|  |  |  |
| --- | --- | --- |
|  |  |  |
| *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 Developpeur d’Applications | |
|  | |
| **Modalité d’accès :** | |
|  | |
|  | 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**](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. | p. | 5 |
|  |  |  |  |
| **Concevoir et développer la persistance des données en intégrant les recommandations de sécurité** | | **p.** |  |
|  |  Projet InfoPlus p. | p. | 13 |
|  |  |  |  |
| **Concevoir et développer une application multicouche répartie en intégrant les recommandations de sécurité** | | **p.** |  |
|  |  E-commerce Backend p. | p. | 17 |
|  |  WookieApp p. | p. | 20 |
|  |  |
|  |  |
| **Titres, diplômes, CQP, attestations de formation** *(facultatif)* | | **p.** |  |
| **Déclaration sur l’honneur** | | **p.** |  |
| **Documents illustrant la pratique professionnelle** *(facultatif)* | | **p.** |  |
| **Annexes** *(Si le RC le prévoit)* | | **p.** |  |

**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, il nous a été demandé de développer une application web **Java EE** pouvant fournir aux utilisateurs un service de gestion de liste de taches à effectuer (Todo). 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 mot clés pour afficher les Todo. Les données doivent être persistantes.  Ce projet devait être réalisée en groupe et nous avons décidé de fonctionner en méthode **Agile Scrum**, la formatrice tenant le rôle de Product Owner. Ça 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 version.  **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 faite, nous avons réalisé des **maquettes IHM** (Annexe 6) et les avons fait valider par le PO.  **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*. On a construit l’application en plusieurs couches et les design pattern **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. Elle n’effectue aucun traitement et est une page écrite en HTML et JAVA. * **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 Modele** (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.      1. **Développement des composants d’accès aux données dans la couche DAL**   [Java Persistence API](https://www.baeldung.com/learn-jpa-hibernate) (**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.  [**@Entity**](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/Entity.html) Définit qu’une classe est une entité.  [**@Id**](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/Id.html) Définit l’attribut qui sert de clé primaire dans la table.  [**@Table**](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/Table.html) Permet de définir les informations sur la table représentant cette entité en base de données.  [**@GeneratedValue**](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/GeneratedValue.html) Indique la stratégie à appliquer pour la génération de la clé lors de l’insertion d’une entité en base.  [**@Column**](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/Column.html) Permet de déclarer des informations relatives à la colonne sur laquelle un attribut doit être mappé  À l’entité *UserEntity* ci-dessus, on pourra faire correspondre la table PostgreSQL ***utilisateur*** :  JPA est une spécification et pour pouvoir y accéder, nous utilisons une implémentation compatible avec JPA qui est [**Hibernate**](http://hibernate.org/). Le point d'entrée du code permettant de manipuler des entités JPA est un objet de type EntityManagerFactory qui est utilisé dans la classe *DAOUitil* 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 donnée, ..)  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.  Ci-dessous la classe *GenericDAOJPAImpl* qui implémente les méthodes de création, lecture, modification et destruction (CRUD) dans la BDD en utilisant un l’EntityManager.  Puis on a créé des classes qui étendent cette classe générique et qui implémentent les méthodes propres aux tables   1. **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.  On a créé la classe FrontController unique qui implémente l’interface HttpServlet définissant une servlet utilisant le protocole http.    Dans cette classe Controller, on fait appel à 2 méthodes qui vont nous permettre de redistribuer dynamiquement vers les JSP  ou les classes *Command* qui implémentent l’interface ICommand.      Méthode getCorrespongingClass*()* qui récupère le paramètre "action" de la requête http, et charge la classe *Command* correspondante.      Méthode *doProcess()* qui grâce à la méthode au-dessus instancie la classe *Command* correspondante, appelle la méthode *execute()* de cette instance et récupère le retour de cette méthode pour soit rediriger vers une classe *Command*, soit vers une **JSP** (partie Vue)  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 pages web dynamiquement.  **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 objet **Entity** (pour interagir avec la BDD).      Ces classes service implémentent des interfaces déclarant les méthodes répondant aux exigences du métier.  Ci-dessous la classe UserServiceImpl utilisé dans la classe CommandLogin vue précédemment et qui implémentent la méthode nécessaire pour renvoyer le UserDTO correspondant au login et mot de passe contenu dans le LoginDTO passé en paramètre.    Pour cela il va instancier un objet userDAO et va faire appel à sa méthode GetUserByLoginByUser() pour récupérer un objet Entity.  Il va ensuite utiliser un mapper générique sur l’Entity pour retourner le UserDTO. | | |

|  |  |
| --- | --- |
| 1. **Développement de la partie frontend de l’interface utilisateur web**   Le **JavaServer Pages** ou **JSP** est une [technique](https://fr.wikipedia.org/wiki/Technique) basée sur [Java](https://fr.wikipedia.org/wiki/Java_(langage)) qui nous a permis de créer dynamiquement du code [HTML](https://fr.wikipedia.org/wiki/Hypertext_markup_language) en incluant des variable Java en les plaçant entre les marqueurs <%= et %>. On a aussi utilisé **Bootstrap** qui est une [collection d'outils](https://fr.wikipedia.org/wiki/Framework) utiles à la création du design et qui qui contient des codes [HTML](https://fr.wikipedia.org/wiki/HTML) et [CSS](https://fr.wikipedia.org/wiki/Cascading_Style_Sheet), des formulaires, boutons et autres éléments interactifs.  On a utilisé aussi un [langage de script](https://fr.wikipedia.org/wiki/Langage_de_script) particulier, appelé [**Expression Language**](https://fr.wikipedia.org/wiki/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**](https://fr.wikipedia.org/wiki/JavaServer_Pages_Standard_Tag_Library). (Ex : **annexe 9**)  Voici la page Login.jsp qui est la première page de notre application (défini dans le web.xml) et vers lequel peut aussi renvoyer la classe CommandLogin présentée plus haut si la combinaison login/mot est absente de la BDD.    Ce **Jsp** va par l’intermédiaire d’un formulaire, récupérer le login et le mot de passe saisie 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 .    Et voici l’IHM correspondant   1. **Les tests**   L’application a été développé en essayant d’implémenter des test au fur et à mesure pour valider les méthodes utilisées. Pour cela on a utilisé **JUnit** pour l’exécution de tests unitaires automatisés.  Ci-dessous le test de la méthode GetUserByLogin() du service UserServiceImpl . | |
|  |
|  | |
| **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 :** | | | | | |
|  | | | | | |
| Le projet InfoPlus consistait à concevoir et développer 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 on pouvait interroger le Maitre d’ouvrage si besoin. Pour cela j’ai utilisé la méthode **MERISE** pour arriver à 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 maitre d’ouvrage si besoin de précision 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 exécuté en m’aidant de l’outil **pgAdmin**.    1 - Créer un utilisateur et la base de données avec **Postgres psql** | | | | | |
| 2 - Script de création des tables et contraintes sur pgAmin :  Ici on peut voir la création des tables T\_INDIVIDU et T\_EMPLOYE en spécifiant leurs types, longueurs, clés primaires et contraintes.    3 - Script d’insertion des données, ici un extrait pour la table T\_EMPLOYE :    4 - Puis j’ai élaboré des **procédures ou fonctions stockées**, **triggers** et **vues** pour répondre aux besoins exprimés :    Exemple Fonction Stockée :    Exemple trigger :  Exemple Vue : | | | | | |
|  | | | | |
| 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 :   * PostgresSQL 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. On a 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. On a 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 étant les objets java contenant les données, la vue pour afficher les informations (assuré par le front Angular) et les contrôleurs assurant la communication en les parties Backend et Frontend.  L’application a été **divisé 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.      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.  [**@Entity**](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/Entity.html) Définit qu’une classe est une entité.  [**@Id**](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/Id.html) Définit l’attribut qui sert de clé primaire dans la table.  [**@Table**](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/Table.html) Permet de définir les informations sur la table représentant cette entité en base de données.  [**@GeneratedValue**](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/GeneratedValue.html) Indique la stratégie à appliquer pour la génération de la clé lors de l’insertion d’une entité en base.  [**@Column**](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/Column.html) Permet de déclarer des informations relatives à la colonne sur laquelle un attribut doit être mappé  [**@ManyToOne**](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/ManyToOne.html) définit une relation n:1 entre deux entités.  [**@JoinColumn**](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/JoinColumn.html) 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 :    La couche service est là où on a développé des composants en implémentant la logique métier (générer un nombre unique pour chaque commande d’achat, calculer le prix total, …). Ces composants utilisent en autre les méthodes d’accès à la BDD des Repository (Ex : Annexe 12)  Ici une méthode pour générer un nombre unique qui est ensuite rattacher à une commande d’achat :      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 à travers des requêtes **http**.  Dans l’exemple ci-dessous, à l’Endpoint « /api/checkout/purchase », la méthode PurchaseReponse() récupéré le corps de la requête grâce à l’annotation **@RequestBody**. Puis il le désérialiser en un DTO (Purchase) et le place en paramètre de la méthode placeOder() du service checkoutService pour renvoyer le résultat. | | | |
|  | | |
|  | | | |
| **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 :** | | | | | | |  | | | | | | | Dans le cadre privé, j’ai décidé de m’entrainer avec **Angular** en développant une interface utilisateur web **multicouche** et **responsive** consommant des données au format **json** envoyés par une l’API.  L’objectif est une sorte d’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.   **1er étape : Analyse**  J’ai tout d’abord établi mon cahier des charges et déterminer quelles seraient les données pertinentes à utiliser et afficher.  **2em 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 soit aussi utilisé comme une **application mobile** : c’est-à-dire dont le code répond à des normes lui permettant d'être adapté à tous types de supports et y compris aux écrans des mobiles**.**  Il y a plusieurs façons de 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, …)  On met une balise <link /> qui permet, dans notre code HTML, de charger un fichier .css et on lui ajoute un attribut media, dans lequel on va écrire la règle qui doit s'appliquer pour que le fichier soit chargé.    On met ensuite les règles à appliquer directement dans la feuille de style.    Ensuite il y a **Bootstrap** que j’ai choisi d’utiliser pour ce projet et qui est une collection d'outils utiles à la création du design de sites et d'applications web. C'est un ensemble qui contient des codes HTML et CSS, des formulaires, boutons, outils de navigation et autres éléments interactifs.  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**)  **3em étapes : Développement**  J’ai commencé le développement en y allant couche par couche en commençant par le service des selfies qui sert en autre à propager les données qui seront implémentés dans le DOM via les Components.  **Pour cela j’ai créé une méthode getAllSelfies\_asObservable() qui récupère le flux de données Observables qui vous permettra de propager et consommer en temps réel les données dans les différents composants de votre application**    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      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 structurelle **ngFor** permet de boucler sur un tableau et d'injecter les éléments dans le DOM :  Dans le fichier html du composant selfie-list :    Dans le fichier Typescript du composant selfie-readonly :    Voici la page html du composant selfie-readonly affiché : **(IHM annexe 14)**    **4em étape : tester**  Pour **tester** cette interface j’ai simulé grâce à **Postman** des endpoints (mocking) qui reçoivent et envoient des requêtes REST (**annexe 15**)  **6em étape : déploiement**  Pour construire votre 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 votre application.  Puis 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 avoir à 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*](https://github.com/MaTromatet)*/deployed-wookieApp.git --name= '*[*MaTromatet*](mailto:MaTromatet)*'* [*--email=mromatet@gmail.com*](mailto:--email=mromatet@gmail.com) *--base-href=/deployed-wookieApp/"*  *},*  Puis j’ai lancé la commande « ng deploy » | | | | | | |  | | | | | | |  | | | | | |  | | | | | | | **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** |
|  |
|  |

Cliquez ici pour taper du texte.

Je soussigné(e) [prénom et nom] ,

déclare sur l’honneur que les renseignements fournis dans ce dossier sont exacts et que je suis l’auteur(e) des réalisations jointes.

Cliquez ici pour taper du texte.

Cliquez ici pour choisir une date

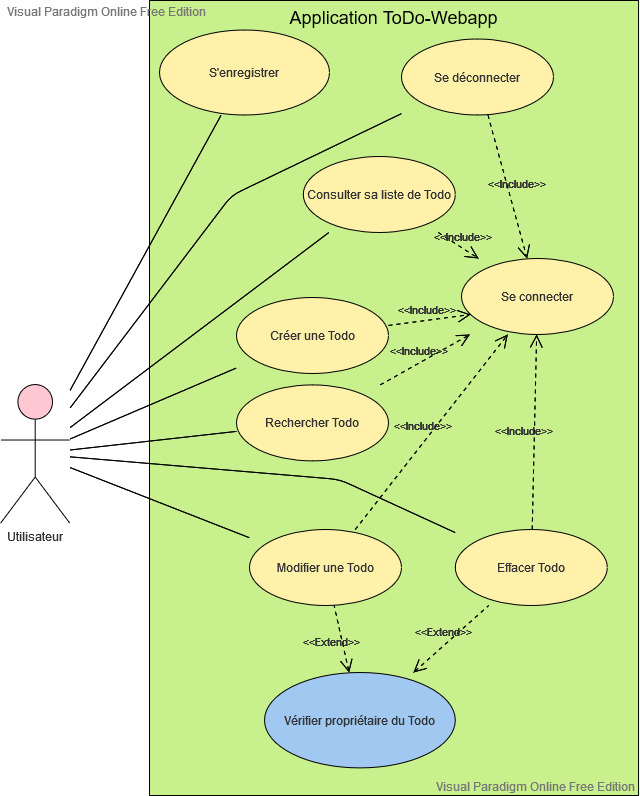
Fait à le

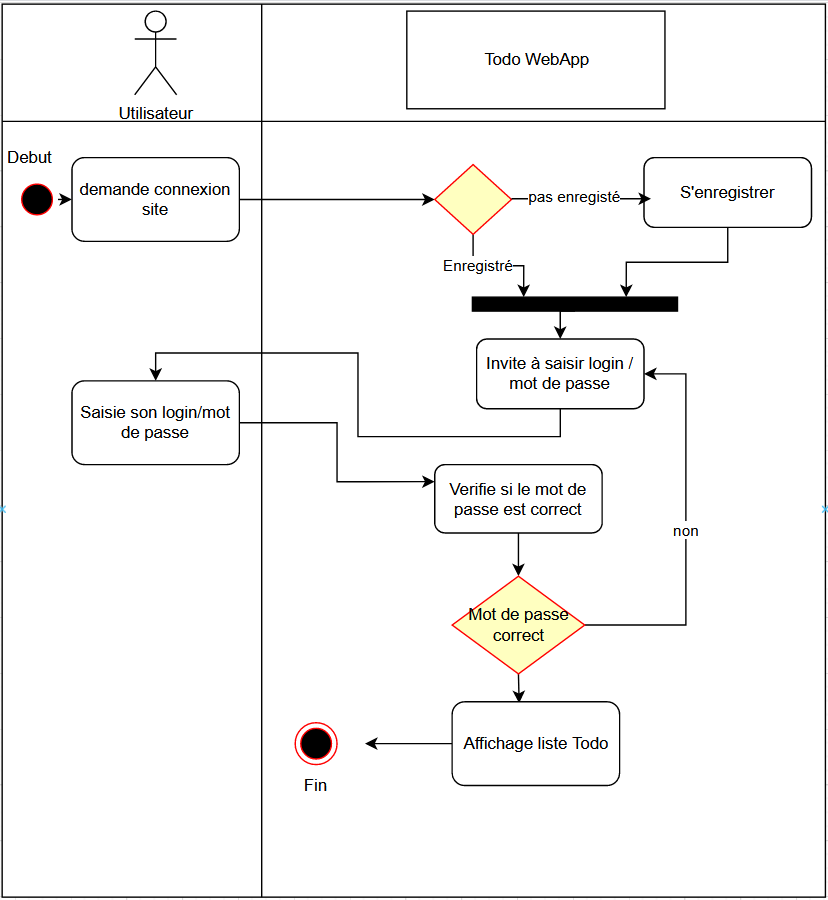
pour faire valoir ce que de droit.

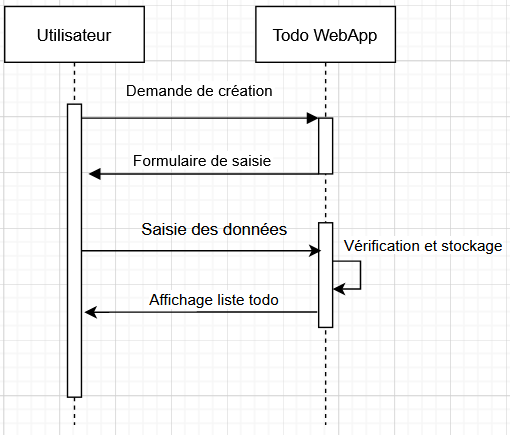
Signature :

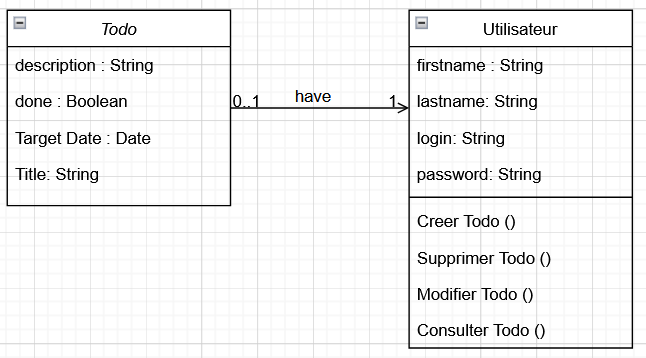
|  |
| --- |
| **Annexes** |
|  |

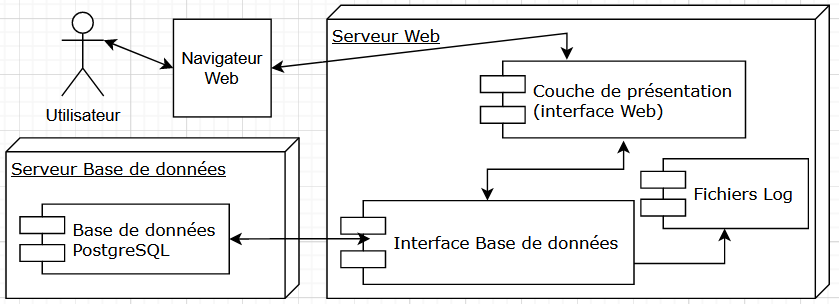
*Annexe 1 : Diagramme des Cas d’utilisation (use case) : Todo WebApp*

**

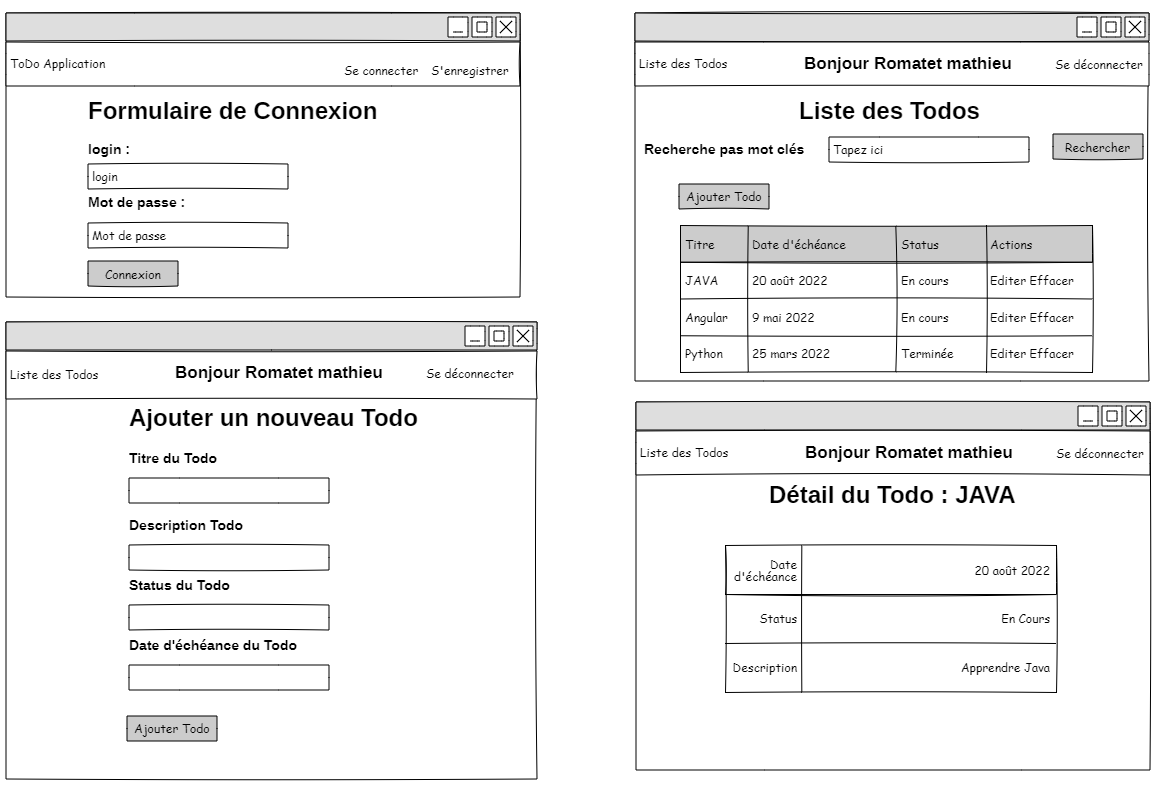
*Annexe 2 : Diagramme d’Activité : Connexion* ***Todo WebApp***

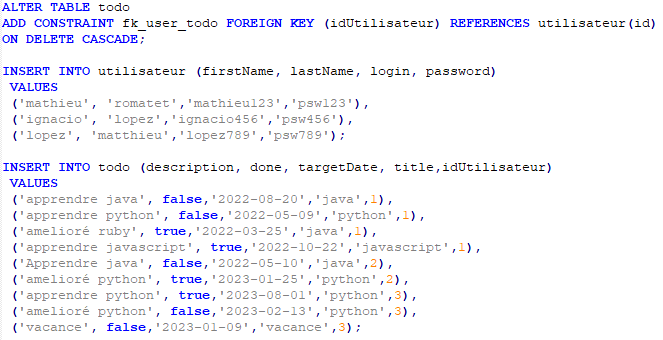
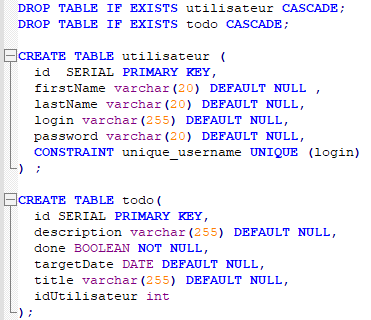
*Annexe 3 : Diagramme de séquence : création d’une Todo :* ***Todo WebApp***

*Annexe 4 : Diagramme de Classe :* ***Todo WebApp***

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

*Annexe 6 : Maquettes* ***Todo WebApp***

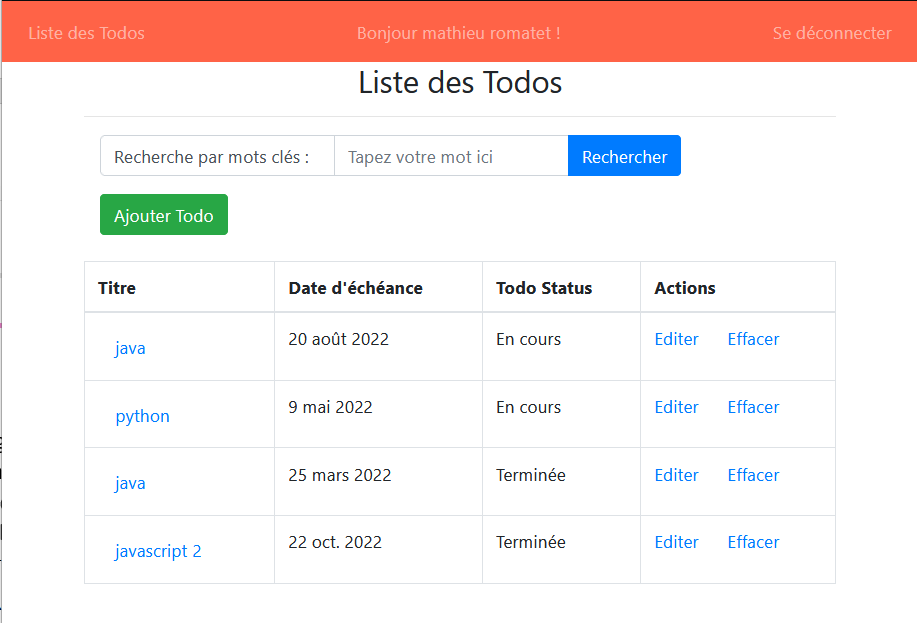


*Annexe 7 : Script création Base de données* ***Todo WebApp***

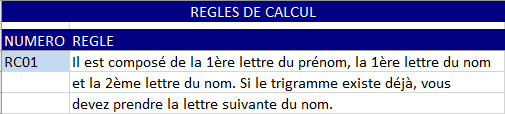
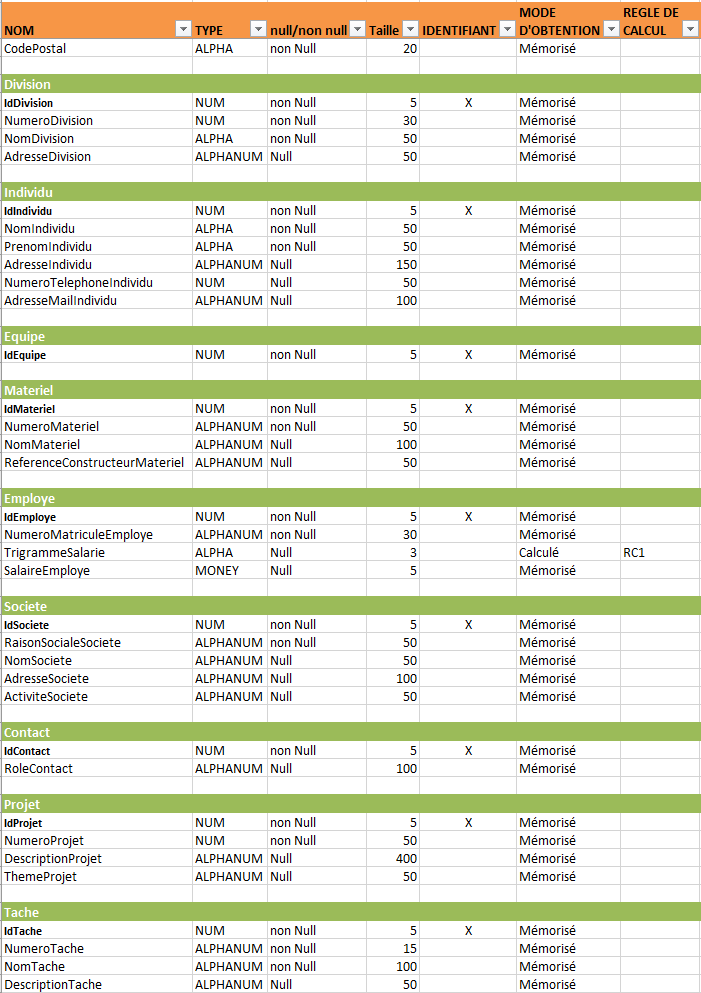
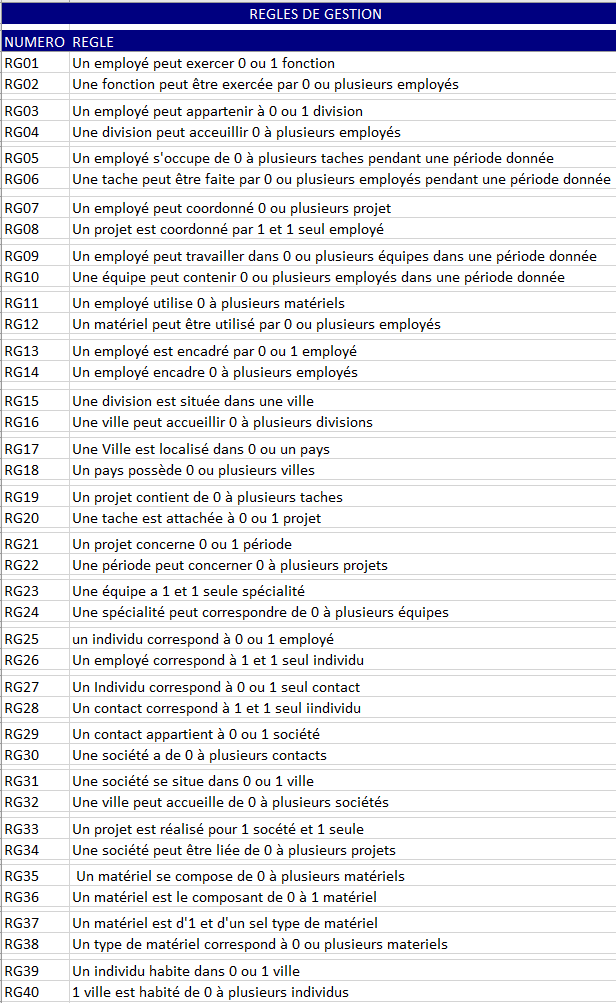
*Annexe 8 :* *description* ***persistence.xml*** *pour connexion JPA* ***Todo WebApp***



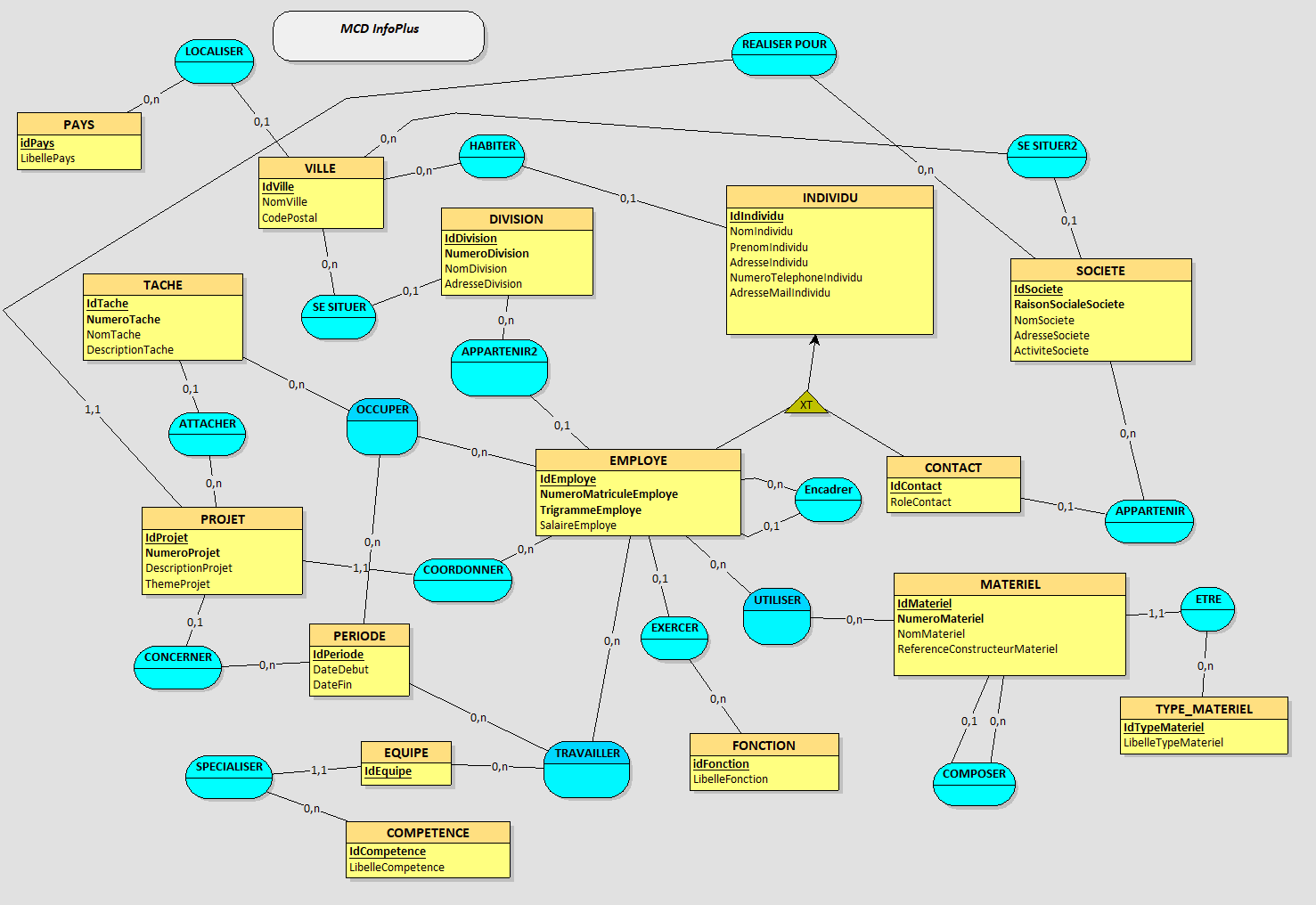
*Annexe 9 : TodoList.JSP et l’IHM correspondant* ***Todo WebApp***

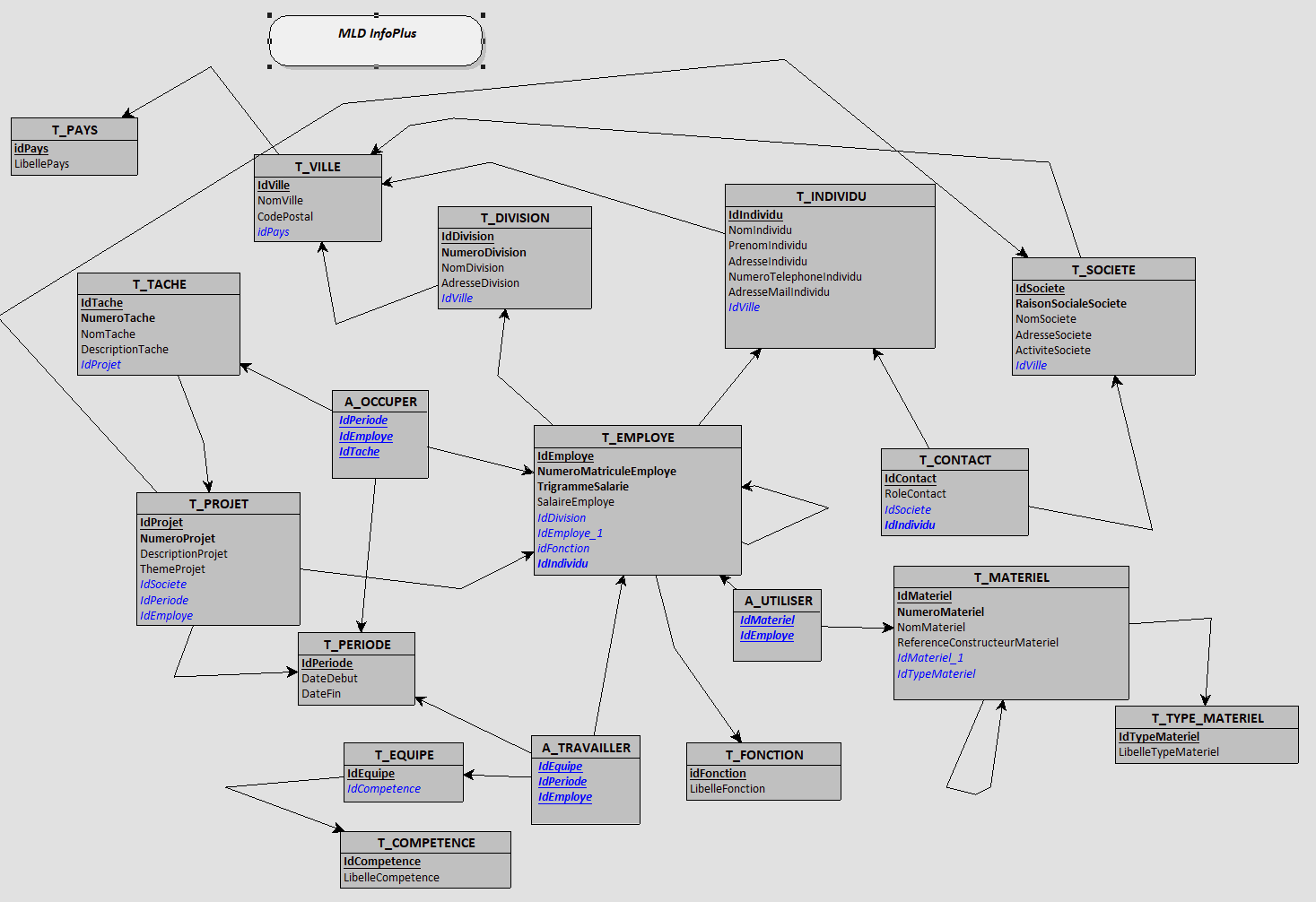


*Annexe 10 : Dictionnaire des données* ***InfoPlus***

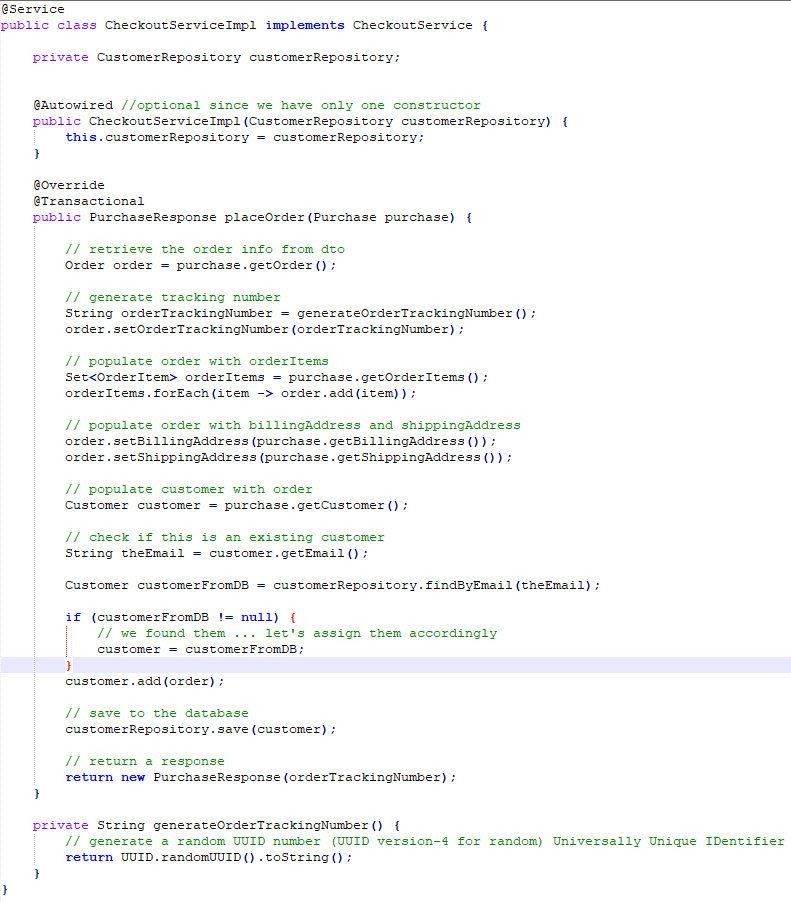
****

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

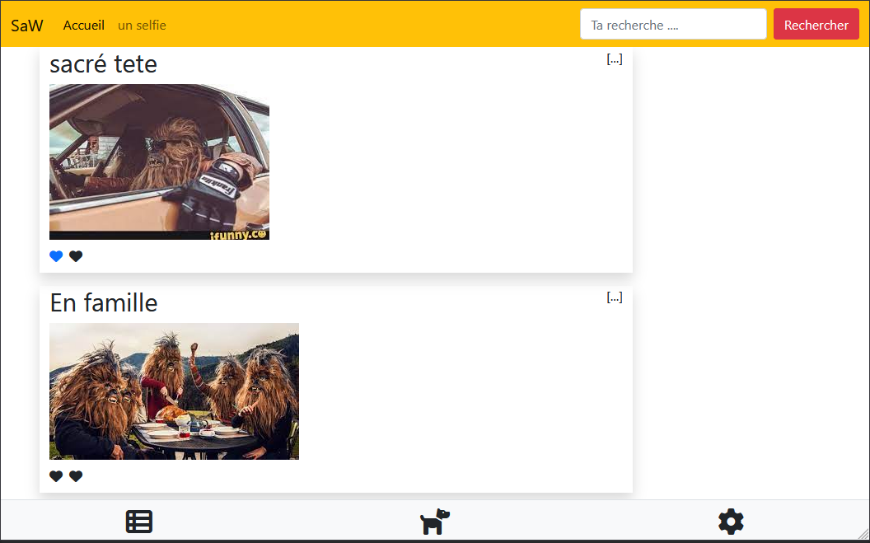
**

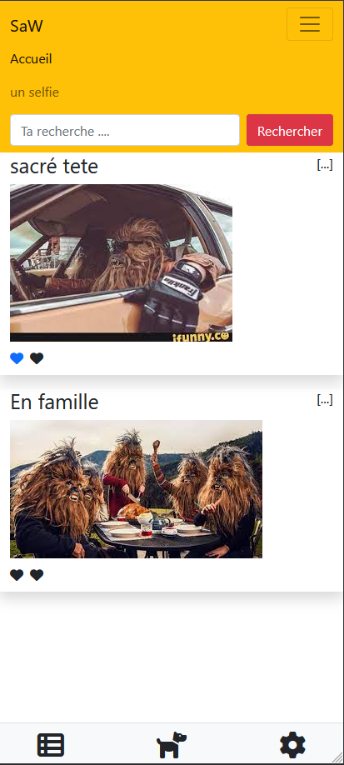


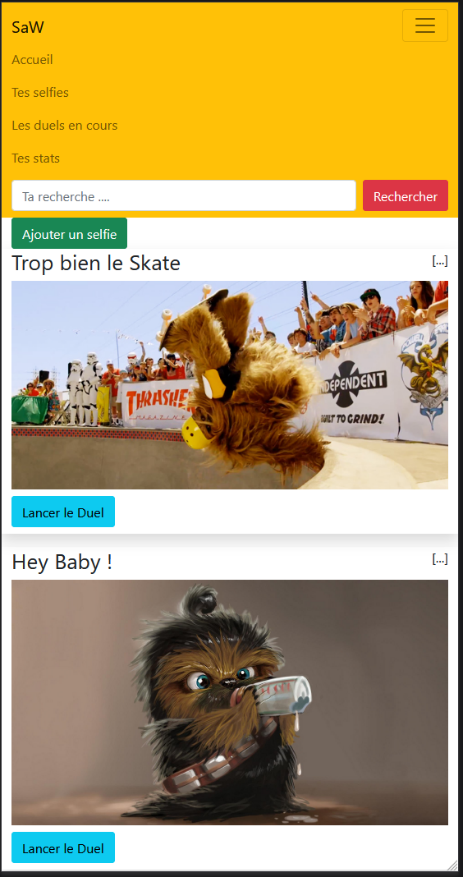
*Annexe 12 : classe de service pour l’application* ***E-commerce***

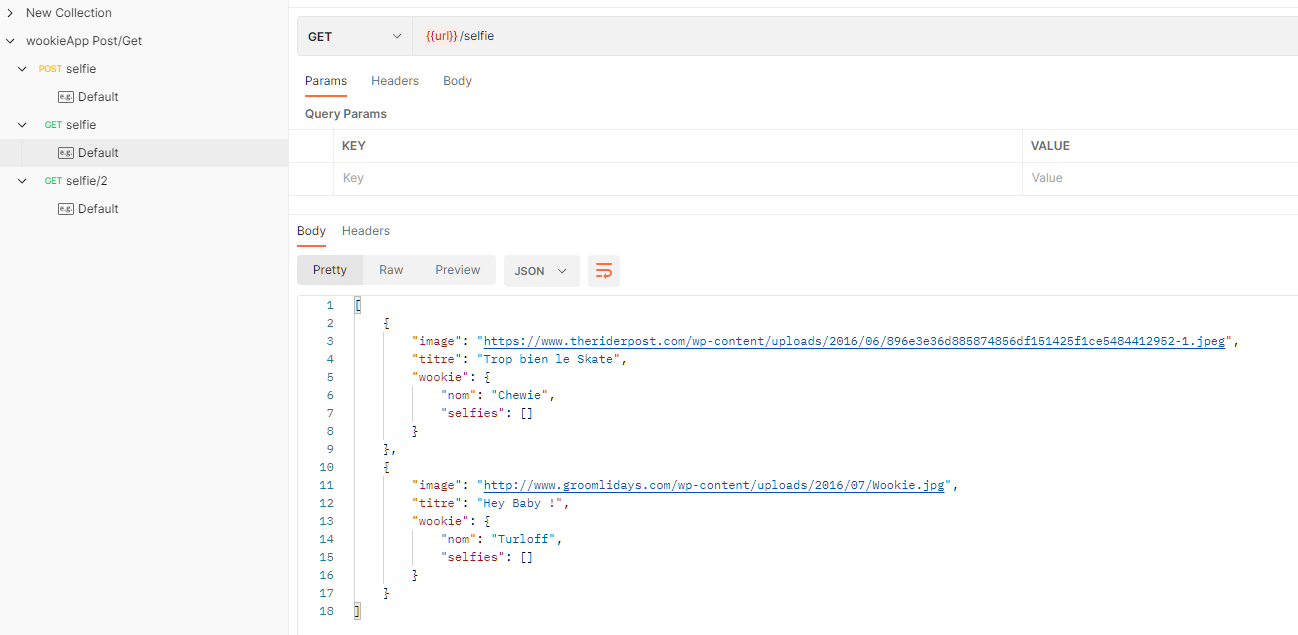


*Annexe 13 : Maquette statique responsive* ***WookieApp***





*Annexe 14 : IHM Liste des selfies* ***WookieApp***

*Annexe 15: Mock server Postman* ***WookieApp***