aplicación y hacer el primer commit.

Parte 2, TDD 1:

Empezaremos con el user story 4 `Como usuario autenticado puedo obtener la información de mi perfil.`. No es el más sencillo, pero será un buen punto de partida. Creamos el archivo `UserControllerTest`, y creamos nuestro primer test con PHPUnit. Para la acción get de nuestro servicio rest.

```php
<?php
// src/AppBundle/Tests/Controller/UserControllerTest.php

namespace AppBundle\Tests\Controller;

use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

class UserControllerTest extends WebTestCase
{
    const NAME = 'meco';
    const PASS = 'Demo1234';
    const ROUTE = '/api/users/%s';

    /**
     * Create a client with a default Authorization header.
     *
     * @param string $username
     * @param string $password
     * @see https://github.com/lexik/LexikJWTAuthenticationBundle/blob/master/Resources/doc/3-functio
nal-testing.md
     *
     * @return \Symfony\Bundle\FrameworkBundle\Client
     */
    protected function createAuthenticatedClient($username = 'user', $password = 'password')
    {
        $client = static::createClient();
        $client->request(
            'POST', '/api/login_check', array(
          '_username' => $username,
          '_password' => $password,
            )
        );

        $data = json_decode($client->getResponse()->getContent(), true);

        if (array_key_exists('token', $data)) {
            $client = static::createClient();
            $client->setServerParameter('HTTP_Authorization', sprintf('Bearer %s', $data['token']));
        }

        return $client;
    }

    protected function getLast($client)
    {
        $em = $client->getContainer()->get('doctrine')->getManager();
        $user = $em->getRepository('AppBundle:User')->findOneByUsername('meco');

        return $user->getId();
    }

    public function testValidGetUser()
    {
        $client = $this->createAuthenticatedClient(self::NAME, self::PASS);

        $id = $this->getLast($client);

        $client->request('GET', sprintf(self::ROUTE, $id));

        $this->assertEquals(200, $client->getResponse()->getStatusCode());
    }
}
```

Si corremos el test, es obvio, que fallará.

```
phpunit -c app/
PHPUnit 4.8.23 by Sebastian Bergmann and contributors.

.F

Time: 666 ms, Memory: 35.25Mb

There was 1 failure:

1) AppBundle\Tests\Controller\UserControllerTest::testValidGetMe
Failed asserting that 404 matches expected 200.

/educaedu/practica-final/src/AppBundle/Tests/Controller/UserControllerTest.php:49

FAILURES!
Tests: 2, Assertions: 3, Failures: 1.
```

Esto está muy bien, la información de este test, es nuestro siguiente paso a seguir, necesitamos una ruta y un controlador. Primero crearemos nuestro controlador dentro del AppBundle.

```
    <?php
    // src/AppBundle/Controller/UserController

    namespace AppBundle\Controller;

    use FOS\RestBundle\Controller\FOSRestController;
    use Nelmio\ApiDocBundle\Annotation\ApiDoc;
    use Sensio\Bundle\FrameworkExtraBundle\Configuration\Security;
    use Symfony\Component\Routing\Annotation\Route;
    use Symfony\Component\HttpFoundation\Request;

    /**
     * UserController.
     */
    class UserController extends FOSRestController
    {
        /**
         * @Security("is_granted('view', user)")
         * @ApiDoc(
         *   description = "Get your own user.",
         *   statusCodes = {
         *     200 = "Show user info.",
         *     401 = "Authentication failure, user doesn't have permission or API token is invalid or out
dated.",
         *     403 = "Authorizationi failure, user doesn't have permission to access this area.",
         *   }
         * )
         *
         * @return json|xml
         */
        public function getUserAction(Request $request)
        {
            $user = $this->get('security.token_storage')->getToken()->getUser();

            $view = $this->view($user);

            return $this->handleView($view);
        }

    }
```

Damos de alta la ruta en el `routin.yml`

```
    # app/config/routing.yml
    ...
    app_user:
        type: rest
        prefix: /api
        resource: AppBundle\Controller\UserController
    ...
```

Por ultimo creamos el usuario de test.

```
php app/console fos:user:create
Please choose a username:meco
Please choose an email:meco@meco.mail
Please choose a password:Demo1234
Created user meco

php app/console fos:user:promote meco VIEW
```

Ahora corremos los tests. Seguimos en rojo, no tenemos permiso para acceder a esa url, está protegida por JWT token, además hemos puesto la anotación `Security`. Si nos fijamos en el error, es un 403, como hemos visto en el curso es un fallo de autorización, esto es porque nuestro user tiene ambos roles `ROLE_USER` y `VIEW` pero todavía no hemos implementado ninguna política de seguridad.

Para esto crearemos una clase `Voter`, dentro de el directorio `Security`

```php
<?php
// src/AppBundle/Security/UserVoter.php

namespace AppBundle\Security;

use Symfony\Component\Security\Core\Authorization\Voter\Voter;
use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
use AppBundle\Entity\User;

/**
 * UserVoter.
 */
class UserVoter extends Voter
{
    const VIEW = 'view';

    public function supports($attribute, $subject)
    {
        return $subject instanceof User && in_array($attribute, array(
              self::VIEW
        ));
    }

    protected function voteOnAttribute($attribute, $currentUser, TokenInterface $token)
    {
        $user = $token->getUser();

        if (!$user instanceof User) {
            return false;
        }

        $roles = $user->getRoles();

        if (
            in_array('ROLE_USER', $roles) &&
            $attribute == self::VIEW &&
            // User only can view own info.
            $user->getEmail() === $currentUser->getEmail()
        ) {
            return true;
        }

        return false;
    }
}
```

Como hemos explicado antes, los voter son la forma adecuada de permitir accesos a deferentes partes de la aplicación, todavía nos falta dar de alta el voter en el `services.yml`.

```yaml
# app/config/services.yml
services:
    ...
    security.access.user_voter:
        class:      AppBundle\Security\UserVoter
        public:     false
        tags:
           - { name: security.voter }
```

Ahora si pasamos los tests, están en verde, es un buen momento para commitear, pero no es más que un falso positivo. Como podemos ver, nuestro controlador está directamente acoplado con el motor de Base de datos, en este caso MySQL.

Parte 3, Repository pattern 1 y TDD 2:

Como hemos visto durante el curso uno de los objetivos principales es desacoplar lo máximo posible nuestro código, para hacerlo re-utilizable. Para lograr esto seguiremos los patrones de diseño que vimos antes, en particular del patrón repositorio. Necesitamos abstraer la capa de base de datos del resto de la aplicación, empezaremos creando un test para nuestro repositorio

```php
<?php
// src/Tests/Model/UserRepositoryTest.php

namespace AppBundle\Tests\Model;

use AppBundle\Entity\User;
use AppBundle\Entity\UserGateway;
use AppBundle\Model\UserFactory;
use AppBundle\Model\UserRepository;
use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

class UserRepositoryTest extends WebTestCase
{
    const USER = 'koldo';
    const EMAIL = 'koldo@koldo.mail';
    const PASS = 'Demo1234';
    const DESC = 'Hola mondo';

    /**
     * @var UserGateway
     */
    private $gateway;

    /**
     * @var UserRepository
     */
    private $repository;

    /**
     * Set up UserRepository.
     */
    public function setUp()
    {
        parent::setUp();
        $gatewayClassname = 'AppBundle\Entity\UserGateway';
        $this->gateway = $this->prophesize($gatewayClassname);
        $this->factory = new UserFactory();
        $this->repository = new UserRepository($this->gateway->reveal(), $this->factory);
    }
}
```

Si nos fijamos en el método `setUp()`, para crear el repositorio, necesitamos tres clases, La factoría o `Factory`, la puerta de enlace o `Gateway` y el repositorio o `Repository`. vamos a añadir el test al final de la clase.

```php
<?php
// src/Tests/Model/UserRepositoryTest.php
    ...
    public function testFindOneByWithParams()
    {
        $fakeUser = new User();
        $fakeUser = $fakeUser->fromArray(array(
            'username' => self::USER, 'email' => self::EMAIL, 'password' => self::PASS, 'description'
=> self::DESC
        ));
        $this->gateway->findOneBy(array('username' => self::USER), array())->willReturn($fakeUser);
        $fakeUser = $this->factory->makeOne($fakeUser);
        $user = $this->repository->findOneBy(array('username' => self::USER));
        $this->assertTrue($user instanceof User);
        $this->assertEquals($user->getUsername(), $fakeUser->getUsername());
        $this->assertEquals($user->getEmail(), $fakeUser->getEmail());
        $this->assertEquals($user->getDescription(), $fakeUser->getDescription());
        $this->assertEquals($user->getUsername(), $user->__toString());
    }
}
```

Pasamos los test, y nos encontramos la primera excepción, la clase `UserFactory` no existe, vamos a crearla. Bueno, paremos un segundo y pensemos, Si queremos seguir el principio de inversión de dependencias necesitamos abstraer nuestras clases, para ello haremos uso de `Interfaces` o contratos que definirán la estructuras que deben recibir las clases de nivel superior, en nuestro caso el

controlador. Creamos nuestro `UserFactoryInterface` en el directorio `Model`.

```php
<?php
// src/AppBundle/Model/UserFactoryInterface.php

namespace AppBundle\Model;
/**
 * UserFactoryInterface.
 */
interface UserFactoryInterface
{
    /**
     * @param \AppBundle\Model\UserInterface $rawUser
     *
     * @return \AppBundle\Model\UserInterface
     */
    public function makeOne(UserInterface $rawUser);
    /**
     * @param array $rawUsers
     *
     * @return array
     */
    public function makeAll(array $rawUsers);
    /**
     * @param \AppBundle\Model\UserInterface $rawUser
     *
     * @return \AppBundle\Model\UserInterface
     */
    public function make(UserInterface $rawUser);
}
```

Creamos también `GatewayInterface` y `UserInterface`, queremos una abstracción total del motor de base de datos y sabemos que las clases `User` y `UserGateway`, dependen directamente del motor de base de datos.

```php
<?php
// src/AppBundle/Model/UserGatewayInterface.php

namespace AppBundle\Model;
/**
 * UserGateway.
 */
interface UserGatewayInterface
{

}
```

Creamos el `Interface` del el objeto user de `FosUserBundle`, en este caso lo utilizaremos como está, pero quien sabe si el día de mañana tenemos que cambiar de framework?

```php
<?php

// src/AppBundle/Model/User.php

namespace AppBundle\Model;

use FOS\UserBundle\Model\GroupInterface;

/**
 * User.
 */
interface UserInterface
{
    /**
     * @param array array().
     */
    public static function fromArray(array $array = array());

    public function __construct();
```

```php
    public function addRole($role);

    /**
     * Serializes the user.
     *
     * The serialized data have to contain the fields used by the equals method and the username.
     *
     * @return string
     */
    public function serialize();

    /**
     * Unserializes the user.
     *
     * @param string $serialized
     */
    public function unserialize($serialized);

    /**
     * Removes sensitive data from the user.
     */
    public function eraseCredentials();

    /**
     * Returns the user unique id.
     *
     * @return mixed
     */
    public function getId();

    public function getUsername();

    public function getUsernameCanonical();

    public function getSalt();

    public function getDescription();

    public function getEmail();

    public function getEmailCanonical();

    /**
     * Gets the encrypted password.
     *
     * @return string
     */
    public function getPassword();

    public function getPlainPassword();

    /**
     * Gets the last login time.
     *
     * @return \DateTime
     */
    public function getLastLogin();

    public function getConfirmationToken();

    /**
     * Returns the user roles.
     *
     * @return array The roles
     */
    public function getRoles();

    /**
     * Never use this to check if this user has access to anything!
     *
     * Use the SecurityContext, or an implementation of AccessDecisionManager
```

```
     * instead, e.g.
     *
     *         $securityContext->isGranted('ROLE_USER');
     *
     * @param string $role
     *
     * @return bool
     */
    public function hasRole($role);

    public function isAccountNonExpired();

    public function isAccountNonLocked();

    public function isCredentialsNonExpired();

    public function isCredentialsExpired();

    public function isEnabled();

    public function isExpired();

    public function isLocked();

    public function isSuperAdmin();

    public function isUser();

    public function removeRole($role);

    public function setUsername($username);

    public function setUsernameCanonical($usernameCanonical);

    /**
     * @param \DateTime $date
     *
     * @return User
     */
    public function setCredentialsExpireAt(\DateTime $date);

    /**
     * @param bool $boolean
     *
     * @return User
     */
    public function setCredentialsExpired($boolean);

    public function setDescription($description);

    public function setEmail($email);

    public function setEmailCanonical($emailCanonical);

    public function setEnabled($boolean);

    /**
     * Sets this user to expired.
     *
     * @param bool $boolean
     *
     * @return User
     */
    public function setExpired($boolean);

    /**
     * @param \DateTime $date
     *
     * @return User
     */
    public function setExpiresAt(\DateTime $date);
```

```
        public function setPassword($password);

        public function setSuperAdmin($boolean);

        public function setPlainPassword($password);

        public function setLastLogin(\DateTime $time);

        public function setLocked($boolean);

        public function setConfirmationToken($confirmationToken);

        public function setPasswordRequestedAt(\DateTime $date = null);

        /**
         * Gets the timestamp that the user requested a password reset.
         *
         * @return null|\DateTime
         */
        public function getPasswordRequestedAt();

        public function isPasswordRequestNonExpired($ttl);

        public function setRoles(array $roles);

        /**
         * Gets the groups granted to the user.
         *
         * @return Collection
         */
        public function getGroups();

        public function getGroupNames();

        public function hasGroup($name);

        public function addGroup(GroupInterface $group);

        public function removeGroup(GroupInterface $group);

        public function __toString();
    }
```

Modificamos la clase `User` para que implemente `UserInterface`

```
    <?php

    // src/AppBundle/Entity/User.php

    namespace AppBundle\Entity;

    use AppBundle\Model\UserInterface;
    use FOS\UserBundle\Model\User as BaseUser;
    use Doctrine\ORM\Mapping as ORM;

    /**
     * @ORM\Entity
     * @ORM\Table(name = "user")
     * @ORM\Entity(repositoryClass="AppBundle\Entity\UserGateway")
     */
    class User extends BaseUser implements UserInterface
    {
        ...
```

creamos sus respectivas implementaciones, `UserGateway` la situaremos junto con `User` dentro de la carpeta `Entity`

```php
<?php
// src/AppBundle/Entity/UserGateway

namespace AppBundle\Entity;
use AppBundle\Model\UserInterface;
use AppBundle\Model\UserGatewayInterface;
use Doctrine\ORM\EntityRepository;
/**
 * UserGateway.
 */
class UserGateway extends EntityRepository implements UserGatewayInterface
{

}
```

Y por último la clase `Factory` dentro del directorio `Model`

```php
<?php
// src/AppBundle/Model/UserFactory

namespace AppBundle\Model;

use AppBundle\Model\UserInterface;
use AppBundle\Model\UserFactoryInterface;
/**
 * UserFactory implements UserFactoryInterface.
 */
class UserFactory implements UserFactoryInterface
{
    /**
     * @param \AppBundle\Entity\User $rawUser
     *
     * @return \AppBundle\Entity\User
     */
    public function makeOne(UserInterface $rawUser)
    {
        return $this->make($rawUser);
    }
    /**
     * @param \AppBundle\Entity\User $rawUser
     *
     * @return \AppBundle\Entity\User
     */
    public function make(UserInterface $rawUser)
    {
        // You can format object, in this case we left it to return as raw object, feedback is welcom
e!
        return $rawUser;
    }
}
```

ya os habéis fijado que el user gateway y su interface están vacíos, es porque de momento utilizaremos los métodos de doctrine.

Ahora necesitamos implementar la configuración para atar todas estas piezas, Symfony nos ofrece la inyección de dependencias como solución, veamos como se hace, abrimos el archivo `services.yml`

```
    services:
        ...
        app.user_factory:
            class: AppBundle\Entity\UserFactory

        app.user_gateway:
            class: AppBundle\Entity\UserGateway
            factory: [ "@doctrine", getRepository]
            arguments: [ "AppBundle:User" ]

        app.user_repository:
            class: AppBundle\Entity\UserRepository
            arguments: [ "@app.user_gateway", "@app.user_factory" ]
```

Y para terminar de implementar el patrón repositorio solo nos falta la clase `UserRepository`

```php
<?php
namespace AppBundle\Model;
use AppBundle\Model\UserGatewayInterface;
use AppBundle\Model\UserFactoryInterface;
/**
 * UserRepository.
 */
class UserRepository
{
    /**
     * @var \AppBundle\Model\UserGatewayInterface
     */
    private $gateway;
    /**
     * @var \AppBundle\Model\UserFactoryInterface
     */
    private $factory;
    /**
     * @param \AppBundle\Model\UserGatewayInterface $gateway
     * @param \AppBundle\Model\UserFactoryInterface $factory
     */
    public function __construct(UserGatewayInterface $gateway, UserFactoryInterface $factory)
    {
        $this->gateway = $gateway;
        $this->factory = $factory;
    }
    /**
     * @param User|int $id
     *
     * @return User
     */
    public function find($id)
    {
        return $this->gateway->find($id);
    }
    /**
     * @param array $criteria
     * @param array $orderBy
     *
     * @return User
     *
     * @throws NotFoundHttpException
     */
    public function findOneBy(array $criteria, array $orderBy = array())
    {
        $user = $this->gateway->findOneBy($criteria, $orderBy);
        if (null === $user) {
            return null;
        }
        return $this->factory->makeOne($user);
    }
    /**
     * @param $id
     *
     * @return User
     */
    public function parse($id)
    {
        $rawUser = $this->gateway->find($id);

        return $this->factory->makeOne($rawUser);
    }
}
```

Pasamos los tests, y tenemos que ver que está todo correcto, ahora podemos commitear. El siguiente paso es unir el controlador con el modelo, lo haremos de la siguiente manera

Actualizamos `UserController` para que utilice nuestro `Repository` y así lo desacoplamos del motor de base de datos

```
// src/AppBundle/Controller/UserController.php
    ...
    public function getUserAction($id)
    {
        $user = $this->get('app.user_repository')->find($id);
        $view = $this->view($user);

        return $this->handleView($view);
    }
```

Si volvemos a pasar los test todo debe seguir funcionando correctamente y podemos volver a commitear nuestro trabajo.

Parte 4, 2º user Story

Las verdad que al realizar el primer user story, nos hemos dejado todo bastante bien organizado, para que sea mas sencillo continuar con los siguientes, igualmente, todavía faltan piezas importantes del puzzle.

Comenzaremos implementando el user story 2 `Como usuario sin autenticar puedo registrarme en el site`, este user story, a parte de lo que ya tenemos creado necesita un formulario y validación. empecemos por los tests. primero crearemos el método post, justo después del método `createAuthenticatedClient` que nos ayudará a realizar las peticiones en los diferentes tests.

```
<?php
// src/AppBundle/Tests/Controller/UserControllerTest.php

class UserControllerTest extends WebTestCase
{
    ...
    protected function post($uri, array $data, $auth = false)
    {
        $client = $this->getClient($auth);
        $client->request('POST', $uri, $data);
        return $client->getResponse();
    }
```

Y añadimos nuestro primer test para el registro, testeara el envío del formulario vacío, nos tiene que devolver un error 400.

```
<?php
// src/AppBundle/Tests/Controller/UserControllerTest.php

class UserControllerTest extends WebTestCase
{
    const REGISTER_ROUTE = '/api/users';
    ...
    public function testRegistrationFailedWithEmptyForm()
    {
        $client = static::createClient();
        $client->request('POST', self::REGISTER_ROUTE);
        $this->assertEquals(400, $client->getResponse()->getStatusCode());
    }
```

Pasamos los test, y obtenemos un 404 en vez del 400 que esperamos, al igual que en nuestro primer user story, necesitaremos una ruta a un controlador, y en este caso además necesitaremos un formulario, vamos con el controlador

```php
<?php
// src/AppBundle/Controller/UserController.php

/**
 * @ApiDoc(
 *   description = "Register new user.",
 *   statusCodes = {
 *     200 = "User correctly added.",
 *     401 = "Authentication failure, user doesn’t have permission or API token is invalid or outdate
d.",
 *   }
 * )
 *
 * @param Request $request
 *
 * @return array
 */
public function postUserAction(Request $request)
{
    $user = $request->request->all();

    $view = $this->view($user);
    return $this->handleView($view);
}
```

Lo bueno de Fos Rest es que se encarga de la pluralización de las URIs de nuestros recursos, también podemos ver que no estamos haciendo uso de formularios, esto será lo siguiente que hagamos. Para crear formularios y validarlos crearemos dos nuevas clases. antes de ello le diremos a nuestro API firewall, que la ruta que acabamos de crear será accesible sin necesidad de autenticación.

```yaml
# app/config/security.yml
security:
    ...
    firewalls:
        ...
        register:
            pattern:  ^/api/users
            methods: [POST]
            anonymous: true

    ...
    access_control:
        ...
        - { path: ^/api/users, roles: IS_AUTHENTICATED_ANONYMOUSLY, methods: [POST] }
        - { path: ^/api, roles: IS_AUTHENTICATED_FULLY }
```

Ahora si pasamos a crear nuestro formulario, para la validación crearemos un simple modelo de formulario, este a su vez nos ayudará a documentar el api, veamos como

```php
<?php
namespace AppBundle\Form\Model;
use Symfony\Component\Validator\Constraints as Assert;
/**
 * RegistrationFormModel.
 */
class RegistrationFormModel
{
    const NAME = 'username';
    const MAIL = 'email';
    const PLAIN = 'plainPassword';
    const PASS = 'password';
    /**
     * @Assert\NotBlank()
     * @Assert\Regex("/[a-zA-Z0-9]/")
     *
     * @var string
     */
    protected $username;
    /**
```

```
         * @Assert\NotBlank()
         * @Assert\Email()
         *
         * @var string
         */
        protected $email;
        /**
         * @Assert\NotBlank()
         *
         * @var string
         */
        protected $plainPassword;
        /**
         * @Assert\NotBlank()
         *
         * @var string
         */
        protected $password;
        public function __construct($username = null, $email = null, $plainPassword = null, $password = n
ull)
        {
            $this->username = $username;
            $this->email = $email;
            $this->plainPassword = $plainPassword;
            $this->password = $password;
        }
        /**
         * @param array $user
         *
         * @return \self
         */
        public static function fromArray(array $user = array(
            self::NAME => null, self::MAIL => null, self::PLAIN => null, self::PASS => null
        ))
        {
            return new self(
                array_key_exists(self::NAME, $user) ? $user[self::NAME] : null,
                array_key_exists(self::MAIL, $user) ? $user[self::MAIL] : null,
                array_key_exists(self::PLAIN, $user) ? $user[self::PLAIN] : null,
                array_key_exists(self::PASS, $user) ? $user[self::PASS] : null
            );
        }
        public function setUsername($username)
        {
            $this->username = $username;
        }
        public function getUsername()
        {
            return $this->username;
        }
        public function setEmail($email)
        {
            $this->email = $email;
        }
        public function getEmail()
        {
            return $this->email;
        }
        public function setPlainPassword($plainPassword)
        {
            $this->plainPassword = $plainPassword;
        }
        public function getPlainPassword()
        {
            return $this->plainPassword;
        }
        public function setPassword($password)
        {
            $this->password = $password;
        }
        public function getPassword()
        {
```

```
            return $this->password;
        }
    }
```

La parte más interesante de esta clase, son las anotaciones escritas sobre la declaración de las variables, de esta manerá añadimos la capa de validación al formulario por ejemplo las anotaciones NotBlank y Regex

```
        /**
         * @Assert\NotBlank()
         * @Assert\Regex("/[a-zA-Z0-9]/")
         *
```

La primera obliga a que el campo no esté vacío en ningún caso, y la segunda, implementa una expresión regular que fuerza a que el texto tan solo contenga caracteres alfanuméricos, sin ningún tipo de símbolo, creamos el formulario para el modelo

```
    <?php
    // src/AppBundle/Form/Type/RegistrationFormType.php

    namespace AppBundle\Form\Type;
    use Symfony\Component\Form\AbstractType;
    use Symfony\Component\Form\FormBuilderInterface;
    use Symfony\Component\Form\Extension\Core\Type;
    use Symfony\Component\OptionsResolver\OptionsResolver;
    /**
     * RegistrationFormType.
     */
    class RegistrationFormType extends AbstractType
    {
        public function buildForm(FormBuilderInterface $builder, array $options)
        {
            $builder
                ->add('username', Type\TextType::class)
                ->add('email', Type\EmailType::class)
                ->add('plainPassword', Type\PasswordType::class)
                ->add('password', Type\PasswordType::class)
            ;
        }
        public function configureOptions(OptionsResolver $resolver)
        {
            $resolver->setDefaults(array(
              'data_class' => 'AppBundle\Form\Model\RegistrationFormModel',
              'csrf_protection' => false,
            ));
        }
        public function getBlockPrefix()
        {
            return 'app_user_registration';
        }
    }
```

Es un formulario muy simple de login, con los campos mínimos para crear un usuario. Ahora le tenemos que decir a nuestro controlador que empiece a utilizarlo.

Para ello utilizaremos las anotaciones de `nelmioApiDocs`, y forzaremos el envío del formulario.

```php
    <?php
    // src/AppBundle/Controller/UserController.php
    ...
    use AppBundle\Form\Type\RegistrationFormType;
    use AppBundle\Form\Model\RegistrationFormModel;
    ...
        /**
         * @ApiDoc(
         *   description = "Register new user.",
         *   input = "AppBundle\Form\Model\RegistrationFormModel",
         *   output = "AppBundle\Model\UserInterface",
         *   statusCodes = {
         *     200 = "User correctly added.",
         *     401 = "Authentication failure, user doesn't have permission or API token is invalid or out
dated.",
         *   }
         * )
         *
         * @param Request $request
         *
         * @return array
         */
        public function postUserAction(Request $request)
        {
            $user = null;
            $form = $this->createForm(RegistrationFormType::class, new RegistrationFormModel(), array('me
thod' => 'POST'));

            $form->submit($request->request->all());

            if ($form->isValid()) {
                try {
                    $rawUser = $this->insertFromForm($form->getData());
                    $user = $this->repository->insert($rawUser);
                    $view = $this->view($user);
                    return $this->handleView($view);
                } catch (\Exception $ex) {
                    //  throw new $ex;
                    $form->addError(new FormError('Duplicate entry for email or username.'));
                    // log this somewhere.
                }
            }
            $view = $this->view($form);
            return $this->handleView($view);
        }
```

Si pasamos los test estaríamos de nuevo en verde, pero como en el primer user story, esto no es más que un falso positivo, crearemos otro test para comprobar los registros validos, para ello crearemos el método `getClient`, para seleccionar si queremos un cliente autenticado o no

```php
<?php
// src/AppBundle/Tests/Controller/UserControllerTest.php

class UserControllerTest extends WebTestCase
{
    const MAIL = 'meco@mail.com';
    ...
    protected function getClient($auth = false)
    {
        if (true === $auth) {
            $client = $this->createAuthenticatedClient(self::NAME, self::PASS);
        } else {
            $client = static::createClient();
        }
        return $client;
    }
    ...
    public function testRegistration()
    {
        $response = $this->post(self::REGISTER_ROUTE, array(
          'username' => self::NAME,
          'email' => self::MAIL,
          'plainPassword' => self::PASS,
          'password' => self::PASS,
            ), true);
        $this->assertEquals(200, $response->getStatusCode());
    }
```

Si volvemos a pasar los tests, vemos que tenemos un método que no existe, podríamos crearlo en el mismo controlador, pero para tener todo mejor organizado, y dejar un fina capa de controladores crearemos un handler para recibir sus valores. Primero definiremos su interface

```php
<?php
// src/AppBundle/Handler/ApiUserHandlerInterface.php
namespace AppBundle\Handler;
/**
 * ApiHandlerInterface.
 */
interface ApiUserHandlerInterface
{
    /**
     * Get user from repository.
     *
     * @param User $user
     */
    public function get($id);
    /**
     * Insert User to repository.
     *
     * @param array $params
     */
    public function post(array $params);
}
```

Para después definimos su implementación

```php
<?php
// src/AppBundle/Handler/ApiUserHandler.php
namespace AppBundle\Handler;
use AppBundle\Model\UserRepository;
use AppBundle\Model\UserInterface;
use AppBundle\Form\Type\RegistrationFormType;
use AppBundle\Form\Model\RegistrationFormModel;
use Symfony\Component\Form\FormFactoryInterface;
use Symfony\Component\Form\FormError;
/**
 * ApiUserHandler.
 */
```

```php
    class ApiUserHandler implements ApiUserHandlerInterface
    {
        /**
         * @var UserRepository
         */
        protected $repository;
        /**
         * @var FormFactoryInterface
         */
        protected $formFactory;
        /**
         * Init Handler.
         *
         * @param UserRepository       $repository
         * @param FormFactoryInterface $formFactory
         */
        public function __construct(UserRepository $repository, FormFactoryInterface $formFactory)
        {
            $this->repository = $repository;
            $this->formFactory = $formFactory;
        }
        /**
         * Get user from repository.
         *
         * @param $id
         *
         * @return User
         */
        public function get($id)
        {
            return $this->repository->parse($id);
        }
        /**
         * Insert User to repository.
         *
         * @param array $params
         *
         * @return User
         */
        public function post(array $params)
        {
            $userModel = RegistrationFormModel::fromArray($params);
            $form = $this->formFactory->create(RegistrationFormType::class, $userModel, array('method' =>
'POST'));
            $form->submit($params);
            if ($form->isValid()) {
                try {
                    $rawUser = $this->insertFromForm($form->getData());
                    $user = $this->repository->insert($rawUser);
                    return $this->repository->parse($user);
                } catch (\Exception $ex) {
                    //  throw new $ex;
                    $form->addError(new FormError('Duplicate entry for email or username.'));
                    // log this somewhere.
                }
            }
            return $form;
        }
        /**
         * @param ProfileFormModel $userModel
         *
         * @return User
         */
        protected function insertFromForm(RegistrationFormModel $userModel)
        {
            $user = $this->repository->findNew();
            $user
                ->setUsername($userModel->getUsername())
                ->setUsernameCanonical($userModel->getUsername())
                ->setPlainPassword($userModel->getPlainPassword())
            ;
            return $this->fromForm($user, $userModel);
```

```
        }
        /**
         * @param User             $user
         * @param ProfileFormModel $userModel
         *
         * @return User
         */
        protected function fromForm(UserInterface $user, RegistrationFormModel $userModel)
        {
            $user
                ->setEmailCanonical($userModel->getEmail())
                ->setEmail($userModel->getEmail())
            ;
            return $user;
        }
    }
```

Lo damos de alta como servicio en el `services.yml`

```
    # spp/config/services.yml
        app.api_user_handler:
            class: AppBundle\Handler\ApiUserHandler
            arguments: [ "@app.user_repository", "@form.factory" ]
```

Y actualizamos el controlador

```
    // src/AppBundle/UserController.php
    ...
        public function postUserAction(Request $request)
        {
            $user = $this->container->get('app.api_user_handler')->post(
                $request->request->all()
            );
            $view = $this->view($user);
            return $this->handleView($view);
        }
```

Ahora debemos añadir varios métodos a nuestro gateway y repository, empezaremos por actualizar `GatewayInterface`

```
    <?php
    // src/AppBundle/Model/GatewayInterface.php

    namespace AppBundle\Model;
    /**
     * UserGateway.
     */
    interface UserGatewayInterface
    {
        /**
         * @param User $user
         *
         * @return User
         */
        public function apiInsert(UserInterface $user);

        /**
         * @return type
         */
        public function findNew();

        /**
         * @param User $user
         *
         * @return User
         */
        public function insert(UserInterface $user);
    }
```

Y añadimos los métodos al repository y al gateway respectivamente

```php
<?php
// src/AppBundle/Entity/Gateway.php

namespace AppBundle\Entity;

use AppBundle\Model\UserInterface;
use AppBundle\Model\UserGatewayInterface;
use Doctrine\ORM\EntityRepository;

/**
 * UserGateway.
 */
class UserGateway extends EntityRepository implements UserGatewayInterface
{
    /**
     * @param User $user
     *
     * @return User
     */
    public function apiInsert(UserInterface $user)
    {
        $user
            ->setEnabled(true)
            ->setExpired(false)
            ->setLocked(false)
            ->addRole('read')
            ->addRole('view')
            ->addRole('edit')
            ->addRole('ROLE_USER')
        ;
        return self::insert($user);
    }

    /**
     * @return type
     */
    public function findNew()
    {
        return User::fromArray();
    }
    /**
     * @param User $user
     *
     * @return User
     */
    public function insert(UserInterface $user)
    {
        $this->_em->persist($user);
        $this->_em->flush();
        return $user;
    }
}
```

Y por último actualizamos nuestro repository

```
    // src/AppBundle/Model/UserRepository.php
    ...
        /**
         * @return User
         */
        public function findNew()
        {
            return $this->gateway->findNew();
        }
        /**
         * @param User $user
         *
         * @return User
         */
        public function insert(UserInterface $user)
        {
            $rawUser = $this->gateway->apiInsert($user);
            return $this->factory->makeOne($rawUser);
        }
```

Si pasamos ahora los test, fallarán, porque el usuario que estamos intentando crear existe ya en la bbdd, así que vamos a crear la acción de borrar usuario para recuperar nuestros tests,

```
    // src/AppBundle/tests/UserControllerTest.php
        ...
        public function testDeleteUser()
        {
            $client = $this->createAuthenticatedClient(self::NAME, self::PASS);

            $id = $this->getLast($client);

            $client->request('DELETE', sprintf(self::ROUTE, $id));

            $this->assertEquals(200, $client->getResponse()->getStatusCode());
        }
```

creamos el método delete en el controlador

```
    // src/AppBundle/Controller/UserController.php
        ...
        /**
         * @Security("is_granted('edit', user)")
         * @ApiDoc(
         *   description = "Delete own user.",
         *   statusCodes = {
         *     204 = "Do no return nothing.",
         *     401 = "Authentication failure, user doesn’t have permission or API token is invalid or out
dated.",
         *   }
         * )
         *
         * @return array
         */
        public function deleteUserAction($id)
        {
            $this->container->get('app.api_user_handler')->delete($id);

            $view = $this->view(array());
            return $this->handleView($view);
        }
```

Después lo creamos en nuestro handler

```php
// src/AppBundle/Handler/ApiUserHandler.php
...
    /**
     * Delete User.
     *
     * @param $id
     */
    public function delete($id)
    {
        $this->repository->remove($id);
    }
```

Y añadimos los métodos a nuestro modelo, primero al interface

```php
// src/AppBundle/Model/GatewayInterface.php
...
    /**
     * @param $id
     */
    public function remove($id);
```

Luego en la clase

```php
// src/AppBundle/Entity/Gateway.php
...
    /**
     * @param $id
     */
    public function remove($id)
    {
        $user = $this->find($id);

        $this->_em->remove($user);
        $this->_em->flush();
    }
```

y por último en el repository

```php
// src/AppBundle/Model/UserRepository.php
...
    /**
     * @param $id
     */
    public function remove($id)
    {
        $this->gateway->remove($id);
    }
```

si pasamos los test tendremos un 403, porque el usuario aunque autenticado, no tiene un voter que le permita el acceso. Vamos a añadir la entrada edit al voter que creamos antes.

Además también quitaremos la dependencia con doctrine en el voter

```php
<?php
// src/AppBundle/Security/UserVoter.php
namespace AppBundle\Security;
use Symfony\Component\Security\Core\Authorization\Voter\Voter;
use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
use AppBundle\Model\UserInterface;
class UserVoter extends Voter
{
    const EDIT = 'edit';
    const VIEW = 'view';
    public function supports($attribute, $subject)
    {
        return $subject instanceof UserInterface && in_array($attribute, array(
              self::VIEW, self::EDIT,
        ));
    }
    protected function voteOnAttribute($attribute, $currentUser, TokenInterface $token)
    {
        $user = $token->getUser();
        if (!$user instanceof UserInterface) {
            return false;
        }
        $roles = $user->getRoles();
        if (
            in_array(strtoupper(self::EDIT), $roles) &&
            $attribute == self::EDIT &&
            $user->getEmail() == $currentUser->getEmail()
        ) {
            return true;
        }
        if (
            in_array('ROLE_USER', $roles) &&
            $attribute == self::VIEW &&
            $user->getEmail() == $currentUser->getEmail()
        ) {
            return true;
        }
        return false;
    }
}
```

Pasamos de nuevo los tests, debemos tener todo correcto, en este punto ya tenemos implementado el patrón repositorio, haciendo uso del principio de inversión de dependencias.

Parte 5, Método PUT

Ahora vamos a por el user story 3 `Como usuario autenticado puedo editar mi perfil`, para esto utilizaremos el método PUT que vimos antes.

Como en los demás user storys empezamos con un test

```php
<?php
// src/AppBundle/Tests/Controller/UserControllerTest.php

namespace AppBundle\Tests\Controller;

use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

class UserControllerTest extends WebTestCase
{
    ...
    const DESCRIPTION = 'ha sido el texto de relleno estándar de las industrias desde el año 1500, '
        . 'cuando un impresor (N. del T. persona que se dedica a la imprenta) desconocido';
    ...
    protected function put($uri, array $data, $auth = false)
    {
        $client = $this->getClient($auth);

        $client->request('PUT', $uri, $data);

        return $client->getResponse();
    }
```

Agregamos el método put en nuestros tests, lo usaremos para no repetir las peticiones en los distintos tests.

```php
    // src/AppBundle/Tests/Controller/UserControllerTest.php

    public function testPutUserWithOutAuthentication()
    {
        $client = static::createClient();

        $id = $this->getLast($client);

        $client->request('PUT', sprintf(self::ROUTE, $id));

        $this->assertEquals(401, $client->getResponse()->getStatusCode());
    }

    public function testPutUserWithOutRequiredParams()
    {
        $client = static::createClient();
        $id = $this->getLast($client);

        $response = $this->put(sprintf(self::ROUTE, $id), array(
          'email' => null,
          'description' => self::DESCRIPTION,
            ), true);

        $this->assertEquals(400, $response->getStatusCode());
    }

    public function testPutUser()
    {
        $client = static::createClient();
        $id = $this->getLast($client);

        $response = $this->put(sprintf(self::ROUTE, $id), array(
          'email' => 'asd' . self::MAIL,
          'description' => self::DESCRIPTION,
            ), true);

        $this->assertEquals(200, $response->getStatusCode());
    }
```

El primer test comprueba el comportamiento cuando el usuario intenta acceder sin autenticar, el segundo valida que el comportamiento cuando los campos obligatorios no se han rellenado y el tercero, comprueba que la actualización se realiza correctamente.

Pasamos los test para obtener información. Como siempre, necesitamos una ruta y un controlador, en este caso, al igual que en el post necesitamos un formulario. Empezamos por el controlador.

```php
    <?php
    // src/AppBundle/Controller/UserController.php
        ...
        /**
         * @Security("is_granted('edit', user)")
         * @ApiDoc(
         *   description = "Update own user.",
         *   input = "AppBundle\Form\Model\ProfileFormModel",
         *   output = "AppBundle\Entity\User",
         *   statusCodes = {
         *     200 = "User data updated.",
         *     401 = "Authentication failure, user doesn't have permission or API token is invalid or out
dated.",
         *   }
         * )
         *
         * @param Request $request
         *
         * @return array
         */
        public function putUserAction(Request $request, $id)
        {
            $user = $this->container->get('app.api_user_handler')->put(
                $id, $request->request->all()
            );

            $view = $this->view($user);
            return $this->handleView($view);
        }
        ...
```

El método `put` de nuestro handler no existe, creemoslo…

Agregamos el método `put` al interface de nuestro handler

```php
    // src/AppBundle/Handler/UserHandlerInterface.php
    ...
        /**
         * Update User from repository.
         *
         * @param $id
         * @param array $params
         */
        public function put($id, array $params);
    }
```

Lo implementamos en el handler

```php
    // src/AppBundle/Handler/UserHandler.php
        ...
        /**
         * Update User from repository.
         *
         * @param array $params
         *
         * @return type
         */
        public function put($id, array $params)
        {
            $userModel = ProfileFormModel::fromArray($params);

            $form = $this->formFactory->create(ProfileFormType::class, $userModel, array('method' => 'POST
'));
            $form->submit($params);

            if ($form->isValid()) {
                $user = $this->updateFromForm($id, $form->getData());

                $this->repository->update();

                return $this->repository->parse($user->getId());
            }

            return $form;
        }
        ...
```

Después actualizamos los métodos `insertFromForm` y `fromForm` para que utilicen el interface `UserFormModelInterface`, que crearemos a continuación. Además añadimos el método `updateFromForm`.

```php
        /**
         * @param ProfileFormModel $userModel
         *
         * @return User
         */
        protected function insertFromForm(UserFormModelInterface $userModel)
        ...
        /**
         * @param type             $id
         * @param ProfileFormModel $userModel
         *
         * @return User
         */
        protected function updateFromForm($id, UserFormModelInterface $userModel)
        {
            $user = $this->repository->find($id);

            $user->setDescription($userModel->getDescription());

            return $this->fromForm($user, $userModel);
        }
        /**
         * @param User             $user
         * @param ProfileFormModel $userModel
         *
         * @return User
         */
        protected function fromForm(UserInterface $user, UserFormModelInterface $userModel)
        ...
```

Por último nos falta el formulario `ProfileFormType`, su modelo. Creamos el interface para los modelos de formulario de usuario

```php
<?php
// src/AppBundle/Form/Model/UserFormModelInterface.php
namespace AppBundle\Form\Model;
interface UserFormModelInterface
{
    /**
     * @param array $user
     *
     * @return \self
     */
    public static function fromArray(array $user = array());
    /**
     * @param string $email
     */
    public function setEmail($email);

    public function getEmail();
}
```

Hacemos que el `RegisterFormModel` lo implemente, e implementamos el nuevo `ProfileFormModel`

```php
<?php
// src/AppBundle/Form/Model/ProfileFormModel.php
namespace AppBundle\Form\Model;
use Symfony\Component\Validator\Constraints as Assert;
/**
 * ProfileFormModel.
 */
class ProfileFormModel implements UserFormModelInterface
{
    const MAIL = 'email';
    const DESC = 'description';
    /**
     * @Assert\NotBlank()
     * @Assert\Email()
     *
     * @var string
     */
    protected $email;
    /**
     * @Assert\Regex("/[a-z\d\-_\s]+/")
     *
     * @var string
     */
    protected $description;
    /**
     *
     * @param type $username
     * @param type $email
     * @param type $description
     */
    public function __construct($username = null, $email = null, $description = null)
    {
        $this->username = $username;
        $this->email = $email;
        $this->description = $description;
    }
    /**
     * @param array $user
     *
     * @return \self
     */
    public static function fromArray(array $user = array(self::MAIL => null, self::DESC => null))
    {
        return new self(
            array_key_exists(self::MAIL, $user) ? $user[self::MAIL] : null,
            array_key_exists(self::DESC, $user) ? $user[self::DESC] : null
        );
    }
    public function setEmail($email)
    {
        $this->email = $email;
    }
    public function getEmail()
    {
        return $this->email;
    }
    public function setDescription($description)
    {
        $this->description = $description;
    }
    public function getDescription()
    {
        return $this->description;
    }
}
```

Nos falta el formulario `ProfileFormType`

```php
<?php
// src/AppBundle/Form/Type/ProfileFormType.php
namespace AppBundle\Form\Type;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\Form\Extension\Core\Type;
use Symfony\Component\OptionsResolver\OptionsResolver;
/**
 * ProfileFormType.
 */
class ProfileFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('username', Type\TextType::class)
            ->add('email', Type\EmailType::class)
            ->add('description', Type\EmailType::class)
        ;
    }

    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults(array(
          'data_class' => 'AppBundle\Form\Model\ProfileFormModel',
          'csrf_protection' => false,
        ));
    }

    public function getBlockPrefix()
    {
        return 'app_user_registration';
    }
}
```

Si pasamos los tests, nos damos cuenta que nos falta por implementar el método `UserRepository::update()`, es más, nos falta toda la parte de persistencia. Vamos a ello. Añadimos `Update` al `GatewayInterface`

```php
// src/AppBundle/Model/GatewayInterface.php
    ...
    /**
     * Update User
     */
    public function update();
    ...
```

Lo implementamos en el `Gateway`

```php
// src/AppBundle/Entity/UserGateway.php
    ...
    /**
     * Update User.
     */
    public function update()
    {
        $this->_em->flush();
    }
    ...
```

Por último actualizamos nuestro `Repository`

```
    // src/AppBundle/Model/UserRepository.php
        ...
        /**
         * Update User.
         */
        public function update()
        {
            return $this->gateway->update();
        }
        ...
```

Pasamos los tests y todo debe seguir correcto. Como ejercicio podríamos implementar el verbo `PATCH`.

Para terminar, comprobaremos la cobertura que estamos dando a nuestro código con la herramienta code coverage de phpunit.

```
phpunit -c app/ --coverage-html ./web/coverage # Para verlo en la intefaz gráfico o
phpunit -c app/ --coverage-text # para ver en informe en consola.
```

Parte 6, Demostrando la Inversión de dependencias

Si todo lo anterior es realmente correcto, cambiar el motor de base de datos no debería ser una tarea complicado. En nuestro caso hemos utilizado Doctrine ORM junto con Mysql, pero por requisitos de proyecto, podríamos necesitar cambiarlo por otro. por ejemplo MongoDB.

Haremos un caso practico de sustitución de Doctrine ORM por Doctrine ODM

Site 2, Cliente

No tiene porque ser symfony, podría servir cualquier aplicación que consuma el API, AngularJS, Backbone, Ember, etc… De momento utilizaremos el sandbox que generamos con NelmioApiDocBundle.