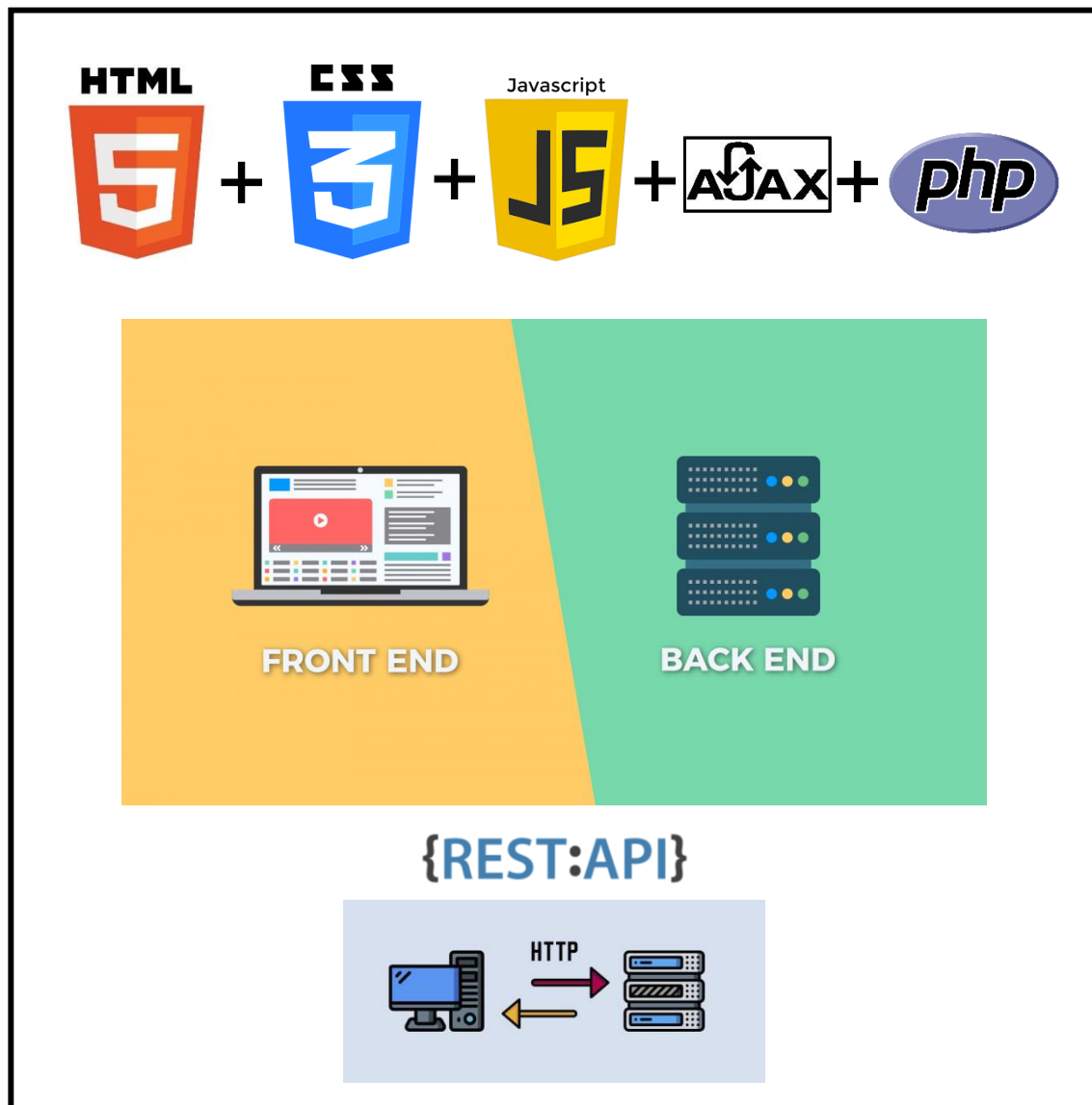


PROYECTO FINAL

TECNOLOGÍAS DE DESARROLLO WEB



- Integrantes del grupo:

Álvaro Avilés Redondo – bp0259

Jesús Antón Díaz – bp0005

Índice

| | | |
|-----|--|----|
| 1. | Introducción..... | 2 |
| 2. | Pasos previos y configuración | 2 |
| 2.1 | Software necesario | 2 |
| 2.2 | Configuración de la base de datos y dependencias | 2 |
| 3.3 | Levantar el servidor..... | 4 |
| 3. | Modelo de datos..... | 5 |
| 4. | Arquitectura del sistema | 6 |
| 5. | Mecanismos de protección utilizados..... | 7 |
| 6. | Especificación OpenAPI de la API REST | 8 |
| 7. | Problemas encontrados | 10 |
| 8. | Bibliografía | 11 |

1. Introducción

El objetivo de esta práctica es la unión de la practica 1 (parte de frontend) con la práctica 2 (backend) mediante el uso de AJAX, obteniendo de esta forma una aplicación completa. Para ello tendremos que remodelar el frontend realizado para que en lugar de usar como persistencia el LocalStorage del navegador use una base de datos, en este caso una relacional MySQL. Para esto trabajaremos sobre una API REST mediante peticiones HTTP a los diferentes endpoints. En algún punto es necesario modificar la API REST para añadir algún end-point.

Todo el proyecto también se encuentra disponible en GitHub en el repositorio:

https://github.com/TDW-Practicas-2022/PROYECTO_FINAL

2. Pasos previos y configuración

Para poder levantar, probar la aplicación y comprobar que todo funciona debemos primero realizar un serie de pasos y configuraciones que se detallan en este apartado. Adicionalmente a las instrucciones de esta memoria, en el repositorio de GitHub, estará disponible un fichero ReadMe en formato markdown donde también se detallarán estos pasos para levantar el servidor y configurarlo todo.

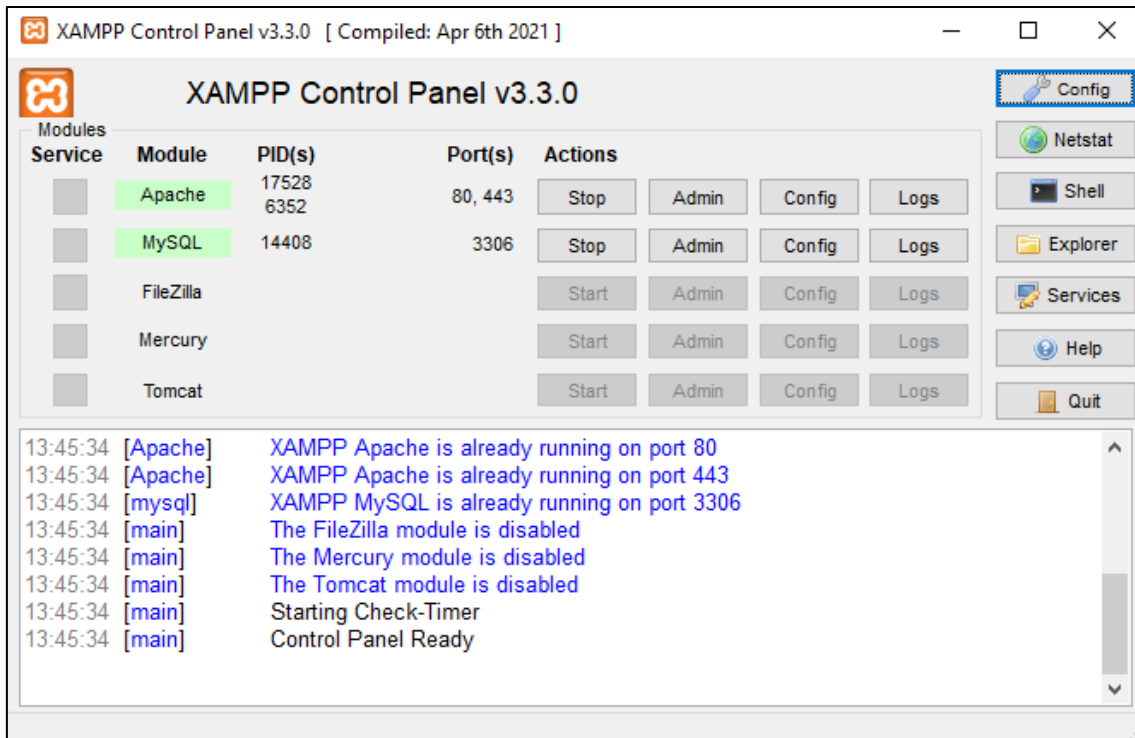
2.1 Software necesario

En cuanto al software necesario para probar y ejecutar esta aplicación debemos tener instalado y debidamente configurado:

- Composer: tener instalado Composer para la gestión de dependencias.
- XAMPP: para levantar el servidor de la base de datos y la interfaz gráfica Phpmyadmin para hacer el dumpeo inicial de la base de datos. También para disponer de un intérprete PHP y poder configurarlo mediante el fichero php.ini.
- PHPSTORM: IDE para abrir el proyecto y muy útil para lanzar los comandos del gestor de dependencias Composer y el ORM Doctrine para el mapeo con la base de datos.

2.2 Configuración de la base de datos y dependencias

Lo primero será levantar la base de datos mediante XAMPP de esta manera:



Una vez está levantado, creamos esquema de base de datos vacío y en el proyecto en PhpStorm creamos un fichero llamado .env.local donde configuraremos los parámetros de conexión con la base de datos. Una vez hecho esto ejecutamos desde la terminal y en la carpeta raíz del proyecto los comandos:

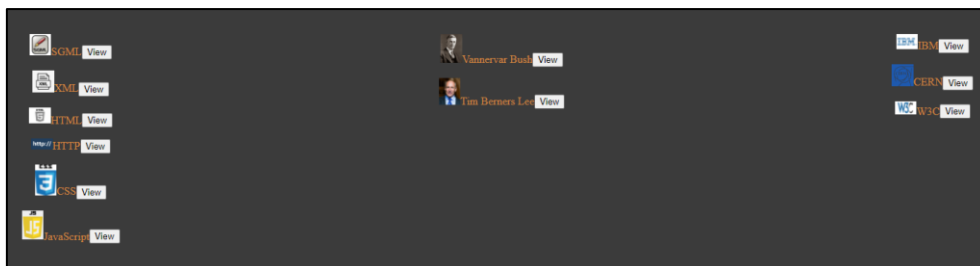
- `composer install`
- `bin/doctrine orm:schema:update --dump-sql --force`

Adicionalmente para comprobar que el mapeo y la sincronización con la base de datos es son correctos, podemos ejecutar el siguiente comando:

- `bin/doctrine orm:validate`

Una vez hecho esto, se habrá insertado en la tabla *users* de la base de datos un usuario administrador (Usuario: *userAdmin*, contraseña: **adminUser**).

Ahora podemos realizar el dumpeo de la base de datos para insertar los productos, entidades y personas iniciales, es decir estos:



Para hacer este dumpeo, se proporciona un fichero sql que sobrescribirá la base de datos con las personas, entidades y productos iniciales. Se mantendrá el usuario *adminUser*. Este fichero se llama *InitialDump_TDW.sql* y está disponible en el directorio raíz del proyecto.

Para realizar este dump de proporcionan los insert de las tablas entity, person y product. Simplemente hay que ejecutar las queries en phpmyadmin para que se inserten los datos. Si se ha realizado un composer install, se habrá insertado un usuario en la tabla users, borrarlo antes de realizar los INSERTS.

***Nota:** Es muy importante hacer este dump, sobretodo para la tabla users, ya que el adminUser tras el composer install tiene el atributo isActive a 0 y este debe estar a 1 para poder acceder a la web tras el login.

3.3 Levantar el servidor

El siguiente paso es levantar el servidor del backend para que funcione de manera local en nuestra máquina, para ello ejecutamos el siguiente comando:

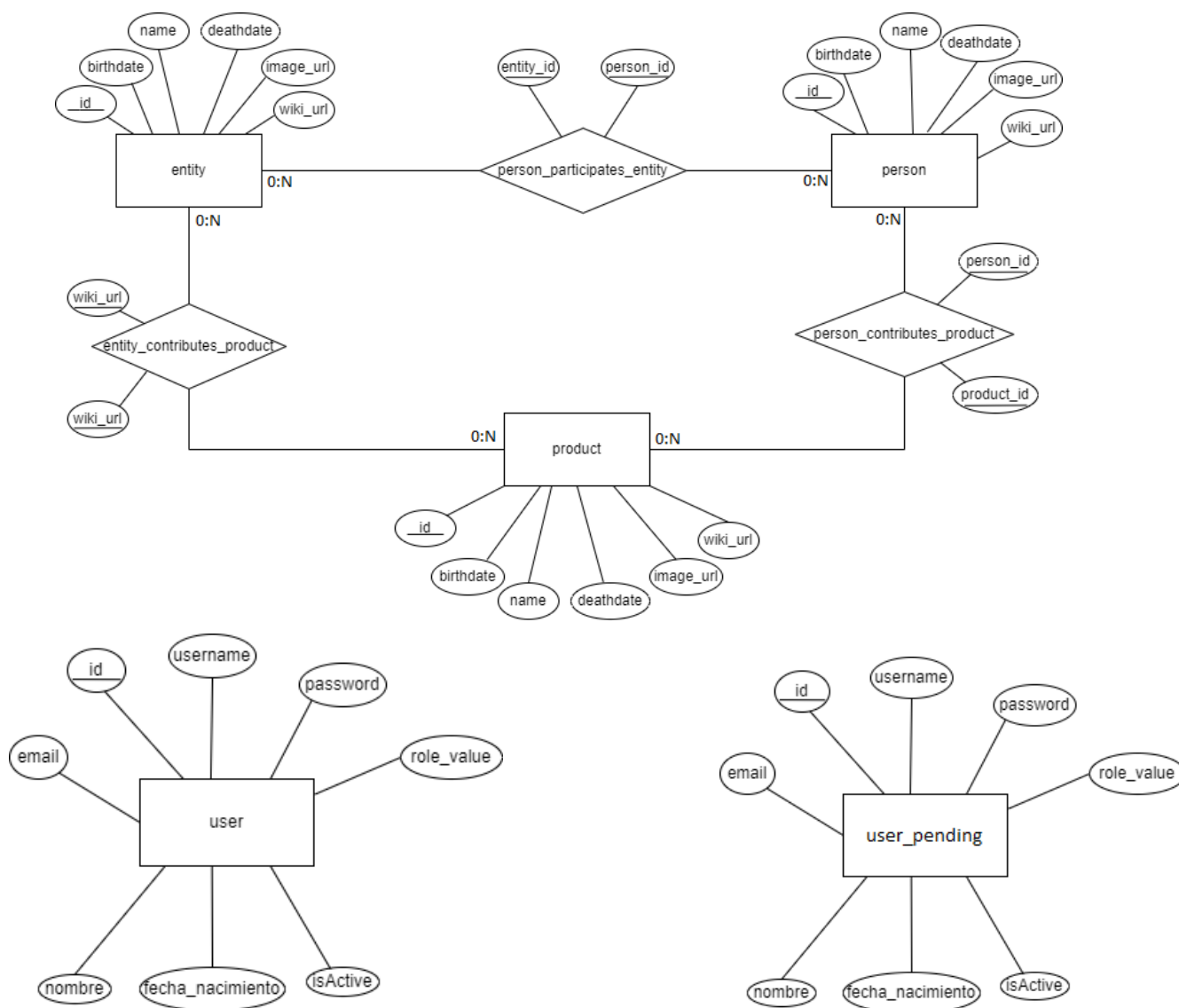
```
- php -S 127.0.0.1:8000 -t public
```

Una vez hecho esto podemos acceder a la especificación OpenAPI en SwaggerUI en la siguiente URL: <http://127.0.0.1:8000/api-docs/index.html>

O bien podemos acceder a nuestro frontend desde:
<http://127.0.0.1:8000/webPage/index.html>

3. Modelo de datos

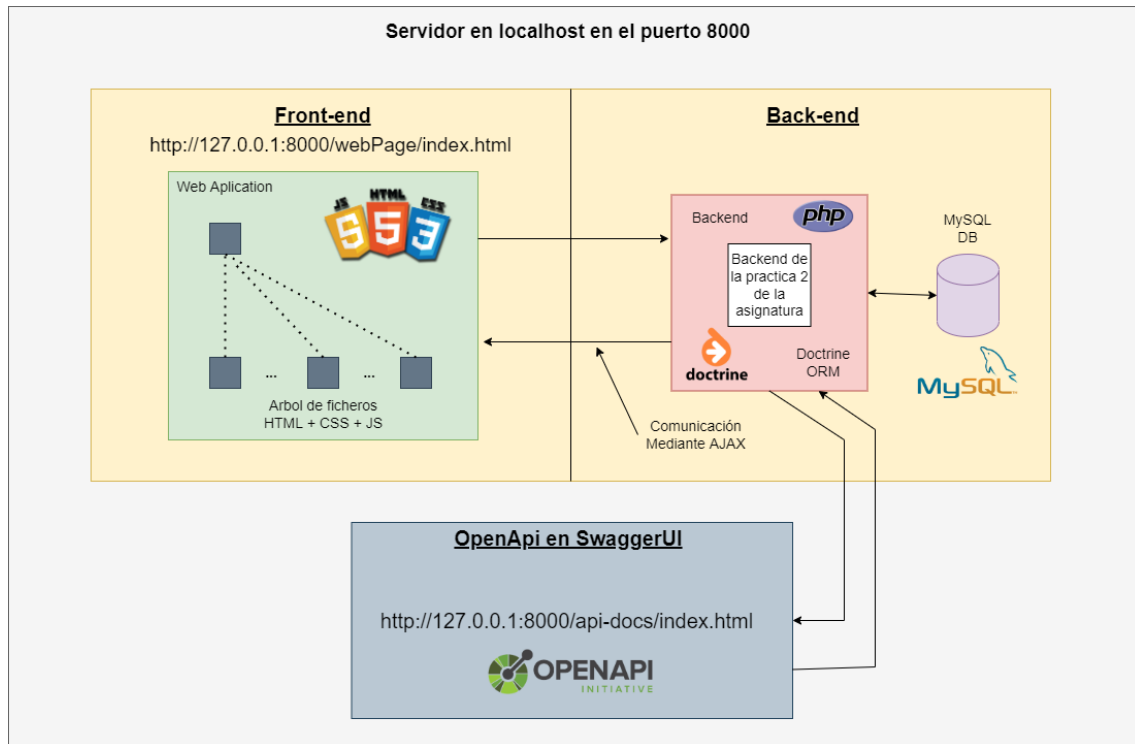
Diagrama ER:



En el modelo de datos que hemos sacado de la aplicación, se puede observar claramente las entidades "Entity", "Person" y "Product" así como sus relaciones. Cada una de estas entidades almacena los datos correspondientes a los productos, personas y entidades.

También tenemos la entidad "User" encargada de almacenar todos los datos de los usuarios registrados y aceptados en el sistema, y la entidad "User_Pending" encargada de almacenar los datos de aquellos usuarios que se han registrado en el sistema y están a la espera de ser aceptados por un writer. Cuando un usuario de la entidad "User_Pending" es aceptado, los datos de este son creados en la entidad "User" y eliminados en la entidad "User_Pending"

4. Arquitectura del sistema



Esta es la arquitectura del sistema. Como podemos observar, se divide en dos:

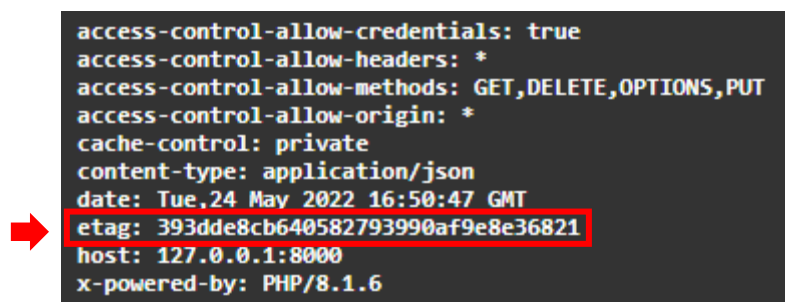
- **Front- end:** Es la parte frontal del sistema, encargada de la interacción con el usuario. Es un sistema de ficheros HTML, CSS y JS. Desde el front se realizan las peticiones necesarias a los diferentes end-points del back-end mediante peticiones HTTP (GET, PUT, DELETE Y POST) haciendo uso de la librería AJAX para poder realizar dichas peticiones.
- **Back-end:** Es la parte trasera del sistema, encargada de la gestión y respuesta de las peticiones HTTP realizadas desde el front. Está codificada en PHP haciendo uso del framework SLIM. También utiliza el ORM Doctrine para el mapeo y comunicación con la base de datos relacional MySQL que hemos utilizado.

Adicionalmente también contamos con una especificación OpenAPI mediante la cual se pueden probar los diferentes end-points del back-end. Esta especificación OpenAPI es posible visualizarla mediante SwaggerUI.

5. Mecanismos de protección utilizados

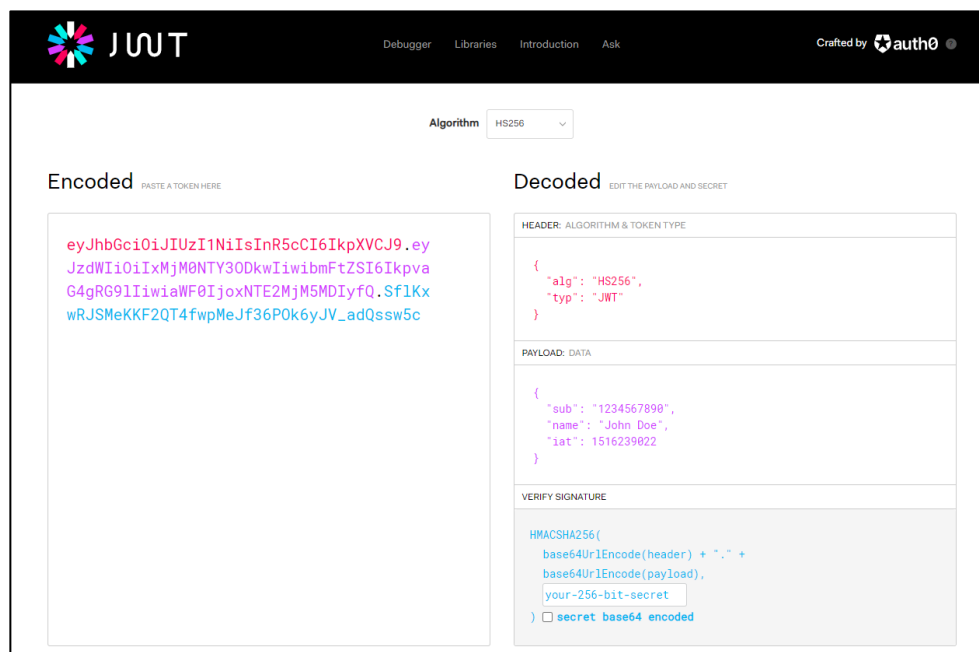
En esta práctica se han utilizado diferentes mecanismos de protección para darle cierta seguridad a la aplicación:

- Encriptación de las contraseñas
- Uso de E-tags para controlar la correcta actualización mediante una petición PUT evitando actualizaciones simultáneas de un recurso, eliminando así la posibilidad de que se puedan sobre escribir entre ellas. Adicionalmente, el uso de E-tags permite a la memoria caché ser más eficiente y ahorrar ancho de banda ya que si un recurso no ha sido modificado, el servidor no tendrá que mandar la respuesta completa si el contenido no ha cambiado.

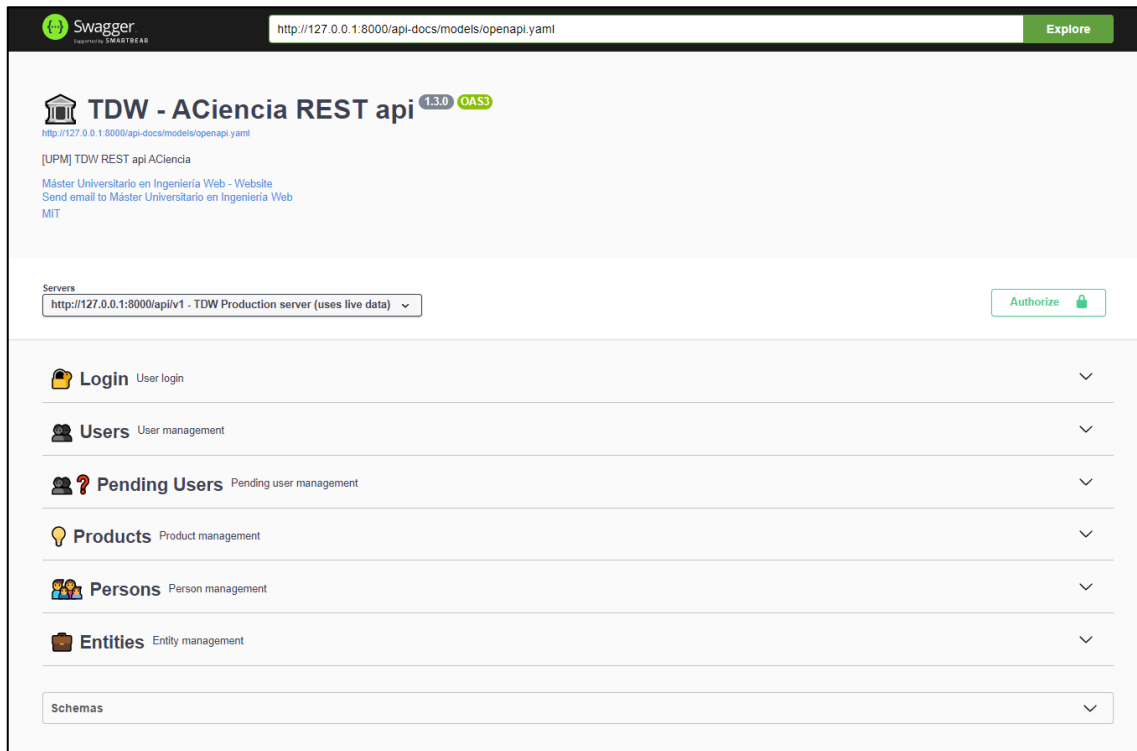


```
access-control-allow-credentials: true
access-control-allow-headers: *
access-control-allow-methods: GET,DELETE,OPTIONS,PUT
access-control-allow-origin: *
cache-control: private
content-type: application/json
date: Tue, 24 May 2022 16:50:47 GMT
etag: 393dde8cb640582793990af9e8e36821
host: 127.0.0.1:8000
x-powered-by: PHP/8.1.6
```

- Uso de Json Web Token (JWT) para evitar que, si un usuario no tiene ciertos permisos dentro de la aplicación, no pueda utilizar ciertos end-points de esta. Básicamente un JWD es una cadena de caracteres dividida en tres campos (header, payload y verify signature) que permite compartir información de forma segura entre un cliente y un servidor.



6. Especificación OpenAPI de la API REST



Swagger
powered by SMARTER

<http://127.0.0.1:8000/api-docs/models/openapi.yaml> Explore

TDW - ACiencia REST api 1.3.0 OAS3

<http://127.0.0.1:8000/api-docs/models/openapi.yaml>

[UPM] TDW REST api ACiencia

Máster Universitario en Ingeniería Web - Website
Send email to Máster Universitario en Ingeniería Web
MIT

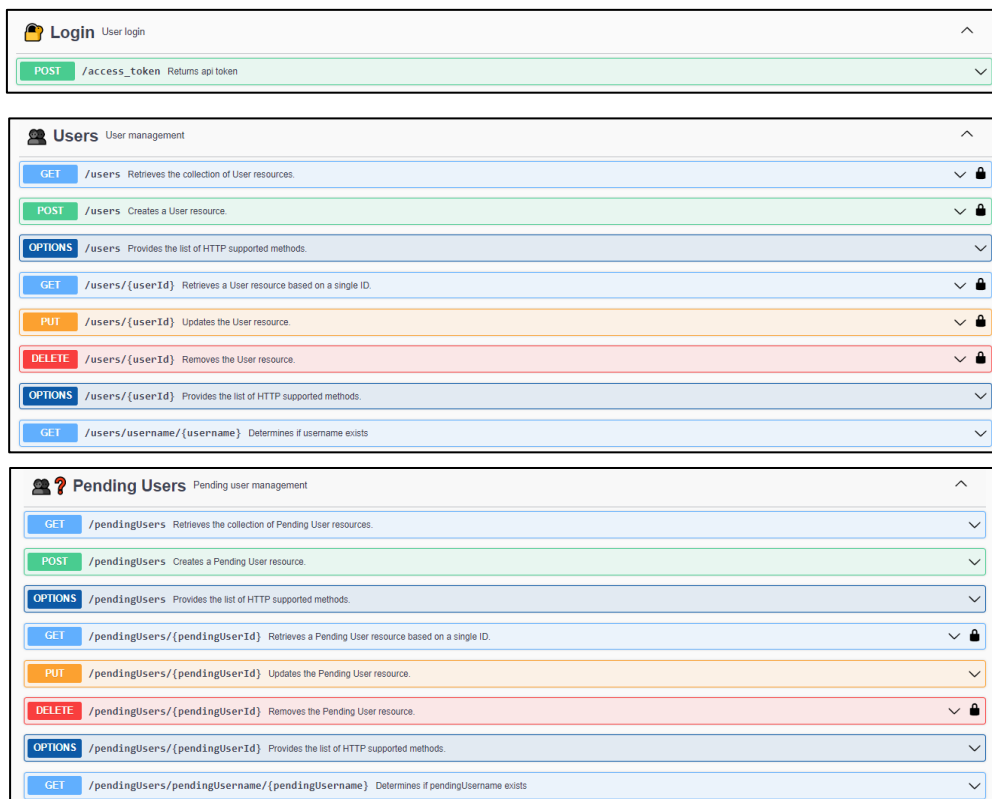
Servers
<http://127.0.0.1:8000/api/v1> - TDW Production server (uses live data) Authorize

- Login** User login
- Users** User management
- Pending Users** Pending user management
- Products** Product management
- Persons** Person management
- Entities** Entity management

Schemas

En esta práctica hemos completado la especificación OpenAPI para los nuevos end-points que hemos creado, así como para los que hemos modificado. Está disponible dentro del proyecto en la carpeta: /public/api-docs/models. Esta especificación puede ser probada levantando la aplicación y accediendo a la siguiente ruta: <http://127.0.0.1:8000/api-docs/index.html>

A continuación, se pueden observar los diferentes end-points que tiene esta API REST:



Login User login

- POST** /access_token Returns api token

Users User management

- GET** /users Retrieves the collection of User resources.
- POST** /users Creates a User resource.
- OPTIONS** /users Provides the list of HTTP supported methods.
- GET** /users/{userId} Retrieves a User resource based on a single ID.
- PUT** /users/{userId} Updates the User resource.
- DELETE** /users/{userId} Removes the User resource.
- OPTIONS** /users/{userId} Provides the list of HTTP supported methods.
- GET** /users/username/{username} Determines if username exists

Pending Users Pending user management

- GET** /pendingUsers Retrieves the collection of Pending User resources.
- POST** /pendingUsers Creates a Pending User resource.
- OPTIONS** /pendingUsers Provides the list of HTTP supported methods.
- GET** /pendingUsers/{pendingUserId} Retrieves a Pending User resource based on a single ID.
- PUT** /pendingUsers/{pendingUserId} Updates the Pending User resource.
- DELETE** /pendingUsers/{pendingUserId} Removes the Pending User resource.
- OPTIONS** /pendingUsers/{pendingUserId} Provides the list of HTTP supported methods.
- GET** /pendingUsers/pendingUsername/{pendingUsername} Determines if pendingUsername exists

| 💡 Products Product management | | ^ |
|-------------------------------|--|-----|
| GET | /products Retrieves the collection of Product resources. | ▼ |
| POST | /products Creates a Product resource. | ▼ 🔒 |
| OPTIONS | /products Provides the list of HTTP supported methods. | ▼ |
| GET | /products/{productId} Retrieves a Product resource based on a single ID. | ▼ |
| PUT | /products/{productId} Updates the Product resource. | ▼ 🔒 |
| DELETE | /products/{productId} Removes the Product resource. | ▼ 🔒 |
| OPTIONS | /products/{productId} Provides the list of HTTP supported methods. | ▼ |
| GET | /products/productname/{productname} Determines if productname exists | ▼ |
| GET | /products/{productId}/entities List of entities related to the product | ▼ |
| GET | /products/{productId}/persons List of persons related to the product | ▼ |
| PUT | /products/{productId}/entities/{operation}/{entityId} Sets or remove the relationship product-entity | ▼ 🔒 |
| PUT | /products/{productId}/persons/{operation}/{personId} Sets or remove the relationship product-person | ▼ 🔒 |

| 👤 Persons Person management | | ^ |
|-----------------------------|---|-----|
| GET | /persons Retrieves the collection of Person resources. | ▼ |
| POST | /persons Creates a Person resource. | ▼ 🔒 |
| OPTIONS | /persons Provides the list of HTTP supported methods. | ▼ |
| GET | /persons/{personId} Retrieves a Person resource based on a single ID. | ▼ |
| PUT | /persons/{personId} Updates the Person resource. | ▼ 🔒 |
| DELETE | /persons/{personId} Removes the Person resource. | ▼ 🔒 |
| OPTIONS | /persons/{personId} Provides the list of HTTP supported methods. | ▼ |
| GET | /persons/personname/{personname} Determines if personname exists | ▼ |
| GET | /persons/{personId}/entities List of entities related to the person | ▼ |
| GET | /persons/{personId}/products List of products related to the person | ▼ |
| PUT | /persons/{personId}/entities/{operation}/{entityId} Sets or remove the relationship person-entity | ▼ 🔒 |
| PUT | /persons/{personId}/products/{operation}/{productId} Sets or remove the relationship person-product | ▼ 🔒 |

| 📦 Entities Entity management | | ^ |
|------------------------------|--|-----|
| GET | /entities Retrieves the collection of Entity resources. | ▼ |
| POST | /entities Creates a Entity resource. | ▼ 🔒 |
| OPTIONS | /entities Provides the list of HTTP supported methods. | ▼ |
| GET | /entities/{entityId} Retrieves a Entity resource based on a single ID. | ▼ |
| PUT | /entities/{entityId} Updates the Entity resource. | ▼ 🔒 |
| DELETE | /entities/{entityId} Removes the Entity resource. | ▼ 🔒 |
| OPTIONS | /entities/{entityId} Provides the list of HTTP supported methods. | ▼ |
| GET | /entities/entityname/{entityname} Determines if entityname exists | ▼ |
| GET | /entities/{entityId}/persons List of persons related to the entity | ▼ |
| GET | /entities/{entityId}/products List of products related to the entity | ▼ |
| PUT | /entities/{entityId}/persons/{operation}/{personId} Sets or remove the relationship entity-person | ▼ 🔒 |
| PUT | /entities/{entityId}/products/{operation}/{productId} Sets or remove the relationship entity-product | ▼ 🔒 |

7. Problemas encontrados

En la realización de la práctica nos hemos encontrado con un problema y tras intentar solucionarlo varias veces, no lo hemos conseguido. Este problema está relacionado con la creación de nuevos end-points en la API.

Concretamente en la parte de registro de usuarios, dado que deben registrarse usuarios que no estén logeados en la aplicación (es decir que no son ni reader ni writer), es necesario crear una nueva tabla de usuarios (PendingUsers) donde un usuario sin identificar pueda hacer un post de sus datos. Posteriormente, un writer logeado en la aplicación podrá ver los usuarios que están dentro de esta nueva tabla y decidir si darlos de alta (hacer un POST de sus datos en la tabla Users y un DELETE de la tabla de PendingUsers).

En la creación de estos nuevos endpoints hemos tenido los siguientes problemas:

Aunque hemos creado correctamente los end-points CGET, GET por id, DELETE y OPTIONS, los end-points POST, PUT y GET por username no hemos conseguido que funcionen.

Todo el código de desarrollado está disponible en el proyecto en los ficheros:

- **PendingUser.php:** entidad PendingUser mapeada con el ORM para crear la nueva tabla en la base de datos.
- **PendingUserController.php:** controlador encargado de la gestión de las peticiones a las rutas de esta nueva tabla.
- **routesPendingUsers.php:** fichero php encargado de la redirección de las rutas para la gestión de las operaciones sobre la tabla PendingUsers.

Consideramos que hemos entendido el procedimiento de creación de nuevos end-points y como gestiona de manera interna el back las peticiones HTTP que recibe a pesar de no haber sido capaces de que las peticiones a esta nueva tabla funcionen correctamente.

No sabemos porque, pero al hacer un POST, en lugar de crearlo en la tabla pendingusers, lo hace en la tabla user. Sin embargo el GET lo hace sobre la tabla pendingusers.

El problema que conlleva para la realización de la práctica que no hayamos sido capaces de poder utilizar una tabla intermedia para el registro de nuevos usuarios es no poder darle funcionalidad a esta parte.

Por ello y como solución temporal (**con simples fines de demostración de funcionalidad**) hemos desarrollado la funcionalidad haciendo uso de LocalStorage como base de datos para guardar temporalmente a los nuevos usuarios. **Sabemos que esta solución es completamente inválida** por razones de seguridad (contraseñas sin encriptar y fáciles de recuperar) y de persistencia (el LocalStorage no se mantiene entre sesiones). Esta funcionalidad está incorporada en el frontend.

Como sería la funcionalidad si hubiéramos logrado hacer los end-points necesarios

Suponiendo que los end-points creados funcionaran correctamente este sería el trabajo que tendríamos que haber realizado para darle soporte al registro de nuevos usuarios:

Primero, un usuario sin identificar se quiere registrar en el sistema, por lo que puede hacer un POST a la ruta pendingUsers ya que este end-point ha sido modificado para no necesitar cabecera de autenticación. Luego se usaría el end-point GET username para comprobar que el nombre de usuario no existe en el sistema. Una vez hecho esto, pasaría a otro HTML donde podría completar el resto de sus datos haciendo un PUT a la ruta pendingUsers.

Posteriormente, un wirtter logeado desde la aplicación podría ver el listado de usuarios que quieren registrarse mediante un CGET, y podría elegir entre:

- Hacer un POST del nuevo usuario en la tabla users dando de alta en el sistema al nuevo usuario y haciendo un DELETE de ese usuario en la tabla de pendingUsers.
- Rechazar el registro y hacer un DELETE de ese usuario en la tabla de pendingUsers.

8. Bibliografía

- <https://www.doctrine-project.org/projects/doctrine-orm/en/2.12/tutorials/getting-started.html>
- <https://swagger.io/specification/>
- <https://www.php.net/docs.php>
- Todo la información disponible en el Moodle de la asignatura