Java SDK

The BellaDati Java SDK is an easy-to-use implementation to access the REST API using Java. All data is made available in Java objects, eliminating the need to manually parse and interpret JSON responses from the server.

Setting up the SDK

⚠ For the latest version of SDK, please use Maven.

The SDK consists of several parts:

- API containing the SDK's interface declarations (with Javadoc online and with Javadoc download)
- standard Java implementation to use in a regular Java VM
- Android implementation to use when building Android apps

To use the SDK, download the API jar and the implementation matching your environment. When upgrading, make sure the versions of both jars match.

The SDK source is available on GitHub.

Dependencies

Dependencies for the SDK depend on the implementation you are planning to use. You can manually set up the dependencies or configure Maven to do it for you.

Regular Java VM

Using Maven

Add the following dependency and dependency management entries:

Click to view Maven configuration ...

Manual Setup

Download and add the following libraries to your classpath:

- Jackson 2.2.x core, annotations and databind libraries
- Signpost 1.2.x core and Commons HTTP implementation
- Apache HttpClient 4.3.x and its dependencies (jars fluent-hc and httpmime are not needed)

Android

Using Maven

In your `build.gradle` file, add the following repository and dependency entries:

Click to view Gradle configuration ...

Manual Setup

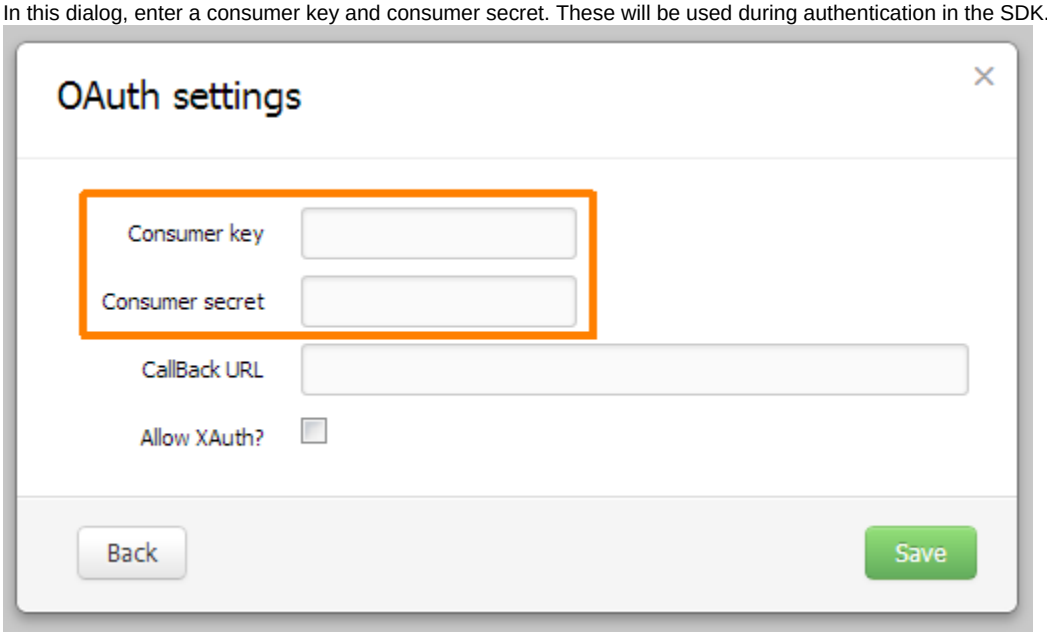
Download and add the following libraries to your project:

- Jackson 2.2.x core, annotations and databind libraries
- Signpost 1.2.x core
- Re-packaged Apache HttpClient for Android provided by BellaDati

Server Setup

In order to access data from BellaDati, you need to enable API access in your **domain**. To do so, open your domain settings and click **Configure** under **OAuth Settings**.

In this dialog, enter a consumer key and consumer secret. These will be used during authentication in the SDK.



You can optionally set a callback URL that will be opened when a user successfully authorizes an OAuth request - more on that in the example below. Finally, if your application environment doesn't allow using a web browser for authentication, you can enable xAuth to authenticate with username and password from within your client application.

Usage Example

In this example, we will use the SDK to authenticate using OAuth or xAuth and retrieve some data visible to our example user.

Authentication

Before we can read data through the SDK, we have to authenticate as a valid user. In this example, let's assume that we have set up a domain with a consumer key **sdkKey** and a consumer secret **sdkSecret**. The first step is to connect to our server:

Alternatively, we could use:

OAuth

The recommended form of authentication is OAuth. Using this protocol, the SDK requests a token from the server that the user then has to authorize using their web browser. The advantage of this mechanism is that the user sends their credentials directly to the server, reducing the risk of them getting intercepted on the computer running the SDK.

In the first step, we let the SDK get a request token from the server:

Now, we need to ask our user to authorize this token:

This is where the callback URL mentioned above comes in: If you are writing a web application, you can use the callback to redirect the user back

to your application after authorizing access.

Once the user has successfully authorized our application, we can request access from the server:

xAuth

For some applications, it is not possible or feasible to use a web browser for authentication. In these situations, you can use xAuth to have the user enter their credentials directly into your application and use them as follows:

To make xAuth in Android studio create class extends AsyncTask and call mathod connection.xAuth there.

Than in your main class:

Retrieving Data

After successful authentication, we now have an instance of the BellaDatiService interface. This interface offers several methods to get data from the server. In our example, let's first get a list of reports along with thumbnail images to display to the user.

This call consists of multiple steps: getReportInfo() gives us a PaginatedList (specifically, a PaginatedIdList) of reports, which is initially empty. Calling load() contacts the server and fetches the first page of reports. A PaginatedList is Iterable (so you can use it in loops) and offers methods like size(), get(), contains() and others to find out about its contents, as well as several methods to work with the pagination itself. Finally, if your use case requires a regular List, you can easily convert it by calling toList().

The ReportInfo objects inside the list contain general information about our reports, such as their names, descriptions and owners. To load the thumbnails, we have to make a separate request to the server for each report we want to load. Since these requests may take a some time depending on server latency, we want to load our thumbnails in parallel. For example:

You may notice that loadThumbnail() returns Object, which is cast to BufferedImage in the example above. This is to support different implementations - on Android (which doesn't have BufferedImage), you'll get an Android Bitmap instead.

In some situations, perhaps in a web application, you might just want to load a thumbnail for a given report ID, without having to load the report itself first. To do this, you can instead use:

Now we have a list of reports including thumbnail images that we can show to our user, asking them to select which one they would like to see.

Let's say our user has selected the 3rd report in the list. We can get its contents:

This Report object contains more details about the report, most importantly a list of its Views. Each chart, table, KPI etc. inside a report is represented by such a View. We can iterate over the views and display them:

As you can see, views can have different types, which are rendered differently. Most views contain JSON content, which is returned as a JsonNode from the Jackson library. Tables however contain Table objects, which have 3 additional load methods to load the table's contents. You can refer to the TableView for details on how to load tables.

In this example, we are loading all views one by one in a single thread, which may be slow. It can easily be done in parallel by using a mechanism similar to what we did for thumbnail images above, but we decided to omit it here for clarity.

General Advice

Try not to make many calls to the server in sequence to avoid causing long loading times for your users. When loading multiple items, consider loading them in parallel. It's easy to recognize which methods should better be run in parallel: Any methods named loadSomething() are making a call to the server.

The BellaDatiService interface offers several shortcut methods that can give you direct access to an object when you know its ID. For example, you can load a report without first getting the report list, or a view without first loading its report.

You can save a user's active session for later use by serializing your BellaDatiService instance and storing it in your application. When the user wants to continue working with your application, you can restore the instance and the user doesn't need to authenticate again until the access token used by the SDK expires.