Confroid

Android Configuration Manager

Group Report

Erasmus

Arnaud Vanspauwen
Dario Curreri
Domenico Di Fina
Emanuele Domingo
Salvatore Antonino Gristina

[GitHub Repository](#)

Summary

Introduction, goals and task

Our project consists of an application: Confroid, that is an app to manage configurations. More specifically Confroid has the goal to keep track of all the configurations of other applications. It allows us to navigate through versions of a specific configuration, it allows us to export or import configuration files from the device or from a server and it lets us edit configurations.

Making a list of which member of the group did what in the project is very complex as each partner took care of every part of the work, but in broad terms we can summarize everything as follows:

- Dario Curreri: Token authentication, API
- Domenico Di Fina: Storage, Import from the device, Export to the device
- Emanuele Domingo: Front-end Confroid, Shopping app
- Salvatore A. Gristina: Import from the Server, Export to the server
- Arnaud Vanspauwen: Geolocalization, Annotations, Testing, Fastlane directory

Teamwork was a fundamental part of our project to achieve the goals that have been set. Every week we held meetings where through the comparison we understood how to continue the project in the best way. We used github features, such as **Issues**, to make requests for help more immediate and efficient to solve bugs in specific parts of the project or pass the baton to a partner for a certain part of the code.

Configuration

A configuration is a data structure representing an application's preference data that must remain persistent.

As specified in the subject, we can represent the data format of a configuration in an EBNF form:

configuration = value

value = dictionary | array | primitive

dictionary = n * (key to value)

key = string

array = n * (value)

primitive = string | bytes | float | integer | boolean

Each configuration is identified by Confroid by a unique name prefixed by the name of the package that created it.

Token Authentication

In order to save a configuration, a client must authenticate with the confroid app.

This is necessary because our goal is to prevent other apps from modifying configurations they don't own.

So before asking Confroid for any service, a client always asks to assign it a token (this happens automatically when the client uses the API), calling the *BroadcastReceiver* *class* *TokenDispenser*.

```java
public static void askToken(Context context) {
    Intent intent = new Intent();
    intent.putExtra( name: "receiver", value: "fr.uge.confroidutils.services.TokenPuller");
    intent.putExtra( name: "name", context.getPackageName());
    intent.setClassName( packageName: "fr.uge.confroid", className: "fr.uge.confroid.receivers.TokenDispenser");
    context.sendBroadcast(intent);
}
```

API method asking for the token

The *TokenDispenser* class uses the *SHA-256* algorithm to generate pseudo-randomly the token based on the name of the application which asked for it.

```java
public static String getToken(String receiver) {
    MESSAGE_DIGEST.update(receiver.getBytes());
    return new String(MESSAGE_DIGEST.digest());
}
```

Token creation

Thus, when an **Intent** is sent to this class, the *OnReceive()* method is called. This method returns the pseudo-random token sending an **Intent** back.

```java
public void onReceive(Context context, Intent incomingIntent) {
    String receiver = incomingIntent.getStringExtra( name: "receiver");
    String name = incomingIntent.getStringExtra( name: "name");
    Intent outgoingIntent = new Intent();
    outgoingIntent.setClassName(name, receiver);
    outgoingIntent.putExtra( name: "token", getToken(name));
    context.startService(outgoingIntent);
}
```

Method called from the client that returns the token to the it

Then, the client should implement a method that awaits the intent and, wherever it arrives, process it. For instance, in the API this method is implemented in the **TokenPuller** class which extends the **Service** class.

```java
public class TokenPuller extends Service {

    private static String TOKEN;

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        TOKEN = intent.getStringExtra( name: "token");
        return START_NOT_STICKY;
    }
```

Reception of the Intent in the API

Send a Configuration

One of the most important features of our project is the possibility to save a configuration in the device. For achieving this we used two approaches: based on Services and based on API.

Save a Configuration: Services

The first way we used to save a configuration is based only on services. A third-party application could save a configuration on *Confroid* with a direct link to the system, using *Bundle* and Intent. For this reason we created a *ConfigurationPusher* service which has this goal. In particular, when we save a new configuration we call the service sending all the configuration's information, such as the name of the configuration, the corresponding token, the tag and the actual content.

First of all, the service checks if the token is correct, then we have two possibilities:
- Update an existing configuration: an user specify which element of the configuration wants to update and as a result we use the method *updateContent* in *ConfroidManager* to actually update the configuration.
- Create a new configuration: we create a new version of a configuration and we save it using the *saveConfiguration* method in *ConfroidManager*, which writes the content of the configuration in a JSON file.

Of course, every time we save a new configuration we notify the observers about the recent changes.

In this part of the project we encountered a problem about services. Every single service we started after 30 seconds ended. We figured out that the reason was in the new version of Android, where after some seconds that a service is in background it will stop the execution automatically. So we had to use the new method to start the service *startForeground*, that forces the app to execute the service in foreground. This method is not available for the old versions of Android, so we have to check which version has the user.

API

To save a configuration using the API, first of all the client needs to add the *ConfroidUtils Android Library*. This library exposes some methods, among which there is the *saveConfiguration* method, which requires a *Java Object* and not a *Bundle*.

```java
public void saveConfiguration(Context context, String name, Object object, String versionName) {
    String token = TokenPuller.getToken();
    if (token != null) {
        Bundle bundle = new Bundle();
        bundle.putString("name", context.getPackageName());
        if (!versionName.equals("")) {
            bundle.putString("tag", name + "/" + versionName);
        }
        bundle.putString("token", token);
        bundle.putBundle("content", FromObjectToBundleConverter.convert(object));
        Intent intent = new Intent();
        intent.setClassName( packageName: "fr.uge.confroid",  className: "fr.uge.confroid.services.ConfigurationPusher");
        intent.putExtra( name: "bundle", bundle);
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
            context.startForegroundService(intent);
        } else {
            context.startService(intent);
        }
    } else {
        Log.e( tag: "Token Error",  msg: "Token didn't arrive yet");
    }
}
```

saveConfiguration method defined in API

Before calling the *Confroid Service* to save, then it has to convert the *JavaObject* into a *Bundle*. To achieve this goal, it uses the class *FromObjectToBundleConverter*. Once the object is converted, it can prepare the sending intent and finally forward it to the *Confroid Service*.

Retrieve a Configuration

Retrieve a Configuration: Services

We created a service called *ConfigurationPuller* which receives a request from an application, it calls immediately the subscribe method that allows us to put the app in the observer list, that is used to notify all the changes in the configuration. After that, we call the *loadConfiguration* method in the *ConfroidManager* class, that returns the version of the configuration required using a *Bundle*.

Finally, we created an Intent to send all the data in the configuration to the requesting application.

We also had to create a new service, used to retrieve every version of a configuration, called *ConfigurationVersions*. Using the name of the configuration we read in the corresponding file JSON all the information and we parse recursively this file to find all the versions.

Finally, we send a new intent to the receiver with all the versions using a Bundle.

API

To retrieve specific configuration, the client needs to specify the version and call the *loadConfiguration* method defined in the API. Thus, this method sends an *Intent* to the *ConfigurationPuller* class defined in the Confroid app.

```java
public <T> void loadConfiguration(Context context, String version, Consumer<T> callback) {
    String token = TokenPuller.getToken();
    if (token != null) {
        Intent intent = new Intent();
        intent.putExtra( name: "name", context.getPackageName());
        intent.putExtra( name: "token", token);
        intent.putExtra( name: "requestId", value: requestId++ + "");
        intent.putExtra( name: "version", version);
        intent.putExtra( name: "receiver", value: "fr.uge.confroidutils.services.ConfigurationPuller");
        intent.setClassName( packageName: "fr.uge.confroid", className: "fr.uge.confroid.services.ConfigurationPuller");
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
            context.startForegroundService(intent);
        } else {
            context.startService(intent);
        }
        this.callbacks.add(callback);
    } else {
        Log.e( tag: "Token Error", msg: "Token didn't arrive yet");
    }
}
```

loadConfiguration method defined in API

Because this service will send an *Intent* back, the API awaits for it using its own implementation of the *ConfigurationPuller* class, which is a *Service*.

```java
public class ConfigurationPuller extends Service {

    private static ConfroidUtils CONFROID_UTILS;

    public static void setConfroidUtils(ConfroidUtils confroidUtils) {
        ConfigurationPuller.CONFROID_UTILS = confroidUtils;
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        CONFROID_UTILS.onReceiveConfigurationPuller(intent);
        return START_NOT_STICKY;
    }
}
```

ConfigurationPuller defined in API

Once this class receives the reply from *Confroid*, it calls a private method defined in the API to notify the reception passing the *Intent* received.

```java
public void onReceiveConfigurationPuller(Intent intent) {
    Bundle contentBundle = intent.getBundleExtra( name: "content");
    if (contentBundle != null) {
        this.callbacks.get(0).accept(FromBundleToObjectConverter.convert(contentBundle));
        this.callbacks.remove( index: 0);
    }
}
```

Private method dedicated to the ConfigurationPuller class

Then, once the API converts the *Bundle* in the original *Java Object*, it is ready to return the required version to the client using the *Callback*. defined by it.

Storage System

SQLite

In the project we had to store all the configurations in some kind of storage system. We could choose between a large number of solutions.

The first solution we adopted was a database SQLite, because saving data to a database is ideal structured data, such as in our case with the configurations. The APIs we needed to use a database on Android was in the android.database.sqlite package.

First of all it was necessary to create a formal declaration of how the database was organized, called the contract. In particular, our database was made up by one single table, which contains:

- name of the configuration;
- tag (optional): label assigned to a configuration;
- version;
- date: creation date;
- content: the configuration content.

```java
public final class ConfroidContract {

    private ConfroidContract(){}

    public static class ConfroidEntry implements BaseColumns {
        public static final String TABLE_NAME = "configurations";
        public static final String COLUMN_NAME_NAME = "name";
        public static final String COLUMN_NAME_CONTENT = "content";


    }
}
```

Example of contract with only two columns

Once we have defined how our database looks, we implemented methods that create and maintain the database and table, using the *SQLiteOpenHelper* class which contains a useful set of APIs for managing our database.

```
public class ConfroidDbHelper extends SQLiteOpenHelper {
    // If you change the database schema, you must increment the database version.
    public static final int DATABASE_VERSION = 1;
    public static final String DATABASE_NAME = "Confroid.db";

    private static final String SQL_CREATE_ENTRIES =
            "CREATE TABLE " + ConfroidContract.ConfroidEntry.TABLE_NAME + " (" +
                    ConfroidContract.ConfroidEntry._ID + " INTEGER PRIMARY KEY," +
                    ConfroidContract.ConfroidEntry.COLUMN_NAME_NAME + " TEXT," +
                    ConfroidContract.ConfroidEntry.COLUMN_NAME_CONTENT + " TEXT)";

    private static final String SQL_DELETE_ENTRIES =
            "DROP TABLE IF EXISTS " + ConfroidContract.ConfroidEntry.TABLE_NAME;
```

Example of DBHelper

Finally, we could add and delete configurations using the *insert* and *delete* methods, and of course, we could read them using the *query* method.

During the design of the database we realized that using a database requires to split the data to elementary tuples with relational links between them and according to the professor this is an hard way to proceed, for this reason we actually changed our mind and we decided to not use the database SQLite, but we preferred to store our configurations in a JSON file.

JSON File

In order to save a configuration in a JSON file, we used the org.json API, which allows us to manage in an easy way with JSON objects.

There is one JSON file for each configuration and in these files we stored all the versions of the configuration. These files are named after the name of the configuration (for example, if the configuration is named "fr.uge.test", the name of the file will be "fr_uge_test.json").

We also created a class named *FileUtils*, to have easy access to the methods used to write and read a file JSON.

We created methods to read also partially a file, keeping only one version or all the versions of a configuration, and methods to update only a piece of configuration (for example only the tag or just the content).

Backup & Restore: import and export of configurations

Import Configurations

Import from the Device

In order to import a configuration from a file JSON previously saved, we created in the main activity a button and a method linked to it called *openFile* which uses an intent with action **ACTION_OPEN_DOCUMENT**, that allow us to open a document, in this case a file json and as category **CATEGORY_OPENABLE**, that allow us to select and get a file wherever we want in our file system.

After we choose the configuration to import, we call the *ImportActivity* where, using the *importConfigurations* method we save all the configurations inside the file that we are going to import (with the *ConfroidManager's* method *saveConfiguration*).

Import from the Server

To access the services offered by the server, the user must be logged in, if not once the client tries to access these services, he will be sent back to a login form or can register on a register form.

Importing a configuration from the server does not happen through a real connection between an OkHTTP client and a MockWebServer but it's simulated. The real direct import from the server takes place during the export to the server phase with *server.getConfigurations()* method. In particular, during export to the server phase after uploading the file to the server, the actual import from the server takes place.
It was decided to do this to avoid problems of losing server configurations after closing the app.

```
try {
    server.start();
    File configurationsFile = new File(getFilesDir(), child: "web." + username + ".json");
    String configuration = FileUtils.readFile(configurationsFile);
    Intent intent = new Intent(getBaseContext(), ImportActivity.class);
    intent.putExtra( name: "CONFIGURATIONS", configuration);
    startActivityForResult(intent, OPEN_REQUEST_CODE_1);
```

Export Configurations

Export Configurations

Export to the Device

In order to export all the configurations in a external file, we created in the main activity a button and a method linked to it called *createAndSaveFile* which uses an intent with action **ACTION_CREATE_DOCUMENT**, that allow us to create a new document, in this case a file json and as category **CATEGORY_OPENABLE**, that allow us to select and save a file wherever we want in our file system.

After we choose a place to save our configurations we use the *getAllConfigurations* method in *ConfroidManager* to retrieve all the configurations, reading every file json concerning a configuration we have in our Confrod's storage and then save them in the new file created previously.

Export to the Server

As we can see from the code, there are threads. This implementation method was chosen to avoid the NetworkOnMainThreadException.
In order to export to the server, the client sends a post request containing the configuration file to the server while the latter is listening thanks to the *server.saveConfiguration()* method.

```java
case R.id.export_server:
    if(auth) {
        Client client = new Client();
        String username = getIntent().getStringExtra( name: "username");
        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    server.start();
                    server.saveConfiguration();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        });
        thread.start();
```

After server is started and is in listen mode thanks to *saveConfigurations* method, the client can send a post request to the server and the configuration will be saved automatically.as mentioned in the import from the server phase, the MockWebServer can not always be active, so we save the configuration sent by the client locally, taking it from the server. This local file will be used to simulate the import from the server phase

```
try {
    server.start();
    client.post(server.getUrl(), ConfroidManager.getAllConfigurations(getBaseContext()).toString());
    List<JSONObject> configurations = server.getConfigurations();
    String configuration = configurations.get(configurations.size()-1).toString();
    File configurationFile = new File(getFilesDir(), child: "web." + username + ".json");
    FileUtils.writeFile(configurationFile, configuration);
```

Confroid Use Interface

The main goal of the user interface development was to keep it simple and functional[1]. To do this,we tried to put, as much as possible, buttons in the menus to leave the screen empty for the data. We have several activities, summarized in the following schema:
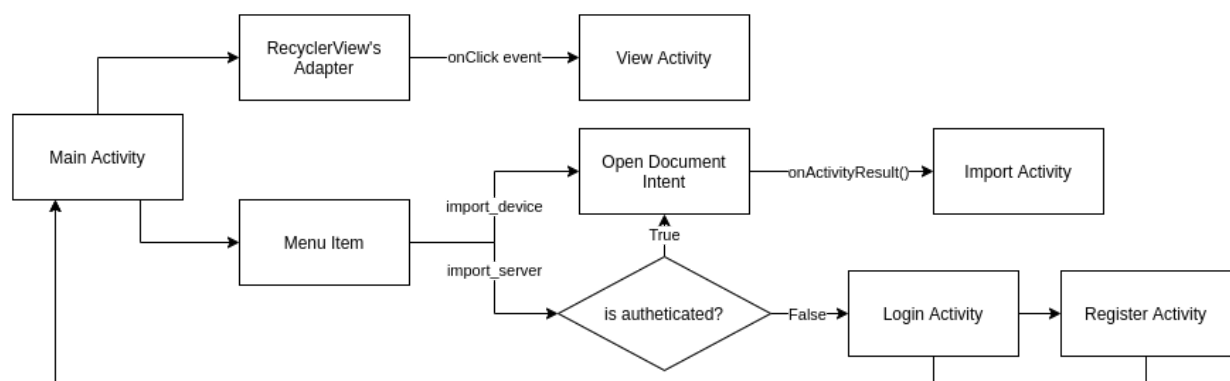
Here a brief introduction, we will see them in details in the next chapters:

- **MainActivity**, the default activity: it contains a list of all configurations stored and a menu. if we click on the name of a configuration we will be redirected to a ViewActivity. From the menu we can import or export.

- **ViewActivity**, here you can see the JSON file that represents your configuration. You can select which version, or edit the file. You can also see the creation date.

- **ImportActivity**, after the click of the import item in the menu, we will be redirected to an android activity to select a previously exported file. Once the file is loaded, from this activity we can select which configuration(s) to load.

  Please note: *a file exported contains all configurations in your confroid application.*

- **LoginActivity**, If a user is not logged in and tries to access the services offered by the server (export and import), he will be sent back to a login form where to insert his data

- **RegisterActivity**, From the login form, if your data are not in the database, you can go in the register form where you can insert your username and password

1: For a better description of the user interface see Emanuele's individual report.

Main Activity

The principal activity of the application, is made by two principal components:

- Menu
- RecyclerView

In the menù we can see the refresh button, once clicked it will load the data from the services and update the recyclerview's adapter (*refreshConfigurationsList()*). In the sub-menu there are all the import/export functions. Being a simple menu with buttons, we didn't have specific troubles.
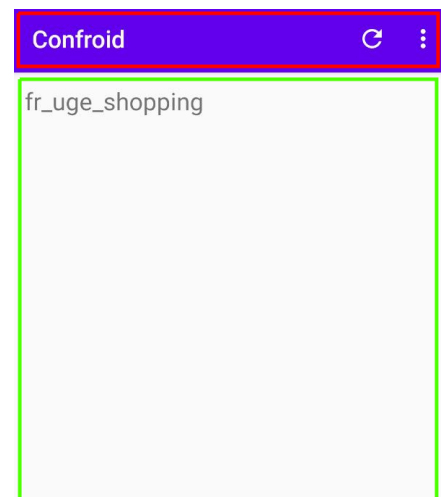
The second part of the activity is the recycler view. Here we store all the names of the saved configurations. The data is just an arraylist of strings, passed to the adapter. In addition, we give

to the adapter also the click listener that will launch ad **ImportActivity** with the name of the specific configuration via bundle.

Apart from graphical components, this activity also manages all the logic of the import and export features, described *here*. The main problems with its implementation were passing data between activities and the event management. The first one was rapidly resolved with bundles. For the latter we used the results of the activities: when we launch a specific activity we also put a code, when that activity ends (and we return in the main activity) we switch the result code and we perform different operations in the *onActivityResult(...)*. More info on the *official documentation*.

View Activity

As said before, this activity is rendered when the user clicks on the name of the configuration. First of all we load the configuration, given the name, from the service. Then the second step is to render the spinner (a.k.a combo box) that contains all the versions available for that configuration.
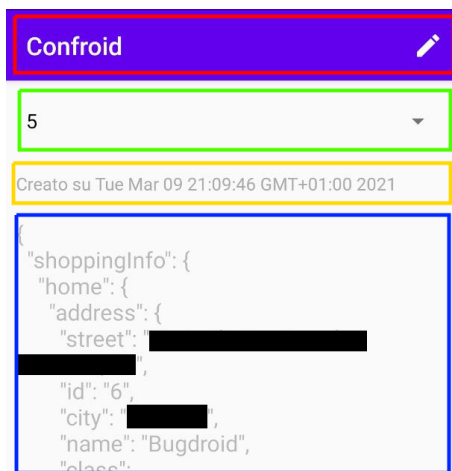
The spinner has an event, when we select a number we set the text of the next two elements: a simple *TextView* with the creation date and another *textView* with the json representation of the configuration.

This activity also has a menu on the top, where we can find the buttons for all the operations you can do:
- Edit the configuration
- Go back and cancel the changes
- Save the new configuration

The main difficulty with this implementation was to synchronize all the elements and update their contents depending on the operation that we are doing. To do this we created the methods to set the state and the content of all the items, like *setMenuItemEnabled(...)* and *updateContent()*.

Import Activity

After the user selects the backup file with all the configurations stored, this activity is launched and permits the user to select which specific configuration he wants to restore.

The logic schema is very similar to the main activity: we have a menu for the operations, import in this case, and a recycler view with the dynamic data. By default the import button is disabled, when at least one configuration is selected it becomes clickable.

Differently, this time the recycler view wraps the entire configuration in the items (and not just strings like before). To do this we created the **importItem** class that wraps a configuration, the name and a static image.
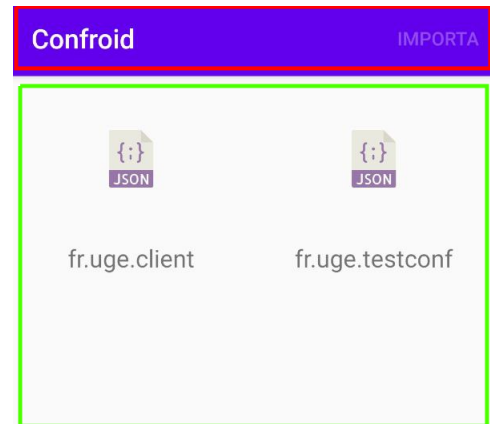
The main difficulty with this class was to find a way to edit the state of the menu from the adapter. The core idea is to access the menu of the activity by retrieving the context.
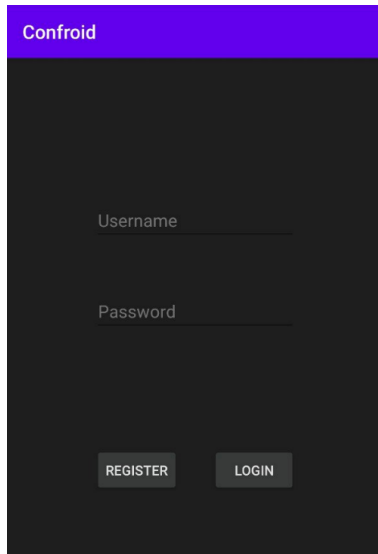
As we can see in the implementation:

```java
@Override
public void onBindViewHolder(@NonNull ViewHolder holder, int position) {
    ImportItem i = configurations.get(position);
    holder.update(i);
    holder.view.setBackgroundColor(i.isSelected() ? Color.CYAN : Color.TRANSPARENT);
    holder.view.setOnClickListener(v -> {
        ((ImportActivity)context).invalidateOptionsMenu();
        i.setSelected(!i.isSelected());
        holder.view.setBackgroundColor(i.isSelected() ? Color.argb( alpha: 127,  red: 52,  green: 152,  blue: 219)
                : Color.TRANSPARENT);
    });
}
```

with the *((ImportActivity)context)* we can access the activity and call its methods.

Login Activity

If a user is not logged in and tries to access the services offered by the server, he will be sent back to a login form managed by LoginActivity

Once the user has entered their data, a check on the *database.json* is carried out to see if the user is registered or not.

If the user's data are not present in the database, an error message appears and the server services cannot be used.

In order to *register* your credentials in the database, just click on the register button which will take the user to the registration form managed by **RegisterActivity**.

The control of the user's data takes place as soon as the login button is clicked.
Thanks to the button listener, this control is launched which checks if username and password are in the database

```java
if(!usernameText.equals("") && !passwordText.equals("")) {
    try {
        File database = new File(this.getFilesDir(), child: "web.database.json");
        JSONObject databaseObj = new JSONObject(FileUtils.readFile(database));
        JSONArray users = databaseObj.getJSONArray( name: "users");
        for (int i = 0 ; i < users.length(); i++) {
            JSONObject user = users.getJSONObject(i);
            if(user.getString( name: "username").equals(usernameText) && user.getString( name: "password").equals(passwordText)){
                auth = true;
            }
        }
    }
```

Register Activity

As shown in the previous figure, to access the registration form, just click the register button in the LoginActivity.

The task of this activity is to add users to *database.json* by entering their username and associated password.

Code that allows you to enter the credentials of the new user in the **users** JSONArray

```java
if(!usernameText.equals("") && !passwordText.equals("")){
    File database = new File(getFilesDir(), child: "web.database.json");
    String accountsJson = FileUtils.readFile(database);
    try {
        JSONObject accounts = new JSONObject(accountsJson);
        JSONArray users = accounts.getJSONArray( name: "users");
        JSONObject newUser = new JSONObject();
        newUser.put( name: "username", usernameText);
        newUser.put( name: "password", passwordText);
        users.put(newUser);
        File databaseFinal = new File(this.getFilesDir(), child: "web.database.json");
        FileUtils.writeFile(databaseFinal, accounts.toString());
        Intent intent = new Intent( packageContext: this, LoginActivity.class);
        startActivity(intent);
```

The biggest problem for the login and registration service was deciding on a way to use a database that was quick to implement and easy to use. Eventually it was decided to use a json file.

Client applications

In order to perform test and try all the functionality of the Confroid application, we also implemented two client applications:

- **Client**, a simple application that tests the services calls.
- **Shopping**, a more complex application that communicates with confroid through api calls.

Client

This was the first client the team made. It's a very simple GUI with just a button to test the service calls. As said before, the Confroid application uses the **services** to communicate with other applications.

A third-part application, in order to use that services, must declare them in its manifest:

```xml
<uses-permission android:name="android.permission.FOREGROUND_SERVICE"/>

<application>
    <activity android:name=".gui.ConfigurationVersionsActivity"...>
    <activity android:name=".MainActivity"...>

    <service android:name=".services.TokenPuller"  android:exported="true"/>
    <service android:name=".services.ConfigurationPuller" android:exported="true"/>
    <service android:name=".services.ConfigurationVersions" android:exported="true"/>
</application>
```

and then implements the class which will use the specified service.

Shopping

This client application is the most complex GUI in the project and in general that we have ever made. We use a lot of elements and objects and also api class to fetch data from confroid. We encountered several problems doing this, mostly related to the asynchronous nature of the api calls and the management of events in the gui. To manage this, we used an *"first create, then populate"* approach: we create empty elements (like dialog boxes, text views, edit texts) and then when the api returns the data we update the content of these elements.

When the application is launched, it's empty. We have two buttons:

- **Load**: this button opens a dialog box that asks which version you want to fetch from the Confroid App, then we load the selected configuration and we activate the button to add a new entry.
- **Add** (disabled by default): This button also opens a dialog box which asks for the name of the new configuration entry that we want to add.
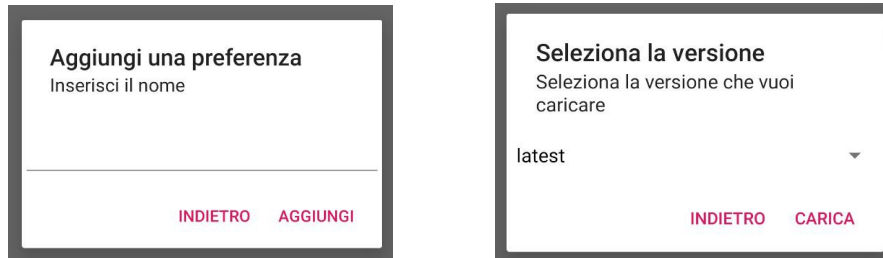
Dialog boxes examples from the shopping app

The load dialog is particularly interesting due to its dynamicity: if at least one version is present in the Confroid application we can load it. But if no configuration is present, the same dialog box turns into a "creation" dialog that initializes the first configuration.
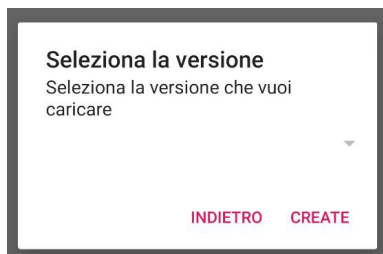

To do this, first we create the "load" dialog. Then, if no versions are found in the Confroid App (this happens only the first start up) we convert the created dialog in a new dialog by redefining the items and the event on the button. Everything is done in the ***makeDialogForCreation()*** method.

Once we load everything, we can select one entry to visualize and edit it into the **EditActivity**.
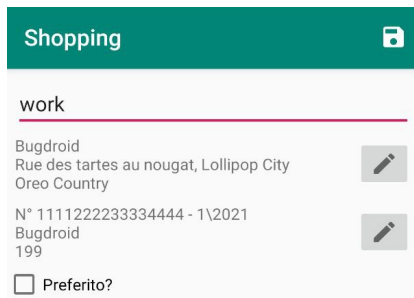
In this activity we can see the name, the address and the payment information about the selected entry. The most difficult part of this activity was to learn how ***fragments*** work. In fact, when we push the edit buttons, to modify the address or the payment card a new fragment in the middle of the activity is created with a form to edit all the fields.
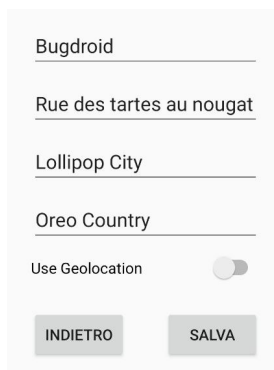
As you can see, the year field is red and the *save* button is disabled. This is because the **EditBillingFragment** has text validation. In the side example, the year is not between 2020 and 2040 as defined in the ***annotations***:
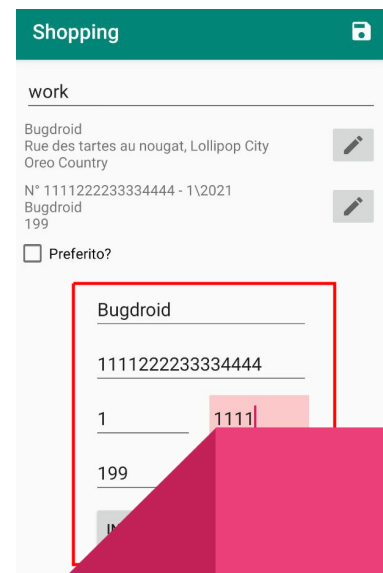
```
@Description (descriptionValue = "Expiration Year")
@RangeValidator (lowerBound = 2020, upperBound = 2040)
public int expirationYear;
```

To make this possible we used the **TextValidator** class that concretises an interface that can be used as *TextChangeListener* for the textviews. Then we use an array to track all the possible errors: if the error array is empty the save button is enabled.

The fragment for the edition of the address is pretty similar, we don't have the text validation but the have the *geolocalization* support:

Bugdroid

Rue des tartes au nougat

Lollipop City

Oreo Country

Use Geolocation

INDIETRO   SALVA

By turning on the switcher, the application uses the geolocalization services to get the coordinates, then with the *parseCoordinates(...)* method we use *Google's api* to convert latitude and longitude to a text address.

Annotations

Annotations were added to the Java language in Java 1.5 and can easily add a whole new dynamic to regular programming. Simply put, annotations provide a way to add metadata to various aspects of a class, additional information that can be used to modify the operation of the class behaves or simply further describe it in ways that are not easily achievable through standard means using Annotations

Annotations are extremely easy to use , and can be applied to multiple areas of your code .

Some examples of single annotation in our project fr.uge.Shopping:

```java
@Override
public boolean equals(Object o){
...
}
```

@Override: Checks that the method is an override. Causes a compilation error if the method is not found in one of the parent classes or implemented interfaces.

```java
@Nullable
@Override
public View onCreateView(@NonNull LayoutInflater inflater, @Nullable ViewGroup
container, @Nullable Bundle savedInstanceState) {
...
}
```

@Nullable: It makes it clear that the method accepts null values, and that if you override the method, you should also accept null values.

@NonNull: Denotes that a parameter, field or method return value can never be null.

```
@Description (descriptionValue = R.string.cardNumber)
```

@Description: The Description annotation allows a description to be specified for a method.

```
@RegexValidator (regex = "[0-9] {16}")
```

@RegexValidator: RegexValidator that checks against a Regular Expression (which is the pattern property). The pattern must resolve to a String that follows the java.util.regex standards.

```
@ClassValidator(clazz = CreditCardChecker.class)
```

@ClassValidator: A validator object is not thread-safe and not reentrant. In other words, it is the application's responsibility to make sure that one Validator object is not used from more than one thread at any given time, and while the validate method is invoked, applications may not recursively call the validate method.

```
@RangeValidator(lowerBound = 1, upperBound = 12)
```

@RangeValidator: A validator that checks with the given bounds if it's in the range

```
@RequiresApi(api = Build.VERSION_CODES.KITKAT)
```

@RequiresApi: Denotes that the annotated element should only be called on the given API level or higher.

Geolocalization

The solution we used in the project for turning on the GPS on the device is by using android gms location request and setting clients. We use a custom class GpsUtils.java with code that provides a method TurnGPSOn().

The method includes a callback listener for checking the GPS's current state. If GPS is already allowed, no more code will be created, and the callback will be set to true.

The SettingsClient class is the primary interface for communicating with location-based APIs. This API allows an app to easily verify that the device's system settings are correctly optimized for the app's location requirements.

```java
public class SettingsClient extends com.google.android.gms.common.api.GoogleApi<com.google.android.gms.common.api.Api.ApiOptions.NoOptions> {
    public SettingsClient(@androidx.annotation.RecentlyNonNull android.content.Context context) { /* compiled code */ }

    public SettingsClient(@androidx.annotation.RecentlyNonNull android.app.Activity activity) { /* compiled code */ }

    @androidx.annotation.RecentlyNonNull
    public com.google.android.gms.tasks.Task<com.google.android.gms.location.LocationSettingsResponse> checkLocationSettings(@androidx.annotation.Rec
}
```

LocationSettingsRequest specifies the types of location resources that the customer wishes to use. The settings will be reviewed to ensure that all requested facilities perform properly. We used LocationSettingsRequest.Builder.

```java
LocationSettingsRequest.Builder builder = new LocationSettingsRequest.Builder()
        .addLocationRequest(locationRequest);

locationSettingsRequest = builder.build();
builder.setAlwaysShow(true);
```

The GPS Dialog will be shown for the SettingsClient's malfunction callback, which will result in an activity's onActivityResult(). So, if the user decides to allow GPS, we will allow the GPS flag in our operation.

```java
@Override
public void onActivityResult(int requestCode, int resultCode, @Nullable Intent
data) {
    super.onActivityResult(requestCode, resultCode, data);
    if (resultCode == Activity.RESULT_OK) {
        if (requestCode == GpsUtils.GPS_REQUEST_CODE) {
            this.gpsAvalaible = true; // flag maintain before get location
        }
    }
}
```

After this we just put the code below in OnViewCreated:

```java
this.gpsUtils.turnGPSOn(isGPSEnable -> this.gpsAvalaible = isGPSEnable);
```

As a result, turning on GPS and obtaining a position works well one by one, but after all the GPS takes some time to pinpoint the device's exact location.