UNamur

Faculté d'informatique

INFOM218 :Evolution de systèmes logiciels

Rétro-Ingénierie d'une base de donnée

Auteurs :
Rosny Anderson Fodja Kouanga
Noé Pozza
Ryad HARITTANE

7 janvier 2025

Table des matières

1 Introduction

Ce rapport décrit l'analyse de la base de données d'une application Android open-source, spécifiquement en ce qui concerne l'identification des requêtes SQL utilisées dans l'application grâce à l'outil SQLInspect et d'en dégager les schéma correspondants.(a détailler et enrichir au fur et à mesure)

2 Analyse du code DDL

Dans cette section, nous allons commencer notre retro-ingénierie de base de donnée par l'analyse du code DDL présent dans la code base de l'application. Ceci nous permettra de ressortir des informations tant bien sur l'aspect technique que sur la signification des données. Pour facilité notre travail, on a aggloméré les informations dans un fichier *MarkDown*.

2.1 Classe d'intérêt

En premier lieu, il est nécessaire de trouver la, ou les, partie de l'application responsable de la gestion de la base de donnée. Le rapport fournit par SQLInspect nous dirige vers deux classes : *DatabaseBackend* et *UnifiedPushDatabase*. Le point commun de ces deux classes est qu'elles sont des filles de la classe abstraite *SQLiteOpenHelper*. Cette classe permet la gestion de base de donnée SQL.

On connaît donc le moteur utilisé, qui est SQLite, sans surprise pour une application mobile. Et on sait qu'il y a deux bases de données due à l'existance de deux classes implémentant la classe *SQLiteOpenHelper*.

2.2 *DatabaseBackend*

La classe *DatabaseBackend* définit la base de donnée nommée *History*. Celle-ci semble avoir la responsabilité de contenir les données relatives aux messages, à la sécurité, aux contactes et aux serveurs. Elle est donc la base de donnée principale de cette application Android.
Elle contient 11 tables différentes. Nous avons reconstitué les requêtes DDL à l'origine des classes afin de pouvoir recréer le schéma physique. Voici la liste :

2.2.1 Contact

Contient les informations sur un contact. Elle fait partie de la fonctionnalité de gestion des contacts. Il y est définit une relation avec la table *Account*. L'application permettant de gérer plusieurs comptes, il est nécessaire de lier un contact à un compte.

Un contact peut être aussi un groupe, ou plus formelement, un *Multi-User Chat (MUC)*. C'est une fonctionnalité du protocole XMPP, qui permet de créer des salons de discussion.

On observe aussi qu'il n'y a pas de primary key, mais des contraintes unicités sont appliquées sur les colonnes de *accountUuid* et *jid*. Cela a comme objectif de ne pas pouvoir enregistrer deux fois le même contacte. Cependant, il semble que les colonnes *jid* et *AccountUuid* soit utilisées comme une clé primaire [1], ce qui peut poser problème car une valeur *null* est possible. *jid* signifie *Jabber ID*, qui est un identifiant utilisé pour le protocole XMPP, protocole de messagerie instantané avec lequel l'application fonctionne.

Il est aussi à noter qu'elle est une des rares classes à ne pas hériter de la classe *AbstractEntity*, ce qui se caractérise par l'absence de la colonne *uuid*.

DDL

```
create table contacts(
    accountUuid TEXT,
    servername TEXT,
    systemname TEXT,
    presence_name TEXT,
    jid TEXT,
    pgpkey TEXT,
    photouri TEXT,
    options NUMBER,
    systemaccount NUMBER,
    avatar TEXT,
    last_presence TEXT,
    last_time NUMBER,
    rtpCapability TEXT,
    groups TEXT,

    FOREIGN KEY(accountUuid) REFERENCES accounts(uuid)
     ON DELETE CASCADE,
    UNIQUE(accountUuid, jid) ON CONFLICT REPLACE
);
```

1. eu.siacs.conversations.entities.Contact 115 : *Jid.of* n'accepte pas de valeur null

2.2.2 Account

Contient les informations sur un compte. Table centrale de cette base de donnée, elle permet la gestion des comptes XMPP. Il est donc à noter que les colonnes *username*, *server* et *ressource* fait donc référence au *jid* utilisé par le protocole XMPP. Il prend la forme de *username@server/ressource*[2].

Tout comme la table *Contact*, le jid semble ne pas pouvoir être null. Dans cette table, on remarque que une requête avec un *username* ou un *server* null renvoie une erreur[3].

Il y a la présence d'une clé primaire, *uuid*, qui est une chaîne de caractères générée aléatoirement.

DDL

```
create table accounts(
    uuid TEXT PRIMARY KEY,
    username TEXT,
    server TEXT,
    password TEXT,
    display_name TEXT,
    status TEXT,
    status_message TEXT,
    rosterversion TEXT,
    options NUMBER,
    avatar TEXT,
    keys TEXT,
    hostname TEXT,
    resource TEXT,
    pinned_mechanism TEXT,
    pinned_channel_binding TEXT,
    fast_mechanism TEXT,
    fast_token TEXT,
    port NUMBER DEFAULT 5222
);
```

2.2.3 Conversation

Cette table contient les informations sur une conversation entre un compte et un contact. Elle est donc une table de liaison entre les tables *Account* et *Contact*. Elle contient aussi des informations sur le nom de la conversation, le statut, le mode et des attributs.

2. xmpp documentation
3. eu.siacs.conversations.entities.Account 176-180 : *Jid.of* n'accepte pas de valeur null

Deux colonnes se reporte implicitement à la table *Contact*. Il s'agit de *contactUuid* et *contactJid*. La première colonne est étrange, tenu du faite que la table *Contact* ne contient pas de colonne *uuid*. Il est possible que cette colonne soit une erreur de conception. La seconde colonne, *contactJid*, est une référence au *jid* du contact (qui peut donc être un groupe). Ce qui tend à confirmer que le *jid* est une clé primaire implicite de la table *Contact*[4].

Fille de la classe *AbstractEntity*, elle contient une clé primaire, *uuid*, qui est une chaîne de caractères générée aléatoirement.

DDL

```
create table conversations (
    uuid TEXT PRIMARY KEY,
    name TEXT,
    contactUuid TEXT,
    accountUuid TEXT,
    contactJid TEXT,
    created NUMBER,
    status NUMBER,
    mode NUMBER,
    attributes TEXT,

    FOREIGN KEY(accountUuid) REFERENCES accounts(uuid)
     ON DELETE CASCADE
);
```

2.2.4 Message

Contient les informations sur un message. Il est à noter que la colonne *conversationUuid* est une clé étrangère vers la table *Conversation*. Cette table contient aussi des informations sur le contenu du message et sur l'apect technique du message, comme le type, le statut, le temps d'envoi, etc.

La colonne *trueCounterpart* est une colonne qui sert à stocker le *jid* du contact, lorsque la conversation est un groupe. Cela permet de savoir qui a envoyé le message. Elle est donc en un sens une clé étrangère implicite vers la table *Contact*[5].

Une table virtuelle est aussi présente. Sous le nom de *messages_index*, elle permet un recherche rapide des messages. Elle est donc une table de cache. Afin de maintenir à jour cette table, des triggers sont utilisés afin que chaque modification, ajout, suppression de la table *Message* soit répercutée sur la table *messages_index*.

4. eu.siacs.conversations.entities.Conversation 673 : Seule la colonne *contactJid* est utilisé comme lien avec la table *Contact*
5. eu.siacs.conversations.entities.Message 352 : *trueCounterpart* est utilisé pour la relation avec *Contact*

DDL

```
create table messages(
    uuid TEXT PRIMARY KEY,
    conversationUuid TEXT,
    timeSent NUMBER,
    counterpart TEXT,
    trueCounterpart TEXT,
    body TEXT,
    encryption NUMBER,
    status NUMBER,
    type NUMBER,
    relativeFilePath TEXT,
    serverMsgId TEXT,
    axolotl_fingerprint TEXT,
    carbon INTEGER,
    edited TEXT,
    read NUMBER DEFAULT 1,
    oob INTEGER,
    errorMsg TEXT,
    readByMarkers TEXT,
    markable NUMBER DEFAULT 0,
    deleted NUMBER DEFAULT 0,
    bodyLanguage TEXT,
    occupantId TEXT,
    reactions TEXT,
    remoteMsgId TEXT,

    FOREIGN KEY(conversationUuid) REFERENCES conversations(uuid)
     ON DELETE CASCADE
);
```

2.2.5 Identity

Table servant à stocker les informations sur les identités. Il s'agit des informations sur les clés cryptographiques. Elle contient des informations sur le compte, le nom, la clé, le certificat, la confiance, etc.

Étant lier à l'aspect sécurité de l'application, nous n'allons pas nous attarder sur les détails de cette table.

DDL

```
CREATE TABLE identities(
    account TEXT,
    name TEXT,
    ownkey INTEGER,
```

```
    fingerprint TEXT,
    certificate BLOB,
    trust TEXT,
    active NUMBER,
    last_activation NUMBER,
    key TEXT,

    FOREIGN KEY(account) REFERENCES accounts(uuid)
     ON DELETE CASCADE
    UNIQUE( account, name, fingerprint)
     ON CONFLICT IGNORE
);
```

2.2.6 Session

Table servant à stocker les informations sur les sessions. Elle contient des informations sur le compte, le nom, l'identifiant du device, la clé, etc.

Étant lier à l'aspect sécurité de l'application, nous n'allons pas nous attarder sur les détails de cette table.

DDL

```
CREATE TABLE sessions(
    account TEXT,
    name TEXT,
    device_id INTEGER,
    key TEXT,

    FOREIGN KEY(account) REFERENCES accounts(uuid)
     ON DELETE CASCADE,
    UNIQUE( account, name, device_id)
     ON CONFLICT REPLACE
);
```

2.2.7 Prekey

Étant lier à l'aspect sécurité de l'application, nous n'allons pas nous attarder sur les détails de cette table.

DDL

```
CREATE TABLE prekeys(
```

```
    account TEXT,
    id INTEGER,
    key TEXT,
    FOREIGN KEY(account) REFERENCES accounts(uuid)
     ON DELETE CASCADE,
    UNIQUE( account , id)
     ON CONFLICT REPLACE
);
```

2.2.8 Signed Prekey

Étant lier à l'aspect sécurité de l'application, nous n'allons pas nous attarder sur les détails de cette table.

DDL

```
CREATE TABLE signed_prekeys(
    account TEXT,
    id INTEGER,
    key TEXT,

    FOREIGN KEY(account) REFERENCES accounts(uuid)
     ON DELETE CASCADE,
    UNIQUE( account , id)
     ON CONFLICT REPLACE
);
```

2.2.9 Resolver Result

Cette table contient les résultats de la résolution DNS. Elle contient des informations sur le domaine, le nom d'hôte, l'adresse IP, la priorité, le TLS direct, l'authentification, le port, etc.

Elle permet à l'application de garder en mémoire les résultats de résolution DNS pour une utilisation ultérieure.

DDL

```
create table resolver_results(
    domain TEXT,
    hostname TEXT,
    ip BLOB,
    priority NUMBER,
```

```
    directTls NUMBER,
    authenticated NUMBER,
    port NUMBER,

    UNIQUE(domain) ON CONFLICT REPLACE
);
```

2.2.10 Discovery Result

Cette table contient le résultat des découvertes de service disponible. Elle est liée aux fonctionnalités de XMPP [6].

DDL

```
create table discovery_results(
    hash TEXT,
    ver TEXT,
    result TEXT,

    UNIQUE(hash, ver) ON CONFLICT REPLACE
);
```

2.2.11 Presence Template

Contient les informations sur les patterne de présence. C'est-à-dire, un message que l'utilisateur veut afficher, et un status. Ce status peut être *chat*, *online*, *away*, *offline*, etc [7].

On remarque qu'aucune des colonnes n'est une clé primaire. Cependant, une contrainte d'unicité est appliquée sur les colonnes *message* et *status*. Cela permet de ne pas avoir deux fois le même message avec le même status.

DDL

```
CREATE TABLE presence_templates(
    uuid TEXT,
    last_used NUMBER,
    message TEXT,
    status TEXT,

    UNIQUE(message,status) ON CONFLICT REPLACE
```

6. XMPP Documentation
7. eu.siacs.conversations.entities.Presence 12

```
);
```

2.3 *UnifiedPushDatabase*

Cette classe définit la base de donnée nommée *unified-push-distributor*. Celle-ci semble avoir la responsabilité de contenir les données relatives aux notifications push. Elle ne contient qu'une seule table, *push*, qui contient les informations sur les notifications push.

2.3.1 Push

Contient les informations sur une notification push. Elle est relié au système de notification d'android.

Elle semble lié à la table *Account*, mais il n'y a pas de clé étrangère déclarée. Cependant, la colonne *account* est utilisée pour se référer à *accounts.uuid*, suggérant une clé étrangère implicite. L'utilisation de *account* comme clé étrangère implicite est sans doute contrainte par le fait que cette table et la table *Account* sont dans des bases de données différentes.

DDL

```
CREATE TABLE push (
        account TEXT,
        transport TEXT,
        application TEXT NOT NULL,
        instance TEXT NOT NULL UNIQUE,
        endpoint TEXT,
        expiration NUMBER DEFAULT 0
);
```

2.4 Résultats

2.4.1 Inférence des Clés Étrangères Explicites

2.4.2 Découverte de Clés Étrangères Implicites Additionnelles

3 Étape 3 : Analyse « What-If » et Scénarios d'Évolution

Introduction

Dans cette étape, nous appliquons une analyse « what-if » pour évaluer l'impact de 10 scénarios d'évolution sur le Sous-Schéma Physique (PSS) de notre système de base de données. Chaque scénario est accompagné d'une description détaillée et de son impact sur le code source, incluant les classes, méthodes, et requêtes SQL affectées.

Scénario 1 : Ajout de contrainte dans le moteur de base de données

Résumé du changement :

Lors de notre analyse du code Java et DDL de l'application, nous avons remarqué que certaines colonnes dans les tables de la base de données peuvent contenir des valeurs nulles et d'autres ne sont plus utilisée. Cependant le code Java ne semble pas gérer ces valeurs nulles. Pour éviter des erreurs potentielles, nous allons ajouter des contraintes *NOT NULL* sur ces colonnes.

Détail des changements :

Colonnes à modifier :

- Dans la table `contacts`, la colonne `jid` peut être `NULL`. Nous allons ajouter une contrainte `NOT NULL` pour garantir que chaque contact a un identifiant Jabber valide.
- Dans la table `accounts`, les colonnes `username` et `server` peuvent être `NULL`. Nous allons ajouter une contrainte `NOT NULL` pour garantir que chaque compte possède un server et un username. Ce qui revient à dire que chaque compte doit avoir un identifiant Jabber valide.
- Dans la table `conversation`, la colonne `contactUuid` ne semble plus re utilisée car ne référence pas une colonne `uuid` dans la table `contact`. Nous allons supprimer

cette colonne. La colonne `contactJid` ne dois aussi pas être null. De plus, cette colonne et la colonne `contactUuid` doivent être unique.

Il est quand même sage de vérifier que les colonnes à modifier ne contiennent pas de valeurs nulles avant d'appliquer les contraintes `NOT NULL`.

Par simplicité, nous allons supprimer les instances qui possèdent des valeurs nulles pour les colonnes où on ajoute la contrainte `NOT NULL`. Mais il faudrait faire un backup de la base de données avant d'appliquer ces changements.

```
DELETE FROM contacts WHERE jid IS NULL;
ALTER TABLE contacts ALTER COLUMN jid SET NOT NULL;

DELETE FROM accounts WHERE username IS NULL OR server IS NULL;
ALTER TABLE accounts ALTER COLUMN username SET NOT NULL;
ALTER TABLE accounts ALTER COLUMN server SET NOT NULL;

DELETE FROM conversation WHERE jid IS NULL;
ALTER TABLE conversation DROP COLUMN contactUuid;
ALTER TABLE conversation ALTER COLUMN contactJid SET NOT NULL;
ALTER TABLE conversation ADD UNIQUE(contactJid, accountUuid)
 ON CONFLICT IGNORE;
```

Il est maintenant possible de déclarer explicitement la clé étrangère `contactJid` dans la table `Conversation`.

```
ALTER TABLE conversation ADD FOREIGN KEY(contactJid)
 REFERENCES contact(jid) ON DELETE CASCADE;
```

On notera que dans la classe `Contact`, seule la colonne `jid` profite de l'ajout de la contrainte `NOT NULL`. La colonne `accountUuid` n'en profite pas, car cela ne poserai pas de problème avec le code actuelle de l'application. Malgré tout, il est nécessaire d'avoir à la fois un `jid` et un `accountUuid` pour identifier un contacte.

Impact sur le code :

L'idée de ce senario est de s'assurer que les colonnes qui ne doivent pas être `NULL` ne le sont pas. Le code Java ne va pas être directement impacté par ces changement. Mais il faut tout de même faire tourner les tests pour s'assurer que l'application fonctionne toujours correctement.

Test à effectuer :

— Vérifier que les colonnes modifiées ne contiennent pas de valeurs nulles.

— Appliquer les contraintes `NOT NULL` et vérifier que les valeurs nulles sont supprimées.
— Tester les fonctionnalités principales de l'application pour s'assurer qu'elles fonctionnent toujours correctement.

Scénario 2 : Ajout d'une fonctionnalité d'appel

Résumé du changement :

Le protocole XMPP vient de se mettre à jour et permet maintenant de gérer des appels. Nous allons ajouter une fonctionnalité d'appel à l'application, en utilisant les fonctionnalités de base de l'application pour gérer les appels.

Détail des changements :

Strategie d'implémentation :

La majorité des modification nécessaire est lié au code Java. La base de données doit s'adapter à ces changements. Ici, nous somme dans un scénario imaginaire, mais on peut imaginer qu'il faut ajouter une table `calls` pour stocker les informations sur les appels. Cette table contiendrait des informations sur le temps de début et de fin de l'appel, la durée, etc. Elle sera en relation avec la table conversation, ce qui permet de minimiser la quantité de données à stocker et de faciliter les requêtes.

Mais aussi de nouvelle colonnes technique permettant d'effectuer des appels dans les tables `Account` et `Contact`. Par exemple, une colonne `voiceSecurityKey` dans la table `Account` pour stocker le mécanisme de chiffrement utilisé pour les appels.

Modification :

Dans un premiers temps nous allons ajouter une table `calls` pour stocker les informations sur les appels.

```
CREATE TABLE Calls (
        uuid TEXT PRIMARY KEY,
        conversationUuid TEXT,
        startTime NUMBER,
        duration NUMBER,
        status NUMBER,
```

```
        FOREIGN KEY(conversationUuid)
          REFERENCES conversation(uuid) ON DELETE CASCADE
);
```

Ensuite, nous allons ajouter des colonnes dans les tables `Account` et `Contact` pour stocker les informations nécessaires pour les appels.

```
ALTER TABLE accounts ADD COLUMN voiceSecurityKey TEXT;
ALTER TABLE contacts ADD COLUMN voiceSecurityKey TEXT;
```

Impact sur le code :

L'ajout d'une telle fonctionnalité nécessite des modifications importantes dans le code Java. Il faudra ajouter des classes, des méthodes, des requêtes SQL, etc. pour gérer les appels. Il faudra aussi mettre à jour l'interface utilisateur pour permettre aux utilisateurs de passer des appels.

Voici une liste non exhaustive des modifications à apporter dans le code Java :

- Ajout d'un package, ou tout du moins d'une classe, contenant la logique pour gérer les appels.
  - Possibilité d'ajouter des appels à la base de données à la fin de l'appel.
  - Permettre de démarrer et de terminer un appel.
  - Lister l'historique des appels. Exemple en utilisant des jointures mais bien sur, on preferera l'utilisation de l'ORM :

    ```
    SELECT Calls.*, Contacts.presence_name FROM Calls
    JOIN Conversation ON Calls.conversationUuid = Conversation.uuid
    JOIN Accounts ON Conversation.accountUuid = Accounts.uuid
    JOIN Contacts ON Conversation.contactJid = Contacts.jid
    AND Conversation.accountUuid = Contacts.accountUuid
    WHERE Accounts.uuid = 'target_account_uuid';
    ```
- Ajout d'un nouveau status possible pour définir un template de présence. Cellui-ci pourrait être `in-call`.
- Mise à jour de l'interface utilisateur pour permettre aux utilisateurs de passer des appels.

Test à effectuer :

- Faire des tests unitaires pour s'assurer que les appels sont correctement ajoutés à la base de données.
  - Que ce passerai t-il si l'appel n'a pas de conversation associée (valeur Null).
  - Que ce passerai t-il si aucune clé de sécurité a été définie pour l'appel.
  - Ajout et suppression d'appel.
- Tester les fonctionnalités d'appel pour s'assurer qu'elles fonctionnent correctement.

Scénario 3 : Ajout de la possibilité de lier un numéro de téléphone à un contact

Résumé du changement :

XMPP s'unifie avec les protocoles de messagerie traditionnels et permet maintenant d'envoyer des messages SMS et d'appeler des numéros de téléphone. Le numéro de téléphone remplace le *jid*, il n'est donc plus question de *jid* dans XMPP.

Nous allons ajouter la possibilité de lier un numéro de téléphone à un contact dans l'application. Et remplacer le *jid* par le numéro de téléphone dans les tables `contacts` et `Account`.

Dans ce scénario, nous allons ignorer que certains contacts peuvent être des groupes, et donc ne pas avoir de numéro de téléphone.

Détail des changements :

Strategie d'implémentation :

Dnas le cas de la table `Account`, il est possible de remplacer le *jid* par le numéro de téléphone. Cependant, dans les tables `Contacts`, le jid est utilisé comme pour une relation avec la table `Conversation` et `Message`. Il faudra donc modifier adéquatement les tables `Conversation` et `Message` pour utiliser le numéro de téléphone à la place du *jid*.

Pour le code Java, une solution permettant de ne pas modifier le code en profondeur est possible dans le cas où le `jid` est simplement remplacé par le numéro de téléphone. On peut ajouter une méthode dans l'ORM permettant de récupérer les numéros de téléphone dans les différentes classes, tout en gardant les methodes `getJid` qui serai un alias de `getPhoneNumber`. On devra, malgré tout, marquer le `getJid` comme `deprecated` afin éviter des erreurs dans le futur.

Modification :

Dans un premiers temps nous allons ajouter le numéro de téléphone dans les tables `Account` et `Contacts`.

```
ALTER TABLE accounts ADD COLUMN phoneNumber TEXT;
ALTER TABLE contacts ADD COLUMN phoneNumber TEXT;
```

Ensuite, il faudra attendre que l'utilisateur ajoute les numéro de téléphone, aussi bien le sien que celui de ses contacts.

Dès que cela est fait, il faudra aussi modifier les tables `Conversation` et `Message` pour utiliser le numéro de téléphone à la place du *jid*.

Pour la table `Conversation` :

```
ALTER TABLE conversation
ADD COLUMN contactPhoneNumber TEXT;

UPDATE conversation
SET contactPhoneNumber = (
        SELECT phoneNumber FROM contacts
        WHERE jid = contactJid
);

ALTER TABLE conversation DROP COLUMN contactJid;

ALTER TABLE conversation
ADD UNIQUE(contactPhoneNumber, accountUuid)
ON CONFLICT IGNORE;
```

Et pour la table `Message` :

```
UPDATE message
WHERE trueCounterpart IS NOT NULL
SET trueCounterpart = (
        SELECT phoneNumber FROM contacts
        WHERE jid = trueCounterpart
);

UPDATE message
WHERE counterpart IS NOT NULL
SET counterpart = (
        SELECT phoneNumber FROM contacts
        WHERE jid = counterpart
);
```

Impact sur le code :

L'impact sur le code Java est relativement faible, car il suffit de remplacer le *jid* par le numéro de téléphone dans les classes qui manipulent les contacts. Il faudra aussi mettre à jour l'interface utilisateur pour permettre aux utilisateurs de saisir et de gérer les numéros de téléphone.

Il faudra aussi consevoir un script permettant à l'utilisateur de migrer ses contacts de *jid* à numéro de téléphone, et puis lancer les requêtes SQL pour effectuer la migration.

Scénario 4 : Préférence de compte

Résumé du changement :

Nous allons ajouter une fonctionnalité de préférence de compte à l'application. Cela permettra aux utilisateurs de définir des préférences pour chaque compte, telles que le thème, la langue, etc.

Détail des changements :

Strategie d'implémentation :

Pour ajouter cette fonctionnalité, nous allons ajouter une nouvelle table `accountPreferences` pour stocker les préférences de compte. Cette table sera en relation avec la table `Account`. Cela permettra à l'utilisateur de sauvegarder des configurations de préférences pour un même compte.

Modification :

Nous allons ajouter une table `preferences` pour stocker les préférences de compte. Cette table sera en relation avec la table `Account`.

```
CREATE TABLE accountPreferences (
        uuid TEXT PRIMARY KEY,
        accountUuid TEXT,
        theme TEXT,
        language TEXT,
        FOREIGN KEY(accountUuid)
        REFERENCES accounts(uuid) ON DELETE CASCADE
);
```

Impact sur le code :

La plupart des modifications nécessaires pour ajouter cette fonctionnalité seront dans le code Java. Il faudra ajouter des classes, des méthodes, des requêtes SQL, etc. pour gérer les préférences de compte. Il faudra aussi mettre à jour l'interface utilisateur pour permettre aux utilisateurs de définir et de gérer leurs préférences.

Il faudra certainement utiliser des outils d'internationalisation pour gérer les préférences de langue.

Scénario 5 : Possibilité de contacter une intelligence artificielle

Résumé du changement :

Nous allons ajouter une fonctionnalité permettant aux utilisateurs de contacter une intelligence artificielle. Cela permettra aux utilisateurs de poser des questions à l'intelligence artificielle et de recevoir des réponses.

Détail des changements :

Strategie d'implémentation :

Pour ajouter cette fonctionnalité, nous allons ajouter une nouvelle table `chatbot` pour stocker les informations sur les différentes intelligences artificielle. Cette table nous oblige de modifier la table `Contact` pour permettre de lier un contact à une intelligence artificielle.

Modification :

Nous allons ajouter une table `chatbot` pour stocker les informations sur les intelligences artificielles. Cette table sera en relation avec la table `Contact`.

Ce choix permet de simuler une conversation avec une intelligence artificielle en utilisant les mêmes fonctionnalités que pour une conversation normale. Cependant, il faudra modifier la table `Contact` pour diviser cette classe en deux classes, une pour les contacts en générale et une pour les personnes réelles.

Création de la table `chatbot` :

```
CREATE TABLE chatbot (
        uuid TEXT PRIMARY KEY,
        contactUuid TEXT,
        name TEXT,
        serviceURL TEXT,
        endpoint TEXT,
        description TEXT,
        FOREIGN KEY(contactUuid)
        REFERENCES contacts(uuid) ON DELETE CASCADE,
        UNIQUE (contactUuid) ON CONFLICT IGNORE
);
```

Création de la table `person` :

```
CREATE TABLE person (
        uuid TEXT PRIMARY KEY,
        contactUuid TEXT,
        servername TEXT,
        jid TEXT,
        systemaccount NUMERIC
        last_presence TEXT,
        last_time NUMERIC,

        FOREIGN KEY(contactUuid)
        REFERENCES contacts(uuid) ON DELETE CASCADE,
        UNIQUE (jid, contactUuid) ON CONFLICT IGNORE
);
```

Modification de la table `contact` :

```
ALTER TABLE contact RENAME TO contactTemp;
CREATE TABLE contact (
        uuid TEXT PRIMARY KEY,
        accountUuid TEXT
        systemname TEXT,
        presence_name TEXT,
        pgpkey TEXT,
        photouri TEXT,
        options NUMERIC,
        avatar TEXT,
        rtpCapability TEXT,
        groups TEXT,

        FOREIGN KEY(accountUuid)
        REFERENCES accounts(uuid) ON DELETE CASCADE
);
```

On ne supprime pas la table `contact` pour éviter de perdre les données. On la renomme en `contactTemp` pour le moment. On va effectuer la migration des données de la table `contactTemp` vers la nouvelle table `contact` et la table `person`. Mais ceci n'est pas détaillé ici.

Il faut aussi ajouter des triggers afin d'évité qu'une personne et qu'un chatbot soit lié à un même contact.

```
CREATE TRIGGER chatbot_trigger
BEFORE INSERT ON chatbot
BEGIN
        IF EXISTS (
                SELECT 1 FROM person
                WHERE contactUuid = NEW.contactUuid
        ) THEN
                RAISE(
                ABORT,
                'Contact already exists in person table'
                );
        END IF;
END;

CREATE TRIGGER person_trigger
BEFORE INSERT ON person
BEGIN
        IF EXISTS (
                SELECT 1 FROM chatbot
                WHERE contactUuid = NEW.contactUuid
        ) THEN
                RAISE(
                ABORT,
                'Contact already exists in chatbot table'
                );
        END IF;
END;
```

Il faut aussi garantir la contrainte de `UNIQUE` sur la colonne `jid` de la table `person` et la colonne `accountUuid` de la table `contact`. Pour cela, on utilise un trigger pour vérifier que la contrainte est respectée.

```
CREATE TRIGGER unique_jid_accountUuid_constraint
BEFORE INSERT ON person
FOR EACH ROW
BEGIN
        SELECT RAISE(
        ABORT,
        'Duplicate jid for the same accountUuid'
        )
        WHERE EXISTS (
```

```
        SELECT 1
        FROM person
        JOIN contacts ON person.contactUuid = contacts.uuid
        WHERE person.jid = NEW.jid
        AND contacts.accountUuid = (
                SELECT accountUuid
                FROM contacts
                WHERE uuid = NEW.contactUuid
        ));
END;
```

Impact sur le code :

Il faudra dans le code Java ajouter des classes, des méthodes, des requêtes SQL, etc. pour gérer les conversations avec les intelligences artificielles. Il faudra aussi mettre à jour l'interface utilisateur pour permettre aux utilisateurs de contacter les intelligences artificielles.

Pour cela, il faudra en premier tant modifier les classes en relation avec l'ORM. C'est-à-dire, *eu.siacs.conversations.persistance.DatabaseBackend* doit intégrer dans sa méthode `onCreate` la création des tables `chatbot` et `person`. Il faudra aussi modifier la méthode `onUpgrade` pour gérer les changements de version de la base de données.

Ensuite, la classe corespondant à la table `Contact` doit être modifié pour représenter cette nouvelle structure. Donc, dans *eu.siacs.conversations.entities.Contact*, il faudra retirer les attribue corresondant au colonnes retirées. Par exemple, servername, jid, etc. Il faudra aussi ajouter des méthodes permettant de gérer les nouveaux attributs. Par exemple, `getChatbot` et `getPerson`.

Pour finir, il faudra ajouter des classes permettant de gérer les conversations avec les intelligences artificielles. *eu.siacs.conversations.entities.Chatbot* et *eu.siacs.conversations.entities.Person* reprendront les attributs de la table `Chatbot` et `Person` respectivement, et seront impléménetées de la même manière que les autre classes du package `entities`.

Il est bien sur à noter que les activités responsables de la gestion des contacts devra être modifié pour gérer les nouveaux types de contacts. Par exemple *eu.siacs.conversations.ui.ContactDetailsActivity* doit prendre en compte que un contact peut être une personne avec un `JID`, mais peut être un bot qui lui ne possède pas de `JID`.

Scénario 6 : Suppression des données de sécurité

Résumé du changement :

Avec l'arrivée des ordinateurs quantiques, les données de sécurité actuelles ne sont plus considérées comme nécessaires. Nous allons supprimer les données et structures associées à la sécurité dans le système.

Détail des changements :

Tables à supprimer :

- `identities`, `signed_prekeys`, et `prekeys` : ces tables stockent des données liées aux clés cryptographiques.

Exemple de requête SQL :

```
DROP TABLE identities;
DROP TABLE signed_prekeys;
DROP TABLE prekeys;
```

Colonnes à supprimer :

- Dans la table `accounts`, les colonnes suivantes seront supprimées :
  - `pinned_mechanism`
  - `fast_mechanism`

Exemple de requête SQL :

```
ALTER TABLE accounts DROP COLUMN pinned_mechanism;
ALTER TABLE accounts DROP COLUMN fast_mechanism;
```

Impact sur le code :

- Les méthodes manipulant ces données doivent être retirées. Exemple :
  - Méthodes dans `UnifiedPushDatabase` et `DatabaseBackend`.
- Les API ou modules front-end utilisant ces données doivent être refactorisés pour fonctionner sans elles.

Test à effectuer :

- Vérifier que les données de sécurité sont bien supprimées sans erreur.

— Confirmer que les fonctionnalités principales restent opérationnelles.

—

Scénario 7 : Renommer certaines colonnes et tables

Résumé du changement :

Pour rendre le schéma plus clair, nous renommons certaines tables et colonnes pour qu'elles soient compréhensibles par de nouveaux collaborateurs.

Détail des changements :

Renommer la table `conversations` en `chats` : Cela implique de modifier toutes les références dans les requêtes SQL, ainsi que dans le code source.

Exemple de requête SQL :

```
ALTER TABLE conversations RENAME TO chats;
```

Renommer la colonne `body` dans `messages` en `content` : Cette colonne est utilisée pour stocker le contenu des messages. Le nouveau nom est plus explicite.

Exemple de requête SQL :

```
ALTER TABLE messages RENAME COLUMN body TO content;
```

Impact sur le code :

— Mettre à jour les méthodes d'accès à la base de données, telles que :
  — Les jointures impliquant `conversations`/`chats`.
  — Les sélections ou inserts utilisant `messages.body`.

Exemple de changement dans le code Java :

```
Avant :
db.query("conversations", null, "name=?", new String[] {chatName}, null, null,
```

```
Apres :
db.query("chats", null, "name=?", new String[] {chatName}, null, null, null);
```

Test à effectuer :

- Tester toutes les fonctionnalités qui utilisent `conversations` ou `messages.body`.
- Valider que les données migrées sont intactes.

—

Scénario 8 : Ajouter des catégories de messages

Résumé du changement :

Ajouter une colonne `category` dans la table `messages` pour permettre de classer les messages en catégories (ex. multimédia, GPS, etc.).

Détail des changements :

Ajouter une nouvelle colonne : Cette colonne permettra d'identifier le type de chaque message.

Exemple de requête SQL :

```
ALTER TABLE messages ADD COLUMN category TEXT;
```

Adapter les requêtes SQL : Lors de l'insertion d'un message, la catégorie doit être précisée :

```
ContentValues values = new ContentValues();
values.put("category", "multimedia");
db.insert("messages", null, values);
```

Interface utilisateur et API :

— Ajouter une logique dans l'interface utilisateur pour afficher les catégories et permettre le filtrage.

Test à effectuer :

— Ajouter différents types de messages et vérifier que la catégorie est correctement stockée.
— Valider que les messages sont affichés correctement par catégorie.

—

Scénario 9 : Envoi automatique de message lors de l'ajout d'un contact

Résumé du changement :

Lorsqu'un nouveau contact est ajouté, un message de bienvenue est automatiquement envoyé.

Détail des changements :

Ajouter un trigger SQL : Le trigger détecte l'insertion dans la table `contacts` et insère un message dans `messages`.

Exemple de requête SQL :

```
CREATE TRIGGER add_contact_message
AFTER INSERT ON contacts
BEGIN
    INSERT INTO messages (conversationUuid, body, type, timeSent)
    VALUES (NEW.uuid, 'Bienvenue !', 1, strftime('%s', 'now'));
END;
```

Impact sur le code :

— Les méthodes de gestion des contacts n'ont pas besoin de modification directe, mais il faut valider que le trigger fonctionne correctement.

Test à effectuer :

— Ajouter un contact et vérifier que le message de bienvenue est automatiquement créé.
— Vérifier que les anciens contacts ne déclenchent pas le trigger.

—

Scénario 10 : Répondre à un message

Résumé du changement :

Ajouter une fonctionnalité permettant de répondre à un message existant, en liant les messages sous forme de threads.

Détail des changements :

Ajouter une colonne `reply_to` : Cette colonne stocke l'identifiant du message auquel on répond.

Exemple de requête SQL :

```
ALTER TABLE messages ADD COLUMN reply_to TEXT;
```

Adapter les requêtes SQL : Lors de l'insertion, indiquer si un message est une réponse :

```
ContentValues values = new ContentValues();
values.put("reply_to", repliedMessageUuid);
db.insert("messages", null, values);
```

Pour récupérer les messages dans un thread :

```
SELECT * FROM messages WHERE reply_to = 'message_uuid';
```

Interface utilisateur et API :

- Modifier l'interface utilisateur pour afficher les messages sous forme de threads.
- Ajouter une option "Répondre" dans le menu contextuel des messages.

Test à effectuer :

- Répondre à différents messages et vérifier que les liens (`reply_to`) sont correctement créés.
- Valider l'affichage des threads dans l'interface utilisateur.

4 Étape 4 : Qualité de la base de données

Introduction

Après avoir analysé la base de donnée, nous pouvons donc maintenant établir une liste des bonnes réalisation et des mal façon. Pour chaque mal façon, nous allons proposer une solution pour corriger le problème.

L'application des correctifs permettra à la base de données d'évoluer de manière plus saine et plus robuste. Cela permettra aussi de faciliter l'ajout de nouvelles fonctionnalités et la contribution exterieur, tout en réduisant les coûts de maintenance.

Bonnes réalisations

La base de données présente plusieurs bonnes réalisations, notamment :

- **Centralisation de la gestion de la base de données** : Dans le code Java, la gestion de la base de données est centralisée principalement dans les classes `DatabaseBackend` et `UnifiedPushDatabase`. Cela permet de faciliter la modification de comportement de la base de données. Par exemple, si on veut changer le système de gestion de la base de données, il suffit de modifier ces classes.
- **Utilisation d'une nomenclature cohérente** : Les noms des tables et des colonnes sont explicites et suivent une nomenclature cohérente. Par exemple, les

tables sont nommées en anglais et font directement référence à leur semantique. Elle suit aussi le lexique utilisé à travers le protocole XMPP. Ceci permet de faciliter la compréhension de la base de données.
— **Versionnage** : La base de données présente un gestionnaire de version. Ce qui permet à la base de données de se mettre à jour d'une version antérieure à une version plus récente. Cela permet de garantir la cohérence des données et de faciliter l'évolution de la base de données.
— **Utilisation d'un UUID** : Dans la plupart des tables, un UUID est utilisé comme clé primaire. Cela permet d'éviter les problèmes de collision de clés et de garantir l'unicité des enregistrements. Cela permet aussi de faciliter la gestion des relations entre les tables.

Mal façons

La base de données présente également quelques mal façon, notamment :

Clé étrangère implicite

Comme on l'a déjà mentionné dans l'analyse des schémas, la base de données contient des clés étrangères implicite. Cela peut poser des problèmes d'intégrité référentielle et de cohérence des données. Il est donc recommandé d'ajouter des clés étrangères explicites pour garantir l'intégrité référentielle des données.

On pense à la relation entre les tables `Conversation` et `Contact`. Il est préférable d'ajouter une clé étrangère explicite pour garantir que chaque conversation est associée à un contact valide. Il y a d'autre cas où cela est nécessaire, voir la section 4.1.2.

Absence de contraintes `NOT NULL`

Comme déjà expliqué, certaines colonnes dans les tables de la base de données peuvent contenir des valeurs nulles. Cependant le code Java ne semble pas gérer ces valeurs nulles. Pour éviter des erreurs potentielles, on recommande l'utilisation de contraintes `NOT NULL`.

Celle-ci permetteront de garantir la bonne intégrité des données et d'éviter les erreurs potentielles lors d'une évolution du code.

Absence de documentation

Une des premières bonnes pratiques lorsqu'on devellope un produit logiciel est de documenter le code. Cela permet de faciliter la compréhension du code par les autres développeurs et de garantir la maintenabilité du code. Cependant, cette application ne contient que très peu de documentation. Il est donc difficile de comprendre le code et de le maintenir.

Notre recommandation serai, dans le meilleur des cas de tout redocumenter. Ceci étant dis, on est conscient que cela peut être une tâche très longue et fastidieuse. On recommande donc de documenter au fur et à mesure que l'on modifie le code. Cela permettra de garantir que le code reste documenté et maintenable.

Utilisation de différentes stratégies pour un même problème

Nous avons observé que les stratégies de stokage du `JID`[8] sont différentes entre la table `Contact` et la table `Account`. La table `Contact` opte pour un stokage en une seule colonne, celle-ci de type TEXT. Tandis que la table `Account` utilise trois colonnes, `username`, `server` et `ressource`, pour stocker le `JID`.

Ceci pose un problème de cohérence et de maintenabilité. Il est préférable d'utiliser une seule stratégie pour stocker le `JID`. On recommande l'utilisation d'une seule colonne de type TEXT pour stocker le `JID` dans les deux tables. Et, dans le meilleur des mondes, ajouter un trigger afin de garantir que le `JID` est bien formé.

8. Pour rappel, le `JID` prend cette syntaxe : `username@server/ressource`