Stage de deuxième année

I. Présentation globale
- A. L'objectif du stage
- B. Organisations et directives du projet
- C. Mon environnement

II. Les technologies utilisées
- A. L'application
- B. Base de données
- C. API REST

III. Fonctionnalités utilisateurs
- A. Fonctionnalités de l'appli
- B. Diagramme des cas d'utilisations

IV. Architecture détaillée du projet
- A. L'application mobile
  - 1. Classes Activity
  - 2. Classes API
  - 3. Classes Data
- B. API REST
  - 1. Les fichiers de configurations
  - 2. Les contrôleurs
  - 3. Les modèles

V. Sécurité
- A. Sécurité technique
  - B. Sécurisation des accès
  - C. Sécurisation des saisies
  - D. Sécurisation des comptes
- VI. Conformité à la RGPD
  - A. Consentement et transparence
  - B. Collecte conforme des données
  - C. Droit des personnes

VII. Pistes d'améliorations
- A. Securite
- B. Base de données
- C. Application

VIII. Conclusion

I. Présentation globale

A. L’objectif du stage

Dans le cadre de mon stage, il m’a été demandé de réaliser une application mobile capable de générer des prises de devis pour des meubles réfrigérés. Cela vise à moderniser le processus de **demande de devis** pour l’entreprise. Actuellement, les clients envoient leurs demandes par e-mail, ce qui entraîne des imprécisions et des allers-retours pour obtenir les informations nécessaires. L’application permet aux clients de remplir un formulaire structuré, garantissant que toutes les données requises pour établir un devis soient bien fournies. Une fois complétée, la demande de devis est envoyée automatiquement par e-mail à Loca Service dans une boite mail dédiée, au client et aux personnes mentionnées par le client. L’appli a pour vocation a être publiée pour les clients, elle doit donc au standard de ce que l’entreprise propose d’ordinaire.

B. Organisations et directives du projet

Lors de mon arrivée, il m’a été fourni une liste des informations à collectées pour que la prises de devis soit valide. Voici les informations demandées:

- Informations sur le commercial (nom, prénom, mail, téléphone et société)
- Informations du contact en magasin (nom, prénom, mail, téléphone)
- Informations sur la commande (les dates, le type de meuble, les dimensions, frais ou surgelé, ouvert ou fermé, une PLV ou non)
- Informations supplémentaires via un champ de texte en fin de formulaire

De cette liste j’ai été laissé en autonomie quant à la direction que prendrait le développement et des outils que j’utiliserais. La seule chose imposée m’a été d’héberger ma base de données sur OVH dans le groupe de l’entreprise, la solution est exploitée pour quelques projets de l’entreprise.

De ce fait j’ai réfléchis à la structure du projet et j’ai décidé de l’orienter de la façon suivante: créer l’application, ensuite développer une base de données et mettre en place un API REST pour assurer la connexion entre les deux. J’ai organisé toutes mes tâches via un planning détaillé dont voici un exemple:

Nom de la tâche	État d'avance...	Responsable	Période	Priorité
Développement de l'appli	Terminé	Maxime	24 février 2025 → 4 mars 2025	Haute
Faire l'arborescence, la structure	Terminé	Maxime	25 février 2025	Haute
Développer le formulaire en statique	Terminé	Maxime	25 février 2025 → 28 février 2025	Haute
Connecter l'API	Terminé	Maxime	24 mars 2025 → 25 mars 2025	Haute
Sécuriser l'appli	Terminé		24 mars 2025 → 28 mars 2025	Haute
Echange de données Appli API	Terminé		24 mars 2025 → 28 mars 2025	Haute
Rendre l'interface moderne	Terminé	Maxime		Basse
+ Nouvelle page imbriquée				
Mise en place base de données	Terminé	Maxime	3 mars 2025 → 7 mars 2025	Haute
Faire le MCD et MLD	Terminé	Maxime	3 mars 2025	Haute
Autoformation sur OVH	Terminé	Maxime	3 mars 2025	Haute
Faire le script SQL	Terminé	Maxime	4 mars 2025	Haute
Déployer la base de données sur PHPMyAdmin	Terminé	Maxime	5 mars 2025	Haute
Mettre en place le SMTP	Terminé	Maxime	6 mars 2025	Haute
+ Nouvelle page imbriquée				
Développement de l'API	Terminé	Maxime	10 mars 2025 → 21 mars 2025	Haute
Définir les méthodes requises	Terminé	Maxime	10 mars 2025	Haute
Développement des méthodes	Terminé	Maxime	10 mars 2025 → 21 mars 2025	Haute
Connecter à la base données	Terminé	Maxime	10 mars 2025	Haute
Sécuriser les accès	Terminé	Maxime	10 mars 2025 → 21 mars 2025	Haute
+ Nouvelle page imbriquée				
Faire une documentation	Terminé	Maxime	31 mars 2025 → 4 avril 2025	Moyenne

De plus, tous les lundis nous faisions un point avec tout le service informatique pour parler des projets en cours, de la vie de l'entreprise. J'avais aussi des points régulièrement avec mon tuteur pour parler de la direction que prenais le projet, des choix que je faisais.

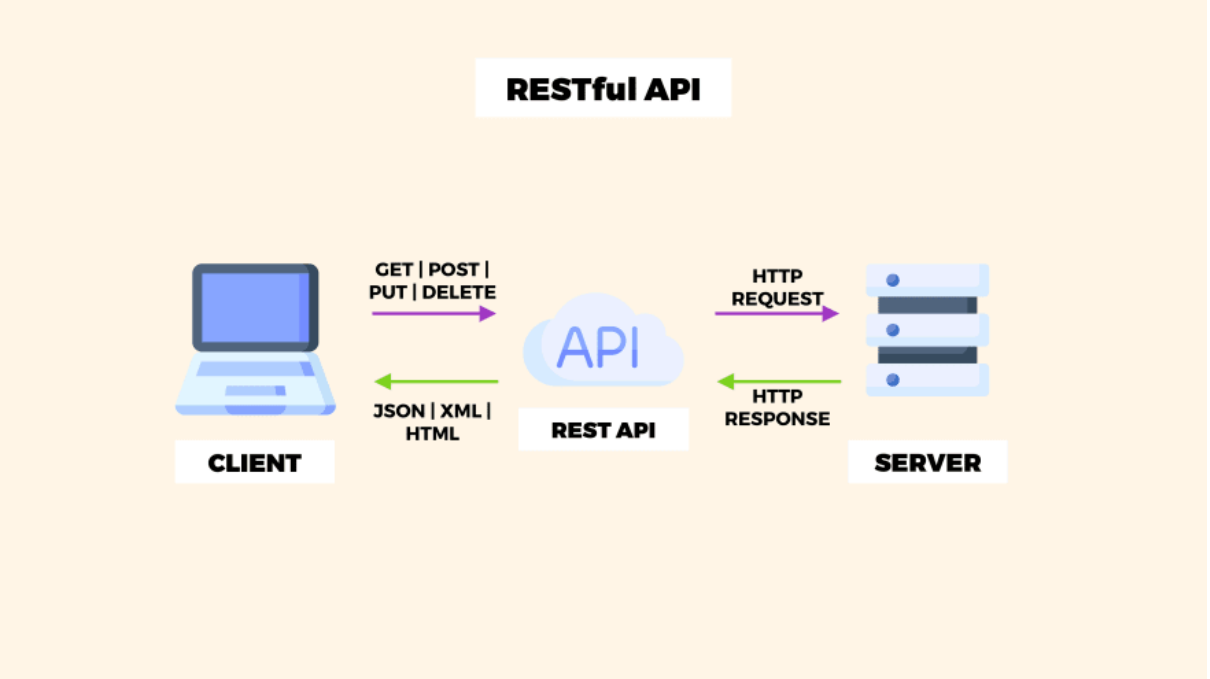
J'ai aussi eu le droit vers la fin de mon stage à une présentation de mon appli à l'équipe qui s'occupe des devis, qui ont unanimement validé l'utilité de l'appli et l'ont qualifié de projet utile et m'ont félicité pour ce dernier.

C. Mon environnement

J'étais dans le service informatique avec les personnes qui s'occupaient des réseaux et systèmes informatiques de l'entreprise. J'ai pu échanger avec eux sur des sujets par exemple d'hébergement de la base de données, d'OVH, de cybersécurité pour mon projet d'appli mobile. Ainsi, j'ai pu m'assurer de mettre en place le développement dans les conditions optimales pour moi et en accord avec les mesures en place de l'entreprise.

II. Les technologies utilisées

Le projet est réalisé sur trois couches : **l'application**, **l'API REST** et la **base de données**. Cette structure permet de mieux sécuriser les accès de l'application à la base de données, de mieux structurer le projet. De plus, un serveur **SMTP** permet l'envoi des mails aux clients et au mail dédié à la réception côté Loca Service.



A. L'application

L'application est développée sur **Android Studio**, un IDE, sous **Android SDK** *(API 26 – Android Oreo)* ce qui le rend compatible avec la majorité des appareils Android sur le marché. Le code est en **Kotlin**, un langage plus adapté et moderne pour le développement mobile Android.

Pour permettre certaines fonctionnalités de l'appli, il faut installer des bibliothèques supplémentaires au projet :

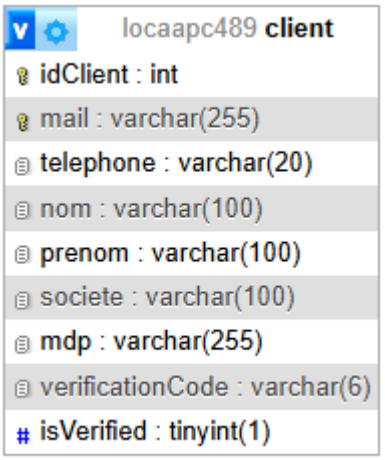
- Pour les PDF, le projet utilise **iText7** qui permet de créer et manipuler des fichiers PDF.
- Pour communiquer avec l'API, le projet utilise **Retrofit** qui permet de faire des appels réseaux (**HTTP**).
- Pour la **programmation asynchrone**, qui n'est pas supportée de base dans Android Studio en Kotlin, le projet contient la bibliothèque **Kotlin Couroutines**.

- Pour le design, la bibliothèque **Material Design 3** sera utilisée pour rendre l'appli plus moderne et responsive.

B. Base de données

Etant donné que les clients doivent pouvoir s'authentifier, une base de données est nécessaire. Elle permet aussi de pouvoir enregistrer les informations du client et ainsi remplir automatiquement certains champs du formulaire.

Ce traitement des données doit être réalisé dans le respect des lois en vigueur, notamment de la **RGPD**. La base de données est en **MySQL** et contient qu'une seule table, la table **client**. Cette dernière contient : le nom, le prénom, le mail, le téléphone, la société, le mot de passe, le stockage d'un code de vérification pour certaines actions et un indicateur de si le compte est vérifié ou non. Le mot de passe est bien entendu haché avant d'être stocké.



	#	Nom	Type	Interclassement	Attributs	Null	Valeur par défaut	Commentaires	Extra
☐	1	**idClient**	int			Non	*Aucun(e)*		AUTO_INCREMENT
☐	2	**mail**	varchar(255)	utf8mb4_0900_ai_ci		Non	*Aucun(e)*		
☐	3	**telephone**	varchar(20)	utf8mb4_0900_ai_ci		Oui	*NULL*		
☐	4	**nom**	varchar(100)	utf8mb4_0900_ai_ci		Oui	*NULL*		
☐	5	**prenom**	varchar(100)	utf8mb4_0900_ai_ci		Oui	*NULL*		
☐	6	**societe**	varchar(100)	utf8mb4_0900_ai_ci		Oui	*NULL*		
☐	7	**mdp**	varchar(255)	utf8mb4_0900_ai_ci		Oui	*NULL*		
☐	8	**verificationCode**	varchar(6)	utf8mb4_0900_ai_ci		Oui	*NULL*		
☐	9	**isVerified**	tinyint(1)			Non	0		

C. API REST

Pour faire communiquer l'application et la base de données, une **API REST** est utilisée. Cette API REST utilise les protocoles **HTTP** *(GET, POST, PUT, DELETE)*

pour effectuer des opérations sur la base de données et des documents **JSON** pour transmettre des données.

L'API est développée en **PHP**, un langage courant pour les manipulations de données, côté serveur. Le langage, du fait de sa longévité, est stable et adapté à cet usage. Utiliser une API REST permet de mieux sécuriser le code et les accès à la base de données.

La connexion à la base de données est réalisée à l'extérieur du dossier racine du domaine, permettant de le cacher à la vue de tous et ne pas divulguer les informations sur cette dernière. Les requêtes sont effectuées en utilisant la méthode **PDO** de PHP, ce qui empêche toutes tentatives d'injection SQL. De plus deux fichiers **.htaccess** sécurisent le tout, ainsi que des vérifications pour les accès.

Le **SMTP** est aussi géré dans l'API REST. La connexion à ce dernier est sécurisée de la même manière que la connexion à la base de données. Pour envoyer des mails, le code PHP utilise la bibliothèque **PHP Mailer**. Cette bibliothèque permet l'utilisation d'un **SMTP** pour envoyer des mails.

III. Fonctionnalités utilisateurs

A. Fonctionnalités de l'appli

L'appli permet principalement de créer une **demande de devis** au format **PDF** qui est envoyée par mail au client et à Loca Service. Pour se faire plusieurs fonctionnalités sont mises en place.

Tout d'abord le client doit pouvoir avoir un compte et se connecter. De ce fait l'appli permet de **créer un compte**, de **se connecter** avec pour y accéder**, le modifier** une fois dans l'application, **changer son mot de passe** en cas d'oubli. Il faut noter que pour le changement de mot de passe, un **code à renseigner** sur l'appli est fourni par mail à l'utilisateur afin de valider la demande. De même, à la création du compte le client doit renseigner un code fourni par mail afin de **valider son compt**e. Sans un compte validé, l'utilisateur de ne peut pas accéder à l'application.

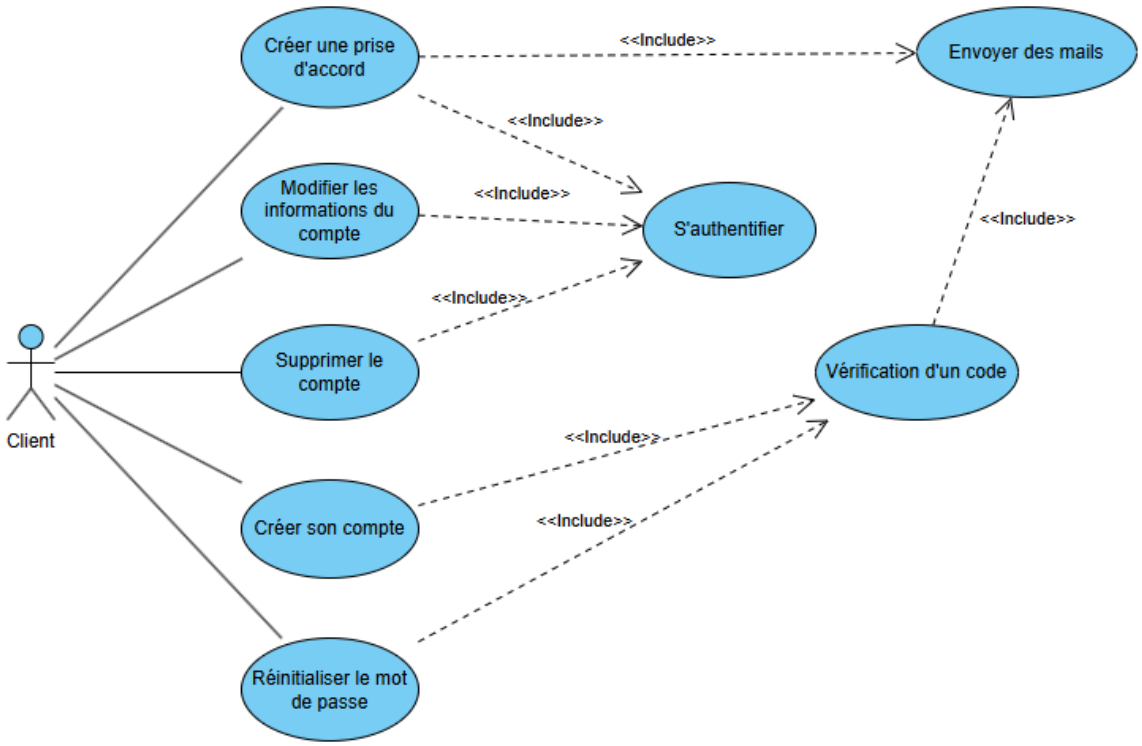
Une fois son compte créé, vérifié, le client se connecte et accède à la page principale où il peut choisir entre différentes actions. C'est sur cette page que l'utilisateur peut accéder à la modification des informations de son compte,

mais aussi **se déconnecter**. C’est aussi à ce moment que sont récupérées les données du client dans la base de données.

Sur la page principale, l’utilisateur peut accéder à une page **FAQ** contenant les questions les plus fréquentes pour apporter une potentielle réponse aux problèmes qu’ils pourraient rencontrer. Sinon il peut aussi y trouver les informations de contact de l’entreprise.

La fonctionnalité principale de l’application est la **création d’une demande de devis**. L’utilisateur peut y accéder depuis la page principale. Ce processus est une **suite de formulaire** pour récupérer toutes les informations nécessaires. Il a été fait le choix de diviser le formulaire en plusieurs pages afin d’alléger ce dernier pour le rendre plus **fluide** et lui donner un aspect moins lourd. D’abord on a les informations du commercial, qui se remplissent automatiquement avec les informations du compte en base de données, les informations du magasin, celles du contact en magasin, les dates, les meubles, un potentiel commentaire du client à propos de sa commande et enfin le récapitulatif de la demande de devis. Ce récapitulatif permet au client de vérifier, modifier si besoin, les informations et ensuite de valider ce dernier pour ainsi créer, au format PDF, un récapitulatif complet des informations saisies. Un exemplaire est enregistré sur le téléphone du client, un est envoyé par mail au client, un au contact en magasin et un à Loca Service.

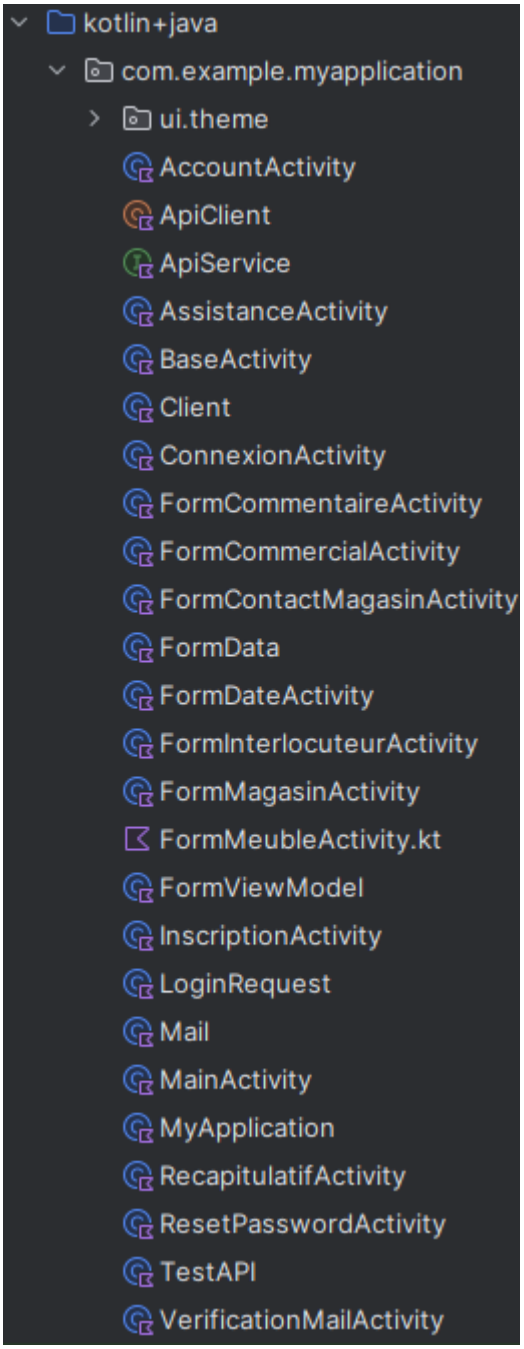
B. Diagramme des cas d’utilisations



IV. Architecture détaillée du projet

A. L'application mobile

Comme dit précédemment, le projet est développé sous **Android Studio** en **Kotlin**, en **API 26** *(Oreo)*. Le projet se divise en plusieurs types de classes qui ont chacun leurs buts et fonctionnalités.



Voici les **classes Kotlins** du projet. On peut distinguer différents types de classes :

- Les **classes Activity**, toutes les classes contenant Activity dans leur nom
- Les **classes API**, avec ApiClient et ApiService
- Les **classes data**, comme Client, LoginRequest, Mail et surtout **FormData**
- La **classe FormViewModel** qui hérite de ViewModel
- La **classe MyApplication**

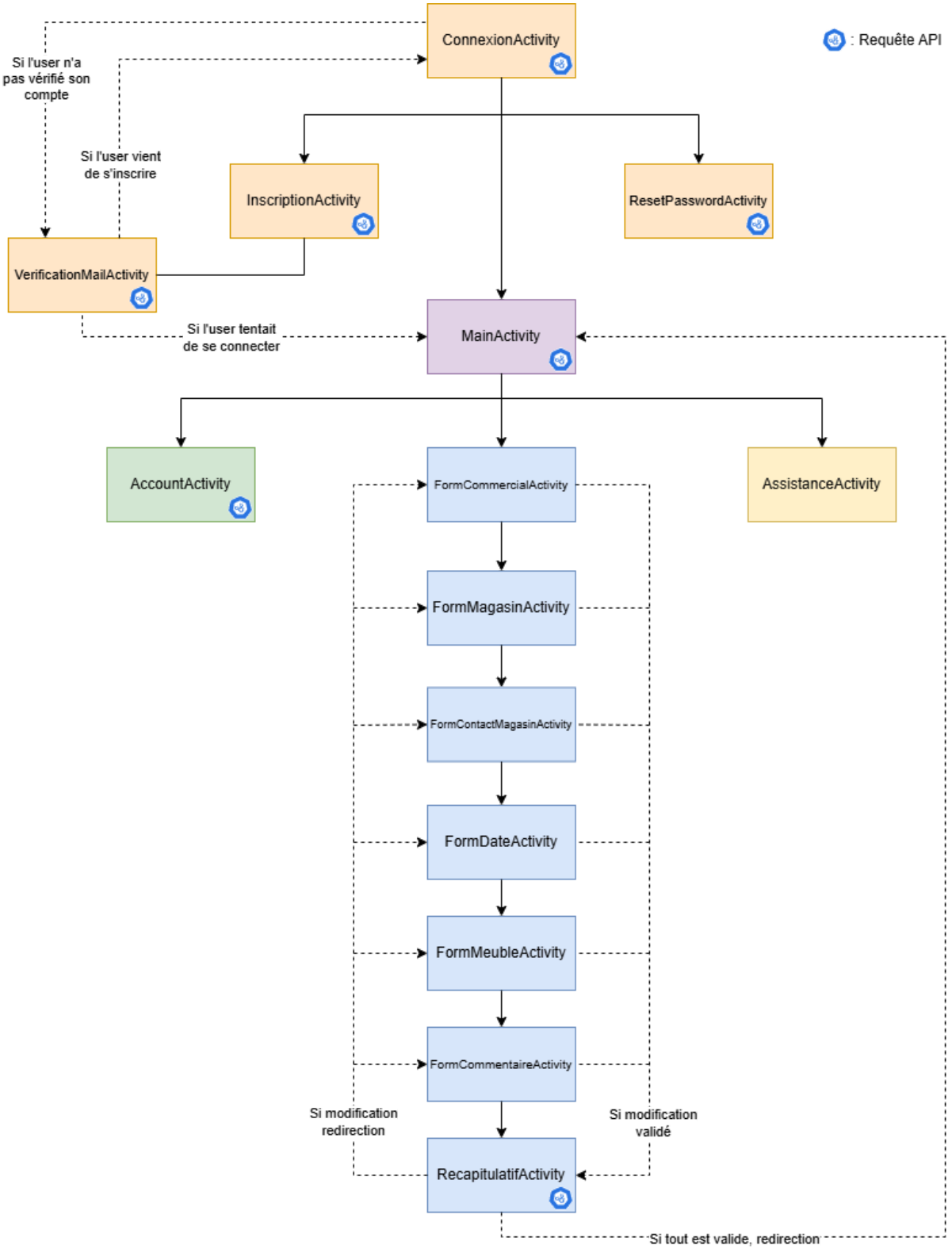
1. Classes Activity

Les **Activity** sont ce que l'on peut considérer comme les pages que l'utilisateur va visiter. Ces classes contiennent les fonctionnalités principales de l'application et avec un fichier **XML** associé on génère un affichage. L'utilisateur va donc naviguer d'Activity en Activity avec l'usage de bouton et de redirection via des **Intent** dans le code. Intent est un message ou une structure de données utilisée pour communiquer entre des composants d'une application ou entre différentes applications, donc en résumé de gérer la navigation dans l'application.

Comme dis plus haut, toutes les Activity contiennent des fonctions spécifiques pour mener leur but à bien. Mais elles ont aussi des fonctions communes, comme celles de vérification des saisies, ces dernières sont contenues dans **BaseActivity**. Cette Activity permet de définir des fonctions qui pourront être utilisées dans toutes les classes qui héritent de cette dernière. Ainsi, toutes les classes Activity héritent de BaseActivity. Avec leurs fonctions elles exploitent les données qu'elles vont récupérer via les **API** ou alors via les saisies de l'utilisateur.

Les Activity contiennent dans leurs noms l'indication de la page, de la fonctionnalité qu'elles remplissent :

- **Pour l'authentification et le compte :** ConnexionActivity, InscriptionActivity, AccountActivity, VerificationMailActivity, ResetPassordActivity.
- **Pour le formulaire :** Les Activity commencent par ''Form'' et RecapitulatifActivity
- **Pour la FAQ :** AssistanceActivity
- **Pour le menu principal :** MainActivity



2. Classes API

Les classes **ApiService** et **ApiClient** sont les deux classes qui vont permettre la connexion à l'API et son exploitation.

ApiClient est un objet qui configure et fournit une instance de **Retrofit** pour faire des **appels API**. De cette manière, toutes les Activity qui ont besoin de faire des appels ne recréent pas une instance à chaque requête mais utilisent celle-ci, pour de meilleures performances. Aussi, cela permet de centraliser la configuration, ce qui rend toute modification plus simple à réaliser.

Voici le code pour configurer l'instance Retrofit. On va chercher l'URL du domaine dans le **BuildConfig**, qui permet de rendre secret cette dernière. **L'intercepteur** va permettre de passer en header la **clé API**, elle aussi dans le BuildConfig, pour la comparer avec celle requise par l'API pour l'accès.

```
package com.example.myapplication

import okhttp3.Interceptor
import okhttp3.OkHttpClient
import retrofit2.Retrofit
import retrofit2.converter.gson.GsonConverterFactory

object ApiClient {
    private val BASE_URL = BuildConfig.API_BASE_URL
    private var retrofit: Retrofit? = null

    // Créer un Interceptor pour ajouter la clé API dans les en-têtes
    private val apiKeyInterceptor = Interceptor { chain ->
        // Ajouter l'en-tête "Authorization" avec la clé API
        val request = chain.request().newBuilder()
            .addHeader( name: "keyapi", BuildConfig.API_KEY)  // Utilisation de BuildConfig.API_KEY
            .build()
        chain.proceed(request)
    }

    // Créer un OkHttpClient avec l'intercepteur
    private val client = OkHttpClient.Builder()
        .addInterceptor(apiKeyInterceptor)  // Ajouter l'intercepteur
        .build()

    val instance: Retrofit
        get() {
            if (retrofit == null) {
                retrofit = Retrofit.Builder()
                    .baseUrl(BASE_URL)
                    .client(client)  // Utiliser notre OkHttpClient personnalisé
                    .addConverterFactory(GsonConverterFactory.create())
                    .build()
            }
            return retrofit!!
        }
}
```

```
interface ApiService {

    @GET("/api/clientAPI")
    fun getClientByMail(@Query("mail") mail: String): Call<List<Client>>

    @POST("/api/authAPI")
    fun verifyMdp(@Body requestBody: LoginRequest): Call<Boolean>

    @POST("/api/clientAPI")
    fun createClient(@Body client:Client): Call<Void>

    @PUT("/api/clientAPI")
    fun updateClient(@Body client:Client): Call<Void>

    @DELETE("/api/clientAPI")
    fun deleteClient(@Query("mail") mail: String): Call<Boolean>

    @POST("/api/mailAPI")
    fun sendMail(@Body mail:Mail): Call<JsonObject>

    @POST("/api/mailAPI")
    fun passwordResetMail(@Body mail:JsonObject): Call<JsonObject>

    @POST("/api/authAPI")
    fun verifyCode(@Body codeMail: JsonObject): Call<JsonObject>

    @PUT("/api/authAPI")
    fun resetPassword(@Body mailMdp: JsonObject): Call<JsonObject>

    @GET("/api/authAPI")
    fun isUserVerified(@Query("mail") mail: String): Call<JsonObject>
}
```

ApiService permet de définir une **interface** pour décrire les **appels API REST** avec **Retrofit**.

On y dresse la liste de toutes les requêtes avec leur **Endpoint** et leur **méthode** *(GET, POST, PUT, DELETE)*.

Toutes ces fonctions sont accessibles dans les **Activity**. Elles permettent aux Activity d'envoyer des requêtes à l'API.

Elles prennent en paramètre la plupart du temps un objet **JSON** contenant des données.

Seulement quelques Activity exécutent des requêtes. Voici un exemple de requête :

```
val apiService = ApiClient.instance.create(ApiService::class.java)
```

```
val mailJson = JsonObject().apply {
    addProperty( property: "mail", id)
}
val sendCode = apiService.passwordResetMail(mailJson)

sendCode.enqueue(object : Callback<JsonObject> {
    override fun onResponse(call: Call<JsonObject>, response: Response<JsonObject>) {
        if (response.isSuccessful) {
            val jsonResponse = response.body()
            val isSend = jsonResponse?.get("success")?.asBoolean ?: false
            val message = jsonResponse?.get("message")?.asString ?: ""
            if (isSend) {
                val intent = Intent( packageContext: this@ConnexionActivity, VerificationMailActivity::class.java)
                intent.flags = Intent.FLAG_ACTIVITY_NEW_TASK or Intent.FLAG_ACTIVITY_CLEAR_TASK
                intent.putExtra( name: "mail",id)
                intent.putExtra( name: "loginAction", value: true)
                startActivity(intent)
            } else {
                println("Mail non envoyé: $message")
            }
        } else {
            val errorBody = response.errorBody()?.string()
            println("Erreur HTTP: ${response.code()} - $errorBody")
        }
    }

    // En cas d'erreur réseau
    override fun onFailure(call: Call<JsonObject>, t: Throwable) {
        println("Network Failure: ${t.message}")
    }
})
```

3. Classes Data

Toutes ces données récoltées doivent être stockées en local pour permettre la complétion du formulaire. Pour se faire on utilise les **data class**. Certaines minimes comme Client, LoginRequest, Mail, mais la principale est **FormData**. FormData est déclarée dans **MyApplication** et permet donc à tout le projet d'y accéder.

Voilà à quoi ressemble FormData :

```kotlin
/**
    Va nous permettre de stocker les données nécessaires à la complétion du formulaire.
    De base toutes les valeurs sont null, elles seront complétées soit automatiquement par des
    requêtes à la base de données, ou entrées par l'utilisateur dans le formulaire.
 **/

data class FormData(
    val dateDevis: String = SimpleDateFormat( pattern: "dd/MM/yyyy", Locale.FRANCE).format(Date()),
    var nomInterlocuteur: String? = null,
    var prenomInterlocuteur: String? = null,
    var telephoneInterlocuteur: String? = null,
    var emailInterlocuteur: String? = null,
    var nomCommercial: String? = null,
    var prenomCommercial: String? = null,
    var telephoneCommercial: String? = null,
    var emailCommercial: String = "",
    var societe: String? = null,
    var nomMagasin: String? = null,
    var codePostal: String? = null,
    var ville: String? = null,
    var rue: String? = null,
    var nomContact: String? = null,
    var prenomContact: String? = null,
    var fonctionContact: String? = null,
    var telephoneContact: String? = null,
    var emailContact: String? = null,
    var dateDebut: String? = null,
    var dateFin: String? = null,
    var dateLivraison: String? = null,
    var dateRemise: String? = null,
    var meubles: MutableList<Meuble> = mutableListOf(),
    var commentaire: String? = null,
    var isEditing: Boolean? = false,
)
```

Déclarée ainsi dans MyApplication :

```kotlin
class MyApplication : Application() {
    lateinit var formViewModel: FormViewModel

    override fun onCreate() {
        super.onCreate()
        formViewModel = ViewModelProvider.AndroidViewModelFactory(applicationContext as MyApplication)
            .create(FormViewModel::class.java)

    }


}
```

On utilise la classe **FormViewModel** pour créer le FormData, et les fonctions associées qui permettront d’y modifier les datas stockées. Voici comment elle

se présente :

```
class FormViewModel : ViewModel() {
    private val _formData = MutableLiveData<FormData>(FormData()) // Initialise un objet vide
    val formData: LiveData<FormData> get() = _formData

    // Met à jour les données sans écraser les autres valeurs
    fun updateData(update: FormData.() -> Unit) {
        val updatedFormData = _formData.value?.copy()?.apply(update) ?: FormData().apply(update)
        _formData.value = updatedFormData // Affecte un nouvel objet pour déclencher l'update
        println("Données mises à jour : $updatedFormData") // Debug pour vérifier si ça fonctionne
    }

    // Réinitialise les données une fois le formulaire validé
    fun clearData() {
        _formData.value = FormData() // Réinitialisation propre
    }
}
```

On doit le déclarer dans les pages où l'on souhaite l'utiliser de la manière suivante :

```
private lateinit var formViewModel: FormViewModel
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    formViewModel = (application as MyApplication).formViewModel
```

Et on peut l'utiliser comme ceci par exemple :

```
// Mettre à jour le ViewModel avec les données du client
formViewModel.updateData{
    nomCommercial = client.nom
    prenomCommercial = client.prenom
    telephoneCommercial = client.telephone
    societe = client.societe
}
```

Ce **FormData** permet donc de stocker les informations requises pour la bonne construction des **demandes de devis** et du fonctionnement de l'application. Ces données sont supprimées à la fin du récapitulatif et non conservées.

B. API REST

Pour faire le pont entre **l'application** et **la base de données**, mais aussi le **SMTP**, une **API REST** est en place. Cette API est divisée en plusieurs parties :

- **Informations de configuration :** smtp.php, config.php, db.php, apiKey.php, token.php
- **Contrôleurs :** clientAPI.php, authAPI.php, mailAPI.php
- **Modèles :** Client.php, Auth.php, Mail.php, TokenManager.php
- **Middleware :** middleware.php

La structure divisée de **l'API** permet de garantir une meilleure organisation et aussi une meilleure **sécurité du code**. Chaque partie va gérer une fonctionnalité spécifique.

1. Les fichiers de configurations

Comme **l'API** doit communiquer avec la base de données et envoyer des mails, il faut la relier aux services associés. C'est pourquoi sont mis en place **trois fichiers de configuration : config.php** et **db.php** pour la base de données, puis **smtp.php** pour le **SMTP**.

Le fichier **config.php** se trouve en dehors du **dossier racine** et il contient toutes les informations nécessaires à l'API pour pouvoir se connecter à la base de données : l'host, l'utilisateur, le mot de passe et le nom de la base de données. Ce document, une fois importé dans le fichier de connexion à la base de données, va retourner les informations contenues. C'est dans le fichier **db.php**, qui se trouve dans le dossier api du domaine, qu'on va effectuer la connexion. Il se construit de la manière suivante :

```php
<?php
$config = include('/home/locaapc/config.php');
try {
    // Connexion base de données
    $pdo = new PDO(
        "mysql:host={$config['db']['host']};dbname={$config['db']['name']}",
        $config['db']['username'],
        $config['db']['password']
    );
    // Définir le mode d'erreur de PDO sur Exception
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
}
catch (PDOException $e) {
    die("Connection failed: " . $e->getMessage());
}
?>
```

Cette connexion, réalisée via **PDO**, permet d'assurer une certaine sécurité pour l'accès à la base de données. Ce point sera abordé plus en détail dans la section dédiée à la sécurité de l'API. La connexion sera ensuite importée et utilisée dans les contrôleurs.

Pour le **SMTP**, c'est le fichier **smtp.php**, lui aussi en dehors du dossier racine, qui contient les informations nécessaires pour assurer l'envoi de mail : l'host, l'utilisateur, le mot de passe et le port. Ce fichier sera ensuite importé dans le **contrôleur mailAPI** pour pouvoir l'utiliser. Voici comment l'importer et utiliser ses informations pour créer un **mailer** dans mailAPI :

```php
$smtp = include('/home/locaapc/smtp.php');

$mailer = new Mail($smtp['smtp']['host'], $smtp['smtp']['username'], $smtp['smtp']['password'], $smtp['smtp']['port']);
```

L'accès à l'API requiert une **clé API**. Cette clé est stockée dans **apiKey.php** et est en dehors du dossier racine aussi. Elle sert au **middleware** à comparer la

clé reçue dans la requête, pour s'assurer que la tentative de connexion est autorisée.

Le fichier **token.php** lui contient toutes les informations nécessaires à la création du token de validité de la session de l'utilisateur. Il contient la clé secrète, l'algorithme de chiffrage, son temps de validité et le domaine. Ces informations sont utilisées par **TokenManager.php** pour créer un token valide pour l'utilisateur qui se connecte.

2. Les contrôleurs

Les contrôleurs sont les fichiers **PHP** responsables de la gestion des requêtes. Chaque **contrôleur** reçoit des **requêtes** *(GET, POST, PUT, DELETE)* de l'application, les valide et les transmet aux modèles. Dans ce projet on en distingue trois : **clientAPI**, **AuthAPI**, **mailAPI**. Ils sont contenus dans le dossier api, lui-même dans le dossier racine.

La partie **clientAPI** va s'occuper de ce qui concerne surtout les **informations du compte**, la création, la suppression, la modification et la récupération d'informations. Elle va récupérer les requêtes **HTTP** et les transmettre au **modèle Client**. Dans ce fichier on importe la connexion à la base de données et le modèle Client, puis on instancie une **classe Client** pour récupérer les méthodes de cette dernière.

Ensuite **authAPI** va s'occuper de la partie authentification et vérification. Cette partie va récupérer les requêtes **HTTP** associées à par exemple la connexion, la vérification des codes uniques et les transmettre au **modèle Auth**. Ici on y importe aussi la connexion à la base de données et cette fois c'est le modèle Auth qu'on va importer et instancier.

Pour finir, **mailAPI** va s'occuper de la partie envoi de mail. Ce contrôleur va gérer la réception des requêtes **HTTP** associées à l'envoi de mail de vérification ou récapitulatif. Dans ce fichier on importe comme précédemment la connexion à la base de données et le modèle associé, ici **Mail**, que l'on instancie mais aussi le **SMTP**. Puis on donne les informations du SMTP en paramètre lors de l'instanciation de l'objet Mail, comme montré plus tôt.

Voici par exemple comment se compose un contrôleur, ici **authAPI.php** :

```php
<?php
require_once 'src/Auth.php';
require_once 'db.php';

$auth = new Auth();

$request_method = $_SERVER["REQUEST_METHOD"];
switch($request_method) {
    case 'GET':
        if (isset($_GET['mail']) && !isset($_GET['mdp'])) {
            $auth->isUserVerified($pdo, $_GET['mail']);
        }
        break;
    case 'POST':
        $data = json_decode(file_get_contents('php://input'), true);
        if (isset($data['mail']) && isset($data['mdp'])) {
            $auth->verifyMdp($pdo, $data['mail'], $data['mdp']);
        }
        elseif(isset($data['mail']) && isset($data['code'])) {
            $auth->verifyCode($pdo, $data['mail'], $data['code']);
        }
        break;

    case 'PUT':
        $data = json_decode(file_get_contents('php://input'),true);
        if(isset($data['mail']) && isset($data['mdp'])) {
            $auth->resetPassword($pdo, $data['mail'], $data['mdp']);
        }
        else {
            header("HTTP/1.0 400 Bad Request");
            echo "Mail manquant dans la requête";
        }
        break;
    default:
        header("HTTP/1.0 405 Method Not Allowed");
        break;
}
?>
```

3. Les modèles

Pour répondre aux **requêtes**, ce sont les **modèles** qui vont être sollicités. Dans ce projet il y en a quatre **: Client.php**, **Auth.php**, **Mail.php** et **TokenManager.php**. Ces derniers contiennent les fonctions qui vont permettre de réaliser le traitement attendu. Ils sont contenus dans le dossier src dans le dossier api.

Client.php va lui permettre de réaliser les traitements sur les comptes, par exemple créer en base de données ces derniers avec les informations reçues par le contrôleur.

Auth.php, lui s'occupe des traitements d'authentification comme vérifier si les logins renseignés sont valides, si l'utilisateur est vérifié.

Mail.php, va s'occuper d'envoyer les mails en utilisant le **SMTP**, mais c'est aussi lui qui va créer les codes de vérification et les mettre en base de données.

TokenManager.php permet de créer le token de validité de la session de l'utilisateur, mais aussi de le vérifier avant certains accès à l'API pour être sûr que la session est valide et authentique.

Ces fichiers sont ceux qui vont réaliser les traitements entre **l'API** et la **base de données**, là où les **contrôleurs** vont plus avoir le rôle de messagers.

Voilà par exemple un extrait du modèle **Client** :

```php
<?php
class Client {

    // Avoir les informations d'un client avec son mail
    function clientByMail($pdo, $mail) {
        $stmt = $pdo->prepare('SELECT mail, telephone, nom, prenom, societe FROM client WHERE mail = :mail');
        $stmt->execute(['mail' => $mail]);
        $row = $stmt->fetch(PDO::FETCH_ASSOC);

        header('Content-Type: application/json');
        http_response_code(200); // Toujours renvoyer 200 OK

        if (!$row) {
            echo json_encode(["found" => false]);
        } else {
            echo json_encode(["found" => true, "client" => $row], JSON_PRETTY_PRINT);
        }
        exit;
    }

    function isMailAvailable($pdo, $mail) {
        $stmt = $pdo->prepare('SELECT mail FROM client WHERE mail = :mail');
        $stmt->execute(['mail' => $mail]);
        $row = $stmt->fetch(PDO::FETCH_ASSOC);

        if (!$row) {
            return true;
        } else {
            header('Content-Type: application/json');
            echo json_encode(["success" => false, "error" => "Le mail est déjà utilisé."]);
        }
    }
```

V. Sécurité

A. Sécurité technique

Stocker des **données** implique une **base de données**, de ce fait, il est essentiel d'assurer une **protection robuste** à chaque couche du projet. Celui-ci étant composé de trois couches principales *(application mobile, API, base de données)*, il est nécessaire de sécuriser chacune d'elles pour garantir un niveau de **sécurité par défaut** le plus élevé possible.

B. Sécurisation des accès

Un point majeur de la sécurisation des données est de s'assurer que les accès soient **filtrés**, **sécurisés**, **contrôlés**. C'est pourquoi une **API** est en place, elle permet de séparer le code de l'application et les requêtes sur la base de données. Mettre directement l'accès à la base de données dans l'application pourrait représenter une **faille de sécurité importante**.

Cette API contient les fonctions pour les requêtes et les accès à la base de données nécessaires pour effectuer des actions **GET**, **DELETE**, **POST** et **PUT**. L'accès à la base de données n'est pas dans l'API même, comme le **SMTP**, car sinon les accès pourraient être dérobés.

Pour sécuriser encore plus l'accès à la base de données, le fichier contenant les informations de connexion se trouve en dehors du dossier racine. De ce fait, il n'est pas accessible sur le web et permet donc de s'assurer que ces informations restent secrètes et cachées à la vue de tous.

Pour autoriser un accès à l'API, il faut que l'utilisateur ait la bonne **clé API**. La clé API est contenue dans le fichier **apiKey.php** en dehors du dossier racine du domaine. Ensuite, ce fichier est importé dans le fichier **middleware.php** contenu dans le dossier src du dossier api. Ce dernier est importé dans les trois **contrôleurs** et permet de comparer la clé renseignée dans le **header de la requête** et celle requise pour avoir une réponse. Ainsi seules les personnes possédant la clé peuvent accéder aux requêtes de l'API. Voici comment se compose le **middleware** :

```php
<?php
$secureConfig = include('/home/locaapc/apiKey.php'); // Import ici

function checkApiKey() {
    global $secureConfig; // Utilisation de la variable globale pour l'API key
    $headers = array_change_key_case(getallheaders(), CASE_LOWER); // Forcer en minuscule

    if (!isset($headers['keyapi'])) {
        http_response_code(403); // Interdit
        echo json_encode(["success" => false, "error" => "Header incorrect", "headers" => $headers]);
        exit;
    }

    if ($headers['keyapi'] !== $secureConfig['API_SECRET_KEY']) {
        http_response_code(403); // Interdit
        echo json_encode(["success" => false, "error" => "API incorrecte", "header_keyapi" => $headers['keyapi']]);
        exit;
    }
}
```

Dans la même idée, pour les fonctions comme l'envoi de mail après le formulaire, la récupération des données clients et d'autres actions sur le compte, l'utilisateur doit avoir un **token valide** pour sa session en cours. Ce **token** est créé à sa connexion avec **TokenManager.php** en utilisant les informations de **token.php**. Il est transmis à l'appli et est vérifié avant les requêtes concernées. De ce fait on s'assure que seuls les utilisateurs autorisés peuvent accéder à certaines des fonctionnalités. Le token est stocké dans les **SharedPreferences** sur l'appli lors de sa récupération.

```kotlin
val token = jsonToken?.get("token")?.asString ?: "Pas trouvé à la connexion"
val sharedPreferences = getSharedPreferences( name: "MyAppPrefs", Context.MODE_PRIVATE)
sharedPreferences.edit().putString("Token", token).apply()
```

Pour assurer une **sécurité optimale** de l'API, deux fichiers **.htaccess** sont utilisés, chacun jouant un rôle spécifique. Le premier fichier, placé dans le **dossier racine**, permet de forcer l'utilisation de **HTTPS** pour toutes les connexions entrantes. Il active également des **en-têtes HTTP** sécuritaires pour prévenir plusieurs types d'attaques, tels que le **clickjacking**, **l'injection de contenu** et les **attaques XSS**. Ces en-têtes incluent **X-Frame-Options** pour interdire l'affichage du site dans des cadres, **X-Content-Type-Options** pour empêcher la détection automatique du type de contenu, et **Strict-Transport-Security** pour exiger des connexions sécurisées. De plus, une politique de sécurité du contenu *(Content-Security-Policy)* est définie pour limiter les sources de contenu autorisées, renforçant ainsi la protection contre l'injection de scripts malveillants.

```
RewriteEngine On
RewriteCond %{ENV:HTTPS} !on
RewriteRule (.*) https://%{HTTP_HOST}%{REQUEST_URI} [L,R=301]

# Sécuriser les en-têtes HTTP pour éviter certaines attaques
Header always append X-Frame-Options SAMEORIGIN
Header set X-Content-Type-Options "nosniff"
Header always set Strict-Transport-Security "max-age=31536000; includeSubDomains; preload"
Header set X-XSS-Protection "1; mode=block"
Header set Content-Security-Policy "default-src 'self'; script-src 'self'; object-src 'none'; style-src 'self';"
```

Le second fichier **.htaccess**, situé dans le dossier api, restreint l'accès aux seuls fichiers **PHP** nécessaires au bon fonctionnement de l'API, à savoir **authAPI.php**, **clientAPI.php** et **mailAPI.php**. Tous les autres fichiers PHP dans ce dossier sont explicitement bloqués pour empêcher un accès non autorisé. Enfin, l'indexation des répertoires est désactivée pour éviter que les utilisateurs n'accèdent à une liste de fichiers dans le dossier API.

```
# Bloquer l'accès direct à tous les autres fichiers PHP dans le dossier API
<FilesMatch "(?!authAPI\.php|clientAPI\.php|mailAPI\.php).*\.php$">
    Require all denied
</FilesMatch>

# Permettre l'accès aux fichiers API spécifiques uniquement (authAPI, clientAPI, mailAPI)
<FilesMatch "(authAPI\.php|clientAPI\.php|mailAPI\.php)">
    Require all granted
</FilesMatch>

# Empêcher l'indexation des répertoires
Options -Indexes
```

Pour se connecter à l'API, l'application doit utiliser l'URL du domaine qui l'héberge. Cependant, il est essentiel de ne pas exposer cette URL directement dans le code, afin d'éviter qu'un utilisateur malveillant ne puisse la récupérer en décompilant l'application. Pour cela, l'URL de l'API est stockée dans le fichier **api.properties** et importé dans **BuildConfig** pour être ensuite utiliser dans le code. Ce qui permet de définir des variables accessibles uniquement lors de la compilation de l'application. Ce fichier n'est pas inclus dans l'APK final visible par l'utilisateur. L'URL est ensuite récupérée dynamiquement dans l'application via la configuration **DefaultConfig**, ce qui empêche son extraction via des outils de **rétro-ingénierie**. La même logique est appliquée à la clé API de l'application.

L'ensemble de ces mesures vise à renforcer la sécurité de l'API en protégeant l'accès à la base de données, en sécurisant les échanges via **HTTPS**, en restreignant les accès aux fichiers critiques et en empêchant l'exposition

d’informations sensibles. Ces bonnes pratiques réduisent considérablement les risques d’attaques et garantissent une meilleure fiabilité du service.

C. Sécurisation des saisies

Les accès étant **sécurisés**, il faut aussi s’assurer que les informations soient valides. De ce fait, il existe diverses vérifications dans l’appli même et dans l’API.

Du côté de l’appli, des fonctions sont mises en place pour prémunir les utilisateurs d’entrer des valeurs non désirées aux champs des formulaires. On vérifie par exemple que les mails soient **valides**, que tous les champs soient **remplis** et **valides**, on vérifie aussi que les caractères spéciaux pouvant présager une **injection SQL** ne soient pas présents dans les champs. De ce fait aucune valeur non conforme ne peut être envoyée, aucune valeur indésirable peut être envoyée, les injections SQL sont écartées.

Pour se prémunir encore plus des injections SQL et des attaques, les API utilisent l’extension PHP **PDO** *(PHP Data Objects)*. Cette extension permet d’accéder aux bases de données de manière **sécurisée**, notamment avec la méthode de **préparation des requêtes**. Cette méthode permet de préparer les requêtes SQL avec des paramètres liés, ce qui aide à se prémunir des injections SQL.

Ainsi avec une double couche de sécurité sur les saisies des utilisateurs, on garantit la sécurité des données face aux injections de SQL et de script.

D. Sécurisation des comptes

Pour garantir la **sécurité des données**, est mis en place une validation des comptes utilisateurs. Avant de pouvoir accéder à l'application, chaque utilisateur doit valider son compte en entrant un code unique envoyé à l'adresse e-mail fournie lors de l'inscription. Cette étape assure que l'adresse e-mail est **valide** et **accessible**, ce qui est essentiel pour les communications futures. De plus, cette validation aide à prévenir la création de comptes fictifs et renforce la sécurité en limitant les tentatives d'accès non autorisé. En intégrant cette mesure, nous protégeons les données sensibles de nos utilisateurs et établissons une relation de confiance dès le départ.

Aussi, avant de mettre le mot de passe en base de données on prend soin de le **hacher** du côté de l’API. On ne stocke pas le mot de passe des utilisateurs en clair dans la base de données. Le langage PHP fourni directement des solutions pour hacher les mots de passe et aussi des fonctions pour procéder à leur

vérification quand on doit s'assurer de la validité et conformité de ces derniers, en cas de login par exemple.

De plus, lors de la création du mot de passe, l'application oblige l'utilisateur à créer un **mot de passe sécurisé**, en accord avec les indications de la **CNIL**. C'est-à-dire au moins **13 caractères**, au moins **1 majuscule**, au moins **1 minuscule**, au moins **1 chiffre** et au moins **1 caractère spécial**.

Ainsi avec cette sécurité sur les comptes utilisateurs, on prémunit l'application de comptes fictifs mais aussi on sécurise l'accès aux comptes des clients.

VI. Conformité à la RGPD

L'application **traite** des informations personnelles relatives aux clients en base de données. De ce fait il faut absolument respecter les **réglementations** en vigueur concernant ces traitements.

A. Consentement et transparence

Lorsque l'on **collecte** et **traite** des **données personnelles** sur l'utilisateur, il doit être au courant de la manière dont ses données vont être utilisées, les **finalités** de la collecte et leurs **droits** en matière de **protection des données**. Une fois toutes ces informations présentées à lui, il doit exprimer librement son consentement **clair** et **explicite** pour permettre à l'entreprise de commencer le traitement des données de ce dernier.

Il faudrait donc rédiger des conditions générales et les politiques de traitement des données pour l'application et demander aux clients de la lire et décider de s'ils veulent oui ou non accepter ce traitement de données.

B. Collecte conforme des données

Les **données collectées** et **stockées** doivent être scrupuleusement choisies en accord avec la finalité du traitement. Ne doivent pas être collectées et traitées les données inutiles au but de notre solution. Dans le cas de **Locapp**, doivent être collectées les données relatives au client qui permettent de l'identifier directement lui et sa société affiliée ainsi que les personnes associées à cette prise d'accord.

Voici un tableau récapitulant les données collectées via nos formulaires, en prenant soin de séparer les données prises en base de données et celles uniquement présentes sur les PDF finaux :

Données stockées en base de données	Données uniquement sur le PDF final
Informations sur le commercial : • Nom • Prénom • Mail • Téléphone • Société	**Informations du contact en magasin :** • Nom • Prénom • Mail • Téléphone **Informations sur la commande :** • Date de livraison et de retour • Date de début et fin de location • Le type de meuble • Les dimensions • Frais ou surgelé • Ouvert ou fermé • PLV ou non et si oui sa référence *Informations supplémentaires via un champ de texte, si le client souhaite ajouter des précisions sur sa commande*

Les données collectées ici sont toutes nécessaires à la bonne complétion de l'objectif de l'application : faire une demande de devis. Aucune information non liée à la finalité du traitement n'est collectée.

La **RGPD** prévoit aussi limiter le stockage des données dans le temps. Les données doivent être conservées uniquement pendant la durée nécessaire à la réalisation des finalités pour lesquelles elles sont traitées. Ici, dans la base de données sera fait tous les mois une vérification de la dernière date de connexion des clients. Au bout d'un mois consécutif de non-connexion, l'utilisateur sera prévenu que lors de la prochaine vérification de la base de données, ses données seront entièrement supprimées s'il ne se connecte pas sur l'application dans le temps imparti. Ainsi on ne conserve pas trop les données, on allège le stockage, mais aussi on respecte la durée de stockage des données personnelles.

C. Droit des personnes

Les clients ont le droit à **l'accès** de leurs données, de **rectification**, **d'effacement**, de **portabilité**, **d'opposition** et de **limitation** du traitement de leurs données, et ce à n'importe quel moment.

C'est pourquoi l'application permet à l'utilisateur de modifier les informations de leur compte, mais aussi de le supprimer totalement de notre base de données.

Il peut aussi contacter directement le support de l'application et demander la suppression totale de ses informations comme les prises d'accords envoyées par mail. Mais aussi pour demander d'assurer potentiellement la portabilité de ses données et de s'opposer au traitement de ses données.

VII. Pistes d'améliorations

A. Securite

Bien que le projet soit déjà sécurisé selon les bonnes pratiques actuelles, certaines améliorations pourraient être explorées à l'avenir pour renforcer encore la protection des données et des accès :

- Mise en place d'un système de **limitation des requêtes** *(Rate Limiting)* pour éviter les abus sur l'API.
- Ajouts de **logs et de surveillance** des accès pour détecter d'éventuels comportements suspects.
- **Expiration** et **renouvellement** automatique des **tokens** pour une meilleure gestion des sessions.
- **Blocage temporaire** des comptes après plusieurs échecs de connexion afin de contrer les **attaques par brute force**.
- **Désactivation automatique** des comptes **inactifs** après une longue période sans connexion.
- Ajout d'une vérification du numéro de téléphone pour vérifier les comptes au lieu de l'adresse mail.

B. Base de données

Pour la base de données, voici quelques pistes d'amélioration ou ajout pour d'autres fonctionnalités :

- Ajout d'un **script mensuel** pour procéder à la vérification des comptes inactifs.
- Ajout d'un champ ''lastLogin'' dans la table client, contenant le timestamp de la dernière connexion d'un utilisateur.

C. Application

L’application actuellement est déjà fonctionnelle et accomplie le but pour lequel elle a été pensé. Néanmoins, certaines améliorations peuvent être apportées :

- Vider le stockage des devis enregistrés par l’appli au bout d’un certain temps ou avec un bouton pour l’utilisateur, pour libérer de l’espace.
- Faire en sorte que l’appli créé un dossier spécial pour les prises d’accord, et pour chaque date, pour faciliter la recherche des devis par l’utilisateur.
- Trouver un meilleur nommage des prises d’accord pour éviter que le plus récent écrase le dernier à la même date.
- Retravailler l’interface.
- Ajouter une fonctionnalité pour permettre à l’utilisateur de rester connecter à l’appli en choisissant de le faire à la connexion.

VIII. Conclusion

Avec ce stage j’ai grandement appris sur la structure d’un projet de tout ce qui en incombe du fait de ma gestion quasi totale du projet et des technos utilisées. Je retiens surtout mon travail sur l’API et la sécurisation de tout ce qui en ressort. Bien que je pense que tout peut être grandement amélioré, notamment car je n’avais pas de contact direct avec des développeurs ce qui peut être un peu pénalisant. Mais dans la globalité je reste très satisfait de mon travail, d’avoir pu délivrer le projet dans un état convenable et fonctionnel.