Dossier de Projet

# I. Liste des compétences couvertes par ce dossier.

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

\*Maquetter une application

\*Développer des composants d’accès aux données

\*Développer la partie back-end d’une interface utilisateur web

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

\*Concevoir une base de données

\*Mettre en place une base de données

\*Développer des composants dans le langage d’une base de données

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

\*Concevoir une application

\*Développer des composants métier

\*Construire une application organisée en couches

\*Développer une application mobile

\*Préparer et exécuter les plans de tests d’une application

# II.Resume

This project follows a previous one where my employer received an application used to help him manage is sellout. Due to limitations in the previous project, mainly the modularity of the accessible data, he wanted to expand the fonctionnality of the application to make it more versatile. My goal was to add fonctionnality to the previous project, add a lot of modularity to it and increase the overall security of it regarding to RGPD, excpecially its API security through the addition of CSRF token. All of this must take the form of a mobile application with a distant database accessible through the previously mentionned API. I was alone on this project so all the conception , from the database to the app maquette, as to be done by me. The whole project was supervised by Loic Creusot the store owner and Christophe Vaucourt its informatical referent and was done mainly in my home when i was remote-working in a almost complete autonomy.

# III. Cahier des charges

1. Introduction

L'objectif de ce projet est d'améliorer une application Android existante pour rendre l'expérience utilisateur plus efficace. L'application actuelle a été développée avec MIT App Inventor. Les fonctionnalités existantes comprennent l'enregistrement des coordonnées des clients, la sélection de produits par catégories, la gestion des quantités, la sélection de la date de commande, l'ajout de produits au panier, le calcul du total du panier, la génération d'un récapitulatif de commande avec un numéro unique, l'impression des bons de commande sur une imprimante thermique Bluetooth, la génération de rapports de commande sur une page web et l'envoi de mails de confirmation de commande. Ces fonctionnalités existantes doivent être améliorées pour rendre l'expérience utilisateur plus fluide et conviviale.

2. Améliorations souhaitées

Les améliorations suivantes doivent être apportées à l'application existante :

2.1 Interface utilisateur conviviale

- Rendre l'interface utilisateur plus intuitive, attrayante et conviviale en améliorant la disposition des éléments, les icônes, les couleurs et les animations.

2.2 Optimisation des performances

- Optimiser les performances de l'application pour garantir une expérience utilisateur fluide et réactive. Cela inclut la réduction des temps de chargement, l'optimisation de l'utilisation des ressources système et la gestion efficace de la mémoire.

2.3 Amélioration de la gestion des coordonnées clients

- Permettre aux utilisateurs de modifier facilement leurs coordonnées enregistrées, ainsi que la possibilité de supprimer leur compte si nécessaire.

2.4 Amélioration de la sélection de produits par catégories

- Améliorer la fonctionnalité de sélection des produits par catégories en rendant la navigation plus rapide, plus intuitive et offrant des options de recherche et de filtrage avancées.

2.5 Amélioration de la gestion des quantités

- Permettre aux utilisateurs de gérer plus efficacement les quantités de produits, en offrant des options pour modifier les quantités directement dans le panier, utiliser des boutons +/- pour ajuster les quantités, ou utiliser des gestes intuitifs.

2.6 Amélioration de la génération du récapitulatif de commande

- Améliorer la présentation et la lisibilité du récapitulatif de commande en y incluant toutes les informations pertinentes, telles que la date de commande, les coordonnées du client, le détail des produits commandés et le montant total.

2.7 Amélioration de l'impression des bons de commande

- Optimiser le processus d'impression des bons de commande sur l'imprimante thermique Bluetooth, en veillant à ce que l'interface soit conviviale, que l'impression soit fiable et de haute qualité, et en offrant des options de personnalisation du format et du contenu des bons de commande.

2.8 Amélioration de la génération des rapports de commande

- Améliorer la génération des rapports de commande sur une page web en utilisant des technologies web modernes, en ajoutant des fonctionnalités de recherche, de filtrage, de tri, de pagination et en améliorant la visualisation des données.

2.9 Amélioration de l'env

oi de mails de confirmation de commande

- Rendre l'envoi de mails de confirmation de commande plus fiable, rapide et personnalisable, en incluant toutes les informations pertinentes et en permettant une personnalisation avancée du contenu du mail.

3. Contraintes techniques

Le projet n'est pas limité à une technologie spécifique et les développeurs sont libres d'utiliser les langages de programmation, les Frameworks et les outils appropriés pour améliorer l'application Android existante.

4. Livrables attendus

Les livrables attendus pour ce projet sont les suivants :

- L'application Android améliorée, avec toutes les fonctionnalités et les améliorations spécifiées dans ce cahier des charges.

- Tout code source, fichiers de projet ou documentation pertinents.

5. Échéancier

L'échéancier du projet sera défini en concertation avec Loïc CREUSOT et l'équipe de développement, en tenant compte de la complexité des améliorations et des ressources disponibles.

# IV. Spécification fonctionnelle

1. Interface utilisateur améliorée

- L'interface utilisateur doit être intuitive, attrayante et facile à naviguer.

- Les fonctionnalités et les options doivent être clairement présentées et accessibles sans confusion pour les utilisateurs.

2. Performances optimisées

- L'application doit être optimisée pour garantir des performances rapides et réactives.

- Les temps de chargement doivent être réduits et les opérations courantes doivent être effectuées de manière instantanée.

3. Gestion des coordonnées clients

- Les utilisateurs doivent pouvoir modifier facilement leurs coordonnées enregistrées, telles que le numéro de téléphone, le nom et l'adresse e-mail.

- Une fonctionnalité de suppression de compte doit être disponible.

4. Sélection de produits par catégories améliorée

- La fonctionnalité de sélection de produits par catégories doit être améliorée pour faciliter la recherche et la sélection des produits.

- Des options de recherche et de filtrage doivent être fournies pour une meilleure expérience utilisateur.

5. Gestion des quantités optimisée

- Les utilisateurs doivent pouvoir modifier les quantités de produits dans le panier de manière facile et intuitive.

- Les modifications de quantités doivent être mises à jour en temps réel dans le panier et le montant total doit être recalculé automatiquement.

6. Génération du récapitulatif de commande améliorée

- Le récapitulatif de commande doit inclure toutes les informations pertinentes, telles que la date de commande, les coordonnées du client, le détail des produits commandés avec les quantités et les prix unitaires, ainsi que le montant total de la commande.

- Le récapitulatif doit être clair et facilement compréhensible pour les utilisateurs.

7. Impression des bons de commande optimisée

- L'interface d'impression des bons de commande sur l'imprimante thermique Bluetooth doit être conviviale et facile à utiliser.

- Les bons de commande imprimés doivent être bien formatés et inclure toutes les informations essentielles, telles que le numéro de commande, les coordonnées du client et le détail des produits commandés.

8. Génération des rapports de commande améliorée

- Les rapports de commande générés sur une page web doivent offrir des fonctionnalités avancées, telles que la recherche, le filtrage, le tri et la pagination des données.

- Les rapports doivent être clairs et faciles à interpréter, avec des graphiques ou des tableaux si approprié, pour faciliter l'analyse des données.

9. Envoi de mails de confirmation de commande amélioré

- L'envoi de mails de confirmation de commande doit être fiable, rapide et personnalisable.

- Les mails doivent inclure toutes les informations essentielles, telles que le numéro de commande, le détail des produits commandés, le montant total et les coordonnées du client.

- Les utilisateurs doivent avoir la possibilité de personnaliser le contenu du mail, en ajoutant des messages spécifiques ou des offres promotionnelles.

# V. Spécifications Technique

Spécifications techniques pour l'amélioration de l'application Android

1. Technologie utilisée

- Utilisation de Xamarin Forms comme framework de développement pour créer l'application Android, permettant ainsi une approche de développement multiplateforme.

- Utilisation de PostgreSQL comme système de gestion de base de données pour stocker les données de l'application.

- Utilisation de Docker pour la gestion des conteneurs, facilitant le déploiement et la portabilité de l'application.

- Utilisation de Visual Studio comme IDE principal pour le développement de l'application.

2. Maquettage et conception de l'interface utilisateur

- Utilisation de Figma comme outil de conception pour créer les maquettes et les wireframes de l'interface utilisateur de l'application Android.

- Les maquettes serviront de référence visuelle lors du développement de l'interface utilisateur, assurant une cohérence visuelle et une expérience utilisateur optimale.

3. Interface de paramétrage

- Développement d'une interface permettant aux utilisateurs de configurer les paramètres de l'application, tels que l'identité de l'entreprise, les informations de connexion au serveur de base de données, les identifiants, etc.

4. Interface de gestion de l'imprimante

- Développement d'une interface permettant la gestion de l'imprimante thermique Bluetooth, facilitant l'impression des bons de commande sur des supports physiques.

5. Interface de gestion du choix des cartes

- Développement d'une interface utilisateur intuitive pour faciliter la gestion des cartes disponibles dans l'application, offrant aux utilisateurs la possibilité de faire des choix de cartes en fonction de leurs préférences.

6. Amélioration du système d'ajout/suppression du panier

- Amélioration du système d'ajout et de suppression d'articles dans le panier pour offrir une expérience utilisateur plus fluide.

- Mise à jour en temps réel du panier et du montant total lors de l'ajout ou de la suppression d'articles.

7. Amélioration de la gestion des quantités

- Optimisation de la gestion des quantités de produits dans le panier, permettant aux utilisateurs de modifier facilement les quantités.

- Mise à jour automatique du panier et du montant total en fonction des modifications de quantités.

8. Interface de gestion des identifiants mails et du corps du mail

- Développement d'une interface conviviale pour la gestion des identifiants mails, tels que les adresses e-mail de l'expéditeur, les informations de connexion au serveur SMTP, etc.

- Possibilité de personnaliser le contenu du mail de confirmation de commande.

9. Interface de gestion du ticket

- Développement d'une interface utilisateur pour la gestion des paramètres du ticket, tels que le logo, les mentions légales, le message de pied de page, les mentions spéciales, les exigences RGPD, etc.

10. Sécurité de l'application

- Ajout de mesures de sécurité, notamment l'utilisation de tokens CSRF (Cross-Site Request Forgery) pour l'accès à l'API. Cela permettra de prévenir les attaques CSRF en vérifiant l'origine des requêtes et en s'assurant que seules les requêtes légitimes provenant de l'application sont autorisées.

11. Développement de la base de données et du backend

- Modification et extension de la base de données

existante pour prendre en charge les nouvelles fonctionnalités et la gestion multi-boutiques.

- Développement d'un backend basé sur l'API ASP.NET avec Visual Studio comme IDE principal pour la gestion des entités, y compris les utilisateurs, les cartes, les catégories, les clients, les commandes, les bas de page du ticket et le système d'envoi d'e-mails.

Ces spécifications techniques décrivent les technologies, les fonctionnalités et les mesures de sécurité à mettre en œuvre pour l'amélioration de l'application Android existante. Elles serviront de guide lors de la conception et du développement du projet, en veillant à l'utilisation optimale des outils et des technologies mentionnées, tout en garantissant la sécurité de l'application.

## VI. Réalisations

## 1.Le modèle

La première chose que j’ai réalisé est le modèle, commun au front end et au backend, de l’application. La première étape à été de réaliser un UML afin de modéliser l’organisation du modèle et d’anticiper les fonctions nécessaires :

//TODOUML corrigé

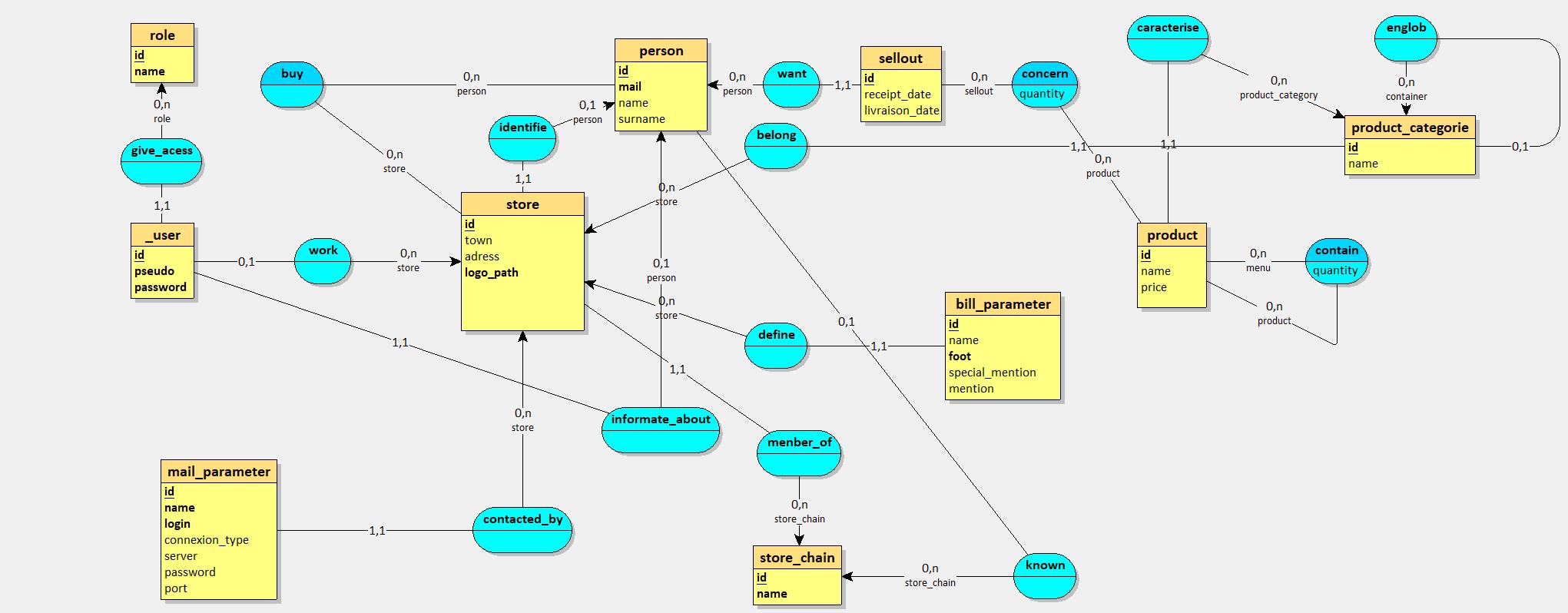
## 2.Back-End

Une fois le modèle conçus j’ai choisi de réaliser pendant mon stage est la partie backend de l’application qui consiste en une Api permettant de réaliser un CRUD de la base de données en étant contrôlé à distance par la partie client. Afin d’atteindre cet objectif je me suis d’abord concentré sur la base de données puis j’ai d’abord réalisé une partie métier avec toutes les classes associées avant de créer une DAO permettant l’échange avec la base de données et enfin des contrôleurs gérant les requêtes HTTP.

### a. Base de données

J’ai commencé par modéliser la base de données via un diagramme MCD. Le principal défi a été d’assurer l’étanchéité des données entre les Magasins, ce qui a nécessité plusieurs réécritures. Pour sécuriser l’accès aux données j’ai fini par arriver à la structure de base de données suivante :

\*MCD :



\*MLD : Une image contenant texte, diagramme, capture d’écran, Plan

Description générée automatiquement

Afin de simplifier l’utilisation de la base de donnée dans le code j’ai choisit de réaliser plusieurs view regroupant les informations nécessaire pour extraitre les informations nécessaire pour l’instantiation des objets du modèle dont voici un des exemples :

CREATE VIEW **fullsellout** AS

SELECT id\_person,receipt\_date,livraison\_date,id\_product,quantity,sellout.id id,id\_store

FROM sellout

JOIN concern ON id\_product=id

JOIN product ON product.id=id\_product

JOIN product\_category ON product.id\_product\_category=product\_category.id ORDER BY sellout.id;

Le but de cette vue est de récupérer les informations d’une vente en prenant en compte le magasin qu’elle concerne afin de filtrer correctement les résultats des requêtes.

J’ai également implémenté des triggers afin d’améliorer la robustesse des données et d’assurer que le paramétrage du mail et du ticket de caisse est par défaut le nom du magasin qui les à créés, si il est disponible, voici un exemple avec la facturation :

CREATE OR REPLACE FUNCTION **rename\_bill**() RETURNS TRIGGER AS $$

BEGIN

IF NEW.name IS NULL THEN

    SELECT name INTO NEW.name

        FROM store

        WHERE id = NEW.id\_store;

END IF;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER **update\_bill\_parameter\_name**

BEFORE INSERT OR UPDATE ON bill\_parameter

FOR EACH ROW

EXECUTE FUNCTION rename\_bill();

Enfin j’ai créé des fonctions permettant d’obtenir des informations non triviales dans la base de données, l’exemple le plus marquant est cette fonction permettant de savoir si un produit correspond à un menu ou non, afin de pouvoir facilement le séparer des autres types de produits :

CREATE OR REPLACE FUNCTION **is\_menu**(id INT) RETURNS BOOLEAN AS $$

BEGIN

    SELECT \* FROM contain WHERE id\_menu=id;

    RETURN FOUND;

END;

$$ LANGUAGE plpgsql;

### b. Api et utilisation du design pattern factory.

#### b.1 Model

L’architecture pour le modèle, commun au back end et au front end, que j’ai utilisé est la suivante :

Placeholder

#### b.2 DAO

Afin d’interagir avec la base de données j’ai décidé d’utiliser un design pattern DAO, nécessaire pour simplifier la création des API. L’interface générique de la DAO est la suivante :

namespace Boucher\_Double\_Back\_End.Models.Manager

{

/// <summary>

/// Interface representing the access to the database, the generic parameter determine which data to retrieve in the concrete implementations of its methods

/// </summary>

/// <typeparam name="T">The entity type we want to interact inside the database </typeparam>

public interface IDAO<T>

{

/// <summary>

/// Attribute that help to securise the application, egal to the stored value in the session and limit the interaction to the user associated shop

/// </summary>

public Store Store { get; set; }

/// <summary>

/// Attribute that help to securise the application, egal to the stored value in the session and limit certain action to certain <see cref="User"/> <see cref="Role"/>

/// </summary>

public User User { get; set; }

/// <summary>

/// Get one element form the database

/// </summary>

/// <param name="id"></param>

/// <returns>The instance of <see cref="T"/></returns>

Task<T> GetBytIdAsync(int id);

/// <summary>

/// Get all the accessible element from the database

/// </summary>

/// <returns>A list of the instance of <see cref="T"/></returns>

Task<List<T>> GetAllAsync();

/// <summary>

/// Delete on element from the database using its id

/// </summary>

/// <param name="id">The id of the entity to delete</param>

/// <returns>Return if the operation was a success or not</returns>

Task<bool> DeleteAsync(int id);

/// <summary>

/// Update the database if the element is accessible to the user

/// </summary>

/// <param name="entity">The <see cref="T"/> entity that contains the updated values</param>

Task UpdateAsync(T entity);

/// <summary>

/// Create an element in the database

/// </summary>

/// <param name="entity">The <see cref="T"/> entity that contains the inserted values</param>

/// <returns>The success or failure of the operation</returns>

Task<bool> CreateAsync(T entity);

}

}

Une fois cette interface crée le principal défi a été de travailler avec les classes du modèle incluant d’autres classes. Un des exemples le plus parlant est la fonction GetAll de la classe SelloutDAO :

namespace Boucher\_Double\_Back\_End.Models.Manager

{

/// <summary>

/// Interface representing the access to the database, the generic parameter determine which data to retrieve in the concrete implementations of its methods

/// </summary>

/// <typeparam name="T">The entity type we want to interact inside the database </typeparam>

public interface IDAO<T>

{

/// <summary>

/// Attribute that help to securise the application, egal to the stored value in the session and limit the interaction to the user associated shop

/// </summary>

public Store Store { get; set; }

/// <summary>

/// Attribute that help to securise the application, egal to the stored value in the session and limit certain action to certain <see cref="User"/> <see cref="Role"/>

/// </summary>

public User User { get; set; }

/// <summary>

/// Get one element form the database

/// </summary>

/// <param name="id"></param>

/// <returns>The instance of <see cref="T"/></returns>

Task<T> GetBytIdAsync(int id);

/// <summary>

/// Get all the accessible element from the database

/// </summary>

/// <returns>A list of the instance of <see cref="T"/></returns>

Task<List<T>> GetAllAsync();

/// <summary>

/// Delete on element from the database using its id

/// </summary>

/// <param name="id">The id of the entity to delete</param>

/// <returns>Return if the operation was a success or not</returns>

Task<bool> DeleteAsync(int id);

/// <summary>

/// Update the database if the element is accessible to the user

/// </summary>

/// <param name="entity">The <see cref="T"/> entity that contains the updated values</param>

Task UpdateAsync(T entity);

/// <summary>

/// Create an element in the database

/// </summary>

/// <param name="entity">The <see cref="T"/> entity that contains the inserted values</param>

/// <returns>The success or failure of the operation</returns>

Task<bool> CreateAsync(T entity);

}

}

Une fois les classes de la DAO créée l’étape suivante à été de mettre en place un plan de test afin de m’assurer du bon fonctionnement du cette partie centrale du code. Comme toutes les fonctions utilise la base de données j’ai dû trouver une solution pour que celle-ci reste identique au fur et a mesure des tests. Afin de réaliser cet objectif j’ai initialisé chacun de mes tests en commençant une transaction sur la base de données et terminé chacun de mes tests en effectuant un Rollback. Ainsi les modifications sur la base de données n’était pas persisté en dehors de la portée des tests, assurant une stabilité des résultats. L’implémentation concrète de cette solution est la suivante :

namespace Boucher\_Double\_Back\_EndTests.Models.Manager

{

[TestClass]

public class TestSetups

{

private TransactionScope \_transaction;

[TestInitialize]

public void Initialize()

{

\_transaction = new TransactionScope();

// Set up any necessary test data here

}

[TestCleanup]

public void Cleanup()

{

\_transaction.Dispose(); // Rolls back the transaction

}

}

}

Pour que tout mes tests soit initialisé correctement je définie chaque classe de test comme un enfant de la classe TestSetups. Grace à cela je peux manipuler la base de données sans compromettre son intégrité. Chaque série de tests suit une trame similaire, en instanciant la classe dao associé avec un User et un Store permettant d’interagir correctement avec la base de données, dont voici un exemple :

[TestClass()]

public class MailParameterDAOTests:TestSetups

{

[TestMethod()]

public async Task CreateTestAsync()

{

User user = await new UserDAO().Connect("Simon", "Test");

MailParameterDAO dao = new()

{

User = user,

Store = user.Store,

};

MailParameter mail = new()

{

Server = "test",

IdStore = user.Store.IdStore,

Name = "test",

ConnexionType = "test",

Port=0,

Login="test",

Password="test123a,",

};

Assert.IsTrue(await dao.CreateAsync(mail));

}

[TestMethod()]

public async Task DeleteTestAsync()

{

User user = await new UserDAO().Connect("Simon", "Test");

MailParameterDAO dao = new()

{

User = user,

Store = user.Store,

};

Assert.IsTrue(await dao.DeleteAsync(9));

}

[TestMethod()]

public async Task GetAllTestAsync()

{

User user = await new UserDAO().Connect("Simon", "Test");

MailParameterDAO dao = new()

{

User = user,

Store = user.Store,

};

Assert.AreEqual(1,(await dao.GetAllAsync()).Count);

}

[TestMethod()]

public async Task GetByIdTestAsync()

{

User user = await new UserDAO().Connect("Simon", "Test");

MailParameterDAO dao = new()

{

User = user,

Store = user.Store,

};

Assert.AreEqual(10, (await dao.GetBytIdAsync(9)).Port);

Assert.AreEqual("testserveur", (await dao.GetBytIdAsync(9)).Login);

Assert.AreEqual("test1", (await dao.GetBytIdAsync(9)).Name);

Assert.AreEqual("simon", (await dao.GetBytIdAsync(9)).Password);

Assert.AreEqual("type", (await dao.GetBytIdAsync(9)).ConnexionType);

}

[TestMethod()]

public async Task UpdateTestAsync()

{

User user = await new UserDAO().Connect("Simon", "Test");

MailParameterDAO dao = new()

{

User = user,

Store = user.Store,

};

MailParameter mail = new()

{

Server = "test",

IdStore = user.Store.IdStore,

Name = "test",

ConnexionType = "test",

Port = 0,

Login = "test",

Password = "test123a,",

Id=9

};

await dao.UpdateAsync(mail);

}

}

La réalisation et la validation de tout ces tests était une condition indispensable à la poursuite de mon travail vers la prochaine étape : la création des contrôleurs de l’API.

#### b.3 Les contrôleurs

Afin de limiter au la maximum la répétition j’ai décidé d’utiliser la généricité de la classe DAO et d’implémenter un design pattern Factory afin d’instancier la bonne classe DAO en fonction de l’objet qu’on essaie de récupérer via l’API. L’implémentation concrète du design pattern a pris la forme suivante :

public abstract class APIController<T>:ControllerBase

{

private IDAO<T>? daoInstance;

/// <summary>

/// DAO attribute, its concrete implementation is determined by the <see cref="T"/> attribute

/// </summary>

public IDAO<T>? DAO

{

get

{

if (daoInstance == null)

{

switch (typeof(T))

{

case Type type when type == typeof(BillParameter):

daoInstance = new BillParameterDAO() as IDAO<T>;

break;

case Type type when type == typeof(Category):

daoInstance = new CategoryDAO() as IDAO<T>;

break;

case Type type when type == typeof(MailParameter):

daoInstance = new MailParameterDAO() as IDAO<T>;

break;

case Type type when type == typeof(User):

daoInstance = new UserDAO() as IDAO<T>;

break;

case Type type when type == typeof(Store):

daoInstance = new StoreDAO() as IDAO<T>;

break;

case Type type when type == typeof(Sellout):

daoInstance = new SelloutDAO() as IDAO<T>;

break;

case Type type when type == typeof(Menu):

daoInstance = new MenuDAO() as IDAO<T>;

break;

default:

return null;

}

}

return daoInstance;

}

}

En fonction de la classe avec laquelle on cherche à interagir la bonne DAO est instancié de manière anonyme, permettant de définir toutes les API une seule fois :

/// <summary>

/// Return all the entity if the User is connected using CSRF protection

/// </summary>

/// <returns>All the entitys as a Json</returns>

/// <exception cref="Exception"></exception>

[HttpGet]

public async Task<IEnumerable<T>> GetAsync()

{

if (HttpContext.Session.GetString("UserId") != null && DAO != null)

{

DAO.Store = new()

{

IdStore = int.Parse(HttpContext.Session.GetString("IdStore")),

};

DAO.User= new User { IdUser = int.Parse(HttpContext.Session.GetString("UserId")), Role = RoleExtensions.GetRoleFromString(HttpContext.Session.GetString("role")) };

return await DAO.GetAllAsync();

}

else

throw new Exception();

}

/// <summary>

/// Return a given entity identified by its id if the User is connected using CSRF protection

/// </summary>

/// <param name="id">The entity id</param>

/// <returns>One entity associated with the request id</returns>

/// <exception cref="Exception"></exception>

[HttpGet("{id}")]

public Task<T> GetBytIdAsync(int id)

{

if (HttpContext.Session.GetString("UserId") != null && DAO != null)

{

DAO.Store = new()

{

IdStore = int.Parse(HttpContext.Session.GetString("IdStore")),

};

DAO.User = new User { IdUser = int.Parse(HttpContext.Session.GetString("UserId")), Role = RoleExtensions.GetRoleFromString(HttpContext.Session.GetString("role")) };

return DAO.GetBytIdAsync(id);

}

else

throw new Exception();

}

/// <summary>

/// Add an entity data to the database if the User is connected using CSRF protection

/// </summary>

/// <param name="value"></param>

/// <returns></returns>

/// <exception cref="Exception"></exception>

[HttpPost]

public Task<bool> CreateAsync([FromBody] T value)

{

if (HttpContext.Session.GetString("UserId") != null && DAO != null)

{

DAO.Store = new()

{

IdStore = int.Parse(HttpContext.Session.GetString("IdStore")),

};

DAO.User = new User { IdUser = int.Parse(HttpContext.Session.GetString("UserId")), Role = RoleExtensions.GetRoleFromString(HttpContext.Session.GetString("role")) };

return DAO.CreateAsync(value);

}

else

throw new Exception();

}

/// <summary>

/// Update an entity data in the database if the User is connected using CSRF protection

/// </summary>

/// <param name="value"></param>

/// <exception cref="Exception"></exception>

[HttpPut]

public void UpdateAsync([FromBody] T value)

{

if (HttpContext.Session.GetString("UserId") != null && DAO != null)

{

DAO.Store = new()

{

IdStore = int.Parse(HttpContext.Session.GetString("IdStore")),

};

DAO.User = new User { IdUser = int.Parse(HttpContext.Session.GetString("UserId")), Role = RoleExtensions.GetRoleFromString(HttpContext.Session.GetString("role")) };

DAO.UpdateAsync(value);

}

else

throw new Exception();

}

/// <summary>

/// Delete an entity data in the database if the User is connected using CSRF protection

/// </summary>

/// <param name="value"></param>

/// <exception cref="Exception"></exception>

[HttpDelete("{id}")]

public async Task<bool> DeleteByIdAsync(int id)

{

if (HttpContext.Session.GetString("UserId") != null && DAO != null)

{

DAO.Store = new()

{

IdStore = int.Parse(HttpContext.Session.GetString("IdStore")),

};

DAO.User = new User { IdUser = int.Parse(HttpContext.Session.GetString("UserId")), Role = RoleExtensions.GetRoleFromString(HttpContext.Session.GetString("role")) };

return await DAO.DeleteAsync(id);

}

else

throw new Exception();

}

}

Pour obtenir toutes les API nécessaire il me suffit maintenant de les définir comme des implémentation concrète de la classe API Controller comme ceci :

namespace Boucher\_Double\_Back\_End.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class BillParameterController : APIController<BillParameter>

{

}

}

Chacune de ces API interagit avec l’utilisateur, le magasin et le token CSRF stocké dans les variables de session afin d’assurer la sécurisation de l’API. Ces variables sont définies lors de la connexion de l’utilisateur dans la classe UserController:

[HttpGet("{Login}/{Password}")]

public async Task<JsonResult> Connect(string Login, string Password)

{

UserDAO userDAO = new ();

User? user=await userDAO.Connect(Login, Password);

if (user!=null)

{

string csrfToken = Security.GenerateCSRFToken();

HttpContext.Session.SetString("UserId",user.IdUser.ToString());

HttpContext.Session.SetString("Username", user.Login);

HttpContext.Session.SetString("IdStore",user.Store.IdStore.ToString());

HttpContext.Session.SetString("CSRF", csrfToken);

HttpContext.Session.SetString("role", user.Role.ToString());

return new JsonResult(new { success = true, csrf = csrfToken, Login ,role=user.Role });

}

else

return new JsonResult(new { success = false, csrf = "",Login});

}

#### b.4 Securisation

De plus afin de rajouter une couche de sécurité supplémentaire les mots de passe des utilisateurs sont haché en utilisant les méthodes de la classe Security, qui gère aussi la création des tokens CSRF :

namespace Boucher\_DoubleModel.Models.Entitys

{

/// <summary>

/// Class that hash password before sending them to the database

/// </summary>

public class Security

{

/// <summary>

/// Parameter of the salt

/// </summary>

private const int SaltSize = 16; //128 / 8, length in bytes

/// <summary>

/// Parameter of the salt

/// </summary>

private const int KeySize = 32; //256 / 8, length in bytes

/// <summary>

/// Parameter of the salt

/// </summary>

private const int Iterations = 10000;

private static readonly HashAlgorithmName \_hashAlgorithmName = HashAlgorithmName.SHA256;

private const char SaltDelimeter = ';';

/// <summary>

/// Return the modified password

/// </summary>

/// <param name="password"></param>

/// <returns></returns>

public static string Hash(string password)

{

byte[] salt = RandomNumberGenerator.GetBytes(SaltSize);

byte[] hash = Rfc2898DeriveBytes.Pbkdf2(password, salt, Iterations, \_hashAlgorithmName, KeySize);

return string.Join(SaltDelimeter, Convert.ToBase64String(salt), Convert.ToBase64String(hash));

}

/// <summary>

/// Check if a password correspond to what is stored in the database

/// </summary>

/// <param name="passwordHash"></param>

/// <param name="password"></param>

/// <returns></returns>

public static bool Validate(string passwordHash, string password)

{

string[] pwdElements = passwordHash.Split(SaltDelimeter);

byte[] salt = Convert.FromBase64String(pwdElements[0]);

byte[] hash = Convert.FromBase64String(pwdElements[1]);

byte[] hashInput = Rfc2898DeriveBytes.Pbkdf2(password, salt, Iterations, \_hashAlgorithmName, KeySize);

return CryptographicOperations.FixedTimeEquals(hash, hashInput);

}

/// <summary>

/// Generate a unique CSRF token to securise the application

/// </summary>

/// <returns>The CSRF token send to the client</returns>

public static string GenerateCSRFToken()

{

byte[] tokenBytes = new byte[32]; // Generate a 32-byte CSRF token

using (RandomNumberGenerator rng = RandomNumberGenerator.Create())

{

rng.GetBytes(tokenBytes);

}

string csrfToken = Convert.ToBase64String(tokenBytes);

return csrfToken;

}

}

Les tokens CSRF mentionnés précédemment jouent un rôle central dans l’application, en effet dans le cadre d’une application mobile l’authentification via des variables de session est impossible, c’est là qu’entre en jeux les tokens CSRF, envoyer dans le header de chaque requête à l’Api, en dehors bien évidemment de la requête de connexion, ils permettent d’identifier un utilisateur à l’aide d’une table Clé-Valeurs des utilisateurs connectés.

#### c.Le déploiement

Maintenant que mon API était construite et tester il était temps de la déployer afin de pouvoir l’utiliser. Nous avons convenu avec mon tuteur de stage d’utiliser un hébergement à distance et après avoir consulté plusieurs offres d’hébergement nous avons décidé d’utiliser l’hébergement ASP Starter de l’hébergeur LWS, car il inclut Asp.Net. Pour déployer j’ai envoyé les fichiers du build par protocole TCP et j’ai reconstruit la base de données sur le serveur distant. L’hébergement ne supportant pas PostgreSQL j’ai dû changer la technologie de la base de données et transitionner vers MySql. Après avoir réécrit le script pour être compatible avec ce SGBD j’ai dû également faire des modifications légères au code afin de pouvoir utiliser ce type de SGBD. Ayant maintenant une adresse web sur laquelle tester mes appels à l’API je pouvais m’atteler à la suite du travail : la création de l’application mobile.

## 3.Le front End

## a. La conception

Afin de pouvoir travailler le plus efficacement possible sur la partie mobile j’ai commencé, avant même la conception du back end, par réaliser une maquette de l’application sur Figma. Mon objectif premier était de me concentrer sur l’expérience utilisateur et de pouvoir discuter avec mon maitre de stage de ma vision du fonctionnement de l’application en pouvant m’appuyer sur des démonstrations visuelles, les détails de l’apparence des objets étant à ce stade non considéré.

Voici quelques exemple des pages de l’application réalisé avec cet outil :

\*Le menu de navigation

Une image contenant texte, capture d’écran, rouge, Police

Description générée automatiquement

\*L’écran de gestion d’une commande

Une image contenant texte, capture d’écran, affichage, diagramme

Description générée automatiquement

Cette étape m’a permis, après plusieurs allez retour par mail ou téléphone et une rencontre avec mon tuteur afin de montrer l’utilisation pratique de l’application de préciser plusieurs choses :

\*la nécessité de pouvoir rester connecter, un client pouvant être amener à prendre la commande lui-même

\* l’importance de pouvoir mettre à jour facilement le catalogue et surtout les commandes.

\*L’organisation des catégories de produits

## b. La création de l’application mobile

### b.1 Le choix d’une navigation à onglet

Afin de s’approcher au mieux de la maquette que j’ai réalisé j’ai choisi de me diriger vers une application a onglet car c’est à mon sens la forme d’application permettant d’intégrer un mieux la navigation par menu que j’avais en tête. Tous les pages de l’application seront alors contenus dans une Shell qui contient à la fois la logique de navigation dans son ViewModel et le menu de navigation dans sa vue.

### b.2. Le modèle MVVM

Le code de l’application mobile a été organisé suivant le modèle MVVM : Model View ViewModel qui est donc composée des trois parties suivantes :

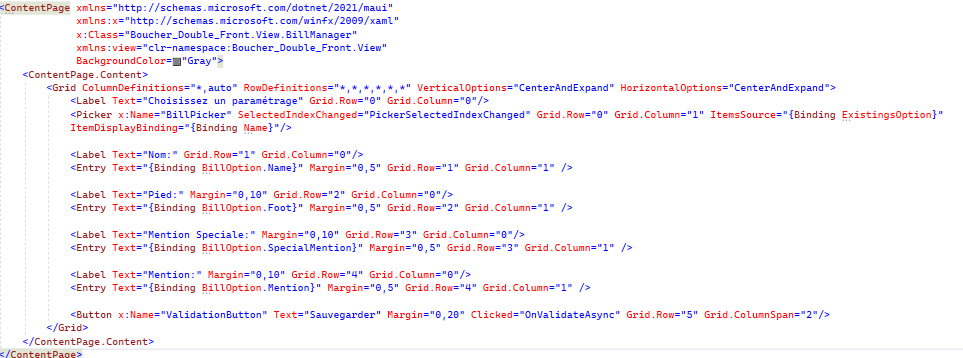
**Le Model :** Définit les types qui représentent vos données métier. Cela inclut tout ce qui est nécessaire pour modéliser le domaine d’application principal, et souvent la logique d’application principale. Dans le cas de mon application il s’agit du modèle commun mentionné précédemment auquel s’ajoute quelques classes utilitaires.

**La View :** La couche de **vue** définit l’interface utilisateur à l’aide du balisage XAML. Le balisage comprend des expressions de liaison de données qui définissent la connexion entre des composants d’interface utilisateur spécifiques et divers membres de modèle et de modèle de vue. La vue comprend également un fichier de code , permettant de gérer les effets des différentes actions et de manipuler l’interface en conséquence, remplissant par analogie le rôle que remplirait un script Javascript dans une page web.

**La ViewModel :** La couche de **modèle de vue** fournit des cibles de liaison de données pour la vue. Dans de nombreux cas, le modèle de vue expose directement le modèle ou fournit des membres qui wrappent des membres de modèle spécifiques. Le modèle de vue peut également définir des membres pour effectuer le suivi de données relatives à l’interface utilisateur, mais pas au modèle, comme l’ordre d’affichage d’une liste d’éléments. Le modèle de vue sert également de point d’intégration à d’autres services comme le code d’accès aux bases de données.

Pour illustrer ces différents concept voici quelques exemples de codes :

Une View :



Et une partie du code associé, ici l’évent listener du bouton de validation :



Et la ViewModel associé :



On voit dans ces deux exemples la séparation claire des rôles, bien que le bouton de validation manipule des données l’organisation de celle-ci et la logique de leur récupération lui est totalement étranger, permettant une meilleure lisibilité et une délimitation plus claire des rôles de chaque composant du code de l’application.

## b.3 Gestion de l’accès au backend

Comme mentionné précédemment il a été nécessaire, pour des questions de sécurité, d’inclure une authentification par token CSRF. Afin de confirmer l’identité de l’utilisateur de l’application celui-ci doit être inclut dans chacune des requêtes envoyées à l’API, en dehors de la requête de connexion.

La première étape est de récupérer le token lors de la connexion :



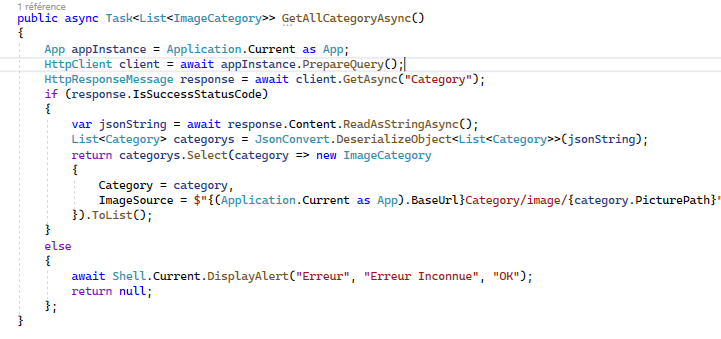
Une fois cette étape effectuée le token sera inclut dans toutes les requêtes suivantes permettant l’identification :



En exportant la préparation du client et l’ajout du token dans une fonction à part je limite le risque d’oublie et simplifie la création des nombreuses requêtes nécessaire au bon fonctionnement de l’application. Cette fonction test également si l’utilisateur est toujours connecté à l’api, celle ci supprimant les connexions automatiquement après un certain temps, et l’y reconnecte le cas échéant.

### b.4 Les appels à l’api

Comme mentionné précédemment le fonctionnement de l’application mobile nécessite de nombreux appels à l’API mentionnée dans une partie précédente du rapport. Tous les appels à l’api en dehors de la connexion ont tous une structure semblable, dont voici une illustration avec l’appel permettant d’obtenir toutes les catégories de produits enregistrées pour un magasin :



De nombreux appels à l’api sont également liées à des event listener, souvent pour la création ou la mise a jour d’une entité. Voici par exemple l’event listener associé au bouton de validation de la page de gestion d’une catégorie unique :

* La création



* La mise à jour :



On vérifie lors de la mise a jour et de la création que le nom de la catégorie est valide, comme la fonction « PrepareQuery » définit déjà l’url commune a toute les requêtes il n’y a plus que a la compléter en fonction de l’entité concerné et