Sistemes Oberts

Pràctica 1

Miriam Duran

Satxa Fortuny

Índex

Introducció

En la primera part de la pràctica construirem una API Web que implementarà la funcionalitat necessària per afegir un article curt.

En aquesta pràctica l'objectiu principal és implementar el Back-End d'una pàgina web en la que els usuaris poden publicar articles. També s'implementa la base de dades.

Respecte al back-end, l'objectiu és tindre varies entitats, en aquest cas Article i Customer.

Després es pretén crear un servei web REST que és amb el que s'interaccionarà amb les entitats que estaran en forma de taula a la base de dades.

Al servei WEB el que es pretén és implementar mètodes http per a poder interactuar amb la pàgina web.

Estructura de la pràctica

A la pràctica hi ha varis directoris però el més important és el de source packages. Allí hi trobem 3 directoris:

Authn

En aquest directori s'hi troba tota la part de la autentificació. Hi tenim 3 fitxers:

Credentials.java: Com indica al nom, són les credencials dels usuaris. És en aquesta taula de la base de dades on hi tindrem el nom d'usuari, el id i el password.

RESTRequestFiltre.java: En aquesta classe el que tenim és el mètode per saber si un usuari està autentificat, el filtre. Bàsicament del que es tracta es sobre una cerca a la taula de credencials per saber si existeixen les credencials que hem obtingut. Cal dir que les credencials estan codificades en base 64 i estan unides tal que així nom:contrasenya.

Secured.java: En aquest document es declara l'anotació Secured que és la que utilitzarà l'autentificació.

Model.entities

D'entitats en tenim vàries però les principals són Article i Customers, la resta existeixen per complementar a en aquestes dues.

Article.java: En aquesta classe tenim els articles que pengen els usuaris a la pàgina web. Els mètodes que tindrem aquí seran els que farà servir el REST per a comunicar-se amb la base de dades, encara que no sempre.

Customer.java: En aquesta classe hi tenim els usuaris de la pàgina web. Veiem que al principi hi tenim les named queries. Aquestes són una alternativa a les queries tradicionals que permeten no haver de repetir la mateixa query més d'un cop si l'hem d'utilitzar freqüentment.

La resta de documents s'explicaran a les decisions de disseny, ja que aquests sí que podrien no existir, és a dir, independentment de la implementació, es tindrà un Article i un Customer, però això no passa amb la resta de documents, per tant es considera decisió de disseny.

Service

Aquí el que hi ha són les implementacions dels serveis web. En tenim dos, la de l'article i la del customer. Aquí és on el usuari de la pagina web interaccionarà realment. Als mètodes que s'hi implementen en aquest mètodes s'hi accedeix mitjançant un path.

Decisions de disseny

Article.java

Named Query

```
@NamedQuery(
    name = "Article.findLatestArticleIdByCustomerId",
    query = "SELECT a " +
            "FROM Article a " +
            "WHERE a.author.id = :customerId " +
            "AND a.publishDate = (SELECT MAX(a2.publishDate) FROM Article a2 WHERE a2.author.id = :customerId)"
)
```

El propòsit d'aquesta consulta és obtenir l'article més recent publicat per un client (autor) identificant-lo mitjançant el customerId. Específicament volem:

- Filtrar per client: La consulta selecciona només els articles escrits per un autor específic, identificat pel seu customerId.

- Seleccionar el més recent: Utilitza una subconsulta per determinar l'article amb la data de publicació més alta (MAX(a2.publishDate)), garantint que se seleccioni únicament el darrer article publicat per aquest autor.

Definir aquesta consulta com una NamedQuery permet que sigui precompilada i reutilitzada per JPA. Això millora el rendiment, ja que no s'ha de tornar a analitzar i compilar la consulta cada cop que s'executa.

Atributs de la classe

Els atributs que definim en aquesta part seran les columnes de la base de dades. Cal dir però que absolutament tots no ho seran, ja que hi hauran relacions que requeriran les seves pròpies taules.

```
@Id
@SequenceGenerator(name="Article_Gen", allocationSize=1)
@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "Article_Gen")
private Long id;

private String title;
@Column(length = 1000)
private String content;
private LocalDate publishDate;
private int views;
private String featuredImageUrl;
private boolean isPublic;
@ElementCollection(targetClass = Topic.class)
@Enumerated(EnumType.STRING)
private List<Topic> topics;
private String resume;
@ManyToOne
@JsonbTransient
private Customer author;
```

Primer de tot tenim el id. Aquest és necessari, ja que tota taula ha de tindre una clau primària i és més pràctic tindre un número identificador abans que intentar trobar algun atribut ho suficientment identificatiu com per a que valgui com clau primària. Les anotacions que té damunt indiquen que es un valor que es genera sol, és a dir, a l'hora de fer un insert, no haurem d'introduir el valor del id, sinó que agafarà l'últim de la taula i li sumarà 1.

També cal comentar que content té una anotació especial degut a que de normal, a JPQL quan es crea un string, li assigna una mida màxima de 250 caràcters, el qual tenint en compte que estem parlant del contingut d'un article, es queda curt. Hem posat que la mida màxima és 1000 caràctes ja que llavors no ens dona problemes al fer els inserts dels articles. Si s'hagués d'ampliar només caldria augmentar aquest número.

També a comentar tenim la llista de topics. Aquest atribut és una llista degut a que un article pot tindre varis tòpics, per tant no val amb una sola variable tipus Topic. Veiem que al damunt hi ha dues annotacions. La primera serveix per especificar que l'atribut pertany a una enumeració. La segona especifica quin és el tipus de variable del contingut del tòpic.

Resume és un String que ens serveix per a quan veiem la previsualització de l'article sapiguem de que va sense haver d'entrar.

També tenim l'autor, que serà de tipus Customer. És una relació ManyToOne, ja que un sol customer pot tindre molts articles però un article només podrà tindre un autor. L'article és el propietari de la relació ja que és el costat Many.

A l'article hi tenim un get del nom de l'autor, no és una columna de l'article ja que el nom ja el tenim al Customer i estaríem repetint la informació inútilment. Hem decidit que el que es mostri és el nom i no el nom d'usuari, ja que considerem que així és més formal.

@JSONBTRANSIENT

Aquesta anotació permet que l'atribut estigui present a la base de dades, però sigui invisible per a les conversions a JSON.

Per què ho fem servir?

Doncs atès que **Customer** i **Article** contenen una relació bidireccional, quan es formava un fitxer JSON es creava una relació cíclica i recursiva entre ambdós elements per la qual cosa ens vam veure obligats a 'tapar' un dels dos. Tot i així, les dades continuen sent vistes atès que s'han creat DTO que mostren la totalitat dels atributs.

ArticleDTO.java

Aquesta classe com diu el nom, és el DTO de Article. Del que es tracta és d'una classe amb menys atributs per a quan al REST vulguem obtenir només certa informació de la classe i no tota. En específic és per a quan volem veure una previsualització del article. Aquesta classe no té mètodes ja que només volem crear la instància per després poder visualitzar-la, és a dir, no hem de treballar amb ella realment.

Topic.java

En aquest document hi tenim una enumeració. És a dir que en aquesta pàgina web no hi poden haver infinits tòpics, venen donats d'estàndard. D'aquesta manera augmentem la seguretat de la pàgina al limitar els permisos dels usuaris a la vegada de poder encarar el tema de la pàgina. Actualment si veiem els tòpics que hi ha podrem concloure que està encarada a la informàtica.

ArticleService.java

Aquesta classe és la que li permet al usuari de la pàgina interactuar amb aquesta, ja sigui per obtenir, crear o eliminar articles. Començarem explicant la part inicial.

```
@Stateless
@Path("article")
public class ArticleService {
    private static final String AUTHORIZATION_HEADER_PREFIX = "Basic ";
    @PersistenceContext(unitName = "Homework1PU")
    private EntityManager em;
    @Inject
    private CustomerService cs;
```

La primera anotació stateless significa que no hi ha sessions, és a dir, que pots executar qualsevol mètode des de qualsevol lloc i seguirà passant el mateix. Aclarir però que pot ser que aquesta no es repeteixi exactament, ja que llavors estem parlant d'un mètode idempotent.

El path és bàsicament l'adreça que s'haurà d'utilitzar a l'hora de fer servir el servei web de l'article.

Després hi tenim un string amb l'autorització, això indica quin tipus de procediment per revisar credencials estem fent servir. D'aquesta manera quan haguem de fer servir un mètode http que necessiti les nostres credencials, que en aquest cas seran el nom d'usuari i la contrasenya, haurem de marcar Basic.

A partir d'aquí tenim l'Entity Manager i el Customer Service. L'Entity Manager és el que s'encarrega de interactuar realment amb les entitats a la base de dades. El customer service és el servei web del customer. Aquest ens permetrà poder fer servir els seus mètodes http.

En aquestes dues instàncies s'hi està produint una injecció, m'explico. En el suposat cas que simplement declaréssim els dos atributs, es crearien dues instàncies separades a les que estem fent servir realment, és a dir que no cal fer servir el constructor. No s'han de fer news, amb el patró d'assignació de dependències, ja s'assigna directament. El servidor gestiona el cicle de vida de les entitats. A en aquest fenomen se l'anomena Dependency Injection.

El problema és que llavors estaríem creant instàncies innecessàriament, per tant l'anotació inject ens permet indicar que s'utilitza l'instancia actual. És com una mena de Singleton.

Al de l'entity manager no hi posem Inject, sinó que li indiquen quina és la unitat de persistència que ha de fer servir. Ara procedirem a explicar els métodes http que s'han implementat.

GET /rest/api/v1/article?topic=${topic}&author=${author}

L'objectiu d'aquest mètode és obtenir tots els articles de la pàgina, encara que dependent dels paràmetres que s'hi introdueixin, l'objectiu variarà. Volem aclarir que existeixen varies maneres d'introduir els paràmetres. En aquest cas no s'introdueixen al path sinó a l'apartat de paràmetres. Això és útil per a quan necessitem varis paràmetres i no només un.

```
@GET
@Produces(MediaType.APPLICATION_JSON)
public Response findAll(
    @QueryParam("topic")List<Topic> topic,
    @QueryParam("author") String author) {
    String baseQuery = "SELECT a FROM Article a WHERE 1=1";
```

Veiem doncs les dues primeres anotacions. La primera indica a quin mètode http fa referència. Si no existís aquesta anotació, al fer la consulta al path no es podria saber quin mètode del ArticleService.java s'hauria de fer servir.

Els mètodes http no poden retornar una variable "Normal"(Enter, String, Array...), ja que llavors s'hauria d'implementar una resposta personalitzada per cada resposta del servei web de cada pàgina. Per això existeix Response, una variable específicament creada per tractar sol·licituds. En ella podem indicar què ha passat amb la sol·licitud i a la vegada podem manar contingut, incloent no només un valor, sinó inclús l'instancia d'una classe. El que indica Produces però, és que el contingut de la resposta estarà en format Json.

Per això tots els mètodes dels serveis web que tinguin una anotació http, retornaran un Response. El nom de la funció ens permet poder fer servir el mètode internament, ja que si un mètode ha de cridar a un altre, no ho farà a través de una operació http.

La anotació QueryParam ens permet extreure els paràmetres de la URL. En aquest cas en tenim dos. El primer és una llista de tòpics. Quan aquest paràmetre estigui present, significarà que haurem de retornar tots els articles de la pàgina web que continguin algun dels tòpics indicats. Hi poden arribar a haver fins a 2.

Després hi tenim l'autor, quan estigui present haurem de retornar tots els articles creats per ell.

Cal aclarir que aquests paràmetres es poden combinar, és a dir, pots fer que et retorni tots els articles d'un autor que tinguin els tòpics especificats.

A partir d'aquí comença el mètode. L'objectiu d'aquest serà anar construint una consulta a la base de dades en funció dels paràmetres que s'hagin introduït. Com que aquestes adicions

seran al WHERE, hauran d'anar amb un AND, per tant hi ha d'haver un WHERE primer. Es podria fer que si no hi hagués cap paràmetre no hi hagués WHERE i que quan se'n posés un fos quan sí que hi hagués WHERE però hem decidit deixar-lo sempre. Com que aquesta primera condició no ha de filtrar res hem decidit posar un 1=1 que donarà sempre true.

```
// Afegir condicions només si hi ha filtres específics
if (author != null) {
    baseQuery += " AND a.author.name = :author";
}
for(int i=0; i<topic.size(); i++){
    baseQuery += " AND :topic"+i+" MEMBER OF a.topics";
}
// Ordenar per popularitat
baseQuery += " ORDER BY a.views DESC";
```

Llavors ens dediquem a afegir a la query en funció de si són nulls els paràmetres. Amb els tòpics simplement hi tindrem una llista buida, no nula, per tant no farà cap iteració però no donarà un NullPointerException al intentar executar un mètode a una instància nul·la.

Finalment hi ha un order by que bàsicament ordena els resultats de manera descendent en funció del número de vistes de l'article, això és així ja que és com ho mana l'enunciat.

```
// Crear la consulta
Query query = em.createQuery(qlString: baseQuery);

// Assignar els paràmetres si cal
if (author != null) {
    query.setParameter(name: "author", value: author);
}
if (!topic.isEmpty()) {
    for(int i=0; i<topic.size(); i++){
        query.setParameter("topic"+i+"", value: topic.get(index: i));
    }
}
// Executar la consulta
List<Article> resultList = query.getResultList();
```

Ara que ja tenim la query haurem d'indicar que ja l'hem construït i procedir a introduir les variables especificades per dos punts al principi del nom. Aquest mètode s'anomena bind variables i serveix per augmentar la seguretat de la pàgina al evitar l'Injection SQL.

Al final de tot executem la comanda amb el .getResultList(). Si sapiguessim que 100% només ha de ser un sol resultat, hauríem de fer servir .getSingleResult(), però en aquest cas és probable que ens retorni varis articles, els quals guardarà a la llista introduïda.

```
if (resultList == null) {
    return Response.status(status: Response.Status.NOT_FOUND).build();
}
// Transformar articles seceers a previsualitzacions
List<ArticleDTO> resultDTOList = new LinkedList<>();
ArticleDTO dto;
for(Article article:resultList){
    dto = new ArticleDTO(title: article.getTitle(), publishDate:article.getPublishDate(),
    views: article.getViews(), featuredImageUrl: article.getFeaturedImageUrl(), isPublic: article.isItPublic(),
    resume: article.getResume(), author: article.getAuthor().getName());
    resultDTOList.add(e: dto);
}

// Retornar la resposta JSON
return Response.ok(entity: resultDTOList).build();
```

Aquí ja tractem els resultats de la llista. Els filtratges s'han de fer a la base de dades ja que portar dades innecessàries des de memòria és molt costós.

El que farem es mirar si no hi ha hagut cap resultat, per poder informar al usuari.

Aquest mètode es farà servir per a quan es vulguin veure molts articles en forma de previsualització, per tant no haurem de mostrar tot el contingut. Per això haurem de transformar un article sencer en una previsualització que porta el nom de ArticleDTO.

Finalment ho posem a la llista que manarem com a resposta indicant que tot ha anat bé.

GET /rest/api/v1/article/${id}

L'objectiu d'aquest mètode és obtenir un article individual sencer. Pot donar-se el cas que l'article sigui privat i per tant l'usuari haurà d'estar registrar per a poder veure el contingut. El contingut de la previsualització es podrà veure sempre però.

```
@GET
@Path("{id}")
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Response getArticleById(@PathParam("id") Long id, @HeaderParam("Authorization") String reg) {
```

D'anotacions extra hi tenim el path. Aquesta anotació ens indica que a part de posar el path estàndard del servei web de l'article, haurem d'introduir el id de l'article que desitgem mostrar.

A part aquest mètode també pot produir una resposta amb xml.

Aquest cop les anotacions dels paràmetres del mètode indiquen que una variable es troba al path, indicant-hi el nom que té, el tipus de variable que és i el nom que se li donarà a dins del mètode. També tenim l'altra anotació que ens indica que farem servir les credencials per revisar permisos.

```
try {
    // Buscar el artículo por su ID usando una consulta
    Article article = em.createQuery(qlString:"SELECT a FROM Article a WHERE a.id = :id", resultClass: Article.class)
                        .setParameter(name:"id", value: id)
                        .getSingleResult();

    // Si el artículo no es público, verifica el registro
    if (!article.isItPublic()) {
        if (reg == null || !isRegistered(reg, id: null)) {
            return Response.status(status: Response.Status.UNAUTHORISED).entity(entity: "Has d'estar registrat per veure aquest article").build();
        }
    }

    // Incrementar el contador de vistas y actualizar en la base de datos
    article.setViews(article.getViews() + 1);
    em.persist(entity: article);

    // Retornar la información del artículo con un 200 OK
    return Response.ok(entity: article).build();
```

Aquí el que fem és consultar a la base de dades directament per obtenir l'article amb el id. Cal especificar però quina és la classe resultat amb un article.class, introduir els valors de les bind variables amb un setParameter i finalment executar la comanda amb el getSingleResult().

Ara ja tenim l'article, però pot ser que sigui privat, per tant haurem de revisar si el client que ha executat la petició està registrat. En cas negatiu ho informem.

Després haurem d'actualitzar el comptador de vistes i persistir-ho al context de persistència, ja que al modificar la variable el que hem fet ha sigut crear una copia. És a dir, que si no féssim el persist, realment no augmentarien les vistes.

Finalment retornem l'article sencer, sense DTOs indicant que ho hem fet bé.

```
} catch (NoResultException e) {
    // Si no se encuentra el artículo, devolver 404 Not Found
    return Response.status(status: Response.Status.NOT_FOUND).entity(entity: "Artículo no encontrado").build();
} catch (Exception e) {
    // Manejar otros errores inesperados
    return Response.status(status: Response.Status.INTERNAL_SERVER_ERROR).entity(entity: "Ocurrió un error interno").build();
}
```

Passa però que es pot donar el cas que no vagi tot perfecte. La primera excepció s'encarrega de controlar que l'article no s'hagi trobat. La segona és un tractament universal per totes les altres excepcions.

```
private boolean isRegistered(String reg, Long id) {

    // Validar si el encabezado es válido
    if (reg == null || reg.isEmpty()) {
        return false;
    }
```

Ara explicarem el mètode que controla si l'usuari està registrat. Passa però que també s'utilitza per saber si les credencials que es donen són de l'usuari amb l'id especificat.

Primer mirem si les credencials hi són, ja que en cas negatiu, sempre retornarem un false.

```
try {
    // Decodificar Base64 del encabezado Authorization
    reg = reg.replace(target: AUTHORIZATION_HEADER_PREFIX, replacement: "");
    String decode = Base64.base64Decode(orig:reg);
    StringTokenizer tokenizer = new StringTokenizer(str: decode, delim: ":");
    String username = tokenizer.nextToken();
    String password = tokenizer.nextToken();

    // Buscar credenciales en la base de datos
    TypedQuery<Credentials> query = em.createNamedQuery(name:"Credentials.findUser", resultClass: Credentials.class);
    Credentials credentials = query.setParameter(name:"username", value: username).getSingleResult();

    if(id != null){
        return ((credentials.getPassword().equals(anObject:password)) && (credentials.getId().equals(obj: id)));
    }
    // Validar contraseña
    return credentials.getPassword().equals(anObject:password);
```

Originalment a l'inici del document del service del article ja hem declarat l'autorization, el problema és que quan el passem per paràmetre no només ens venen el username i la contrasenya, també ve el tipus d'autorització i això no ens interessa en aquesta funció ja que no es dinàmica en base al tipus d'autorització, referint-me a que només funciona amb

autoritzacions bàsiques. Per això ho únic que fa és molestar a l'hora d'haver de separar els atributs.

Les credencials que ens passen per la capçalera no estan en un format corrent. Realment es mostren en el format username:password i codificat en base 64, per això a les primeres línies del codi, el que fem es descodificar i separar les credencials per poder obtenir les variables separadament.

Després tocarà fer ús d'una named query ara que tenim el username per a poder obtenir les credencials. Ara si volem mirar si el id és de les credencials, compararem les credencials que hem aconseguit amb la consulta a traves del username.

Ara mirarem si d'aquest username hi ha coincidència amb la contrasenya i amb el id que ens han passat.

Sinó mirarem tan sols el password.

```
} catch (IllegalArgumentException | NoResultException e) {
    // Error en la decodificación o usuario no encontrado
    return false;
}
```

Cal també controlar que no hagi trobat res o que l'argument sigui problemàtic.

En conclusió, isRegistered et diu si les credencials exactes coincideixen, podent augmentar la revisió fins al id.

DELETE /rest/api/v1/article/${id}

L'objectiu d'aquest mètode és eliminar un article de la base de dades a través d'un id.

```
@Secured
@DELETE
@Path("{id}")
public Response deleteArticle(@PathParam("id") Long id, @HeaderParam("Authorization") String reg){
    // Borrar el artículo usando su id.
    try{
    Long identificacio =
            em.createQuery(qlString: "SELECT a.author.credentials.id FROM Article a WHERE a.id = :id", resultClass: Long.class)
            .setParameter(name: "id", value: id)
            .getSingleResult();
    if(!isRegistered(reg, id: identificacio))return Response.status(status: Response.Status.UNAUTHORIZED).build();
    em.createQuery(qlString: "DELETE FROM Article a WHERE a.id = :id", resultClass: Article.class)
                        .setParameter(name: "id", value: id).executeUpdate();
    } catch (IllegalArgumentException | NoResultException e) {
        // Error en la decodificación o usuario no encontrado
        return Response.status(status: Response.Status.UNAUTHORIZED).build();
    }
    // Al ser idempotent no hay necesidad de revisar su existencia.
    // Retornar un 200 OK
    return Response.ok().build();
}
```

Veiem que al igual que el get d'un article individual, el id s'especifica pel path. També hi tenim l'anotació de la operació http. L'única novetat és que ara hi tenim l'anotació secured, la qual indica que s'utilitza un procés d'autentificació posterior al mètode a traves de les credencials proporcionades pel HeaderParam.

Abans de poder borrar l'article hem de mirar que l'usuari estigui registrat, per tant haurem de mirar si l'autor de l'article coincideix amb les credencials proporcionades.

En cas negatiu informem a l'usuari que no esta autoritzat.

Si sí que ho és, podem procedir a eliminar l'article, informant també a l'usuari.

POST/rest/api/v1/article

L'objectiu d'aquest mètode és publicar un article, és a dir, afegir-lo a la base de dades.

```
@Secured
@POST
@Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
public Response create(Article article, @HeaderParam("Authorization") String reg){
```

Les úniques novetats que hi veiem a la capçalera és el consumes. Aquesta anotació indica que aquest mètode utilitza documents Xml i Json com a entrada.

Aquest cop també hi tenim un article per paràmetre sense cap anotació. Això significa que el arxiu que consumeix el mètode es transforma en una instància de Article.

El funcionament del mètode és bàsicament que s'ha d'estar registrat a la base de dades per a poder publicar un article i que el customer es posa sol en base a en aquestes.

Així doncs el procediment que seguirà serà el següent.

```
reg = reg.replace(target: AUTHORIZATION_HEADER_PREFIX, replacement: "");

// Decodificar las credenciales Base64
String decoded = Base64.base64Decode(orig: reg);
StringTokenizer tokenizer = new StringTokenizer(str:decoded, delim: ":");
String username = tokenizer.nextToken();
String password = tokenizer.nextToken();

// Verificar las credenciales en la base de datos
TypedQuery<Long> query = em.createQuery(
    qlString: "SELECT c.id FROM Credentials c WHERE c.username = :username AND c.password = :password",
    resultClass: Long.class);
Long id = query.setParameter(name: "username", value: username)
               .setParameter(name: "password", value: password)
               .getSingleResult();
```

Descodifiquem les credencials i les busquem a la base de dades. Amb això obtenim el id.

```
// Obtener el objeto de Credentials y el Customer asociado
Credentials client = em.find(entityClass: Credentials.class, primaryKey: id);
if (client == null) {
    return Response.status(status: Response.Status.UNAUTHORIZED)
                   .entity(entity: "Unauthorized access to private article")
                   .build(); // Credenciales no válidas
}
if(article.getTitle()==null){
    return Response.status(status: Response.Status.BAD_REQUEST)
                   .entity(entity: "An article must have a title")
                   .build(); // Título necesario
}
if(article.getContent()==null){
    return Response.status(status: Response.Status.BAD_REQUEST)
                   .entity(entity: "An article must have a body")
                   .build(); // Título necesario
}
// Obtener el Customer asociado a las credenciales
TypedQuery<Customer> query2 = em.createQuery(
    qlString:"SELECT c FROM Customer c WHERE c.credentials.id = :id",
    resultClass: Customer.class);
query2.setParameter(name:"id", value: id);
Customer customer = query2.getSingleResult();
```

Ara que tenim les credencials, obtenim el client. Tocarà revisar que realment existeixi, que ens hagin proporcionat un títol i un contingut, ja que sense cap d'aquests no tindrà gaire sentit.

Finalment si tot és correcte, obtenim el customer associat a les credencials, d'aquesta manera el podrem associar a l'article.

```
if(article.getTopic().size()>2){
    return Response.status(status: Response.Status.BAD_REQUEST)
                   .entity(entity: "No hi poden haver més de 2 tòpics")
                   .build();
}
if(!article.getTopic().isEmpty()){
    for (Topic topic : article.getTopic()) {
        if (!isValidTopic(topic)) {
            return Response.status(status: Response.Status.BAD_REQUEST)
                           .entity("Tòpic no vàlid: " + topic)
                           .build(); // Si algún tema no es válido
        }
    }
}
// Establecer el autor y la fecha de publicación
article.setAuthor(author: customer);
customer.addArticle(article);
article.setPublishDate(publishDate:LocalDate.now());

// Persistir el artículo
em.persist(entity: article);
em.merge(entity: customer);

// Se devuelve CREATED y el id del artículo
return Response.status(status: Response.Status.CREATED)
               .entity("Article creat amb l'ID: " + article.getId())
               .build();
```

Ara farem el control del número de tòpics, a la pàgina web no hi pot haver cap article amb més de dos tòpics, i aquests han de permetre a la enumeració, per tant també cal controlar-ho. Després afegim fem la relació customer article i hi posem la data de publicació. Per últim hi fem un merge i un persist al context de persistència per a que es guardi després del mètode. Finalment li passem el id del article publicat, informant també que tot ha anat bé.

Customer.java

```java
public class Customer implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @SequenceGenerator(name="Customer_Gen", allocationSize=1)
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "Customer_Gen")
    private Long id;
    @OneToOne
    private Credentials credentials;
    private String name;
    private String featuredImageUrl;
    @OneToMany(mappedBy = "author", fetch = FetchType.LAZY)
    private List<Article> articles;
    @Transient
    private Link links;
```

L'entitat **Customer** és l'encarregada de desar les dades de les persones que es registren al sistema, ja siguin autors o simples lectors.

A continuació s'expliquen les diferents dades que es guarden i els seus usos:

- **id**: número identificatiu i únic per a cada registre que s'autogenera en ser persistit, tot i ser necessari per trobar el registre a la base de dades l'id a ho què donem veritable importància és al dels credencials.

- **credentials**: variable de tipus Credentials que desa el nom d'usuari del client, una contrasenya i un ID únic. Conté la relació @OneToOne atès que per a cada credencial només hi pot haver un usuari i viceversa.

- **name**: Com a part de decisió de disseny es proposa crear una variable cadena per desar el nom de l'autor, això permet separar el nom d'usuari del compte del nom que apareix al perfil. D'aquesta manera, diversos autors poden tenir el mateix nom amb diferents usernames. Això ens és útil i pràctic per a un dels mètodes GET.

- **featuredImageUrl**: cadena que desa el path cap a la imatge de perfil de l'usuari.

- **articles**: llista d'articles que ha escrit o publicat l'usuari, aquesta variable és mapejada per la variable autor dins 'Article.java' atès que conté una relació @OneToMany.

- **links**: està marcada com a @Transient perquè no cal que sigui persistida a la base de dades. Aquesta variable desa el valor del link al darrer article escrit per l'usuari en cas que en tingui. Només és utilitzada en un dels mètodes GET de la classe 'Customer.java'.

Aquesta classe a més conté un mètode per afegir un article a la llista d'articles del customer de manera controlada. Aquesta lògica ens és útil per relacionar de manera bidireccional els nous articles que publiquen a partir del POST que hi ha a 'ArticleService' i així després persistir els canvis a totes dues entitats.

```
public void addArticle(Article article){
    if(article != null){
        if (!this.articles.contains(o: article)){
            this.articles.add(e: article);
            article.setAuthor(author: this);
        }
    }
}
```

Link.java

```
@XmlRootElement
public class Link implements Serializable {
    private static final long serialVersionUID = 1L;

    private String article;
```

La classe **Link** no és cap entitat atès que com anteriorment s'ha comentat no cal que sigui persistida a la nostra base de dades. Ha estat creada específicament per complir els requisits d'un dels mètodes GET de CustomerService.

El tag que conté **@XmlRootElement** indica a JAXB que la classe pot ser transformada (serialitzada) en un document XML.

Aquesta classe conté un sol atribut:

- **links**: guarda el link a l'últim article pujat per l'autor.

CustomerDTO.java

```
public class CustomerDTO implements Serializable {
    private static final long serialVersionUID = 1L;

    private Long id;
    private String user;
    private String featuredImageUrl;
    private List<ArticleDTO> articles;
    private boolean includeArticles;
    @Transient
    private Link links;

    // Constructor para convertir entidad a DTO
    public CustomerDTO(Customer customer, boolean includeArticles) {
        if (customer != null) {
```

Aquesta classe que tampoc no és una entitat té com a ús principal poder tornar els elements necessaris per al client que realitza la petició. Ens permet col·locar amb flexibilitat aquells elements que es corresponguin segons la petició que s'estigui atenent.

Els atributs que conté són els següents:

- **id**: número d'identificació de l'usuari retornat a la base de dades. Està de més aclarir que l'id que es torna en realitat és el de les credencials, atès que, per motius de decisió de disseny es creu que té més importància i veritablement vincula al registre.

- **user**: cadena que desa el nom del client.

- **featuredImageUrl**: cadena que desa el path a la imatge de perfil de l'autor o usuari.

- **articles**: llista d'articles de l'usuari en cas que en tingui. És important recalcar que es guarden en format **'ArticleDTO'** per mostrar la versió curta dels articles en comptes del contingut total.

- **includeArticles**: aquest booleà que vindrà indicat per paràmetre al constructor de la classe ens permet saber si volem que la classe conservi els articles a la sortida de la petició. Amb això, permetem ampliar l'ús de **CustomerDTO** perquè sigui útil en diferents mètodes de servei.

- **links**: està marcada com a @Transient perquè no cal que sigui persistida a la base de dades. Aquesta variable desa el valor del link al darrer article escrit per l'usuari en cas que en tingui.

RestAPP.java

```
package service;

import jakarta.ws.rs.core.Application;

@jakarta.ws.rs.ApplicationPath("rest/api/v1")
public class RESTapp extends Application {

}
```

Aquesta classe defineix l'arrel de la URL base per a tots els recursos de REST dins d'aquesta aplicació. En aquest cas, l'arrel és rest/api/v1 tal com es demanava a l'enunciat.

AbstractFacade.java

Deixem aquest servei per poder utilitzar-lo al servei **CustomerService**. D'aquesta manera, aprofitem les consultes ja implementades per ser usades en alguns dels nostres mètodes o peticions.

CustomerService.java

```
@Stateless
@Path("customer")
public class CustomerService extends AbstractFacade<Customer> {
    @PersistenceContext(unitName = "Homework1PU")
    private EntityManager em;

    public CustomerService() {
        super(entityClass: Customer.class);
    }
```

Classe que desa els serveis web relatius a l'usuari. L'anotació @Stateless permet executar cada trucada del client com si fos la primera vegada. També conté l'anotació @Path per indicar que l'URL on es troben els recursos dels usuaris són a 'customer'. Tot seguit s'explica com s'ha implementat cadascun dels serveis proposats:

GET /rest/api/v1/customer

```
@GET
@Path("{id}")
@Produces(MediaType.APPLICATION_JSON)
public Response getCustomerById(@PathParam("id") Long id) {
    Customer customer = super.find(id);

    if (customer == null) {
        return Response.status(status:Response.Status.NOT_FOUND)
                       .entity("Customer not found for id: " + id)
                       .build();
    }

    return Response.ok(new CustomerDTO(customer, includeArticles: true)).build();
}
```

Aquest mètode està fet per cercar un client pel ID. Conté diverses anotacions que bàsicament serveixen perquè el servei REST sàpiga com manejar les sol·licituds:

- @GET diu que això respon a un "GET" (és a dir, només demana dades).
- @Path("{id}") és com una plantilla per a la URL, on {id} és la part variable i des d'on s'agafarà l'ID de l'usuari a tornar.
- @Produces(MediaType.APPLICATION_JSON) diu que el que surti del mètode serà convertit a JSON.

Explicació del funcionament del mètode:

1. Cerca el client pel seu ID amb super.find(id) (mètode proporcionat per l'AbstractFacade).
2. Si no el troba, retorna un error 404 dient "No hi ha client amb aquest ID".
3. Si el troba, el converteix en un DTO amb les dades de l'usuari (indicant amb un *true* que en cas de que tingui articles els retorni en format curt) i el torna amb un 200, que indica que la petició ha sigut resposta amb èxit.

GET /rest/api/v1/customer/${id}

```java
@GET
@Produces(MediaType.APPLICATION_JSON)
public Response getAllCustomers() {
    List<Customer> customers = super.findAll();
    List<CustomerDTO> updatedCustomers = new ArrayList<>();  // Nueva lista para almacenar los clientes con su artículo

    // Itera sobre los clientes
    for (Customer customer : customers) {
        CustomerDTO customer2 = new CustomerDTO(customer, includeArticles: false);
        try {
            // Ejecutar la consulta para obtener el artículo más reciente del cliente
            List<Article> articles = em.createNamedQuery(name: "Article.findLatestArticleIdByCustomerId", resultClass: Article.class)
                                       .setParameter(name: "customerId", value: customer.getId())
                                       .getResultList();
            Article article = null;
            if (articles.get(index: 0)!= null) {
                article = articles.get(index: 0);
            }

            // Crea un nuevo enlace y asigna el artículo más reciente al cliente
            Link link = new Link();
            if (article != null) {
                link.setArticle(article);
            }

            customer2.setLinks(links: link);  // Asigna el Link al cliente
        } catch (Exception e) {
            // Si ocurre una excepción (por ejemplo, si no se encuentra el artículo), simplemente no asignamos un artículo.
            // Log o manejar la excepción según sea necesario
            customer2.setLinks(links: null);  // Asigna null si no hay artículo
        }

        updatedCustomers.add(e: customer2);  // Agrega el cliente con su artículo (o sin artículo) a la lista nueva
    }

    // Retorna la lista de clientes con su artículo más reciente
    return Response.ok(entity:updatedCustomers).build();
}
```

Aquest mètode es fa servir per tornar la llista de tots els clients, però a més, afegeix l'article més recent que hagi escrit cada client. Conté diverses anotacions que bàsicament serveixen perquè el servei REST sàpiga com manejar les sol·licituds:

- @GET diu que això respon a un "GET" (és a dir, només demana dades).
- @Produces(MediaType.APPLICATION_JSON) diu que el que surti del mètode serà convertit a JSON.

Explicació del funcionament del mètode:

1. Primer s'aconsegueixen tots els clients de la base de dades (super.findAll()).
2. Després, passa per cada client i cerca el seu article més recent usant la consulta (**NamedQuery**) "Article.findLatestArticleIdByCustomerId" creada a l'entitat 'Article'.
3. Si trobeu un article, el desa en com a "link" i l'associa amb el client. Si no, simplement segueix sense assignar res.
4. Posa cada client processat en una nova llista (updatedCustomers) i al final torna aquesta llista en format JSON.

Alguns dels aspectes importants del disseny són:

- NamedQuery per a Articles: És molt eficient perquè busca només l'article més recent en lloc de tornar tots. Això optimitza el rendiment, sobretot si hi ha molts articles.

- Control d'excepcions: Si passa un error, no s'interromp el flux complet. Simplement no assigna el link al client, assegurant que sempre es torna una resposta.

- Ús de DTO: Els CustomerDTO permeten enviar dades netes i estructurades, sense exposar l'entitat completa. Gràcies a la seva flexibilitat en el codi, podem indicar amb un booleà que no volem que es retornin els articles.

- Assignació Dinàmica dels Links: Crear un enllaç amb l'article més recent és una forma elegant d'enriquir la informació del client sense sobrecarregar l'API amb dades innecessàries. I com l'atribut es Transient, la informació del link desapareix una vegada s'envia la resposta.

PUT /rest/api/v1/customer/${id}

```java
@PUT
@Secured
@Path("{id}")
@Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
@Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
public Response updateCustomer(@PathParam("id") Long id, Customer updatedData, @HeaderParam("Authorization") String reg) {
    // Buscar el cliente existente en la base de datos
    Customer existingCustomer = super.find(id);
    if (existingCustomer == null) {
        return Response.status(status:Response.Status.NOT_FOUND).entity(entity:"No s'ha trobat el teu usuari.").build(); // Cliente no encontrado
    }

    if (!isSameUser(reg, id)){
        return Response.status(status:Response.Status.FORBIDDEN).entity(entity:"Les identificacions no coincideixen.").build(); // Cliente intenta modificar a otro
    }

    // Verificar si el objeto de credentials está presente
    if (updatedData.getCredentials() != null) {
        if (updatedData.getCredentials().getUsername() != null) {
            TypedQuery<Credentials> query = em.createNamedQuery(name: "Credentials.findUser", resultClass: Credentials.class);
            Credentials credentials = query.setParameter(name: "username", value: updatedData.getCredentials().getUsername()).getSingleResult();
            if(credentials != null){
                return Response.status(status:Response.Status.NOT ACCEPTABLE).entity(entity:"Aquest username ja està en ús!").build();
            }
            existingCustomer.getCredentials().setUsername(username: updatedData.getCredentials().getUsername());
        }

        if (updatedData.getCredentials().getPassword() != null && !updatedData.getCredentials().getPassword().isEmpty()) {
            existingCustomer.getCredentials().setPassword(password: updatedData.getCredentials().getPassword());
        }
    }

    // Actualizar otros campos si están presentes
    if (updatedData.getName() != null) {
        existingCustomer.setName(name: updatedData.getName());
    }

    if (updatedData.getFeaturedImageUrl() != null) {
        existingCustomer.setFeaturedImageUrl(featuredImageUrl:updatedData.getFeaturedImageUrl());
    }

    if (updatedData.getArticles() != null) {
        return Response.status(status:Response.Status.FORBIDDEN).entity(entity:"No pots modificar articles!").build();
    }

    // Actualizar el cliente en la base de datos
    em.merge(entity:existingCustomer);
```

```java
    // Retornar respuesta de éxito
    return Response.status(status:Response.Status.OK)
                   .entity(entity:"Les dades s'han modificat correctament.")
                   .build();
}

// Método para saber si el cliente que está siendo modificado es el mismo que el autentificado
private boolean isSameUser(String reg, Long id) {

    // Validar si el encabezado es válido
    if (reg == null || reg.isEmpty()) {
        return false;
    }

    try {
        // Decodificar Base64 del encabezado Authorization
        reg = reg.replace(target: AUTHORIZATION_HEADER_PREFIX, replacement: "");
        String decode = Base64.base64Decode(orig: reg);
        StringTokenizer tokenizer = new StringTokenizer(str:decode, delim: ":");
        String username = tokenizer.nextToken();

        // Buscar credenciales en la base de datos
        TypedQuery<Credentials> query = em.createNamedQuery(name: "Credentials.findUser", resultClass: Credentials.class);
        Credentials credentials = query.setParameter(name: "username", value: username).getSingleResult();

        // Validar ids
        return credentials.getId().equals(obj:id);
    } catch (IllegalArgumentException | NoResultException e) {
        // Error en la decodificación o usuario no encontrado
        return false;
    }
}

@Override
protected EntityManager getEntityManager() {
    return em;
}
```

Aquest mètode és per actualitzar la informació d'un client existent. La idea és que el client passi el seu ID a la URL i les noves dades que vol actualitzar al cos de la sol·licitud. Conté

diverses anotacions que bàsicament serveixen perquè el servei REST sàpiga com manejar les sol·licituds:

- @PUT respon a sol·licituds HTTP PUT, que solen utilitzar-se per actualitzar recursos existents al servidor.
- @Path("id") defineix un paràmetre dinàmic a l'URL anomenat id. Això significa que el client ha d'enviar l'ID del client com a part de l'URL.
- @Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON}) indica que aquest mètode accepta sol·licituds en format XML o JSON.
- @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON}) especifica que la resposta que retorna el mètode estarà en format XML o JSON, depenent del que prefereixi el client.
- @Secured és una anotació personalitzada que indica que aquest mètode requereix autenticació o autorització.
- @HeaderParam("Authorization") permet accedir a un paràmetre de capçalera de la sol·licitud HTTP. En aquest cas, s'extreuen les credencials de l'usuari.

Explicació del funcionament del mètode:

1. S'obté el client de la base de dades usant un mètode super.find(id).
2. Si el client no és a la base de dades (existingCustomer == null), es retorna un codi d'error 404 (NOT_FOUND).
3. Abans de continuar, es verifica que l'usuari que està fent la sol·licitud sigui el mateix que el client la informació del qual es vol actualitzar.
4. Si el client intenta actualitzar el nom d'usuari o la contrasenya, es fa una cerca per verificar que el nou nom d'usuari no és utilitzat per un altre client. Si el username està en ús, es retorna un error NOT_ACCEPTABLE. Si no, s'actualitzen les credencials del client.
5. S'actualitzen els altres camps com ara el nom, l'URL de la imatge destacada, i els articles associats al client, però només si els nous valors no són nuls.
6. Un cop s'han actualitzat els camps, es desa el client actualitzat a la base de dades usant em.merge(existingCustomer) per assegurar que els canvis al client es persisteixin a la base de dades.
7. Si tot ha anat bé, es retorna una resposta amb el codi 200 OK i l'objecte del client actualitzat com a cos de resposta.

Alguns dels aspectes importants del disseny són:

- Control d'Errors: Si el client no es troba, respon amb 404.

- Cada camp s'actualitza només si és present a les dades enviades. Això evita sobreescriure valors amb null innecessàriament.

- Maneig de Credencials: Actualitza credencials (com a usuari i contrasenya) de forma separada, assegurant-se que siguin vàlides abans de desar.

- Creació de mètode addicional: Es crea el mètode isSameUser() per verificar que l'usuari autentificat modifica el seu propi compte i no el d'un altre. A més, aquesta funció ens estructura el codi per fer-ho més llegible.

Jocs de proves realitzats

Per assegurar-nos que el projecte funciona correctament en tots els casos possibles, hem realitzat un joc de proves extensiu i hem intentat cobrir tots els escenaris que podrien sorgir durant l'ús de l'aplicació. Hem utilitzat una combinació de tècniques de testing, incloent-hi TDD (Test-Driven Development) i BDD (Behavior-Driven Development) per garantir la qualitat del codi.

La plataforma API escollida per testar els nostres mètodes ha estat **Postman**. Les proves realitzades verifiquen casos positius i negatius, gestió d'excepcions i proves d'integració.

Per a la realització de les proves següents s'exposa que els paràmetres que no són comentats són escrits de forma correcta als testeigs. No són descrits a la taula per no sobrecarregar-la.

GET /rest/api/v1/article?topic=${topic}&author=${author}

Prova 1. Obtenir tots els articles de la pàgina.

Paràmetres: Cap.

Sortida esperada: Estat 200. Llista de tots els articles.

Correcte: Sí

Prova 2. Obtenir tots els articles d'un autor amb el seu nom formal.

Paràmetres: autor = Satxa

Sortida esperada: Estat 200. Llista de 2 articles.

Correcte: Sí

Prova 3. Obtenir tots els articles que tinguin un tòpic.

Paràmetres: tòpic = ["Software"]

Sortida esperada: Estat 200. Llista de 4 articles.

Correcte: Sí

Prova 4. Obtenir tots els articles que tinguin dos tòpics diferents.

Paràmetres: tòpic = ["Software", "AI"]

Sortida esperada: Estat 200. 1 article.

Correcte: Sí

Prova 5. Obtenir tots els articles d'un tòpic inexistent.

Paràmetres: tòpic = ["Empanadas"]

Sortida esperada: Estat 404.

Correcte: Sí

Prova 6. Obtenir tots els articles amb dos tòpics repetits.

Paràmetres: tòpic = ["Software", "Software"]

Sortida esperada: Estat 200. Llista de 4 articles.

Correcte: Sí

Prova 7. Obtenir tots els articles d'un autor amb dos tòpics.

Paràmetres: tòpic = ["Software", "CyberSecurity"] | autor = Satxa

Sortida esperada: Estat 200. 1 article.

Correcte: Sí

Prova 8. Obtenir tots els articles d'un autor amb tres tòpics.

Paràmetres: tòpic = ["Software", "Hardware", "CyberSecurity"]

Sortida esperada: Estat 404

Correcte: Sí

Prova 9. Obtenir tots els articles d'un autor utilitzant el username.

Paràmetres: autor = miriamduran

Sortida esperada: Estat 404.

Correcte: Sí

GET /rest/api/v1/article/${id}

Prova 1. Obtenir un article existent i públic.

Paràmetres: id = 1

Sortida esperada: Estat 200. Article sencer.

Correcte: Sí

Prova 2. Obtenir un article existent i privat, estant registrat.

Paràmetres: id = 6 | username = marcprofe | password = supersecurepassword

Sortida esperada: Estat 200. Article sencer.

Correcte: Sí

Prova 3. Obtenir un article existent i privat, no estant registrat.

Paràmetres: id = 6 | username = jovanivazquez | password = puertorico

Sortida esperada: Estat 401

Correcte: Sí

DELETE /rest/api/v1/article/${id}

Prova 1. Eliminar un article existent, sent l'autor d'aquest.

Paràmetres: id = 1 | username = sob | password = sob

Sortida esperada: Estat 200

Correcte: Sí

Prova 2. Eliminar un article existent, no sent l'autor d'aquest.

Paràmetres: id = 2 | username = miriamduran | password = 1234

Sortida esperada: Estat 401

Correcte: Sí

Prova 3. Eliminar un article no existent.

Paràmetres: id = 99 | username = satxa | password = password

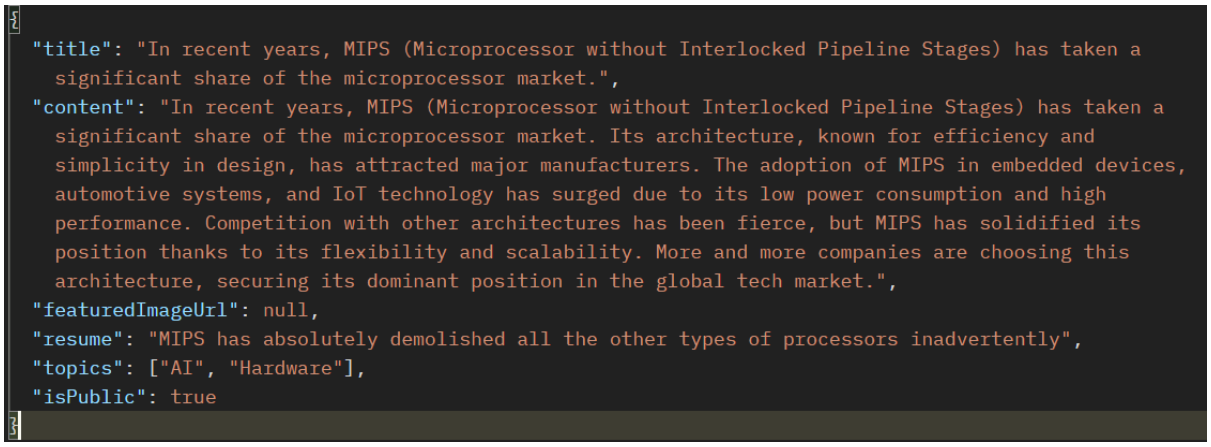
Sortida esperada: Estat 401

Correcte: Sí

POST/rest/api/v1/article

Totes les proves d'aquest mètode utlitzaran de entrada aquest document Json. Les proves consistiran basicament en anar treient informació per veure quina és esencial.

Donat que per obtindre l'article després del POST s'ha de fer un GET amb l'id i que depenent de l'orde en el que s'efectuin les proves l'id de l'article variarà, la manera més efectiva és mirar a la taula d'articles de la base de dades si s'ha fet l'inserció.



Prova 1. Inserir un article amb tots els paràmetres.

Paràmetres: Especificats a l'inici de la secció.

Sortida esperada: Estat 201.

Correcte: Sí

Prova 2. Inserir un article sense tòpics.

Paràmetres: Especificats a l'inici de la secció, exceptuant topics.

Sortida esperada: Estat 201.

Correcte: Sí

Prova 3. Inserir un article amb tres tòpics.

Paràmetres: Especificats a l'inici de la secció, afegint Software.

Sortida esperada: Estat 400.

Correcte: Sí

Prova 4. Inserir un article sense contingut.

Paràmetres: Especificats a l'inici de la secció, exceptuant content.

Sortida esperada: Estat 400.

Correcte: Sí

Prova 5. Inserir un article sense títol.

Paràmetres: Especificats a l'inici de la secció exceptuant titol.

Sortida esperada: Estat 400.

Correcte: Sí

Prova 6. Inserir un article privat.

Paràmetres: Especificats a l'inici de la secció, però private.

Sortida esperada: Estat 201.

Correcte: Sí

Prova 7. Inserir un article sense indicar privacitat.

Paràmetres: Especificats a l'inici de la secció, sense el isPublic.

Sortida esperada: Estat 201. L'haurà fet públic d'estandard.

Correcte: Sí

Prova 8. Inserir un article sense estar registrat.

Paràmetres: Especificats a l'inici de la secció i a la part de credencials, res.

Sortida esperada: Estat 401.

Correcte: Sí

PUT /rest/api/v1/customer/${id}

Prova 1. Verificar que el client està autentificat i és el mateix.

Paràmetres: {id} = 2 | username = miriamduran | password = 1234

Sortida esperada: Estat 200

Correcte: Sí

Prova 2. Verificar que les credencials d'autentificació no coincideixen.

Paràmetres: {id} = 2 | username = satxa | password = password

Sortida esperada: Estat 403

Correcte: Sí

Prova 3. Verificar que el id del client no es troba

Paràmetres: {id} = 5 | username = miriamduran | password = 1234

Sortida esperada: Estat 404

Correcte: Sí

Prova 4. Verificar quan el nom d'usuari ja està en ús.

Paràmetres: En forma de body.

```
{
    "credentials": {
        "username": "satxa"
    }
}
```

Sortida esperada: Estat 406

Correcte: Sí

Prova 5. Verificar que no es poden modificar articles.

Paràmetres: En forma de body.

```
{
    "articles": [
        {
```

```
            "id": 1,
            "title": "New title"
        }
    ]
}
```

Sortida esperada: Estat 403

Correcte: Sí

Prova 6. Verificar quan no s'envia la capçalera d'autorització.

Paràmetres: username= | password=

Sortida esperada: Estat 401

Correcte: Sí

Prova 7. Verificar un error intern del servidor.

Paràmetres: En forma de body.

```
{
    "name":
}
```

Sortida esperada: Estat 500

Correcte: Sí

Prova 8. Modificació correcta de les dades de l'usuari JSON.

Paràmetres: En forma de body.

```
{
    "name": "Miriam",
    "credentials":{
        "password": "12345678"
    }
}
```

Sortida esperada: Estat 200

Correcte: Sí

Prova 9. Modificació correcta de les dades de l'usuari XML.

Paràmetres: En forma de body.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<root>
   <credentials>
      <password>constrasenasegura</password>
   </credentials>
   <name>Marc</name>
</root>
```

Sortida esperada: Estat 200

Correcte: Sí

GET /rest/api/v1/customer

Prova 1. Obtenir tots els usuaris (i el link a l'últim article si en tenen)

Paràmetres: No

Sortida esperada: Estat 200 | Llistat customers

Correcte: Sí

Prova 2. No obtenir cap tipus de credencial en el JSON (username and password)

Paràmetres: No

Sortida esperada: Estat 200 | Cap username o password

Correcte: Sí

Prova 3. Actualització del link amb una adició d'un article

Paràmetres: No

Sortida esperada: Estat 200 | Link actualitzat

Correcte: Sí

GET /rest/api/v1/customer/${id}

Prova 1. Validar que el JSON no retorni la contrasenya o usuari.

Paràmetres: {id} = 1

Sortida esperada: Estat 200.

```
{
```

```
    "articles": [
        {
            "author": "sob",
            "public": true,
            "publishDate": "2024-01-20",
            "resume": "MIPS has absolutely demolished all the other types of processors inadvertently",
            "title": "MIPS takes over the market",
            "views": 0
        },
        {
            "author": "sob",
            "public": true,
            "publishDate": "2024-02-20",
            "resume": "IA has been advancing without stop, lets make a quick recap about it",
            "title": "Artificial Intelligence (AI) is revolutionizing industries worldwide, from healthcare to finance.",
            "views": 0
        },
        {
            "author": "sob",
            "public": true,
            "publishDate": "2024-02-20",
            "resume": "In today world we hear a lot about software but, what is it exactly?",
            "title": "Small bites about software",
            "views": 0
        }
    ],
    "id": 1,
    "user": "sob"
}
```


Correcte: Sí

Prova 2. Mostra els articles de l'usuari en versió curta (quan en té algun)

Paràmetres: {id} = 1

Sortida esperada: Estat 200.


```
{
    "articles": [
```

```
        {
            "author": "sob",
            "public": true,
            "publishDate": "2024-01-20",
            "resume": "MIPS has absolutely demolished all the other types of processors inadvertently",
            "title": "MIPS takes over the market",
            "views": 0
        },
        {
            "author": "sob",
            "public": true,
            "publishDate": "2024-02-20",
            "resume": "IA has been advancing without stop, lets make a quick recap about it",
            "title": "Artificial Intelligence (AI) is revolutionizing industries worldwide, from healthcare to finance.",
            "views": 0
        },
        {
            "author": "sob",
            "public": true,
            "publishDate": "2024-02-20",
            "resume": "In today world we hear a lot about software but, what is it exactly?",
            "title": "Small bites about software",
            "views": 0
        }
    ],
    "id": 1,
    "user": "sob"
}
```


Correcte: Sí

Prova 3. No mostra cap article en cas de que l'usuari no en tingui cap.

Paràmetres: {id} = 2

Sortida esperada: Estat 200.


```
{
    "articles": [],
    "id": 2,
```

```
    "user": "Miriam Durán"
}
```


Correcte: Sí

Prova 4. Actualitzar el contingut d'articles a mostrar una vegada s'ha inserit un de nou.

Paràmetres: {id} = 2

Sortida esperada: Estat 200.


```
{
    "articles": [
        {
            "author": "Miriam Durán",
            "public": true,
            "publishDate": "2024-12-04",
            "resume": "MIPS has absolutely demolished all the other types of processors inadvertently",
            "title": "In recent years, MIPS (Microprocessor without Interlocked Pipeline Stages) has taken a significant share of the microprocessor market.",
            "views": 0
        }
    ],
    "id": 2,
    "user": "Miriam Durán"
}
```


Correcte: Sí

Prova 5. Buscar dades d'un usuari que no existeix.

Paràmetres: {id} = 6

Sortida esperada: Estat 404.

Correcte: Sí

Scripts

Per tal de provar adequadament els mètodes a Postman s'han creat diferents scripts per comprovar els diferents resultats segons els paràmetres indicats.

CustomerService

Scripts per provar el mètode updateCustomer (PUT) a diferents escenaris:

```
pm.test("Status code is 200", function () {
      pm.response.to.have.status(200);
});

pm.test("Responde con Forbidden", function () {
      pm.response.to.have.status(403);
      pm.response.to.have.body("Les identificacions no coincideixen.");
});

pm.test("Responde con Not Found", function () {
      pm.response.to.have.status(404);
      pm.response.to.have.body("No s'ha trobat el teu usuari.");
});

pm.test("Username ya está en uso", function () {
      pm.response.to.have.status(406);
      pm.response.to.have.body("Aquest username ja està en ús!");
});

pm.test("No puedes modificar artículos", function () {
      pm.response.to.have.status(403);
      pm.response.to.have.body("No pots modificar articles!");
});

pm.test("Sin autorización", function () {
    pm.response.to.have.status(400);
});

pm.test("Error interno del servidor", function () {
      pm.response.to.have.status(500);
});
```

Scripts per provar el mètode getAllCustomers() (GET) i obtenir els links dels últims articles publicats pels autors:

```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});

pm.test("Response is an array", function () {
    const jsonData = pm.response.json();
    pm.expect(jsonData).to.be.an('array');
});

pm.test("Users who are authors have article links", function () {
    const jsonData = pm.response.json();
    jsonData.forEach(user => {
        if (user.links) {
            pm.expect(user.links).to.have.property('article');
        }
    });
});

pm.test("Users do not have sensitive information", function () {
    const jsonData = pm.response.json();
    jsonData.forEach(user => {
        pm.expect(user).to.not.have.property('password');
    });
});

pm.test("Users who are authors have article links", function () {
    const jsonData = pm.response.json();
    jsonData.forEach(user => {
        if (user.links) {
            pm.expect(user.links).to.have.property('article');
        }
    });
});
```

Scripts per provar el mètode getCustomerById() (GET) i obtenir les dades d'un usuari pel seu id:

```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});

pm.test("Response contains valid user data", function () {
    const jsonData = pm.response.json();
    pm.expect(jsonData).to.be.an('object');
    pm.expect(jsonData).to.have.property('id');
    pm.expect(jsonData).to.have.property('user');
    pm.expect(jsonData).to.not.have.property('password'); // No debe incluir la
contraseña
});

pm.test("Links are properly set if the user is an author", function () {
    const jsonData = pm.response.json();
    if (jsonData.links) {
        pm.expect(jsonData.links).to.have.property('article');
    }
});

pm.test("Response contains valid user data", function () {
    const jsonData = pm.response.json();
    pm.expect(jsonData).to.not.have.property('articles');
    pm.expect(jsonData).to.have.property('id');
    pm.expect(jsonData).to.have.property('user');
    pm.expect(jsonData).to.not.have.property('password'); // No debe incluir la
contraseña
});

pm.test("La respuesta tiene las propiedades esperadas", function () {
    let jsonData = pm.response.json();

    // Verificar que los campos principales existen
    pm.expect(jsonData).to.have.property("articles");
    pm.expect(jsonData).to.have.property("id");
    pm.expect(jsonData).to.have.property("user");

    // Verificar que 'articles' es un array
    pm.expect(jsonData.articles).to.be.an("array");
    pm.expect(jsonData.articles.length).to.be.greaterThan(0);
```

```
});

pm.test("Non existent customer", () => {
  pm.response.to.have.status(404);
});
```

ArticleService

En aquest la estructura és molt similar en tots els scripts. Primer tenim la comprovació de status. En funció del que vulguem revisar canviarem el numero del status i ja.

```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});
```

Després, si la prova ens ha de retornar un article, haurem de revisar el contingut d'aquest. Ho farem mirant si tenen les propietats d'un article.

```
pm.test("Response contains valid user data", function () {
    const jsonData = pm.response.json();
    pm.expect(jsonData).to.be.an('object');
    pm.expect(jsonData).to.have.property('id');
    pm.expect(jsonData).to.have.property('title');
    pm.expect(jsonData).to.have.property('content');
    pm.expect(jsonData).to.have.property('publishDate');
    pm.expect(jsonData).to.have.property('views');
    pm.expect(jsonData).to.have.property('topic');
    pm.expect(jsonData).to.have.property('authorName');
    pm.expect(jsonData).to.not.have.property('author');
});
```

Aquí veiem per exemple que no ha de tindre l'autor, ja que estem parlant del objecte Customer sencer, quan ho únic que volem és el nom d'aquest, que obtindrem pel authorName que s'obté amb un get del article que fa un get del autor.

També haure, de revisar quants tòpics s'han introduït finalment.

```
pm.test("Correct number of topic", function () {
    const jsonData = pm.response.json();
    pm.expect(jsonData.topic).to.have.length(1);
});
```

Abans ja s'han mirat les propietats d'un article, però aquest test en específic ens serveix per a quan ens retornen molts articles. Bàsicament fa la comprovació de les propietats però per cada article de la llista.

```
pm.test("Response contains valid article data", function () {
    const jsonData = pm.response.json();
    pm.expect(jsonData).to.be.an('array');
    jsonData.forEach(article => {
        pm.expect(article).to.be.an('object');
        pm.expect(article).to.not.have.property('id');
        pm.expect(article).to.have.property('title');
        pm.expect(article).to.not.have.property('content');
        pm.expect(article).to.have.property('publishDate');
        pm.expect(article).to.have.property('views');
        pm.expect(article).to.not.have.property('topics');
        pm.expect(article).to.not.have.property('authorName');
        pm.expect(article).to.have.property('author');
    });
});
```

I per últim tenim un test que ens permet saber quants articles ens ha retornat el get.

```
pm.test("Response contains 4 articles", function () {
    const jsonData = pm.response.json();
    pm.expect(jsonData).to.have.lengthOf(4);
});
```

És bàsicament com el dels tòpics, però és una variació que val molt la pena comentar ja que ens confirma la correctesa dels filtres d'autor i de tòpics.

Conclusions

Aquesta pràctica ha consistit a dissenyar, implementar i provar una API RESTful que permet gestionar usuaris i articles amb funcionalitats com la creació, modificació i consulta de dades. També hem aplicat mecanismes de seguretat, com l'autenticació per capçalera, i hem creat proves exhaustives per verificar que tots els endpoints funcionen correctament.

Què hem après de la pràctica?

Durant aquesta pràctica hem après a dissenyar API RESTful seguint bones pràctiques, incloent-hi codis de resposta coherents i estructures clares. També hem treballat amb bases de dades relacionals utilitzant JPA i gestionar correctament les relacions entre entitats (com Customer i Article).

Hem exercitat les validacions i filtres directament a les consultes a la base de dades per millorar l'eficiència. Al utilitzar eines com Postman per crear i executar jocs de proves automatitzats, ens ha ajudat a detectar errors ràpidament. I per últim, hem après a gestionar la seguretat amb autenticació i verificar permisos d'accés per a operacions sensibles.

Quines dificultats hem tingut?

Les principals dificultats han estat configurar les relacions bidireccionals com @OneToMany i @ManyToOne, assegurant la coherència de les dades als dos costats i gestionant correctament els articles associats als usuaris en les actualitzacions. També ens ha costat implementar la seguretat amb capçaleres Authorization, validant usuaris i permisos per garantir que només es permetessin accions autoritzades, així com gestionar errors de credencials. Finalment, automatitzar un joc de proves complet ha estat laboriós, especialment assegurar que les respostes fossin correctes en casos complexos amb filtres combinats i dades relacionades.

Tot i les dificultats, aquesta pràctica ens ha ajudat molt a entendre com funciona el desenvolupament d'APIs RESTful i com aplicar seguretat, relacions de dades i proves automatitzades.

No hi ha manera de que no comentem el mètode post. Realment ha sigut el mètode que més temps ens ha portat ja que és pràcticament la culminació de tots els coneixements de la pràctica i si alguna cosa del projecte estava malament el post ja es veia afectat.

Manual d’instal·lació

La instal·lació s'ha de reduir als següents passos:
1. Obrir projecte a NetBeans.

2. Connectar-se a la base de dades “sob_grup_08”.

3. Obrir el Glassfish

4. Donar-li a deploy de la vostra pràctica:

5. Donar-li a run i clicar al botó de "Install" a la web:

Després d'executar l'aplicació, accediu a la interfície web (si s'ha desenvolupat alguna) i feu clic al botó Install.

Opcionalment: Si aquest pas no funciona correctament, executeu manualment els scripts SQL proporcionats per crear la base de dades.

6. Passos per executar el client REST i fer proves:

5.1. Obrir Postman (des de la versió Desktop): Assegureu-vos que teniu Postman instal·lat a l'ordinador. Configureu els endpoints amb les URL i mètodes HTTP (GET, POST, PUT, DELETE) corresponents.

5.2. REST de Customer: Configureu Postman per interactuar amb l'endpoint http://localhost:8080/Homework1/rest/api/v1/customer. Això inclou:

- Afegir el mètode HTTP correcte.
- Configurar els headers necessaris, com Content-Type o Authorization.
- Provar amb payloads JSON/XML si és necessari (per mètodes com POST o PUT).

5.3. REST d’Article: Feu el mateix que amb el punt anterior, però per a l'endpoint d'articles: http://localhost:8080/Homework1/rest/api/v1/article.

Manual per executar el joc de proves:

A continuació, s'expliquen els passos per executar les proves automatitzades incloses a la col·lecció Homework1.postman_collection.json. Aquest joc de proves valida els endpoints definits a la teva API per als serveis de Customer i Article.

1. Importar la col·lecció a Postman.

- Obre Postman (versió Desktop).
- A la part superior esquerra, seleccioneu Import.
- Seleccioneu el fitxer Homework1.postman_collection.json que heu descarregat.
- La col·lecció s'afegirà al vostre espai de treball sota el nom Homework1.

2. Configurar l'URL base

- Abans d'executar les proves, assegureu-vos que el vostre servidor està en funcionament a la URL correcta.
- La URL base utilitzada és http://localhost:8080/Homework1.
- Si el vostre entorn és diferent (per exemple, un altre port o ruta), actualitzeu manualment els URL dins de la col·lecció o utilitzeu variables d'entorn:
- Ves a Environment a Postman.
- Crea una variable anomenada baseUrl amb el valor http://localhost:8080/Homework1.
- Reemplaça les URLs a la col·lecció amb {{baseUrl}}/rest/api/v1/....

3. Estructura de la col·lecció

La col·lecció té dues seccions principals:

CustomerService:

- GET /customer: Obté una llista de tots els clients.
- GET /customer/{id}: obté els detalls d'un client per ID.
- PUT /customer/{id}: Modifica les dades d'un client existent.

ArticleService:

- GET /article: Llista tots els articles amb filtres opcionals.
- GET /article/{id}: Obté els detalls d'un article per ID.

- DELETE /article/{id}: Elimina un article si l'usuari està autenticat i és l'autor.
- POST /article: Crea un nou article.

4. Execució de proves

- Proves individuals
- Selecciona qualsevol sol·licitud dins de la col·lecció, es recomana fer-les en ordre per tal de que els scripts tinguin sentit en el cas de modificacions.
- Feu clic a Send per enviar la sol·licitud.
- Observa la resposta a la part inferior. Es mostraran els resultats esperats, com ara un codi d'estat 200 o un cos JSON.

  Execució de totes les proves:

- Aneu al menú principal de Postman i seleccioneu Collection Runner.
- Seleccioneu la col·lecció Homework1.
- Feu clic a Run per executar totes les sol·licituds en ordre.

Els resultats de cada prova apareixeran a la finestra d'execució, mostrant si les proves passen o fallen.

5. Resultats esperats

  Status codes:

- 200: Sol·licituds reeixides.
- 404: Recursos no trobats.
- 403: Prohibit (per exemple, accés a un article privat sense autenticació).
- 201: Recurs creat amb èxit.

  Validacions automàtiques:

- Dades tornades correctament (per exemple, usuaris sense contrasenyes).
- Enllaços de HATEOAS per a usuaris que són autors.
- Filtres per topic i author en articles.

6. Personalització

- Si necessiteu canviar credencials de prova (per exemple, username i password), actualitzeu els headers Authorization a cada sol·licitud.

- Per provar usuaris o articles específics, editeu els valors dels IDs directament als URL.