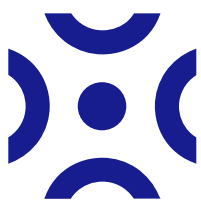


CONFROID

APPLICATION ANDROID ET BIBLIOTHEQUE DE GESTION CENTRALISEE DE CONFIGURATIONS

RAPPORT DE PROJET



**Université
Gustave Eiffel**

TABLE DES MATIERES

Description	3
Fonctionnalités.....	3
Sources des données	3
Editeur de configuration	3
Sauvegarde automatique vers le cloud	3
Disponibilité de l'application en plusieurs langues.....	3
Schéma fonctionnel	4
Architecture technique	5
Configuration	5
Conversions objet Java ⇔ Bundle ⇔ Configuration	5
Base de données.....	6
Service web.....	8
Serveur	8
Client	8
Sauvegardes automatiques	8
Stockage de fichiers	8
Sérialisation et désérialisation d'une configuration.....	8
Utilisation du stockage externe.....	9
Front-End.....	9
Paramètres	9
Récupération et affichage des configurations depuis plusieurs sources	10
Affichage des versions.....	11
Explorateur de configurations	11
Schéma – résumer de l'architecture technique.....	13
Gestion des intents	14
Bibliothèque utilitaire	14
Qualité du code	14
Gestion du projet	15
Difficultés rencontrées.....	15
Auteurs du projet.....	15

DESCRIPTION

Le projet Confroid consiste à réaliser une application Android permettant la gestion centralisée de configurations. Toute application installée sur l'appareil aura la possibilité de confier le stockage de sa configuration à Confroid plutôt que de réaliser cette tâche elle-même. L'utilisateur aura la possibilité de modifier la configuration d'une application en utilisant Confroid. Les configurations seront stockées selon un mode transactionnel avec un historique de versions.

Dans le but de faciliter l'utilisation de l'application Confroid, une librairie android permettant d'interagir avec Confroid est mise à disposition. Les utilisateurs de Confroid n'auront donc pas à se préoccuper de la gestion des intents, de l'authentification et autres.

Ce rapport a pour but d'expliquer l'architecture globale de l'application, sans traiter en détails les fonctionnalités ou le serveur web.

FONCTIONNALITES

SOURCES DES DONNEES

L'application permet de centraliser des configurations provenant de 3 sources différentes :

- Depuis une application tierce (via l'API mise à disposition notamment), les configurations seront alors sauvegardées dans une base de données SQLite.
- Depuis un service web.
- Depuis un fichier.

Il est possible de transférer une configuration d'une source de données vers une autre, par exemple récupérer une configuration depuis un service web puis l'exporter vers un fichier (nouveau ou déjà existant), ou de lire un fichier et sauvegarder une version spécifique dans la base de données.

EDITEUR DE CONFIGURATION

Une fois les configurations récupérées, il est possible de lister les noms de celles-ci, puis d'en récupérer les versions associées (uniques), et de les supprimer.

L'éditeur de configurations Confroid permet de naviguer dans une configuration et de modifier les valeurs à la volée. Des contraintes ou métadonnées peuvent être associées à un champ permettant de contrôler sa valeur, ou de l'afficher avec une vue spécifique.

Par exemple, il est possible de limiter la plage de valeurs d'un entier à un intervalle donné, de valider la valeur selon un prédicat, d'importer le numéro de téléphone d'un contact si le champ correspond à un numéro de téléphone, de saisir ses coordonnées GPS et d'autres.

SAUVEGARDE AUTOMATIQUE VERS LE CLOUD

Le service web peut être également utilisé comme moyen de sauvegarde automatique dans un cloud (de manière sécurisée), en spécifiant un intervalle de durée auquel les configurations locales (base de données) pourront être mises à jour régulièrement et automatiquement. Cette tâche de fond peut être paramétrée pour n'être exécutée que selon certains critères : batterie de l'appareil à un niveau acceptable, lorsque l'appareil est connecté en Wi-Fi uniquement ou lorsqu'il est branché à une source d'énergie.

DISPONIBILITE DE L'APPLICATION EN PLUSIEURS LANGUES

Toutes les chaînes de caractères sont écrites dans le dossier ressources de l'application, où une traduction est disponible en anglais et en français.

D'autres langues sont facilement implémentables, en rajoutant uniquement un xml correspondant à la langue voulue.

SCHEMA FONCTIONNEL

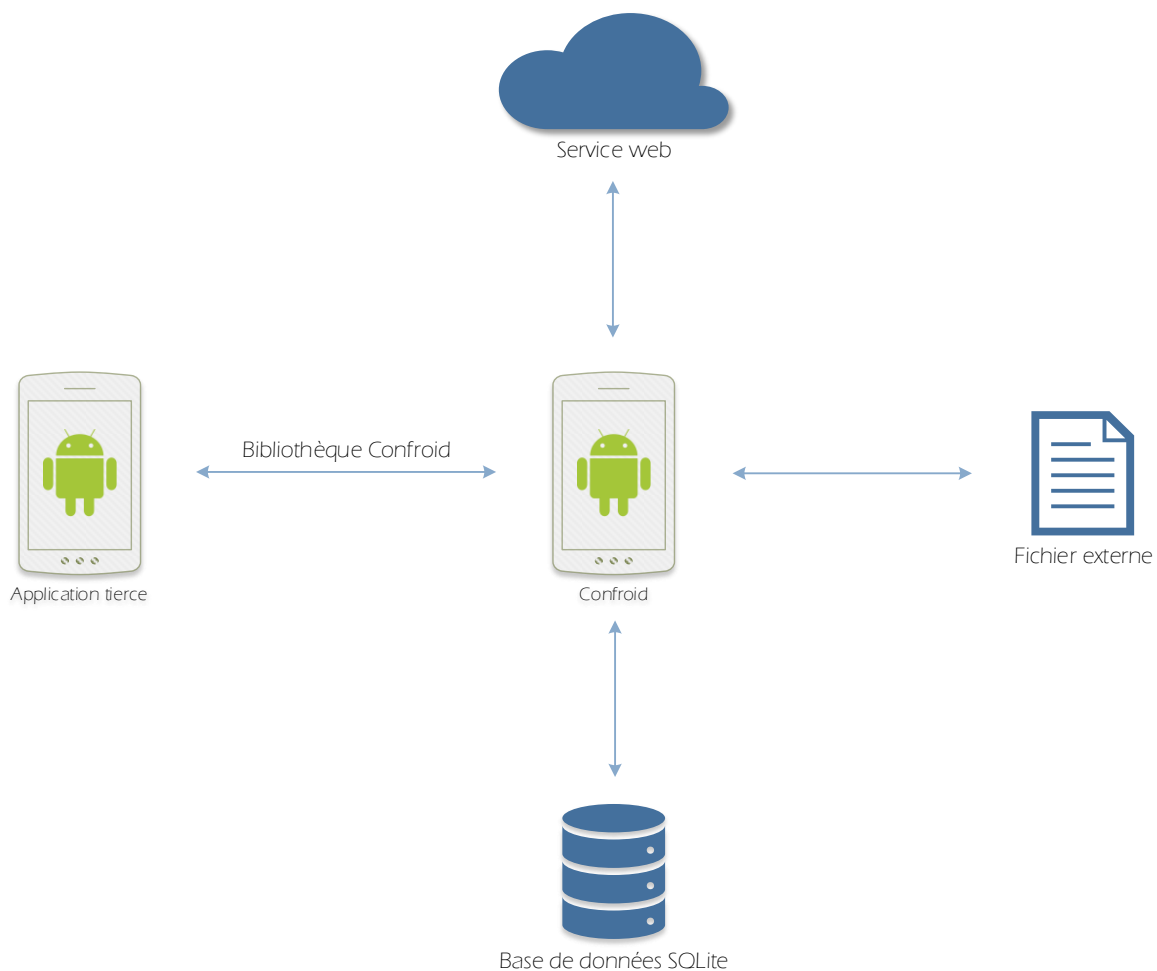


Figure 1 Schéma fonctionnel

ARCHITECTURE TECHNIQUE

L'architecture de l'application se découpe en plusieurs parties indépendantes, identifiables par le [schéma fonctionnel](#).

L'application est développée entièrement en Java, la version minimale du SDK Android est 24.

CONFIGURATION

Une configuration est une structure de données récursive permettant de représenter les informations qu'un utilisateur souhaite stocker dans Confroid. La richesse des informations qu'il est possible de stocker dépend donc de l'expressivité qu'apporte cette structure de données.

Dans ce projet le format d'une configuration est très similaire à une structure de données JSON, il est ainsi possible de stocker des dictionnaires, des listes et bien sûr des primitives.

Voici la grammaire du format de données d'une configuration :

```
Configuration = value
value = dictionary | array | primitive
dictionary = n * (key to value)
key = string
array = n * (value)
primitive = string | byte | float | double | integer | long | boolean
```

On remarque qu'il y a le type double et long en plus de la grammaire proposée par le sujet, car ce sont 2 types relativement très utilisés, à contrario du type byte.

Une configuration peut donc être vue comme un arbre, où une primitive est une feuille.

Une limitation notable de cette structure de données est que les clés des dictionnaires doivent nécessairement être de type String.

CONVERSIONS OBJET JAVA ⇔ BUNDLE ⇔ CONFIGURATION

Un des principaux atouts de cette application est sa faculté à transformer un objet java quelconque (en se limitant tout de même au format d'une configuration) en configuration et vis-versa. Cela est possible en utilisant l'API de réflexion de Java.

L'utilisateur n'ayant pas accès à la classe représentant une configuration, l'objet doit d'abord être transformé en Bundle qui sera envoyé à Confroid puis transformé en Configuration. Il y a donc deux étapes de transformation, en plus de la conversion inverse.

Les champs des objets ainsi que leur valeur sont retranscrits dans le bundle de façon récursive. On ajoute également le type instancié du champ pour permettre la reconstruction de l'objet. De plus pour alléger la taille et pour permettre l'inclusion circulaire des objets entre eux, on sauvegarde un identifiant unique pour chaque référence d'objet, ce qui permet de ne renseigner que l'identifiant d'un objet si celui-ci se retrouve référencé plus d'une fois dans « l'objet racine ».

Il est également possible en plus des champs de sauvegarder des annotations sur les champs.

Dans un second temps, le bundle sera ensuite transformé en Configuration, dans laquelle les objets seront traduits par des **Dictionary**, les différents types de listes et tableaux par des **Array** et les valeurs primitives (y compris String) par des **Primitive**, qui constitueront les feuilles de l'arbre.

La transformation inverse est possible pour transformer une Configuration en objet, grâce aux métadonnées enregistrées dans la Configuration.

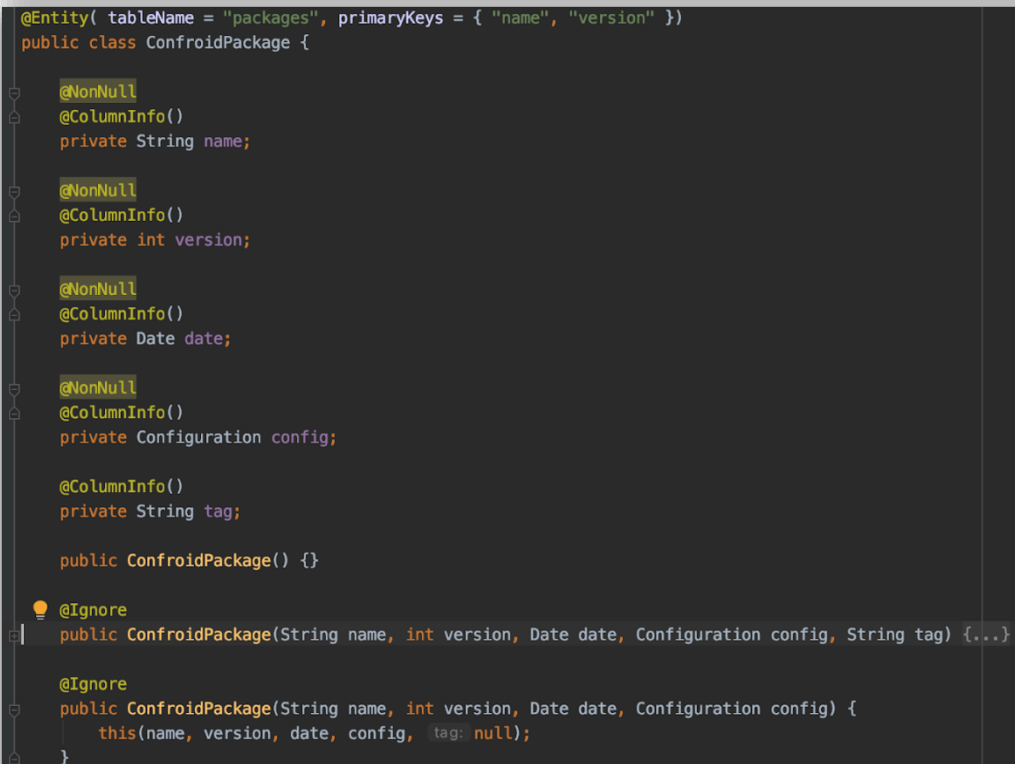
L'application permet ainsi de créer une Configuration depuis un objet Java en passant par un Bundle. L'objet Java peut être une primitive, un tableau ou n'importe quel autre objet avec seule contrainte que les types dérivés de Map doivent avoir une String comme type de clé.

BASE DE DONNEES

Pour la sauvegarde des configurations, nous avons choisi de les stocker dans une base de données en utilisant le moteur **SQLite** qui est embarqué dans les téléphones Android. Elles sont stockées dans une table avec les colonnes (name: **string**, version: **int**, tag: **string**, date: **long**, config: **string**). La colonne config est celle qui contient les données de la configuration au format JSON.

Pour interagir avec SQLite, on utilise l'API **Room** d'Android, qui est un ORM (Object Relational Mapping) permettant d'abstraire l'interaction avec SQLite. Dans notre architecture room, nous avons :

- Une classe **ConfroidPackage** transformée en une table par Room.



```
@Entity( tableName = "packages", primaryKeys = { "name", "version" })
public class ConfroidPackage {

    @NonNull
    @ColumnInfo()
    private String name;

    @NonNull
    @ColumnInfo()
    private int version;

    @NonNull
    @ColumnInfo()
    private Date date;

    @NonNull
    @ColumnInfo()
    private Configuration config;

    @ColumnInfo()
    private String tag;

    public ConfroidPackage() {}

    @Ignore
    public ConfroidPackage(String name, int version, Date date, Configuration config, String tag) {...}

    @Ignore
    public ConfroidPackage(String name, int version, Date date, Configuration config) {
        this(name, version, date, config, tag: null);
    }
}
```

Figure 2 Classe ConfroidPackage représentant également une table

- Une interface **ConfroidPackageDAO** définissant la liste des requêtes permettant de créer et de lire des données depuis la table.

```

@Dao
public interface ConfroidPackageDao {

    @Insert
    void create(ConfroidPackage confroidPackage);

    @Update
    void update(ConfroidPackage confroidPackage);

    @Delete
    void delete(ConfroidPackage confroidPackage);

    @Query("UPDATE packages SET tag = NULL WHERE name LIKE :name AND tag LIKE :tag")
    void removeTag(String name, String tag);

    @Query("SELECT * FROM packages WHERE name LIKE :name AND tag LIKE :tag LIMIT 1")
    ConfroidPackage findByTag(String name, String tag);

    @Query("SELECT * FROM packages WHERE name LIKE :name AND version LIKE :version LIMIT 1")
    ConfroidPackage findByVersion(String name, int version);

    @Query("SELECT * FROM packages WHERE name LIKE :name ORDER BY version DESC LIMIT 1")
    ConfroidPackage findLastVersion(String name);

    @Query("SELECT * FROM packages WHERE name LIKE :name ORDER BY version")
    List<ConfroidPackage> findAllVersions(String name);

    @Query("SELECT DISTINCT name FROM packages")
    List<String> findAllNames();

    @Query("SELECT * FROM packages WHERE name LIKE :name ORDER BY version DESC LIMIT 1")
    LiveData<ConfroidPackage> findLastVersionChanges(String name);

}

```

Figure 3 Listes des requêtes possibles sur la base de données

- Une classe **ConfroidDatabase** permettant d'accéder à la base de données.

```

@Database(entities = { ConfroidPackage.class }, version = 1)
@TypeConverters({ ConfroidConverters.class })
public abstract class ConfroidDatabase extends RoomDatabase {
    public abstract ConfroidPackageDao packageDao();

    public static void exec(Context context, Consumer<ConfroidPackageDao> consumer) {
        ConfroidDatabase db = Room.databaseBuilder(
            context, ConfroidDatabase.class, name: "confroid.db"
        ).allowMainThreadQueries().build();

        try {
            consumer.accept(db.packageDao());
        } finally {
            db.close();
        }
    }
}

```

Figure 4 Point d'accès à la base de données

L'implémentation de l'interface **ConfroidPackageDAO** est générée par Room à la compilation. L'API room va générer le code java nécessaire pour exécuter les requêtes définies par les annotations @Query et convertir les réponses dans les types de retour de nos méthodes.

L'utilisation de Room nous permet aussi de simplifier l'accès à notre base de données dans nos tests unitaires notamment grâce à sa méthode `Room.inMemoryDatabaseBuilder` qui crée une version in-memory de notre base durant nos tests pour ne pas polluer la base installée sur le téléphone virtuel.

SERVICE WEB

SERVEUR

Le projet inclut un service web avec une **API REST**, qui permet de stocker des configurations, pour un utilisateur. Le service est développé en Java avec **Spring Boot** et **Hibernate** et en utilisant une base de données PostgreSQL (configurable).

Cette base de données contient les tables des utilisateurs et des configurations cryptées selon la méthode indiquée dans le sujet. Ainsi le client (Confroid) peut se connecter au serveur en envoyant un login (en clair) et un mot de passe (crypté), le serveur effectue un cryptage supplémentaire (Bcrypt) puis authentifie le client selon les informations de la base de données et va lui attribuer une session.

Une fois authentifié, le client peut envoyer des packages contenant la configuration (cryptée également) et les métadonnées (nom, version et tag) envoyées en clair.

Si le serveur reçoit plusieurs fois un package avec le même nom et la même version (ou tag), la configuration est écrasée par la nouvelle.

Il est possible de récupérer toutes les versions correspondant à un nom d'une configuration, une version spécifique ou d'en supprimer.

CLIENT

L'application Confroid est un client du service web, la classe Client notamment permet de s'authentifier et d'interagir avec les fonctionnalités du serveur. Les paramètres de configuration (adresse du serveur, identifiant, mot de passe) sont paramétrables depuis une interface, et stockés dans la mémoire de l'application (*SharedPreferences*).

La classe Client gère également le cryptage et décryptage des informations, c'est notamment dans cette classe qu'est enregistrée la clé secrète.

Les méthodes fournies par cette classe sont asynchrones et possèdent des callbacks en cas de succès / échec des requêtes envoyées au serveur.

L'application utilise la bibliothèque [Volley](#) pour l'envoi et la réception de requêtes HTTP.

SAUVEGARDES AUTOMATIQUES

L'API Android **WorkManager** permet de soumettre une tâche régulière à l'application. En l'occurrence l'application envoie à intervalles réguliers (paramétrables) les configurations locales de la base de données au serveur web, si le client est correctement configuré.

Des contraintes de sauvegardes automatiques sont ajoutées et paramétrables depuis l'interface graphique. On peut ainsi choisir d'effectuer des sauvegardes / mises à jour lorsque l'appareil est à un niveau acceptable de batterie, qu'il est connecté à un réseau Wi-Fi seulement (pas de données cellulaires) ou s'il est nécessaire qu'il soit branché à une source d'énergie.

STOCKAGE DE FICHIERS

SERIALISATION ET DESERIALISATION D'UNE CONFIGURATION

La 3^{ème} source de données de l'application est les fichiers stockés sur le disque de l'appareil.

Confroid permet de sérialiser et de désérialiser une configuration sans perdre en information.

Étant donné le format utilisé pour une configuration, le format de (dés)sérialisation le plus évident est le JSON. Les méthodes `Configuration.toJson()` et `Configuration.fromJson(String json)` permettent

d'effectuer ces actions.

Ce format ne faisant pas la différence entre 2 types numériques différents, il est nécessaire d'ajouter à chaque valeur son type réel.

Il est possible également de construire une configuration « from scratch » en n'indiquant pas les informations permettant de reconstruire un objet, c'est-à-dire seulement avec les valeurs des configurations, elles seront affichables, éditables, enregistrables mais il ne sera pas possible de créer un objet avec la bibliothèque Conifold.

UTILISATION DU STOCKAGE EXTERNE

L'interface graphique propose un onglet permettant de lire les configurations d'un fichier présent sur le disque, avec un bouton d'importation prévu à cet effet.

La lecture d'un fichier se fait avec une Uri récupérée en lançant une activité de navigation de fichiers Android, grâce à l'[API Storage Access Framework](#).

C'est également cette activité qui est lancée lorsqu'on souhaite exporter une configuration depuis l'explorateur de configurations.

L'application permet ainsi d'importer et d'exporter des configurations, dans un fichier séparé à chaque fois ou bien en cumulant les configurations dans un seul fichier.

FRONT-END

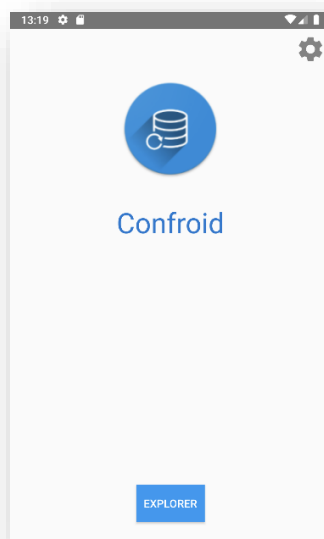


Figure 5 Écran d'accueil de l'application

La partie Front-End de notre application est constituée d'une série d'activités et de fragments permettant de lister toutes les configurations stockées ou récupérées par Conifold et de les modifier grâce à des pages générées dynamiquement.

PARAMETRES

L'icône de réglages en haut à droite de l'activité principale redirige vers une nouvelle activité dont le layout contient seulement un *FrameLayout* qui sert de conteneur pour les « sous-layout » des paramètres. On a donc 3 autres layouts et fragments associés pour afficher des *PreferenceScreen*.

Les paramètres qui sont affichés avec les layout Preference possèdent une clé qui sont enregistrés dans les *SharedPreferences* de l'application.

Ainsi il n'est pas nécessaire de reconfigurer le client web, les paramètres de sauvegarde et autres à chaque redémarrage de l'application.

CONNEXION AU SERVEUR WEB

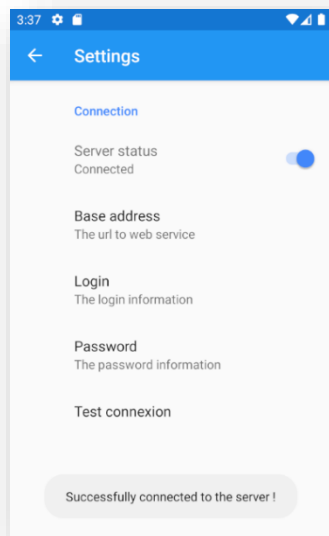


Figure 6 Page de configuration du serveur web

Un des écrans des paramètres permet de renseigner ses informations de connexion et d'effectuer un test de connexion au serveur. Si le client n'est pas connecté, il ne pourra pas profiter des fonctionnalités du service web, à savoir la sauvegarde et la récupération de configurations.

RECUPERATION ET AFFICHAGE DES CONFIGURATIONS DEPUIS PLUSIEURS SOURCES

Une fois qu'un utilisateur clique sur le bouton « Explorer » de l'activité principale, il est redirigé vers l'activité **ConfigBrowserActivity** qui permet de lister les configurations provenant de différentes sources de données connues de Conifold.

Le layout de cette activité est constitué d'un *TabLayout* pour lister les sources de données dans des onglets. Ce *TabLayout* est associé à un *ViewPager* qui permet de contrôler la navigation entre les onglets représentés par des *Fragments*. Les 3 onglets du *TabLayout* ont chacun une instance du même fragment

ConfigNamesFragment qui liste les noms des configurations d'une source de données dans un *RecyclerView*.

Chaque élément du *recycler* affiche le nom de l'application propriétaire de la configuration ainsi que le logo de cette application (l'icône enregistrée dans son manifeste) et par défaut l'icône d'Android.

Le *RecyclerView* de cette page comme tous les *recycler* de notre application est constitué d'un *Adapter* dans le package `fr.uge.confroid.front.adapters`, d'une classe *Model* qui est la représentation Java des éléments du *recycler* dans le package `fr.uge.confroid.front.models` et une classe *Holder* dans le package `fr.uge.confroid.front.holders`.

La gestion des événements du *RecyclerView* est toujours effectuée dans les classes **Models** ainsi les **Holders** permettent uniquement d'initialiser les éléments graphiques.

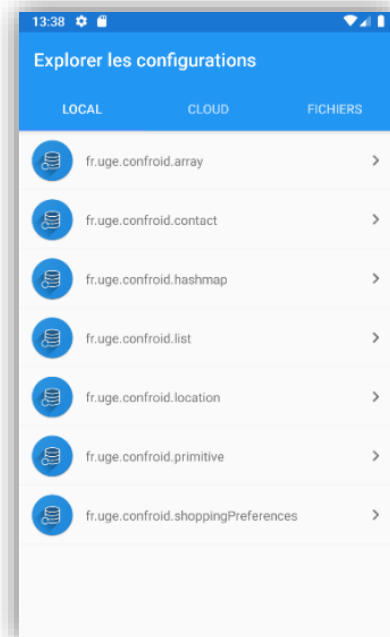


Figure 7 *TabLayout* répertoriant les différentes sources de données

AFFICHAGE DES VERSIONS

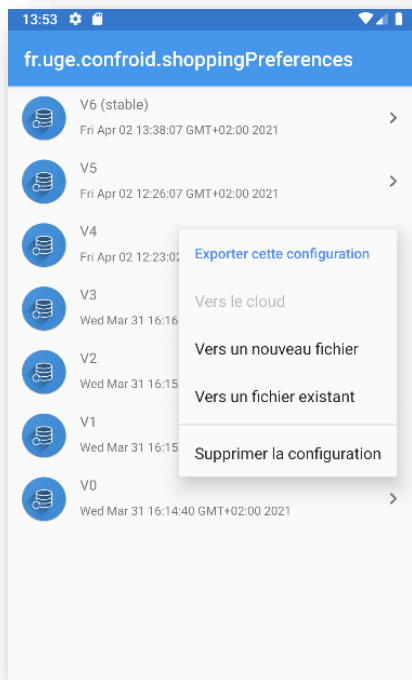


Figure 8 Menu contextuel après appuis long sur une version

En cliquant sur un des éléments du recycler de l'activité précédente, l'utilisateur est redirigé vers l'activité **ConfigVersionsActivity**.

Le layout de cette activité est constitué d'un nouveau *RecyclerView* qui liste les versions d'une configuration de la plus récente à la plus ancienne. En effectuant un clic long sur un item du *RecyclerView*, un menu contextuel s'affiche pour proposer différentes actions réalisables sur la version (suppression, export vers un fichier...). La liste des items est ainsi modifiée et actualisé sans relancer l'activité avec la fonction de notification `Adapter.notifyDataChanged()`.

EXPLORATEUR DE CONFIGURATIONS

En cliquant sur une version dans l'activité précédente, l'activité **ConfigEditorActivity** s'affiche. Elle permet d'explorer et d'éditer le contenu d'une configuration.

Cette page est constituée d'un simple *FrameLayout* qui fait office de conteneur pour afficher le contenu d'une configuration dans des fragments. L'édition d'une configuration consiste en l'empilement d'une suite de fragments pour éditer la valeur d'objet si c'est une primitive (nombre, string, boolean) ou lister ses champs dans le cas contraire. Par exemple, pour éditer une configuration qui est une classe ayant 2 champs (un entier, et un objet B), l'activité commence par afficher les 2 champs dans un *RecyclerView*.

- En cliquant sur le champ correspondant à l'entier, un fragment permettant d'éditer un nombre dans un *EditText* s'affiche.
- En cliquant sur le champ B, un fragment permettant de lister les champs de l'objet B s'affiche.
- En cliquant sur un champ de B, si celui-ci est une primitive, alors un fragment permettant d'éditer la primitive s'affiche sinon un fragment permettant de lister ses sous-champs.

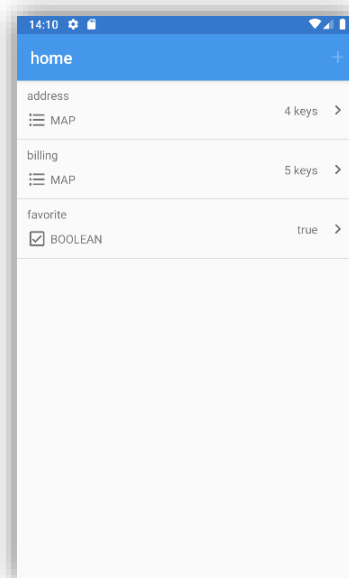


Figure 9 Editeur de champs

L'algorithme est récursif, ainsi on peut ainsi éditer de manière récursive un objet et ses sous-champs tout en restant dans la même activité.

Pour l'implémentation de cet algorithme, l'activité **ConfigEditorActivity** déclare une liste *openers* qui permet d'enregistrer des objets implémentant l'interface **EditorOpener**. Ces objets déclarent une méthode `Fragment createEditor()` qui renvoie un fragment permettant d'éditer un objet et une méthode `canHandle(EditorArgs args)` qui indique si le fragment peut éditer l'objet en question.

Il existe 6 implémentations de l'interface :

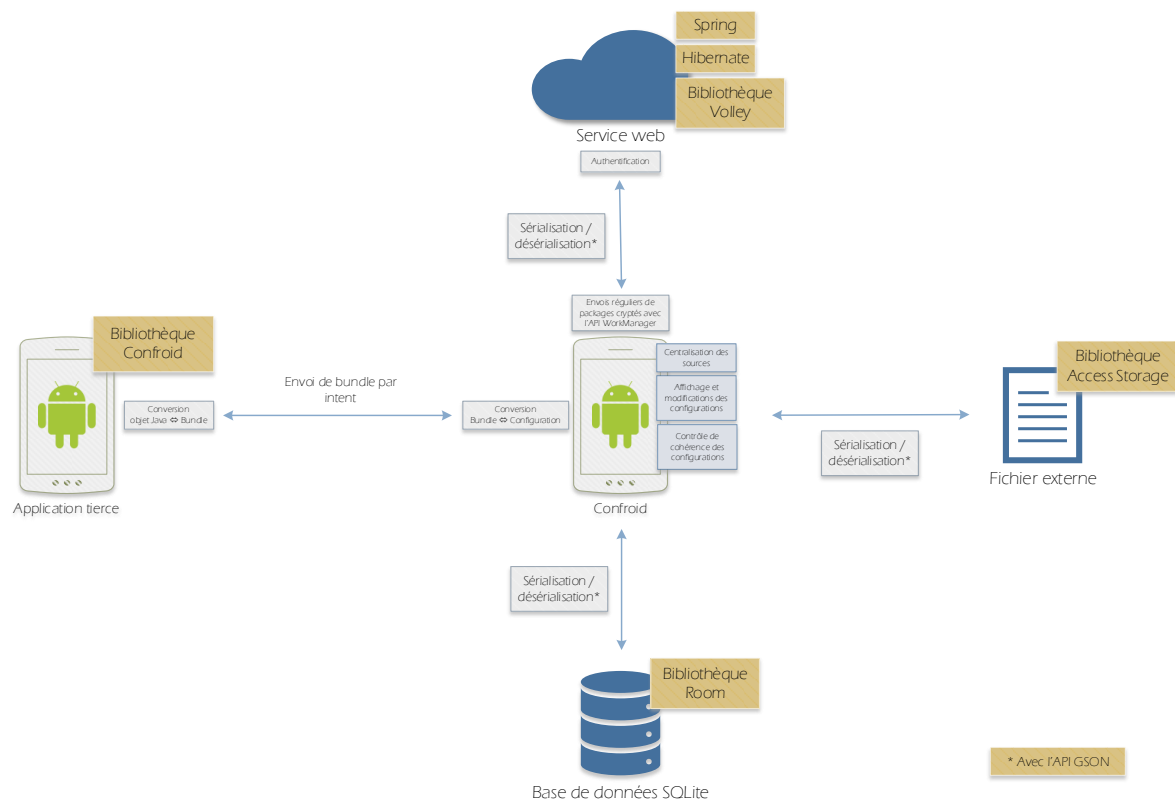
- `ArrayEditorFragment.Opener` pour lister les éléments d'un tableau dans une recycler
- `MapEditorFragment.Opener` pour lister champs d'un objet dans une recycler
- `BoolEditorFragment.Opener` pour éditer un boolean à l'aide d'un Switch
- `TextEditorFragment.Opener` pour éditer une String ou un nombre dans un *EditText*
- `GeoCoordinatesEditorFragment.Opener` pour éditer un tableau ayant une annotation **@GeoCoordinates** dans une *View* qui contient 2 *EditText* pour la latitude et la longitude ainsi qu'un bouton permettant de définir ces 2 valeurs depuis la localisation de l'appareil.
- `PhoneNumberEditorFragment.Opener` pour éditer une chaîne de caractères ayant une annotation **@PhoneNumber** dans une *View* qui contient un *EditText* pour afficher la valeur de la chaîne et un bouton pour définir la valeur à partir des infos d'un contact dans le répertoire de l'utilisateur.

La gestion des annotations est possible dans cette implémentation car pour éditer un objet, c'est le premier *opener* qui renvoie `true` lors d'un appel de sa méthode `canHandle()` qui est affiché. Donc il faut juste déclarer ces *openers* avant les autres. Ainsi même si un objet est une chaîne de caractères, s'il a une annotation **@PhoneNumber**, c'est l'*opener* `PhoneNumberEditorFragment.Opener` qui sera pris en compte car il est déclaré avant `TextEditorFragment.Opener`.

Chacun de ces fragments hérite de la classe **EditorFragment** qui déclare une méthode abstraite `onUpdateArgs` qui permet de passer l'objet à éditer au fragment et définit une méthode `updateValue` pour indiquer à l'activité **ConfigEditorFragment** que l'objet à éditer a changé et une méthode `pushEditor` pour lancer d'édition d'un sous-champ dans un autre fragment.

A la fin de l'édition d'une configuration (lorsque tous les fragments sont détruits) l'activité notifie la méthode l'ayant instancié grâce à un appel à la méthode `sendBroadcast` avec un paramètre *action* qui est défini dans les extras de l'intent de lancement de l'activité.

SCHEMA – RESUMER DE L'ARCHITECTURE TECHNIQUE



GESTION DES INTENTS

Pour qu'une autre application puisse communiquer avec Confroid, nous avons mis en place les services **ConfigurationPusher** pour envoyer une configuration à sauvegarder sous forme de bundle, un service **ConfigurationPuller** pour récupérer une version spécifique d'une configuration et un service **ConfigurationVersions** pour lister toutes les versions d'une configuration. Ces 3 services héritent toutes de la classe **BackgroundService** qui contient le code nécessaire pour qu'un service puisse tourner en arrière-plan dans les nouvelles versions d'Android (SDK >= 26).

Pour que les services puissent envoyer des réponses aux méthodes appelantes, ceux-ci doivent passer un paramètre « receiver » associé à l'intent du service. Ce paramètre est une chaîne de caractères qui représente le nom d'un **BroadcastReceiver** à appeler grâce à la méthode `sendBroadcast(new Intent(receiver))`.

BIBLIOTHEQUE UTILITAIRE

En plus de l'application Confroid, le projet inclut une bibliothèque proposant une API permettant d'interagir automatiquement avec l'application Confroid.

C'est cette bibliothèque qui permet notamment de transformer un objet en Bundle et de l'envoyer à l'application via Confroid.

Les méthodes fournies dans la classe **ConfroidUtils** envoient des intents permettant de lancer l'activité d'édition de configuration (`editObject()` ou `updateObject()` par exemple) ou d'exécuter un service Confroid (avec `saveConfiguration()` et `loadConfiguration()` entres autres).

Pour authentifier les applications auprès de Confroid, les méthodes de l'api commencent toutes par une phase d'authentification avec un appel à la méthode privée `withAuth` qui prend un context en paramètre et un callback qui sera appelé avec le token envoyé à Confroid. Cette méthode extrait l'identification de l'application depuis le Context qui lui est passé en paramètre. Ensuite il essaye de lire le token attribué à l'application dans les préférences de cette dernière. Si le token n'existe pas (première utilisation de Confroid), alors la méthode envoie un message au *BroadcastReceiver TokenDispenser* de Confroid pour créer ou récupérer le token associé à l'application. Confroid sauvegarde aussi le token créé dans ses propres préférences.

QUALITE DU CODE

Une attention particulière a été apporté au code de l'application et de la librairie lors de son développement. Ainsi l'architecture a été soigneusement pensée et les 84 classes, sans compter les classes de tests et les layouts ont chacune leur utilité.

A chaque fonctionnalité est associée une classe de test avec JUNIT, dans les packages « *Test* » si les tests sont spécifiques au code Java, ou dans les packages « *AndroidTest* » si les tests nécessitent une fonctionnalité propre à Android, comme l'utilisation de Context.

De plus, le code est suffisamment documenté, avec des noms de classes et de méthodes parlantes, des packages structurés, et surtout une **javadoc** complète et à jour.

Ainsi l'application et l'ensemble du projet est clair, maintenable et évolutif.

De plus, une classe **Demo** est incluse dans le code, qui propose une méthode `create()` permettant d'ajouter des données de démonstration dans la base de données.

Enfin, une application à part entière est également fournie avec le projet, pour permettre de tester les envois de Bundle via intents.

GESTION DU PROJET

Avant de commencer le développement de l'application, nous avons découpé le projet en plusieurs parties indépendantes, ce qui nous a permis de se répartir aisément les tâches, de plus le code étant bien documenté et lisible, il a été facile pour l'un d'entre nous de revenir sur une fonctionnalité d'une autre personne.

Un dépôt sur **Google Drive** a été mis en place pour la gestion interne (brouillons d'architecture, découpage et répartition des features...).

Le dépôt versionné des sources est accessible à [cette adresse](#). Un soin a également été apporté quant à l'utilisation de Git avec notamment l'utilisation des branches pour chaque grosse fonctionnalité.

DIFFICULTES RENCONTREES

Il n'y a pas eu de difficultés majeures rencontrées lors de l'avancement du projet si ce n'est le temps et les moyens humains mis à disposition.

Cependant certains points nous ont ralenti lors du développement comme la conversion d'objet Java en configuration et inversement. En effet de nombreux cas d'usage étaient à prévoir, et la manipulation de la structure d'une configuration (similaire à une bibliothèque JSON) n'est pas des plus aisées.

Enfin la documentation des bibliothèques Android en général a été omniprésente.

AUTEURS DU PROJET

CISSE Mamadou	mcisse06@etud.u-pem.fr
SANCHES FERNANDES Stéphane	ssanches@etud.u-pem.fr
TRAN Éric	etran06@etud.u-pem.fr
WADAN Samy	swadan@etud.u-pem.fr
ZEMMOUR Axel	azemmour@etud.u-pem.fr
ZULAL Volkan	vzulal@etud.u-pem.fr