ESCOLA TÈCNICA SUPERIOR D'ENGINYERIA
Universitat Rovira i Virgili

[DΣIM]

# Sistemes Oberts

# Pràctica 2

**Components**: Satxa Fortuny

Miriam Durán

**Professor de Laboratori:** Marc Sánchez Artigas

# Índex

# Introducció

En aquesta segona pràctica del curs, l'objectiu principal és aplicar els coneixements adquirits en el desenvolupament d'aplicacions web per construir una aplicació completa basada en l'API Web creada durant el primer lliurament. Aquesta pràctica es divideix en diverses parts per abordar tant la funcionalitat pública com la privada de l'aplicació, afegint elements de seguretat, usabilitat i disseny adaptatiu.

El desenvolupament d'aquesta aplicació web representa una oportunitat per explorar el model MVC, integrar les especificacions de backend REST, i aplicar tècniques avançades de desenvolupament, com ara la gestió d'autenticació d'usuaris i la implementació d'interfícies interactives. A més, es potenciaran habilitats en l'ús d'eines com NetBeans, i en la documentació i presentació d'un projecte professional.

Aquest document descriu detalladament els objectius, requisits tècnics, i criteris d'avaluació associats al lliurament de la pràctica. Alhora, proporciona les directrius necessàries per estructurar el codi i documentar-lo adequadament.

# Estructura de la pràctica

L'estructura del projecte segueix un enfocament organitzat i modular basat en el patró arquitectònic MVC (Model-View-Controller). Els fitxers i carpetes estan dissenyats per separar les responsabilitats clau, facilitant la gestió, el manteniment i l'escalabilitat del projecte.

Aquesta estructura es combina amb el backend desenvolupat en la primera part de la pràctica, que es basa en una API REST. Aquest backend gestiona la comunicació entre la base de dades i la interfície d'usuari, permetent l'autenticació d'usuaris, la gestió d'articles, i altres operacions relacionades amb la funcionalitat de l'aplicació. La integració de l'API REST amb el model MVC garanteix que la lògica de negoci, emmagatzemada en el backend, es comunica de manera eficaç amb els components del frontend, oferint una experiència d'usuari dinàmica i segura.

A continuació es detalla l'estructura i funció principal de cadascun dels components:

## 1. Controladors

Aquesta carpeta conté les classes responsables de gestionar la interacció entre la vista i el model. Les funcionalitats principals són:

- **Gestió dels articles**: ArticleController.java
- **Gestió de l'autenticació**: LoginController.java, LogoutController.java
- **Gestió de perfils**: ProfileController.java, ModifiedProfileController.java
- **Registre d'usuaris**: SignUpFormController.java
- **Utilitats addicionals**: CustomerViewBean.java, CustomerUpdate.java

## 2. Excepcions

Aquesta carpeta inclou les classes per gestionar errors específics de l'aplicació. Actualment, només inclou:

- **PropertyException.java**: Maneig d'errors relacionats amb les propietats del sistema.

## 3. Model

El model representa les dades i les regles de negoci de l'aplicació. Inclou classes que defineixen els objectes centrals del projecte:

- **Articles**: Article.java, ArticleDTO.java
- **Clients**: Customer.java, CustomerDTO.java
- **Tòpics**: Topic.java
- **Missatges d'alerta i intents de registre**: AlertMessage.java, SignUpAttempts.java
- **Credencials i enllaços**: ViewCredentials.java, Link.java

## 4. Recursos

Aquesta carpeta conté fitxers i configuracions per gestionar els recursos compartits de l'aplicació, incloent-hi:

- **Configuració Jakarta EE**: JakartaEE9Resource.java

## 5. Serveis

La lògica de negoci es gestiona en aquesta carpeta. Els serveis proporcionen funcionalitats per interactuar amb els models i els controladors. Conté:

- **Serveis d'articles**: ArticleService.java, ArticleServiceImpl.java
- **Serveis de clients**: CustomerService.java, CustomerServiceImpl.java
- **Utilitats diverses**: BeanUtilHelper.java

# Decisions de disseny

## Customers

Per implementar les funcions relacionades amb els usuaris, com el registre, l'inici de sessió i la modificació de dades, s'ha dissenyat una arquitectura clara i funcional.

Abans d'explicar la segona fase dissenyada es farà un breu resum de les modificacions afegiments a la fase 1 per tal de mantenir coherència amb les especificacions de la pràctica.

### Homework1

#### *Customer (model)*

```
    private String firstName;
    private String lastName;
    private String email;
```

La primera modificació ha estat la separació del nom en **firstName** i **lastName** i l'agregació del correu electrònic.

Aquestes dades són necessàries per l'enregistrament de nous usuaris i per tindre més seguretat en l'addició d'ells.

Conseqüentment, aquests canvis també s'han reflectit al **CustomerDTO**.

#### *EmailValidator (model)*

La classe **EmailValidator** serveix per comprovar si una adreça de correu electrònic té el format correcte. Utilitza una expressió regular per validar si l'**email** segueix el patró estàndard d'un correu electrònic vàlid. Té un mètode anomenat **isValid**, que rep l'**email** com a paràmetre i retorna **true** si és vàlid o **false** si no ho és. És útil per evitar que es guardin adreces de correu mal escrites o incorrectes.

### *CustomerService (service)*

S'han agregat dos mètodes nous, **authenticateCustomer** i **registerCustomer**.

El primer fa un get a partir de l'url "Customer/auth" per obtenir totes les dades del customer passant les credencials encapçalats.

```
    @GET
    @Path("auth")
    @Produces(MediaType.APPLICATION_JSON)
    public Response authenticateCustomer(@HeaderParam("Authorization") String
reg)
```

El segon fa un post per registrar un nou usuari, validant que totes les dades siguin correctes.

```
    @POST
    @Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public Response registerCustomer(Customer nouCustomer)
```

Tots dos mètodes segueixen la lògica dels anteriors fets i, tot i que s'han creat de nous, no indica que els anteriors no s'utilitzaran també.

A més d'això, s'han modificat els ja implementats per tal de arreglar i garantir un millor i òptim funcionament de les funcions. Els canvis realitzats han estat:

- Garantir que tan variables a **null** com variables buides (' ') no passin la validesa en l'enregistrament o actualització de customers.
- Afegir la validació del **correu electrònic**.
- Millorar **comentaris** de sortida en cas d'errors.
- Garantir que els correus electrònic **no es repeteixin** entre usuaris.

## Homework2

Una vegada explicat l'apartat anterior ja podem continuar amb l'explicació del disseny d'aquesta segona part.

### *Model*

Per representar les dades dels customers i continuar la relació amb Homework1 s'afegeixen les classes java següents:

- **Customer.java**

Classe necessària per obtenir els objectes de tipus **Customer** que obtenim del backend i que inclourem en la sessió per confirmar l'autentificació de l'usuari.

- **CustomerDTO.java**

Classe necessària per obtenir els objectes de tipus **CustomerDTO** que obtenim del backend.

- **Link.java**

Classe necessària per obtenir els objectes de tipus **Link** que obtenim del backend.

- **ViewCredentials.java**

Classe necessària per comunicar-se amb els objectes de tipus **Credentials** que hi han al backend i que aporten informació a l'inici de sessió de l'usuari.

### *Controller*

Abans de desenvolupar la descripció dels controladors, s'exposaran els diferents formularis que s'han creat i els seus usos:

#### Credentials

```
    @Named("credentials")
    @RequestScoped
    public class Credentials implements Serializable {
        private static final long serialVersionUID = 1L;

        // JSR 303 validation for `username`
        @NotBlank(message = "Username cannot be empty")
        @Size(min = 4, max = 20, message = "Username must be between 4 and 20
characters")
        @FormParam("username")
```

```
        private String username;

        // JSR 303 validation for `password`
        @NotBlank(message = "Password cannot be empty")
        @Size(min = 8, max = 30, message = "Password must be between 8 and 30
characters")
        @FormParam("password")
        private String password;
```

El formulari s'utilitza per capturar i validar les dades d'usuari que introdueix un client, com el *nom d'usuari* (username) i la *contrasenya* (password). Aquest formulari es fa servir iniciar sessió.

Apartats interessants:

- Es fa servir **Bean Validation** per assegurar que els valors introduïts compleixin certs requisits.
- @**NotBlank**: S'assegura que el nom d'usuari i la contrasenya no siguin buits.
- @**Size**: Defineix els límits mínims i màxims de longitud per al nom d'usuari (4-20 caràcters) i per la contrasenya (8-30 caràcters).
- @**FormParam**: indica que els valors de username i password seran proporcionats a través d'un formulari HTML. Aquestes dades es capturen des de la petició HTTP i es llegeixen per ser utilitzades posteriorment en la lògica de l'aplicació.

### CustomerViewBean

```
        @NotBlank
        @Size(min=2, max=30, message = "First name must be between 2 and 30
characters")
        @FormParam("firstName")
        private String firstName;

        @NotBlank
        @Size(min=2, max=30, message = "Last name must be between 2 and 30
characters")
        @FormParam("lastName")
        private String lastName;

        @NotBlank
        @Email(message = "Email should be valid")
        @FormParam("email")
        private String email;

        @NotBlank
```

```
        @FormParam("username")
        private String username;

        @NotBlank
        @FormParam("password")
        private String password;

        @NotBlank
        @FormParam("featuredImageUrl")
        private String featuredImageUrl;
```

Aquest formulari s'encarrega de recollir i validar les dades bàsiques de l'usuari, així com altres dades relacionades amb el contingut personalitzat (articles i enllaços). La seva implementació amb validacions i formulació adequada assegura que les dades introduïdes al SignUp siguin correctes.

Apartats interessants:

- Es fa servir **Bean Validation** per assegurar que els valors introduïts compleixin certs requisits.
- @**NotBlank**: S'assegura que els camps no siguin buits.
- @**Size**: Es defineixen límits de mida per als camps de nom i cognom, establint un rang de caràcters entre 2 i 30.
- @**Email**: S'assegura que l'email tingui el format correcte.

### CustomerUpdate

```
    @Named("customerView")
    public class CustomerViewBean {
        @Size(min=2, max=30, message = "First name must be between 2 and 30
characters")
        @FormParam("firstName")
        private String firstName;

        @Size(min=2, max=30, message = "Last name must be between 2 and 30
characters")
        @FormParam("lastName")
        private String lastName;

        @Email(message = "Email should be valid")
        @FormParam("email")
        private String email;

        @NotBlank(message = "Username cannot be empty")
        @Size(min = 4, max = 20, message = "Username must be between 4 and 20
characters")
```

```
        @FormParam("username")
        private String username;

        @NotBlank(message = "Password cannot be empty")
        @Size(min = 8, max = 30, message = "Password must be between 8 and 30
characters")
        @FormParam("password")
        private String password;

        @FormParam("featuredImageUrl")
        private String featuredImageUrl;
```

CustomerUpdate es fa servir per gestionar el procés de modificació de dades d'un client. Aquesta classe funciona com a formulari de recollida i actualització de la informació d'un usuari en el sistema.

Apartats interessants:

- Es fa servir **Bean Validation** per assegurar que els valors introduïts compleixin certs requisits.
- @**NotBlank**: S'assegura que els camps no siguin buits.
- @**Size**: Es defineixen límits de mida per als camps de nom i cognom, establint un rang de caràcters entre 2 i 30. Defineix els límits mínims i màxims de longitud per al nom d'usuari (4-20 caràcters) i per la contrasenya (8-30 caràcters).
- @**Email**: S'assegura que l'email tingui el format correcte.

Passem a explicar els controladors:

### ProfileController

```
@Path("Profile")
@Controller
public class ProfileController {
    @Inject BindingResult bindingResult;
    @Inject Logger log;
    @Inject CustomerService service;
    @Inject Models models;
    @Inject AlertMessage flashMessage;
    @Inject SignUpAttempts attempts;
    @Inject HttpServletRequest request;

    @GET
    public String showProfilePage(@Context HttpServletRequest request) {
        Customer customer = (Customer) request.getSession().getAttribute(string:"customer");
        if (customer == null) {
            return "redirect:/Login"; // Redirigir al login si no está autenticado
        }
        models.put(string:"customer", o: customer);
        return "profile-page.jsp"; // Vista para mostrar el perfil del usuario
    }

    @Path("Articles")
    @GET
    public String showProfileArticles(@Context HttpServletRequest request) {
        Customer customer = (Customer) request.getSession().getAttribute(string:"customer");
        if (customer == null) {
            return "redirect:/Login"; // Redirigir al login si no está autenticado
        }
        models.put(string:"customer", o: customer);
        return "CustomerArticles.jsp"; // Vista para mostrar el perfil del usuario
    }

}
```

Aquest controlador gestiona dues accions principals:

1. Mostrar la pàgina de perfil de l'usuari.
2. Mostrar els articles associats al perfil de l'usuari.

En ambdós casos, assegura que l'usuari estigui autentificat abans de permetre l'accés. Si no ho està, redirigeix a la pàgina de Login.

Explicació del codi:

1. Anotacions de la classe:

- **@Path("Profile")**: Defineix el prefix de la ruta URL associada a aquest controlador. Per exemple, qualsevol acció dins d'aquest controlador tindrà una ruta que començarà amb /Profile.
- **@Controller**: Indica que aquesta classe és un controlador que gestiona peticions HTTP.

2. Injeccions:

- **BindingResult** bindingResult: Per gestionar errors de validació.

- **Logger** log: Per registrar esdeveniments o errors.
- **CustomerService** service: Servei per a la lògica de negoci relacionada amb el client.
- **Models** models: Per passar dades a les vistes.
- **AlertMessage** flashMessage: Per mostrar missatges d'alerta.
- **SignUpAttempts** attempts: Per gestionar intents de registre.
- **HttpServletRequest** request: Per obtenir informació de la petició HTTP.

3. Mètode showProfilePage:

- @**GET**: Indica que aquest mètode es cridarà quan es faci una petició HTTP GET.
- Customer customer = (Customer) request.getSession().getAttribute("customer");: Obté l'objecte customer de la sessió. Si no hi ha cap client autentificat, el valor serà null.
- if (customer == null): Verifica si l'usuari està autentificat.
- Si no ho està, redirigeix a la pàgina de login (redirect:/Login).
- models.put("customer", customer);: Si l'usuari està autentificat, afegeix el client al model per passar-lo a la vista.
- return "profile-page.jsp";: Retorna la vista JSP per mostrar la pàgina del perfil de l'usuari.

4. Mètode showProfileArticles: Aquest mètode és molt similar al showProfilePage, però mostra una vista diferent:

- La ruta associada a aquest mètode és /Articles gràcies a l'anotació @Path("Articles").
- La vista retornada és CustomerArticles.jsp, que mostra una llista d'articles associats al perfil de l'usuari.

## LoginController

```
@Controller
@Path("Login")
public class LoginController {

    @Inject BindingResult bindingResult;
    @Inject Logger log;
    @Inject CustomerService service;
    @Inject Models models;
    @Inject AlertMessage flashMessage;
    @Inject SignUpAttempts attempts;
    @Inject HttpServletRequest request;
    //@Inject HttpServletResponse response;



    @GET
    public String showForm() {
        return "login-form.jsp"; // Inyecta el token CSRF
    }


    @POST
    @UriRef("log-in")
    @CsrfProtected
    public String authenticateUser(@Valid @BeanParam Credentials credentials) throws IOException {
        if (bindingResult.isFailed()) {
            AlertMessage alert = AlertMessage.danger(text: "Error de validació.");
            bindingResult.getAllErrors().forEach(t -> alert.addError(field: t.getParamName(), code: "", message: t.getMessage()));
            log.log(level: Level.WARNING, msg:"Error de validació al processar el formulari de inici de sessió.");
            models.put(string:"errors", o: alert);
            return "login-form.jsp";
        }
```

```
// Crear i copiar dades a un ViewCredentials
ViewCredentials viewCredentials = new ViewCredentials();
BeanUtilHelper.copyCredentials(source:credentials, target:viewCredentials);

try {
    Customer customer = service.authenticateCustomer(credentials: viewCredentials);
    if (customer == null) {
        log.log(level: Level.WARNING, msg: "Usuari o contrasenya incorrectes. Intenta-ho de nou.");
        models.put(string:"message", o: "Usuari o contrasenya incorrectes. Intenta-ho de nou.");
        attempts.increment();
        return "login-form.jsp";
    }
    log.log(level: Level.INFO, msg:"Inici de sessió amb èxit per l'usuari: {0}", param1:customer.getUsername());
    attempts.reset();
    CustomerViewBean viewCustomer = new CustomerViewBean();
    ViewCredentials credencials2 = new ViewCredentials();
    viewCustomer.setCredentials(credentials: credencials2);
    BeanUtilHelper.copyCustomer(source:customer, target:viewCustomer);

    models.put(string:"customer", o: customer);
    // Recuperar la URL de redirección desde la sesión
    HttpSession session = request.getSession();
    session.setAttribute(string:"customer", o: customer); // 'customer' se guarda en la sesión
    session.setAttribute(string:"customerView", o: viewCustomer);
    String redirectUrl = (String) session.getAttribute(string:"redirectUrl");
    if (redirectUrl != null) {
        session.removeAttribute(string:"redirectUrl"); // Limpiar la URL después de usarla
        //response.sendRedirect(redirectUrl); // Redirigir a la página anterior
        return null; // No es necesario devolver una vista ya que hemos hecho una redirección
    }

    // Si no hay URL de redirección, redirigir a la página de perfil
    return "redirect:/Profile"; // Tornar null perquè la redirecció ja s'ha fet
} catch (PropertyException ex) {
    log.log(level: Level.WARNING, msg: "Usuari o contrasenya incorrectes. Intenta-ho de nou.");
    models.put(string:"message", o: "Usuari o contrasenya incorrectes. Intenta-ho de nou.");
    AlertMessage.danger(text: ex.getMessage());
    attempts.increment();
    return "login-form.jsp";
}
```

Aquest controlador gestiona el Login dels usuaris:

1. Mostra el formulari de login (amb **showForm**).
2. Processa les credencials enviades (amb **authenticateUser**):
    a. Verifica errors de validació.
    b. Autentica l'usuari amb el servei corresponent.
    c. Gestiona intents d'autenticació fallits o reeixits.
    d. Guarda l'usuari a la sessió i el redirigeix a la pàgina corresponent.

Explicació del codi:

1. Anotacions de la classe:

- @**Controller**: Indica que aquesta classe és un controlador que gestiona peticions HTTP.
- @**Path**("Login"): Defineix el prefix de la ruta URL associada a aquest controlador. Les peticions gestionades per aquest controlador començaran amb /Login.

2. Injeccions:

- **BindingResult** bindingResult: Per gestionar errors de validació del formulari.
- **Logger** log: Per registrar esdeveniments o errors.
- **CustomerService** service: Servei per a la lògica de negoci relacionada amb l'usuari.
- **Models** models: Per passar dades a les vistes.
- **AlertMessage** flashMessage: Per mostrar missatges d'alerta.
- **SignUpAttempts** attempts: Per gestionar intents d'autenticació fallits.
- **HttpServletRequest** request: Per obtenir informació de la petició HTTP.

3. Mètode showForm:

- @**GET**: Aquest mètode es cridarà quan es faci una petició HTTP GET.
- **return** "login-form.jsp";: Retorna la vista del formulari de login. Aquesta vista podria contenir un token CSRF per protegir-se contra atacs d'aquest tipus.

4. Mètode authenticateUser:

- Customer customer = service.authenticateCustomer(viewCredentials);: Utilitza el servei per autenticar l'usuari.
- Si l'autenticació falla (customer == null), es mostren missatges d'error i es retorna al formulari.
- Si l'autenticació és correcta:
- Es registra l'èxit al log.
- Es guarda l'usuari a la sessió (session.setAttribute("customer", customer)).

- Si hi ha una URL de redirecció guardada a la sessió (redirectUrl), redirigeix l'usuari a aquesta URL.
- Si no hi ha URL de redirecció, redirigeix a la pàgina del perfil (redirect:/Profile).

5. Línies importants:

- **@CsrfProtected:** És crucial per protegir aquest endpoint contra atacs CSRF, ja que el mètode gestiona dades sensibles (usuari i contrasenya).
- **bindingResult.isFailed()**: Permet verificar si les dades enviades pel formulari són vàlides.
- **service.authenticateCustomer(viewCredentials)**: Crida al servei d'autenticació per validar l'usuari.
- **session.setAttribute("customer", customer)**: Guarda l'usuari autentificat a la sessió, cosa que permet identificar-lo en altres parts de l'aplicació.
- **redirect:/Profile**: Si no hi ha URL de redirecció específica, envia l'usuari al seu perfil després d'un Login amb èxit.

## ModifiedProfileController

```
@Path("ModifyProfile")
@Controller
public class ModifiedProfileController {
    @Inject BindingResult bindingResult;
    @Inject Logger log;
    @Inject CustomerService service;
    @Inject Models models;
    @Inject AlertMessage flashMessage;
    @Inject SignUpAttempts attempts;
    @Inject HttpServletRequest request;

    private Customer customer;
    // Handle the profile modification page
    @GET
    @Controller
    public String showModifyPage(@Context HttpServletRequest request) {
        this.customer = (Customer) request.getSession().getAttribute(string:"customer");
        if (this.customer == null) {
            models.put(string:"error", new Error(message: "Session expired. Please log in again."));
            return "redirect:/Login"; // Redirect to login if customer is not found
        }
        return "profile-modify.jsp"; // View for profile modification
    }

    // Update the customer profile (changed to POST for compatibility with HTML forms)
    @POST
    @UriRef("modify")
    @CsrfProtected
    public String updateCustomer(@Valid @BeanParam CustomerUpdate customerForm) {
        if (bindingResult.isFailed()) {
            AlertMessage alert = AlertMessage.danger(text: "Validation failed!");
            bindingResult.getAllErrors().forEach((ParamError t) -> {
                alert.addError(field: t.getParamName(), code: "", message: t.getMessage());
            });
        log.log(level: Level.WARNING, msg:"Data binding for CustomerForm failed.");
        models.put(string:"errors", o: alert);
        return "signup-form.jsp"; // Redisplay the modification form with errors
    }

    // Obtener el cliente de la sesión
    this.customer = (Customer) request.getSession().getAttribute(string:"customer");
    if (this.customer == null) {
        models.put(string:"error", new Error(message: "Session expired. Please log in again."));
        return "redirect:/Login"; // Redirect to login if customer is not found
    }

    // Crear objeto de credenciales
    ViewCredentials credentials = new ViewCredentials(username: this.customer.getUsername(), password: this.customer.getPassword());

    // Crear objeto Customer con los datos del formulario
    Customer updatedCustomer = new Customer();
    updatedCustomer.setFirstName(firstName:customerForm.getFirstName());
    updatedCustomer.setLastName(lastName: customerForm.getLastName());
    updatedCustomer.setEmail(email: customerForm.getEmail());
    ViewCredentials cred = new ViewCredentials(username: customerForm.getUsername(), password: customerForm.getPassword());
    updatedCustomer.setCredentials(credentials: cred);  // Ensure credentials are validated

    // Llamar al servicio para actualizar el cliente
    try {
        service.updateCustomer(id: this.customer.getCredentials().getId(), credentials, updatedData: updatedCustomer);
    } catch (Exception e) {
        log.log(level: Level.SEVERE, "Error during updating: " + e.getMessage(), thrown:e);
        models.put(string:"message", o: e.getMessage());  // Pasar el mensaje de error a la vista
        return "profile-modify.jsp";  // Redirigir a la página de error
    }
    // Redirigir a la página de perfil
    return "redirect:/Logout/Success";
}
```

Aquest controlador té dos mètodes principals:

1. **showModifyPage**: Mostra el formulari de modificació de perfil.
2. **updateCustomer**: Processa les dades enviades i actualitza el perfil de l'usuari.

A més, garanteix la seguretat amb proteccions CSRF, validació de formularis i verificacions d'autenticació.

Explicació del codi:

1. Anotacions de la classe:

- @Path("ModifyProfile"): Defineix el prefix de la ruta URL associada a aquest controlador. Les peticions gestionades per aquest controlador començaran amb /ModifyProfile.
- @Controller: Indica que aquesta classe és un controlador que gestiona peticions HTTP.

2. Injeccions:

- **BindingResult bindingResult**: Permet detectar i gestionar errors de validació dels formularis.
- **Logger log**: Registra esdeveniments i errors.
- **CustomerService service**: Servei per gestionar operacions relacionades amb el client.
- **Models models**: Per passar dades a les vistes.
- **AlertMessage flashMessage**: Per gestionar missatges d'alerta a la vista.
- **SignUpAttempts attempts**: Per controlar intents fallits d'acció.
- **HttpServletRequest request**: Per accedir a la sessió de l'usuari i altres dades de la petició.

3. Mètode showModifyPage: Aquest mètode gestiona la vista per modificar el perfil.

- **@GET**: Indica que aquest mètode es cridarà quan es faci una petició HTTP GET.
- **Obtenció del client de la sessió**:
- this.customer = (Customer) request.getSession().getAttribute("customer");: Es recupera l'objecte Customer de la sessió.
- **Verificació d'autenticació**:
- Si el client no existeix a la sessió (this.customer == null), es mostra un missatge d'error i es redirigeix a la pàgina de login (redirect:/Login).
- **Retorn de la vista**:
- Si el client està autentificat, es retorna la vista profile-modify.jsp, que mostra el formulari per modificar el perfil.

4. Mètode updateCustomer: Aquest mètode gestiona la modificació del perfil després d'enviar el formulari.

- @**POST**: Indica que aquest mètode es crida quan es fa una petició HTTP POST.
- @**UriRef("modify")**: Assigna una referència URI al mètode.
- Si hi ha errors de validació (bindingResult.isFailed()), es mostra un missatge d'error i es retorna a la vista amb els errors (signup-form.jsp).
- Igual que al mètode anterior, es comprova si l'usuari està a la sessió. Si no ho està, es redirigeix al Login.
- Es crea un nou objecte Customer amb les dades del formulari (customerForm).
- Es validen les credencials de l'usuari amb un objecte ViewCredentials.
- service.updateCustomer(...): Es crida al servei per actualitzar el perfil del client a la base de dades.
- Si hi ha errors durant l'actualització, es registra l'error i es mostra un missatge a la vista.
- Si l'actualització és exitosa, l'usuari és redirigit a /Logout/Success.

5. Línies importants:

- **this.customer = (Customer) request.getSession().getAttribute("customer");**:
  Recupera el client de la sessió per comprovar que està autentificat.
- **if (this.customer == null)**:
  Garanteix que només usuaris autentificats poden accedir a les funcionalitats.
- **service.updateCustomer(...)**:
  Actualitza les dades del client a través del servei.
- **Gestió d'excepcions**:
  En cas d'error durant l'actualització es mostra un missatge a la vista.

### *Service*

## CustomerService

Aquesta interfície està dissenyada per gestionar la lògica del negoci relacionada amb els clients dins l'aplicació. És a dir, els mètodes que conté són els que tenim al backend, d'aquesta forma ens comuniquem amb les API REST de Homework1 d'una manera més organitzada.

```
public interface CustomerService {
    public List<CustomerDTO> getAllCustomers();
    public CustomerDTO getCustomerById(Long id);
    public void updateCustomer(Long id, ViewCredentials credentials, Customer updatedData);
    public Customer authenticateCustomer(ViewCredentials credentials);
    public boolean registerCustomer(Customer customer);
}
```

## CustomerServiceImpl

Classe Java on s'implementen els mètodes anteriors.

1. getAllCustomers()

```
@Override
public List<CustomerDTO> getAllCustomers() {
    try {
        // Solicita el "GET /customer"
        Response response = webTarget.request(strings: MediaType.APPLICATION_JSON).get();

        if (response.getStatus() == Response.Status.OK.getStatusCode()) {
            // Convertir la respuesta JSON en una lista de CustomerDTO
            CustomerDTO[] customers = response.readEntity(type: CustomerDTO[].class);
            return Arrays.asList(a: customers);
        } else {
            throw new RuntimeException("Error fetching customers: " + response.getStatusInfo().toString());
        }
    } catch (RuntimeException e) {
        throw new RuntimeException(message: "Error retrieving customers", cause: e);
    }
}
```

Aquest mètode fa una petició GET a l'endpoint /customer per obtenir una llista de tots els clients.

- La resposta JSON es converteix en una llista d'objectes CustomerDTO.
- Si la resposta no és satisfactòria, es llança una excepció amb el codi d'estat.

2. getCustomerById(Long id)

```java
@Override
public CustomerDTO getCustomerById(Long id) {
    try {
        // Solicita el "GET /customer/{id}"
        Response response = webTarget.path(string:String.valueOf(obj:id))
                .request(strings:MediaType.APPLICATION_JSON)
                .get();

        if (response.getStatus() == Response.Status.OK.getStatusCode()) {
            // Convertir la respuesta JSON en un objeto CustomerDTO
            return response.readEntity(type:CustomerDTO.class);
        } else if (response.getStatus() == Response.Status.NOT_FOUND.getStatusCode()) {
            throw new RuntimeException("Customer not found for ID: " + id);
        } else {
            throw new RuntimeException("Error fetching customer: " + response.getStatusInfo().toString());
        }
    } catch (RuntimeException e) {
        throw new RuntimeException(message:"Error retrieving customer by ID", cause:e);
    }
}
```

Obté un client específic a partir del seu ID fent una petició GET a /customer/{id}.

- Es processa la resposta JSON i es converteix a un objecte CustomerDTO.
- Si el client no existeix, retorna una excepció amb el missatge "Customer not found".

3. updateCustomer(Long id, ViewCredentials credentials, Customer updatedData)

```java
@Override
public void updateCustomer(Long id, ViewCredentials credentials, Customer updatedData) {
    try {
        // Construir el encabezado Authorization para Basic Authentication
        String username = credentials.getUsername();
        String password = credentials.getPassword();

        // Concatenar username:password
        String credencials = username + ":" + password;

        // Codificar las credenciales en Base64
        String encodedCredentials = Base64.getEncoder().encodeToString(src:credencials.getBytes(charset:StandardCharsets.UTF_8));
        log.log(level:Level.WARNING, msg:"Data encoded: {0}", param1:encodedCredentials);

        ObjectMapper mapper = new ObjectMapper();
        String jsonPayload = mapper.writeValueAsString(value:updatedData);
        log.log(level:Level.WARNING, msg:"Sending Payload: {0}", param1:jsonPayload);

        // Crear la solicitud con el encabezado Authorization
        Response response = webTarget.path(string:String.valueOf(obj:id))
                                     .request(strings:MediaType.APPLICATION_JSON)
                                     .header(string:"Authorization", "Basic " + encodedCredentials) // Agregar el encabezado Authorization
                                     .put(entity:Entity.entity(entity:updatedData, mediaType:MediaType.APPLICATION_JSON));

        if (response.getStatus() == Response.Status.OK.getStatusCode()) {
            System.out.println(x:"Customer updated successfully.");
        } else {
            // Obtener el mensaje de error detallado del cuerpo de la respuesta
            String errorMessage = response.readEntity(type:String.class);  // Leemos el cuerpo de la respuesta

            // Log de error con el código de estado y el mensaje
            log.log(level:Level.SEVERE, msg:"Error updating customer. Status code: {0}. Message: {1}",
                    new Object[]{response.getStatus(), errorMessage});

            // Lanzar una excepción con el mensaje de error detallado
            if (response.getStatus() == Response.Status.FORBIDDEN.getStatusCode()) {
                throw new RuntimeException("Unauthorized to update customer: " + errorMessage);
            } else if (response.getStatus() == Response.Status.BAD_REQUEST.getStatusCode()) {
                throw new RuntimeException("Bad request when updating customer: " + errorMessage);
            } else {
                throw new RuntimeException(message:errorMessage);
            }
        }
    } catch (JsonProcessingException ex) {
        Logger.getLogger(name:CustomerServiceImpl.class.getName()).log(level:Level.SEVERE, msg:null, thrown:ex);
    }
}
```

Actualitza les dades d'un client especificat a través d'una petició PUT a /customer/{id}.

- Utilitza Basic Authentication (encapçalament HTTP Authorization).

- Fa servir l'objecte ObjectMapper per convertir Customer en un JSON que s'envia com a càrrega útil.
- Si la resposta té errors, retorna el missatge d'error detallat del cos de la resposta.

4. authenticateCustomer(ViewCredentials credentials)

```
@Override
public Customer authenticateCustomer(ViewCredentials credentials) {
    WebTarget targetWithAuth = client.target(string: BASE_URI).path(string:"customer/auth");
    String username = credentials.getUsername();
    String password = credentials.getPassword();

    // Concatenar username:password
    String credencials = username + ":" + password;

    // Codificar las credenciales en Base64
    String encodedCredentials = Base64.getEncoder().encodeToString(src:credencials.getBytes(charset: StandardCharsets.UTF_8));
    // Construir la URL con los parámetros query de username y password
    Response response = targetWithAuth.request(strings: MediaType.APPLICATION_JSON)
                                      .header(string:"Authorization", "Basic " + encodedCredentials)
                                      .get();
    String jsonResponse = response.readEntity(type: String.class);
    log.log(level: Level.INFO, msg:"Response JSON: {0}", param1:jsonResponse);
    // Verificar si la respuesta es exitosa
    if (response.getStatus() == 200) {
        // Si la respuesta es exitosa, deserializar el objeto Credentials
        ObjectMapper objectMapper = new ObjectMapper();
        objectMapper.registerModule(new JavaTimeModule());
        objectMapper.configure(f: DeserializationFeature.READ_ENUMS_USING_TO_STRING, state: true);
        Customer customer = null;
        try {
            customer = objectMapper.readValue(content: jsonResponse, valueType:Customer.class);
        } catch (JsonProcessingException ex) {
            Logger.getLogger(name: CustomerServiceImpl.class.getName()).log(level: Level.SEVERE, msg:null, thrown:ex);
        }
        return customer;
    }

    // Si no fue exitoso, devolver null o lanzar una excepción
    return null;
}
```

Autentica un client utilitzant un encapçalament de Basic Authentication.

- Envia les credencials a /customer/auth per autenticar l'usuari.
- La resposta es deserialitza en un objecte Customer utilitzant ObjectMapper.

5. registerCustomer(Customer customer)

```java
@Override
public boolean registerCustomer(Customer customer) {
    try {
        log.log(level: Level.INFO, msg: "Attempting to register customer: {0}", param1: customer);

        Response response = webTarget
                            .request(strings: MediaType.APPLICATION_JSON)
                            .post(entity: Entity.entity(entity: customer, mediaType: MediaType.APPLICATION_JSON));

        // Comprovar que el resultado ha sido exitoso
        if (response.getStatus() == Response.Status.OK.getStatusCode()) {
            System.out.println(x: "Customer registered successfully.");
            return true;
        } else {
            // Si no ha sido exitoso devolver el error
            String responseBody = response.readEntity(type: String.class);
            throw new RuntimeException(message: responseBody);
        }
    } catch (ProcessingException e) {
        // Manejar excepciones
        System.err.println("Network error while registering customer: " + e.getMessage());
        throw new RuntimeException(message: "Network error occurred during customer registration.", cause: e);
    }
}
```

Registra un nou client enviant una petició POST amb el cos del client a /customer.

- Si la resposta no és satisfactòria, llança una excepció amb el missatge d'error del cos de la resposta.

6. Línies importants del codi:

client = jakarta.ws.rs.client.ClientBuilder.newClient();
webTarget = client.target(BASE_URI).path("customer");

- Defineix la configuració bàsica per fer peticions HTTP al servei REST.

String credencials = username + ":" + password;
String encodedCredentials =
Base64.getEncoder().encodeToString(credencials.getBytes(StandardCharsets.UTF_8));

- Codifica les credencials de l'usuari en Base64 per afegir-les al capçalera Authorization.

ObjectMapper mapper = new ObjectMapper();
String jsonPayload = mapper.writeValueAsString(updatedData);

- Converteix l'objecte Java updatedData a JSON per enviar-lo com a paràmetre en la petició PUT.

objectMapper.readValue(jsonResponse, Customer.class);

- Converteix la resposta JSON del servei en un objecte Java.

```
if (response.getStatus() != Response.Status.OK.getStatusCode()) {
    String errorMessage = response.readEntity(String.class);
    throw new RuntimeException("Error: " + errorMessage);
}
```

- Llegeix el cos de la resposta per obtenir un missatge d'error detallat en cas d'error HTTP.

```
log.log(Level.SEVERE, "Error updating customer. Status code: {0}. Message: {1}",
        new Object[]{response.getStatus(), errorMessage});
```

- Registra errors amb el codi d'estat i el missatge d'error.

### *Vistes*

#### Header.jsp

Aquesta vista genera una pàgina web amb un encapçalament dinàmic que varia en funció de si un client està autenticat o no.

Segueix el següent funcionament:

a. Usuari no autenticat:
   - Veu el missatge "Benvingut, visitant!".
   - Té accés als botons de **SignUp** i **Login** per registrar-se o iniciar sessió.
b. Usuari autenticat:
   - Veu el missatge de benvinguda personalitzat amb el seu nom on si clica pot veure les seves dades personals.
   - Té accés als botons **MyArticles** per veure els seus articles i **Logout** per tancar sessió.
c. Usuari autentificat/no autentificat:
   - Estigui o no autentificat, l'usuari té sempre l'opció de clicar sobre la **imatge** de la **albergínia** per ser redirigit a la pàgina on surten tots els articles.

#### SignUp-Form.jsp

Aquesta vista en JSP proporciona un formulari d'enregistrament d'usuaris amb un disseny diferent i responsiu gràcies a l'ús de Bootstrap. A més, incorpora funcionalitats dinàmiques per gestionar errors i situacions especials, com ara un límit d'intents d'enregistrament.

Segueix el següent funcionament:

1. Usuari accedeix al formulari:
   - Omple els camps necessaris.
   - Si hi ha un **error**, es mostra un missatge dins d'una alerta vermella.
2. Massa intents d'enregistrament:
   - Si l'usuari supera el **límit**, apareix un modal amb un missatge destacat indicant que ha d'esperar.
3. Èxit en l'enregistrament:
   - El formulari s'envia a l'URL del controlador (${mvc.uri('sign-up')}).
   - Aquest retorna la vista de SignUp-Success.

#### SignUp-Success.jsp

Aquesta vista en JSP mostra una pàgina de confirmació per a l'usuari després de completar el procés d'enregistrament amb èxit.

Segueix el següent funcionament:

a. Enregistrament completat amb èxit:
   - Quan l'usuari completa el procés d'enregistrament, és redirigit a aquesta pàgina.
b. Visualització de la informació:
   - L'usuari pot veure les dades que ha proporcionat (First Name, Last Name, Email, Username) dins d'una taula.
c. Acció posterior:
   - L'usuari pot utilitzar el botó "Go Back" per tornar a la pàgina SignUp.

### LogIn-Form.jsp

Presenta una pàgina de Login (inici de sessió) amb un disseny modern gràcies a Bootstrap i FontAwesome.

Segueix el següent funcionament:

1. Carregar la pàgina:
- Quan l'usuari visita la pàgina, el servidor genera i envia el contingut HTML a partir del codi JSP.
    - El servidor JSP utilitza dades predefinides per incloure'ls a la pàgina.
    - El navegador mostra el formulari amb el disseny i estils aplicats, incloent:
        - Camps d'entrada per a usuari i contrasenya.
        - Un botó per iniciar sessió.
        - Alertes dinàmiques si es detecten errors d'intents previs.
2. Interacció de l'usuari:
   - L'usuari introdueix:
     **Usuari** (username)
     **Contrasenya** (password)
   - Opcionalment, marca la casella "Recuérdame".
   - Quan l'usuari fa clic al botó "Iniciar sesión", el formulari es valida inicialment al servidor i s'envia al servidor mitjançant un HTTP POST amb:
       - Dades introduïdes per l'usuari.
       - Un **token CSRF** per prevenir atacs de falsificació de peticions.
3. Validació al servidor:
   - El servidor rep la sol·licitud i executa diverses validacions:
     - Validació de camps: Comprova si l'usuari i la contrasenya **no** són buits.
     - Autenticació:
   - Cerca l'usuari a la **base de dades**.
4. Resultats de la validació
   - Si l'inici de sessió és correcte:
     El servidor redirigeix l'usuari a una pàgina de **perfil**.
   - Si l'inici de sessió falla:
     El servidor torna a carregar la pàgina d'inici de sessió amb un missatge d'error.

5. Superació dels intents màxims
    - Es mostra un modal de **bloqueig** a la pàgina, informant que ha superat el nombre màxim d'intents.
    - L'usuari ha d'esperar abans de poder intentar iniciar sessió de nou.
6. Enllaç de registre
    - Si l'usuari no té un compte:
    - Pot fer clic a l'enllaç "**Regístrate aquí**".
    - Aquest enllaç redirigeix a una altra pàgina JSP que conté el formulari d'inscripció.

### Profile-Page.jsp

Pàgina de perfil d'usuari dissenyada per mostrar les dades de l'usuari actual de manera responsiva i estilitzada.

Segueix el següent funcionament:

1. Carregament inicial:
    - El servidor JSP genera la pàgina HTML amb les dades del **customer** proporcionades pel controlador del servidor.
    - Si el **customer** no és nul (és a dir, si hi ha un usuari autenticat), es copien les dades d'aquest usuari al formulari.

2. Estructura del contingut:

La pàgina es divideix en dues columnes:

a. Columna esquerra (imatge de perfil):
    - Mostra la imatge de perfil de l'usuari.
b. Columna dreta (informació de l'usuari):
    - Mostra les dades de l'usuari en camps de formulari desactivats (readonly), per tal que no es puguin modificar directament.
    - Inclou un botó per editar el perfil.

3. Accions:

Modificar perfil:

- Hi ha un botó "Modificar datos" que enllaça amb una altra pàgina on l'usuari pot editar les seves dades.

### Profile-Modify.jsp

Pàgina de perfil d'usuari que permet modificar la informació de l'usuari, inclosa la pujada d'una nova imatge de perfil.

Segueix el següent funcionament:

1. Carregament inicial:
   - Si l'usuari està autenticat (i el model conté dades de customer), es mostren els camps **preemplenats** amb la seva informació.
   - La pàgina inclou estils personalitzats per fer-la visualment atractiva.
2. Edició del perfil:
   - L'usuari pot:
     - Modificar informació com nom, correu electrònic o contrasenya.
     - Seleccionar una nova imatge de perfil, que es previsualitza **automàticament**.
3. Submissió del formulari
   - En fer clic a "**Save Changes**", les dades s'envien al servidor amb el mètode POST, juntament amb el token CSRF.
   - Automàticament, si la modificació s'ha realitzat de manera correcta, es redirigeix al customer la pàgina de Success.
4. Gestió del servidor
   - Si hi ha un **error**, el servidor retorna la pàgina amb el missatge d'error, que es mostra a l'usuari.

### Profile-Modify-Success.jsp

Informa a l'usuari que les dades del perfil han estat actualitzades correctament.

Segueix el següent funcionament:

1. L'usuari arriba a la pàgina:
   - Després d'actualitzar les dades del **perfil**, el servidor redirigeix l'usuari a aquesta pàgina.
2. Missatge informatiu:
   - Es mostra un missatge que confirma que les dades s'han **actualitzat** correctament.
3. Logout:
   - Quan l'usuari fa clic al botó "**Logout**", una sol·licitud GET s'envia al servidor per tancar la sessió.
4. Redirecció:
   - El servidor gestiona el tancament de sessió i redirigeix a la pàgina d'inici de sessió.

## Articles

### Homework1

Ara procedirem a explicar quins han sigut els canvis principals de la pràctica 1 a la 2 en l'àmbit dels articles. Com que no hi ha gaires canvis serà millor no dividir per gaires apartats l'explicació. Principalment tenim dos raons de canvi. La primera ve a conseqüència d'haver canviat el autor i la segona per necessitar més informació pel frontend.

#### *A causa dels canvis als autors*

El canvi que afecta a la part dels articles és que ara els autors tenen nom i cognom, per la qual cosa quan es faci un filtratge al getArticleList al ArticleService, al where hi haurà condicions diferents.

```
@GET
@Produces(MediaType.APPLICATION_JSON)
public Response findAll(
    @QueryParam("topic")List<Topic> topic,
    @QueryParam("authorName") String authorName) {
    String baseQuery = "SELECT a FROM Article a WHERE 1=1";
    System.out.println("AL BACKEND author: " + authorName + " topics: " + topic);
    // Afegir condicions només si hi ha filtres específics
    // Agregamos condiciones al query en base a los valores
    if (authorName != null) {
        baseQuery += " AND a.authorName = :authorName";
    }
    if(topic != null && !topic.isEmpty()){
        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";

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

    // Assignar els paràmetres si cal
    if (authorName != null ) {
        query.setParameter(name: "authorName", value: authorName);
    }
    if (topic != null && !topic.isEmpty()) {
        for(int i=0; i<topic.size(); i++){
            query.setParameter("topic"+i+"", value: topic.get(index: i));
        }
    }
```

En aquest cas el authorName és el nom i cognom junts. També al canviar les estructures del Article i del ArticleDTO quan s'hagi de fer conversions aquestes seràn diferents.

Per últim s'han posat printlns per poder veure si mana bé alguns paràmetres al frontend sense haver d'emprar el Postman.

### *A causa del frontend*

Principalment el problema que tenim és amb el marhsalling això afecta en que no podem fer que un get consulti a una clase, per exemple per obtenir el nom de l'autor no podrem fer author.getName() ja que author serà un String realment.

En conclusió que hauran canviat els getters, setters per adoptar-se al marshalling correctament.

## Homework2

En aquesta part hi tindrem el front-end i el víncle al backend. Per a fer l'explicació ho més clara possible, començarem a partir de les interaccions de l'usuari fins a les seves repercussions al backend. Abans però, per a no deixar conceptes pendents explicarem conceptes generals sobre els elements del recorregut.

### *ArticleController*

Veiem dues anotacions abans del principi de la classe. @Controller indica que la classe és un controlador i el path ens indica la direcció de la url per a quan es facin les sol·licituds.

```
@Controller
@Path("Article")
public class ArticleController{
    @Inject
    private ArticleServiceImpl articleService;
    @Inject
    private Models models;
    @Inject
    private CustomerServiceImpl customerService;
    @Inject
    private HttpServletRequest request;
    @Inject Logger log;
```

Després tenim el model que estem utilitzant, el qual ens permetrà interaccionar amb les vistes al poder introduir-hi les dades. També tenim els dos serveis, els quals més endavant explicarem per a què serveixen. A continuació tenim HttpServletRequest que ens permetrà treballar amb les sol·licituds http i el logger ens permetrà veure què ha passat. El inject ens permetran no haver de crear més instancies de les classes, sino reutilitzar-les.

### *ArticleServiceImpl*

Ara tenim el servei web dels articles, el qual implementa una interfície amb els mètodes bàsics.

```
public class ArticleServiceImpl implements ArticleService {
    private final WebTarget webTarget;
    private final jakarta.ws.rs.client.Client client;
    private static final String BASE_URI = "http://localhost:8080/Homework1/rest/api/v1";

    public ArticleServiceImpl() {
        client = jakarta.ws.rs.client.ClientBuilder.newClient();
        webTarget = client.target(string: BASE_URI).path(string: "article");
    }
```

Ara tenim tres atributs, els qual estan enfocats a vincular el frontend amb el backend. El primer paràmetre més interessant és la base_uri, la qual ens indica la url del backend. Després tenim el webtarget que és la classe final que farà referència al backend amb el client.

Ara explicarem el recorregut per les diferents classes. Començarem per la llista d'articles, ja que aquesta serà la primera situació que experimentarà l'usuari al principi gràcies al index.jsp.

### *FindAll*

Al principi de la pàgina web tenim con ja hem dit el index.jsp el qual redireccionarà l'usuari a /Web/Article. La funció dels controladors és tractar les sol·licituds que es fan a la pàgina web.

```
@GET
@UriRef("see-articles")
public String findAll(@QueryParam("topic")List<Topic> topic, @QueryParam("authorName") String authorName){
    System.out.println("topics Controller: " + topic);
    System.out.println("author Controller: " + authorName);
    List<ArticleDTO> articleList = articleService.findAll(topic, authorName);
    //for(ArticleDTO article : articleList) System.out.println("Resposta: " + article.toString());
    models.put("articleList", articleList);
    Topic topicList[] = Topic.values();
    models.put("topicList", topicList);
    return "ArticleList.jsp";
}
```

Llavors el que tenim és l'anotació @GET que ens diu quin tipus de sol·licitud és. Després tenim la @UriRef que és un path absolut per accedir al controlador.

Cal dir que els paràmetres d'entrada segueixen els mateixos conceptes del backend.

Ara del que es tracta és de cridar al articleService per anar al backend a resoldre la sol·licitud i posar-ho al model per a que la vista pugui accedir les dades. Per últim retorna la vista per a que la vegi l'usuari. Anem llavors a explicar millor el service d'aquest cas.

```java
@Override
public List<ArticleDTO> findAll(List<Topic> topic, String authorName) {
    /*Response response = webTarget
            .request(MediaType.APPLICATION_JSON)
            .get();*/
    WebTarget creatingTarget = webTarget;
    if (topic!=null && !topic.isEmpty()){
        for (Topic onetopic : topic){
            System.out.println("topics service: " + onetopic);
            if (onetopic!=null) creatingTarget = creatingTarget.queryParam(string: "topic", os: onetopic);
        }
    }
    if (authorName!=null && !authorName.trim().isEmpty()){
        System.out.println("author service: " + authorName);
        creatingTarget = creatingTarget.queryParam(string: "authorName", os: authorName);
    }

    System.out.println("creatingTarget: " + creatingTarget);
    Response response = creatingTarget.request(strings:MediaType.APPLICATION_JSON).get();
    //System.out.println("Resposta: " + response);
    ArticleDTO articleList[] = response.readEntity(ArticleDTO[].class);
    if (response.getStatus() == Response.Status.OK.getStatusCode()) {
        for(ArticleDTO article : articleList) System.out.println("Resposta: " + article.toString());
        return Arrays.asList(a: articleList);
    }
    else {
        return null;
    }
}
```

Sense entrar molt en detall, l'objectiu del service aquí es poder manar la solicitud al backend, per a fer-ho haurem d'anar construint el que seria la URL depenent dels paràmetres de filtratge que s'hagin fet. Finalment es fa la sol·licitud i després es passen els articles al controller.

### *GetArticleById*

A en aquest article hi accedirem a través de la llista d'articles, hi haurà un botó que ens redireccioni al controller.

```java
@GET
@UriRef("see-article")
@Path("{id}")
public String getArticleById(@PathParam("id") Long id) {
    try{
        System.out.println("id controller: " + id);
        Customer customer = (Customer) request.getSession().getAttribute(string: "customer");
        ViewCredentials credentials = new ViewCredentials(null, null);
        log.log(level: Level.SEVERE, msg: "customer = {0}", params: customer);
        if (customer != null){
            credentials = new ViewCredentials(customer.getUsername(), customer.getPassword());
        }
        Article article = articleService.getArticleById(id, credentials);
        if (article == null) {
            return "Error404.jsp";
        }
        models.put("article", article);
        return "Article.jsp";
    } catch (RuntimeException e){
        String currentUrl = request.getRequestURL().toString();
        String relativeUrl = currentUrl.replace(target: "http://localhost:8080/Homework2/Web/", replacement: "");
        request.getSession().setAttribute(string: "redirectUrl", o: relativeUrl);
        return "redirect:/Login";
    }
}
```

Les anotacions ja les hem comentat abans i els paràmetres també. Les principals coses a comentar és que primer hem de mirar si les l'usuari està registrat ja que pot ser que l'article sigui privat. Això ho mirem amb la sessió i les credencials. Després farem la sol·licitud a la pàgina i en cas d'error manem a l'usuari a la vista d'error. Si tot ha anat bé es posa l'article al model per a manar a l'usuari a la vista de l'article i que es puguin veure les dades.

```
@Override
public Article getArticleById(Long id, ViewCredentials credentials) {
    String username = credentials.getUsername();
    String password = credentials.getPassword();

    // Concatenar username:password
    String credenciales = username + ":" + password;

    // Codificar las credenciales en Base64
    String encodedCredentials = Base64.getEncoder().encodeToString(src: credenciales.getBytes(charset:StandardCharsets.UTF_8));
    System.out.println("id service: " + id);
    WebTarget creatingTarget = webTarget.path(string: id.toString());
    System.out.println("creatingTarget: " + creatingTarget);

    Response response = creatingTarget.request(strings:MediaType.APPLICATION_JSON)
                                      .header(string: "Authorization", "Basic " + encodedCredentials)
                                      .get();

    if (response.getStatus() == Response.Status.OK.getStatusCode()) {
        Article article = response.readEntity(type: Article.class);
        System.out.println("article author photo: " + article.getAuthorPhoto());
        return article;
    } else if (response.getStatus() == Response.Status.UNAUTHORIZED.getStatusCode()) {
        throw new RuntimeException(message:"Unauthorized: Private article. Please log in to read it.");
    } else if (response.getStatus() == Response.Status.NOT_FOUND.getStatusCode()) {
        throw new RuntimeException("Not Found: Article with ID " + id + " does not exist");
    } else {
        throw new RuntimeException("Unexpected error: HTTP " + response.getStatus());
    }
}
```

El que farem ara realment és el que fèiem al postman de posar les credencials. Allí es posaven i ja, aquí les hem de codificar nosaltres. Ja després retornem l'article que passarà al controller que passarà al model que passarà a la vista.

### *DeleteArticle*

A en aquest controller hi accedirem a través de la llista d'articles personals, ja que a en aquesta només hi podrem accedir si som els autors. D'aquesta manera en assegurem que l'autor de l'article és l'únic capac de borrar-lo.

```java
@POST
//@UriRef("delete")
@Path("Delete/{id}")
public String deleteArticle(@PathParam("id") Long id) {
    log.log(level: Level.SEVERE, msg: "id = {0}", param1: id);
    if (id == null || id < 1) {
        // Redirigir a la pàgina d'error si la id és invàlida
        return "Error404.jsp";
    }
    Customer customer = (Customer) request.getSession().getAttribute(string: "customer");
    log.log(level: Level.SEVERE, msg: "customer = {0}", params: customer);
    ViewCredentials credentials = new ViewCredentials(customer.getUsername(), customer.getPassword());
    Boolean deleted = articleService.deleteArticle(id, credentials);
    if (Boolean.FALSE.equals(obj: deleted)) {
        log.log(level: Level.SEVERE, msg: "deleted = {0}", param1: deleted);
        // Si no es pot eliminar, mostrar la pàgina d'error
        return "Error404.jsp";
    }
    customer = customerService.authenticateCustomer(credentials);
    request.getSession().setAttribute(string: "customer", o: customer);
    // Si tot va bé, redirigir a la pàgina de confirmació d'eliminació
    return "CustomerArticles.jsp";
}
```

Tenim les anotacions anteriorment comentades i després tenim un comprovació de que el id de l'article sigui vàlid.

Després hi ha una comprovació extra sobre el id de l'usuari, d'aquesta manera tenim una doble capa de seguretat davant dels altres a altres autors. Per últim fem la petició al servei i després informarem a l'usuari si s'ha eliminat amb èxit. Si s'ha fet amb èxit, haurem de renovar el customer de la sessió ja que s'hauràn consumit.

```java
@Override
public boolean deleteArticle(Long id, ViewCredentials credentials) {
    try {
        // Verificar si el ID es válido
        if (id == null || id < 1) {
            return false;
        }

        // Construir el encabezado Authorization para Basic Authentication
        String username = credentials.getUsername();
        String password = credentials.getPassword();

        // Concatenar username:password
        String credenciales = username + ":" + password;

        // Codificar las credenciales en Base64
        String encodedCredentials = Base64.getEncoder().encodeToString(src: credenciales.getBytes(charset:StandardCharsets.UTF_8));

        // Realizar la solicitud DELETE con el encabezado Authorization
        Response response = webTarget.path(String.valueOf(obj: id))
                                     .request(MediaType.APPLICATION_JSON)
                                     .header("Authorization", "Basic " + encodedCredentials)
                                     .delete();

        // Verificar si el estado de la respuesta es 200 (OK)
        return response.getStatus() == 200;
    } catch (Exception ex) {
        Logger.getLogger(name: CustomerServiceImpl.class.getName()).log(Level.SEVERE, ex.getMessage(), ex);
        return false;
    }
}
```

Ara tornarem a fer una comprovació de les credencials i farem com al mètode de buscar un article pel seu id. Codificarem les credencials com fa postman i manarem la solicitud al backend. Per últim retornarem què ha passat al controller.

### *ArticleList.jsp*

Del que s'encarrega aquesta vista es d'ensenyar tots els articles que es troben a la pàgina web. Aquesta vista segueix el comportament bàsic del mètode d'obtenir la llista dels articles del backend, per la qual cosa haurà de passar els paràmetres necessaris, és per això que al principi de tot tindrem un formulari amb tres entrades: dos tòpics i el nom de l'autor.

Al jsp això vidria a ser la part del form. En la qual tenim indicat la uri-ref del controlador. Tenim les dos opcions de formulari que per a poder indicar a l'usuari quins son els possibles valor fem ús de la llista al model proporcionada pel controlador. Després tenim la part d'omplir on hi posarem el nom i cognom de l'autor. Hem decidit no admetre ús de caràcters especials per la qual cosa si n'emprem no sortirà res.

A partir d'aquí emprarem jsp ja que amb html i css no podrem reflectir realment la informació, m'explico.

Hem decidit que els articles es vegin en forma de cards, per la qual cosa si féssim el format d'una card només sortiria una o el numero d'aquestes que haguéssim fet. Per a que n'hi hagi tantes com articles hem d'emprar jsp amb funcionalitats com el foreach que permeten fer bucles, d'aquesta manera podem reflectir tota la informació del model a la vista. Aquestes cards contenen tota la informació proporcionada pel controlador a traves del model y en format d'una llista d'articlesDTO. Per accedir a l'article haurem de clicar sobre la targeta i la pàgina cridarà al controlador del article individual. Per a fer-ho primer hem d'aconseguir el id de l'article que haguem aconseguit, per la qual cosa haurem de consultar l'id de l'articleDTO.

A dins del controlador ja es farà el control de la privacitat.

### *Article.jsp*

En aquesta vista simplement reflectirem tota la informació de l'article sencer exceptuant el resum. Haurem de procurar que segueixi una estructura comprensible. Les dades com ja s'ha comentat anteriorment s'extreuen del model que en aquest vas vindran proporcionades pel controlador d'obtenir un article. L'últim a comentar seria que les vistes son responsives, per la qual cosa al fer més petita la pantalla pràcticament tots els elements disminuiran en mida per a que segueixi amb una estructura comprensible malgrat la inferior finestra.

## Dependències

Per tal de poder treballar amb les dades que obteniem de formularis o de fitxers JSON provinents de les API REST de manera més senzilla i sense problemes s'han afegit depències al fitxer pom.xml.

Les dependències afegides al projecte tenen diverses funcions importants que ajuden a gestionar i processar dades de manera més eficient. A continuació s'explica el que fa cadascuna:

**commons-beanutils**:

Aquesta dependència prové de la biblioteca *Apache Commons BeanUtils*. Proporciona utilitats per treballar amb JavaBeans, com ara copiar propietats entre diferents objectes o convertir tipus de dades de manera fàcil. Aquesta biblioteca pot ser útil quan es treballa amb objectes Java i es necessiten funcions automàtiques per manipular les propietats d'aquests objectes, com convertir entre tipus de dades o copiar atributs d'un bean a un altre.

**jackson-databind**:

Aquesta dependència és part de la llibreria Jackson, que s'utilitza per al processament de dades en format JSON. *jackson-databind* proporciona les eines necessàries per serialitzar i deserialitzar dades entre objectes Java i JSON. En altres paraules, permet convertir un objecte Java en una cadena JSON i viceversa. Aquesta biblioteca és molt utilitzada en aplicacions web per gestionar la comunicació entre el client i el servidor a través de JSON.

**jackson-datatype-jsr310**:

Aquesta dependència és una extensió de Jackson que proporciona suport per als tipus de dades de *Java 8 Date/Time API* (les classes com LocalDate, LocalTime, LocalDateTime, etc.). Sense aquesta dependència, Jackson no seria capaç de processar correctament aquestes noves classes de data i hora introduïdes a Java 8. Aquesta biblioteca facilita la serialització i deserialització d'objectes que contenen tipus de dades de data i hora moderns.

# Jocs de proves realitzats

## 1. Proves d'autentificació

**URL**: http://localhost:8080/Homework2/Web/Login

### [Prova 1.1]. Login incorrecte

**Descripció**: Prova d'inici de sessió sense dades.
**Paràmetres/Dades**:

- _csrf = csrf_token
- username =
- password =

**Sortida esperada**:

- **Camps buits**: Missatge d'error demanant omplir els camps requerits.

**Correcte**: Sí

### [Prova 1.2]. Login correcte

**Descripció**: Prova d'inici de sessió amb dades correcte
**Paràmetres/Dades**:

- _csrf = csrf_token
- username = sob
- password = sob

**Sortida esperada**:

- **Login correcte**: Redirecció a la pàgina corresponent.

**Correcte**: Sí

### [Prova 1.3]. Login incorrecte

**Descripció**: Prova d'inici de sessió amb contrasenya incorrecta.
**Paràmetres/Dades**:

- _csrf = csrf_token
- username = sob
- password = 1234

**Sortida esperada**:

- **Camps d'error**: Missatge d'error indicant que l'usuari o contrasenya són incorrectes.

**Correcte**: Sí

#### [Prova 1.3]. Tancar sessió

**Descripció**: Validar que el tancament de sessió es realitza correctament. **Paràmetres/Dades**:

- _csrf = csrf_token
- username = sob
- password = sob

**Sortida esperada**:

- **Logout correcte**: Redirecció a una pàgina de login després de fer logout.

**Correcte**: Sí

## 2. Proves d'accés a articles de l'usuari

**URL**: http://localhost:8080/Homework2/Web/Profile/Articles

#### [Prova 2.1]. Accés a articles privats sense autenticació

**Descripció:** Verificar l'accés a contingut privat sense estar autenticat.

**Paràmetres/Dades:**

- _csrf = csrf_token

**Sortida esperada:**

- Redirecció a la pàgina de login.

- Restricció d'accés directe amb el formulari de login visible.

**Correcte:** Sí

#### [Prova 2.2]. Accés a articles privats després del login

**Descripció:** Verificar que l'usuari pot accedir als articles privats després d'autenticar-se.

**Paràmetres/Dades:**

- _csrf = csrf_token
- username = sob
- password = sob

**Sortida esperada:**

- Contingut de l'article visible.
- El comptador de visualitzacions incrementa en 1.

**Correcte:** Sí

## 3. Proves de visualització de dades de l'usuari

**URL**: http://localhost:8080/Homework2/Web/Profile

#### [Prova 3.1]. Visualització del nom d'usuari

**Descripció:** Verificar que el nom de l'usuari es mostra en el missatge de benvinguda i que amb un clic s'accedeix a les dades privades.

**Paràmetres/Dades:**

- _csrf = csrf_token
- username = sob
- password = sob

**Sortida esperada:**

- Missatge de benvinguda amb el nom de l'usuari.
- Dades del perfil visibles al fer clic sobre el nom.

**Correcte:** Sí

## 4. Proves de modificació de dades de l'usuari

**URL**: http://localhost:8080/Homework2/Web/ModifyProfile

### [Prova 4.1]. Modificar el nom d'usuari per un ja existent

**Descripció:** Intentar modificar el nom d'usuari per un que ja existeix en la base de dades.

Paràmetres/Dades:

- _csrf = csrf_token
- username = miriamduran

**Sortida esperada:** Missatge d'alerta indicant que el nom d'usuari ja està en ús.

**Correcte:** Sí

### [Prova 4.2]. Modificar el nom d'usuari correctament

**Descripció:** Modificar el nom d'usuari per un de nou i vàlid.

Paràmetres/Dades:

- _csrf = csrf_token
- username = soba

**Sortida esperada:** Sol·licitud acceptada i redirecció a la pàgina de success.

**Correcte:** Sí

### [Prova 4.3]. Format incorrecte del correu electrònic

**Descripció:** Provar amb un format d'email incorrecte.

Paràmetres/Dades:

- _csrf = csrf_token
- email = formatincorrecte

**Sortida esperada:** Missatge d'alerta indicant format incorrecte.

**Correcte:** Sí

## 5. Proves de modificació de dades de l'usuari

**URL**: http://localhost:8080/Homework2/Web/SignUp

### [Prova 5.1]. Registre de nou usuari

**Descripció:** Verificar la funcionalitat de registre d'usuari amb dades correctes.

**Paràmetres/Dades:**

- First Name = Maria
- Last Name = Debe
- Email = maria@gmail.com
- Username = maria
- Password = 12345678

**Sortida esperada:**

- Registre correcte: Usuari creat correctament.

**Correcte:** Sí

### [Prova 5.2]. Registre de nou usuari amb camp buits

**Descripció:** Verificar la funcionalitat de registre d'usuari amb camp buits.

**Paràmetres/Dades:**

- First Name = Maria
- Last Name = Debe
- Email = maria@gmail.com
- Username =
- Password =

**Sortida esperada:**

- Camp buits: Missatge d'error indicant que s'han d'omplir els camps buits.

**Correcte:** Sí

#### [Prova 5.3]. Registre de nou usuari amb dades no vàlides

**Descripció:** Verificar la funcionalitat de registre d'usuari indicant un correu electrònic ja en ús.

**Paràmetres/Dades:**

- First Name = Mario
- Last Name = Debe
- Email = maria@gmail.com
- Username = mari0
- Password = 12345678

**Sortida esperada:**

- Correu existent: Missatge d'error indicant que el correu ja està en ús.

**Correcte:** Sí

## 6. Proves de seguretat i usabilitat

#### [Prova 6.1]. Usabilitat del formulari de Login

**Descripció:** Verificar que el formulari de Login és usable i es mostra correctament en diferents dispositius.

**Paràmetres/Dades:** Formulari de Login accedit des de diferents dispositius.

**Sortida esperada:**

- **Pantalles petites (mòbils):** El formulari de Login es mostra correctament i permet l'ús fàcil en dispositius mòbils.
- **Pantalles grans (escriptoris):** El formulari de Login s'ajusta correctament a pantalles grans i és fàcilment accessible.

**Correcte:** Sí

### [Prova 6.2]. Seguretat del sistema d'autenticació

**Descripció:** Verificar que el sistema d'autenticació és segur davant atacs comuns com SQL Injection.

**Paràmetres/Dades:**

- SQL Injection: ' OR 1=1 --

**Sortida esperada:**

- **SQL Injection:** El sistema bloqueja correctament la injecció de codi SQL i no retorna dades sensibles.

**Correcte:** Sí

## 7. Proves de llistes d'articles

### [Prova 7.1]. No filtrar cap article

**Descripció:** Entrar a la vista a l'inici de la pàgina (quan encara no haurem sigut capaços d'emprar filtres).

**Paràmetres/Dades:** cap.

**Sortida esperada:** Tots els articles (en total 6).

### [Prova 7.2]. Filtrar amb un tòpic

**Descripció:** Omplir el formulari amb només un tòpic i clicar al botó de filtratge. La prova es farà dos cops, una per la primera casella i una per la segona.

**Paràmetres/Dades:** topic=Software

**Sortida esperada:** Tots els articles que tenen com a algun dels seus tòpics Software (en total )

**[Prova 7.3]. Filtrar amb dos tòpics**

**Descripció:** Omplir les dues caselles dels tòpics i clicar al botó de filtratge.

**Paràmetres/Dades:** Software, AI

**Sortida esperada:** Articles amb els dos tòpics. Sense insercions noves només 1.

**[Prova 7.4]. Filtrar amb dos tòpics i l'autor existent**

**Descripció:** Omplir les dues caselles dels tòpics i la de l'autor i clicar al botó de filtratge.

**Paràmetres/Dades:** Software, AI, sob sob.

**Sortida esperada:** Articles amb els dos tòpics i creats per l'usuari sob sob. Sense insercions noves només 1.

**[Prova 7.5]. Filtrar amb dos tòpics i l'autor sense articles dels dos topics.**

**Descripció:** Omplir les dues caselles dels tòpics i la de l'autor i clicar al botó de filtratge però sabem que l'autor no té articles amb aquells tòpics.

**Paràmetres/Dades:** Software, AI, Marc Guifre.

**Sortida esperada:** "No s'han trobat articles"

**[Prova 7.6]. Filtrar amb dos tòpics i l'autor sense articles d'un dels topics.**

**Descripció:** Omplir les dues caselles dels tòpics i la de l'autor i clicar al botó de filtratge però sabem que l'autor no té articles amb un dels tòpics.

**Paràmetres/Dades:** Software, AI, Satxa Fortuny.

**Sortida esperada:** "No s'han trobat articles"

**[Prova 7.7]. Filtrar amb un tòpic i l'autor**

**Descripció:** Omplir una casella dels tòpics i la de l'autor i clicar al botó de filtratge.

**Paràmetres/Dades:** Software, Satxa Fortuny.

**Sortida esperada:** Tots els articles de Satxa Fortuny que vagin sobre software. Sense noves insercions només hi ha 2.

**[Prova 7.8]. Filtrar per l'autor no existent**

**Descripció:** Omplir la casella de l'autor amb un autor inexistent a la pàgina.

**Paràmetres/Dades:** Jovani Vazquez

**Sortida esperada:** "No s'han trobat articles"

**[Prova 7.9]. Filtrar per un autor sense articles**

**Descripció:** Omplir la casella de l'autor amb un autor que sabem que no té articles.

**Paràmetres/Dades:** Miriam Duran

**Sortida esperada:** "No s'han trobat articles"

**[Prova 7.10]. Filtrar sense res**

**Descripció:** Clicar al botó de filtratge sense haver omplert cap camp.

**Paràmetres/Dades:** cap.

**Sortida esperada:** Tots els articles. Sense noves insercions són 6.

**[Prova 7.11]. Filtrar per un autor existent però li posem un accent.**

**Descripció:** Omplirem la casella de l'autor però li afegirem un accent extra.

**Paràmetres/Dades:** sob sób

**Sortida esperada:** "No s'han trobat articles"

# Conclusions

Aquest projecte ens ha servit per posar en pràctica tot el que hem vist sobre desenvolupament web: des de com estructurar un backend amb API REST fins a crear un frontend que sigui usable i, el més important, segur. He pogut treballar amb el patró MVC, que ajuda molt a organitzar tot el codi. També ha estat clau entendre com connectar el backend amb el frontend i que tot funcioni de manera fluida.

Pel que fa a la seguretat, hem vist com de fàcil és deixar forats si no ho penses tot bé (gràcies a les proves amb SQL Injection i la validació de dades). Les proves han estat un punt clau per assegurar-nos que no només funcionés, sinó que també resistís errors típics. Tot plegat, ha estat una bona experiència per veure com es pot construir una aplicació completa des de zero.

### Què he après de la pràctica?

He après molt sobre com estructurar un projecte web de manera ordenada i eficient. Integrar APIs REST al backend i connectar-les amb el frontend ha estat un dels grans aprenentatges. També hem millorat en seguretat (CSRF, validacions, etc.), i hem vist com d'important és fer bones proves per evitar errors que poden acabar sent un problema. També hem emprat eines com Postman o NetBeans per validar si realment la informació es passava bé del backend al frontend.

### Quines dificultats he tingut?

Primerament el marshalling ja que és una de les parts més complicades de comprovar i a l'hora de solucionar errors era ho més curós. Vam haver de revisar diversos cops els getters, setters i com es gestionaven les dades JSON.
Adaptar el disseny, ja que fer que tot es veiés bé en mòbils i ordinadors ha estat més complicat del que semblava al principi. Però, amb paciència, s'anava fent i era prou entretingut.

# 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. Donar-li a deploy a **Homework1**.
4. Donar-li a **run** a Homework1 i clicar al botó de "**Install**" a la web.
5. Tornar a NetBeans i fer un **Build with Dependencies** de Homework2.
6. Donar-li run a **Homework2**.
7. Ja pots provar la nostra pàgina web!