Practica 1: Desarrollo de una API Web que actúa como back-end con JPA y JAX-RS

Componentes del grupo

Khady Yade Diouf

Roger García Doncel

Índex

1. Introducción

1.1. Objetivos del proyecto

- Implementar una API REST completa para la gestión de un catálogo de modelos LLM (Large Language Models)
- Aplicar los principios REST: recursos bien definidos, métodos HTTP estándar, códigos de estado apropiados
- Implementar versionado de API (/rest/api/v1/)
- Soportar formatos JSON y XML
- Implementar autenticación HTTP Basic Authentication mediante anotación @Secured
- Crear tests automatizados con Postman
- Gestionar relaciones complejas entre entidades JPA (ManyToOne, ManyToMany, OneToOne)

1.2. Descripción general

Sistema web basado en Jakarta EE 9+ que expone servicios REST para:

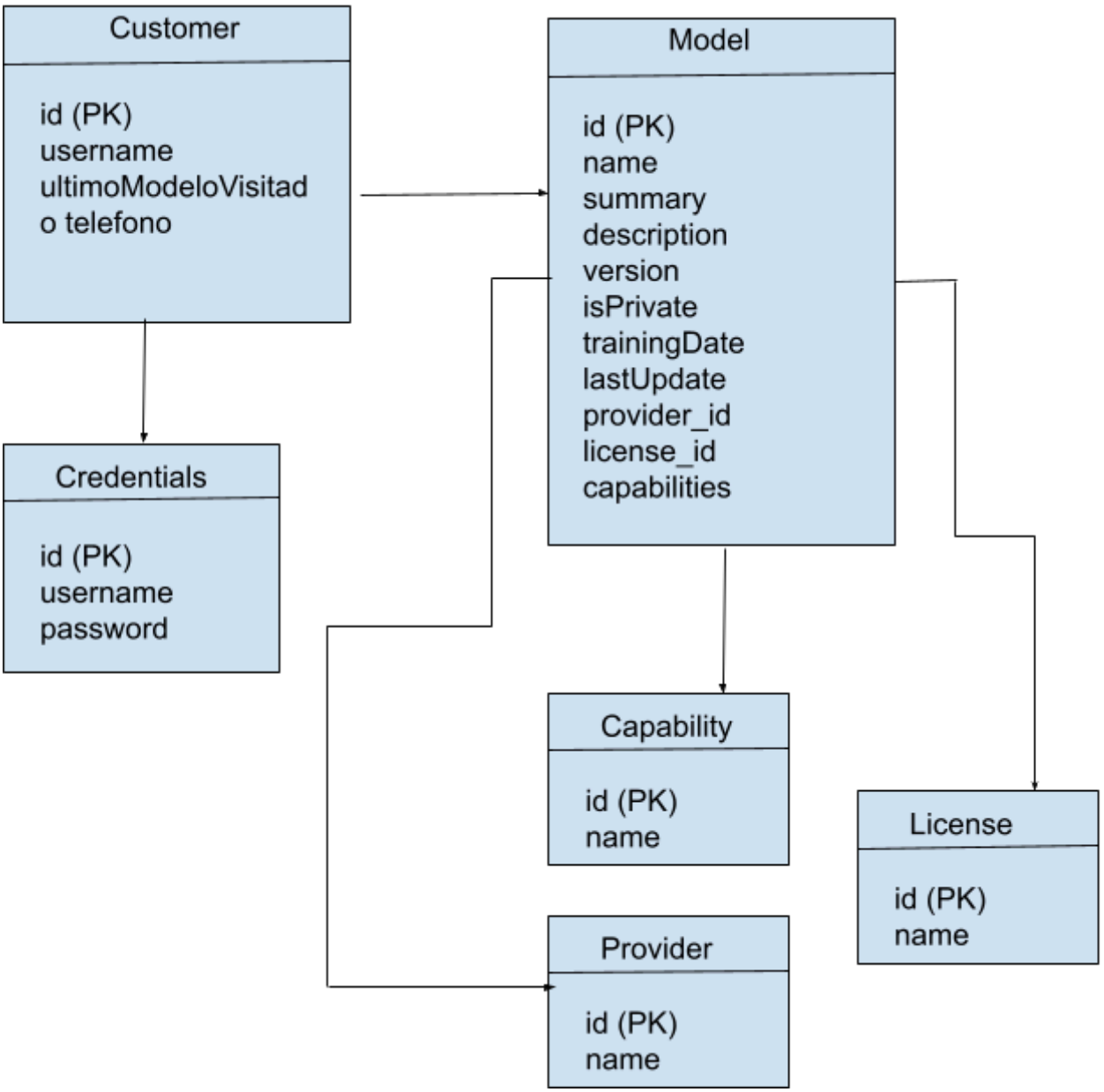
- Gestión de modelos LLM (listar, crear, modificar, eliminar)
- Gestión de usuarios/customers
- Filtrado avanzado por capabilities y providers
- Soporte HATEOAS (enlaces a recursos relacionados)

2. Estructura de la práctica

2.1. Estructura de directorios

2.2. Entidades JPA

2.2.1. Diagrama de clases



2.2.2. Descripción de las entidades

2.2.2.1. Model

La entidad **Model** representa un **modelo de lenguaje LLM** dentro del catálogo del sistema. Es el recurso principal gestionado por la API y contiene toda la información necesaria para describir un modelo generativo, su proveedor, sus capacidades y la metainformación relacionada.

Esta clase está anotada con @Entity y vinculada a una tabla relacional gestionada mediante JPA. Además, incorpora @XmlRootElement para permitir la serialización automática tanto en JSON como en XML dentro de los servicios REST.

Atributos:

- **id**: Identificador único del modelo. Es la clave primaria y se genera automáticamente mediante un @SequenceGenerator.
- **name**: Nombre del modelo (por ejemplo, *GPT-4.1-mini* o *Mistral Nemo*). Es un atributo obligatorio para registrar un modelo.
- **summary**: Resumen corto del modelo, utilizado en el listado general.
- **description**: Descripción extensa utilizada en la visualización detallada del modelo.
- **version**: Versión o fecha oficial del modelo.
- **isPrivate**: Indica si el modelo está protegido y solo puede consultarse mediante autenticación.
- **trainingDate**: Fecha aproximada del entrenamiento del modelo.
- **lastUpdateDate**: Fecha de la última actualización o revisión del modelo.

Relaciones:

- **provider (@ManyToOne)**: Cada modelo pertenece a un único proveedor (OpenAI, Mistral, Anthropic…).
  Esta relación utiliza FetchType.LAZY para cargar solo la información necesaria.
- **license (@ManyToOne)**: Indica la licencia bajo la cual se distribuye el modelo (Apache 2.0, Proprietary, Custom…).
  Permite distinguir modelos abiertos de modelos accesibles únicamente para usuarios registrados.

- **capabilities (@ManyToMany)**: Lista de capacidades soportadas por el modelo (por ejemplo, *chat-completion*, *code-generation*, *audio-generation*…).
  Esta relación se almacena mediante una tabla intermedia **model_capability**

2.2.2.2. Provider

La entidad **Provider** representa el **proveedor** de un modelo de lenguaje LLM dentro del catálogo de la aplicación. Un proveedor puede ser una organización o empresa que desarrolla modelos de inteligencia artificial, como *OpenAI*, *Mistral AI* o *Anthropic*. Esta entidad permite clasificar los modelos por procedencia y filtrar resultados en los servicios REST.

Está anotada con @Entity y @XmlRootElement, lo que permite su persistencia en la base de datos y su serialización directa en JSON o XML.

Atributos:

- **id**: Identificador único del proveedor.
  Se genera automáticamente mediante un @SequenceGenerator y actúa como clave primaria.

- **name**: Nombre del proveedor.
  Identifica la organización responsable de los modelos.

Relaciones

- **models (@OneToMany)**: Lista de modelos asociados al proveedor.
  Esta relación indica que un proveedor puede ofrecer varios modelos distintos.
  Se define como:

  - mappedBy = "provider" para que la relación sea bidireccional.

  - FetchType.LAZY para evitar cargar todos los modelos automáticamente.

  - @XmlTransient y @JsonbTransient para impedir que JAXB o JSON-B serialicen recursivamente la lista, evitando ciclos en JSON/XML.

2.2.2.3. Capability

La entidad **Capability** representa una **capacidad funcional** que puede poseer un modelo de lenguaje LLM dentro del catálogo. Las capacidades definen qué tipo de tareas es capaz de

realizar un modelo, como *chat-completion*, *code-generation*, *audio-generation*, *text-to-image* o *text-to-speech*. Esta entidad permite clasificar y filtrar modelos según su funcionalidad.

La clase está anotada con @Entity y @XmlRootElement, lo que facilita su persistencia mediante JPA y su serialización en los servicios REST.

Atributos:

- **id**: Identificador único de la capacidad.
  Es la clave primaria de la entidad y se genera automáticamente mediante un @SequenceGenerator.

- **name**: Nombre de la capacidad.

Relaciones:

- **models (@ManyToMany)**: Lista de modelos que soportan esta capacidad.
  Se trata de una relación bidireccional donde:
  - Cada modelo puede tener varias capacidades.
  - Cada capacidad puede estar asociada a múltiples modelos.

- La relación se marca con:
  - mappedBy = "capabilities" para indicar que el lado propietario está en la entidad Model.
  - FetchType.LAZY para evitar cargas innecesarias.
  - @XmlTransient y @JsonbTransient para impedir que se serialice la lista de modelos y evitar ciclos recursivos al generar JSON o XML.

2.2.2.4. License

La entidad License representa el **tipo de licencia** con el que se distribuye un modelo de lenguaje LLM dentro del catálogo del sistema. Permite clasificar los modelos según su régimen de uso (por ejemplo, *Apache-2.0*, *MIT*, *Proprietary*, *Custom*) y forma parte del conjunto de metadatos que describen cada modelo.

Está anotada con @Entity y @XmlRootElement, lo que posibilita su persistencia mediante JPA y su serialización en JSON y XML en los servicios REST.

Atributos:

- **id**: Identificador único de la licencia.
  Es la clave primaria de la entidad y se genera automáticamente mediante un @SequenceGenerator.

- **name**: Nombre de la licencia.
  Indica el tipo de permiso asociado al modelo, pudiendo ser una licencia abierta, restrictiva o personalizada. Este atributo permite identificar claramente el régimen legal del modelo.

Relaciones:

- **models (@OneToMany)**: Lista de modelos que están asociados a esta licencia.
  La relación se define como:
  - mappedBy = "license" para establecer la relación bidireccional con la entidad Model.
  - FetchType.LAZY para evitar la carga automática de todos los modelos relacionados.
  - @XmlTransient y @JsonbTransient para impedir que esta lista se serialice, evitando ciclos recursivos en JSON/XML.

2.2.2.5. Customer

La entidad **Customer** representa a un **usuario registrado** dentro del sistema. Cada customer está vinculado a unas credenciales de acceso y, opcionalmente, a un modelo LLM que ha visitado recientemente. Esta entidad se utiliza para gestionar usuarios de la API y permitir operaciones autenticadas mediante credenciales.

Está anotada con @Entity y @XmlRootElement, lo que permite su persistencia mediante JPA y su serialización en los servicios REST. También define consultas nombradas que permiten obtener todos los customers o buscar uno a partir de su nombre de usuario.

Atributos:

- **id**: Identificador único del customer.
  Es la clave primaria y se genera mediante un SequenceGenerator.

- **credentials (@OneToOne)**:
  Representa las credenciales asociadas al usuario (nombre de usuario y contraseña).
  Esta relación es **obligatoria**, ya que no puede existir un customer sin credenciales.
  Se mapea mediante la columna credentials_id.

- **ultimoModeloVisitado (@ManyToOne)**:
  Modelo LLM que el usuario ha visitado más recientemente.
  Es una relación opcional, ya que un customer nuevo puede no haber visitado ningún modelo todavía.

Se marca con @XmlTransient para evitar que se serialice el modelo completo en XML/JSON, exponiendo solo su ID mediante un método auxiliar.

- **telefono**:
  Número de teléfono opcional del customer.
  Es un dato adicional que no participa en relaciones con otras entidades.

Relaciones:

- **Con Credentials**
  La relación es @OneToOne, ya que cada usuario tiene un único par de credenciales, y cada credencial pertenece exactamente a un usuario.

- **Con Model**
  La relación es @ManyToOne, porque muchos customers pueden haber visitado el mismo modelo.
  Se utiliza para registrar la navegación del usuario y mejorar la interacción en la API (por ejemplo, recomendar o recordar el último recurso visitado).

2.3. Servicios REST

El proyecto implementa un conjunto de servicios RESTful para gestionar las entidades del sistema.

2.3.1. Servicio REST de Model (ModelFacadeREST)

El servicio está disponible en: /api/models

Funcionalidades principales

- Obtener todos los modelos, con o sin filtros.
- Filtrar por proveedor y capabilities.
- Obtener un modelo individual por ID.
- Crear un modelo nuevo.
- Modificar un modelo existente.
- Eliminar un modelo

2.3.1.1. GET /models

Este endpoint permite obtener todos los modelos o aplicar filtros opcionales:

- **GET /api/models** : Se devuelve la lista completa utilizando la NamedQuery Model.findAll.

- **GET /api/models?provider=X :** Se devuelve la lista filtrada por provider usando la NamedQuery Model.findByProvider.

- **GET /api/models?capability=A** y **GET /api/models?capability=A&capability=B :** El servicio soporta hasta dos capabilities.
  Si se envían más, se devuelve un 400 Bad Request. Para 1 o 2 capabilities, se genera dinámicamente una consulta JPQL con los JOIN necesarios sobre m.capabilities.

- **GET /api/models?capability=A&provider=B:** Soporta una mezcla de filtros.

2.3.1.2. GET /models/{id} – Consulta por ID

GET /api/models/{id}

Comportamiento:

- **200 OK** si el modelo existe y es público.
- **401 Unauthorized** si el modelo es privado y no se incluye cabecera Authorization.
- **404 Not Found** si el modelo no existe.

Este endpoint está protegido lógicamente: los modelos privados solo pueden verse con autenticación.

2.3.1.3. POST /models – Creación de un modelo

POST /api/models

Requiere autenticación.

Validaciones principales:

- El body no puede ser nulo.
- El nombre del modelo es obligatorio.

- El provider es obligatorio.
- Si el provider viene solo con el nombre, se intenta resolver en la base de datos.
- Si el provider no existe → **400 Bad Request.**
- La license es opcional, pero si se envía solo con el nombre se intenta resolver.

Resultado:

- Si todo es correcto, se crea el modelo.
- Se devuelve **201 Created**.

2.3.1.4. PUT /models/{id} – Actualización parcial

PUT /api/models/{id}

Requiere autenticación.

- Se permite actualizar solos los campos necesarios presentes en el body
- Se valida que al menos un campo sea válido.
- Si se envía el nombre vacío (`""`) → **400 Bad Request**.
- Si se envía un provider nuevo pero este no existe en la BD → error.

2.3.1.5. DELETE /models/{id} – Eliminación

DELETE /api/models/{id}

Requiere autenticación.

Comportamiento:

- Si el modelo existe → se elimina y devuelve **204 No Content**.
- Si no existe → **404 Not Found**.

2.3.2. Servicio REST de Customer (CustomerFacadeREST)

El servicio está disponible en: /api/customer

Como la entidad Customer contiene una relación **OneToOne** con Credentials, el servicio está diseñado para **no exponer nunca la contraseña** ni información sensible. Por este motivo, las respuestas JSON se construyen manualmente mediante **JsonObjectBuilder**, en lugar de devolver directamente la entidad JPA.

2.3.2.1. GET /customer – Listado de clientes

GET /api/customer

Este endpoint devuelve un array JSON con todos los clientes del sistema.
Para evitar exponer los campos de Credentials, el servicio solo incluye:

- id
- username (customer.getUsername())
- telefono (si existe)
- ultimoModeloVisitado (si existe), incluyendo:
  - id del modelo
  - nombre
  - link HATEOAS al recurso /models/{id}

La información se genera de forma manual usando JsonObjectBuilder.

Ejemplo de salida:

```
[
  {
    "id": 1,
    "username": "sob",
    "telefono": "123456789",
    "ultimoModeloVisitado": {
      "id": 3,
      "nombre": "Model A",
      "link": "/rest/api/v1/models/3"
    }
  }
]
```

2.3.2.2. GET /customer/{id} – Consulta individual

GET /api/customer/{id}

Este endpoint devuelve la información de un cliente concreto.

Si el cliente no existe → 404 Not Found.

De nuevo, el JSON se crea manualmente para evitar exponer credenciales.

2.3.2.3. PUT /customer/{id} – Actualización parcial

PUT /api/customer/{id}

Este endpoint requiere autenticación y permite dos únicas modificaciones:

Campos que se pueden modificar:

- telefono
- ultimoModeloVisitadoId

El servicio no permite modificar:

- username
- credentials
- la relación con Credentials
- otros campos no previstos

Si se incluye ultimoModeloVisitadoId, se comprueba que el modelo exista en BD. Si no existe → **404 Model not found**.

Comportamiento:

- Si se modifica al menos un campo → **204 No Content**
- Si no se modifica nada o el JSON no incluye claves válidas → **400 Bad Request**

3. Decisiones de Diseño

3.1. Recurso MODELS

Una de las primeras decisiones fue cómo organizar la información de los modelos. Teníamos **dos alternativas**:

Usar enumerados para cada campo, por ejemplo:

- Capabilities (CHAT_COMPLETION, CODE_GENERATION, etc.)
- Providers (OPENAI, MISTRAL, ANTHROPIC, etc.)
- Licenses (PROPRIETARY, APACHE_2_0, etc.)

La otra opción era **crear entidades separada**s con sus propias tablas en la base de datos y establecer relaciones JPA entre ellas, es mejor por varias razones:

1) Con enumerados, cada vez que se quisiera añadir un nuevo provider, capability o license, habría que modificar el código fuente, recompilar y redesplegar la aplicación.
2) Evitamos duplicación de información. Si 20 modelos son de "OpenAI", en vez de almacenar ese string 20 veces, almacenamos solo una referencia al proveedor.
3) Las relaciones JPA permiten hacer JOINs optimizados en la base de datos.

Para hacer esto, hemos implementado estas relaciones JPA:

- **Model a Provider (ManyToOne)**

  Varios modelos pueden pertenecer a un mismo provider (OpenAI tiene GPT-4, GPT-4-turbo, GPT-3.5, etc.)

  Es unidireccional porque desde Model necesitamos saber su provider, pero desde Provider no nos interesa listar todos sus modelos en la mayoría de consultas

- **Model → Capability (ManyToMany)**

  Un modelo puede tener múltiples capabilities (GPT-4 tiene chat-completion, code-generation, image-generation)

  Una capability puede estar en múltiples modelos (chat-completion está en GPT-4, Claude, Mistral, etc.)

  Requiere tabla intermedia `**model_capability**` que JPA gestiona automáticamente

- **Model → License (ManyToOne)**

  Similar a Provider: muchos modelos pueden tener la misma license, pero una license solo pertenece a un modelo a la vez.

Acerca de los métodos, ya hemos mencionado en el punto anterior los que eran obligatorios:

1) **GET /rest/api/v1/models (con filtros opcionales)**

Este fue uno de los métodos más complejos porque debía soportar múltiples combinaciones de filtros:

- Sin filtros: devolver todos los modelos
- Solo provider: filtrar por proveedor
- Una capability: filtrar por esa capability
- Dos capabilities: filtrar modelos que tengan AMBAS
- Capability + provider: combinación de ambos

Para hacer las consultas necesarias usamos JPQL dinámico, ya que en el enunciado se pide: "El filtratge s'ha de fer mitjançant una consulta a la base de dades. No s'acceptarà com a vàlid retornar el llistat de tots els models i filtrar-los de forma programàtica amb Java."

```
StringBuilder jpql = new StringBuilder("SELECT DISTINCT m FROM Model m ");

if (capabilities.size() == 1) {

    jpql.append("JOIN m.capabilities c0 WHERE LOWER(c0.name) =
LOWER(:cap0)");

    if (provider != null && !provider.trim().isEmpty()) {

        jpql.append(" AND LOWER(m.provider.name) = LOWER(:provider)");

    }

} else {

    jpql.append("JOIN m.capabilities c0 JOIN m.capabilities c1 ");

    jpql.append("WHERE LOWER(c0.name) = LOWER(:cap0) ");

    jpql.append("AND LOWER(c1.name) = LOWER(:cap1)");

}
```

2) **GET /rest/api/v1/models/{id}**

Retorna los detalles completos de un modelo. El enunciado decía: "Si el model és privat, només es podrà retornar si l'usuari està registrat (autentificat)."

Para eso verificamos el campo isPrivate del modelo y, si es true, comprobamos que haya un header de Authorization. Si no lo hay, retornamos **401 Unauthorized.**

3) **POST /rest/api/v1/models**

Se ha implementado todo lo solicitado (comprobar que el proveedor indicado existe, que las capacidades sean válidas y que los parámetros tengan valores razonables)

En este método hemos añadido una mejora adicional para buscar de manera automática si el Provider y License ya existen. Es decir, si en el POST nos llega un provider o license con solo el nombre (sin ID), intentamos buscarlo en la base de datos antes de crear uno nuevo.

Ahora, hablando de los métodos opcionales

4) **PUT /rest/api/v1/models/{id}**

Este método permite actualizar un modelo existente. Lo implementamos validando que el modelo exista, que se proporcione al menos un campo para actualizar que si se intenta cambiar el nombre a vacío salte error.

También implementamos la misma lógica de búsqueda inteligente de Provider/License que en POST.

5) **DELETE /rest/api/v1/models/{id}**

Permite eliminar un modelo del catálogo. Sobre la implementación no tenemos mucho que mencionar. Es un metodo seguro.

```
@DELETE
@Path("{id}")
@Secured
public Response remove(@PathParam("id") Long id) {
    try {
        Model existing = super.find(id);
        if (existing == null) {
            return Response.status(Response.Status.NOT_FOUND)
                    .entity("Model not found")
                    .build();
        }

        super.remove(existing);
        return Response.noContent().build();
    } catch (Exception e) {
        return Response.status(Response.Status.INTERNAL_SERVER_ERROR)
                .entity(e.getMessage())
                .build();
```

```
        }
    }
```

3.2. Recurso CUSTOMERS

En esta caso, como ya se ha mencionado un poco en el punto anterior, teníamos varias opciones para gestionar la clase customer.

Originalmente, habíamos pensado en gestionar customers por separado, sin ningún tipo de relación con los usuarios que tenían una credencial.

Creando los campos de correo, username, todo dentro de una nueva clase.

Pero eso no era muy eficiente, ya que por cada usuario autentificado teníamos datos duplicados. Así que como la única manera de que el servidor sepa quién es la persona que acaba de acceder es autentificándose y además queremos proteger la contraseña a toda costa así que es inteligente que lógica esté separada, por un lado, los datos privados, como la contraseña y por el otro los datos menos delicados.

Como ya almacenamos el username en el otro lado, simplemente era cuestión de añadir una clave foránea hacia la inserción de Credentials y obtener el username o los datos que necesitemos (no la contraseña) llamando a sus métodos propios.

Para hacer esto hemos creado una relación unidireccional hacia Credentials del tipo @OneToOne (un customer solo tiene una credencial).

Otra decisión importante que tomamos fue decidir como guardar el último modelo visitado.

Una opción era por cada customer guardar el nombre del modelo y nada más. La problemática de esto, además de los datos duplicados, es que si queríamos saber más cosas de ese modelo, deberíamos hacer una búsqueda sobre todos los modelos para saber cuál es.

Así que decidimos también guardar una referencia al modelo con otra clave foránea. En esta caso también es unidireccional, ya que desde modelo no nos interesa saber qué customers le han visitado. En este caso la relación es @ManyToOne, ya que un modelo puede ser visitado por varios customers.

Además, hay otra razón muy simple para implementarlo así. El enunciado dice:

“Cal indicar l'enllaç a l'últim model que va consultar per suportar el principi de HATEOAS. Per exemple, en JSON:

```
"links": {
   "model": "/models/12345"
        }
```

Aquesta crida no pot retornar informació confidencial, p. ex., la contrasenya d'aquest usuari.”

En lo que respecta a la última línea ningún problema, ya que en customers no guardamos la contraseña, así que es imposible que la mandemos. No hay que filtrar nada.

La parte de enviar el “puntero” del modelo también está solucionado en parte, solo quedaba que soportara HATEOAS (enviarlo como un link).

Para ello hemos cambiado la forma de pasar la respuesta, en vez de hacerlo “fácil” con…

```
Model model = super.find(id);

return Response.ok(model).build();
```

Hemos creado un JsonObjectBuilder builder = Json.createObjectBuilder() y con instrucciones .add() hemos añadido el campo “link”.

```
JsonObject modelInfo = Json.createObjectBuilder()

        .add("id", modelo.getId())

        .add("nombre", modelo.getName())

        .add("link", "/rest/api/v1/models/" + ultimoModeloId)

        .build();
```

Así que hasta ahora, en lo que respecta a atributos tenemos el atributo id que es automático, el “puntero” a credenciales, otro a modelo y por último añadimos, para tener algún tipo de información extra, además de para poder hacer más tests con el PUT, el atributo teléfono que es un String, ya que un teléfono puede contener caracteres diferentes a números (+, -, *).

Y, lo que nos queda por mencionar es el servicio PUT /customer/{id} que es una función adicional, pero hemos decidido implementarla.

Este servicio permite actualizar solo los campos seguros (telefono y ultimoModeloVisitado).

Para hacerlo nos hemos basado en los PUTs que ya habíamos hecho en los métodos de los servicios de Model.

Lo interesante de este PUT es que había que revisar que valor se estaba intentando modificar. Puede ser el teléfono o el modelo, además de que pueden llegarnos los dos a la vez. Para eso hemos implementado un bucle que obtiene la .keySet() del JSON que nos entra y miramos si es alguno de los dos.

```java
// El JSON puede contener varios cambios (saber que valor contiene)
for (String key : inputJson.keySet()) {
    switch (key) {
    case "telefono":

        // Obtener teléfono a modificar
        String nuevoTelefono = inputJson.getString("telefono");
        existing.setTelefono(nuevoTelefono);
        modificado = true;
        break;

    case "ultimoModeloVisitadoId":

        // Obtener modelo a modificar
        Long modeloId = inputJson.getJsonNumber("ultimoModeloVisitadoId").longValue();
        Model modelo = em.find(Model.class, modeloId);

        if (modelo == null)
            return Response.status(Response.Status.NOT_FOUND).entity("{\"error\": \"Model con id " + modeloId + " no encontrado\"}").build();

        existing.setUltimoModeloVisitado(modelo);
        modificado = true;
        break;

    default:
        break;
    }
}
```

3.3. Filtrado de Responses

Como parte extra, se ha creado una clase nueva que funciona como un observador entre las respuestas de los servicios y el mensaje real que sale del servidor. El archivo **ErrorResponseFilter.java** se encuentra en la carpeta **service**.

Esta clase, llamada **ErrorResponseFilter**, implementa **ContainerResponseFilter**, que es una interfaz que proporciona **jakarta.ws.rs.container.ContainerResponseFilter**.

La clase tiene tres métodos:

1) public void **filter**(ContainerRequestContext r1, ContainerResponseContext r2)
2) private String **getHttpStatusText**(int status)
3) private String **getDefaultMessageForStatus**(int status)

El método principal, filter(), obtiene del responseContext el status. Los casos que nosotros no solemos gestionar son los errores, los cuales están más allá de los mensajes 400 y los 500. Por tanto, nos centramos en todos esos. Los 200 OK u otros de este tipo no los revisamos.

Primero se revisa si ese tipo de contenido que nos llega es diferente de JSON (en nuestro caso mayoritariamente serán HTML) y revisa si se podrá transformar.

De ser así, formamos el nuevo mensaje de error en 3 pasos.

1) Obtener los Strings que queremos mostrar dependiendo del código de error recibido. Tanto el título de error, como el mensaje. (Usamos las llamadas a las otras dos funciones que devuelven en función del status un String)
2) Después formamos el archivo JSON aprovechando el método .format() del tipo String y añadimos todos los datos.
3) Modificamos la respuesta añadiendo este nuevo texto dentro de la respuesta y modificamos los headers para que se interprete como un JSON.

Es importante mencionar que no hace falta hacer nada mas, ya que la clase RESTapp, al no tener ninguna clase inicializada en el interior hace una exploración automática de las clases que se necesitarán. Para asegurarnos de que se leerá nuestra nueva clase la hemos marcado con la anotación: `@Priority(Priorities.USER)`

Función principal:

```java
@Override
public void filter(ContainerRequestContext requestContext, ContainerResponseContext responseContext) throws IOException {
    int status = responseContext.getStatus();

    // Solo procesar si es un error (4xx o 5xx)
    if (status >= 400) {
        // Verificar si ya es JSON
        MediaType mediaType = responseContext.getMediaType();
        if (mediaType == null || !mediaType.isCompatible(MediaType.APPLICATION_JSON_TYPE)) {
            // Convertir a JSON
            String errorTitle = getHttpStatusText(status);
            String message = getDefaultMessageForStatus(status);

            String jsonError = String.format(
                "{\"timestamp\":\"%s\",\"status\":%d,\"error\":\"%s\",\"message\":\"%s\",\"path\":\"%s\"}",
                Instant.now().toString(),
                status,
                errorTitle,
                message,
                requestContext.getUriInfo().getPath()
            );

            responseContext.setEntity(jsonError);
            responseContext.getHeaders().putSingle("Content-Type", MediaType.APPLICATION_JSON);
        }
    }
}
```

Funciones privadas:

```java
private String getHttpStatusText(int status) {
    switch (status) {
        case 400: return "Bad Request";
        case 401: return "Unauthorized";
        case 403: return "Forbidden";
        case 404: return "Not Found";
        case 405: return "Method Not Allowed";
        case 409: return "Conflict";
        case 500: return "Internal Server Error";
        case 503: return "Service Unavailable";
        default: return status >= 500 ? "Server Error" : "Client Error";
    }
}

private String getDefaultMessageForStatus(int status) {
    switch (status) {
        case 400: return "The request could not be understood";
        case 401: return "Authentication is required";
        case 403: return "You don't have permission to access this resource";
        case 404: return "The requested resource was not found";
        case 500: return "An internal server error occurred";
        default: return "An error occurred";
    }
}
```

4. Juegos de pruebas realizados

4.1. Scripts bash

En un primer momento, durante el desarrollo inicial, automatizamos ciertos tests con scripts en bash. Estos, mediante operaciones curl, permitían testear algunos aspectos que necesitábamos en ese momento. *Estos tests se desarrollaron con la ayuda de IA.*

Este script de manera automática poblaba la base de datos con las inserciones de **install.jsp**, esto era una de las cosas opcionales que se mencionaba en el enunciado de la práctica:

(Opcionalment) Executar scripts per crear la base de dades

```
# Función para poblar la base de datos
populate_database() {
    print_header "POBLANDO BASE DE DATOS"

    echo "Ejecutando install.jsp..."
    local response=$(curl -s "$INSTALL_URL")

    if echo "$response" | grep -q "INSERT"; then
        echo -e "${GREEN}✓${NC} Base de datos poblada correctamente"
    else
        echo -e "${YELLOW}!${NC} No se pudo verificar la población de datos"
        echo "Respuesta: $response"
    fi

    # Esperar un momento para que los datos se persistan
    sleep 2
}
```

También verificaba que todo estuviera corriendo:

```
# Función para verificar si el servidor está corriendo
check_server() {
    print_header "VERIFICANDO SERVIDOR"

    if curl -s -f "$BASE_URL/customer" > /dev/null 2>&1; then
        echo -e "${GREEN}✓${NC} Servidor GlassFish está corriendo"
        return 0
    else
        echo -e "${RED}✗${NC} Servidor GlassFish no está accesible"
        echo -e "${YELLOW}→${NC} Revisa que GlassFish este corriendo y la app esté deploy"
        exit 1
    fi
}
```

Simplificaba los tests y llevaba control de los resultados:

```
# Configuración
BASE_URL="http://localhost:8080/practica-sob/rest/api/v1"
INSTALL_URL="http://localhost:8080/practica-sob/install.jsp"

# Contadores
TESTS_PASSED=0
TESTS_FAILED=0
TESTS_TOTAL=0
```

Y printeaba los resultados de manera “amigable”

```
# Función para imprimir resultado del test
print_result() {
    local test_name="$1"
    local result="$2"
    local details="$3"

    TESTS_TOTAL=$((TESTS_TOTAL + 1))

    if [ "$result" = "PASS" ]; then
        echo -e "${GREEN}✓ PASS${NC} - $test_name"
        [ -n "$details" ] && echo -e "  ${YELLOW}→${NC} $details"
        TESTS_PASSED=$((TESTS_PASSED + 1))
    else
        echo -e "${RED}✗ FAIL${NC} - $test_name"
        [ -n "$details" ] && echo -e "  ${RED}→${NC} $details"
        TESTS_FAILED=$((TESTS_FAILED + 1))
    fi
}
```

Este es el main del script:

```
main() {

    # Verificaciones previas
    check_dependencies
    check_server

    # Preguntar si poblar la base de datos
    echo -e "\n${YELLOW}¿Deseas repoblar la base de datos antes de ejecutar los tests?${NC}"
    echo -e "${YELLOW}Esto ejecutará install.jsp y reiniciará todos los datos.${NC}"
    read -p "Repoblar BD? (s/N): " -n 1 -r
    echo
    if [[ $REPLY =~ ^[SsYy]$ ]]; then
        populate_database
    fi

    # Ejecutar tests
    test_get_all_customers
    test_get_customer_by_id
    test_get_customer_not_found
    test_put_customer_unauthorized
    test_put_customer_update_telefono
    test_put_customer_update_modelo
    test_put_customer_invalid_modelo
    test_put_customer_empty_json

    # Resumen final
    print_header "RESUMEN DE RESULTADOS"

    echo -e "Total de tests ejecutados: ${BLUE}$TESTS_TOTAL${NC}"
    echo -e "Tests exitosos: ${GREEN}$TESTS_PASSED${NC}"
    echo -e "Tests fallidos: ${RED}$TESTS_FAILED${NC}"

    if [ $TESTS_FAILED -eq 0 ]; then
        echo -e "\n${GREEN}╔═══════════════════════════════════════╗${NC}"
        echo -e "${GREEN}║  ✓ TODOS LOS TESTS PASARON            ║${NC}"
        echo -e "${GREEN}╚═══════════════════════════════════════╝${NC}\n"
        exit 0
    else
        echo -e "\n${RED}╔═══════════════════════════════════════╗${NC}"
        echo -e "${RED}║  ✗ ALGUNOS TESTS FALLARON             ║${NC}"
        echo -e "${RED}╚═══════════════════════════════════════╝${NC}\n"
        exit 1
    fi
}
```

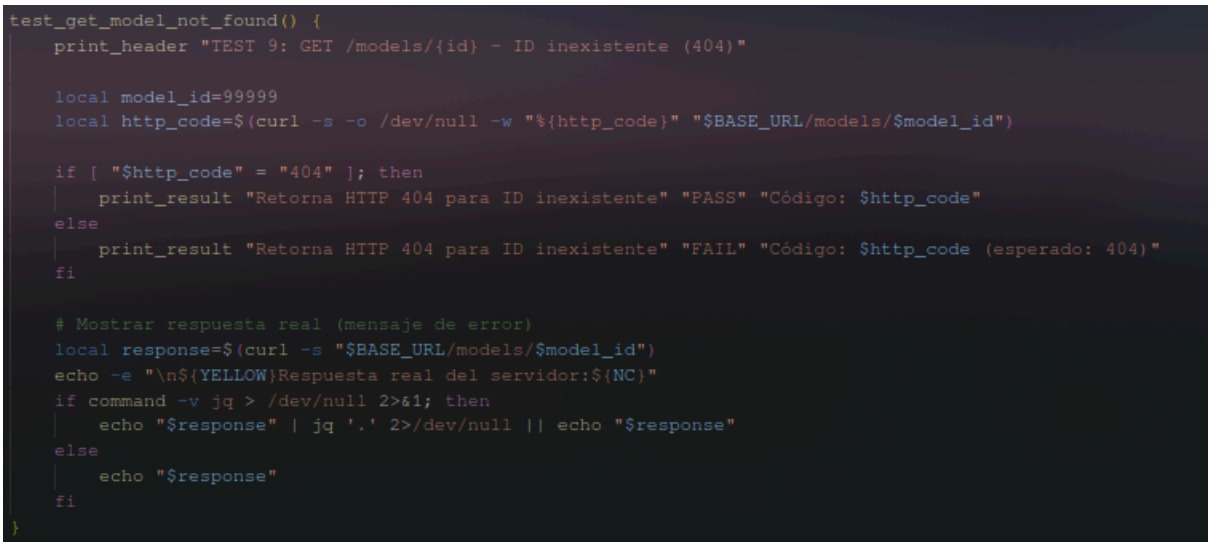
Aquí dejamos algún ejemplo de los tests para cada modelo:

4.1.1. Models

GET /models?capability={name} (filtrar por 1 capability)

```
test_get_models_by_capability() {
    print_header "TEST 3: GET /models?capability=chat-completion - Filtrar por capability"

    local capability="chat-completion"
    local response=$(curl -s "$BASE_URL/models?capability=$capability")
    local http_code=$(curl -s -o /dev/null -w "%{http_code}" "$BASE_URL/models?capability=$capability")

    # Verificar código HTTP 200
    if [ "$http_code" = "200" ]; then
        print_result "GET /models?capability retorna HTTP 200" "PASS" "Código: $http_code"
    else
        print_result "GET /models?capability retorna HTTP 200" "FAIL" "Código: $http_code"
        return
    fi

    # Verificar que hay resultados
    local count=$(echo "$response" | jq '. | length' 2>/dev/null)
    if [ "$count" -ge 1 ]; then
        print_result "Contiene modelos con capability $capability" "PASS" "Total: $count modelos"
    else
        print_result "Contiene modelos con capability $capability" "FAIL" "Total: $count modelos"
    fi

    # Verificar que todos tienen la capability
    local all_have_cap=$(echo "$response" | jq -e --arg cap "$capability" 'all(.capabilities | map(.name) | contains([$cap]))' 2>/dev/null)
    if [ "$all_have_cap" = "true" ]; then
        print_result "Todos los modelos tienen la capability" "PASS" "✓ Filtro correcto"
    else
        print_result "Todos los modelos tienen la capability" "FAIL" "⚠ Filtro incorrecto"
    fi

    # Mostrar respuesta real
    echo -e "\n${YELLOW}Respuesta real del servidor:${NC}"
    if command -v jq > /dev/null 2>&1; then
        echo "$response" | jq '.' 2>/dev/null || echo "$response"
    else
        echo "$response"
    fi
}
```

GET /models/{id} - ID inexistente (debe retornar 404)



```
test_get_model_not_found() {
    print_header "TEST 9: GET /models/{id} - ID inexistente (404)"

    local model_id=99999
    local http_code=$(curl -s -o /dev/null -w "%{http_code}" "$BASE_URL/models/$model_id")

    if [ "$http_code" = "404" ]; then
        print_result "Retorna HTTP 404 para ID inexistente" "PASS" "Código: $http_code"
    else
        print_result "Retorna HTTP 404 para ID inexistente" "FAIL" "Código: $http_code (esperado: 404)"
    fi

    # Mostrar respuesta real (mensaje de error)
    local response=$(curl -s "$BASE_URL/models/$model_id")
    echo -e "\n${YELLOW}Respuesta real del servidor:${NC}"
    if command -v jq > /dev/null 2>&1; then
        echo "$response" | jq '.' 2>/dev/null || echo "$response"
    else
        echo "$response"
    fi
}
```

POST /models - Validación de campos obligatorios (debe retornar 400)

```bash
test_post_model_validation() {
    print_header "TEST 12: POST /models - Validación campos obligatorios (400)"

    # Intentar crear modelo sin nombre
    local http_code=$(curl -s -o /dev/null -w "%{http_code}" -u sob:sob \
        -X POST \
        -H "Content-Type: application/json" \
        -d '{"provider": {"name": "Test"}}' \
        "$BASE_URL/models")

    if [ "$http_code" = "400" ]; then
        print_result "Retorna HTTP 400 sin nombre" "PASS" "Código: $http_code (✓ Validación correcta)"
    else
        print_result "Retorna HTTP 400 sin nombre" "FAIL" "Código: $http_code (esperado: 400)"
    fi

    # Intentar crear modelo sin provider
    local http_code2=$(curl -s -o /dev/null -w "%{http_code}" -u sob:sob \
        -X POST \
        -H "Content-Type: application/json" \
        -d '{"name": "Test Model"}' \
        "$BASE_URL/models")

    if [ "$http_code2" = "400" ]; then
        print_result "Retorna HTTP 400 sin provider" "PASS" "Código: $http_code2 (✓ Validación correcta)"
    else
        print_result "Retorna HTTP 400 sin provider" "FAIL" "Código: $http_code2 (esperado: 400)"
    fi

    # Mostrar respuesta real del primer caso (sin nombre)
    local response=$(curl -s -u sob:sob \
        -X POST \
        -H "Content-Type: application/json" \
        -d '{"provider": {"name": "Test"}}' \
        "$BASE_URL/models")
    echo -e "\n${YELLOW}Respuesta real del servidor (sin nombre):${NC}"
    if command -v jq > /dev/null 2>&1; then
        echo "$response" | jq '.' 2>/dev/null || echo "$response"
    else
        echo "$response"
    fi
}
```

4.1.2. Customers

GET /customer/{id} con ID inexistente (debe retornar 404)

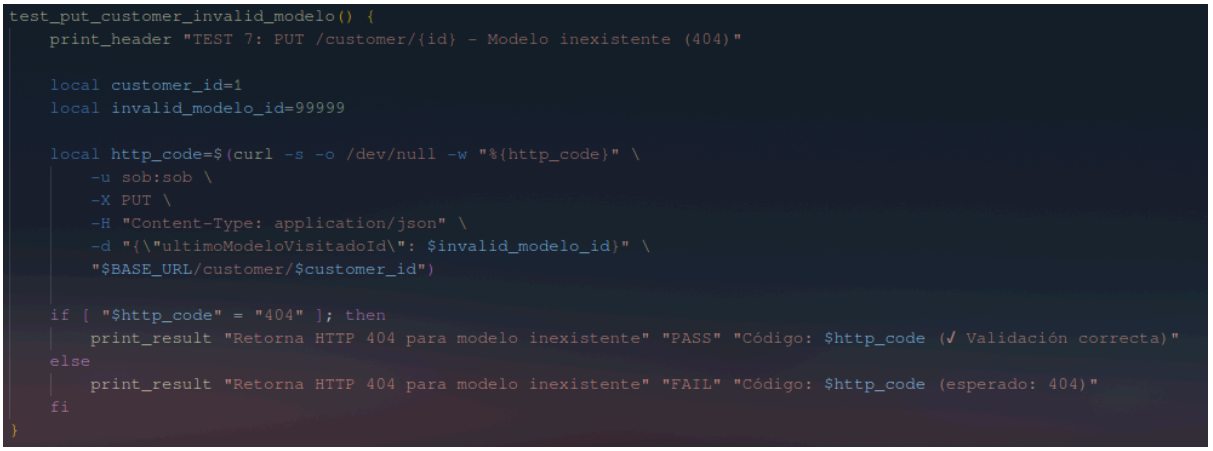
```
test_get_customer_not_found() {
    print_header "TEST 3: GET /customer/{id} - ID inexistente (404)"

    local customer_id=99999
    local http_code=$(curl -s -o /dev/null -w "%{http_code}" "$BASE_URL/customer/$customer_id")

    if [ "$http_code" = "404" ]; then
        print_result "Retorna HTTP 404 para ID inexistente" "PASS" "Código: $http_code"
    else
        print_result "Retorna HTTP 404 para ID inexistente" "FAIL" "Código: $http_code (esperado: 404)"
    fi
}
```

PUT /customer/{id} con autenticación - actualizar teléfono

```
test_put_customer_update_telefono() {
    print_header "TEST 5: PUT /customer/{id} - Actualizar teléfono (autenticado)"

    local customer_id=1
    local new_telefono="+34611222333"

    # Realizar PUT con autenticación
    local http_code=$(curl -s -o /dev/null -w "%{http_code}" \
        -u sob:sob \
        -X PUT \
        -H "Content-Type: application/json" \
        -d "{\"telefono\": \"$new_telefono\"}" \
        "$BASE_URL/customer/$customer_id")

    if [ "$http_code" = "204" ]; then
        print_result "PUT retorna HTTP 204 (No Content)" "PASS" "Código: $http_code"
    else
        print_result "PUT retorna HTTP 204 (No Content)" "FAIL" "Código: $http_code"
        return
    fi

    # Verificar que el teléfono se actualizó
    sleep 1
    local response=$(curl -s "$BASE_URL/customer/$customer_id")
    local updated_telefono=$(echo "$response" | jq -r '.telefono' 2>/dev/null)

    if [ "$updated_telefono" = "$new_telefono" ]; then
        print_result "Teléfono actualizado correctamente" "PASS" "Nuevo teléfono: $updated_telefono"
    else
        print_result "Teléfono actualizado correctamente" "FAIL" "Esperado: $new_telefono, Obtenido: $updated_telefono"
    fi

    # Restaurar teléfono original
    curl -s -u sob:sob \
        -X PUT \
        -H "Content-Type: application/json" \
        -d '{"telefono": "+34612345678"}' \
        "$BASE_URL/customer/$customer_id" > /dev/null
}
```

PUT /customer/{id} - modelo inexistente (debe retornar 404)

Como se puede observar, en este punto de la práctica aún no habíamos filtrado que los resultados siempre fueran JSON, aquí se comprobaba un texto en HTML.



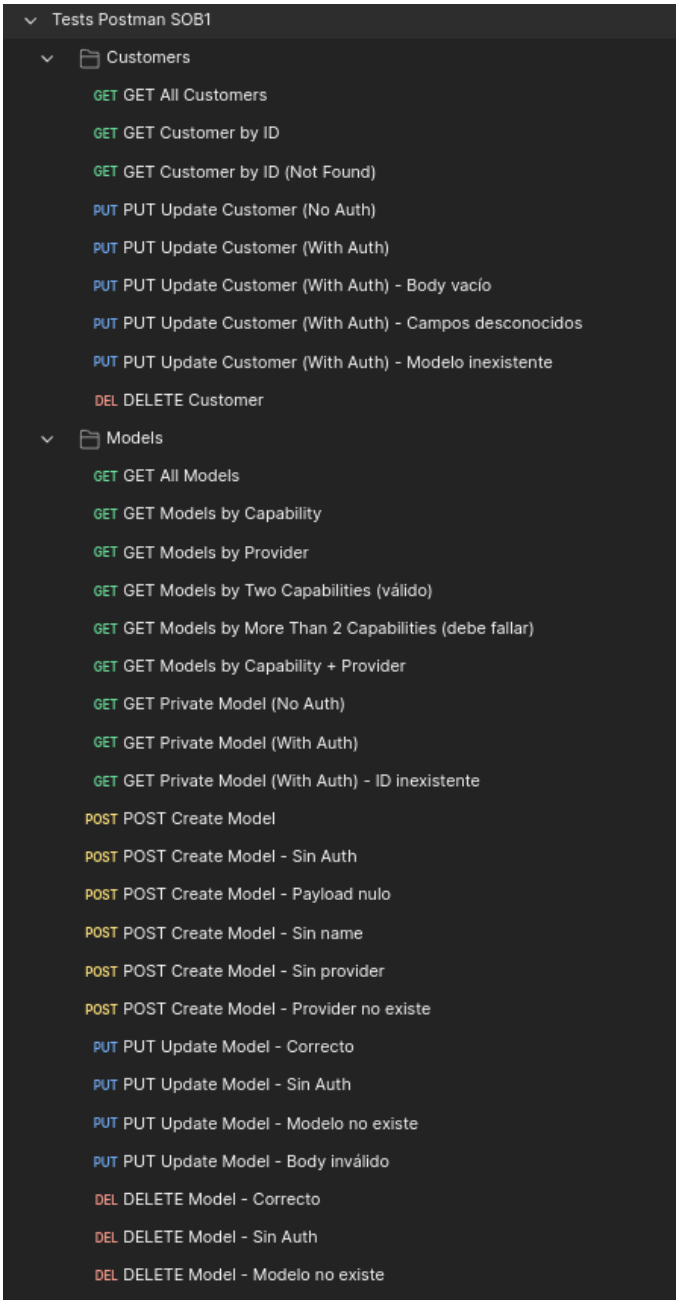
```
test_put_customer_invalid_modelo() {
    print_header "TEST 7: PUT /customer/{id} - Modelo inexistente (404)"

    local customer_id=1
    local invalid_modelo_id=99999

    local http_code=$(curl -s -o /dev/null -w "%{http_code}" \
        -u sob:sob \
        -X PUT \
        -H "Content-Type: application/json" \
        -d "{\"ultimoModeloVisitadoId\": $invalid_modelo_id}" \
        "$BASE_URL/customer/$customer_id")

    if [ "$http_code" = "404" ]; then
        print_result "Retorna HTTP 404 para modelo inexistente" "PASS" "Código: $http_code (✓ Validación correcta)"
    else
        print_result "Retorna HTTP 404 para modelo inexistente" "FAIL" "Código: $http_code (esperado: 404)"
    fi
}
```

4.2. Tests de Postman

Para simplificar el trabajo, tener una interfaz más amigable y hacer más visual la ejecución de los tests, utilizamos, tal y como se recomendaba, Postman. Estos tests los creamos manualmente en la propia interfaz de Postman. Esta es la estructura de la colección:

- Carpeta "Customers" (9 tests)
- Carpeta "Models" (23 tests)

Estructura de los tests:



En cada test probamos diversas cosas:

- Cambiamos el tipo de método (GET, POST, PUT, DELETE…)
- Modificar el campo Auth
  - No Auth
  - Basic Auth (username: sob; password: sob)
- Añadir texto al Body (tipo raw: JSON)
- Añadir tests en el campo Scripts con pm.test()
  - Controlar el codigo de salida
  - Revisar los parametros devueltos
  - Revisar el formato de la respuesta (aquí todos los errores ya se reciben como JSON, no permitimos respuestas HTML)

Tests:

4.2.1. Models

Test 6: GET Models by Capability + Provider

➔ Request: `GET {{baseUrl}}/models?capability=chat-completion&provider=openai`

➔ Tests:

```
pm.test("Status 200", () => pm.response.to.have.status(200));


pm.test("Filtra por capability y provider", () => {
    const data = pm.response.json();
    data.forEach(m => {
        const caps = m.capabilities.map(c => c.name.toLowerCase());
        pm.expect(caps).to.include("chat-completion");
        pm.expect(m.provider.name.toLowerCase()).to.eql("openai");
    });
});
```

➔ Resultado esperado: Modelos de OpenAI con chat-completion

Test 7: GET Private Model (No Auth)

➔ Request: `GET {{baseUrl}}/models/1 (sin Authorization header)`

➔ Tests:

```
pm.test("401 Unauthorized", () => pm.response.to.have.status(401));

pm.test("Debe incluir WWW-Authenticate", () => {

    pm.expect(pm.response.headers.has("WWW-Authenticate")).to.be.true;

});
```

➔ Resultado esperado: `401 Unauthorized con WWW-Authenticate header`

Test 8: GET Private Model (With Auth)

➔ Request: `GET {{baseUrl}}/models/1 (Authorization Basic sob:sob)`

➔ Tests:

```
pm.test("Status code is 200", function () {

    pm.response.to.have.status(200);

});
```

➔ Resultado esperado: `200 OK con datos del modelo`

Test 9: GET Private Model (With Auth) - ID inexistente

➔ Request: `GET {{baseUrl}}/models/99999 (Authorization Basic sob:sob)`

➔ Tests:

```
pm.test("Status 404", () => pm.response.to.have.status(404));


pm.test("Body contiene 'Model not found'", () => {

    pm.expect(pm.response.text()).to.include("Model not found");

});
```

➔ Resultado esperado: 404 Not Found

Test 10: POST Create Model

➔ Request: `POST {{baseUrl}}/models (Authorization Basic: sob:sob)`

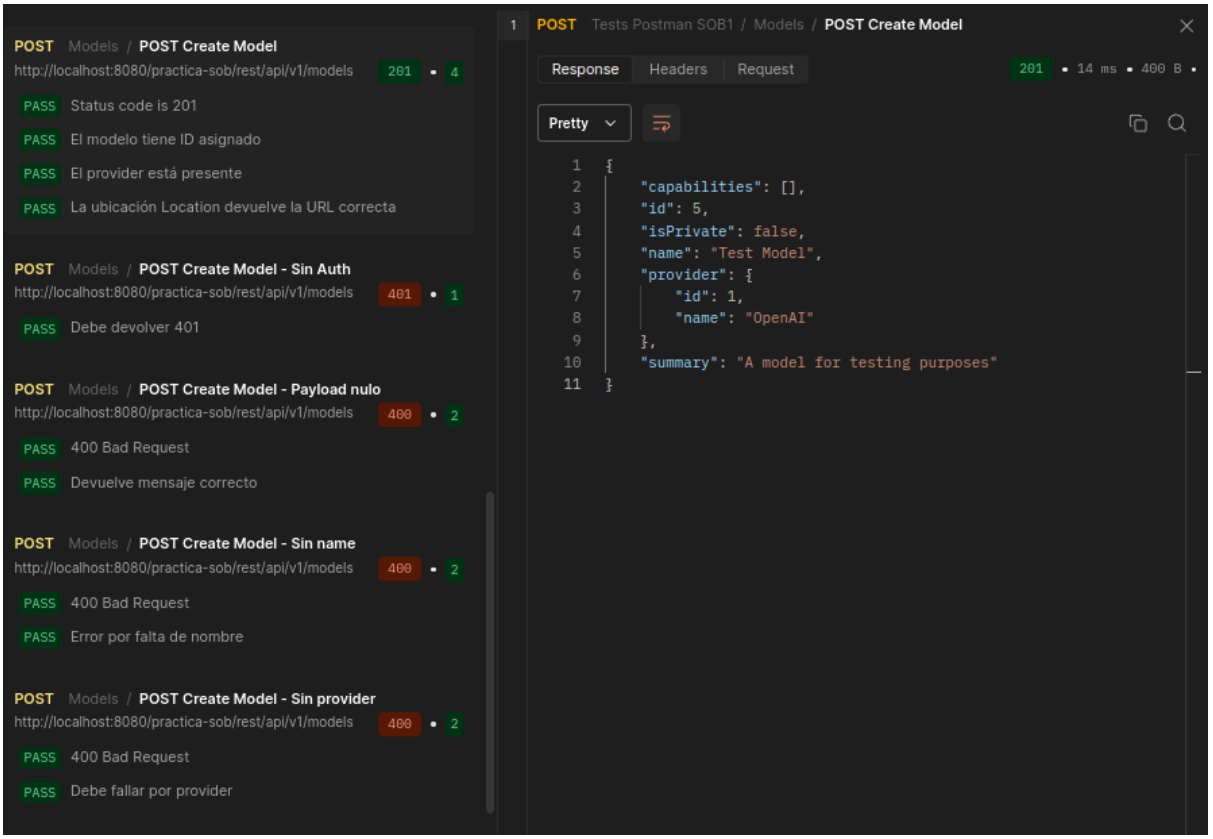
Body:

```
{
  "name": "Test Model",
  "provider": {
    "name": "OpenAI"
  },
  "summary": "A model for testing purposes",
  "isPrivate": false
}
```

➔ Tests:

```
pm.test("Status code is 201", function () {
    pm.response.to.have.status(201);
});
const response = pm.response.json();
pm.collectionVariables.set("newModelId", response.id);
pm.test("El modelo tiene ID asignado", () => {
    const body = pm.response.json();
    pm.expect(body.id).to.be.a("number");
});
pm.test("El provider está presente", () => {
    const body = pm.response.json();
    pm.expect(body.provider).to.be.an("object");
});
pm.test("La ubicación Location devuelve la URL correcta", () => {
    const location = pm.response.headers.get("Location");
    pm.expect(location).to.include("/models/");
});
```

➔ Resultado esperado: `201 Created con Location header`


POST Models / POST Create Model
http://localhost:8080/practica-sob/rest/api/v1/models 201 • 4
PASS Status code is 201
PASS El modelo tiene ID asignado
PASS El provider está presente
PASS La ubicación Location devuelve la URL correcta
POST Models / POST Create Model - Sin Auth
http://localhost:8080/practica-sob/rest/api/v1/models 401 • 1
PASS Debe devolver 401
POST Models / POST Create Model - Payload nulo
http://localhost:8080/practica-sob/rest/api/v1/models 400 • 2
PASS 400 Bad Request
PASS Devuelve mensaje correcto
POST Models / POST Create Model - Sin name
http://localhost:8080/practica-sob/rest/api/v1/models 400 • 2
PASS 400 Bad Request
PASS Error por falta de nombre
POST Models / POST Create Model - Sin provider
http://localhost:8080/practica-sob/rest/api/v1/models 400 • 2
PASS 400 Bad Request
PASS Debe fallar por provider
POST Tests Postman SOB1 / Models / POST Create Model
Response Headers Request
201 • 14 ms • 400 B
Pretty
{
"capabilities": [],
"id": 5,
"isPrivate": false,
"name": "Test Model",
"provider": {
"id": 1,
"name": "OpenAI"
},
"summary": "A model for testing purposes"
}

Test 16: PUT Update Model

➔ Request: PUT {{baseUrl}}/models/{{newModelId}} (Authorization Basic sob:sob)

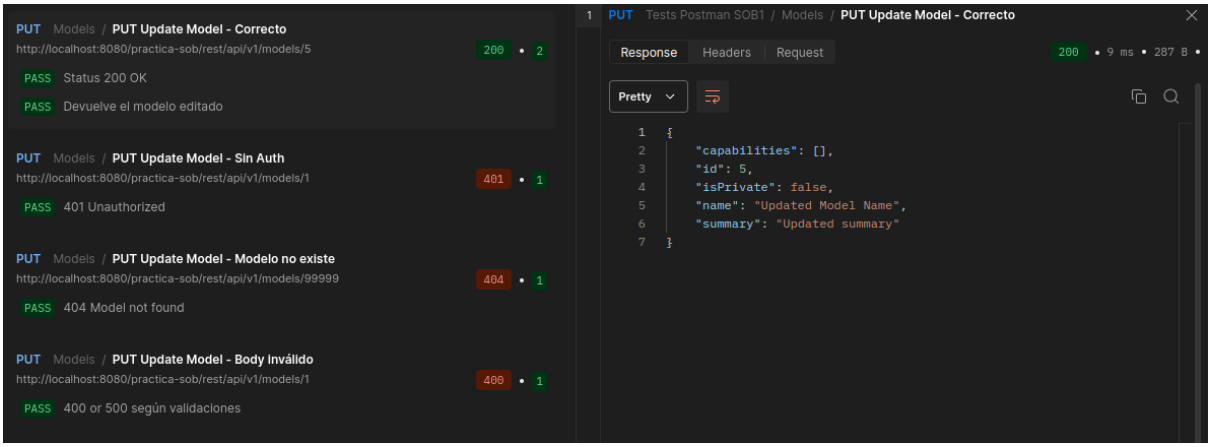
Body:

```
{
  "name": "Updated Model Name",
  "summary": "Updated summary"
}
```

➔ Tests:

```
pm.test("Status 200 OK", () => pm.response.to.have.status(200));

pm.test("Devuelve el modelo editado", () => {
    const m = pm.response.json();
    pm.expect(m).to.have.property("name");
});
```

➔ Resultado esperado: `200 OK con modelo actualizado`



Test 18: PUT Update Model - Modelo no existe

➔ Request: PUT {{baseUrl}}/models/99999 (Authorization Basic sob:sob)

Body:

```
{
  "name": "Updated Model Name"
}
```

➔ Tests:

```
pm.test("404 Model not found", () => pm.response.to.have.status(404));
```

Resultado esperado: `404 Not Found`

Test 20: DELETE Model - Correcto

➔ Request: DELETE {{baseUrl}}/models/{{newModelId}} (Authorization Basic sob:sob)

➔ Tests:

```
pm.test("204 No Content", () => pm.response.to.have.status(204));
```

➔ Resultado esperado: `204 No Content`

4.2.2. Customers

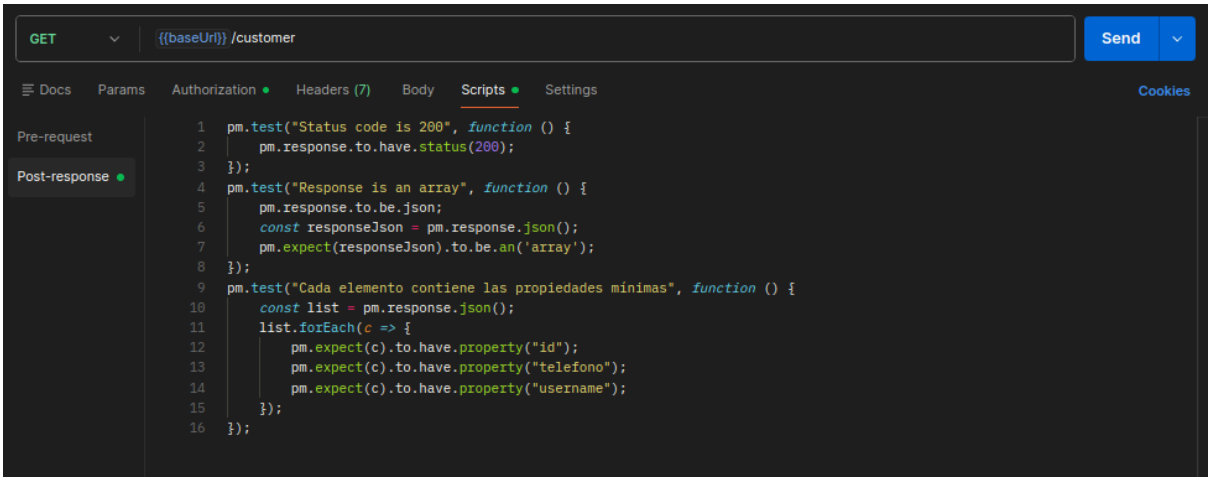
Test 1: GET All Customers

➔ Request: GET {{baseUrl}}/customer

➔ Tests automatizados:

```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});
pm.test("Response is an array", function () {
    pm.response.to.be.json;
    const responseJson = pm.response.json();
    pm.expect(responseJson).to.be.an('array');
});
pm.test("Cada elemento contiene las propiedades mínimas", function () {
    const list = pm.response.json();
    list.forEach(c => {
        pm.expect(c).to.have.property("id");
        pm.expect(c).to.have.property("telefono");
        pm.expect(c).to.have.property("username");
    });
});
```

➔ Resultado esperado: `200 OK con array de customers`



Test 2: GET Customer by ID

- ➔ Request: GET {{baseUrl}}/customer/1
- ➔ Tests automatizados:

```
pm.test("Debe devolver 200 OK", () => pm.response.to.have.status(200));

pm.test("Body matches string", function () {
    pm.expect(pm.response.text()).to.include('sob');
});

pm.test("Debe devolver el customer correcto", () => {
    const c = pm.response.json();
    pm.expect(c).to.have.property("id", 1);
});
```

- ➔ Resultado esperado:

```
{
  "id": 1,
  "username": "sob",
  "telefono": "600123456",
  "links": {
    "model": "/rest/api/v1/models/3"
  }
}
```

Test 4: PUT Update Customer (No Auth)

➔ Request: PUT {{baseUrl}}/customer/1 (sin Authorization header)

Body:

```
{
  "telefono": "+34999888777"
}
```

➔ Tests:

```
pm.test("Debe devolver 401 Unauthorized", () => {
    pm.response.to.have.status(401);
});
```

➔ Resultado esperado: `401 Unauthorized`

Test 5: PUT Update Customer (With Auth)

➔ Request: PUT {{baseUrl}}/customer/1 (Authorization Basic: sob:sob)

Body:

```
{
  "ultimoModeloVisitadoId": 1,
  "telefon": "+34648706708"
}
```

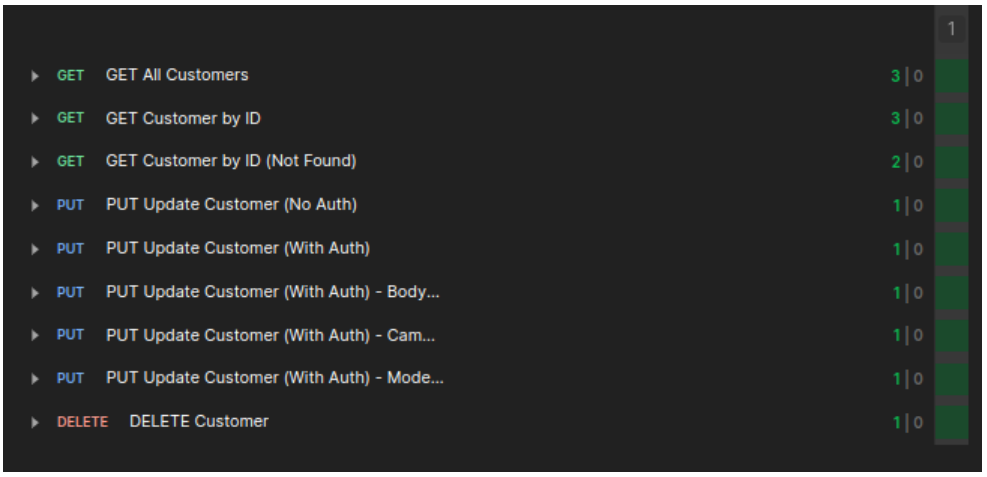
➔ Tests:

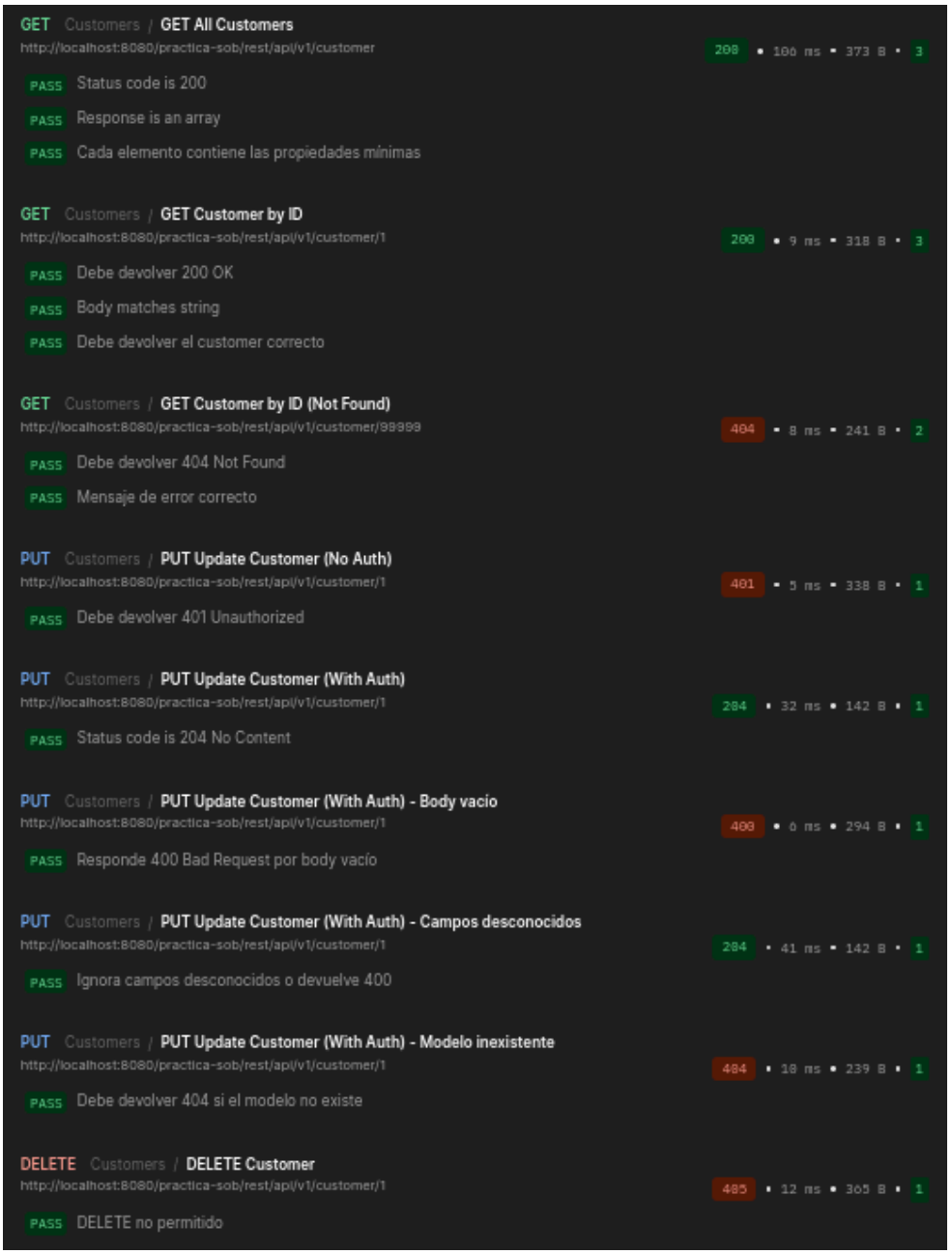
```
pm.test("Status code is 204 No Content", function () {
    pm.response.to.have.status(204);
});
```

➔ Resultado esperado: `204 No Content`

Entre otros tests.

Vista de la salida de los tests:





5. Conclusiones

Consideramos que hemos cumplido todos los objetivos que se había propuesto:

1) La API REST funciona completamente
2) Todos los métodos obligatorios están implementados
3) También hemos hecho todas las funcionalidades opcionales
4) Soporte JSON y XML
5) Autenticación con @Secured
6) Tests automatizados en Postman
7) Salidas todas con JSON

Hemos profundizado en el funcionamiento de varias cosas explicadas en la asignatura:

1) Relaciones JPA
2) Consultas JPQL dinámicas
3) Códigos del protocolo HTTP
4) Seguridad con Credentials y @Secured

En conclusión, en esta práctica hemos desarrollado una API REST funcional y escalable que cumple todos los requisitos obligatorios y opcionales, combinando diversos campos estudiados en la asignatura.