Android SDK : How to integrate it within your app

Last modification : 7th sept 2018

By : Guillaume Agis

Revision : phase 2

Introduction

The android SDK is separated in several librairies:

- One containing the Profile and Brushing modules, called **WebServiceSDK (**phase1)
- One containing the **Stats** module **(**phase1)
- One containing the **Jaws3D module,** use to render the mouth zone **(**phase1)
- One containing the **Offline brushing module** (phase2)
- One containing the **SMS auth** (login and account creation using phone number verification, phase1)
- One containing the **Pairing module** (phase 1)

Each module has its own detailed documentation and explanation and contain a public interface to communicate with our modules.

The purpose of this documentation is for you to understand the general concept of the SDK and how to integrate it into your app.

Technologies used within the SDK

- **RxAndroid**
- Android Room
- Retrofit
- **Dagger 2**
- MVVM

Dependencies of the SDK in your app

Your app will need to contain those dependencies in order to use the SDK.

- RxAndroid : to make the app reactive. If you are not familiar with rxAndroid, please refer to some documentations such as this one.
- Dagger 2 : to inject dependencies into the SDK. If you are not familiar with Dagger 2, please refer to some documentations such as this one.

How to integrate the SDK into your app

1. Type mkdir -p ~/.m2/repository/com/ && open ~/.m2/repository/com/ This will open a Finder window

2. Extract Kolibree.zip into that folder

3. Update SDKDemo/build.gradle

Add those line inside your project build.gradle. It is for downloading the modules included in the SDK from our server.

```
allprojects {
    repositories {
        google()
        jcenter()
  mavenLocal()
    }
}
```

4. Update the build.gradle of your app and gradle.properties

Add the added modules as a dependencies and add the following dependencies to your project.

As mentioned earlier, You will need them to work with our modules.

The versions used in that example are just an example, you are not required to use those versions.

Add this library inside your app build.gradle to include **all** the modules of the SDK into your project.

```
dependencies {
    def modulesVersion = "1.0.1"
    implementation group: 'com.kolibree.android', name: 'core', version:
"$modulesVersion"

 ...  //thats the only lib needed
}
```

Sync your project, and now you are ready to include the last missing piece of code to use the SDK.

3. Inject the client_id and client_secret into the SDK

Those dependencies are required to be provided from your side to make the SDK work correctly.

The following dependencies need to be provided :

- Your Client Id, Integer format
- Your client Secret, Integer format

You can find an example in the demoApp in the CredentialsModule.java file.

In order to inject the dependencies you will need to use **DAGGER 2** !

I'll recommend you to read about Dagger and understand correctly how it works before using the code sample I provide below.

Dagger errors are very tricky and needs a good understanding of Dagger in order to be fixed, for most of them.

With all the modules of the core SDK inside your app, your component file should look like this :

While building your Component, it's important to do not forget to provide the Context and the SdkComponent.

```
import android.content.Context;
import com.kolibree.android.jaws.JawsView;
import com.kolibree.android.sdk.dagger.SdkComponent;
import com.kolibree.core.dagger.CoreModule;
import dagger.BindsInstance;
import dagger.Component;
import dagger.android.AndroidInjectionModule;
import dagger.android.AndroidInjector;
import javax.inject.Singleton;
import kolibree.com.demoapp.MainApp;
import kolibree.com.demoapp.demo.di.DemoSDKModule;

@Singleton
@Component(
    dependencies = {SdkComponent.class},
    modules = {
  AndroidSupportInjectionModule.class, // required to be able to inject
  DemoApplicationModule.class, // include the activities that will use the
@Inject annotation
  DemoSDKModule.class, // dependency for this demo app ONLY
  CoreModule.class, // include the android SDK with all the modules
  CredentialsModule.class // provide the client_id and client_secret to the
SDK
    })
public interface DemoAppComponent extends AndroidInjector<MainApp> {

  void inject(JawsView mouthView);

  @Component.Builder
  interface Builder {

    @BindsInstance
    Builder context(Context context);

    Builder sdkComponent(SdkComponent sdkComponent);

    DemoAppComponent build();
  }
}
```

Content of the DemoApplicationModule.java

Only contains the Activities you are going to inject the dependencies with the @Inject annotation. Here is en example.

```
@Module
public abstract class DemoApplicationModule {

  @ContributesAndroidInjector
  abstract MainActivity contributeMainActivityInjector();


... other activity
}
```

Regarding the clientId and clientSecret in staging and production, you will just need to provide them in the string.xml file.

```
<string name="new_client_id_production">add_your_key</string>
<string name="new_client_secret_production">add_your_key</string><string
name="new_client_id">"TnphQzF5YzJFQUFBQUJJd0FBQVFFQXEyQTdoUkdtZG5tOXRVRGJP
OUlEU3dCSzZUYlFhK1BYWVBDUHk2cmJUclR0dzdQSGtjY0tycHAweVZocDVIZEVJY0tyNnBMbF
ZEQmZPTFg5UVVzeUNPVjB3emZqSUpObEdFWXNkbExkaXpIaGJuMm1VanZTQUhRcVpFVFlQODFl
RnpMUU5uUEh0NEVWVlVoN1ZmREVTVTg0S2V6bUQ1UWxXcFhMbXZVMzEveU1mK1NlOHhoSFR2S1
NDWklGSW1Xd29HNm1iVW9XZjluenBJb2FTakIrd2VxcVVVbXBhYWFzWFZhbDcySitV"</strin
g>
<string
name="new_client_secret">"QNnPHt4EVVUh7VfDESU84KezmD5QlWpXLmvU31/yMf+Se8xh
HTvKSCZIFImWwoG6mbUoWf9nzpIoaSjB+weqqUUmpaaasXVal72J+UX2B+2RPW3RcT0eOzQgql
JL3RKrTJvdsjE3JEA"</string>ur production keys are not generated yet, please
ask them.
```

4. Update the androidManifest.xml

Some permissions are needed in order to be able to use the pairing module and discover nearby toothbrushes.

Do not forget to ask the grant the permission during the runtime.

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
```

5. Update of your Application class

Your application class needs to implement the HasActivityInjector, HasServiceInjector and you will need to provide the Context and a SdkComponent.

To do so, here is an example you can re-use in your own app.

All of theses are needed to be able to inject dependencies into the activities and our services we use within the SDK.

```
import android.app.Activity;
import android.app.Application;
import android.app.Service;
import com.jakewharton.threetenabp.AndroidThreeTen;
import com.kolibree.android.sdk.KolibreeAndroidSDK;
import com.kolibree.android.sdk.dagger.SdkComponent;
import dagger.android.AndroidInjector;
import dagger.android.DispatchingAndroidInjector;
import dagger.android.HasActivityInjector;
import dagger.android.HasServiceInjector;
import javax.inject.Inject;
import kolibree.com.demoapp.di.DaggerMyComponent;
import kolibree.com.demoapp.di.MyComponent;

public class MainApp extends Application implements HasActivityInjector,
HasServiceInjector {

  public static MyComponent appComponent;

  @Inject
  DispatchingAndroidInjector<Activity> dispatchingAndroidInjector;

  @Inject
  DispatchingAndroidInjector<Service> dispatchingServiceAndroidInjector;

  @Override
  public void onCreate() {
    super.onCreate();
    AndroidThreeTen.init(this);
    initDagger();
  }

  private void initDagger() {
    SdkComponent sdkComponent = KolibreeAndroidSDK.init(this); // required,
otherwise the pairing module will not work

    appComponent = DaggerMyComponent.builder()
        .context(this)
        .sdkComponent(sdkComponent)
        .build();

    appComponent.inject(this);
  }

  @Override
  public DispatchingAndroidInjector<Activity> activityInjector() {
    return dispatchingAndroidInjector;
  }

  @Override
  public AndroidInjector<Service> serviceInjector() {
```

```
    return dispatchingServiceAndroidInjector;
  }
}
```

Once you have all of these, you are setup ! All the dependencies should be provided correctly !

You are ready to use the SDK !

Authentication module

Last modification : 6th sept 2018

By : Guillaume Agis

Revision : phase 2

What is it?

This module is responsible for account creation and login using SMS.

The authentication module is part of the core module.

Authentication by SMS

WITH UI

Authentication module provides UI (Activity with nested Fragments) for authentication user by SMS
Authentication flow contains 2 cases:

- account creation flow - creates brand new user, it should be used at the very beginning. *AccountData* object provides all needed information about user (name, gender, country etc)

```
Intent accountCreation =
AuthenticationFlowActivity.createAccountFlow(Context ctx, AccountData
data);
```

- login flow - login to already existing account.

```
Intent login = AuthenticationFlowActivity.loginToAccountFlow(Context ctx,
AccountData data);
```

in both cases, Intent should be started by *Activity.startActivityForResult*, like:

```
final int AUTH_REQUEST_CODE = 123;
Intent accountCreation =
AuthenticationFlowActivity.createAccountFlow(getContext(),
getAccountData());
startActivityForResult(accountCreation, AUTH_REQUEST_CODE);
```

after *AuthenticationFlowActivity* is finished, you can access returned result and data inside *onActivityResult* method:

```
@Override
void onActivityResult(int requestCode, int resultCode, Intent data) {


  switch (requestCode) {
    ...

    case AUTH_REQUEST_CODE:
      if (AuthenticationFlowActivity.wasSuccess(resultCode)) {
        AuthenticationResultData resultData =
AuthenticationFlowActivity.extractResultData(data);
        // TODO:
      }
      break;
  }
}
```

WITHOUT UI

SmsAccountManager provides all needed method for authentication user by SMS.
It provide 3 simple methods:

```
/**
* This method informs backend to send SMS with the code to given phone
number
* @param phoneNumber the number where SMS with the code will be send
* @return [Single] with object [SmsToken]
*/
fun sendSmsCodeTo(phoneNumber: String): Single<SmsToken>


/**
* This method will create new account with [AccountData] data
* @param smsToken, object returned by method [sendSmsCodeTo]
* @param smsCode code which has been sent to your phone by SMS
* @param data detailed data about account
* @return [Single] with list of profiles [IProfile]
*/
fun createAccount(smsToken: SmsToken, smsCode: String, data: AccountData):
Single<List<IProfile>>


/**
* Login to already existing account
* @param smsToken, object returned by method [sendSmsCodeTo]
* @param smsCode code which has been sent to your phone by SMS
* @return [Single] with list of profiles [IProfile]
*/
fun loginToAccount(smsToken: SmsToken, code: String):
Single<List<IProfile>>
```

- for account creation, please use *sendSmsCodeTo* and *createAccount* methods, like:

```
// SmsAccountManager instance will be injected by Dagger
@Inject
SmsAccountManager smsAccountManager;

// local variable for token
private SmsToken token;

/**
* This method informs backend, that has to send SMS with the code to given
phone number
**/
public void userProvidedNumber(String phoneNumber) {
 smsAccountManager.sendSmsCodeTo(phoneNumber) // execute sendSmsCodeTo
method
  .subscribeOn(Schedulers.io()) // on background thread
  .observeOn(AndroidSchedulers.mainThread()) // but result will be executed
on UI thread
  .subscribe(
   smsToken -> token = smsToken, // if success, assign result to local
variable
   Throwable::printStackTrace // if failure, print stack trace
  );
}

/**
* After receiving SMS with the code, this method will create a new account
* (information about phone number are included inside SmsToken)
**/
public void userProvidedCodeFromSms(String code) {
 AccountData data = createAccountData(); // get/create AccountData
 smsAccountManager.createAccount(token, code, data) // execute
createAccount method
  .subscribeOn(Schedulers.io()) // on background thread
  .observeOn(AndroidSchedulers.mainThread()) // but result will be executed
on UI thread
  .subscribe(
   this::newUserCreated, // if success, invoke newUserCreated method
   Throwable::printStackTrace); // if failure, print stack trace
}

void newUserCreated(List<IProfile> profiles) {
//TODO
}
```

- for login to already existing account, please use *sendSmsCodeTo* and *loginToAccount* methods, like:

```
// SmsAccountManager instance will be injected by Dagger
@Inject
SmsAccountManager smsAccountManager;

// local variable for token
private SmsToken token;

/**
* This method informs backend, that has to send SMS with the code to given
phone number
**/
public void userProvidedNumber(String phoneNumber) {
 smsAccountManager.sendSmsCodeTo(phoneNumber) // execute sendSmsCodeTo
method
  .subscribeOn(Schedulers.io()) // on background thread
  .observeOn(AndroidSchedulers.mainThread()) // but result will be executed
on UI thread
  .subscribe(
   smsToken -> token = smsToken, // if success, assign result to local
variable
   Throwable::printStackTrace); // if failure, print stack trace
}

/**
* After receiving SMS with the code, this method will login to appropriate
account
* (information about phone number are included inside SmsToken)
**/
public void userProvidedCodeFromSms(String code) {
 smsAccountManager.loginToAccount(token, code) // execute loginToAccount
method
  .subscribeOn(Schedulers.io()) // on background thread
  .observeOn(AndroidSchedulers.mainThread()) // but result will be executed
on UI thread
  .subscribe(
   this::loginToAccount, // if success, invoke loginToAccount method
   Throwable::printStackTrace); // if failure, print stack trace
}

void loginToAccount(List<IProfile> existingProfiles) {
//TODO
}
```

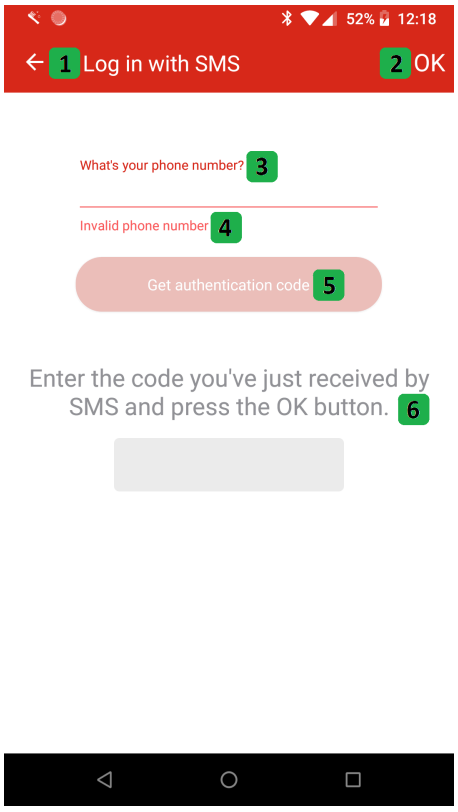
Custom translations

If you want to provide custom translations for *AuthenticationFlowActivity* then you have to use *TranslationProvider* object.
All needed keys can be found in class *AuthTranslationKey* (or *AuthTranslationKeyKt* for Java applications)
These keys have to be added together with custom translation to *Map<String, String>* object.
And map object has to be added by *TranslationProvider.addLanguageSupport* method.

Here is the example code:

```
// create TranslationsProvider object
TranslationsProvider provider = new TranslationsProvider();
// create map object
Map<String, String> authTranslations = new HashMap<>();
// add custom translations to map
authTranslations.put(AuthTranslationKeyKt.TOOLBAR_TITLE, "Custom title");
authTranslations.put(AuthTranslationKeyKt.TOOLBAR_SUBMIT, "Custom submit");
authTranslations.put(AuthTranslationKeyKt.CODE_DESCRIPTION, "Custom code
description");
authTranslations.put(AuthTranslationKeyKt.PHONE_NUMBER_HINT, "Custom phone
number hint");
authTranslations.put(AuthTranslationKeyKt.PHONE_NUMBER_INVALID, "Custom
phone number invalid");
authTranslations.put(AuthTranslationKeyKt.PHONE_NUMBER_SUBMIT, "Custom
phone number submit");
// add custom translations (in this case for English language)
provider.addLanguageSupport(Locale.ENGLISH, authTranslations);


// apply translations changes
Translations.init(context, translationsProvider);
```

All keys with related UI views can be found here:



where:

```
1 -> TOOLBAR_TITLE
2 -> TOOLBAR_SUBMIT
3 -> PHONE_NUMBER_HINT
4 -> PHONE_NUMBER_INVALID
5 -> PHONE_NUMBER_SUBMIT
6 -> CODE_DESCRIPTION
```

Default English translations for these keys:

```
TOOLBAR_TITLE => "Log in with SMS"
TOOLBAR_SUBMIT => "OK"
PHONE_NUMBER_HINT => "What's your phone number?"
PHONE_NUMBER_SUBMIT => "Get authentication code"
PHONE_NUMBER_INVALID => "Invalid phone number"
CODE_DESCRIPTION => "Enter the code you've just received by SMS and press
the OK button."
```

Default Chinese translations for these keys:

```
TOOLBAR_TITLE => ""
TOOLBAR_SUBMIT => ""
PHONE_NUMBER_HINT => ""
PHONE_NUMBER_SUBMIT => ""
PHONE_NUMBER_INVALID => ""
CODE_DESCRIPTION => "OK"
```

Brushing module

Last modification : 10th sept 2018

By : Guillaume Agis

Revision : phase 2

What is it?

This module is responsible for managing the brushings for a profile.

It's really important to call the **synchronised** method every time you load a profile as it's going to sync the local and remove brushings

and store locally the latest version of the brushings, used everywhere else in the app later on. (Stat included)

By Injecting in your code, using the @Inject annotation, the interface BrushingFacade you get access to all the methods of this module.

All the methods need and will provide you a IBrushing interface that contains the required attributes for this modules.

To use these methods, your custom brushing object is required to **implement** the IBrushing interface, otherwise you won't be able to use it.

The brushing module is part of the web service module.

Dependencies in your app

- RxAndroid: if you are not familiar with Single and the other RX type, please read some documentations such as this one.

Tools

BrushingFacade interface

```
/**
 * Interface to communicate with the brushing module.
 * Your custom brushing object needs to implement the IBrushing interface.
 * At my sense, the most important method in this module is the
synchronizeBrushing
 * that needs to be called every time you load a profile in order to sync
the local and
 * remote brushings.
 */
@Keep
interface BrushingFacade {
    /**
     * Add a new brushing to a profile
     *
     * @param brushing brushing to add
```

```
     * @param profile profile to associated the brushing to
     * @return non null [IBrushing] [Single] brushing added
     */
    fun addBrushing(brushing: IBrushing, profile: IProfile):
Single<IBrushing>

    /**
     * Get all the  brushings stored locally for a given profile,
associated by its profile
     * Id. Do not forget to call the sync method of this module when you
load a profile in order to
     * sync the local and remote brushings.
     *
     * @param profileId profileId of the profile to get the brushings from
     * @return non null [IBrushing] [List] [Single] list of brushings
stored locally associated to the profile
     */
    fun getBrushings(profileId: Long): Single<List<IBrushing>>

    /**
     * Get all the brushings for a given profile since a given date
     *
     * @param startTime date to get the brushings from
     * @param profileId profileId of the profile to get the brushings from
     * @return non null [IBrushing] [List] [Single] list of brushings
stored for that user since this date
     */
    fun getBrushingsSince(startTime: ZonedDateTime, profileId: Long):
Single<List<IBrushing>>

    /**
     * Get the latest brushing sessions for a profile
     * Throw a [NoExistingBrushingException] exception if there is no
existing brushing for this user
     * @param profileId profileId of the profile to get the latest brushing
from
     * @return non null [IBrushing] [Single] last brushing if exist
     */
    fun getLastBrushingSession(profileId: Long): Single<IBrushing>

    /**
     * delete a  brushing for a profile
     *
     * @param brushing brushing to delete
     * @param profileId profileId of the profile to get the latest brushing
from
     * @return non null [Completable]
     */
    fun deleteBrushing(profileId: Long, brushing: IBrushing): Completable


    /**
     * Synchronized the local brushing with the remote brushing on the
```

```
server to make sure the local
     * and remote database contains the latest updated version.
     * If there are any brushings, It always stores it locally
     * It's important to call this method every time your load a profile.
     *
     * @param profileId profileId of the profile to get the latest brushing
from
     * @return non null [Boolean] [Single] true is succeed, otherwise false
```

```
     */
    fun synchronizeBrushing(profileId: Long): Single<Boolean>
}
```

This snippet of code is written in Kotlin.

IBrushing interface

Your **Brushing object should implement the *IBrushing*** interface and
your profile object should implement the *IProfile* to be able to use this module.

```
/**
 * Brushing interface.
 * Your brusing object will need to implement this interface in order to
use the brushing module
 * Those fields are required.
 */
interface IBrushing {
 val duration: Long // duration of the brushing
 val goalDuration: Int // goal duration set in the profile
 val timestamp: Long // timestamp of the brushing
 val quality: Int // quality of the brushing, value on 100
 val processedData: String? // processed data during the brushing . no need
to use or modify the json inside
 val date: ZonedDateTime
        get() = ZonedDateTime.ofInstant(Instant.ofEpochMilli(timestamp),
ZoneId.systemDefault())


    fun hasProcessedData(): Boolean {
        return !processedData.isNullOrEmpty()
    }
}
```

This snippet of code is written in Kotlin.

USAGE : How to load the brushings for a profile

```
/**
 * Get all the brushings for a profile
 */
private void getBrushingsByProfile(long profileId) {
  disposables.add(brushingManager.getBrushings(profileId)
      .subscribeOn(Schedulers.io())
      .observeOn(AndroidSchedulers.mainThread())
      .subscribe(
          listOfBrushings -> doStuffWithTheBrushings(listOfBrushings),
          Throwable::printStackTrace
      ));
}
```

This snippet of code is written in Java.

Coach+ module

Last modification : 17th sept 2018

By : Guillaume Agis

Revision : phase 4

What is it?

This library contains the Coach+ brushing activity
The memory management is done automatically, but you can preload the models to avoid latencies in your app (the models are heavy).

The coach+ module is included in the core module.

How to use

Preload a coach+

You can preload a coach+ at any time using the following method of the *MemoryManager* class.

```
/**
 * Preload the model of the COACH using the memory manager
 * from the COACH+ module in order to improve the loading performance and
avoid
 * any latencies
 */
private void preloadCoachModel() {
  disposables.add(memoryManager.preloadFromAssets(Kolibree3DModel.COACH)
      .subscribeOn(Schedulers.computation())
      .subscribe());
}
```

How to use

Once the previous section's steps are done, you can start the CoachPlusActivity creating an Intent using the following lines:

```
// to connect to a specific TB
Intent intent = CoachPlusActivity.createIntent(context,
toothbrushMacAddress, optionalColorSet);


OR


// to connect to the first TB that vibrates
Intent intent = CoachPlusActivity.createIntent(context);
```

- context is a valid Android context
- toothbrushMacAddress is the MAC address of the toothbrush you want to use for this session
- colorSet, you can provide a color set that will be used on the view. If you pass null to this parameter, the default color set will be used.

```
public static Intent createIntent(@NonNull Context context, @NonNull String
macAddress,
    @Nullable CoachPlusColorSet colorSet) {
  final Intent intent = new Intent(context, CoachPlusActivity.class);
  intent.putExtra(INTENT_TOOTHBRUSH_MAC, macAddress);
  intent.putExtra(INTENT_COLOR_SET, colorSet);
  return intent;
}
```

The activity will return **RESULT_OK** if a brushing session has been recorded, RESULT_CANCELED otherwise.

You can create a color set by using **CoachPlusColorSet.create()**.

The 4 arguments are explained below:

- backgroundColor: the color of the activity's background
- titleColor: the color of the "Coach+" labeled title
- neglectedColor: the neglected teeth color (not brushed color)
- cleanColor: the clean teeth color (fully brushed color)

Core module

Last modification : 05th oct 2018

By : Guillaume Agis

Revision : phase 3

The problem

More modules we develop, more dependencies the app will need to add.

It's very heavy and not as flexible as we wish.

So we develop the core module.

What is it?

The core SDK includes **all the modules** of the public SDK and provide all the dependencies that does not need to be provided by the client/app.

Still some dependencies will need to be provided such as the context, the clientId and the client secret as we can't provide this ourself.

Only one include is then needed to get access to all the modules and **that's all**.

Modules included

- Profile and Brushing module
- Stats module
- Jaws3D
- Offline brushing module
- Bluetooth module
- SMS auth
- Pairing module
- Coach+
- Pirate Game

Usage

Dagger

– includes CoreModule.*class in your Component file and that's all !*

Build.gradle

Add the library as a dependency inside your app build.gradle file.

```
def modulesVersion = "3.0.0"

    implementation group: 'com.kolibree.android', name: 'core', version:
"$modulesVersion"
```

How to include the core SDK

- Your @component needs to have a dependency on the SdkComponent
- Needs to include the coreModule.class
- Needs to provide a SdkComponent and a Context using the builder

```
@Singleton
@Component(
    dependencies = {SdkComponent.class},
    modules = {
        ...
        coreModule.class // dep for core module

    })
public interface MyComponent extends AndroidInjector<MainApp> {


  @Component.Builder
  interface Builder {

    @BindsInstance
    Builder context(Context context);

    Builder sdkComponent(SdkComponent sdkComponent);

    MyComponent build();
  }
}
```

Those dependencies can be provided in your Application class in such way :

```
SdkComponent sdkComponent = KolibreeAndroidSDK.init(this); // required,
otherwise the pairing module inside the core module will not work

MyComponent appComponent = DaggerMyComponent.builder()
    .context(this) // inject context dep
    .sdkComponent(sdkComponent) // inject SdkComponent dep
    .build();

appComponent.inject(this);
```

This snippet of code is in Java.

Jaws 3D module

Last modification : 10th sept 2018

By : Guillaume Agis

Revision : phase 2

What is it?

This module contains 3D jaws models and OpenGL renderers.
The memory management is done automatically, but you can preload the models to avoid latencies in your app (the models are heavy)
The library has been designed so you can use the 3D views in ViewPagers or multiple screens with the same memory usage.

The jaws 3D module is included in the core module.

How to use

Preload a model

You can preload a model at any time using the following method of the *MemoryManager* class.

```
/**
 * Preload the model of the JAWS using the memory manager
 * from the Jaws 3D module in order to improve the loading performance and
avoid
 * any latencies
 */
private void preloadModelJaws() {
  disposables.add(memoryManager.preloadFromAssets(Kolibree3DModel.CHECKUP)
      .subscribeOn(Schedulers.computation())
      .subscribe());
}
```

Use the model to render the mouth zone view

Make sure Dagger knows about your view by inject the view using the AppComponent variable that you have stored in your App.

You are required to do this when you load your activity/fragment and before using the view.

```
private JawsView jawsView;


...
MainApp.demoAppComponent.inject(jawsView);
```

The custom view to use is available here : com.kolibree.android.jaws.JawsView and you use it as a normal custom view in your layout XML file.

```
<com.kolibree.android.jaws.JawsView
  android:id="@+id/mouthView"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"/>
```

Only the **setData** method is public and needs to be used to provide the processed data (from the brushing session object) to the renderer .

You won't need to process any data received from the toothbrush. Just leave the raw data into String format and the JawView will know how to handle it.

```
/**
 * Set the raw processed data into the jawsView .
 * No need to do any computation of the data, the jawsView will handle that
part and convert
 * the raw processed data into the right format to build the view.
 * @param proccessedData raw data from the brushing
 */
public void setData(@NonNull String proccessedData) {
  setupRenderer();
 jawsRenderer.setProcessedData(proccessedData);
}
```

Offline and Orphan Brushings

Last modification : 4th dec 2018

By : Miguel Aragüés

Revision : phase 6

What is it?

This module has two goals
1. Retrieve Offline Brushings from connected toothbrush
2. Manage Orphan brushings, which are Brushings for which we don't know the owner. This can happen for shared toothbrushes on a multiprofile account

The offline and orphan brushings module is included in the core module.

Tools

OfflineBrushingsRetrieverViewModel

This ViewModel is a lifecycle-aware compoenent. When initialized correctly, it automatically detects and fetches an offline brushing from a connected toothbrush that belongs to the active profile or to a shared toothbrush.

In order to initialize it

1. *OfflineBrushingsRetrieverViewModel.Factory* needs to be injected. Make sure that you are providing a *Scheduler*, as shown in the *MyModule.java* snippet
2. Instantiate an *OfflineBrushingsRetrieverViewModel*
3. Add the instance as a *LifecycleObserver*
4. Subscribe to *OfflineBrushingsRetrieverViewState*
5. Update the UI when a new *OfflineBrushingsRetrieverViewState* is emitted

This snippet shows a complete usage of *OfflineBrushingsRetrieverViewModel*

OfflineBrushingsRetrieverViewModel

```
class MainActivity extends AppCompatActivity{

  //1. Inject a Factory
  @Inject
  OfflineBrushingsRetrieverViewModel.Factory
offlineRetrieverViewModelFactory;

  private void initOfflineBrushingsRetrieverViewModel() {
    //2. Instantiate the ViewModel
    OfflineBrushingsRetrieverViewModel offlineBrushingsRetrieverViewModel =
ViewModelProviders
        .of(this, offlineRetrieverViewModelFactory)
        .get(OfflineBrushingsRetrieverViewModel.class);

    //3. Add as a LifecycleObserver
    getLifecycle().addObserver(offlineBrushingsRetrieverViewModel);

    //4. Subscribe to OfflineBrushingsRetrieverViewState

disposables.add(offlineBrushingsRetrieverViewModel.viewStateObservable()
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(
            this::renderOfflineBrushingsRetrieverViewState,
            Throwable::printStackTrace
        ));
  }

  //5. Update the UI
  private void renderOfflineBrushingsRetrieverViewState(
      OfflineBrushingsRetrieverViewState offlineRetrieverViewState) {
    if
(offlineRetrieverViewState.haveRecordsBeenRetrievedForCurrentProfile()) {

showOfflineRecordsSynchedDialog(offlineRetrieverViewState.getRecordsRetrie
ved());
    }
  }
}
```

In order to inject the Factory, the easiest way is to use _dagger-android_. Refer to https://confluence.kolibree.com/display/SOF/Android+SDK+%3A+How+to+integrate+it+within+your+app for instructions

OrphanBrushing

Class representing a Brushing without a profile associated. This can happen on offline brushings when a Toothbrush is shared by multiple users.

SDKOrphanBrushingRepository interface

SDKOrphanBrushingRepository is reactive interface

- Notifies subscribers whenever an _OrphanBrushing_ is added or removed
- Exposes methods to manipulate the data

```
interface SDKOrphanBrushingRepository {

  @NonNull
  Flowable<Integer> count();

  @NonNull
  Single<OrphanBrushing> read(long id);

  @NonNull
  Flowable<List<OrphanBrushing>> readAll();
}
```

Check SDKDemo's CheckupFragment and OrphanBrushingsActivity for a complete example for counting, reading all Orphan brushings and assign/delete one of them. The read operation is not demonstrated, but using it should be straightforward.

OrphanBrushingsMapper interface

Exposes *assign(profileId, List<OrphanBrushing>)* and *delete(List<OrphanBrushing)*

Once the user has selected an action on the _OrphanBrushing_, _OrphanBrushingMapper_ should be used to assign a brushing to a profile or to delete it. This will perform the remote addition/deletion of the
associated brushing

Pairing module

Last modification : 17th oct 2018

By : Guillaume Agis

Revision : phase 4

What is it?

This module is responsible for scanning and pairing with the toothbrushes, using the public interface PairingAssistant.

By Injecting in your code, using the @Inject annotation, the interface PairingAssistant you get access to all the functionalities to discover and pair with toothbrushes .

The pairing module is included in the core module.

Add permissions in the manifest

For this module to work correctly and discover the toothbrushes nearby it's required to add those permissions into the manifest of your app and then ask to grant the permissions during the runtime.

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
```

Here is an example of how you can ask to grand those permissions :

```
/**
 * It's required to accept the locations permissions to make the scan works
 */
private void checkPermissions() {
  if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
    if
(this.checkSelfPermission(Manifest.permission.ACCESS_COARSE_LOCATION)
        != PackageManager.PERMISSION_GRANTED ||
        this.checkSelfPermission(permission.ACCESS_FINE_LOCATION)
            != PackageManager.PERMISSION_GRANTED) {
      final AlertDialog.Builder builder = new AlertDialog.Builder(this);
      builder.setTitle("This app needs location access");
      builder.setMessage("Please grant location access so this app can
detect toothbrushes.");
      builder.setPositiveButton(android.R.string.ok, null);
      builder.setOnDismissListener(
          dialog -> requestPermissions(new
String[]{Manifest.permission.ACCESS_COARSE_LOCATION,
                  Manifest.permission.ACCESS_FINE_LOCATION},
              PERMISSION_REQUEST_LOCATION));
      builder.show();
    }
  }
}
```

This snippet of code is in Java.

Tools : The pairing Assistant

```
/**
 * Pairing assistant
 *
 * Use this interface to  discover nearby toothbrushes,  pair with a
toothbrush and get the list of paired
 * toothbrushes for the current account
 */
interface PairingAssistant {

    /**
     * Check if the Bluetooth LE scanner is currently active
     *
     * @return true if the BLE scanner is currently scanning for
toothbrushes, false otherwise
     */
    fun isScanning(): Boolean
```

```
    /**
     * Get the toothbrush scanner [Observable]
     *
     * This observable will make the assistant start scanning for
toothbrushes as soon as an
     * observer subscribes to it.
     * The BLE scan will stop when the last observer leaves
     *
     * Don't forget to dispose your subscribers or the scanner will keep on
scanning
     *
     * The observable emits [ToothbrushScanResult] objects that you can
then pair with the
     * pair() method
     *
     * This observable will never complete but can return an error in case
of non recoverable hardware
     * failure
     *
     * @return non null [ToothbrushScanResult] [Observable]
     */
    @NonNull
    fun scannerObservable(): Observable<ToothbrushScanResult>

    /**
     * Pair a scanned toothbrush given the toothbrush information
     *
     * This toothbrush will be added to the database and the reconnection
process will be automated
     *
     * @param mac mac addr of the toothbrush
     * @param model model of the toothbrush
     * @param name name of the toothbrush
     * @return non null [Single] The created Pairing session
     */
    @NonNull
    fun pair(mac: String, model: ToothbrushModel,
             name: String, isRunningBootloader: Boolean = false):
Single<PairingSession>


    /**
     * Pair a scanned toothbrush given the toothbrush information
     *
     * This toothbrush will be added to the database and the reconnection
process will be automated
     *
     * @param result non null [ToothbrushScanResult] result of the Scan
     * @return non null [Single] The created Pairing session
     */

    @NonNull
    fun pair(@NonNull result: ToothbrushScanResult): Single<PairingSession>
```

```
    /**
     * Unpair toothbrush given a mac addr
     *
     * This method will remove toothbrush from database and forget current
connection
     *
     * @param mac toothbrush mac which will be removed [String]
     * @return non null [Completable]
     */
    @NonNull
    fun unpair(mac: String): Completable

    /**
     * Get the list of paired toothbrushes for the current account,
associated by its accountId
     * @return non null [Single]  [List]  [AccountToothbrush] list of
AccountToothbrush object
```

```
     */
    fun getPairedToothbrushes(): Single<List<AccountToothbrush>>
}
```

- *PairingSession* - cancelable pairing session, use this object to manage a connection to a toothbrush

Only accessible after Pairing a toothbrush.

```
interface PairingSession {
/**
* Cancel a pairing session
*/
fun cancel()
/**
* Returns connected [ToothbrushFacade] object
*/
fun toothbrush() : ToothbrushFacade
/**
* Returns connection [KLTBConnection] object
*/
fun connection(): KLTBConnection
}
```

This snippets of code are written in Kotlin.

Sample Usage

Inject *PairingAssistant* object by Dagger:

If you want to start scan, you should subscribe *scannerObservable()* like below:

```
@Inject // inject the interface in the current view
PairingAssistant pairingAssistant;



Disposable pairingAssistantDisposable;

void startScan() {
 pairingAssistantDisposable = pairingAssistant.scannerObservable()
  .observeOn(AndroidSchedulers.mainThread())
  .subscribe(
   this::onToothbrushFound,
   Throwable::printStackTrace
  );
}

void onToothbrushFound(@NonNull ToothbrushScanResult result) {
 //TODO: do something with result, if you want to pair to first result you
can use _pair_ method
}
```

If you want to stop scan, you can simply dispose previously created *Disposable* object:

```
void stopScan() {
 if (pairingAssistantDisposable != null &&
!pairingAssistantDisposable.isDisposed()) {
  pairingAssistantDisposable.dispose();
 }
}
```

If you want to pair with toothbrush, please use method *pair* and object returned by method *scannerObservable()*:

```
ToothbrushScanResult scanResult = scanResultReturnedByScannerObservable();
pairingAssistant.pair(scanResult)
 .observeOn(AndroidSchedulers.mainThread())
 .subscribe(session -> {
  session.connection(); // it returns _KLTBConnection_ object
  session.toothbrush(); // it returns _ToothbrushFacade_ object
 },
 Throwable::printStackTrace);
```

Those snippets of code are in Java.

Tools : The VibrationCheckerViewModel

Use to detect when a toothbrush vibrates.

```
private void initVibrationChecker() {
  vibrationCheckerViewModel =
      ViewModelProviders.of(this,
vibrationCheckerFactory).get(VibrationCheckerViewModel.class);

  disposeOnStopDisposables.add(
      vibrationCheckerViewModel
          .getViewStateObservable()
          .observeOn(AndroidSchedulers.mainThread())
          .subscribe(
              state -> {
                // check if the vib is on and there is a connection
                if (state.isVibrating() && state.getConnection() != null) {

startCoachPlusForToothbrush(state.getConnection().toothbrush().getMac());
                }
              },
              Throwable::printStackTrace));

  getLifecycle().addObserver(vibrationCheckerViewModel);
}


private void startCoachPlusForToothbrush(@Nullable String
toothbrushMacAddress) {
  Intent intent = CoachPlusActivity.createIntent(this,
toothbrushMacAddress, null);
  startActivityForResult(intent, REQUEST_CODE_GAME);
}
```

Pirate game

Last modification : 04th oct 2018

By : Guillaume Agis

Revision : phase 3

The pirate game is included in the core module.

How to use

```
Intent intent = PirateActivity.createPirateIntent(activity,
ActivityToComeBack.class);
```

```
/**
   * Create a pirate intent to use to open the game from another activity
   *
   * @param context current context
   * @param activityClassToOpen activity's class to open when finishing the
game
   * @return pirate game intent
   */
  public static Intent createPirateIntent(@NonNull Context context,
Class<?> activityClassToOpen) {
    Intent intent = new Intent(context, PirateActivity.class);
    intent.putExtra(ACTIVITY_TO_OPEN, activityClassToOpen);
    intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
    return intent;
  }
```

Profile module

Last modification : 17th octobre 2018

By : Guillaume Agis

Revision : phase 4

What is it?

This module is responsible for managing the profiles related to the current account.

By Injecting in your code, using the @Inject annotation, the interface ProfileFacade you get access to all the methods to add, create, edit and delete a profile associated to the current account.

All the methods need and will provide you a IProfile interface that contains the required attributes for this modules.

To use these methods, your custom profile object is required to **implement** the IProfile interface, otherwise you won't be able to use it.

The profile module is part of the web service module.

Dependencies in your app

- RxAndroid :Please read about Single<T> type if you are not familiar with Rx.

Tools

ProfileFacade interface

```
interface ProfileFacade {

    /**
     * Get the profile associated to a profileId
     *
     * @param profileId profileID associated to the profile to get
     *
     * @return non null [IProfile] [Single] profile
     */
    fun getProfile(profileId: Long): Single<IProfile>

    /**
     * delete the profile associated to a profileId
     *
     * @param profileId profileID to the profile to delete
     * @return non null [Boolean] [Single] true success, false otherwise
     */
    fun deleteProfile(profileId: Long): Single<Boolean>

    /**
     * Add a new profile to the current Profile
```

```
     * Your object needs to implement the IProfile interface to be able to
     * use this method.

     * @param profile profile to add
     * @return non null [IProfile] [Single] profile created
     */
    fun createProfile(profile: IProfile): Single<IProfile>

    /**
     * Edit a profile
     * Your object needs to implement the IProfile interface to be able to
     * use this method.
     *
     * @param profile profile to edit
     * @return non null [Boolean] [Single] success
     */
    fun editProfile(profile: IProfile): Single<IProfile>

    /**
     * Get the list of profiles associated to the current account
     *
     * @return non null [List] [IProfile] [Single] profiles associated to
the current account
     */
    fun getProfilesList(): Single<List<IProfile>>

    /**
     * Change the profile picture for a given profile
     *
     * @return non null [IProfile] [Single] profile updated
     */
    fun changeProfilePicture(profile: IProfile, picturePath: String?):
Single<IProfile>

    /**
     * Url to get an access token for account linking with external
backend.
     * @return non null [PrivateAccessToken] [Single] Response object
     */
    fun getPrivateAccessToken(): Single<PrivateAccessToken>


    /**
     * Login with wechat
     * @return non null [List] [IProfile] [Single] profiles fetched
     */
    fun loginWithWechat(code: String): Single<List<IProfile>>

    /**
     * registrer with wechat
     * @return non null [List] [IProfile] [Single] profiles created
     */
    fun registerWithWechat(code: String, profile: IProfile):
```

```
Single<List<IProfile>>

    /**
     * Return public Id
```

```
     */
    fun getPublicId(): String?
}
```

IProfile interface

Your custom profile object should implement the *IProfile* to be able to use this module. Those attributes are required in your object, but feel free to add any other attributes you need.

Don't forget our back-end who save your custom attributes. The back-end will just store the attributes contained in the IProfile interface.

If your code is in Kotlin, you will get access to the methods included in the interface. Those are only some helpers so if you code in JAVA, you can add them yourself in your custom Profile object.

```
interface IProfile {
    val id: Long
    val firstName: String // first name
    val gender: Gender // gender of the user
    val handedness: Handedness // hand used by the user
    val brushingGoalTime: Int // brushing goal time
    val createdDate: String // date of the profile creation
    val birthday: LocalDate? // date of birth of the user
    val pictureUrl: String? // profile picture url , can start with http or
any local url
    val country : String? // country of the user
    /**
     * Check if the current profile is a Male
     *
     * @return true if it's a Male, false otherwise [Gender]
     */
    fun isMale() = gender == Gender.MALE

    /**
     * Check if the current profile is a Right handed
     *
     * @return true if the user is Right Handed, false otherwise
[Handedness]
     */
    fun isRightHanded() = handedness == Handedness.RIGHT_HANDED

    /**
     * Get the age of the user given the birthday atribute.
     * Return -1 if the birthday has not been defined
     *
     * @return the age of the user [Int]
     */
    fun getAgeFromBirthday() = birthday?.let {
KolibreeUtils.getAgeFromBirthDate(it) } ?: -1
}
```

This snippet of code is written in kotlin.

Example of Profile object you can use

```
/**
 * Your custom profile that will implement our IProfile from the SDK
 * you can add as many attributes as you wish  as long as there are the
ones needed by the IProfile
 * interface.
 *
 */
public class DemoProfile implements IProfile {

  private final Long id;
  private final String firstName;
  private final Gender gender;
  private final Handedness handedness;
  private final int brushingGoalTime;
  private final String createdDate;
  private final LocalDate birthday;

  /**
   * Basid constructor, but you can have as many attributes as you need,
those ones are only
   * the one needed by the SDK, that you will need to provide
   * @param id
   * @param firstName
   * @param gender
   * @param handedness
   * @param brushingGoalTime
   * @param createdDate
   * @param birthday
   */
  public DemoProfile(Long id, String firstName, Gender gender, Handedness
handedness,
      int brushingGoalTime, String createdDate, LocalDate birthday) {
    this.id = id;
    this.firstName = firstName;
    this.gender = gender;
    this.handedness = handedness;
    this.brushingGoalTime = brushingGoalTime;
    this.createdDate = createdDate;
    this.birthday = birthday;
  }


  @Nullable
  @Override
  public LocalDate getBirthday() {
    return birthday;
  }
```

```
  @Override
  public int getBrushingGoalTime() {
    return brushingGoalTime;
  }

  @NotNull
  @Override
  public String getCreatedDate() {
    return createdDate;
  }

  @NotNull
  @Override
  public String getFirstName() {
    return firstName;
  }

  @NotNull
  @Override
  public Gender getGender() {
    return gender;
  }

  @NotNull
  @Override
  public Handedness getHandedness() {
    return handedness;
  }

  @Override
  public long getId() {
    return id;
  }

  @Override
  public int getAgeFromBirthday() {
    return KolibreeUtils.getAgeFromBirthDate(birthday);
  }

  @Override
  public boolean isMale() {
    return gender == Gender.MALE;
  }

  @Override
  public boolean isRightHanded() {
    return handedness == Handedness.RIGHT_HANDED;
  }


 @Override
  public String getCountry() {
```

```
    return "CN";
 }
}
```

USAGE : How to load a profile and how to register with wechat

```
/**
 * Load current profile
 *
 * @param profile current profile
 */
private void loadProfile(long profileId) {

  disposables.add(profileManager.getProfile(profileId)
      .subscribeOn(Schedulers.io())
      .observeOn(AndroidSchedulers.mainThread())
      .subscribe(profileFetched -> displayProfileInfo(profileFetched),
          Throwable::printStackTrace
      ));
}


private void registerWechat(DemoProfile profile, String wechatCode) {
  disposables.add(
      profileManager
          .registerWithWechat(wechatCode, profile)
          .subscribeOn(Schedulers.computation())
          .subscribe(
              result -> Log.e("registerWechat", result.toString()),
              Throwable::printStackTrace));
}
```

This snippet of code is written in java.

Stats module

Last modification : 10th sept 2018

By : Guillaume Agis

Revision : phase 2

What is it?

This module is responsible for computing statistics based on brushing sessions for a profile, associated by a profile id.

Under the hood, everytime there is an update into the local brushing table , the brushings stored in the local table are converted into

stat and store locally. It keeps the stat up to date at any moment.

As a reminder, you need to use the synchronised method from the BrushingModule in order to synchronised your remote (on the server) and local brushings,

that get converted into Stats automatically.

By Injecting in your code, using the @Inject annotation, the interface DashboardCalculatorView you get access to the methods to get the past weekly stats for a given user.

The stat module is included in the core module.

Tools

DashboardCalculatorView interface

Responsible for providing the weeklyStat for a given profile.

In overall, you should be needed the **getPastWeekStats** method from the DashboardCalculatorView mainly as it s the one responsible for getting your list of Stat for the past week and convert it for you into a single WeeklyStat object for the user you provided the profileId.

```
/**
  * Get the past weekly stat for a given profile for a given profileId
  *
  * @param profileId the id of the profile to get the stat to.
  * @return a Flowable with the past weekly stats
  */
  fun getPastWeekStats(profileId: Long?): Flowable<WeeklyStat>

  /**
   * Calculates the week startDate from an endDate
   *
   * If endDate is after 4AM (Kolibree start time), we want to go back 6
days. Otherwise, we'll go
   * back 7 days
   *
   * Imagine the following scenarios. Both of them use 7 days.
   *
   * 1. Today is Sunday at 11:00 AM
   * · We want to calculate the averages including today's brushings, so we
want to take into
   * account brushings starting from Monday's brushings (today - 6.days)
   *
   * 2. Today is Sunday at 3:00 AM
   * · We don't want to take into account today's brushings because from
Kolibree's point of view,
   * today is still Saturday, so we need to take into account brushings
starting from past Sunday
   * (today - 7.days)
   *
   *
   * @param endDate date to adjust to
   * @return a ZonedDateTime with the adjusted date
   */
  fun adjustedStartDate(endDate: ZonedDateTime): ZonedDateTime
```

StatRepository interface

Manage the stat data into and store them locally into Room
Every time some data into the `BrushingsRepository` is updated, it will emit the brushing data
in this class and update the stat data stored locally accordingly.

PS : This interface is use by the DashboardCalculatorView so you should not need to use directly this interface.

Methods

```
/**
 * Get the list of stats from a given date for a given profileId
 *
 * @param startTime the date to get the stat from , a ZonedDateTime object
 * @param profileId the id of the profile to update the stat
 * @return a Flowable with a list of Stats object
 */
fun getStatsSince(startTime: ZonedDateTime, profileId: Long):
Flowable<List<Stat>>
```

This method is used by the implementation of the DashboardCalculatorView, but you probably won'y need it exactly you will prefer to use a List of Stat instead of WeeklyStat object.

Bear in mind that the WeeklyStat object contains everything you need to display the weekly average session.

Please notice that those methods return a Flowable. Flowable emits every time a new update has been made, then your snippet of code that subscribes to this method will always be called every time there is an update in the Stat database, and then impact your Weeklystat.

If you are not familiar with a reactive approach, I'll recommend you to inform you about it and Flowable to better understand how it works and what to expect.

Usage

```
/**
 * Get the weekly stat for the current profile
 *
 * @param profileId profile Id of the profile to laod the data
 */
private void getWeeklyStatForProfile(long profileId) {

disposables.add(dashboardCalculatorView.getWeeklyStatForProfile(profileId)
// get the weekly stat for a user
      .subscribeOn(Schedulers.io())
      .observeOn(AndroidSchedulers.mainThread())
      .subscribe(
          this::updateStatGraphes, // get called every time the weeklyStat
is updated
          Throwable::printStackTrace // print the stacktrace of the error
      ));
}

/**
 * Display some stats from the weeklyStat
 */
private void updateStatGraphes(WeeklyStat weeklyStat) {
  tvAverageBrushingDuration.setText(String
      .format(getString(R.string.avg_brushing_duration),
weeklyStat.getAverageBrushingTime())); // display the avg brushing time on
screen

  tvAverageBrushingSurface.setText(String
      .format(getString(R.string.avg_brushing_surface),
weeklyStat.getAverageSurface())); // display the avg surgace on screen
}
```

Those snippets of code are written in Kotlin.

The Webservice module

Last modification : 6th sept 2018

By : Guillaume Agis

Revision : phase 2

The Web service module contains the core of the project. It contains the hidden part of the iceberg and everything you don't see to make sure the brushingModule and the ProfileModule and the Stats works correctly.

The Brushing and Profile modules are inside the core SDK and then share the same dependencies.

The Stats is only dependent of this module as it requires the brushingModule to work.

The web service module is included in the core module.

Technologies used

- Android Room
- SQLite
- Dagger
- RxAndroid
- Retrofit
- Gson
- ....

Please read the documentations related to those modules in order to learn about their usage.

Modules included :

- Brushing module
- Profile module

Toothbrush Module

Last modification : 6th sept 2018

By : Guillaume Agis

Revision : phase 2

What is it?

This module is responsible for all Kolibree toothbrush Bluetooth connections, commands and hardware-related functionnality.

Tools

ToothbrushFacade interface

```
interface ToothbrushFacade {

    /**
     * Get the toothbrush connection state interface
     *
     * @return non null [ConnectionStateWrapper]
     */
    fun state(): ConnectionStateWrapper


    /**
     * @return Observable that will emit true if there's an OTA available,
false otherwise. Each
     * invocation can return a new instance
     */
    fun hasOTAObservable(): Observable<Boolean>

    /**
     * Set default brushing duration in seconds
     *
     * @param defaultDurationSeconds int seconds
     * @return non null [Completable]
     */
    fun setDefaultBrushingDuration(defaultDurationSeconds: Int):
Completable

    /**
     * Get default brushing session duration
     *
     * @return non null [Integer] [Single] seconds
     */
    fun getDefaultBrushingDuration(): Single<Int>
```

```
    /**
     * Set auto shutdown timeout
     *
     * @param autoShutdownTimeout auto shutdown timeout in seconds
     * @return non null [Completable]
     */
    fun setAutoShutdownTimeout(autoShutdownTimeout: Int): Completable

    /**
     * Get auto shutdown timeout
     *
     * @return non null [Integer] invoker
     */
    fun getAutoShutdownTimeout(): Single<Int>


    /**
     * Get toothbrush's current user ID
     *
     * @return non null Single
     */
    fun getUserId(): Single<Long>

    /**
     * Set the toothbrush in single user mode and set the user ID
     *
     * @param userId long user ID
     * @return non null [Completable]
     */
    fun setUserId(userId: Long): Completable

    /**
     * Enable multi user mode
     * Offline brushing will not be recorded anymore
     *
     * @param enable true to enable multi user mode, false to disable it
     * @return non null [Completable]
     */
    fun setMultiMode(enable: Boolean): Completable

    /**
     * Get toothbrush's multi users mode state
     *
     * @return a Single that will emit true if the toothbrush is in multi
users mode, false otherwise
     */
    fun isMultiModeEnabled(): Single<Boolean>


    /**
     * Get the toothbrush bluetooth name
     *
     * @return bluetooth device name
```

```
 */
fun getName(): String

/**
 * Set the toothbrush bluetooth name
 *
 * @param name non null name
 * @return non null [Completable]
 */
fun setName(name: String): Completable

/**
 * Get toothbrush bluetooth mac address
 *
 * @return non null String encoded mac address
 */
fun getMac(): String

/**
 * Get toothbrush model
 *
 * @return non null [ToothbrushModel]
 */
fun getModel(): ToothbrushModel

/**
 * Get the toothbrush hardware version
 *
 * @return non null [HardwareVersion]
 */
fun getHardwareVersion(): HardwareVersion

/**
 * Get the toothbrush serial number
 *
 * @return non null serial number
 */
fun getSerialNumber(): String

/**
 * Get the firmware version
 *
 * @return non null [SoftwareVersion]
 */
fun getFirmwareVersion(): SoftwareVersion

/**
 * Check if the firmware is running in bootloader mode
 *
 * @return true if running in bootloader mode, false otherwise
 */
fun isRunningBootloader(): Boolean
```

```
/**
 * Check if the battery is currently being charged
 *
 * @return non null [Single]
 */
fun isCharging(): Single<Boolean>

/**
 * Get the battery level in percents
 *
 * @return non null [Single]
 */
fun getBatteryLevel(): Single<Int>

/**
 * Check if the toothbrush firmware has valid GRU data for RNN detector
 *
 * @return true if the GRU data is valid, false otherwise
 */
fun hasValidGruData(): Boolean


/**
 * Check if the updates are available (GRU or firmware)
 *
```

```
     * @return non null [Single]
     */
    fun checkUpdates(): Single<GruwareData>
```

ConnectionStateWrapper interface

```
interface ConnectionStateWrapper {

    /**
     * Get the current state of the connection
     *
     * @return non null [KLTBConnectionState]
     */
    fun getCurrent(): KLTBConnectionState

    /**
     * Return true if connected to toothbrush connection, false otherwise
     */
    fun isActive(): Boolean

    /**
     * check if the connection is registered or not
     */
    fun isRegistered(): Observable<Boolean>
}
```

Those snippets of code are written in kotlin.

Once your pair with a toothbrush, you get access to the toothbrushWrapper.

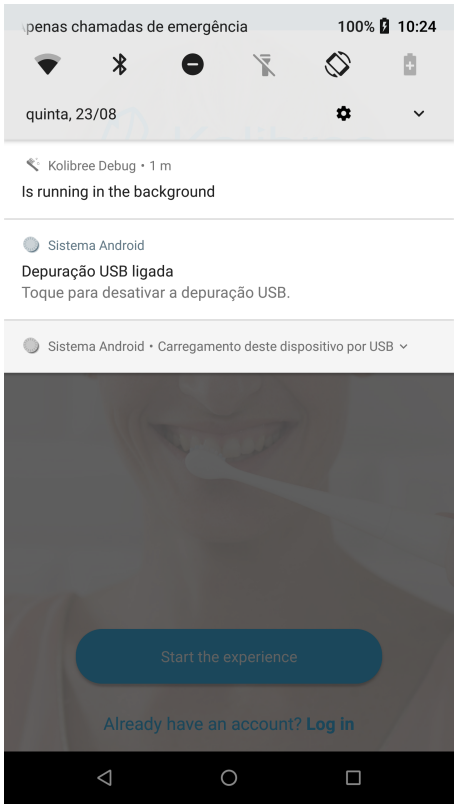
Providing Translations

If you wish to change the text displayed in the status bar when KolibreeService is running, you can provide your own TranslationsProvider when initializing the component

```
class MyApplication extends Application {
  ...


  private void initDagger() {
    TranslationsProvider translationsProvider = new TranslationsProvider();

    Map<String, String> map = new HashMap<>();
    map.put(RUNNING_IN_BACKGROUND_KEY, "New english message");
    translationsProvider.addLanguageSupport(Locale.US, map);

    Map<String, String> chinaMap = new HashMap<>();
    chinaMap.put(RUNNING_IN_BACKGROUND_KEY, "Chinese message");
    translationsProvider.addLanguageSupport(Locale.SIMPLIFIED_CHINESE,
chinaMap);

    SdkComponent sdkComponent = KolibreeAndroidSDK.init(this,
translationsProvider);

    //noinspection NullableProblems
    appComponent = DaggerAppComponent.builder()
        ...
        .sdkComponent(sdkComponent)
        .build();

    appComponent.inject(this);
  }
}
```

Supported keys

- RUNNING_IN_BACKGROUND_KEY:
  - fallback text = "Is running in background"
  - text displayed in a notification in the status bar when we there's a background service running
  - A portuguese user would see the default message


penas chamadas de emergência
100% 10:24
quinta, 23/08
Kolibree Debug • 1 m
Is running in the background
Sistema Android
Depuração USB ligada
Toque para desativar a depuração USB.
Sistema Android • Carregamento deste dispositivo por USB
Start the experience
Already have an account? Log in

Toothbrush Update

What is it?

This module has two goals
1. Provide an easy way to check if any of the connected toothbrush needs to be updated
2. Provide the screens and logic to update a Toothbrush's Firmware and Gru data

The Toothbrush Update module is included in the core module.

Important information

There are two circumstances under which we need to update a toothbrush

1. Periodically, we release new firmwares that improve the toothbrush' performance or zone detection reliability
2. The toothbrush is in bootloader mode and is unusable (bright green led)

Upgrading the toothbrush is a critical process that can leave the toothbrush in an unusable state (in bootloader)

The update files are provided by the backend, thus they need to be downloaded. On application start, we need to check if new versions have been released for our connected toothbrushes or if it's in bootloader mode

Classes

OTA = Over The Air update

OtaForConnection

Specifies if there's an update available for the KLTBConnection, and the type of update

- STANDARD: A new version is available for the toothbrush.
- MANDATORY: The toothbrush is in an unusable state and needs to be upgraded with some files we have already downloaded. The user should be forced to update as soon as possible.
- MANDATORY_NEEDS_INTERNET: The toothbrush is in an unusable state, but we haven't been able to download the files. We should ask the user to enable internet so that we can proceed to download them. Our code will detect when there's connectivity and start the download immediately, there's no further action needed. Once we've downloaded the files, we will emit a new *OtaForConnection* with type *MANDATORY*

OtaChecker

Exposes *otaForConnectionObservable()*, which will emit [0-N] *OtaForConnection*, where N is the number of *KLTBConnections*.

From the javadoc of *otaForConnectionObservable()*

> *Checks if any KLTBConnection needs an Over The Air (OTA) update*
>
> *If it's a mandatory update and we can't fetch the data to update the toothbrush, the Observable will emit a OtaForConnection with type MANDATORY_NEEDS_INTERNET.*
>
> *Other possible types include MANDATORY and STANDARD.*
>
> *MANDATORY and MANDATORY_NEEDS_INTERNET need immediate action from the user. Since the toothbrush is in an unusable state, they are not safe to ignore.*
>
> *STANDARD updates will flag that a connection needs to be updated. This can be checked on any connection by subscribing to hasOtaObservable()*
>
> *If there's no mandatory update AND there's no internet, the observable won't emit an item for that KLTBConnection*

An example of how to use it

```
public class OtaCheckerViewModel extends ViewModel
    implements DefaultLifecycleObserver {
  @Override
  public void onStart(@NonNull LifecycleOwner owner) {
    if (checkOtaDisposable == null || checkOtaDisposable.isDisposed()) {
      checkOtaDisposable = otaChecker.otaForConnectionObservable()
          .subscribeOn(Schedulers.io())
          .observeOn(AndroidSchedulers.mainThread())
          .subscribe(
              this::onOtaForConnection,
              Throwable::printStackTrace
          );

      disposables.add(checkOtaDisposable);
    }
  }

  @Override
  public void onStop(@NonNull LifecycleOwner owner) {
    disposables.clear();
  }

  @VisibleForTesting
  void onOtaForConnection(OtaForConnection otaUpdateForConnection) {
    switch (otaUpdateForConnection.getOtaUpdateType()) {
      case STANDARD:
        //This KLTBConnection now will emit true for hasOTAObservable()
        break;
      case MANDATORY:
        onMandatoryUpdateNeeded();
        break;
      case MANDATORY_NEEDS_INTERNET:
        requestEnableInternet();
      default:
        //do nothing
    }
  }
  }
```

OtaUpdateActivity

When we want to start a firmware upgrade on a KLTBConnection, start *OtaUpdateActivity*

```
Intent intent = OtaUpdateActivity.createIntent(context, mac,
isMandatoryUpdate);
startActivityForResult(intent);
```

If there's no OTA for the KLTBConnection associated to the mac address, when the user clicks *Upgrade* in *OtaUpdateActivity* an error message will tell the user that there's no update for the toothbrush

The Activity will finish with RESULT_OK or RESULT_CANCELED

Custom Translations

Usage

If you whish to change any text displayed, you can provide your own TranslationsProvider

```
void provideTranslations() {
    TranslationsProvider translationsProvider = new TranslationsProvider();

    Map<String, String> map = new HashMap<>();
    map.put(FIRMWARE_UPGRADE_WELCOME_KEY, "New english message");
    translationsProvider.addLanguageSupport(Locale.US, map);

    Map<String, String> chinaMap = new HashMap<>();
    chinaMap.put(FIRMWARE_UPGRADE_WELCOME_KEY, "Chinese message");
    translationsProvider.addLanguageSupport(Locale.CHINA, chinaMap);

    Translations.init(context, translationsProvider);
  }
```

Supported keys

The default messages are subject to changes if POEditor literals are changed

- FIRMWARE_UPGRADE_WELCOME_KEY: Welcome message when landing on the screen

default english: "Upgrading your toothbrush will take a few minutes, please don't turn off your toothbrush."
default chinese: ""

- FIRMWARE_UPGRADE_INSTALLING_KEY: Displayed while installing

default: "Installing..."
default chinese: ""

- FIRMWARE_UPGRADE_REBOOTING_KEY: Displayed while rebooting

default: "Rebooting..."
default chinese: ""

- FIRMWARE_UPGRADE_ERROR_KEY: Displayed when there's an unknown error

default: "Something went wrong. Please try again later."
default chinese: ""

- FIRMWARE_UPGRADE_ERROR_DIALOG_KEY: Displayed in a popup dialogwhen there's an error

default: "Firmware upgrade failed:
<placeholder>"
default chinese: "
<placeholder>"

- FIRMWARE_UPGRADE_CANCEL_DIALOG_MESSAGE_KEY: confirm upgrade exit

default: "Are you sure you want to cancel the update?"
default chinese: ""

- POPUP_TOOTHBRUSH_UNAVAILABLE_MESSAGE_KEY: message when we can't find the toothbrush

default: "Make sure your toothbrush is on and bring it closer to your device, it will connect automatically."

default chinese: ""

- FIRMWARE_UPGRADE_OTA_NOT_AVAILABLE_KEY: no update available for the toothbrush

default: "No update available"
default chinese: ""

- FIRMWARE_UPGRADE_SUCCESS_KEY: Update succeeded

default: "Your toothbrush is up to date."
default chinese: ""

- FIRMWARE_UPGRADE_CANCEL_KEY: Cancel firmware upgrade. Only shown in non-mandatory updates.

default: "Cancel"
default chinese: ""

- FIRMWARE_UPGRADE_KEY: Start firmware upgrade

default: "Upgrade"
default chinese: ""