Guide Programmeur - SmallBrother

Version 1.2
12 mai 2023

Introduction

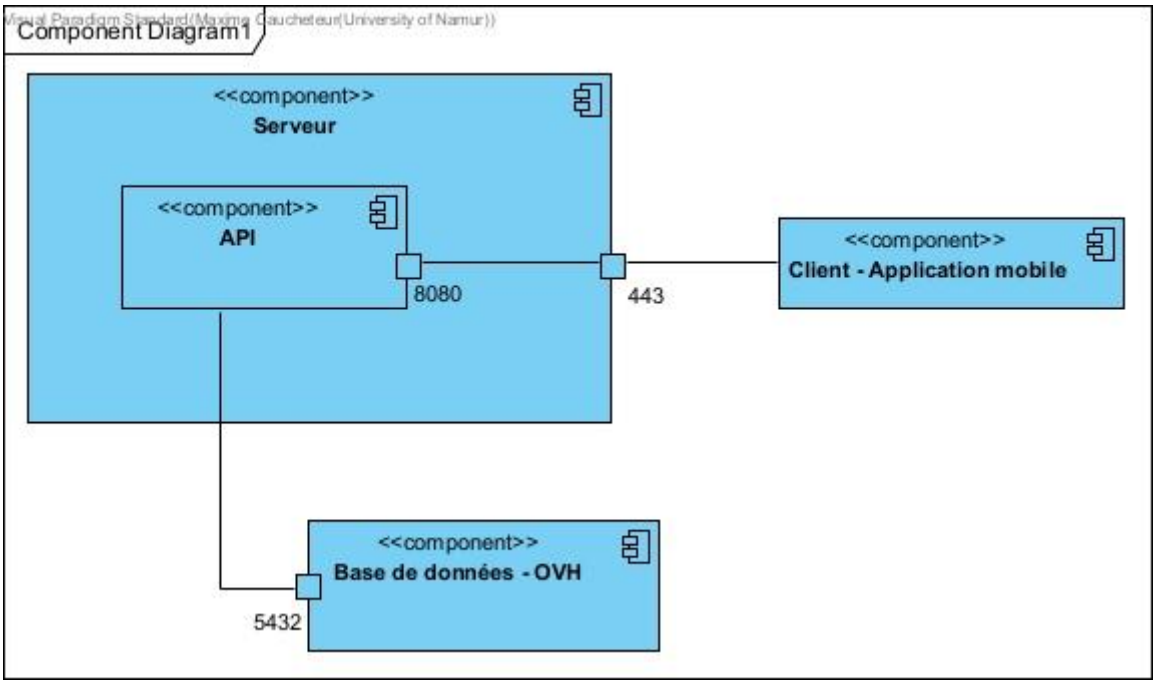
SmallBrother est une application destinée principalement aux personnes atteintes de la maladie d'Alzheimer ainsi qu'à leur accompagnant. Les objectifs principaux de SmallBrother sont de rendre de l'autonomie et de soulager les personnes malades et leurs proches. Ce guide a pour but de détailler le système informatique implémenté pour rendre l'application SmallBrother opérationnelle afin de faciliter la tâche de futurs développeurs ou des personnes responsables de la maintenance du système.

Ce guide est séparé en différentes sections:

1. L'architecture du système présentant la structure générale du système et ses différents composants;
2. La structure de la base de données du serveur;
3. Le code de l'application reprenant les points d'attention principaux, les choix d'implémentation expliqués;
4. Le serveur, sa mise en place, les différentes configurations;

Architecture du système

L'architecture du système SmallBrother se présente comme suit:



1. Le client, sous la forme d'une application mobile, effectue des appels à l'API pour envoyer et récupérer les captures de contexte et les informations associées.
2. L'API est déployée sur le serveur afin de permettre les échanges entre les différentes applications clientes et la base de données.
3. La base de données contient les tables nécessaires au stockage des données utiles à l'application.
4. Un serveur, hébergeant l'API, permet de stocker les fichiers des captures de contexte.

Application

Le code de l'application est rédigé en Kotlin, pour Android et est disponible à l'adresse suivante: [Insérer adresse du repo github public]. La version minimale d'Android est Android 6 (API 23). Le code est séparé en plusieurs packages:

- *Models*, reprenant les *data class* représentant les utilisateurs et les données des captures de contexte
- *Activities*, reprenant les différentes activités de l'application.
- *Libs*, contenant le code pour gérer l'accéléromètre ainsi que le code de la librairie permettant de capturer des photos sans afficher d'aperçu au préalable.
- *Utils*, au sein duquel on retrouve des fichiers contenant des méthodes communes à plusieurs endroits du code qui ont été factorisées.

On retrouve également plusieurs classes/objets qui ne sont dans aucun package, ils feront l'objet d'une section à la fin de ce chapitre.

Modèles

Le package *Models* contient deux *data class*, UserData et AideData. Cette dernière représente les informations associées aux captures de contexte et est utilisée pour transmettre les informations au serveur. Ce mécanisme est expliqué plus en détail dans la section sur l'API. La première, UserData, est instanciée pour chaque utilisateur et reprend ses informations. Cette classe de données représente un point central du code, étant utilisée dans chaque activité pour accéder et modifier les données de l'utilisateur.

Activités

L'application contient 14 activités détaillées ci-dessous, dans l'ordre alphabétique.
Un point commun à chaque activité est qu'elles ont toutes besoin d'utiliser la classe UserData. Pour ce faire, une classe Singleton a été créée: *UserDataManager* qui sera expliquée plus en détail dans ce guide.
Certaines des méthodes, utilisées dans ces activités, sont définies dans des fichiers sous le répertoire Utils et seront traitées dans la section correspondante.

AidantActivity

Gère les actions que l'aidant peut effectuer depuis son écran d'accueil telles qu'envoyer un message à l'aidé, appeler l'aidé, demander une capture de contexte de l'aidé. Elle permet également de rediriger vers d'autres activités.
La méthode intéressante dans cette activité est *getDataOnServer()* qui permet de récupérer et de déchiffrer la capture de contexte sur le serveur. Cette méthode est découpée comme suit:

- Un appel à la méthode *downloadFileRequest()* qui réalise une requête GET sur l'API pour récupérer le fichier de la capture de contexte.
- 1 requête GET pour récupérer les informations associées à cette capture: la clé AES chiffrée, la signature du fichier et le vecteur d'initialisation.
- La signature du fichier est vérifiée grâce à la clé publique de l'aidé, récupérée à l'installation par l'aidant.
- La clé AES est déchiffrée à l'aide de la clé RSA privée de l'aidant.

- Le fichier est déchiffré grâce à la clé AES, elle-même déchiffrée à l'étape précédente.
- L'archive est enregistrée dans le dossier de téléchargement avant d'être extraite.

AideActivity

Gère les actions que l'aidé peut effectuer depuis son écran d'accueil telles que répondre au message de l'aidant, appeler l'aidant, lancer une capture de contexte pour signaler une urgence ou encore activer le mode privé. *AideActivity* contient essentiellement des méthodes permettant d'actualiser l'interface de l'aidé, comme le temps restant du mode privé ou encore le texte du log.

InstallActivity

Le rôle de cette activité est double, le premier étant de demander les permissions nécessaires à l'utilisation de l'application, le second étant de demander à l'utilisateur de remplir ses informations telles que son nom, celui de son partenaire ainsi que le numéro de téléphone de ce dernier. Les permissions demandées permettent essentiellement de gérer les aspects téléphoniques, le stockage des fichiers, la géolocalisation, la caméra ainsi que le magnétophone (ces trois dernières ne sont pas demandées pour l'aidant).

Launch1Activity

Sert de point de départ au lancement de l'application. Définie comme activité *launcher* dans le manifeste. C'est ici qu'est activé le SMSReceiver, classe essentielle détaillée plus tard.
Launch1Activity contient une méthode, *checkFirstLaunch()*, qui va rediriger l'utilisateur selon que ce soit la première fois qu'il lance l'application, qu'il l'a réinitialisée ou bien qu'il l'a déjà lancée et installée et que ses données puissent être chargées.

Launch2Activity

Permet d'attribuer le rôle que va prendre l'utilisateur en fonction du bouton choisi.

QRCodeInstallActivity

Ce QR code est à scanner lors de l'installation pour les deux utilisateurs. Il va servir à échanger les clés RSA publiques utilisées, d'une part, par l'aidé dans le chiffrement des informations de la capture de contexte (voir *SecurityUtils*) et, d'autre part, par l'aidant pour vérifier la signature de la capture de contexte lui permettant de vérifier l'origine du fichier.
La méthode *qrEncoder()* permet de créer et d'afficher le QR code grâce à la méthode QRGEncoder présente dans la librairie *androidmads.library.qrgenerator*.

QRCodeScannerInstallActivity

Afin de scanner le QR code, l'application utilise la librairie *com.budiyev.android.codescanner* pour instancier un objet *CodeScanner*. Cet objet permet également, une fois le QR code scanné, de traiter le résultat (avec la méthode *processScanResult()*) et de sauvegarder la clé publique dans le fichier des données de l'utilisateur grâce à la méthode *userData.saveData()*.

SettingsActivity

Permet à l'aidant de réinitialiser les informations de son application et de celle de l'aidé, simultanément.

SuccessScanActivity

Cette activité permet de notifier l'aidé qu'il doit présenter le prochain QR code à l'aidant. Elle permet aussi de marquer une pause lors de l'installation pour éviter toute erreur lors des scans de QR code.

WorkActivity

Représente avec *Work2Activity* les deux activités responsables de la capture de contexte. *WorkActivity* vérifie d'abord que l'aidé ait accès à Internet, ensuite une série de captures se produit. D'abord, un enregistrement audio de 10 secondes, durant lequel l'accélération et les coordonnées sur les axes x, y et z sont capturées. Ensuite, le niveau d'exposition à la lumière est également capturé. Les mesures d'accélération sont interprétées dans la méthode *interpretAcceleration()* et les mesures sur les axes sont comparées. Le fichier audio est enregistré sur le smartphone et toutes les autres informations sont transmises à l'activité *Work2Activity* pour terminer la capture de contexte.

Work2Activity

Une fois les informations de *WorkActivity* reçues, cette classe continue la capture de contexte en récupérant la géolocalisation du smartphone de l'aidé (si disponible) via les coordonnées géographiques dont est dérivée une adresse via la méthode *getAddress()*. Cette géolocalisation est également utilisée pour aider à déterminer le mouvement de l'aidé. Également, deux photos sont prises grâce à la classe *APictureCapturingService* implémentée dans la classe *PictureCapturingServiceImpl* par hzitoun[1] et convertie en Kotlin avec quelques modifications telles que le temps d'exposition, l'ISO, etc.

Une fois ces photos prises, le niveau de batterie est récupéré. Les informations précédemment obtenues dans *WorkActivity* sont traitées et une interprétation du mouvement est faite dans la méthode *interpretMotionData()*. Ensuite, l'adresse (et coordonnées géographiques liées), le niveau de batterie, les informations sur le mouvement, le niveau de lumière et la date de capture sont compilés dans un fichier texte.

Après avoir fini de récolter toutes les informations, les différents fichiers sont zippés dans une archive zip grâce à la méthode *zipAll()* procédant comme suit:

- La méthode prend en paramètre le répertoire où se trouvent les fichiers à zipper et le nom final de cette archive zip;
- La méthode utilise ensuite un *ZipOutputStream* pour zipper chaque fichier et l'ajouter à l'archive via la méthode *zipFiles()*;
- Cette dernière méthode va zipper récursivement les fichiers présents dans le répertoire en omettant les fichiers zip présents ainsi que le fichier d'informations utilisateur "donnees.txt";
- Le ZipOutputStream est ensuite fermé et l'archive est composée.

[1] https://github.com/hzitoun/android-camera2-secret-picture-taker

Finalement, *Work2Activity* envoie ce fichier sur le serveur via la méthode *uploadZip()* qui fonctionne de la manière suivante:

- La méthode prend en paramètre le *HttpClient* nécessaire pour effectuer les requêtes au serveur ainsi que le fichier zip à upload;
- Un nom aléatoire est généré pour le fichier;
- Les informations de chiffrement, la clé AES et le vecteur d'initialisation, sont générées;
- Les bytes du fichier zip sont ensuite chiffrés à l'aide de la clé AES 192 bits dans le mode AES/CBC/PKCS7Padding;
- Ces bytes chiffrés sont ensuite signés en utilisant la clé RSA privée de l'aidé;
- La clé AES est chiffrée à l'aide de la clé RSA publique de l'aidant;
- Le fichier est envoyé sur le serveur via la méthode *uploadFileRequest*() qui effectue une requête POST;
- Les données liées au fichier, c'est-à-dire son nom, la clé AES chiffrée, la signature et le vecteur d'initialisation sont finalement envoyées sur le serveur via la méthode *uploadAideDataRequest()* effectuant également une requête POST en envoyant un objet *AideData* dont le contenu est sérialisé en format JSON.

Librairies

L'application reprend le code de deux librairies différentes:

1. La gestion de l'accéléromètre avec les classes et interfaces *AccelerometerListener* et *AccelerometerManager*
2. La prise de photos avec les classes et interfaces *APictureCapturingService*, *PictureCapturingServiceImpl* et *PictureCapturingListener*.

La première vérifie l'accélération du smartphone avec certaines options configurables telles que le seuil de détection ou l'intervalle de capture. La deuxième permet de capturer des photos sans afficher d'aperçu au préalable, pratique pour capturer le contexte de l'aidé sans qu'il doive lui-même prendre les photos. Il est également possible de régler certaines options comme le délai avant capture, le temps d'exposition, l'ISO ou encore l'activation du flash.

Utils

Ces fichiers reprennent chacun un ensemble de fonctions similaires ayant attrait au même sujet. L'utilisation de ces fichiers permet d'éviter la duplication de fragments de code et de faciliter la maintenance de ces méthodes.

SecurityUtils

Cet objet possède plusieurs méthodes utiles aux (dé-)chiffrements et signatures de la capture de contexte. On y retrouve les différents modes de chiffrement, la génération des clés RSA et AES. Ces clés sont stockées dans le *AndroidKeyStore*, les instances de clés publiques sont accessibles mais il n'est possible d'obtenir qu'une référence à l'instance de la clé privée, rendant la clé privée sécurisée et inutilisable à une autre application ou personne. Cet objet a été implémenté en s'inspirant de la solution proposée par Farhan Majid[2].

[2] https://yggr.medium.com/how-to-generate-public-private-key-in-android-7f3e244c0fd8

SMSUtils

Ce fichier comprend des méthodes ayant un rapport avec le service des SMS, que ce soit l'envoi de SMS, l'activation/désactivation du SMSReceiver, etc.

UIUtils

Les méthodes reprises dans ce fichier concernent l'interface de l'application comme le réglage du texte du cadre de log, l'affichage des Toast, etc.

Utils

Ce fichier reprend les méthodes utilisées à plusieurs endroits différents ne rentrant dans aucun autre fichier "utils". Il est important d'essayer de maintenir ce fichier aussi petit que possible pour éviter de le rendre peu clair et donc plus dur à maintenir.

Classes/Objets supplémentaires

Le code de l'application comprend également des classes et objets n'entrant dans aucun package. Elles sont listées et expliquées ci-dessous.

PhoneStatReceiver

Permet de récupérer le numéro de l'appelant et de vérifier s'il s'agit de l'aidant pour actualiser le log en cas d'appel de la part de ce dernier.

SmsReceiver

Lorsqu'un SMS est reçu par le smartphone de l'aidant ou de l'aidé, celui-ci est traité par cette classe qui agit comme un *BroadcastReceiver*. Chaque SMS envoyé par l'application a un format bien précis:

- Le SMS commence par "SmallBrother :"
- Le SMS termine par un code "[#SBxx]"

Si le SMS suit ce format, le SMSReceiver traite le message, sinon il ne s'en occupe pas. De cette manière, les applications SmallBrother peuvent être fermées, le SMSReceiver rouvrira l'application à la réception d'un SMS qui concerne l'application. En fonction du code présent dans le SMS, l'utilisateur est soit redirigé vers l'activité correspondante soit le message du cadre de log est actualisé.

UserDataManager

Pour éviter une duplication des instances de userData au sein des activités et pour respecter les patterns d'implémentation, cette classe a été créée et implémentée à la manière d'un Singleton. De cette façon, il est assuré que userData n'aura qu'une instance et qu'elle sera la même pour toutes les activités y accédant.

Vibration

Cette classe permet l'utilisation du vibreur du smartphone, via la méthode *vibration()* pour que les utilisateurs aient un retour haptique pour chacune de leurs actions.

Documentation

Plusieurs documents sont disponibles sur le répertoire Github pour fournir plus d'informations (diagramme de classes, de séquence, de use cases et un guide utilisateur): https://github.com/Mcauchet/SmallBrother_2023/tree/main/documentation

Manifest.xml

Dans ce fichier se trouvent toutes les permissions que l'application peut demander, les features utilisées telles que l'accéléromètre, le capteur de lumière, les caméras avant et arrière ainsi que le *GPS* pour la géolocalisation. Ensuite, dans la balise <application></application>, sont définies toutes les activités de l'application ainsi que le *receiver* pour les SMS.

Un point d'attention est l'attribut *android:usesCleartextTraffic="false"* dans la balise <application> qui indique que lors de l'utilisation du protocole *HTTP*, *SmallBrother* ne transmettra que des informations via *HTTPS*.

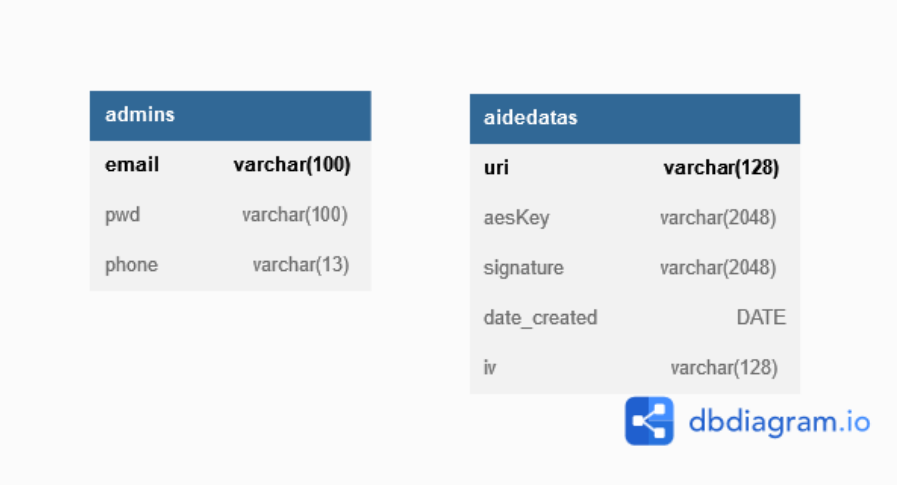
Il est important de noter la présence de l'activité *Launch1Activity*, à laquelle on assigne un *intent-filter* de catégorie *LAUNCHER*. Cette activité est chargée au démarrage de l'application et l'utilisateur est redirigé vers le bon écran s'il a déjà installé l'application ou si elle a été réinitialisée et il n'est pas redirigé si c'est la première fois qu'il lance l'application.

Build.gradle

Le code de l'application Android contient deux fichiers *build.gradle*, un au niveau du projet, l'autre au niveau de l'application. Le premier contient des informations telles que les versions de *Kotlin*, *Ktor* et *Gradle* utilisées dans le projet mais également les plugins. Le second contient des informations sur l'application telles que son nom, la version de l'API Android minimale et cible, le Runner utilisé pour réaliser les tests de l'interface et toutes les dépendances du projet. Les dépendances du projet sont également définies dans ce fichier. SmallBrother utilise les dépendances classiques d'Android *AppCompat* et *ConstraintLayout*, l'API de géolocalisation de *Google*, les frameworks *JUnit*, *Espresso*, les coroutines *Kotlin*, la librairie pour générer et scanner les QR codes, la librairie *security-crypto* et toute une série de dépendances *Ktor*.

Base de données

La base de données utilisée par la REST API déployée sur le serveur est composée de 2 tables. Une pour les administrateurs afin qu'ils puissent accéder à la plateforme Web pour gérer les enregistrements présents dans la base de données. Enfin, la deuxième table contient les informations relatives aux captures de contexte des aidés. Voici un schéma reprenant les détails des différentes tables:



- Table "admins": On retrouve dans cette table les données permettant l'identification des admins à la plateforme web https://smallbrother.be. Cette table est composée de trois colonnes: email, pwd et phone contenant respectivement l'email de l'admin, son mot de passe ainsi que son numéro de téléphone (chaîne de caractères vide s'il n'en a pas renseigné). Les trois champs sont *NOT NULL* et la colonne email joue le rôle de clé primaire.
- Table "aidedatas": Les données liées aux captures de contexte se retrouvent dans cette table, composée de cinq colonnes: uri, aesKey, signature, date_created et iv.
    - **uri** sert à stocker le nom du fichier afin de pouvoir aller le rechercher sur le serveur au moment du téléchargement par l'aidant. Cette colonne joue également le rôle de clé primaire
    - **aesKey** contient la clé AES, chiffrée par la clé RSA publique de l'aidant, servant à déchiffrer la capture de contexte une fois récupérée par l'aidant.
    - **signature** contient l'encodage en Base64 de la signature de l'archive zip et est utilisé pour vérifier l'origine du fichier.
    - **date_created** est un *timestamp* afin de savoir quand la capture de contexte a été faite pour pouvoir supprimer le fichier douze heures après sa mise sur le serveur.
    - **iv** sert à stocker le vecteur d'initialisation pour le déchiffrage de la capture de contexte, utilisé dans le cadre du déchiffrement AES/CBC/PKCS7Padding.

API

L'API a également été codée en Kotlin en utilisant le framework Ktor[3]. Elle est séparée en plusieurs packages:

- **dao**: contient la logique des appels entre l'API et la base de données.
- **models**: représente les différents objets utilisés pour obtenir les informations à envoyer à la base de données.
- **plugins**: c'est dans ce package qu'on retrouve les différents fichiers de configuration des plugins.
- **routes**: on y retrouve les fichiers contenant les routes de l'API et les appels au DAO.

DAO

Le design pattern *Data Access Object* a été utilisé pour fournir une interface qui interagit avec la base de données tout en encapsulant les détails au reste de l'application. Ce DAO a été implémenté en utilisant le framework ORM *Exposed*[4]. On retrouve trois fichiers dans ce package:

1. **DAOFacade,** l'interface pour accéder aux fonctions implémentées dans *DAOFacadeImpl*.
2. **DAOFacadeImpl,** où les différentes méthodes de l'interface sont implémentées. On retrouve les opérations *CRUD* de base. Les deux premières méthodes: *resultRowToAideData()* et *resultRowToAdmin()* servent à transformer le résultat de la requête sur la base de données en objet définis dans le package models, respectivement *AideData()* et *Admin()*.
3. **DatabaseFactory** contenant le code d'initialisation de la base de données, expliqué ici[5], ainsi qu'une méthode *dbQuery()* permettant de traiter les demandes à la base de données comme des transactions non bloquantes, et donc de tirer avantage de l'utilisation des coroutines (supportées par Ktor). Les différentes variables nécessaires à l'initialisation de la base de données sont stockées comme variable d'environnement pour ne pas mettre les informations en clair dans le code.

Models

L'API de SmallBrother contient trois modèles:

1. **Admin,** représenté par une *data class*, qui va servir à *Exposed* pour transformer les données qu'il reçoit de la base de données et des applications client en objet. L'utilisation de l'annotation @Serializable indique au plugin de sérialisation de générer automatiquement l'implémentation de *KSerializer*[6] pour la classe Admin afin de pouvoir (dé-)sérialiser la classe.
2. **AideData,** représente le même objet que dans le code de l'application et permet de normaliser les échanges d'informations entre le client et l'API.

---

[3] https://ktor.io
[4] https://github.com/JetBrains/Exposed
[5] https://ktor.io/docs/interactive-website-add-persistence.html
[6] https://kotlinlang.org/api/kotlinx.serialization/kotlinx-serialization-core/kotlinx.serialization/-k-serializer/

3. **ServerSession,** *data class* utilisée par le plugin *Session* pour créer un cookie de connexion sur la plateforme Web pour les administrateurs.

Plugins

Le framework Ktor permet une grande personnalisation et il est donc possible d'ajouter facilement des plugins. L'API utilise six de ces plugins répartis dans six fichiers:

1. **Authentication,** dans le fichier Authentication.kt, pour gérer la connexion à la plateforme Web. En cas de connexion réussie, une session est créée pour l'email.
2. **CallLogging,** dans le fichier CallLogging.kt, afin de filtrer et formater les logs des requêtes à l'API.
3. **Routing,** dans le fichier Routing.kt, pour configurer les routes de l'API. On installe le plugin à l'aide de la méthode *routing()* au sein de laquelle on renseigne les différentes méthodes de routing définies dans le package *routes*. Il est également possible de définir les chemins d'accès aux fichiers statiques sur le serveur. C'est un plugin principal de Ktor.
4. **ContentNegotiation,** dans le fichier Serialization.kt, sert aux communications entre client, API et base de données. On voit ici que le plugin *ContentNegotiation* est installé avec l'option *json()* étant donné que nos données sont envoyées au format JSON entre le client et l'API.
5. **Sessions,** dans le fichier Sessions.kt, nous permet d'utiliser des cookies et de définir leur durée de vie. Le cookie *server_session* est utilisé pour maintenir l'administrateur connecté pour une durée définie à la variable *cookie.maxAgeInSeconds*.
6. **FreeMarker,** dans le fichier Templating.kt, va nous permettre d'afficher plus facilement les enregistrements de la base de données sur le front-end en utilisant le modèle *AideData*.

Routes

Deux fichiers sont présents dans ce package, *AdministrationRoutes* et *AideDataRoutes*.

- Le premier gère les requêtes sur la plateforme Web. On y retrouve deux routes principales: /admin et /login. Pour pouvoir accéder à la route /admin, il faut être authentifié (grâce à Route.authenticate()). La vérification du mot de passe lors de l'identification est effectué à l'aide de la librairie BCrypt[7]. Si les identifiants sont vérifiés, un cookie est créé et attribué à la session en cours. Ce cookie a une durée de vie de 5 minutes. Une fois connecté, l'administrateur voit la liste des captures de contexte ajoutées sur le serveur (sans toutefois avoir accès à leur contenu). Il peut cliquer sur un enregistrement pour avoir la possibilité de le supprimer individuellement. Il peut également cliquer sur le bouton "Delete" pour supprimer tous les enregistrements de la base de données et leurs fichiers correspondants sur le serveur. Le bouton "Clean" sert à supprimer les fichiers dont les enregistrements ont été manuellement supprimés. Enfin, il peut éditer le profil administrateur en changeant adresse mail, mot de passe et numéro de téléphone en cliquant sur le lien "Edit Admin profile".
- Le second gère les requêtes envoyées par les applications clientes. La première route principale est /upload qui permet à l'aidé de téléverser sa capture de contexte

[7] https://www.npmjs.com/package/bcrypt

sur le serveur ainsi que les données liées (uri, aesKey, etc.). Il y a cependant plusieurs critères pour que le fichier soit accepté: l'extension .zip, la taille inférieure à *MAX_SIZE*, le nom du fichier inférieur à *NAME_SIZE* et la taille de l'extension inférieure à *EXT_SIZE*. Le "Content-Type" du header de la requête doit être "application/zip" pour que la requête soit traitée. La deuxième route principale est /download qui permet de récupérer le fichier de la capture de contexte sur le serveur. La troisième route principale est /aideData qui permet de récupérer les informations de la capture de contexte sous la forme d'un objet *AideData* que l'application pourra exploiter.

Application.kt

Ce fichier permet d'initialiser l'*engine* utilisé par Ktor et permet d'appeler toutes les méthodes définies pour configurer les plugins. C'est cette fonction *main()* qui est utilisée lorsque le jar est exécuté.

Ressources

Dans le dossier templates, on retrouve les fichiers .ftl (FreeMarker) qui représentent le frontend de la plateforme Web ainsi que différents fichiers statiques. Un fichier est intéressant ici, c'est *application.conf* qui contient différentes informations:

- le port utilisé par l'API, défini ici sur 8080;
- les informations pour la configuration *SSL* (l'emplacement du KeyStore, son alias et les mots de passe, définis dans des variables d'environnement);
- les informations pour la base de données lors des tests en local.

Serveur

Déployer le code de l'API sur le serveur

Afin de transférer le *.jar* de l'API sur le serveur, la commande *scp* est utilisée. Le processus est le suivant:

1. Exécuter la commande gradle *clean*
2. Exécuter la commande gradle *shadowJar*
3. Se déplacer dans le répertoire *SmallBrother2018/API/smallBrother-api/build/libs*
4. Exécuter la commande *scp smallBrother-api-all.jar [nomUtilisateurServeur]@[IP_Serveur]:~/smallbrother/*

Cron Job

Pour effacer les captures de contexte au bout de 12 heures, un cron job est lancé toutes les 10 minutes qui exécute un script vérifiant que chaque entrée dans la base de données est plus récente que 12 heures. Dans le cas contraire, l'entrée et le fichier correspondant sont supprimés grâce au script suivant:

```
export PGPASSWORD=$DATABASE_PASSWORD
files=$(psql -h $DATABASE_HOST -p $DATABASE_PORT -U $DATABASE_USERNAME -d
$DATABASE_NAME -t -c "SELECT uri FROM aidedatas WHERE date_created < NOW() -
INTERVAL '12 hours';")

psql -h $DATABASE_HOST -p $DATABASE_PORT -U $DATABASE_USERNAME -d $DATABASE_NAME
-c "DELETE FROM aidedatas WHERE date_created < NOW() - INTERVAL '12 hours';"

present=$(psql -h $DATABASE_HOST -p $DATABASE_PORT -U $DATABASE_USERNAME -d
$DATABASE_NAME -c "SELECT uri from aidedatas;")

for file in $files
do
       if [ -f "/upload/$file" ]; then
       rm "/upload/$file"
       fi
done
for file in "/upload"/*;
do
       if ! [[ "${present[*]}" =~ $(basename "$file") ]]; then
       rm "$file"
       fi
done
```

Service

Pour que l'*API* soit tout le temps en marche, même après le redémarrage du *VPS*, un service a été mis en place pour exécuter le *.jar* et maintenir l'*API* en état de marche. Voici le code de ce service, situé à l'emplacement */etc/systemd/system/smallbrother.service*:

```
[Unit]
Description=Smallbrother Service
After=network.target
StartLimitIntervalSec=10
StartLimitBurst=5

[Service]
Type=simple
Restart=always
RestartSec=1
User=root
EnvironmentFile=/etc/environment
ExecStart=/usr/lib/jvm/default-java/bin/java  -jar
/root/smallbrother/smallBrother-api-all.jar

[Install]
WantedBy=multi-user.target
```

Connexion à la base de données PostgreSQL

L'adresse, les paramètres en var d'env, l'attribution des droits sur OVH (en précisant que cela dépend du fournisseur.
Pour effectuer la connexion à la base de données PostgreSQL, hébergée chez OVH, utilisée pour stocker les données relatives aux captures de contexte, voici les différentes étapes:

1. récupérer les différentes données accessibles sur le tableau de bord OVH en rapport avec la base de données;
2. créer un utilisateur avec les droits d'administrateur sur le tableau de bord OVH;
3. ajouter les différentes valeurs nécessaires à la connexion à la base de données en variables d'environnement;
4. implémenter la fonction *init()* dans l'objet *DatabaseFactory* de la manière suivante:

```
fun init() {
   val driverClassName = "org.postgresql.Driver"
   val jdbcURL = "jdbc:postgresql://${System.getenv()["DATABASE_HOST"]}:" +
"${System.getenv()["DATABASE_PORT"]}/${System.getenv()["DATABASE_NAME"]}"
   val database = Database.connect(jdbcURL, driverClassName,
System.getenv()["DATABASE_USERNAME"].toString(),
System.getenv()["DATABASE_PASSWORD"].toString())
   transaction(database) {
       SchemaUtils.create(AideDatas, Admins)
   }
}
```

Certificat SSL

Pour sécuriser le transfert des données entre le serveur et les applications clientes, un certificat SSL a été obtenu via Certbot[8]. Pour obtenir ce certificat, il faut premièrement suivre les instructions recensées sur le site officiel[9] et ensuite exécuter les commandes suivantes:

- Convertir la clé et les certificats fournis par *Certbot* dans un format acceptable pour *Ktor* en remplaçant *yourDomain* par le domaine pour lequel le certificat *Certbot* a été créé et *example* par l'alias voulu pour le *KeyStore*:

```
openssl pkcs12 -export -out
/etc/letsencrypt/live/yourDomain/keystore.p12 -inkey
/etc/letsencrypt/live/yourDomain/privkey.pem -in
/etc/letsencrypt/live/yourDomain/fullchain.pem -name example
```

- Fournir un mot de passe pour le *KeyStore*
- Générer un *KeyStore Java* en remplaçant *yourDomain* par le domaine pour lequel le certificat *Certbot* a été créé et *example* par l'alias choisi pour le *KeyStore*:

```
keytool -importkeystore -alias example -destkeystore
/etc/letsencrypt/live/yourDomain/keystore.jks -srcstoretype PKCS12
-srckeystore /etc/letsencrypt/live/yourDomain/keystore.p12
```

Règles de redirection IP

Une fois le certificat obtenu, les règles de redirection IP ont dû être ajoutées pour rediriger le trafic des ports 80 et 443 vers le port 8080, utilisé pour accéder à l'API. Pour ce faire, voici les commandes à exécuter dans un terminal sur le VPS:

```
sudo iptables -A INPUT -p tcp --dport 8080 -j ACCEPT
sudo iptables -A INPUT -p tcp -m tcp --dport 443 -j ACCEPT
sudo iptables -A INPUT -p tcp -m tcp --dport 80 -j ACCEPT

sudo iptables -A PREROUTING -p tcp -m tcp --dport 443 -j REDIRECT
--to-ports 8080
sudo iptables -A PREROUTING -p tcp -m tcp --dport 80 -j REDIRECT
--to-ports 8080
```

Zone DNS

Une fois un nom de domaine obtenu auprès d'OVH, la zone DNS du serveur OVH doit être configurée selon les instructions du site officiel[10]. Il faut changer les enregistrements A de la zone DNS pour les faire correspondre à l'adresse IP du VPS.

[8] https://certbot.eff.org/
[9] https://certbot.eff.org/instructions?ws=other&os=ubuntufocal
[10] https://help.ovhcloud.com/csm/fr-dns-edit-dns-zone?id=kb_article_view&sysparm_article=KB0051684