

DOSSIER PROFESSIONNEL (DP)

Nom de naissance

✉ Romatet

Nom d'usage

✉ Entrez votre nom d'usage ici.

Prénom

✉ Mathieu

Adresse

✉ 53 Grande rue Saint Michel
31400 Toulouse

Titre professionnel visé

Concepteur Développeur d'Applications

MODALITE D'ACCES :

- ☒ Parcours de formation
- ☐ Validation des Acquis de l'Expérience (VAE)

Présentation du dossier

Le dossier professionnel (DP) constitue un élément du système de validation du titre professionnel.
Ce titre est délivré par le Ministère chargé de l'emploi.

Le DP appartient au candidat. Il le conserve, l'actualise durant son parcours et le présente **obligatoirement à chaque session d'examen.**

Pour rédiger le DP, le candidat peut être aidé par un formateur ou par un accompagnateur VAE.

Il est consulté par le jury au moment de la session d'examen.

Pour prendre sa décision, le jury dispose :

1. des résultats de la mise en situation professionnelle complétés, éventuellement, du questionnaire professionnel ou de l'entretien professionnel ou de l'entretien technique ou du questionnement à partir de productions.
2. du **Dossier Professionnel** (DP) dans lequel le candidat a consigné les preuves de sa pratique professionnelle.
3. des résultats des évaluations passées en cours de formation lorsque le candidat évalué est issu d'un parcours de formation
4. de l'entretien final (dans le cadre de la session titre).

[Arrêté du 22 décembre 2015, relatif aux conditions de délivrance des titres professionnels du ministère chargé de l'Emploi]

Ce dossier comporte :

- ▶ pour chaque activité-type du titre visé, un à trois exemples de pratique professionnelle ;
- ▶ un tableau à renseigner si le candidat souhaite porter à la connaissance du jury la détention d'un titre, d'un diplôme, d'un certificat de qualification professionnelle (CQP) ou des attestations de formation ;
- ▶ une déclaration sur l'honneur à compléter et à signer ;
- ▶ des documents illustrant la pratique professionnelle du candidat (facultatif)
- ▶ des annexes, si nécessaire.

Pour compléter ce dossier, le candidat dispose d'un site web en accès libre sur le site.



<http://travail-emploi.gouv.fr/titres-professionnels>

Sommaire

Exemples de pratique professionnelle

Concevoir et développer des composants d'interface utilisateur en intégrant les recommandations de sécurité	p.	5
📁 TODO_WebAPP.....	p.	5
Concevoir et développer la persistance des données en intégrant les recommandations de sécurité	p.	
📁 Projet InfoPlus.....	p.	16
Concevoir et développer une application multicouche répartie en intégrant les recommandations de sécurité	p.	
📁 E-commerce Backend.....	p.	21
📁 WookieApp.....	p.	25
Déclaration sur l'honneur	p.	31
Annexes <i>(Si le RC le prévoit)</i>	p.	32

EXEMPLES DE PRATIQUE PROFESSIONNELLE

Activité-type

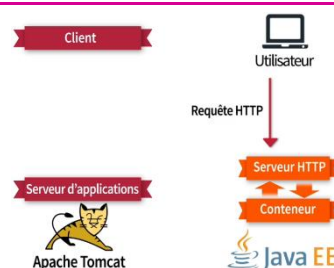
1

Concevoir et développer des composants d'interface utilisateur en intégrant les recommandations de sécurité

Exemple n°1  TODO_WebAPP

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

Dans le cadre de la formation, j'ai développé en équipe une application web **Java EE** pouvant fournir aux utilisateurs un service de gestion des listes de tâches à effectuer (To do). L'utilisateur doit pouvoir se connecter et se déconnecter à son compte via un site hébergé sur un serveur. Il pourra créer une liste de **Todo**, les consulter, les modifier et les supprimer. Il peut aussi faire une recherche par mots clés pour afficher les Todos. Les données doivent être persistantes.



Ce projet devait être réalisé en groupe et nous avons décidé de fonctionner en méthode **Agile Scrum**, la formatrice tenant le rôle de Product Owner. Cette approche nous a permis de segmenter le projet en plusieurs étapes incrémentielles, classés par ordre d'importance, incluant un feedback à intervalles réguliers et une mise à jour des tickets **JIRA**. Cette méthode a permis de favoriser la collaboration, de procéder à des ajustements à intervalles réguliers afin de s'assurer de répondre aux besoins. Nous avons aussi créé un dépôt distant sur **GitLab** pour le contrôle de versions.

Première étape : Conception

Nous avons, dans un premier temps, utilisé le langage de représentation **UML** destiné à la modélisation objet.

Modèles fonctionnels

Diagramme de cas d'utilisation (ou Use Cases) qui décrit les besoins d'utilisation du système, en privilégiant le point de vue de l'utilisateur (ex : Annexe 1).

Diagramme d'activité qui montre l'enchaînement des activités qui concourent au processus (ex : Annexe 2).

Modèles statiques

Diagramme de classes qui décrit la structure des classes et les relations qui existent entre elles (ex : Annexe 3).

Modèles dynamiques

Diagramme de séquence qui décrit la séquence par scénario de messages échangés entre des objets (ex : Annexe 4).

Modèles architecturaux

Diagramme de déploiement qui montre la configuration physique des matériels du système (ex : Annexe 5).

Une fois la modélisation réalisée, nous avons élaboré des **maquettes IHM** (Annexe 6) qui ont été validées par le PO et créé la base de données (annexe 7).

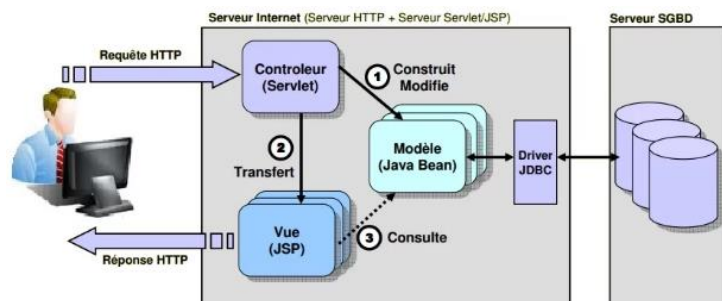
Deuxième Etape : Développement

1) Définition de l'architecture du projet

Pour ce projet, nous avons décidé d'utiliser l'outil **Maven** pour automatiser certaines tâches (compilation, tests unitaires) et gérer des dépendances vis-à-vis des bibliothèques nécessaires au projet via le fichier *pom.xml*. Nous avons construit l'application en plusieurs couches et les design patterns **Modèle / Vue / Contrôleur** (MVC) et **Data Access Object** (DAO) seront appliqués.

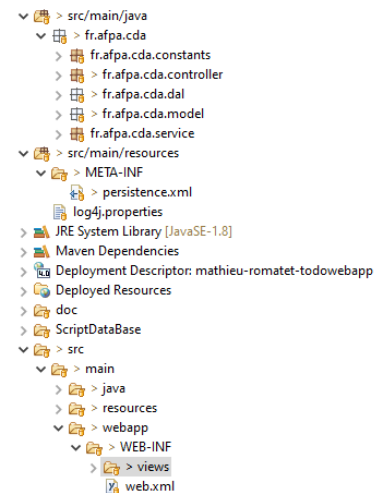
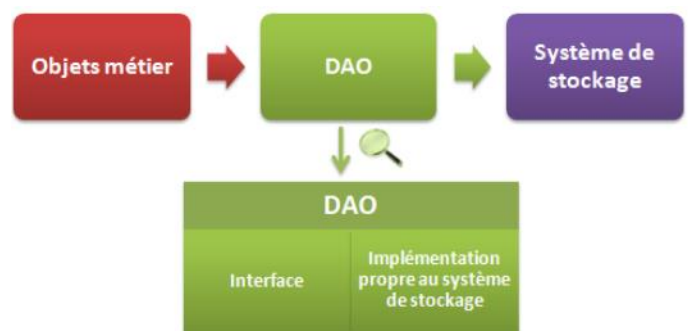
Le modèle **MVC** offre une séparation claire des responsabilités au sein d'une application, en conformité avec les principes de conception : responsabilité unique, couplage faible et cohésion forte.

- **La partie Modèle** (Java Bean) correspond à la logique métier. Elle représente le comportement de l'application : traitements des données, interactions avec la base de données, etc. Elle assure la gestion de ces données et garantit leur intégrité.
- **La partie Vue** (JSP) correspond à l'interface graphique avec laquelle l'utilisateur interagit. La vue a pour rôle d'afficher et de présenter les données ou les résultats renvoyés par le modèle. La page est écrite en HTML et JAVA et n'effectue aucun traitement.
- **La partie Contrôleur** (Servlet) gère les requêtes HTTP, demande au Modèle de trouver les données correspondantes dans la base de données, analyse les données fournies par le Modèle et décide ce qui doit être affiché par la partie Vue.



Le modèle **DAO** permet de faire la distinction entre les données auxquelles on souhaite accéder, et la manière dont elles sont stockées et se compose de 3 parties :

- **Une interface DAO** qui définit les opérations standard à effectuer sur un ou plusieurs objets du modèle.
- **Une Classe DAO** concrète qui implémente l'interface ci-dessus. Cette classe est responsable de l'obtention de données à partir d'une source de données qui peut être une base de données / xml ou tout autre mécanisme de stockage.
- **Un Modèle** (objet métier) qui est un simple POJO ou Entity (avec JPA) contenant des méthodes get/set pour stocker les données récupérées à l'aide de la classe DAO.



2) Développement des composants d'accès aux données dans la couche DAL

Java Persistence API (JPA) est une norme Java qui nous a permis de définir des objets Java (**entity**) qui sont simplement des instances de classes qui seront **persistantes**. Il s'agit d'une approche de l'Object Relationship Mapping (**ORM**) qui nous permet de récupérer, stocker, mettre à jour et supprimer les données dans la base de données relationnelle. Les annotations JPA décrivent comment une classe Java donnée et ses variables correspondent à une table donnée et à ses colonnes dans une base de données.

```
* The persistent class for the utilisateur database table.
*/
@Entity
@Table(name = "utilisateur")
public class UserEntity implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Integer id;

    @Column(name = "firstname")
    private String firstname;

    @Column(name = "lastname")
    private String lastname;

    @Column(name = "login")
    private String login;

    @Column(name = "password")
    private String password;
}
```

À l'entité *UserEntity* ci-dessus, on pourra faire correspondre la table PostgreSQL *utilisateur* :

```
*Classe qui gère la connection et la déconnection avec la BDD
* @CDA
*/
public class DAOUtil {

    private static EntityManagerFactory emf = null;

    private DAOUtil() {
    }

    public static synchronized EntityManagerFactory getEmf() {
        if(emf == null){
            emf = Persistence.createEntityManagerFactory("todo-web-app");
        }
        return emf;
    }

    public static synchronized void close(){
        if(emf != null){
            emf.close();
            emf = null;
        }
    }
}
```

@Entity Définit qu'une classe est une entité.

@Id Définit l'attribut qui sert de clé primaire dans la table.

@Table Permet de définir les informations sur la table représentant cette entité en base de données.

@GeneratedValue Indique la stratégie à appliquer pour la génération de la clé lors de l'insertion d'une entité en base.

@Column Permet de déclarer des informations relatives à la colonne sur laquelle un attribut doit être mappé.

```
DROP TABLE IF EXISTS utilisateur CASCADE;
DROP TABLE IF EXISTS todo CASCADE;

CREATE TABLE utilisateur (
    id SERIAL PRIMARY KEY,
    firstName varchar(20) DEFAULT NULL,
    lastName varchar(20) DEFAULT NULL,
    login varchar(255) DEFAULT NULL,
    password varchar(20) DEFAULT NULL,
    CONSTRAINT unique_username UNIQUE (login)
);

CREATE TABLE todo (
    id SERIAL PRIMARY KEY,
    description varchar(255) DEFAULT NULL,
    done BOOLEAN NOT NULL,
    targetDate DATE DEFAULT NULL,
    title varchar(255) DEFAULT NULL,
    idutilisateur int
);
```

JPA est une spécification, pour pouvoir y accéder, nous utilisons une implémentation compatible avec JPA qui est **Hibernate**. Le point d'entrée du code permettant de manipuler des entités JPA est un objet de type **EntityManagerFactory** qui est utilisé dans la classe *DAOUtil* qui gère les connexions avec la BDD :

Le nom « *todo-web-app* » correspond à une unité de persistance définie dans le fichier **persistence.xml** (annexe 8) qui contient certaines informations dont l'implémentation JPA a besoin (drivers, nom de base de données, ...).

Cet objet de type **EntityManagerFactory** modélise notre unité de persistance. C'est à partir de lui que l'on peut construire des objets de type **EntityManager**, qui nous permettent d'interagir avec la base.

```
* Classe générique qui implémente les méthodes CRUD du stockage dans la BDD avec JPA
* @author CDA
* @param <T>
* @param <ID>
*/
public abstract class GenericDAOJPAImpl<T, ID> implements IDAO<T, ID> {

    private Class<T> clazz=getClazz();

    public final void setClazz(final Class<T> clazzToSet) {this.clazz = clazzToSet;}

    abstract public Class<T> getClazz();

    // Créer un EM et ouvrir une transaction
    protected EntityManager getEntityManager() {
        EntityManager em = DAOUtil.getEmf().createEntityManager();
        return (em);
    }

    // Fermer un EM et défaire la transaction si nécessaire
    protected void closeEntityManager(EntityManager em) {
        if (em != null) {
            if (em.isOpen()) {
                EntityTransaction t = em.getTransaction();
                if (t.isActive()) {
                    try {
                        t.rollback();
                    } catch (PersistenceException e) {}
                }
            }
            em.close();
        }
    }

    public T create(T t) {
        EntityManager em = getEntityManager();
        try {
            em.getTransaction().begin();
            em.persist(t);
            em.getTransaction().commit();
        } catch (Exception e) {
            em.getTransaction().rollback();
            throw new DataAccessException("Error création : " + t,e);
        } finally {closeEntityManager(em);}
        return t;
    }
}
```

Voici la classe *GenericDAOJPAImpl* qui implémente les méthodes de création, lecture, modification et destruction (CRUD) dans la BDD en utilisant un *EntityManager*.

Puis nous avons créé des classes qui étendent cette classe générique et qui implémentent les méthodes propres aux tables.

```
* Classe qui implémente les méthodes propres à la table des Utilisateurs dans la BDD avec JPA
* @author CDA
*/
public class UserDaoImpl extends GenericDAOJPAImpl<UserEntity, Integer> implements IUserDAO {

    // logger
    private static final Logger logger = LoggerFactory.getLogger(UserDaoImpl.class);

    @Override
    public Class<UserEntity> getClazz() {

        return UserEntity.class;
    }

    /**
     * Retourne le UserEntity correspondant au couple login/password dans la BDD
     */
    @Override
    public UserEntity GetUserByLoginByUser(String login, String password) {

        EntityManager em = DAOUtil.getEmf().createEntityManager();

        UserEntity user = null;

        try {
            em.getTransaction().begin();
            String req = "select Object(t) from " + UserEntity.class.getName()
                + " t where t.login = :login and t.password =:password";

            user = em.createQuery(req, UserEntity.class).setParameter("login", login).setParameter("password", password)
                .getSingleResult();

        } catch (Exception e) {
            logger.error("Association login + password non trouvé dans la BDD");
        } finally {
            em.close();
        }
        return user;
    }
}
```


3) Développement de la partie back-end de l'interface utilisateur web

La partie Controller (servlet)

Une servlet est un **composant Web** de Java EE. Elle permet de traiter une requête entrante sur un serveur et de générer une réponse dynamique.

Nous avons utilisé là le patron de conception « **front Controller** » où toutes les demandes seront traitées par un seul gestionnaire, puis envoyées au gestionnaire approprié pour ce type de demande. Le contrôleur frontal utilisera d'autres aides pour réaliser le mécanisme de répartition.

Nous avons créé la classe `FrontController` unique qui implémente l'interface `HttpServlet` définissant une servlet utilisant le protocole http.

```
@WebServlet(urlPatterns = "/")
public class FrontController extends HttpServlet {

    // logger
    private static final Logger logger = LoggerFactory.getLogger(FrontController.class);

    private static final long serialVersionUID = 1L;

    public FrontController() {
        super();
    }

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

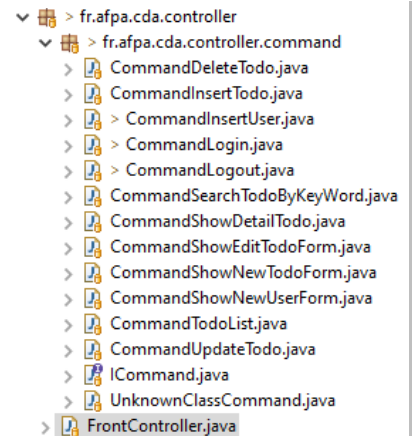
        doProcess(request, response);
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        doProcess(request, response);
    }
}
```

Dans cette classe Controller, on fait appel à 2 méthodes qui nous permettent de redistribuer dynamiquement vers les JSP ou les classes `Command` (qui implémentent l'interface `ICommand`).

```
public interface ICommand {
    public String execute(HttpServletRequest request, HttpServletResponse response);
}
```



Les 2 méthodes sont :

- `getCorrespondingClass ()` qui récupère le paramètre "action" de la requête http, et charge la classe `Command` correspondante.

```
private Class<?> getCorrespondingClass(HttpServletRequest request) {
    // 1. récupérer le paramètre "action" sous forme d'un string
    String action = request.getParameter("action");

    Logger.info("action récupérée : " + action );

    String CommandClass = null ;

    try {
        CommandClass = Constants.CMD_PACKAGE + Constants.CMD_PREFIX + action;
        // 2. loader la classe commande correspondante
        return Class.forName(CommandClass);
    } catch (ClassNotFoundException e) {
        Logger.info("Command Classe : " + CommandClass + " est inconnue");
        //e.printStackTrace();
        return UnknownClassCommand.class;
    }
}
```

- `doProcess ()` qui, grâce à la méthode ci-dessus, instancie la classe `Command` correspondante, appelle la méthode `execute ()` de cette instance et récupère le retour de cette méthode pour rediriger, soit vers une classe `Command`, soit vers une `JSP` (partie Vue).

```
protected void doProcess(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

    ICommand commandInstance = null;

    try {
        // 3. l'instancier
        commandInstance = (ICommand) getCorrespondingClass(request).newInstance();
    } catch (InstantiationException | IllegalAccessException e) {
        e.printStackTrace();
    }

    // 4. appelle la méthode execute() de cette instance
    String nextStep = commandInstance.execute(request, response);

    // 5. récupérer le retour de cette méthode pour décider de la next step
    if (nextStep.startsWith("redirect")) {
        //Renvoie vers le front controller
        response.sendRedirect(request.getContextPath() + "/do?action=" + nextStep.replace("redirect:", ""));
    } else {
        //Renvoie vers une JSP
        request.getRequestDispatcher(Constants.JSP_ROOT + nextStep + ".jsp").forward(request, response);
    }
}
```

Par exemple ici la classe `CommandLogin` qui teste que l'association Login / Password récupérée est présente dans la base de données.

Si c'est bon, elle redirige vers `CommandToDoList` qui va permettre d'afficher la liste de l'utilisateur, sinon elle renvoie à `login.jsp` qui va générer la page web dynamiquement.

```
public class CommandLogin implements ICommand {

    // logger
    private static final Logger logger = LoggerFactory.getLogger(CommandLogin.class);

    IUserService userService;

    @Override
    public String execute(HttpServletRequest request, HttpServletResponse response) {

        logger.info("== In CommandLogin ==");

        userService= new UserServiceImpl();

        //Récupère la session ou en crée une
        HttpSession session = request.getSession(true);

        //recupère la valeur de l'input dont l'attribut "name" = "login"
        String login = request.getParameter("login");
        //recupère la valeur de l'input dont l'attribut "name" = "password"
        String password = request.getParameter("password");

        try {
            LoginDTO loginDTO = new LoginDTO(login,password);

            //test de l'existence de l'association Login / password dans la BDD
            UserDTO userDTO = userService.getUser(loginDTO);

            logger.info("== Login/password trouvé ==");

            //On passe l'attribut user en session
            session.setAttribute("user", userDTO);

            //On demande l'affichage des Todos
            return "redirect:ToDoList";
        } catch (Exception e) {

            logger.info("== Login/password non trouvé ==");
            //On renvoie vers la JSP login
            return "login";
        }
    }
}
```

La partie service

La partie service regroupe les composants de traitement métier utilisés essentiellement par les classes Command dans la partie contrôleur.
Ce sont des classes délivrant un service en utilisant des objets de transfert de données (DTO), des **mappeurs** et des objets **Entity** (pour interagir avec la BDD).

```
public class UserDTO {  
  
    private Integer id;  
    private String firstname;  
    private String lastname;  
    private String login;  
    private String password;  
}
```

```
public class LoginDTO {  
  
    private String login;  
    private String password;  
}
```

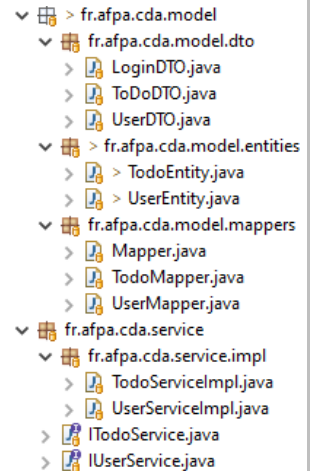
Ces classes service implémentent des interfaces déclarant les méthodes répondant aux exigences du métier.

```
public interface IUserService {  
  
    public UserDTO GetUser(LoginDTO loginDTO);  
  
    public UserDTO InsertUser(UserDTO userDTO);  
  
    public List<UserDTO> GetUsersByKeyword(String keyword);  
  
    public UserDTO GetUserByLogin(String login);  
}
```

Ci-dessous la classe `UserServiceImpl` utilisée dans la classe `CommandLogin` vue précédemment. Elle permet d'implémenter la méthode nécessaire pour renvoyer le `UserDTO` correspondant au login et mot de passe contenu dans le `LoginDTO` passé en paramètre.

```
public class UserServiceImpl implements IUserService {  
  
    // logger  
    private static final Logger logger = LoggerFactory.getLogger(UserServiceImpl.class);  
  
    private IUserDAO userDAO = new UserDAOImpl();  
  
    // -----  
  
    /**  
     * retourne le UserDTO dans la BDD correspondant au LoginDTO passé en  
     * paramètre  
     */  
    @Override  
    public UserDTO GetUser(LoginDTO loginDTO) {  
  
        if (loginDTO.getLogin() == null || loginDTO.getPassword() == null) {  
            throw new RuntimeException();  
        } else {  
  
            UserEntity user = userDAO.GetUserByLoginByUser(loginDTO.getLogin(), loginDTO.getPassword());  
  
            logger.info("== UserDTO recu : " + Mapper.map(user, UserDTO.class));  
  
            return Mapper.map(user, UserDTO.class);  
        }  
    }  
}
```

Pour cela, elle va instancier un objet `userDAO` et va faire appel à sa méthode `GetUserByLoginByUser()` pour récupérer un objet Entity.



Elle va ensuite utiliser un mapper générique sur l'Entity pour retourner le [UserDTO](#).

```
/**
 * Mapper générique qui mappe d'un objet à un autre ou d'une liste d'objet à une autre liste
 * @author CDA
 */
public class Mapper {

    private static final ModelMapper mapper = new ModelMapper();

    private Mapper() {}

    public static <T, E> T map(E source, Class<T> destinationType) {
        return destinationType.cast(mapper.map(source, destinationType));
    }

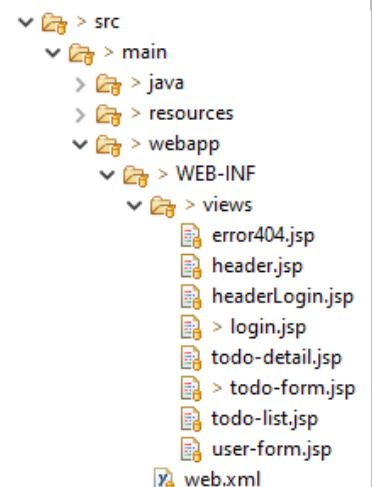
    public static <T, E> List<T> map(List<E> source, Class<T> destinationType) {
        List<T> listDestination = new ArrayList<T>();
        for (E e : source) {
            listDestination.add(map(e, destinationType));
        }
        return listDestination;
    }
}
```

4) Développement de la partie frontend de l'interface utilisateur web

Le **JavaServer Pages** ou **JSP** est une technique basée sur Java qui nous a permis de créer dynamiquement du code HTML en incluant des variables Java en les plaçant entre les marqueurs `<%= et %>`.

Nous avons aussi utilisé **Bootstrap** qui est une collection d'outils utiles à la création du design et qui contient des codes HTML et CSS, des formulaires, boutons et autres éléments interactifs.

Nous avons par ailleurs utilisé un langage de script particulier, appelé **Expression Language** (EL) destiné à réduire l'injection de code java au sein des pages JSP ainsi qu'à étendre les possibilités des taglibs, tel que la **JSTL** (ex : [annexe 9](#)).



Voici la page Login.jsp qui est la première page de notre application (définie dans le web.xml) et vers laquelle peut aussi renvoyer la classe `CommandLogin` présentée plus haut, si la combinaison login/mot de passe est absente de la BDD.

```
<%@page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@page import="java.util.Date"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>Veuillez vous authentifier</title>
<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css">
</head>

<body>
<jsp:include page="headerLogin.jsp"></jsp:include>
<div class="container col-md-8 col-md-offset-3">
<h1>Formulaire de Connexion</h1>
<form action="<%=request.getContextPath()%>/do?action=Login" method="POST">
<div class="form-group">
<label for="Login">Login :</label> <input type="text"
    class="form-control" id="Login" placeholder="Login" name="Login" value='${login}' autofocus
    required>
</div>
<div class="form-group">
<label for="password">Mot de passe :</label> <input type="password"
    class="form-control" id="password" placeholder="Mot de passe"
    name="password" required>
</div>
<button type="submit" class="btn btn-primary">Connexion</button>
</form>
</div>
</body>
</html>
```

Ce **Jsp** va par l'intermédiaire d'un formulaire, récupérer le login et le mot de passe saisi par l'utilisateur et va renvoyer une requête http POST vers le Contrôleur (Servlet) qui va ensuite comme on l'a vu l'aiguiller vers la classe `CommandLogin`.

ToDo Application

[Se connecter](#) [S'enregistrer](#)

Formulaire de Connexion

Login :

Mot de passe :

Et voici l'IHM correspondant

5) Les tests

L'application a été développée en y implémentant des test unitaires. Pour cela, nous avons utilisé **JUnit** pour l'exécution de tests unitaires automatisés.

Ci-dessous le test de la méthode GetUserByLogin() du service UserServiceImpl :

```
class UserServiceImplTest {

    String login="mathieu123";
    String password="psw123";

    IUserService serviceUser;

    @BeforeEach
    public void initUserService() {
        serviceUser = new UserServiceImpl();
    }

    @Test
    @Timeout(5)
    void testGetUserByLogin() {

        // ARRANGE
        LoginDTO loginDTO = new LoginDTO(login,password);

        UserDTO ExpecteduserDTO = serviceUser.GetUser(loginDTO);

        //ACT
        UserDTO TestuserDTO = serviceUser.GetUserByLogin(login);

        //ASSERT
        assertThat(TestuserDTO.getFirstname()).isEqualTo(ExpecteduserDTO.getFirstname());
    }
}
```

2. Précisez les moyens utilisés :

Diagrammes : www.diagrams.net

Maquettes : Pencil

IDE : Eclipse

Gestion BDD : PgAdmin

Gestion de versions : Git/Gitlab

Suivi de production : JIRA

3. Avec qui avez-vous travaillé ?

Ignacio Lopez (CDA 8)

Activité-type

2

Concevoir et développer la persistance des données en intégrant les recommandations de sécurité

Exemple n° 1  **Projet InfoPlus**

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

Dans le cadre du projet InfoPlus, j'ai conçu et développé une Base de Données relationnelle pour la gestion de stock de la société « InfoPlus ». On m'a fourni un cahier des charges contenant l'expression du besoin et je pouvais interroger le Maître d'ouvrage si besoin. Pour cela, j'ai utilisé la méthode **MERISE** pour concevoir un Système d'Information (SI) en séparant les données et les traitements à effectuer en plusieurs modèles conceptuels, logique et physique.

Première étape : Analyse des données (annexe 10)

Cette étape consiste à analyser toutes les informations du cahier des charges et interroger le maître d'ouvrage si besoin de précisions dans le but d'établir :

- Le **dictionnaire des données** qui est un tableau qui fait l'inventaire des éléments nécessaires au SI et leurs caractéristiques (entités, identifiant, nom, type, taille, persistés ou calculés, règles de calcul, etc.).
- **Les règles de gestion** qui à l'échelle de l'entreprise va s'appliquer systématiquement dans les cas qu'elle doit régir.

Deuxième étape : Modèle Conceptuel des Données (annexe 11)

J'ai ensuite élaboré le MCD permettant de décrire le SI à l'aide d'entités et d'associations en contrôlant les règles de normalisation (nom **entité**, **association** et **attribut** uniques, présence d'identifiant, **cardinalité** des associations).

Troisième étape : Modèle Logique des Données (annexe 11)

A partir du MCD, j'ai fait le MLD qui est la représentation de la structure de la base de données (transformation en **tables**, **champs** et **clés primaires**, migration des **clés étrangères**).

Quatrième étape : Modèle Physique des Données

Elaborer le MPD consiste à implémenter le modèle dans le **SGBD** (PostgreSQL). Pour cela j'ai créé les scripts suivants que j'ai exécutés en m'aidant de l'outil **pgAdmin**.

1 - Créer un utilisateur et la base de données avec **Postgres psql**

```
//Connexion psql avec l'utilisateur postgres
psql -p 5432 -h localhost -U postgres

//Créer un compte utilisateur nommé u_entreprise
CREATE USER u_entreprise;

//Affectez les privilèges de login et de création de base de données à l'utilisateur u_entreprise.
ALTER ROLE u_entreprise WITH PASSWORD 'postgres' CREATEROLE CREATEDB;

//Connexion avec utilisateur u_entreprise sur serveur postgres
\c postgres u_entreprise

//Creation de la base de données nommée db_infoplus ayant comme propriétaire le compte utilisateur nommé u_entreprise
CREATE DATABASE db_infoplus OWNER u_entreprise;

//Connexion avec utilisateur u_entreprise sur serveur infoplus
\c db_infoplus u_entreprise
```

2 - Script de création des tables et contraintes sur pgAdmin :

DOSSIER PROFESSIONNEL (DP)

Ici on peut voir la création des tables T_INDIVIDU et T_EMPLOYE en spécifiant leurs types, longueurs, clés primaires et contraintes.

```
-- Création des Tables InfoPlus-----
CREATE TABLE T_INDIVIDU(
    IdIndividu SERIAL,
    NomIndividu VARCHAR(50),
    PrenomIndividu VARCHAR(50),
    AdresseIndividu VARCHAR(150),
    NumeroTelephoneIndividu VARCHAR(50),
    AdresseMailIndividu VARCHAR(100),
    IdVille INTEGER
);

CREATE TABLE T_EMPLOYE(
    IdEmploye SERIAL,
    NumeroMatriculeEmploye VARCHAR(50),
    TrigrammeSalarie VARCHAR(3),
    SalaireEmploye MONEY,
    IdEmployeChef INTEGER,
    IdEquipe INTEGER,
    idFonction INTEGER,
    IdIndividu INTEGER,
    IdDivision INTEGER
);

-- Création des clés primaires -----
ALTER TABLE T_INDIVIDU ADD PRIMARY KEY (IdIndividu);
ALTER TABLE T_EMPLOYE ADD PRIMARY KEY (IdEmploye);

----- Création des contraintes -----

-- Création des clés étrangères --
ALTER TABLE T_INDIVIDU
ADD CONSTRAINT fk_individu_ville FOREIGN KEY (IdVille) REFERENCES T_VILLE(IdVille)
ON DELETE CASCADE;

ALTER TABLE T_EMPLOYE
ADD CONSTRAINT fk_employe_employe FOREIGN KEY (IdEmployeChef) REFERENCES T_EMPLOYE(IdEmploye)
ON DELETE CASCADE,
ADD CONSTRAINT fk_employe_equipe FOREIGN KEY (IdEquipe) REFERENCES T_EQUIPE(IdEquipe)
ON DELETE CASCADE,
ADD CONSTRAINT fk_employe_fonction FOREIGN KEY (idFonction) REFERENCES T_FONCTION(idFonction)
ON DELETE CASCADE,
ADD CONSTRAINT fk_employe_individu FOREIGN KEY (IdIndividu) REFERENCES T_INDIVIDU(IdIndividu)
ON DELETE CASCADE,
ADD CONSTRAINT fk_employe_division FOREIGN KEY (IdDivision) REFERENCES T_DIVISION(IdDivision)
ON DELETE CASCADE;

-- Champ non null --
ALTER TABLE T_INDIVIDU
ALTER COLUMN NomIndividu SET NOT NULL;

ALTER TABLE T_EMPLOYE
ALTER COLUMN IdIndividu SET NOT NULL;

-- Champ unique --
ALTER TABLE T_EMPLOYE ADD UNIQUE (NumeroMatriculeEmploye);
ALTER TABLE T_EMPLOYE ADD UNIQUE (TrigrammeSalarie);
```

3 - Script d'insertion des données, ici un extrait pour la table T_EMPLOYE :

DOSSIER PROFESSIONNEL (DP)

```
INSERT INTO T_EMPLOYE (NomIndividu, PrenomIndividu, AdresseIndividu, NumeroTelephoneIndividu, AdresseMailIndividu, IdVille) VALUES
('White', 'Lila', '920-6296 Sed St.', '09 23 11 28 42', 'odio@faciliseget.co.uk', 26),
('Slater', 'Maxwell', '920-6296 Sed St.', '06 44 70 43 52', 'sociis.natoque@elitafeugiat.ca', 58),
('Rodgers', 'Justina', 'Ap #824-5783 Suspendisse Ave', '08 70 21 71 10', 'diam.luctus.lobortis@sedeu.org', 67);
```

4 - Puis j'ai élaboré des **procédures ou fonctions stockées, triggers et vues** pour répondre aux besoins exprimés :

Exemple Fonction Stockée :

```
• FCT02 : fonction qui, pour un salarié donné, génère un trigramme à partir de son
prénom et de son nom de famille.
le trigramme est unique. Il est composé de la 1ère lettre du prénom, la 1ère lettre du nom
et la 2ème lettre du nom. Ex : trigramme (Bob Marley) = BMA. Si le trigramme existe déjà, vous
devez prendre la lettre suivante du nom.
*/

CREATE OR REPLACE FUNCTION FCT02 (IN IdEmpl INT)
RETURNS Char(3)
AS $$
DECLARE

-- Variable nom et prénom de l'employé qui sont du même type présent dans la table
v_nom T_INDIVIDU.NomIndividu%TYPE;
v_prenom T_INDIVIDU.PrenomIndividu%TYPE;

-- Les 2 premières lettres du trigramme
DebutTrigramme char(2);
-- La dernière lettre du trigramme
FinTrigramme char(1);
-- Le trigramme formé des 3 lettres
Trigramme Char(3);

BEGIN
--Récupération du nom et prénom dans la table T_INDIVIDU en fonction de l'IdEmpl rentré en paramètre
v_nom=(SELECT NomIndividu From T_INDIVIDU inner join T_EMPLOYE using(IdIndividu) Where IdEmploye =IdEmpl);
v_prenom=(SELECT PrenomIndividu From T_INDIVIDU inner join T_EMPLOYE using(IdIndividu) Where IdEmploye =IdEmpl);

--On purge les caractères - et ' du nom et prénom
v_prenom = REGEXP_REPLACE(v_prenom, '-', '');
v_prenom = REGEXP_REPLACE(v_prenom, ' ', '');

-- Le trigramme aura toujours les 2 mêmes lettres de début (1er lettre du prénom concaténée avec la 1er du nom)
-- On profite pour faire disparaître les espaces et mettre en majuscule
DebutTrigramme = SUBSTR(trim(upper(v_prenom)), 1, 1) || SUBSTR(trim(upper(v_nom)), 1, 1);

-- On parcourt le nom lettre par lettre à partir de la seconde lettre jusqu'à la fin du nom
FOR i IN 2..length(v_nom) LOOP
--on concaténne au début de trigramme déjà généré chacune son tour les lettres composant le nom
FinTrigramme = SUBSTR(trim(upper(v_nom)), i, 1);
Trigramme = DebutTrigramme || FinTrigramme;

-- On teste si la combinaison du Trigramme généré est déjà présente dans la table
if (SELECT COUNT(TrigrammeSalarie) FROM T_EMPLOYE where TrigrammeSalarie = Trigramme)=0
-- si le trigramme généré n'existe pas encore, on le renvoi
then RETURN Trigramme;
END if;
END LOOP;

-- si le trigramme existe déjà, on renvoi '000'
if (SELECT COUNT(TrigrammeSalarie) FROM T_EMPLOYE where TrigrammeSalarie = Trigramme)>=1
then RETURN '000';
END if;
END;
$$ LANGUAGE plpgsql;
```

Exemple trigger :

DOSSIER PROFESSIONNEL (DP)

```
TRG03 : trigger permettant, lors de la création d'un salarié, de calculer et d'insérer le trigramme d'un salarié. Ce trigger utilise la fonction FCT02.
--*/
CREATE OR REPLACE FUNCTION FC_NouvEmploye()
RETURNS trigger AS
$$
BEGIN
    IF FCT02 (New.IdEmploye) = '000' THEN
        -- On signale que les différentes combinaisons de trigramme existent déjà dans la base mais on permet l'insertion des autres données pour l'employé
        RAISE NOTICE 'Trigramme déjà présent dans la table donc non généré pour %',
        (SELECT PrenomIndividu || ' ' || NomIndividu FROM T_INDIVIDU INNER JOIN T_EMPLOYE using(idIndividu) WHERE idEmploye=New.IdEmploye) ;
        RETURN NEW;
    ELSE
        -- Si le trigramme généré n'est pas déjà présent dans la table, on l'insère dedans
        UPDATE T_EMPLOYE
        SET TrigrammeSalarie = FCT02 (New.IdEmploye)
        WHERE IdEmploye=New.IdEmploye;
        RETURN NEW;
    END IF;
END;
--*/
LANGUAGE 'plpgsql';

--CREATION TRIGGER--
DROP TRIGGER IF EXISTS TRG03 ON T_EMPLOYE CASCADE;

CREATE TRIGGER TRG03 AFTER INSERT ON T_EMPLOYE
FOR EACH ROW
EXECUTE PROCEDURE FC_NouvEmploye();

--/* TEST-----
UPDATE T_EMPLOYE
SET TrigrammeSalarie = FCT02 (IdEmploye) ;
--*/
```

Exemple Vue :

```
VUE01 : vue permettant de connaître l'ensemble des informations des salariés
(nom, prénom, mail, numéro de sa division, nom de sa division, etc...)
--*/
CREATE OR REPLACE VIEW VUE01 AS
SELECT distinct
    t_individu.nomindividu as Nom,
    t_individu.prenomindividu as Prenom,
    t_employe.numeromatriculeemploye as Matricule,
    t_individu.adresseindividu || ' ' || t_ville.codepostal || ' ' || t_ville.nomville || ' ' || t_pays.libellepays as Adresse,
    t_individu.numerotelephoneindividu as "Numero de telephone",
    t_individu.adressemailindividu as "Adresse Mail",
    t_fonction.libellefonction as Fonction,
    t_employe.salaireemploye as Salaire,
    t_division.numerodivision || ' ' || t_division.nomdivision as Division,
    A_TRAVAILLER.IdEquipe as "Dernière Equipe",
    T_COMPETENCE.LibelleCompetence as "Spécialité de l'équipe"
FROM T_INDIVIDU,
T_EMPLOYE,
T_FONCTION,
T_DIVISION,
T_VILLE,
T_PAYS,
A_TRAVAILLER,
T_EQUIPE,
T_COMPETENCE,
T_PERIODE
WHERE t_individu.idindividu = t_employe.idindividu
AND t_employe.idfonction = t_fonction.idfonction
AND t_employe.iddivision = t_division.iddivision
AND t_individu.idville = t_ville.idville
AND t_ville.idpays = t_pays.idpays
AND t_employe.IdEmploye = A_TRAVAILLER.IdEmploye
AND A_TRAVAILLER.IdPeriode = T_PERIODE.IdPeriode
AND A_TRAVAILLER.IdEquipe=T_EQUIPE.IdEquipe
AND T_EQUIPE.IdCompetence=T_COMPETENCE.IdCompetence

-- TEST --
-- SELECT * From VUE01
```

DOSSIER PROFESSIONNEL (DP)

5 - Exécution des scripts :

Afin d'assurer le bon fonctionnement de l'ensemble, la phase de test a été effectuée à l'aide de scripts manuels exécutés dans un ordre précis :

01-Script_CreationUtilisateur.sql
02-Script_SuppressionTables.sql
03-Script_CreationTables.sql
04-Script_CreationFonctionsProcédures.sql
05-Script_CreationTriggers.sql
06-Script_CreationVues.sql
07-Script_InsertionDonnees.sql

2. Précisez les moyens utilisés :

Conception de la Base de Données :

- Dictionnaire des données sous Excel
- MCD et MLD sous Looping

Mise en place de la BDD :

- PostgreSQL avec pgAdmin 4

Développement des composants :

- Langage PL/PGSQL

3. Avec qui avez-vous travaillé ?

Seul

4. Contexte

Nom de l'entreprise, organisme ou association  Cliquez ici pour taper du texte.

Chantier, atelier, service  Cliquez ici pour taper du texte.


Période d'exercice  Du : Cliquez ici au : Cliquez ici

5. Informations complémentaires (facultatif)

Activité-type

3

Concevoir et développer une application multicouche répartie en intégrant les recommandations de sécurité

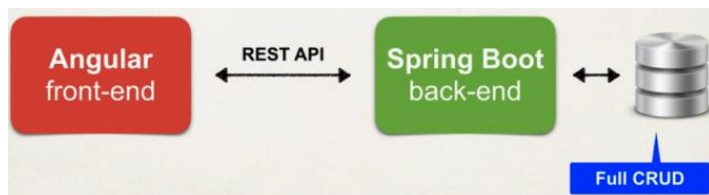
Exemple n° 1  E-commerce Backend

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

En équipe, nous avons développé une application de e-commerce. Nous avons analysé le **cahier des charges** pour déterminer les exigences et les attentes du projet. **GitLab**, se basant sur les fonctionnalités du logiciel Git, nous a permis de piloter des dépôts de code source et de gérer leurs différentes versions. Nous avons choisi **JIRA** pour assurer la gestion des exigences et la répartition des tâches.

Conception :

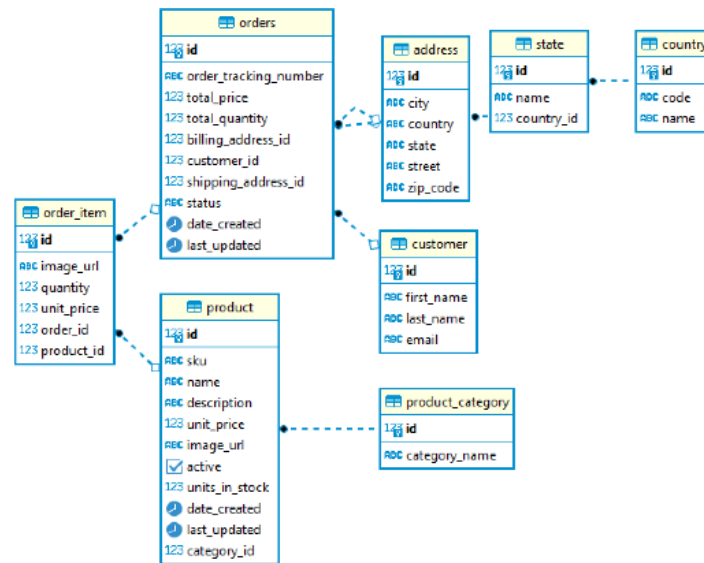
Pour notre back-end nous avons utilisé **Spring Boot**, afin de tirer parti de **Spring Data REST** pour minimiser le code et créer une **API REST** qui fournira des services à la partie front. **Maven** nous a servi à gérer les dépendances et lancer les builds. Concernant Spring Boot, nous avons utilisé **Spring Boot JPA** afin de faciliter les échanges avec la BDD (via son ORM) et **Spring MVC** afin de gérer les requêtes http.



Le **design pattern MVC** a été appliqué : Le modèle se compose des objets Java qui contiennent les données, de la vue qui affiche les informations (par le biais du front Angular) et des contrôleurs qui assurent la communication entre le backend et le frontend.

L'application a été **divisée en plusieurs couches** en conformité avec les principes de conception (responsabilité unique, couplage faible et cohésion forte) ce qui a permis de faciliter la maintenance du code et d'intervenir indépendamment sur chacune des couches de l'application sans que les autres soient impactées.

```
fr.mat.ecommerce
├── config
│   ├── MyAppConfig.java
│   └── MyDataRestConfig.java
├── controller
│   └── CheckoutController.java
├── dao
│   ├── CountryRepository.java
│   ├── CustomerRepository.java
│   ├── ProductCategoryRepository.java
│   ├── ProductRepository.java
│   └── StateRepository.java
├── dto
│   ├── Purchase.java
│   └── PurchaseResponse.java
├── entity
│   ├── Address.java
│   ├── Country.java
│   ├── Customer.java
│   ├── Order.java
│   ├── OrderItem.java
│   ├── Product.java
│   ├── ProductCategory.java
│   └── State.java
├── service
│   ├── CheckoutService.java
│   ├── CheckoutServiceImpl.java
│   └── SpringBootEcommerceApplication.java
├── src/main/resources
└── application.properties
```



Développement :

Pour la création des modèles dans la couche « entity », l'utilisation de Spring data JPA et de ses annotations permettent de rattacher le Bean à la BDD.

```

@Table(name="order_item")
@Getter
@Setter

public class OrderItem {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name="id")
    private Long id;

    @Column(name="image_url")
    private String imageUrl;

    @Column(name="unit_price")
    private BigDecimal unitPrice;

    @Column(name="quantity")
    private int quantity;

    @Column(name="product_id")
    private Long productId;

    @ManyToOne
    @JoinColumn(name = "order_id")
    private Order order;
}
    
```

@Entity Définit qu'une classe est une entité.

@Id Définit l'attribut qui sert de clé primaire dans la table.

@Table Permet de définir les informations sur la table représentant cette entité en base de données.

@GeneratedValue Indique la stratégie à appliquer pour la génération de la clé lors de l'insertion d'une entité en base.

@Column Permet de déclarer des informations relatives à la colonne sur laquelle un attribut doit être mappé.

@ManyToOne définit une relation n:1 entre deux entités.

@JoinColumn est utilisé pour spécifier une association entre une colonne et une entité ou une collection.

JPA Repository est une interface qui a implémenté dans la couche **DAO** les méthodes **CRUD** sans avoir à les définir explicitement :

```
//@CrossOrigin("url") accept calls from web browser scripts for the url origin
//@CrossOrigin("http://localhost:4200")
@RepositoryRestResource
public interface ProductRepository extends JpaRepository<Product, Long> {

    Page<Product> findByCategoryId(@RequestParam("id") Long id, Pageable pageable);

    //requete automatique : select * from Product p where p.name like concat('%',:name,'%')
    Page<Product> findByNameContaining(@RequestParam("name") String name, Pageable pageable);

    //ignore si les lettres en majuscule et minuscule
    Page<Product> findByNameIgnoreCaseContaining(@RequestParam("name") String name, Pageable pageable);
}
```

Dans La **couche service** se trouve les composants implémentant la logique métier (générer un nombre unique pour chaque commande d'achat, calculer le prix total, ...). Ces composants utilisent, entre autres, les méthodes d'accès à la BDD des Repository.

Ici une méthode pour générer un nombre unique qui est ensuite rattaché à une commande d'achat :

```
private String generateOrderTrackingNumber() {

    // generate a random UUID number (UUID version-4 for random) Universally Unique Identifier
    return UUID.randomUUID().toString();
}
```

La **couche Controller** gère les requêtes des utilisateurs. Elle est responsable de retourner une réponse avec l'aide du Model en créant des **Endpoints** afin que l'IHM puisse recevoir un service au moyen de requêtes **http**.

Dans l'exemple ci-dessous, à l'Endpoint « /api/checkout/purchase », la méthode `PurchaseResponse ()` récupère le corps de la requête grâce à l'annotation **@RequestBody**. Puis, elle le déserialise dans un *DTO* (`Purchase`) et le place en paramètre de la méthode `placeOrder ()` du service `checkoutService` pour renvoyer le résultat.

```
@RestController
@RequestMapping("/api/checkout")
public class CheckoutController {

    private CheckoutService checkoutService;

    public CheckoutController(CheckoutService checkoutService) {
        this.checkoutService = checkoutService;
    }

    @PostMapping("/purchase")
    public PurchaseResponse placeOrder(@RequestBody Purchase purchase) {

        PurchaseResponse purchaseResponse = checkoutService.placeOrder(purchase);

        return purchaseResponse;
    }
}
```

DOSSIER PROFESSIONNEL (DP)

2. Précisez les moyens utilisés :

IDE : Eclipse

Gestion BDD : PgAdmin

Gestion de versions : Git/Gitlab

Langage : Java

Framework : Spring

3. Avec qui avez-vous travaillé ?

Groupe CDA 8

Activité-type

3

Concevoir et développer une application multicouche répartie en intégrant les recommandations de sécurité

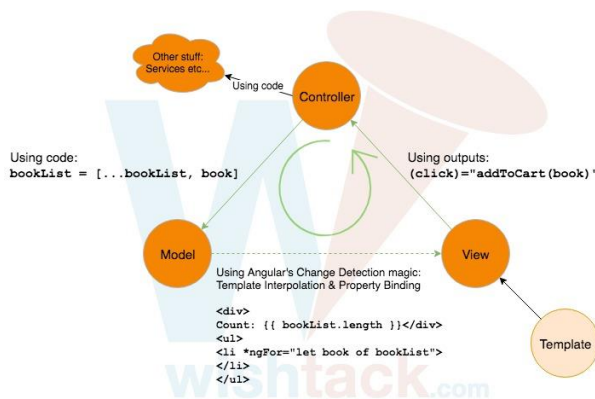
Exemple n° 1 ► WookieApp

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

En complément de la formation à l'AFPA, j'ai décidé de m'entraîner avec **Angular** en développant une interface utilisateur web **multicouche** et **responsive** consommant des données au format **json** envoyés par une API. L'objectif est de développer un Instagram pour Wookies.

Angular repose principalement sur une approche MVC où :

- Le **Controller** et le **Model** sont représentés par l'instance de la classe TypeScript du composant.
- La **Vue** (DOM) est générée à partir des instructions du **Template**. Elle déclenche des actions sur le Controller via des "outputs" (ou event binding).
- Le **Controller** met à jour l'état du "model".
- Grâce à son mécanisme de "Change Detection", Angular détecte les changements et met à jour la vue si nécessaire.



1^{ère} étape : Analyse

J'ai d'abord établi mon cahier des charges et déterminé quelles seraient les données pertinentes à utiliser et afficher.

2^{ème} étape : Maquettage de l'interface

J'ai décidé de maquetter l'interface avec du code HTML/CSS en statique en la rendant **responsive** pour que ce puisse aussi être utilisé comme une **application mobile** : c'est-à-dire avec un code qui répond à des normes lui permettant d'adapter l'affichage à tous types de supports, y compris aux écrans des mobiles.

Il y a plusieurs façons de le faire :

Tout d'abord en utilisant les **media queries** qui sont des règles CSS à appliquer pour changer le design d'un site en fonction des caractéristiques de l'écran (résolution, nombre de couleur, type d'écran, orientation, ...). Dans ce cas, c'est la balise `<link />` qui permet, dans le code HTML, de charger un fichier .css. On lui ajoute un attribut `media`, dans lequel on va écrire la règle qui doit s'appliquer pour que le fichier soit chargé.

```
HTML:
<head>
<link rel="stylesheet" href="style.css" /> <!-- Pour tout le monde -->
<link rel="stylesheet" media="screen and (max-width: 1280px)" href="petite_resolution.css" /> <!-- Pour ceux qui ont une résolution inférieure à 1280px -->
<title>Exemple Media queries</title>
</head>
```

On met ensuite les règles à appliquer directement dans la feuille de style.

```
CSS:
@media screen and (max-width: 1280px)
{
  /* Rédigez vos propriétés CSS ici */
}
```

Une autre solution est d'utiliser **Bootstrap** (une collection d'outils utiles à la création du design de sites et d'applications web) qui est plus simple et rapide à mettre en place.

Bootstrap comprend un système de grille fluide, réactif et adapté aux mobiles, qui s'adapte de manière appropriée jusqu'à 12 colonnes lorsque la taille de l'appareil ou de la fenêtre d'affichage augmente. Il comprend des classes prédéfinies pour des options de mise en page faciles (**annexe 13**).

3^{ème} étape : Développement

J'ai développé couche par couche en commençant par le service des **selfies** qui me sert récupérer les données qui seront implémentées dans le DOM via les Components.

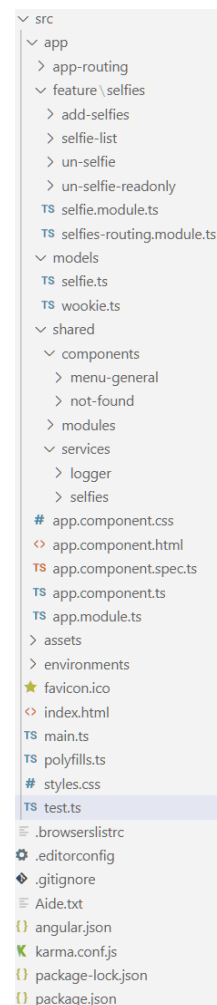
Pour cela, j'ai créé une méthode **getAllSelfies_asObservable ()** qui récupère le flux des données Observables ce qui permet de propager et consommer en temps réel les données dans les différents composants de l'application.

```
@Injectable({
  providedIn: 'root',
})
export class SelfieService {

  constructor(
    private _loggerService: LoggerService,
    private _httpClient: HttpClient
  ) {}

  //-----

  /**
   * methode pour récupérer les données en retournant une observable pour s'inscrire a la reception des tableaux de selfies
   * @returns
   */
  //on renvoie un observable contenant des tableaux de selfies
  getAllSelfies_asObservable(): Observable<Selfie[]>={
    // recupere les requetes http via api Mock simulé par Postman
    return this._httpClient.get<Selfie[]>(environment.apis.selfies.url);
  }
}
```



L'un des principaux concepts d'Angular est de voir une application comme une arborescence de composants. J'ai développé le composant `selfieList` qui affiche la liste des selfies à travers un autre composant enfant `selfie-readonly`.

Le composant `selfieList` souscrit au service `selfies` dans la méthode `ngOnInit()` et récupère un tableau de `Selfie`.

```
import { Wookie } from './wookie';
/**
 * représente un selfie d'un wookie
 */
You, last week | 1 author (You)
export class Selfie {
  id: number | undefined;
  image: string = '';
  imageAs63: string = '';
  wookie: Wookie;
  titre: string = '';

  constructor() {
    this.wookie = new Wookie();
  }
}

export class SelfieListComponent implements OnInit, OnDestroy {
  //liste des souscriptions
  _lesSouscriptions: Subscription[] = [];

  _lesSelfies: Selfie[] = [];

  constructor(
    private _loggerService: LoggerService,
    private _selfieService: SelfieService
  ) {}

  ngOnInit(): void {
    //pour recuperer les données en dur grace au selfieService
    //this._lesSelfies = this._selfieService.getAllSelfies();

    //pour souscrire à un observable
    const subscriptionEnCours = this._selfieService
      .getAllSelfies_asObservable()
      .subscribe((unTableau) => (this._lesSelfies = unTableau));
    // = .subscribe (function (unTableau) {this._lesSelfies = unTableau})

    //on récupère la souscription en cours dans le tableau de souscriptions
    this._lesSouscriptions.push(subscriptionEnCours);
  }

  //si le composant est détruit, on désouscrire toutes les subscriptions
  ngOnDestroy(): void {
    this._lesSouscriptions.forEach((item) => item.unsubscribe());
  }
}
```

```
import { Selfie } from './selfie';
/**
 * Classe représentant le wookie
 */
You, 4 months ago | 1 author (You)
export class Wookie {
  nom: string = '';
  selfies: Selfie[] = [];
}
```

Enfin, j'ai codé la vue : Pour transmettre des données à un "child component", j'ai communiqué avec ce dernier à l'aide du **Propertie Binding**. La directive structurale **ngFor** permet de boucler sur un tableau et d'injecter les éléments dans le DOM :

Dans le fichier html du composant `selfie-list` :

```
<app-un-selfie-readonly [unSelfie]="selfie" *ngFor="let selfie of lesSelfies"></app-un-selfie-readonly>
```

Dans le fichier Typescript du composant `selfie-readonly` :

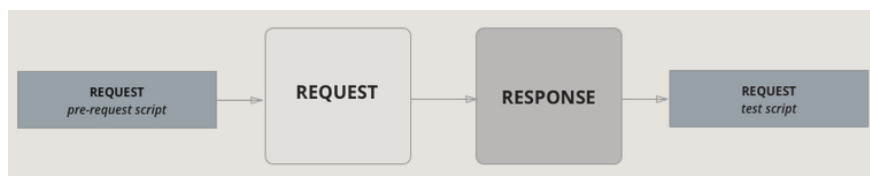
```
export class UnSelfieReadonlyComponent implements OnInit {  
  @Input()  
  public unSelfie: Selfie = new Selfie();  
}
```

Voici la page html du composant `selfie-readonly` affiché (IHM annexe 14) :

```
<article class="row mb-3 Regular shadow" (click)="clickPourConsultation()"> You, 3 days ago • deploiem  
  
...<header class="col">  
...<div class="row">  
...<div class="col-9">  
...<h2>{{ unSelfie.titre }}</h2>  
...</div>  
...<div class="col text-end">  
...[...]  
...</div>  
...</div>  
...</header>  
  
...<!-- force le retour à la ligne -->  
...<div class="w-100"></div>  
  
...<div class="col pb-2">  
...<img class="img-fluid" [src]="unSelfie.image" alt="">  
...<!-- ou src={{unSelfie.image}} -->  
...</div>  
  
...<!-- force le retour à la ligne -->  
...<div class="w-100"></div>  
  
...<div class="col pb-2 footer">  
...<div class="d-flex">  
...<div class="mr-2">  
...<button (click)="clickPourDuel()" type="submit" class="btn btn-info">Lancer le Duel</button>  
...</div>  
...</div>  
...</div>  
  
</article>
```

4^{ème} étape : Tests

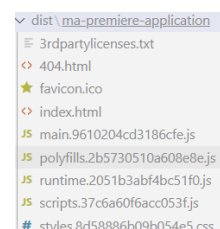
Pour **tester** cette interface, j'ai simulé grâce à **Postman** des Endpoints (mocking) qui reçoivent et envoient des requêtes REST (**annexe 15**).



5^{ème} étape : Déploiement

Pour compiler l'application pour la production, j'ai utilisé la commande « **ng build** ». Par défaut, cette commande utilise la configuration de construction de production.

Cette commande crée un dossier **dist** dans le répertoire racine de l'application avec tous les fichiers dont un service d'hébergement a besoin pour déployer l'application.



Ensuite, j'ai copié le contenu du dossier *dist/mon-nom-de-projet* sur mon serveur web local. Comme ces fichiers sont statiques, on peut les héberger sur n'importe quel serveur web (Node.js, Java, .NET) ou n'importe quelle plateforme de conception et d'hébergement d'applications web comme Firebase, Google Cloud ou App Engine.

J'ai utilisé un serveur local **httpserveur** en utilisant la commande **npx** qui me permet d'exécuter le script sans devoir l'installer (`npx http-server -p 8080 -c-1 ./dist/mapremierapplication/`).

On peut aussi configurer un déploiement automatique sur la plateforme **github pages**. Pour cela, j'ai installé le déploiement spécifique qui est **angular-cli-ghpages** trouvable sur le site **npm** (gestionnaire de paquets par défaut pour l'environnement d'exécution JavaScript Node.js).

J'ai modifié le fichier de [angular.json](#) en ajoutant la commande « **deploy** »

```
"scripts": {  
  "ng": "ng",  
  "start": "ng serve",  
  "build": "ng build",  
  "watch": "ng build --watch --configuration development",  
  "test": "ng test",  
  "deploy": "ng deploy --repo=https://github.com/MaTromatet/deployed-wookieApp.git --name= 'MaTromatet' -  
-email=mromatet@gmail.com --base-href=/deployed-wookieApp/"  
},
```

Puis, j'ai lancé la commande « **ng deploy** ».

DOSSIER PROFESSIONNEL (DP)

2. Précisez les moyens utilisés :

IDE : VSCode

Gestion de Versions : Git/Github

Language : HTML CSS Typescript

Framework: Angular

3. Avec qui avez-vous travaillé ?

seul

4. Contexte

Nom de l'entreprise, organisme ou association ► Cliquez ici pour taper du texte.

Chantier, atelier, service ► Cliquez ici pour taper du texte.

Période d'exercice ► Du : Cliquez ici au : Cliquez ici

5. Informations complémentaires (facultatif)

Déclaration sur l'honneur

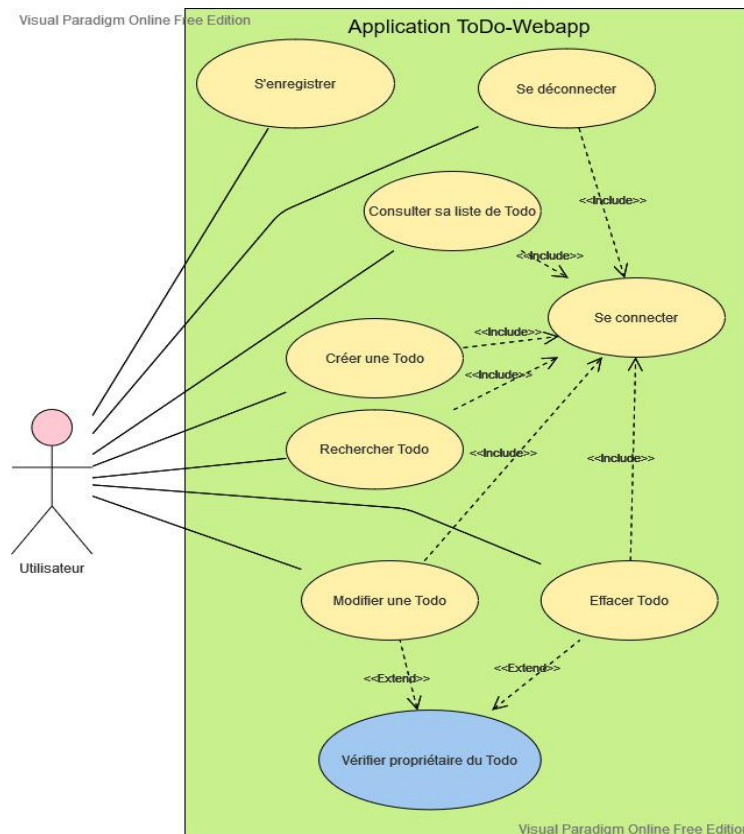
Je soussigné(e) [prénom et nom] **Mathieu romatet** ,
déclare sur l'honneur que les renseignements fournis dans ce dossier sont exacts et que je suis
l'auteur(e) des réalisations jointes.

Fait à **Toulouse** le **Cliquez ici pour choisir une date**
pour faire valoir ce que de droit.

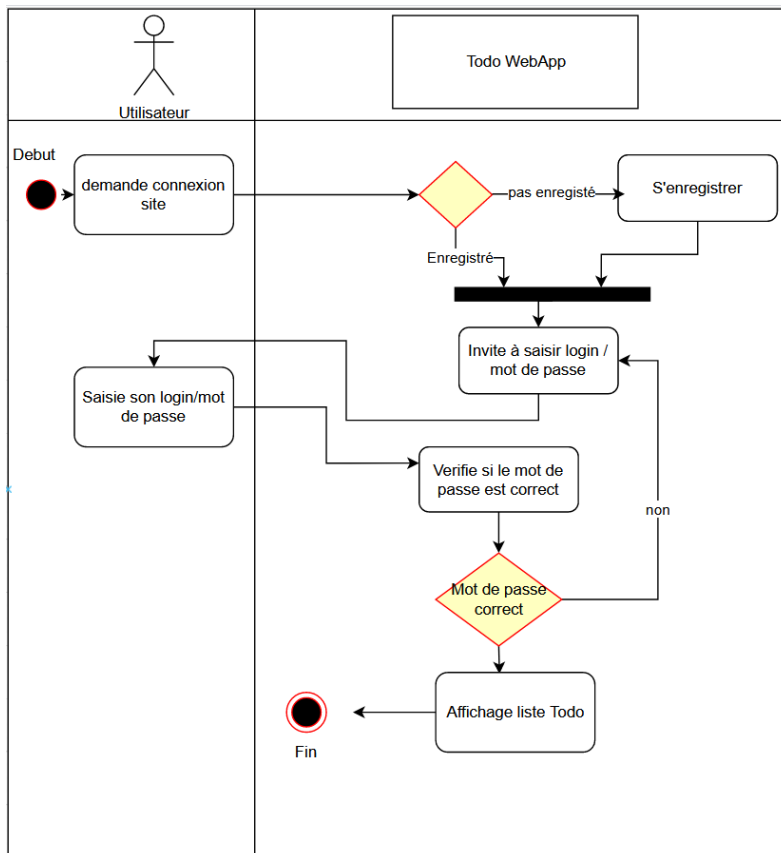
Signature :

ANNEXES

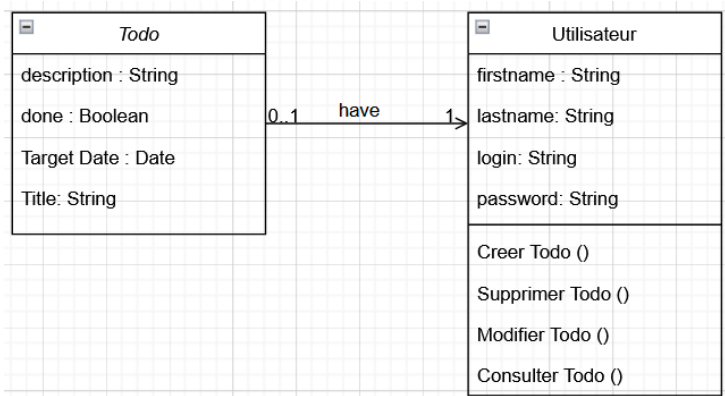
Annexe 1 : Diagramme des Cas d'utilisation (use case) : Todo WebApp



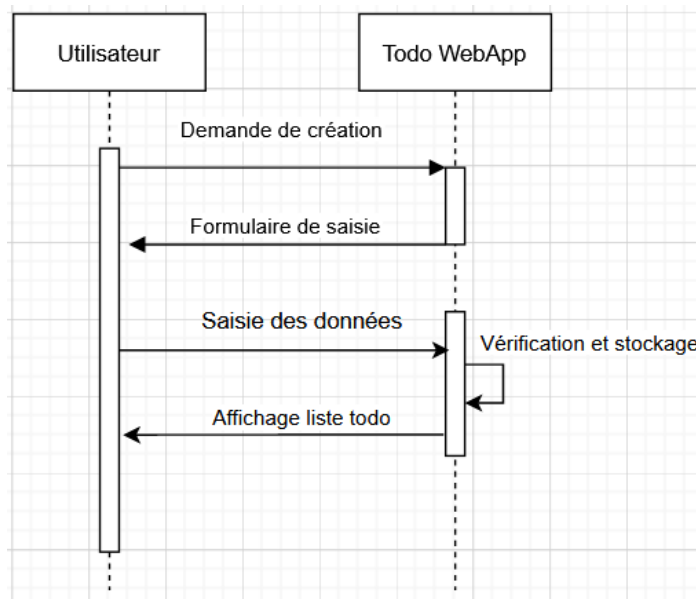
Annexe 2 : Diagramme d'Activité : Connexion *Todo WebApp*



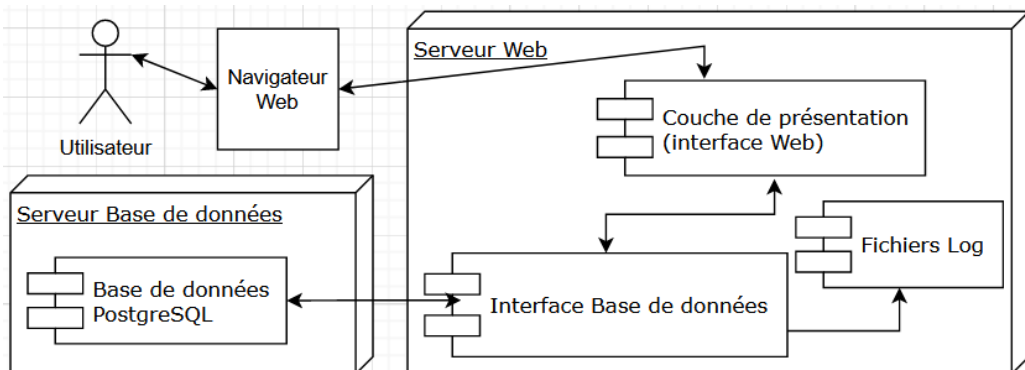
Annexe 3 : Diagramme de Classes : *Todo WebApp*



Annexe 4 : Diagramme de séquence : création d'une Todo : **Todo WebApp**



Annexe 5 : Diagramme de déploiement : **Todo WebApp**



DOSSIER PROFESSIONNEL (DP)

Annexe 6 : Maquettes IHM Todo WebApp

The wireframes illustrate the user interface for the Todo WebApp. The first wireframe shows a login form with fields for 'login' and 'Mot de passe', and buttons for 'Se connecter' and 'S'inscrire'. The second wireframe shows a form to 'Ajouter un nouveau Todo' with fields for 'Titre du Todo', 'Description Todo', 'Status du Todo', and 'Date d'échéance du Todo', and buttons for 'Rechercher' and 'Ajouter Todo'. The third wireframe shows the 'Liste des Todos' with a table of tasks. The fourth wireframe shows the 'Détail du Todo : JAVA' with a table of task details.

Titre	Date d'échéance	Status	Actions
JAVA	20 août 2022	En cours	Editer Effacer
Angular	9 mai 2022	En cours	Editer Effacer
Python	25 mars 2022	Terminée	Editer Effacer

Date d'échéance	Status	Description
20 août 2022	En Cours	Apprendre Java

Annexe 7 : Script création Base de données Todo WebApp

```
DROP TABLE IF EXISTS utilisateur CASCADE;  
DROP TABLE IF EXISTS todo CASCADE;
```

```
CREATE TABLE utilisateur (  
  id SERIAL PRIMARY KEY,  
  firstName varchar(20) DEFAULT NULL ,  
  lastName varchar(20) DEFAULT NULL,  
  login varchar(255) DEFAULT NULL,  
  password varchar(20) DEFAULT NULL,  
  CONSTRAINT unique_username UNIQUE (login)  
);
```

```
CREATE TABLE todo(  
  id SERIAL PRIMARY KEY,  
  description varchar(255) DEFAULT NULL,  
  done BOOLEAN NOT NULL,  
  targetDate DATE DEFAULT NULL,  
  title varchar(255) DEFAULT NULL,  
  idUtilisateur int  
);
```

```
ALTER TABLE todo  
ADD CONSTRAINT fk_user_todo FOREIGN KEY (idUtilisateur) REFERENCES utilisateur(id)  
ON DELETE CASCADE;
```

```
INSERT INTO utilisateur (firstName, lastName, login, password)  
VALUES  
( 'mathieu', 'romatet', 'mathieul23', 'psw123'),  
( 'ignacio', 'lopez', 'ignacio456', 'psw456'),  
( 'lopez', 'matthieu', 'lopez789', 'psw789');
```

```
INSERT INTO todo (description, done, targetDate, title, idUtilisateur)  
VALUES  
( 'apprendre java', false, '2022-08-20', 'java', 1),  
( 'apprendre python', false, '2022-05-09', 'python', 1),  
( 'amelioré ruby', true, '2022-03-25', 'java', 1),  
( 'apprendre javascript', true, '2022-10-22', 'javascript', 1),  
( 'Apprendre java', false, '2022-05-10', 'java', 2),  
( 'amelioré python', true, '2023-01-25', 'python', 2),  
( 'apprendre python', true, '2023-08-01', 'python', 3),  
( 'amelioré python', false, '2023-02-13', 'python', 3),  
( 'vacance', false, '2023-01-09', 'vacance', 3);
```

Annexe 8 : description *persistence.xml* pour connexion JPA **Todo WebApp**

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <persistence version="2.2"
3   xmlns="http://xmlns.jcp.org/xml/ns/persistence"
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd">
6   <persistence-unit name="todo-web-app">
7     <properties>
8       <property name="javax.persistence.jdbc.driver"
9         value="org.postgresql.Driver" />
10      <property name="javax.persistence.jdbc.url"
11        value="jdbc:postgresql://localhost:5432/DBTODO" />
12      <property name="javax.persistence.jdbc.user"
13        value="postgres" />
14      <property name="javax.persistence.jdbc.password"
15        value="postgres" />
16      <!-- Affichage des requetes -->
17      <property name="hibernate.show_sql" value="false" />
18      <property name="hibernate.format_sql" value="true" />
19      <property name="hibernate.hbm2ddl.auto" value="none" />
20      <property name="hibernate.dialect"
21        value="org.hibernate.dialect.PostgreSQL94Dialect" />
22    </properties>
23  </persistence-unit>
24 </persistence>
```

DOSSIER PROFESSIONNEL (DP)

Annexe 9 : TodoList.JSP et l'IHM correspondant **Todo WebApp**

[illegible]

Liste des Todos

Bonjour mathieu romatet !

Se déconnecter

Liste des Todos

Recherche par mots clés :

Tapez votre mot ici

Rechercher

Ajouter Todo

Titre	Date d'échéance	Todo Status	Actions
java	20 août 2022	En cours	Editer Effacer
python	9 mai 2022	En cours	Editer Effacer
java	25 mars 2022	Terminée	Editer Effacer
javascript 2	22 oct. 2022	Terminée	Editer Effacer

DOSSIER PROFESSIONNEL (DP)

Annexe 10 : Dictionnaire des données InfoPlus

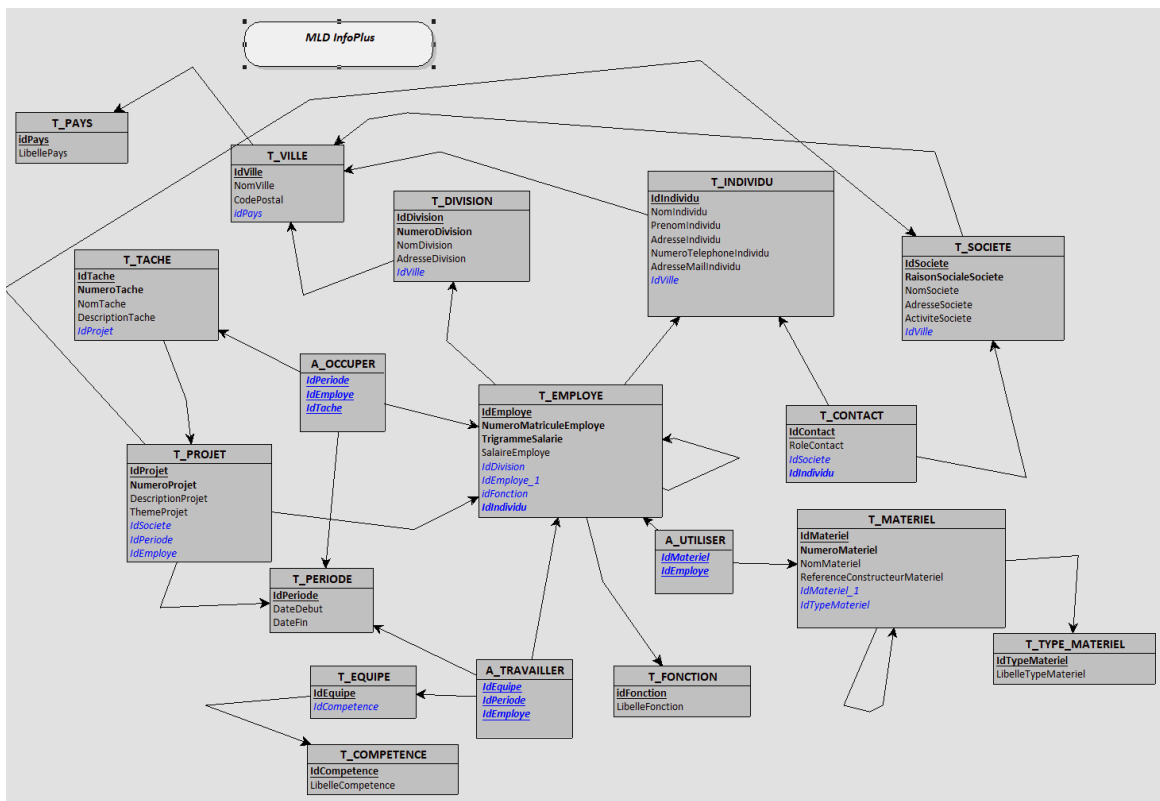
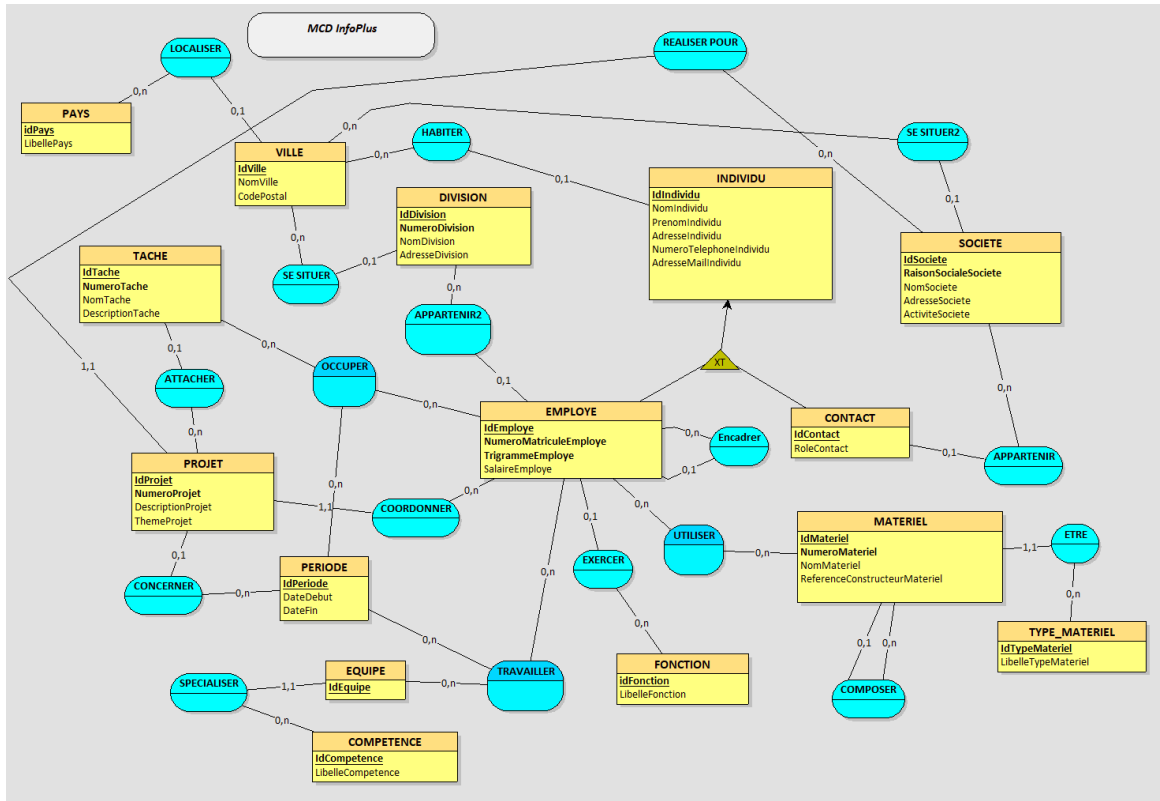
NOM	TYPE	null/non null	Taille	IDENTIFIANT	MODE D'OBTENTION	REGLE DE CALCUL
CodePostal	ALPHA	non Null	20		Mémorisé	
Division						
IdDivision	NUM	non Null	5	X	Mémorisé	
NumeroDivision	NUM	non Null	30		Mémorisé	
NomDivision	ALPHA	non Null	50		Mémorisé	
AdresseDivision	ALPHANUM	Null	50		Mémorisé	
Individu						
IdIndividu	NUM	non Null	5	X	Mémorisé	
NomIndividu	ALPHA	non Null	50		Mémorisé	
PrenomIndividu	ALPHA	non Null	50		Mémorisé	
AdresseIndividu	ALPHANUM	Null	150		Mémorisé	
NumeroTelephoneIndividu	NUM	Null	50		Mémorisé	
AdresseMailIndividu	ALPHANUM	Null	100		Mémorisé	
Equipe						
IdEquipe	NUM	non Null	5	X	Mémorisé	
Materiel						
IdMateriel	NUM	non Null	5	X	Mémorisé	
NumeroMateriel	ALPHANUM	non Null	50		Mémorisé	
NomMateriel	ALPHANUM	Null	100		Mémorisé	
ReferenceConstructeurMateriel	ALPHANUM	Null	50		Mémorisé	
Employe						
IdEmploye	NUM	non Null	5	X	Mémorisé	
NumeroMatriculeEmploye	ALPHANUM	non Null	30		Mémorisé	
TrigrammeSalarie	ALPHA	Null	3		Calculé	RC1
SalaireEmploye	MONEY	Null	5		Mémorisé	
Societe						
IdSociete	NUM	non Null	5	X	Mémorisé	
RaisonSocialeSociete	ALPHANUM	non Null	50		Mémorisé	
NomSociete	ALPHANUM	Null	50		Mémorisé	
AdresseSociete	ALPHANUM	Null	100		Mémorisé	
ActiviteSociete	ALPHANUM	Null	50		Mémorisé	
Contact						
IdContact	NUM	non Null	5	X	Mémorisé	
RoleContact	ALPHANUM	Null	100		Mémorisé	
Projet						
IdProjet	NUM	non Null	5	X	Mémorisé	
NumeroProjet	NUM	non Null	50		Mémorisé	
DescriptionProjet	ALPHANUM	Null	400		Mémorisé	
ThemeProjet	ALPHANUM	Null	50		Mémorisé	
Tache						
IdTache	NUM	non Null	5	X	Mémorisé	
NumeroTache	ALPHANUM	non Null	15		Mémorisé	
NomTache	ALPHANUM	non Null	100		Mémorisé	
DescriptionTache	ALPHANUM	Null	50		Mémorisé	

REGLES DE CALCUL	
NUMERO	REGLE
RC01	Il est composé de la 1ère lettre du prénom, la 1ère lettre du nom et la 2ème lettre du nom. Si le trigramme existe déjà, vous devez prendre la lettre suivante du nom.

REGLES DE GESTION	
NUMERO	REGLE
RG01	Un employé peut exercer 0 ou 1 fonction
RG02	Une fonction peut être exercée par 0 ou plusieurs employés
RG03	Un employé peut appartenir à 0 ou 1 division
RG04	Une division peut accueillir 0 à plusieurs employés
RG05	Un employé s'occupe de 0 à plusieurs taches pendant une période donnée
RG06	Une tache peut être faite par 0 ou plusieurs employés pendant une période donnée
RG07	Un employé peut coordonné 0 ou plusieurs projet
RG08	Un projet est coordonné par 1 et 1 seul employé
RG09	Un employé peut travailler dans 0 ou plusieurs équipes dans une période donnée
RG10	Une équipe peut contenir 0 ou plusieurs employés dans une période donnée
RG11	Un employé utilise 0 à plusieurs matériels
RG12	Un matériel peut être utilisé par 0 ou plusieurs employés
RG13	Un employé est encadré par 0 ou 1 employé
RG14	Un employé encadre 0 à plusieurs employés
RG15	Une division est située dans une ville
RG16	Une ville peut accueillir 0 à plusieurs divisions
RG17	Une Ville est localisé dans 0 ou un pays
RG18	Un pays possède 0 ou plusieurs villes
RG19	Un projet contient de 0 à plusieurs taches
RG20	Une tache est attachée à 0 ou 1 projet
RG21	Un projet concerne 0 ou 1 période
RG22	Une période peut concerner 0 à plusieurs projets
RG23	Une équipe a 1 et 1 seule spécialité
RG24	Une spécialité peut correspondre de 0 à plusieurs équipes
RG25	un individu correspond à 0 ou 1 employé
RG26	Un employé correspond à 1 et 1 seul individu
RG27	Un Individu correspond à 0 ou 1 seul contact
RG28	Un contact correspond à 1 et 1 seul individu
RG29	Un contact appartient à 0 ou 1 société
RG30	Une société a de 0 à plusieurs contacts
RG31	Une société se situe dans 0 ou 1 ville
RG32	Une ville peut accueille de 0 à plusieurs sociétés
RG33	Un projet est réalisé pour 1 socété et 1 seule
RG34	Une société peut être liée de 0 à plusieurs projets
RG35	Un matériel se compose de 0 à plusieurs matériels
RG36	Un matériel est le composant de 0 à 1 matériel
RG37	Un matériel est d'1 et d'un sel type de matériel
RG38	Un type de matériel correspond à 0 ou plusieurs materiels
RG39	Un individu habite dans 0 ou 1 ville
RG40	1 ville est habité de 0 à plusieurs individus

DOSSIER PROFESSIONNEL (DP)

Annexe 11 : MCD et MLD de la base de données InfoPlus



Annexe 12 : Classe de service pour l'application E-commerce

```
@Service
public class CheckoutServiceImpl implements CheckoutService {

    private CustomerRepository customerRepository;

    @Autowired //optional since we have only one constructor
    public CheckoutServiceImpl(CustomerRepository customerRepository) {
        this.customerRepository = customerRepository;
    }

    @Override
    @Transactional
    public PurchaseResponse placeOrder(Purchase purchase) {

        // retrieve the order info from dto
        Order order = purchase.getOrder();

        // generate tracking number
        String orderTrackingNumber = generateOrderTrackingNumber();
        order.setOrderTrackingNumber(orderTrackingNumber);

        // populate order with orderItems
        Set<OrderItem> orderItems = purchase.getOrderItems();
        orderItems.forEach(item -> order.add(item));

        // populate order with billingAddress and shippingAddress
        order.setBillingAddress(purchase.getBillingAddress());
        order.setShippingAddress(purchase.getShippingAddress());

        // populate customer with order
        Customer customer = purchase.getCustomer();

        // check if this is an existing customer
        String theEmail = customer.getEmail();

        Customer customerFromDB = customerRepository.findByEmail(theEmail);

        if (customerFromDB != null) {
            // we found them ... let's assign them accordingly
            customer = customerFromDB;
        }
        customer.add(order);

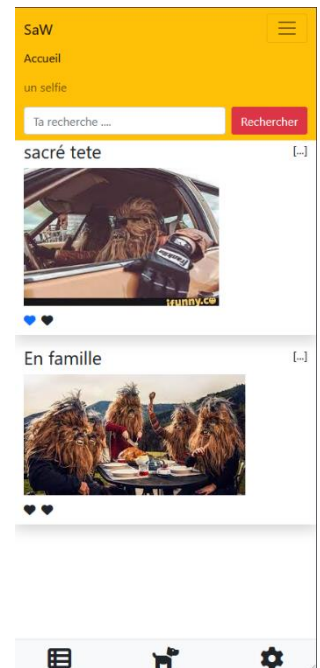
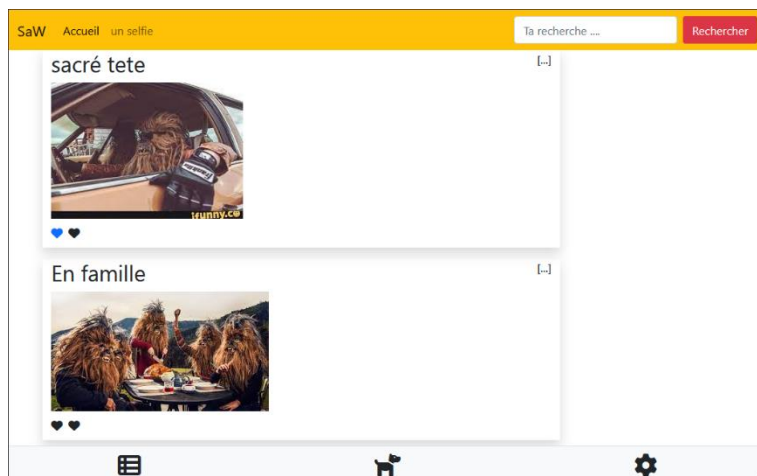
        // save to the database
        customerRepository.save(customer);

        // return a response
        return new PurchaseResponse(orderTrackingNumber);
    }

    private String generateOrderTrackingNumber() {
        // generate a random UUID number (UUID version-4 for random) Universally Unique Identifier
        return UUID.randomUUID().toString();
    }
}
```

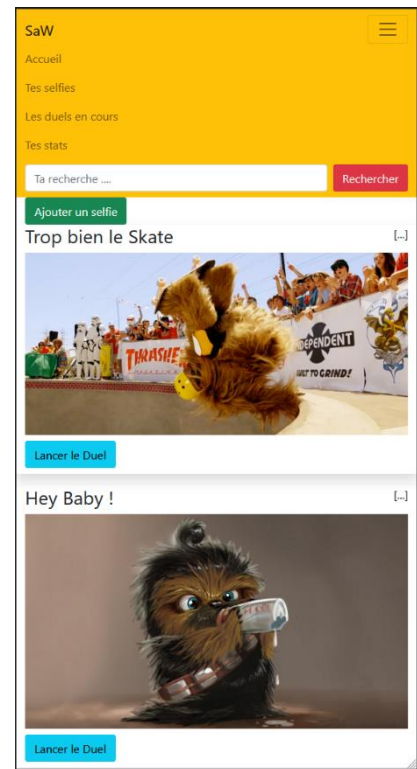

Annexe 13 : Maquette statique responsive *WookieApp*

```
<header>      You, 4 months ago • Site Bootstrap ok
<!-- NavBar -->
<nav class="navbar navbar-expand-lg navbar-light bg-warning">
  <div class="container-fluid">
    <a class="navbar-brand" href="#">SaW</a>
    <button class="navbar-toggler" type="button" data-bs-toggle="collapse"
      data-bs-target="#navbarSupportedContent" aria-controls="navbarSupportedContent"
      aria-expanded="false" aria-label="Toggle navigation">
      <span class="navbar-toggler-icon"></span>
    </button>
    <div class="collapse navbar-collapse" id="navbarSupportedContent">
      <ul class="navbar-nav me-auto mb-2 mb-lg-0">
        <li class="nav-item">
          <a class="nav-link active" aria-current="page" href="#">Accueil</a>
        </li>
        <li class="nav-item">
          <a class="nav-link" href="#">un selfie</a>
        </li>
      </ul>
      <form class="d-flex">
        <input class="form-control me-2" type="search" placeholder="Ta recherche..."
          aria-label="Search">
        <button class="btn btn-danger" type="submit">Rechercher</button>
      </form>
    </div>
  </div>
</nav>
</header>
```



DOSSIER PROFESSIONNEL (DP)

Annexe 14 : IHM Liste des selfies WookieApp



Annexe 15: Mock server Postman WookieApp

New Collection

- wookieApp Post/Get
 - POST selfie
 - Default
 - GET selfie
 - Default
 - GET selfie/2
 - Default

GET `{{url}}/selfie`

Params Headers Body

Query Params

KEY	VALUE
Key	Value

Body Headers

Pretty Raw Preview JSON

```
1 {  
2   {  
3     "image": "https://www.theriderpost.com/wp-content/uploads/2016/06/896e3e36d885874856df151425f1ce6484412952-1.jpeg",  
4     "titre": "Trop bien le Skate",  
5     "wookie": {  
6       "nom": "Chewie",  
7       "selfies": []  
8     }  
9   },  
10  {  
11    "image": "http://www.groomliday.com/wp-content/uploads/2016/07/Wookie.jpg",  
12    "titre": "Hey Baby !",  
13    "wookie": {  
14      "nom": "Turloff",  
15      "selfies": []  
16    }  
17  }  
18 }
```