sa.engine Android Interfaces

Stream Analyze Sweden AB
Sweden

Version 2.0
2020-12-11
`sa_AndroidAPI_2.0.pdf`

The sa.android library is a modified version of the sa.engine Java Interface that is tailored to be able to use sa.engine in Android applications. This document describes how to set up and use the sa.android library as well as the main differences in usage, with regard to the sa.engine Java Interface.

Table of contents

1. Introduction

There are two main differences between sa.android and the sa.engine Java API, the *setup procedure* and *background execution management*:

- The *setup procedure* of sa.android is slightly different from the the sa.engine Java API. It involves adding the sa.android Android Archive-file (AAR) to an application project and performing some additional runtime setup to tie the context of an Android application to sa.engine.

- The Android platform is built with mobility in mind. This usually involves devices powered by batteries. To decrease the battery consumption of applications, the operating system of Android devices may incorporate strict rules on when and how an application can execute. In order to properly utilize the sa.android library special attention might have to be given to *background execution management*.

This documentation introduces the Android-specific parts of the Android API of sa.engine through a number of examples. Since sa.android is an extension of the sa.engine Java API you are assumed to be familiar with OSQL and the sa.engine Java API. You are also assumed to be familiar with the basic concepts of the Android platform.

2. Set up the sa.android library

The first step that needs to be taken to be able to use the sa.android library is to add it to the project of the application where the library will be used. Download the latest version of the `sa.android.aar` and add it to a library directory somewhere in your project. After adding the AAR-file, add it as a dependency to the module where you want to use it. The following example demonstrates how to add AAR-files as a dependency in a Gradle module:

```
// In build.gradle

dependencies {
    implementation fileTree(include: ['*.aar'], dir: 'libs')
}
```

Example 1. Adding AAR-files in the directory named 'libs' as dependencies to a Gradle module.

2.1. Initializing the Android connection

The Java API objects of the class `Connection` represent connections to an sa.engine server. This is true in sa.android as well but in order to be able to execute queries objects of the `AndroidConnection` class, which is a subclass of `Connection`, should be used instead.

Before creating an `AndroidConnection` some additional run time set up must be performed. First the Android connection must be initialized with an Android application `Context` by calling `AndroidConnection.init(context)`. If you are going to use dispatched queries in your models (see Section 3.2) you can supply a thread pool size to the `init` -method if the default thread pool size is insufficient for the models of your dispatched queries.

```
// In an Android Activity, Service, Application etc.

// Using default thread pool size.
Context applicationContext = getApplicationContext();
AndroidConnection.init(applicationContext);

// Using a supplied thread pool size.
Context applicationContext = getApplicationContext();
int threadPoolSize = 8;
AndroidConnection.init(applicationContext, threadPoolSize);
```

Example 2. Initializing the AndroidConnection with the default dispatcher thread pool size and with a supplied thread pool size.

2.2 Enabling logging

The log messages of sa.engine are flushed to the standard streams by default. Since these streams are normally not available to a user in an Android application the log messages can be intercepted when using sa.android. To enable or disable logging in sa.android the method `AndroidConnection.enableAndroidLogger(enable)` should be called. When enabled, a registered logging-method will be called every time a new sa.engine log message is flushed. To register such a logger-method, call `AndroidConnection.registerLogger(logger)`. The log message can then be handled in whatever way is best suited for your application.

```
boolean enableLogging = true;
AndroidConnection.enableAndroidLogger(enableLogging);
AndroidConnection.registerLogger(s -> Log.d("sa.engine", s));
```

Example 2. Logging enabled with a logging method that logs the message to the debug logs of Logcat with the tag "sa.engine".

2.3 Initializing the sensor interface

To be able to query the sensors of an Android device in sa.android the sensor interface must be initialized with an application context. If your model will not query any Android sensors this initialization step can be omitted.

```
// In an Android Activity, Service, Application etc.

Context applicationContext = getApplicationContext();
DeviceSensors.setContext(this);
```

Example 3. Enabling the possibility to execute queries that accesses Android device sensors.

3. Background execution management

Starting from Android 8.0 (Oreo) limitations were introduced on how and when an application can execute tasks while in the background, i.e., while an application activity is visible, or a foreground service is running. While these changes decrease unnecessary power consumption in most applications, they introduced additional effort for applications having use cases that are required to run while the application is in the background.

Continuous queries are a first class citizens in sa.engine and such queries may execute indefinitely. If your application will run in a controlled environment, and its power supply will be virtually infinite, the indefinite execution of a continuous query can be achieved by simply never letting the application be paused. However, if the device that is running your application is battery powered and your application can be paused, the continuous queries in your models are subject to be halted by the operating system. To counter this, a combination of a foreground `Service` and `WakeLock` can be used.

NOTE: By using wake locks your application may consume considerably more battery power. If your model does not need to run indefinitely the actions in this section should be disregarded.

3.1. Executing in a foreground service with wake lock

The first step to make it possible to execute your model in the background indefinitely is to start a foreground `Service`. When a foreground `Service` is running, your application is in a foreground state even if no activity of your application is visible. A foreground service is required to execute a long running task without being interrupted.

A foreground service alone is not enough to execute a query indefinitely since the operating system might determine that the CPU usage of the application should be put to sleep. To prevent this from happening the application must acquire and hold a `WakeLock` while the query is executing.

```
PowerManager powerManager =
    (PowerManager) context.getSystemService(POWER_SERVICE);
WakeLock wakeLock = null;

try {
    wakeLock = powerManager.newWakeLock(PARTIAL_WAKE_LOCK, "my_wake_lock");
    wakeLock.acquire();
    AndroidConnection connection = new AndroidConnection();
    connection.query("my_continous_query()")
              .mapAll(result -> {
                  // Do something with the result
              });
} finally {
    // Release the wake lock regardless of query outcome
    if (wakeLock != null) {
        wakeLock.release();
    }
}
```

Example 4. Acquiring and releasing a partial wake lock when executing a query.

3.2 Dispatched queries and calls

Android Services are executed on the Android main thread since they are considered UI components. The `Connection`-methods `query` and `call` are blocking and should therefore be executed on a different thread. If you do not want to manage the threads and wake locks on your own, sa.android provides *dispatched queries and calls*. They can be used in a similar way as the regular `query`- and `call`-methods with the difference that they are non-blocking and return a `DispatchedObjectStream` instead of an `ObjectStream`. When executing a stream operation (e.g., `map`, `mapAll`, or `run`) on a `DispatchedObjectStream` a wake lock will be automatically acquired and released. One thing to keep in mind is that the results returned in the callback methods of the stream operations will be running on a separate thread.

```
AndroidConnection connection = new AndroidConnection();
DispatchedObjectStream objectStream =
    connection.dispatchedQuery("my_continous_query()");
objectStream.mapAll(result -> {
    // Do something with the result
});
```

Example 5. Performing a dispatched query that will be executed on a different thread.

The purpose of dispatched queries and calls is to enable continuous queries while an application is running in the background on Android devices. As previously mentioned, this requires running a foreground service as well. If a stream operation is performed on a

`DispatchedObjectStream` and no foreground service is running in the process of the application a warning message will be logged since your query is then prone to be halted by the operating system.

To ensure that you are not holding any wake locks you should terminate the dispatched object streams when they should no longer execute, for example when a service is killed by the operating system. This can be done by calling the `terminate`-method on a dispatched object stream. If all dispatched need to be terminated the method `Connection.terminateDispatchedQueries()` should be called.

```
// In the foreground Service.

@Override
public void onDestroy() {
    super.onDestroy();
    AndroidConnection.terminateDispatchedQueries();
}
```

Example 6. Terminating all previously dispatched queries when the foreground Service is being destroyed.

3.3 Disabling battery optimizations

Newer Android devices may have built-in battery optimizers. The purpose of these battery optimizers is to detect and stop applications that perform extensive background work, even after the application has acquired wake locks and running a foreground service. The behaviour of the battery optimizers differs between device manufacturers and may require the user to update his/her settings to prevent an application to be stopped or killed by the operating system. To read more about how to prevent battery optimization issues for a device, read the instructions on https://dontkillmyapp.com/.