Institution des Chartreux

PAPPE Lourd

Epreuve E5

ADAM Emilien
[Date]

Contexte

L'entreprise pharmaceutique Galaxy Swiss Bourdin (GSB) est le leader de ce secteur industriel. Suite à la fusion avec le géant américain Galaxay le groupe chercher à optimiser son activité dans ses laboratoires. Pour cela un nouveau service a été créé pour l'entreprise.

Application

Description

L'application est une solution de gestion de livraison pour les différents livreurs qui s'occupent de la livraison de médicaments produits en laboratoire. La solution est une application mobile pour téléphones Android.

Technologies

Cette application est donc disponible sur les appareils Android et a été développée en Kotlin. Pour ce qui est de l'API connecté à la base de données, elle a été développée en python avec la librairie Flask.

Accès au code

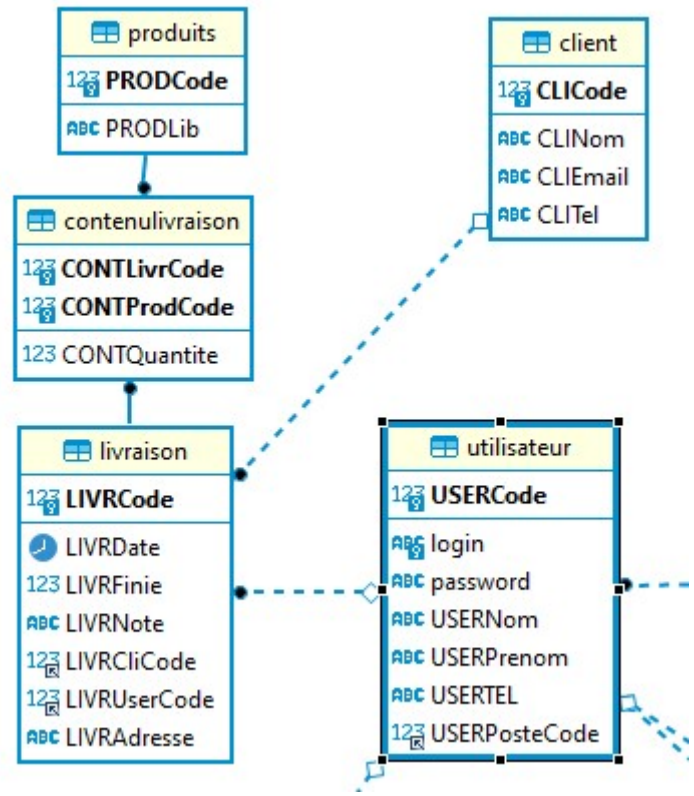
https://gitlab.com/sco-chartreux/slam-22-23/solo/adam-emilien/gsb_leger

https://gitlab.com/sco-chartreux/slam-22-23/solo/adam-emilien/gsb_api

Fonctionnalités

Base de données

La base de données de l'application est une base de données MariaDB composée de 5 tables. Une table utilisateur qui contient les comptes des livreurs, une table client qui contient tous les clients du laboratoire et enfin les livraisons et leur contenu.

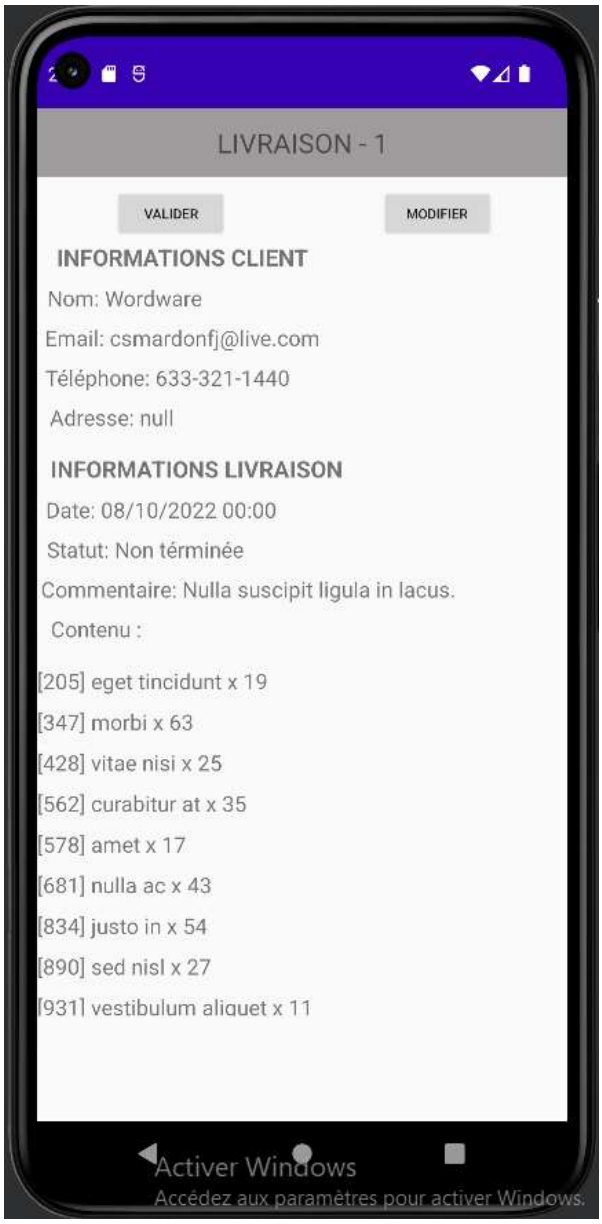


Gestion des livraisons

Tout d'abord l'utilisateur arrive sur une page de connexion où il doit rentrer ses identifiants pour accéder à l'application.

Le livreur arrive ensuite sur une page présentant toutes les livraisons qu'il doit réaliser avec les informations importantes de la livraison.

En haut de l'application apparait le nom et prénom du livreur. En haut apparait également un bouton qui permet d'afficher toutes les livraisons passées du livreur, c'est-à-dire celles qu'il a validé une fois terminée. Que ce soit une livraison passée ou actuelle, l'utilisateur peut cliquer dessus pour accéder au détail plus complet de la livraison choisie.



Depuis cette page, l'utilisateur peut alors voir le contenu entier des produits de la livraison au format [ID PRODUIT] NomProduit x Quantité. L'utilisateur voit également l'adresse, le téléphone et l'email du client qui attend la livraison. En cliquant sur un de ces éléments, cela ouvre l'application d'exploitation de ces données (Téléphone -> Application téléphone avec le bon numéro, Adresse -> Google Maps…). L'utilisateur peut enfin utiliser deux boutons. Le premier, Valider, valide la livraison une fois terminée, elle passe alors dans les livraisons passées et l'utilisateur est ramené à la page précédente. Le bouton modifier lui amène sur l'activité de modification d'une livraison.

API

Pour communiquer avec la base donnée, le choix à été d'utiliser une API, dans le cas où l'application est fermée abruptement par l'utilisateur, il n'y a alors pas de perte de données non sauvegardée. L'API ne fera que de la simple récupération de données de depuis la base de données, une simple API en python avec Flask sera assez rapide et complète. L'arborescence de l'API est simple, le fichier du server web, un singleton et un DAO pour gérer la base de données.

```python
class MariaDBSingleton:
    _instance = None

    def __new__(cls):
        if not cls._instance:
            cls._instance = super().__new__(cls)
        return cls._instance

    def __init__(self):
        if not hasattr(self, 'connection'):
            self.connection = mariadb.connect(
                host="192.168.1.15",
                user="gsb_Access",
                password="cIZHq8QBX5",
                database="GSB_DB"
            )

    def query(self, query):
        cursor = self.connection.cursor()
        cursor.execute(query)

        return cursor.fetchall()

    def execute_query(self, query):
        cursor = self.connection.cursor()
        cursor.execute(query)
        self.connection.commit()
        cursor.close()
```

Le Singleton permet d'instancier une seule et unique connexion à la base de données et de réaliser des requêtes envers cette dernière. Deux méthodes sont présente pour réaliser des requêtes selon le résultat attendu de la requête SQL sur la base de données.

```python
class UserDAO:

    def __init__(self):
        self.db = MariaDBSingleton()

    def find_by_login(self, login):

        query = f"""SELECT * FROM UTILISATEUR u, POSTE p
                    WHERE u.USERPosteCode = p.POSTECode
                    AND u.login like '{login}';
                """
        result = self.db.query(query)[0]

        column_names = ['USERCode', 'login', 'password', 'USERNom', 'USERPrenom', 'USERTEL',

        json_data = json.dumps(dict(zip(column_names, result)))

        return json_data
```

Le DAO lui permet de réaliser les requêtes au près de la base données à l'aide différentes méthodes.

Enfin, on fait appel dans le fichier Flask à ces méthodes du DAO avec les données passées dans la requête vers notre API.

```python
@app.route('/findLivraison/', methods=["POST"])
def findLivraison():
    id = request.form.get('id')
    return DAO.livraisons_en_cours(id)
```

Tout les routes de l'API nécessitant de passer des paramètres sont de type 'POST' afin que l'API puisse être adaptée à une utilisation web basée sur des formulaires. Enfin, le DAO retourne toutes les données au format JSON, format fréquent des retours d'API et schéma de données facilitant l'utilisation des ces dernières.

```
livraisons = []
for result in results:
    livraison = {
        "LIVRCode": result[0],
        "LIVRDate": result[1],
        "LIVRFinie": result[2],
        "LIVRNote": result[3],
        "LIVRCliCode": result[4],
        "LIVRAdresse":result[6],
        "client": {
            "CLICode": result[7],
            "CLINom": result[8],
            "CLIEmail": result[9],
            "CLITel": result[10]
        },
        "contenuList": []
    }
```

Pour ce qui est du côté d'Android, pour se connecter à l'API, l'application utilise Retrofit. Retrofit permet de se connecter, faire des requêtes et traiter des données depuis l'API de notre choix. Pour initier cette connexion, on utilise également un Singleton :

```
7   object singletonAPI {
8
9       private var retrofit: Retrofit? = null
10
        EmilienAdam *
11      fun getInstance(): Retrofit {
12          if (retrofit == null) {
13              retrofit = Retrofit.Builder() Retrofit.Builder
14                  .baseUrl("http://51.83.42.249:5000") Retrofit.Builder
15                  .addConverterFactory(GsonConverterFactory.create())
16                  .build()
17          }
18          return retrofit!!
19      }
20  }
```

Le GsonConverterFactory permet de convertir les données JSON reçues directement vers les modèles de données de notre application. Exemple de modèle de données :

```kotlin
data class Livraison(

    val LIVRCode: Int,
    val LIVRDate: Date,
    val LIVRFinie: Number,
    val LIVRNote: String,
    val LIVRCliCode: Int,
    val LIVRUserCode: Int,
    val LIVRAdresse: String,
    var client: Client ?,
    var contenuList: List<ContenuLivraison>

) : Serializable
```

Ensuite, pour faire des requêtes vers Retrofit il faut lister les routes et méthodes à utiliser pour ces appels :

```kotlin
interface API {

    EmilienAdam
    @POST("/login/")
    @FormUrlEncoded
    fun getUserByLogin(@Field("login") login: String, @Field("password") password: String): Call<Utilisateurs>

    EmilienAdam
    @POST("/findLivraison/")
    @FormUrlEncoded
    fun getLivById(@Field("id") id: Int): Call<List<Livraison>>

    EmilienAdam
    @POST("/findLivraisonFinie/")
    @FormUrlEncoded
    fun getLivFinie(@Field("id") id: Int): Call<List<Livraison>>

    EmilienAdam
    @POST("/validateLivr/")
    @FormUrlEncoded
    fun validateLivr(@Field("id") id: Int): Call<String>

    new *
    @POST("/editLivr/")
    @FormUrlEncoded
    fun editLivr(@Field("id") id: Int, @Field("date") date: Date, @Field("note") note: String): Call<String>
}
```

Les méthodes réutilisent donc ici le POST et transmettent les données comme si elles provenaient d'un formulaire HTML. Le retour des données est automatiquement converti dans un objet du type que l'on chercher, utilisateur par exemple.

Enfin, pour faire les appels à l'API on instancie l'API avec Retrofit :

```kotlin
val retrofit = singletonAPI.getInstance()
```

Puis on appelle la méthode voulue avec les données nécessaires puis on agit selon le résultat obtenu par notre requête.

```
val API = retrofit.create(API::class.java)

val request = API.getUserByLogin(textLogin, textPassword)

request.enqueue(object: Callback<Utilisateurs> {
```

On peut alors récupérer le résultat dans une variable :

```
override fun onResponse(call: Call<Utilisateurs>, response: Response<Utilisateurs>) {
    val user = response.body()
```

Authentification

Changement d'Activité

Le changement d'activité est une partie importante d'une application Android, pour assurer sa fluidité et qu'elle ne crash pas. Dans l'application, les activités sont réutilisées plusieurs fois à différents moments, il faut donc bien gérer leur fin le transfert de données.

Tout d'abord, lorsque l'on crée une nouvelle activité on utilise la méthode onCreate(), qui est executé quand l'activité est créée avec la méthode startActivity()

```
val homeIntent = Intent( packageContext: this@MainActivity, ActivityHome::class.java)
homeIntent.putExtra( name: "user", user)
startActivity(homeIntent)
```

La ligne putExtra permet de rajouter des données que l'on veut passer l'activité suivante. Récupération des données dans une autre activité :

```
val user : Utilisateurs = intent.getSerializableExtra( name: "user") as Utilisateurs
```

Lorsque l'on passe à une autre activité, l'activité précédente est donc mise en état de 'pause' par Android. Dans le cas d'un appuie sur une activité de la page principale, on accède à son contenu. Alors, l'activité qui liste toutes les livraisons est mise en pause et celle de contenu est affichée. Une fois que l'utilisateur a appuyé sur les boutons de validation, alors l'activité est terminée est la précédente est 'resumed' (repise). On recharger alors les livraisons avec une nouvelle requête API pour que celle validée disparaisse et les données des livraisons soient mises à jour.

```
27    override fun onCreate(savedInstanceState: Bundle?) {...}
128
      new *
129   override fun onResume() {...}
```

Futur de l'application

A terme l'application devrait pouvoir présenter une interface spéciale pour les administrateurs, interface qui permettrait de voir l'intégralité des livraisons à venir, les modifier, supprimer, valider et enfin pouvoir en créer.