Documentation technique de l’application de gestion des palettes

Table des matières

Introduction

Contexte du projet

Une application utilisée en production devait être reprogrammée entièrement dû à son obsolescence. Elle était codée avec Delphi, qui posait aujourd'hui de nombreux problèmes, comme notamment une instabilité de l'application, des problèmes de sécurité, mais surtout l'impossibilité de maintenir une telle technologie à jour. Il était alors nécessaire de reprogrammer l'application, le frontend comme le backend, en utilisant des technologies plus récentes afin de pallier ces différentes problématiques.

Objectifs du projet

L'objectif était alors de créer une application mobile Android stable, sécurisée, et maintenable, basée sur l'ancienne application, et répondant aux nouveaux besoins utilisateurs. Elle comprend trois fonctionnalités majeures : rechercher l'emplacement d'une palette, modifier l'emplacement d'une palette, et rechercher les palettes d'un client.

En parallèle, la partie backend était également à refaire, et a donc demandé la réalisation d'une API REST faisant le lien directement entre une base de données sur serveur et l'application mobile via une connexion HTTPS.

Technologies utilisées

Android Studio

Ce projet a été entièrement réalisé sous Android Studio qui est l'IDE officiel de développement d'applications Android. Il contient tout le nécessaire pour le développement d'une telle application, de l'écriture de code aux tests, et jusqu'au déploiement.

Kotlin

Le backend comme le frontend ont été développés en Kotlin, qui est un langage de programmation moderne entièrement compatible avec Java. Il est bien plus sécurisé et notamment simple d'utilisation que ce dernier, permettant ainsi une programmation bien plus productive.

Spring

Concernant le backend, l'application utilise Spring qui est un framework puissant mais léger, très répandu dans le développement Java comme Kotlin, et permettant un développement plus productif et surtout maintenable. Sa modularité permet également de n'utiliser que les composants nécessaires, comme dans notre cas, l'utilisation de modules liés aux bases de données. Cette technologie nous permet donc d'échanger avec notre base de données pour renvoyer au client les informations qu'il a requêtées.

Retrofit + Gson

Concernant le frontend, l'application utilise Retrofit qui est une librairie permettant d'envoyer des requêtes HTTP au serveur et de récupérer ses réponses. Il peut facilement se coupler avec Gson qui permet de convertir les réponses interceptées par Retrofit directement en objets de données.

Gradle

De plus, le projet utilise Gradle comme outil de build, fournissant divers fichiers de build permettant l'entière configuration des projets. C'est là où les langages, dépendances, et autres configurations sont explicitées, afin que Gradle s'assure de leur bonne synchronisation. C'est également lui qui va se charger de générer un fichier JAR par exemple du projet compilé et optimisé.

NSSM

Enfin, afin de créer un service Windows à partir du backend généré en un fichier JAR, l'outil NSSM a été utilisé. Il permet de gérer un service dans sa globalité et a permis, dans notre cas, de créer un service à partir de notre fichier JAR de backend.

Architecture système

Description des composants

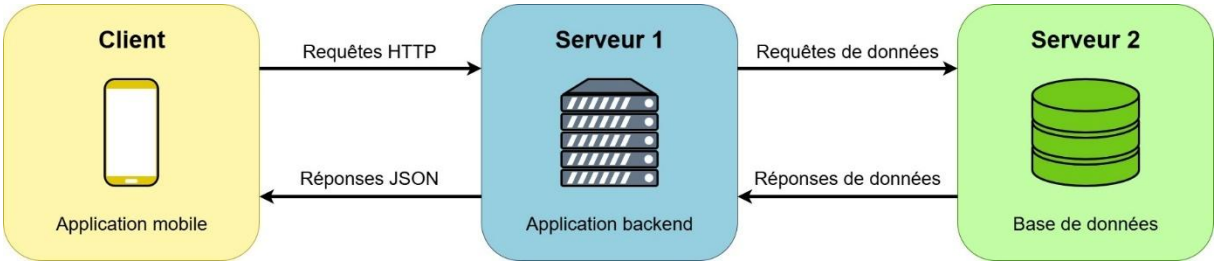
Le projet fonctionne grâce à trois composants notables : l'application mobile disponible sur les terminaux des utilisateurs, l'application backend hébergée sur le serveur 1, ainsi que la base de données présente sur le serveur 2. Contrairement à la précédente version de l'application, celle-ci ne met plus directement en relation le client et la base de données, ce qui permet la séparation des responsabilités mais également la garantie d'une meilleure sécurité.

La partie client constitue l'application mobile téléchargée sur les terminaux des utilisateurs. C'est la seule partie du système leur étant accessible, ils n'ont aucune visibilité sur le système des deux serveurs utilisés derrière. C'est cette partie qui va s'occuper de requêter le serveur backend (via des requêtes HTTP émises grâce à la librairie Retrofit) en fonction des informations demandées par le client : soit la recherche d'une palette, son changement d'emplacement, ou la recherche des palettes d'un client. Il va ensuite récupérer la réponse du serveur et traiter ces données en les affichant à l'utilisateur par exemple.

La partie backend est hébergée sur le serveur 1 et est invisible aux utilisateurs. C'est elle qui va intercepter les requêtes HTTP des utilisateurs clients, les lire, et requêter le serveur de bases de données en fonction des informations demandées par l'utilisateur. Une fois les données demandées récupérées, le backend va transmettre à l'utilisateur ces données reçues en format JSON.

La partie base de données, présente sur le serveur 2, va s'occuper de traiter la demande du serveur backend pour lui fournir les informations dont il a besoin. Cette partie ne communique donc jamais directement avec la partie client.

Schéma de l'architecture



Serveur backend

Rôle de l'application

La partie backend du projet va permettre d'intercepter les requêtes HTTP émises par les clients et demander au serveur de bases de données les informations dont il a besoin pour répondre aux demandes des clients. Après réception d'une réponse, il transmettra les informations aux clients sous la forme d'une réponse JSON.

Structure de l'application

Le backend est organisé en plusieurs packages : *entity*, *repository*, *service*, *controller*, *dto*, regroupant chacun des fichiers leur étant propres. Ces différents fichiers vont permettre de gérer la totalité des requêtes émises : de la réception de ces requêtes jusqu'à la retransmission des données souhaitées vers l'application client.

D'autres fichiers non pas moins importants sont également présents, comme ceux du package *whitelist* permettant le whitelisting d'adresses IP, le fichier de configuration *application.properties*, ou encore le certificat auto-signé utile aux requêtes HTTPS *server.p12*.

Entity

Le package *entity* va contenir des "Entity" qui sont une traduction des tables de la base de données en classes de données Kotlin. Un point positif de cet élément est qu'on peut ne renseigner que les champs que l'on souhaite manipuler. Par exemple, la table PALETTE de la base de données possède une cinquantaine de colonnes, alors que l'application n'en nécessite que six :

```kotlin
@Entity
@Table(name = "PALETTE")
// Classe de données traduisant la table PALETTE dans la base de données
data class PaletteEntity(
    @Id
    @Column(name = "Num")
    val num: Int = 0,

    @ManyToOne(optional = true)
    @JoinColumn(name = "Nume_Empl", referencedColumnName = "Num")
    var numeEmpl: EmplacementEntity? = null,

    @ManyToOne(optional = true)
    @JoinColumn(name = "Nume_Anci_Empl", referencedColumnName = "Num")
    var numeAnciEmpl: EmplacementEntity? = null,

    @Column(name = "Nume_Exte")
    val numePalCli: String? = "0",

    @ManyToOne(optional = true)
    @JoinColumn(name = "Nume_Deta_Rece", referencedColumnName = "Num")
    val numeDetaRece: DetaReceEntity? = null,

    @Column(name = "Poid_Brut_Expe")
    val poidBrutExpe: Float? = 0F
)
```

Repository

Le package *repository* contient des "Repository" qui sont des interfaces contenant des fonctions qui permettent de requêter la base de données et de stocker les informations récupérées dans des Entity. À noter que les requêtes sont écrites de base en JPQL mais peuvent également être écrites en SQL.

```
@Repository
// Interface permettant l'interaction avec la base de données
interface PaletteRepository : JpaRepository<PaletteEntity, Long> {
    @Query("SELECT p FROM PaletteEntity p WHERE p.num = :number")

    // Fonction renvoyant l'entité Palette correspondant au paramètre 'number'
    fun findPalette(@Param("number") number: Int): PaletteEntity?
}
```

Service

Le package *service* possède des "Service" qui contiennent la logique métier des interactions. C'est ce qui va faire le lien entre un Controller et un Repository : le Controller appelle le Service qui va effectuer des opérations sur les données du Repository. C'est cette partie qui va s'occuper de mettre à jour l'emplacement d'une palette dans la base par exemple :

```
@Service
// Classe gérant la logique métier de l'application
class PaletteService(private val repository: PaletteRepository) {
    fun findPalette(number: Int): PaletteDTO? {
        val palette = repository.findPalette(number)
        return palette?.toDTO()
    }

    fun updatePaletteEmpl(number: Int, newEmpl: EmplacementDTO): PaletteDTO? {
        val palette = repository.findPalette(number) ?: return null
        palette.numeAnciEmpl = palette.numeEmpl
        repository.save(palette)
        palette.numeEmpl = newEmpl.toEntity()
        repository.save(palette)
        return palette.toDTO()
    }
}
```

Controller

Le package *controller* contient des "Controller" qui interceptent les requêtes HTTP depuis des URLs spécifiques en provenance des clients, traitent les données demandées grâce à des Service, et envoient au client une réponse (accompagnée d'un code HTTP).

```
@RestController
@RequestMapping("/palettes")
// Classe permettant de gérer les requêtes HTTP du client
class PaletteController(private val service: PaletteService) {
    @GetMapping("/search/{number}")
    // Fonction pour gérer les requêtes GET de '/search/{number}'
    // et renvoyer les données de la palette trouvée
    fun getPalette(@PathVariable number: Int): ResponseEntity<PaletteDTO> {
        val dto = service.findPalette(number)
        return if (dto != null) {
            ResponseEntity.ok(dto)
        } else {
            ResponseEntity.notFound().build()
        }
    }

    @PutMapping("/update/{number}")
    // Fonction pour gérer les requêtes PUT de '/update/{number}'
    // et modifier l'emplacement de la palette trouvée
    fun updatePaletteEmpl(@PathVariable number: Int,
                          @RequestBody newEmpl: EmplacementDTO): ResponseEntity<PaletteDTO> {
        val updatedDto = service.updatePaletteEmpl(number, newEmpl)
        return if (updatedDto != null) {
            ResponseEntity.ok(updatedDto)
        } else {
            ResponseEntity.notFound().build()
        }
    }
}
```

DTO

Le package *dto* contient des "DTO" (*Data Transfer Object*) qui permettent de notamment transférer des données depuis le backend vers le frontend. Les données que l'on récupère vont être stockées dans nos différents DTO, qui vont être convertis en une réponse JSON directement à destination de l'application frontend.

```
// Classe de données DTO permettant de transférer les
// données de l'entité Palette au reste de l'application
data class PaletteDTO (
    val num: Int,
    val empl: EmplacementDTO?,
    val anciEmpl: EmplacementDTO?,
    val numePalCli: String?,
    val detaRece: DetaReceDTO?,
    val poidBrutExpe: Float?
)
```

Cas du PaginatedResponseDTO

Ce DTO est utile lors de la récupération de données à paginer comme, dans notre cas, les palettes d'un client. Il prend comme données :

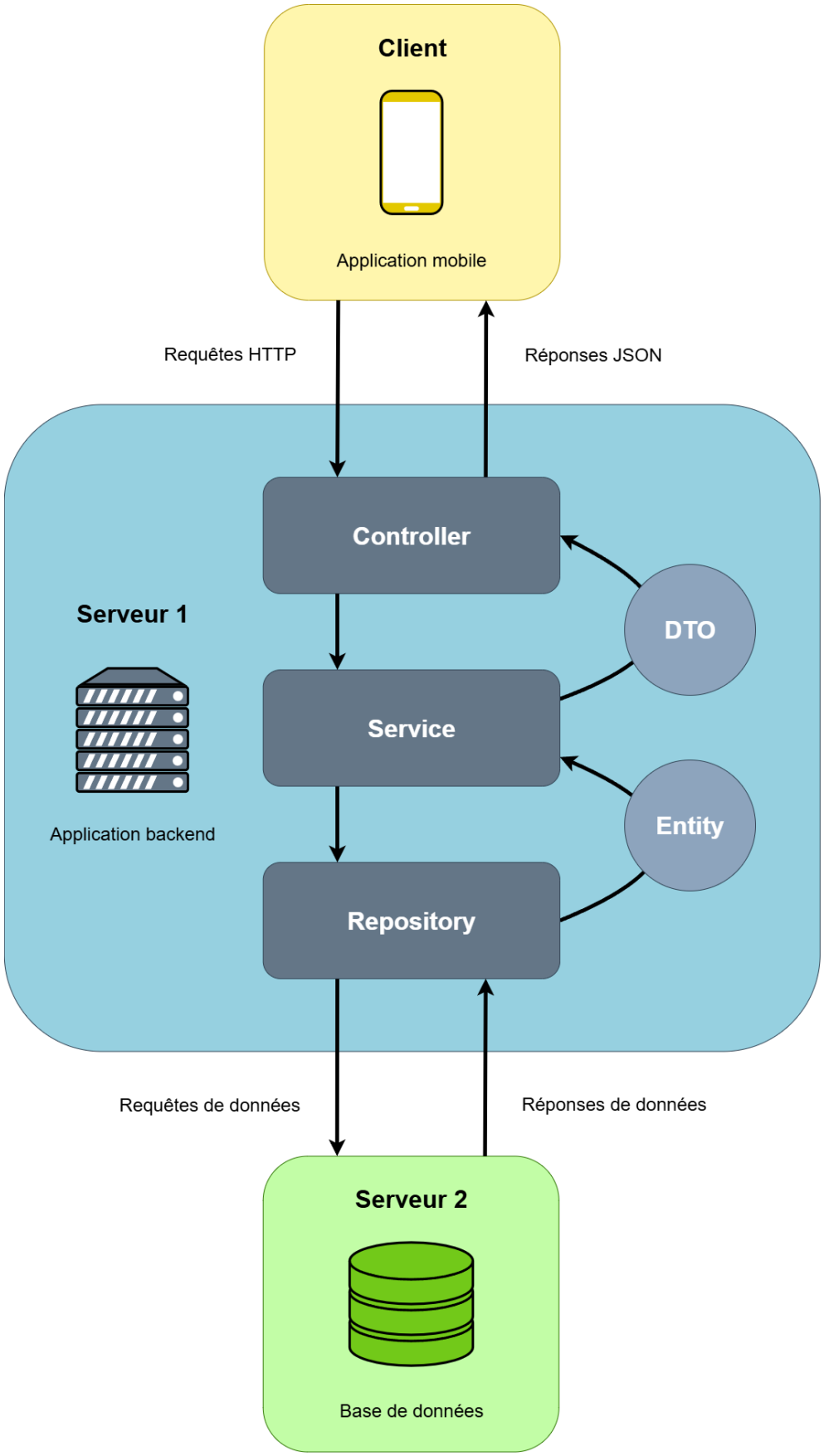
- `totalItems` correspondant au nombre total d'éléments à afficher
- `totalPages` correspondant au nombre total de pages à afficher
- `currPage` correspondant à la page actuelle affichée
- `data` correspondant à une liste d'un type de données et qui comprendra seulement les ***`totalItems`*** données requêtées

```
data class PaginatedResponseDTO<T> (
    val totalItems: Long,
    val totalPages: Int,
    val currPage: Int,
    val data: List<T>
)
```

Fichier *application.properties*

Le fichier *application.properties* diffère des autres puisqu'il est le fichier qui concentre la configuration de nombreuses parties du projet, que ce soit la mise en place d'une connexion à la base de données externe, la configuration de la connexion entre les applications mobiles frontend et ce backend via une connexion HTTPS, la fonctionnalité de whitelisting d'adresses IP d'appareils spécifiques, ou encore la mise en place d'un système de logging depuis le serveur backend.

Schéma détaillé de l'architecture



Endpoints

Afin d'intercepter les requêtes HTTP en provenance du frontend, les Controllers écoutent sur différents endpoints (points de terminaison) correspondant à des URLs de communication. C'est donc via cette URL que la requête du client est transmise, récupérée par le controller, et qui ensuite y renvoie une réponse au format JSON, que le frontend se chargera d'intercepter d'une manière similaire.

/palettes/search/{number}

Cet endpoint utilise la méthode HTTP "GET" et permet de rechercher une palette via l'URL */palettes/search/{number}* à l'aide d'un paramètre *number* qui correspond au numéro de la palette à rechercher. Le controller y transmet ensuite une réponse au format JSON du DTO de la palette récupérée :

```
@GetMapping("/search/{number}")
// Fonction pour gérer les requêtes GET de '/search/{number}'
// et renvoyer les données de la palette trouvée
fun getPalette(@PathVariable number: Int): ResponseEntity<PaletteDTO> {
    val dto = service.findPalette(number)
    return if (dto != null) {
        ResponseEntity.ok(dto)
    } else {
        ResponseEntity.notFound().build()
    }
}
```

/palettes/update/{number}

Cet endpoint utilise la méthode HTTP "PUT" et permet de mettre à jour l'emplacement d'une palette via l'URL */palettes/update/{number}* à l'aide d'un paramètre *number* qui correspond au numéro de la palette dont l'emplacement doit être modifié. En parallèle, on passe dans le corps de la requête le paramètre *newEmpl* correspondant au nouvel emplacement que la palette prendra :

```
@PutMapping("/update/{number}")
// Fonction pour gérer les requêtes PUT de '/update/{number}'
// et modifier l'emplacement de la palette trouvée
fun updatePaletteEmpl(@PathVariable number: Int,
                      @RequestBody newEmpl: EmplacementDTO): ResponseEntity<PaletteDTO> {
    val updatedDto = service.updatePaletteEmpl(number, newEmpl)
    return if (updatedDto != null) {
        ResponseEntity.ok(updatedDto)
    } else {
        ResponseEntity.notFound().build()
    }
}
```

/emplacements/search/{number}

Cet endpoint utilise la méthode HTTP "GET" et permet de rechercher un emplacement via l'URL */emplacements/search/{number}* à l'aide d'un paramètre *number* qui correspond au numéro de l'emplacement à rechercher. Le controller y transmet ensuite une réponse au format JSON du DTO de l'emplacement récupéré :

```
@GetMapping("/search/{number}")
// Fonction pour gérer les requêtes GET de '/search/{number}'
// et renvoyer les données de l'emplacement trouvé
fun getEmplacement(@PathVariable number: Int): ResponseEntity<EmplacementDTO> {
    val dto = service.findEmplacement(number)
    return if (dto != null) {
        ResponseEntity.ok(dto)
    } else {
        ResponseEntity.notFound().build()
    }
}
```

/clients/

Cet endpoint utilise la méthode HTTP "GET" et permet de récupérer la liste des clients via l'URL */clients/*. Le controller y transmet ensuite une réponse au format JSON d'une liste des DTOs des clients récupérés :

```
@GetMapping("/")
// Fonction pour gérer les requêtes GET de '/'
// et renvoyer les données des clients trouvés
fun getClients(): ResponseEntity<List<ClientDTO>> {
    val clients = service.findClients()
    return if (clients?.isNotEmpty() == true) {
        ResponseEntity.ok(clients)
    } else {
        ResponseEntity.noContent().build()
    }
}
```

/clients/search/{code}/palettes

Cet endpoint utilise la méthode HTTP "GET" et permet de rechercher les palettes d'un client via l'URL */clients/search/{code}/palettes/* à l'aide d'un paramètre *code* qui correspond au code du client dans la base de données. En parallèle, l'URL peut être accompagnée de paramètres afin de spécifier la page à renvoyer et le nombre de palettes par page à renvoyer, l'URL pouvant ainsi prendre une forme telle que */clients/search/{code}/palettes?page=8&size=12*. Le controller y transmet ensuite une réponse au format JSON d'une PaginatedResponseDTO des palettes récupérées :

```
@GetMapping("/search/{code}/palettes")
// Fonction pour gérer les requêtes GET de '/search/{code}/palettes'
// et renvoyer les données des palettes trouvées
fun getPalettesClient(@PathVariable code: String,
                      @RequestParam(defaultValue = "0") page: Int,
                      @RequestParam(defaultValue = "10") size: Int
): ResponseEntity<PaginatedResponseDTO<PaletteDTO>> {
    val palettesPage = service.findPalettesClient(code, page, size)
    return if (palettesPage == null) {
        ResponseEntity.status(HttpStatus.NOT_FOUND).body(null)
    } else if (palettesPage.content.isNotEmpty()) {
        ResponseEntity.ok(palettesPage.toPaginatedResponseDTO())
    } else {
        ResponseEntity.noContent().build()
    }
}
```

Connexion à la base de données

Le serveur backend communique avec le serveur 1 où est présente la base de données PEM afin d'accéder aux informations requises. Pour cela, une connexion est établie grâce au composant Spring Boot du framework Spring dans le fichier de configuration *application.properties* :

- `spring.datasource.url` : permet de spécifier l'URL de connexion à la base de données ainsi que des paramètres de sécurité.
- `spring.datasource.username` : permet de spécifier le nom d'utilisateur pour se connecter à la base de données.
- `spring.datasource.password` : permet de spécifier le mot de passe à utiliser pour se connecter à la base de données.
- `spring.datasource.driver-class-name` : permet de spécifier le nom de la classe du driver JDBC à utiliser pour se connecter à la base de données, ici, en Microsoft SQL Server. Ce driver agit comme un pont entre l'application Kotlin et la base de données, en traduisant les appels de l'application en commandes compréhensibles par la base de données.

Ce fichier *application.properties* étant passé en paramètre de l'exécutable du backend, il est directement modifiable sans recompilation du projet et permet donc une reconfiguration rapide de la base dès que nécessaire.

Sécurité

Contrairement à la précédente version de l'application, celle-ci est désormais plus sécurisée grâce à notamment ce serveur backend qui agit comme une passerelle entre l'application mobile et la base de données, afin d'empêcher une connexion directe entre le frontend et le serveur de bases de données. De plus, la communication s'effectue par une connexion sécurisée HTTPS et propose un système de whitelisting d'adresses IP, afin de ne laisser accès au backend qu'à certains appareils présélectionnés. Ces mesures sont toutes configurables dans le fichier *application.properties* ce qui permet, comme pour la connexion à la base de données, une reconfiguration rapide dès que nécessaire.

Connexion HTTPS

La connexion via HTTPS est configurée dans le fichier *application.properties* par de nombreuses propriétés que sont :

- `server.port` : spécifie le port sur lequel le serveur va écouter les connexions.
- `server.ssl.enabled` : active le support SSL pour le serveur, permettant une connexion sécurisée.
- `server.ssl.key-store` : indique l'emplacement du keystore contenant le certificat SSL.
- `server.ssl.key-store-password` : permet de spécifier le mot de passe pour accéder au keystore.
- `server.ssl.key-store-type` : permet de spécifier le type de keystore.
- `server.ssl.key-alias` : indique l'alias de la clé à utiliser dans le keystore.

Whitelisting

Le fichier *application.properties* contient une liste *whitelist.ips* à laquelle on peut affecter les différentes IPs à autoriser à accéder au backend. Tout appareil dont l'adresse IP ne figure pas dans cette liste ne sera pas autorisé à communiquer avec le serveur.

Deux fichiers présents dans le package *whitelist* sont nécessaires à cette fonctionnalité :

- *WhitelistProperties.kt* : va chercher au sein d'*application.properties* la propriété *whitelist* et stocke son contenu (les adresses IP autorisées) en une liste en Kotlin.
- *IpWhitelistFilter.kt* : prend en paramètre la liste des IPs autorisées et vérifie si l'adresse IP associée à l'utilisateur ayant émis la requête HTTP est présente dans la liste. Si tel est le cas, sa requête est traitée normalement, autrement, un JSON est renvoyé au client lui spécifiant que sa requête n'a pas pu aboutir car son adresse IP n'est pas autorisée par le serveur, accompagné d'un code d'erreur 403.

Logging

Le serveur backend possède une fonctionnalité de logging en fichiers LOG utilisée dans le fichier *logback-spring.xml* et utilisant le framework Logback qui est présent par défaut dans Spring. Ce fichier XML utilise des paramètres qu'on renseigne directement dans notre fichier *application.properties* afin de permettre une reconfiguration plus simple de la fonctionnalité encore une fois :

- `spring.profiles.active` : en choisissant de rentrer le paramètre `logging-on` ou `logging-off`, cela charge un profil Spring différent permettant respectivement l'activation ou la désactivation du système de logging
- `logging.file.name` : spécifie le fichier d'output du logging

- **`logging.file.pattern`** : décrit le format à suivre dans le nom des fichiers LOG archivés (ici, le fichier archivé prendra la date de son existence)
- **`logging.pattern`** : décrit le format dans lequel le logging sera écrit au sein du fichier
- **`logging.file.max-size`** : spécifie la taille maximale d'un fichier LOG
- **`logging.file.max-history`** : spécifie la durée de vie d'un fichier LOG
- **`logging.file.total-size-cap`** : spécifie la taille maximale du répertoire *logs* avant la suppression des plus anciens logs
- **`logging.level.org.springframework.web=DEBUG`** : permet d'afficher des logs supplémentaires à propos des requêtes web HTTP interceptées

```
# Logging
spring.profiles.active=logging-on
#spring.profiles.active=logging-off

logging.file.name=logs/palettes.log
logging.file.pattern=logs/palettes.%d{dd-MM-yyyy}.log
logging.pattern=%d{yyyy-MM-dd HH:mm:ss} %-5level %logger{36} - %msg%n
logging.file.max-size=10MB
logging.file.max-history=30
logging.file.total-size-cap=1GB

logging.level.org.springframework.web=DEBUG
```

Application mobile frontend

Rôle de l'application

La partie frontend du projet correspond à l'application mobile disponible sur les différents terminaux. C'est là où les utilisateurs vont pouvoir interagir afin de rechercher l'emplacement d'une palette, modifier cet emplacement, ou encore rechercher les palettes d'un client. Pour cela, l'application va requêter le backend en HTTP et traiter ses réponses JSON afin d'afficher les informations demandées par l'utilisateur.

Structure de l'application

Le frontend est organisé en de nombreux fichiers, les plus notables étant :

- Un fichier *MainActivity.kt* et un fichier *CManager.kt* permettant la gestion globale de l'application, de ses pages, son initialisation
- Des classes Kotlin correspondant à chaque page de l'application : c'est là où les différents éléments de chaque page vont être gérés, comme leurs interactions
- Des objets Kotlin ayant un fonctionnement similaire aux classes mais utiles dans notre cas pour la création d'overlays : c'est là où l'overlay de chargement ou les pop-ups de retours (application comme serveur) vont être gérés
- Des fichiers liés à la connexion à l'API backend : c'est là où les requêtes HTTP et les réponses JSON vont être gérées notamment
- Des classes de données jouant un rôle similaire aux DTOs du backend : elles vont permettre de stocker les différentes données réceptionnées
- Des *layouts* correspondant aux différentes interfaces à afficher : ces fichiers sont au format XML et ne contiennent que des éléments visuels, ce sont ensuite les classes Kotlin qui vont se charger de leurs interactions
- Des fichiers divers d'images XML, JPG, de stockage de variables comme des couleurs ou des chaînes de caractères...

Navigation dans l’application

Au lancement de l'application, le layout *layout_startup* est chargé grâce à la classe principale du projet *MainActivity.kt*. Il s'occupe d'afficher à l'utilisateur un écran de "chargement" contenant le logo de la société PEM, à la manière de l'ancienne application. Passé un bref délai (actuellement, 2000ms), la classe initialise enfin l'application : le header et le footer sont créés et leurs différents boutons deviennent interactifs. Également, la première page de l'application est chargée à l'aide du *CManager.kt* : la recherche d'une palette.

Recherche Palette

La première page correspond à la recherche d'une palette. L'utilisateur y voit un champ de saisie où il peut saisir un nombre correspondant au numéro de palette souhaité. À la validation de sa requête, il verra alors s'afficher les informations sur l'emplacement de la palette recherchée : son emplacement, son bâtiment, son site, et son ancien emplacement.

Rangement Palette

La seconde page correspond au rangement d'une palette. L'utilisateur y voit cette fois-ci deux champs de saisie : l'un pour y rentrer le numéro de palette dont l'emplacement doit être modifié, et l'autre pour rentrer l'identifiant du nouvel emplacement. Il peut rechercher des informations sur la palette grâce au premier champ de saisie qui possède son propre bouton de validation, et peut, s'il a également saisi un nouvel emplacement, valider sa saisie d'emplacement au niveau du second champ, mettant ainsi à jour l'emplacement de la palette préalablement sélectionnée (ainsi que son *ancien emplacement* évidemment).

Palettes Client

La troisième page correspond à la recherche des palettes d'un client. L'utilisateur y retrouve un unique champ de saisie qui prend cette fois-ci un texte comme input. En effet, il va pouvoir y renseigner le nom d'un client et voir une liste de quelques suggestions apparaître au fil de sa saisie. Une fois le client aperçu, il peut cliquer sur la suggestion correspondant au client afin de compléter sa saisie.

L'utilisateur verra d'ailleurs que le champ avec le nom du client est complété par le code du client. Cela est utile pour différencier quelques entrées dans la base qui possèdent le même nom de client mais pas le même identifiant, par exemple, pour un client ayant plusieurs adresses différentes. C'est ce code qui est d'ailleurs indispensable et non le nom du client, puisque c'est grâce à lui que l'on va récupérer les différents champs dans la base de données.

Si sa requête est validée, l'utilisateur pourra voir une popup s'afficher lui demandant de chercher les palettes soit au site de Siaugues, soit de Saugues. Une fois son choix fait, il verra une liste de palettes avec comme informations leur numéro, leur emplacement, leur bâtiment, leur site, d'une manière similaire à la recherche d'une palette précise. Le nombre de palettes par client étant souvent élevé, un système de pagination est mis en place de sorte que l'utilisateur puisse naviguer grâce à des flèches au bas de la page entre les différentes pages de la liste.

Paramètres

Le footer de l'application est présent sur chacune des pages et comporte un bouton représenté par un rouage permettant d'accéder à une page de paramètres. Dans cette page, l'utilisateur y retrouve deux champs de saisie : l'un lui permettant de modifier l'adresse du serveur backend, et l'autre lui permettant de modifier le port de connexion du serveur backend. La page est tout autant intuitive que les autres et permet à quiconque de modifier facilement ces informations si nécessaire.

Interactions avec l'API backend

Afin de permettre à notre application mobile frontend de communiquer avec le serveur backend, nous allons utiliser la librairie Retrofit. On va configurer cet outil de sorte qu'on puisse communiquer des requêtes HTTP sur une URL et qu'on y reçoive ensuite des réponses du serveur backend au format JSON.

ApiConfig

L'objet *ApiConfig* va permettre de stocker dans un singleton l'adresse du serveur ainsi que son port de connexion. Cela va permettre de n'avoir qu'une instance de ces informations, et qui pourront être lues depuis n'importe quelle partie de l'application mais surtout être rendues modifiables (réalisable depuis l'onglet de paramètres).

```
object ApiConfig {
    var address: String = "
    var port: Int =
}
```

RetrofitInstance

L'objet *RetrofitInstance* permet de créer une URL de connexion HTTP (dans notre cas, en HTTPS) à partir de l'adresse et du port de connexion du serveur spécifiés dans l'instance *ApiConfig*. Cet objet va également s'occuper de mettre à jour l'URL de connexion dès qu'une des variables est modifiée dans l'objet *ApiConfig* (dès qu'un changement est apporté dans les paramètres).

```
object RetrofitInstance {
    private var retrofit: Retrofit? = null
    private var currBaseUrl: String? = null

    fun getRetrofitInstance(): Retrofit {
        val baseUrl = "https://${ApiConfig.address}:${ApiConfig.port}"

        if (retrofit == null || baseUrl != currBaseUrl) {
            currBaseUrl = baseUrl
            val client = UnsafeOkHttpClient.getUnsafeOkHttpClient()

            retrofit = Retrofit.Builder()
                .baseUrl(baseUrl)
                .client(client)
                .addConverterFactory(GsonConverterFactory.create())
                .build()
        }

        return retrofit!!
    }
}
```

ApiInstance

L'objet *ApiInstance* permet de créer une interface *ApiService* à partir de l'objet retrofit. Sa structure est très similaire à celle de l'objet *RetrofitInstance* du fait qu'il recrée l'*ApiService* dès que l'objet retrofit est modifié, et donc dès que l'URL de connexion est modifiée par la même occasion.

```
object ApiInstance {
    private var apiService: ApiService? = null
    private var lastBaseUrl: String? = null

    fun getApiService(): ApiService {
        val baseUrl = "https://${ApiConfig.address}:${ApiConfig.port}"

        if (apiService == null || lastBaseUrl != baseUrl) {
            lastBaseUrl = baseUrl
            val retrofit = RetrofitInstance.getRetrofitInstance()
            apiService = retrofit.create(ApiService::class.java)
        }

        return apiService!!
    }
}
```

ApiService

L'interface *ApiService*, créé grâce à l'*ApiInstance*, permet de créer des fonctions requêtant l'API du backend via l'URL Retrofit créée et en y rajoutant différentes routes utilisant des méthodes HTTP se connectant ainsi aux endpoints côté backend, que les Controllers vont se charger d'intercepter. Une fois récupérées, les données vont être stockées dans les différentes classes de données Kotlin pour ensuite être utilisées dans l'application.

```
interface ApiService {
    @GET("/palettes/search/{number}")
    // Fonction requêtant l'API pour récupérer la palette passée en paramètre
    fun getPalette(@Path("number") palNumber: Int): Call<DataPalette>

    @PUT("/palettes/update/{number}")
    // Fonction requêtant l'API pour mettre à jour l'emplacement de la palette passée en paramètre
    fun updatePaletteEmpl(@Path("number") palNumber: Int,
                          @Body newEmpl: DataEmplacement): Call<DataPalette>

    @GET("/emplacements/search/{number}")
    // Fonction requêtant l'API pour récupérer l'emplacement passé en paramètre
    fun getEmplacement(@Path("number") emplNumber: Int): Call<DataEmplacement>

    @GET("/clients/")
    // Fonction requêtant l'API pour récupérer les clients
    fun getClients(): Call<List<DataClient>>

    @GET("/clients/search/{number}/palettes")
    // Fonction requêtant l'API pour récupérer les palettes du client passé en paramètre
    fun getPalettesClient(@Path("number") cliNumber: String?,
                          @Query("page") page: Int,
                          @Query("size") size: Int,
                          @Query("site") site: Int): Call<PaginatedResponse<DataPalette>>
}
```

Data classes

Les classes de données vont, à la manière des DTOs côté backend, traduire en langage Kotlin les différentes entités du backend afin de pouvoir y stocker les informations récupérées sur le frontend. Ce sont ces variables qu'elles contiennent qui vont, par exemple, être appelées lorsqu'on a besoin de les afficher.

```
// Classe de données traduisant la structure de l'entité Palette côté backend
data class DataPalette (
    val num: Int,
    val empl: DataEmplacement?,
    val anciEmpl: DataEmplacement?,
    val numePalCli: String?,
    val detaRece: DataDetaRece?,
    val poidBrutExpe: Float?
)
```

Gestion d'erreurs

Afin d'éviter des requêtes invalides, trop lourdes, ou même des retours inattendus de la part du serveur, un certain nombre d'erreurs ont été gérées, promettant ainsi une application stable, qui ne plante pas, et qui informe l'utilisateur dès qu'un problème est rencontré.

Erreurs HTTP

La majeure partie de la gestion des erreurs de l'application concerne la gestion des codes HTTP renvoyés par le serveur à l'issue de chaque requête de l'utilisateur. Ces codes vont être lus par le frontend et en fonction de leur valeur, l'application mobile affichera un retour à l'utilisateur : la réponse à sa requête ou une notification d'erreur.

Cas du code 200

Dans le cas d'un code 200, cela signifie que la requête a bien abouti, l'utilisateur n'est notifié d'aucune erreur et sa requête s'effectue avec succès.

Cas du code 204

Dans le cas d'une erreur 204, l'utilisateur est informé que l'information qu'il a requêtée existe mais que son contenu est vide. Cela arrive par exemple lorsque l'utilisateur recherche les palettes d'un client : si le client existe, pas d'erreur 404 ne sera renvoyée, mais si ce client ne possède aucune palette, c'est l'erreur 204 qui sera renvoyée.

Cas du code 403

Dans le cas d'une erreur 403, l'utilisateur est informé qu'il n'a pas l'autorisation du serveur d'accéder aux informations requêtées. Cela survient actuellement lorsque l'adresse IP du client n'est pas autorisée côté backend.

Cas du code 404

Dans le cas d'une erreur 404, l'utilisateur est informé que l'information qu'il a requêtée n'a pas été trouvée par le serveur. Cela peut par exemple survenir quand l'utilisateur effectue une recherche sur une palette qui n'est pas présente dans la base de données.

Autres cas

Si le code lu est différent des 4 cas ci-dessus, une erreur générique est affichée à l'utilisateur en lui disant qu'une erreur inattendue est survenue.

Erreurs de connexion au serveur

Dans le cas où l'application mobile n'arrive pas à communiquer avec le serveur pour une raison quelconque (temps d'attente trop long, connexion impossible, URL non existante...), la fonction *onFailure* est appelée. Cette fonction est présente dans chacune des fonctions faisant un appel au serveur à l'aide de la librairie Retrofit. À son appel, l'utilisateur est informé que la connexion au serveur est impossible, sans que l'application plante et devienne inutilisable. Il peut donc réessayer quand il le souhaite.

Prévention d'erreurs dans l'application mobile

En prévention de mauvaises requêtes aboutissant forcément à une erreur HTTP, des conditions dans l'application mobile ont été mises en place pour empêcher de faire des requêtes qui, à coup sûr, n'auraient pas abouti.

- Pour tous les champs de saisie exceptés celui du nom de client et celui de l'adresse serveur, tous sont au format numérique et n'acceptent que les dix chiffres comme possibles caractères.
- Que ce soit au sein des layouts XML ou des classes Kotlin, les différents champs de saisie ont une contrainte de longueur, empêchant ainsi l'utilisateur de saisir, par exemple, plus de 6 chiffres pour un numéro de palettes, ou même de ne rien saisir du tout et de requêter un champ vide.

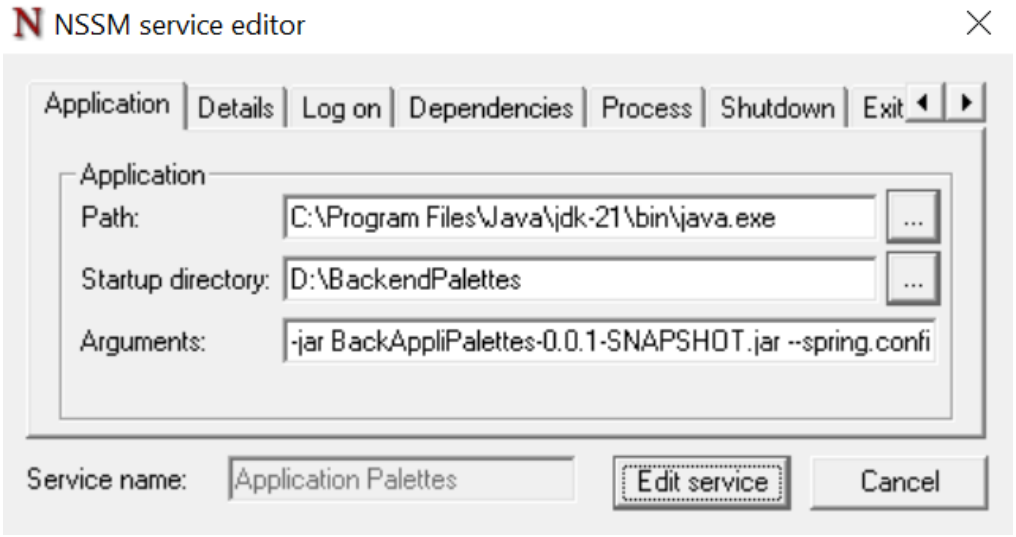
Déploiement

Backend

L'application backend peut être built en un fichier JAR dans Android Studio grâce à l'outil de build Gradle en entrant la commande `.\gradlew.bat clean build` dans le terminal d'Android Studio (par exemple). Cela générera un fichier JAR localisé dans *.\build\libs* qui peut être ensuite exécuté directement via un terminal, ou dans notre cas, utilisé dans un service Windows.

Pour mettre en place un service à partir de notre fichier JAR, l'outil NSSM a été utilisé. En écrivant depuis un terminal Windows lancé en mode administrateur sur le serveur la commande `nssm install "Application Palettes"`, une interface de création du service s'ouvre pour configurer le service à créer. Notamment, l'application a besoin de connaître :

- *Path* : où chercher l'exécutable, dans notre cas, de Java
- *Startup directory* : le répertoire de démarrage de l'application
- *Arguments* : la commande que le service permet d'exécuter

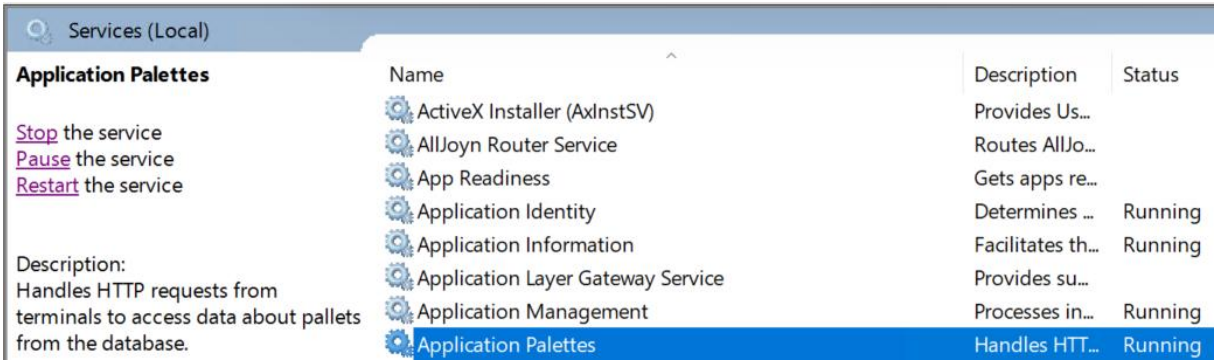


Arguments correspond, ici, à la commande :

```
-jar BackAppliPalettes-0.0.1-SNAPSHOT.jar
--spring.config.location=.\application.properties
```

La première ligne permet l'exécution de notre backend JAR, et la seconde le passage d'un argument à cette commande : cet argument indique au programme d'utiliser le fichier de configuration application.properties spécifié plutôt que celui généré de base dans le fichier JAR.

Après la création du service, il est toujours possible de lui appliquer des modifications à l'aide de la commande `nssm edit "Application Palettes"`. À noter qu'après la mise en place du service, il est plus naturel et plus pratique de passer directement via l'outil de gestion des services de Windows, accessible en ouvrant le programme *Services* de Windows, pour gérer son statut, notamment : le lancer, l'arrêter, le redémarrer.



Application mobile

L'application mobile peut être built en un fichier APK dans Android Studio grâce à l'outil de build Gradle via : *Build* > *Generate App Bundles or APKs* > *Generate APKs* qui générera un fichier .apk localisé dans .*app**build**outputs**apk**debug*\ qui peut ensuite être transféré via USB (par exemple) aux différents terminaux nécessitant l'application. En parcourant les fichiers de l'appareil mobile, il n'y a ensuite qu'à cliquer sur le fichier .apk afin de télécharger l'application.

Une fois l'application correctement installée, il y est possible de modifier sa configuration vis-à-vis du serveur. En effet, une icône de rouage dans le coin inférieur droit de l'écran peut être cliquée afin d'être dirigé sur une page de paramétrage de la connexion au serveur. Ici, l'adresse serveur peut être changée en cas de besoin tout comme le port utilisé (de base, l'adresse et le port sont ceux actuellement renseignés dans le serveur 1). À noter que le backend doit, de son côté, autoriser l'adresse IP du terminal s'y connectant afin de lui permettre d'accéder aux données.