Developer Tools

Contents

Charts SDK

belladati-charts.js is a javascript library for rendering charts based on JSON input generated by BellaDati. Rendering is implemented using **raphael.js**.

BellaDati Charts SDK is available on GitHub: https://github.com/BellaDati/belladati-charts

You can get the charts JSON using the REST API endpoint GET Chart.

- Integrate to HTML page
- Integrate to Java application
- Line Chart
- Pie Chart

Integrate to HTML page

BellaDati charts.js library allows you to render BellaDati JSON charts data directly on your web page.

✓ You can find this example on GitHub https://github.com/BellaDati/belladati-charts/tree/master/example

Prerequisites

BellaDati-charts.js requires following javascript libraries:

- jquery 1.7.1+
- raphael.js 2.1.1+

Step-by-step

1. Link required libraries in your page

2. Get or define the charts data in JSON format
3. Define the container, where the chart should be rendered

4. Render the chart by executing *Charts.createAndRender("container_id", "json_chart_data");*

Example

And here it is all-together. Click to download a sample HTML file.

Expand source

Integrate to Java application

BellaDati charts.js library allows you to render BellaDati JSON charts data directly on web page. This web page can be rendered in your Java application.

You can find this example on GitHub https://github.com/BellaDati/belladati-charts-java-demo

Prerequisites

During this tutorial you will need following:

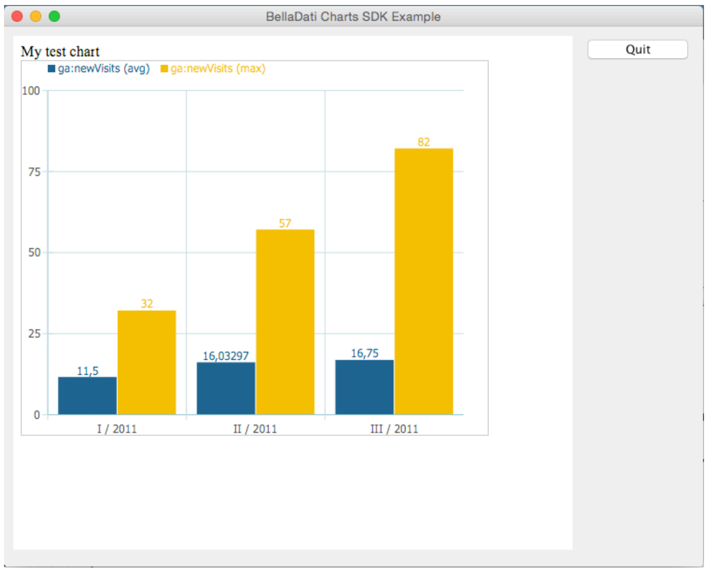
- JDK 8
- IDE to implement Java standalone application
- jquery 1.7.1+
- raphael.js 2.1.1+
- belladati-charts.js

Step-by-step

1. Create new Java project in your IDE and setup build configuration to use JDK 8.
2. Create HTML file (e.g. *index.html*) in *src/main/resources*. This HTML page will be rendered in Java application. Example:
3. Download 3 required JavaScript libraries to *src/main/resources* (to the same folder where *index.html* is located). Add relative links to these libraries in HTML head:
4. Define the container in HTML body, where the chart should be rendered:
5. Create class used as main UI window. You can use Swing for basic UI components and JavaFX WebEngine to render HTML code:

   Expand source

6. Create main class to run standalone Java application. Create MainWindow, load URL with *index.html* and render the chart by executing JavaScript *Charts.createAndRender("container_id", "json_chart_data")*:
7. Run it



Example

You can download standalone application as JAR file and run it:

```
java -jar belladati-charts-java-demo.jar
```

Line Chart

Line chart

JSON used as input for Line chart rendering consists of multiple elements.

- Basic structure
- Common elements
- "x_axis" content
- "y_axis" content
- "elements" content
- Examples

Basic structure

Common elements

Name	Description
`chartId`	Chart view identifier. Refer to REST API.
`bg_color`	Defines the color of the background for the entire chart
`is_decimal_separator_comma`	Determines whether to use comma as decimal separator. [0, 1]
`is_thousand_separator_disabled`	Determines whether to use the thousands separator or not. [0, 1]
`tooltip`	Defines the basic tooltip settings for the entire chart. **Name** | **Description** `mouse` | Mouse sensitivity `shadow` | [true, false] `stroke` | [0, 1] `colour` | Default font color for tooltips `background` | Default background for tooltips.

Example

"x_axis" content

Name	Description
`colour`	Color of the axis label

`grid-colour`	Color of the x-axis grid (horizontal)
`steps`	Number of the elements displayed on the x-axis
`labels`	Nested object defining the axis labels

Name	Description	Example
`colour`	Color of the labels	`#4e4e4e`
`labels`	Array of labels to display.	`[ "2009", "2010", "2011", "2012" ]`

Example:

"y_axis" content

Y axis is defined by *min* and *max* value, size of one step (*steps*) and *labels*. Each label consists of value on Y axis (*y*) and displayed *text*. You can specify also *colour* and *grid-colour* for Y axis.

Name	Description
`colour`	Color of the axis label
`grid-colour`	Color of the y-axis grid (vertical)
`steps`	Size of the steps of the axis
`min`	Minimal value displayed on the axis
`max`	Maximal value displayed on the axis
`labels`	Nested object defining the axis labels

Name	Description	Example
`colour`	Color of the labels	`#4e4e4e`
`labels`	Array of label and values	`[ { "text": "12.500 km", "y": 12500 }, { "text": "15.000 km", "y`

Example:

"elements" content

Each object in array *elements* describe one chart element. In example above we have only one element with *title* Grade, that has *type* line and contains array with 4 *values*. Each value object consists of *label, tool tip, value, context. Colour* and *dot-style* is specified at the end of this line element.

Name	Description
`type`	Type of the chart. For line chart, the value is "line"
`text`	Name of the chart
`font-size`	Font size used for displaying text
`colour`	Color used for displayed text
`dot-style`	Style of the dot

Name	Description	Example
`dot-size`		6
`halo-size`		10
`type`		star

Name	Description
`values`	

Name	Description	Exampl
`value`	Numerical value to display	12345.4
`tip`	Tooltip text to display on mouse hover	This is t
`label`	Value label displayed with the dot (optional)	This is l
`context`	If defined, context will be send to the client as event parameter	

Examples

File	Modified
ChartLINE.json	Oct 09, 2015 by Lubomir Micko
ChartLINENoTime.json	Oct 09, 2015 by Lubomir Micko
ChartLINEwithDrilldown.json	Oct 09, 2015 by Lubomir Micko
ChartLINEwithDrilldownHideAxesAndShowValues.json	Oct 09, 2015 by Lubomir Micko
ChartLINEwithDrilldownNoTime.json	Oct 09, 2015 by Lubomir Micko
ChartLINEwithDrilldownShowAllIndicatorsCustom.json	Oct 09, 2015 by Lubomir Micko

ChartLINEwithDrilldownShowAllIndicatorsCustomNoData.json	Oct 09, 2015 by Lubomir Micko

Download All

Pie Chart

Pie chart

JSON used as input for Pie chart rendering consists of multiple elements.

Basic structure

Common elements

Name	Description
`chartId`	Chart view identifier. Refer to REST API.
`bg_color`	Defines the color of the background for the entire chart
`is_decimal_separator_comma`	Determines whether to use comma as decimal separator. [0, 1]
`is_thousand_separator_disabled`	Determines whether to use the thousands separator or not. [0, 1]
`tooltip`	Defines the basic tooltip settings for the entire chart.

Name	Description
`mouse`	Mouse sensitivity
`shadow`	[true, false]
`stroke`	[0, 1]
`colour`	Default font color for tooltips
`background`	Default background for tooltips.

Example

"elements" content

Each object in array *elements* describe one chart element. In example above we have only one element with *type* pie, that has *start-angle* as 0 and contains array with 4 *values*. Each value object consists of *label, tool tip, value, context* and *highlight. Colours* are specified at the end of this pie chart element.

Name	Description

`type`	Type of the chart. For pie chart, the value is "pie"
`start-angle`	Starting angle for the first value. "0" means that the first value starts at 90 degrees from the top of the pie
`values`	(see table below)
`colours`	Array of colors used for each value

Name	Description	Exa
`value`	Numerical value to display	1234
`tip`	Tooltip text to display on mouse hover	This i
`label`	Value label displayed with the dot (optional)	This i
`context`	If defined, context will be send to the client as event parameter	

Examples

File	Modified
ChartPIE.json	Oct 26, 2015 by Lubomir Elko
ChartPIENoTime.json	Oct 26, 2015 by Lubomir Elko
ChartPIEShowAllIndicators.json	Oct 26, 2015 by Lubomir Elko
ChartPIEShowAllIndicatorsNoTime.json	Oct 26, 2015 by Lubomir Elko
ChartPIEwithDrilldown.json	Oct 26, 2015 by Lubomir Elko
ChartPIEwithDrilldownNoTime.json	Oct 26, 2015 by Lubomir Elko
ChartPIEwithDrilldownShowAllIndicators.json	Oct 26, 2015 by Lubomir Elko
ChartPIEwithDrilldownShowAllIndicatorsNoTime.json	Oct 26, 2015 by Lubomir Elko
ChartPIEwithDrilldownWithFilter.json	Oct 26, 2015 by Lubomir Elko

Download All

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

The SDK consists of several parts:

- API containing the SDK's interface declarations (with Javadoc online and 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 ...

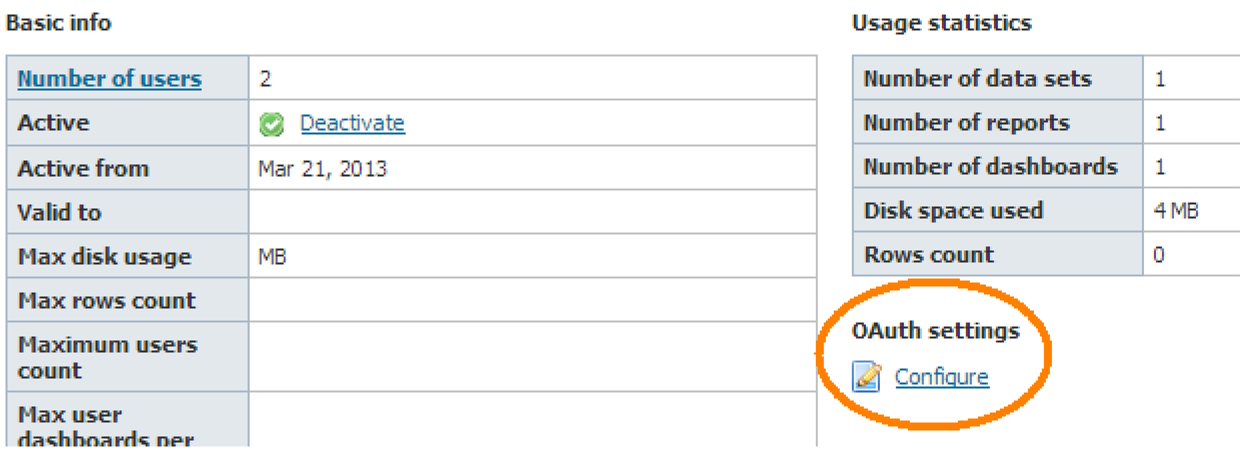
Manual Setup

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**.



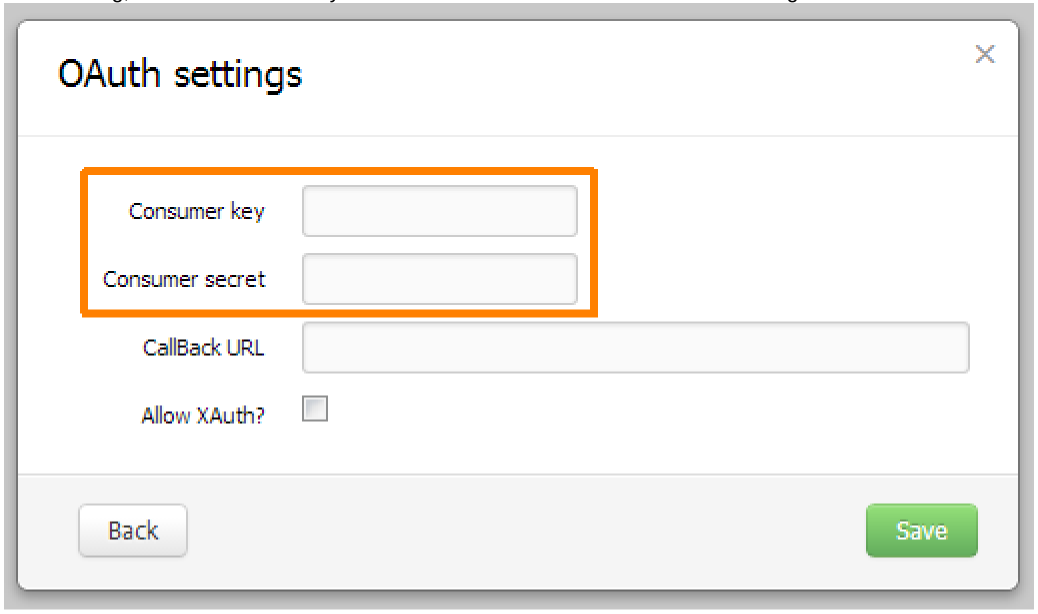
Basic info

Number of users	2
Active	✓ Deactivate
Active from	Mar 21, 2013
Valid to	
Max disk usage	MB
Max rows count	
Maximum users count	
Max user dashboards per	

Usage statistics

Number of data sets	1
Number of reports	1
Number of dashboards	1
Disk space used	4 MB
Rows count	0

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:

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 JsonNod e 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.

REST API

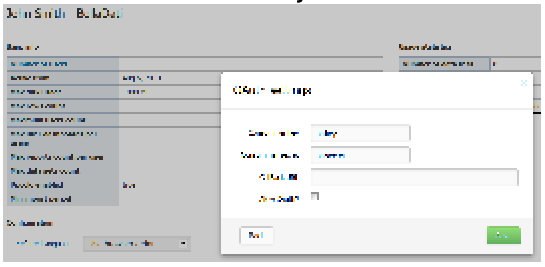
The REST API allows other applications to directly access data provided by BellaDati and is the underlying interface for our mobile Business Intelligence BellaDati Piccolo. Through the API, users can retrieve reports, charts and even entire dashboards to integrate with their own custom client application.

General Notes

Enable REST API

Before you can access the REST API, you need to enable it in your domain configuration.

1. Open your domain configuration page. To reach this page, move the mouse over your name in the upper right corner and click on the domain link.
2. Under **OAuth Settings**, click **Configure**.
3. Enter a **Consumer Key** and a **Consumer Secret**. You can ignore the other settings for now.



Base URL

The base address to access the REST API is `https://service.belladati.com/api/`.

If you are using an on-premise deployment, it is `https://your-server/belladati/api/`.

SSL Only

All API requests to BellaDati cloud must be sent over SSL.

With an on-premise deployment, although not mandatory, we strongly recommend using SSL for security reasons.

UTF-8 Encoding

Every string passed to and from the BellaDati REST API needs to be UTF-8 encoded. For maximum compatibility, normalize to Unicode Normalization Form C (NFC) before UTF-8 encoding.

Locale

BellDati REST API uses the locale parameter to specify language settings of content responses. If you want to retrieve data in a language other than English, insert the appropriate IETF language tag. When a supported language is specified, BellaDati will return translated content where applicable.

Response Format

The BellaDati REST API uses the JSON format for any responses to API calls.

Error Handling

Errors are returned using standard HTTP error code syntax. Any additional info is included in the body of the return call in JSON format. Error codes not listed here are described in the respective REST API method.

Standard API errors are:

Code	Description

400	Bad input parameter. Refer to the error message to find out which one and why.
401	Bad or expired token. This can happen if the access token has expired or is otherwise invalid. To fix this, re-authenticate the user.
403	Bad OAuth request (incorrect consumer key, bad nonce, expired time stamp...). Refer to the error message for details.
404	File or folder not found at the specified path. Check if the URL you're trying to access is correct.
405	Unexpected request method. The request method should be GET or POST depending on the request you're trying to make.
503	Your app is making too many requests and is being rate limited. 503s can trigger on a per-app or per-user basis.
5xx	Server error. Refer to the error message for details.

Access Control

BellaDati uses the OAuth protocol to ensure only authorized users may access the API.

Authentication

OAuth uses a three-step handshake to authenticate users to the system. If a 3rd-party client application is accessing the API on the user's behalf, the OAuth protocol allows users to log in directly with Belladati without having to trust the client application with their credentials.

1. Client application gets a request token from BellaDati.
2. User authorizes the request token with BellaDati using their web browser.
3. Client application exchanges authorized request token for an access token.

Authorization

When a client application is making a request to the BellaDati API, it needs to prove that it has been authorized by a valid user. To do this, it includes the access token received during the authentication process in every API request. BellaDati verifies the token's validity before returning any data to the client application.

Request Integrity

When writing data to the API, client applications may want to ensure their API requests reach the server in the exact way they were issued, e.g. because the client is running in an untrusted network environment. To prevent tampering, the client can attach an OAuth signature to their request, allowing the server to verify that the content of the message hasn't been modified.

Authentication

To access the BellaDati API, your client application needs to authenticate the user through the OAuth protocol. You can find more information about how to use OAuth in the OAuth guide.

As part of the regular OAuth protocol, the user is required to log in to BellaDati in their web browser. This way, client applications do not need to take responsibility for their users' credentials, reducing the risk of attackers being able to intercept login data. If using a web browser is not an option for your client workflow, you may instead use the xAuth variant described at the bottom of this document.

POST /oauth/requestToken - Obtaining a Request Token

Description

In the first step of the authentication process, the client application obtains a request token to use during the remaining steps. This method corresponds to Obtaining an Unauthorized Request Token in the OAuth Core 1.0 specification.

Request Structure

URL	https://service.belladati.com/oauth/requestToken
Method	`POST`
Parameters	• `oauth_consumer_key`: Your account's **consumer key**. To configure it, please visit your domain settings page. • `oauth_nonce` : A random string, uniquely generated for each request to prevent replay attacks. • `oauth_timestamp` : The current timestamp. • `oauth_callback: (optional)` Callback to redirect to after authorization is complete
Returns	A request token and the corresponding request token secret in URL encoding. This token/secret pair is later used to complete the authentication process and obtain an access token. It cannot be used for any other API calls.

Sample Request / Response

Authorizing the Request Token

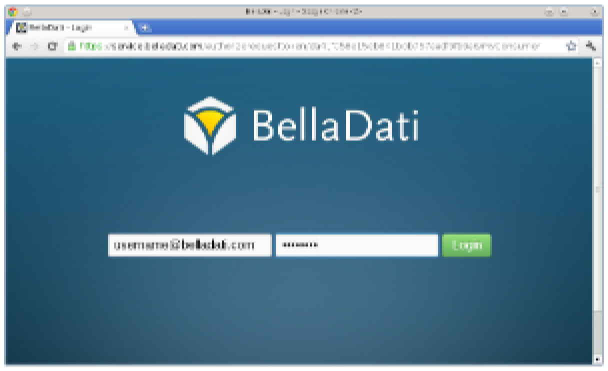
Description

After your client application has received a request token, the user needs to authorize the token for the application to gain access to the API. This is done in the user's web browser.

If your client is a web application, you can configure the **CallBack URL** on the domain settings page to point to your application. After successfully authorizing the request token, BellaDati will use this URL to redirect the user back to your application.

Request Structure

URL	https://service.belladati.com/authorizeRequestToken?oauth_token=requetToken&oauth_consumer_key=consumerKey&callbackU (open in the user's web browser)
Parameters	• `oauth_consumer_key`: Your account's **consumer key**. To configure it, please visit your domain settings page. • `oauth_token` : The request token obtained in the previous step. • `callbackUrl: (optional)` Callback to redirect to after authorization is complete
Returns	After successful login, the request token is authorized. If a **Callback URL** is was set or defined on the domain settings page, the u page.



POST /oauth/accessToken - Obtaining an Access Token

Description

This step exchanges the authorized request token for an access token. You can subsequently use the access token to access the BellaDati API. This method corresponds to Obtaining an Access Token from the OAuth Core 1.0 specification.

Please make sure that:

- The request token has been obtained using the same consumer key.
- The request token has never been exchanged for an access token before.

Request Structure

URL	https://service.belladati.com/oauth/accessToken
Method	`POST`
Parameters	• `oauth_consumer_key`: The consumer key. • `oauth_nonce` : A random string, uniquely generated for each request to prevent replay attacks. • `oauth_timestamp` : The current timestamp. • `oauth_token` : The authorized request token received in step 1 and authorized in step 2.
Returns	An access token and the corresponding access token secret in URL encoding. This token can now be used to make API calls.

Sample Request / Response

Authentication without a Web Browser

Description

If your authentication workflow doesn't allow using a web browser to log in to BellaDati, you can use the xAuth protocol variant to obtain an access token in one step. The downside is that your client application will have to deal with user credentials directly and must ensure they are handled in a secure way.

xAuth access is restricted to domains for which this feature has been **explicitly enabled**. If it's not possible to use the regular OAuth workflow with your application you can enable xAuth in your domain settings.

To use xAuth with BellaDati On-Premise, you must **enable SSL**.

Request Structure

URL	https://service.belladati.com/oauth/accessToken

Method	`POST`
Parameters	• `oauth_consumer_key`: The consumer key. • `oauth_nonce` : A random string, uniquely generated for each request to prevent replay attacks. • `oauth_timestamp` : The current timestamp. • `x_auth_username` : The user trying to authenticate. • `x_auth_password` : The user's password.
Returns	An access token and the corresponding access token secret in URL encoding. This token can now be used to make API calls.

Sample Request / Response

Authorization

When attempting to access the BellaDati API, a client application needs to prove it has been authorized by a valid user. To do this, the application supplies an access token previously obtained during Authentication.

Authorization Header

The access token is passed in the HTTP authorization header of an API request. OAuth authorization requires parameters set in the following way:

- Parameter names and values are encoded per parameter encoding (e.g. UTF-8).
- Each parameter name is immediately followed by an ‘=’ character (ASCII code 61), a ‘”’ character (ASCII code 34), the parameter value (may be empty), and another ‘”’ character (ASCII code 34).
- Parameters are separated by a comma character (ASCII code 44) and optional linear whitespace.

A valid header might look like this:

Sample Request / Response

Request Integrity

To ensure API requests cannot be modified by an attacker on the way to the server, a client application can add a digital signature to the request.

Using OAuth, signatures are created by **calculating a hash** of the **unsigned request**, the **consumer secret** corresponding to the consumer key and the **token secret** for the OAuth token used in the request. Since the consumer secret is never transmitted over the network, even an attacker who can intercept the entire communication cannot forge a request.

For security reasons, we strongly recommend using a standard implementation of the OAuth protocol to sign your application's requests.

Please refer to the OAuth specification for more details on how to create a request signature.

REST API Resources

This document lists all objects and methods available in the BellaDati REST API.

Data sets

Resource	URL	Method	Overview
GET Data Sets	/api/dataSets	GET	Lists all data sets available to the user.
GET Data Set Detail	/api/dataSets/:id	GET	Shows detailed information about the data set with the specified ID.
GET Data Sources	/api/dataSets/:id/dataSources	GET	Lists all data sources related to the data set with the specified ID.
GET Data Source Executions	/api/dataSets/dataSources/:id/executions	GET	Lists all data source executions specified by data source ID.
POST Schedule Execution	/api/dataSets/dataSources/:id/schedule	POST	Schedule execution for data source specified by ID.

Reports

Resource	URL	Method	Overview
GET Reports	/api/reports	GET	Lists all reports available to the user.
GET Report Detail	/api/reports/:id	GET	Shows detailed information about the report with the specified ID.
GET Report Thumbnail	/api/reports/:id/thumbnail	GET	Returns a report thumbnail image in PNG format.
GET Report Comments	/api/reports/:id/comments	GET	Returns all comments for the report with the specified ID.
POST Comments	/api/reports/:id/comments	POST	Posts a comment to the report with the specified ID.
DELETE Comments	/api/reports/comments/:id	DELETE	Deletes a comment specified by ID.
GET Filter Attribute Values	/api/reports/:id/filter/attributeValues	GET	Returns attribute values for provided attribute code.

Views

Resource	URL	Method	Overview
GET KPI View	/api/reports/views/:id/kpi	GET	Returns data and metadata of the specified KPI view.
GET Chart	/api/reports/views/:id/chart	GET	This method returns chart metadata (for HTML5 renderer).
GET View as Image	/api/reports/views/:id/image	GET	This method returns view (chart or table) as image in PNG format.
GET Custom Content	/api/reports/views/:id/text	GET	This method returns custom content and its metadata.
GET Table Bounds	/api/reports/views/:id/table/bounds	GET	This method returns bounds of the table.
GET Table Left Header	/api/reports/views/:id/table/leftHeader	GET	This method returns left header in form of HTML.
GET Table Top Header	/api/reports/views/:id/table/topHeader	GET	This method returns top header in form of HTML.
GET Table Data	/api/reports/views/:id/table/data	GET	This method returns table data in HTML format.
GET Table JSON	/api/reports/views/:id/table/json	GET	This method returns entire table in JSON format.

Dashboards

Resource	URL	Method	Overview
GET Dashboards	/api/dashboards	GET	This method performs the listing of all dashboards available for user.
GET Dashboard Detail	/api/dashboards/:id	GET	This method performs loading of dashboard details specified by the id parameter.
GET Dashboard Thumbnail	/api/dashboards/:id/thumbnail	GET	This method returns dashboard thumbnail represented by the first view in PNG format.

Forms

Resource	URL	Method	Overview
GET Forms	/api/import/forms	GET	This method returns list of data collection forms for signed user.
POST Form Data	/api/import/forms/:id	POST	This method posts data to the form specified by ID.

Users

Resource	URL	Method	Overview
GET Users	/api/domains/:domain_id/users	GET	List users for domain specified by domain ID.
GET User Detail	/api/users/:id	GET	Loads details about user specified by ID.
GET User Status	/api/users/:id/status	GET	Returns the active status for the user specified by ID.
GET User Photo	/api/users/:id/image	GET	Returns the image for the user specified by ID.
GET User Groups	/api/domains/:domain_id/userGroups	GET	List user groups for domain specified by domain ID.
POST Create User	/api/users/create	POST	Creates new user.
POST Edit User	/api/users/:id	POST	Modifies the user specified by ID.
POST Set User Status	/api/users/:id/status	POST	Activates/Deactivates user specified by ID.
POST Create User Group	/api/users/groups/create	POST	Creates new user group.
POST Create User Request	/api/users/:username/requests	POST	Creates new user request for user specified by username.
POST Create Access Token	/api/users/:username/accessToken	POST	Creates new access token for user specified by username.

POST Create Access Token

POST /users/{username}/accessToken

Creates an access token (oauth_token and oauth_token_secret) for user specified by username.

Resource Information

Resource URL	https://service.belladati.com/api/users/{username}/accessToken
HTTP Method	`POST`

Parameters

Parameters	Detail
validity (form paramter)	If set, specifies the validity (in seconds) of issued token.
domainId (form parameter)	Specifies the domain for which the access should be granted. This parameter is applicable for multi-domain deployments only.

Sample Request / Response

Using `curl`:

Response in format oauth_token;oauth_token_secret:

POST Create User Request

POST /users/{username}/requests

Creates an user request of desired type.

Resource Information

Resource URL	https://service.belladati.com/api/users/{username}/requests
HTTP Method	POST

Parameters

Parameters	**Detail**
username (path paramter)	Username of the user the request is created for.
request_type (form parameter)	Specifies the type of the request. Available types are: • LOGIN_UNATTENDED • PASSWORD_SET • PASSWORD_RESET • UNLOCK_ACCOUNT • LOGIN

Returns

request_id and *request_code* of the created request.

Example: 1544;RDQX1Qx9UokSf4n3KAVWgNClvrFUqncSZg7fK3gnVAfNIAOylN

Gathered *request_id* and *request_code* are consumed by the specific BellaDati service available at:

http://belladati_host/user/processRequest/{request_id}/{request_code}?redirect={redirect_url}

Sample Request / Response

Using `curl`:

Response in format oauth_token;oauth_token_secret:

Domains

Resource	URL	Method	Overview
GET Users	/api/domains/:domain_id/users	GET	List users for domain specified by domain ID.
GET Domains	/api/domains	GET	List domains.
GET Domain	/api/domains/:id	GET	Returns details about domain specified by ID.
GET User Groups	/api/domains/:domain_id/userGroups	GET	List user groups for domain specified by domain ID.
POST Create Domain	/api/domains/create	POST	Creates new domain.
POST Edit Domain	/api/domains/:id	POST	Modifies the domain specified by ID.

Data sets

Data sets

Resource	URL	Method	Overview
GET Data Sets	/api/dataSets	GET	Lists all data sets available to the user.
GET Data Set Detail	/api/dataSets/:id	GET	Shows detailed information about the data set with the specified ID.
GET Data Sources	/api/dataSets/:id/dataSources	GET	Lists all data sources related to the data set with the specified ID.
GET Data Source Executions	/api/dataSets/dataSources/:id/executions	GET	Lists all data source executions specified by data source ID.
POST Schedule Execution	/api/dataSets/dataSources/:id/schedule	POST	Schedule execution for data source specified by ID.

GET Data Sets

/api/dataSets

Lists all reports available to the user.

Resource Information

Resource URL	https://service.belladati.com/api/dataSets
HTTP Method	`GET`

Parameters

Parameters	**Detail**
`filter` (optional)	If set, the result only contains data sets that contain the filter text in name. **Example**: Sales
`offset` (optional)	Specifies the page offset if pagination is necessary. **Example**: 2
`size` (optional)	Specifies the number of entries on each page if pagination is necessary. **Example**: 15

Sample Request / Response

Using `curl`:

The JSON format response:

GET Data Set Detail

/api/dataSets/:id

Shows detailed information about the data set with the specified ID.

Resource Information

Resource URL	https://service.belladati.com/api/dataSets/:id
HTTP Method	`GET`

Parameters

Parameters	Detail
`id`	ID of the data set to fetch. You can find the ID in the response to `/api/dataSets`. **Example**: 123

Sample Request / Response

Using `curl`:

The JSON format response:

GET Data Sources

/api/dataSets/:id/dataSources

List data sources for data set with the specified ID.

Resource Information

Resource URL	https://service.belladati.com/api/dataSets/:id/dataSources
HTTP Method	`GET`

Parameters

Parameters	Detail
id	ID of the data set to fetch. You can find the ID in the response to `/api/dataSets`. **Example**: 123

Sample Request / Response

Using `curl`:

The JSON format response:

GET Data Source Executions

/api/dataSets/dataSources/:id/executions

List data sources for data set with the specified ID.

Resource Information

Resource URL	https://service.belladati.com/api/dataSets/dataSources/:id/executions
HTTP Method	`GET`

Parameters

Parameters	Detail
id	ID of the data source to fetch. You can find the ID in the response to `/api/dataSets/:id/dataSources`. **Example**: 123

Sample Request / Response

Using `curl`:

The JSON format response:

POST Schedule Execution

/api/dataSets/dataSources/:id/schedule

Posts a data source execution schedule to the data source with the specified ID.

Resource Information

Resource URL	https://service.belladati.com/api/dataSets/dataSources/:id/schedule
HTTP Method	`POST`

Parameters

Parameters	Detail
`id`	ID of the data source for which to schedule an execution. You can find the ID in the response to `/api/dataSets/:id/dataSources`. **Example**: 123
`params`	Execution parameters in JSON format. **Example**:

Sample Request / Response

Using `curl`:

When the schedule has been posted successfully, the response has the HTTP status code 200 and the content "OK".

Reports

Reports

Resource	URL	Method	Overview
GET Reports	/api/reports	GET	Lists all reports available to the user.
GET Report Detail	/api/reports/:id	GET	Shows detailed information about the report with the specified ID.
GET Report Thumbnail	/api/reports/:id/thumbnail	GET	Returns a report thumbnail image in PNG format.
GET Report Comments	/api/reports/:id/comments	GET	Returns all comments for the report with the specified ID.
POST Comments	/api/reports/:id/comments	POST	Posts a comment to the report with the specified ID.
DELETE Comments	/api/reports/comments/:id	DELETE	Deletes a comment specified by ID.
GET Filter Attribute Values	/api/reports/:id/filter/attributeValues	GET	Returns attribute values for provided attribute code.

GET Reports

/api/reports

Lists all reports available to the user.

Resource Information

Resource URL	https://service.belladati.com/api/reports
HTTP Method	`GET`

Parameters

Parameters	Detail
`filter` (optional)	If set, the result only contains reports that contain the filter text in name, description or owner name. **Example**: Sales
`offset` (optional)	Specifies the page offset if pagination is necessary. **Example**: 2
`size` (optional)	Specifies the number of entries on each page if pagination is necessary. **Example**: 15

Sample Request / Response

Using `curl`:

The JSON format response:

GET Report Detail

/api/reports/:id

Shows detailed information about the report with the specified ID.

Resource Information

Resource URL	https://service.belladati.com/api/reports/:id
HTTP Method	`GET`

Parameters

Parameters	Detail
`id`	ID of the report to fetch. You can find the ID in the response to `/api/reports`. **Example**: 123

Sample Request / Response

Using `curl`:

The JSON format response:

GET Report Thumbnail

/api/reports/:id/thumbnail

Returns a report thumbnail image in PNG format.

Resource Information

Resource URL	https://service.belladati.com/api/reports/:id/thumbnail
HTTP Method	`GET`

Parameters

Parameters	Detail
`id`	ID of the report for which to fetch a thumbnail. You can find the ID in the response to `/api/reports`. **Example**: 123

Sample Request / Response

Using `curl`:

The response is binary image data in PNG format.

GET Report Comments

/api/reports/:id/comments

Returns all comments for the report with the specified ID.

Resource Information

Resource URL	https://service.belladati.com/api/reports/:id/comments
HTTP Method	`GET`

Parameters

Parameters	Detail
`id`	ID of the report whose comments to fetch. You can find the ID in the response to `/api/reports`. **Example**: 123
`offset` (optional)	Specifies the page offset if pagination is necessary. **Example**: 2
`size` (optional)	Specifies the number of entries on each page if pagination is necessary. **Example**: 15

Sample Request / Response

Using `curl`:

The JSON format response:

POST Comments

/api/reports/:id/comments

Posts a comment to the report with the specified ID.

Resource Information

Resource URL	https://service.belladati.com/api/reports/:id/comments
HTTP Method	POST

Parameters

Parameters	Detail
`id`	ID of the report for which to post a comment. You can find the ID in the response to `/api/reports`. **Example**: 123
`text`	Text of the comment to post. **Example**: Nice report!

Sample Request / Response

Using `curl`:

When the comment has been posted successfully, the response has the HTTP status code 200 and the content *comment ID*.

DELETE Comments

/api/reports/comments/:id

Delete comment specified by ID.

Resource Information

Resource URL	https://service.belladati.com/api/reports/comments/:id
HTTP Method	`DELETE`

Parameters

Parameters	Detail
`id`	ID of the comment to delete. **Example**: 123

Sample Request / Response

Using `curl`:

When the comment has been deleted successfully, the response has the HTTP status code 200 and the content *OK*.

GET Filter attribute values

/api/reports/:id/filter/drilldownAttributeValues

Lists members for provided attribute.

Resource Information

Resource URL	https://service.belladati.com/api/reports/:id/filter/drilldownAttributeValues
HTTP Method	`GET`

Parameters

Parameters	Detail
`id`	ID of the report to fetch. You can find the ID in the response to `/api/reports`. **Example**: 123
`filter` (optional)	If set, the result only contains attribute values that contain the filter text. **Example**: Sales
`code`	The code of the attribute **Example**: L_CITY
`selectedValues` (optional)	List of already selected values **Example**: Prague, London, Seoul, Chicago

Sample Request / Response

Using `curl`:

The JSON format response:

Views

Views

Resource	URL	Method	Overview
GET KPI View	/api/reports/views/:id/kpi	GET	This method returns data and metadata of specified KPI view.
GET Chart	/api/reports/views/:id/chart	GET	This method returns chart metadata (for HTML5 renderer).
GET View as Image	/api/reports/views/:id/text/chart	GET	This method returns view (chart or table) as image in PNG format.
GET Custom Content	/api/reports/views/:id/text	GET	This method returns custom content and its metadata.
GET Table Bounds	/api/reports/views/:id/table/bounds	GET	This method returns bounds of the table.
GET Table Left Header	/api/reports/views/:id/table/leftHeader	GET	This method returns left header in form of HTML.
GET Table Top Header	/api/reports/views/:id/table/topHeader	GET	This method returns top header in form of HTML.
GET Table Data	/api/reports/views/:id/table/data	GET	This method returns table data in HTML format.
GET Table JSON	/api/reports/views/:id/table/json	GET	This method returns entire table in JSON format.
GET Map	/api/reports/views/:id/map	GET	This method returns data for rendering a map in JSON format.

GET Chart

/api/reports/views/:id/chart

Returns chart metadata for rendering.

✓ Charts can be rendered using BellaDati Charts SDK.

Resource Information

Resource URL	https://service.belladati.com/api/reports/views/:id/chart
HTTP Method	`GET`

Parameters

Parameters	Detail
`id`	ID of the view. You can find the ID in the response to `/api/reports/:id`. **Example**: 123
`filter`	Filter used in the chart. **Example**:
`dateTimeDefinition`	Date time definition used in the chart. **Example**:

Sample Request / Response

Using `curl`:

The JSON format response:

GET KPI View

/api/reports/views/:id/kpi

This method returns data and metadata of specified KPI view.

Resource Information

Resource URL	https://service.belladati.com/api/reports/views/:id/kpi
HTTP Method	`GET`

Parameters

Parameters	Detail
`id`	ID of the KPI view. This ID is a part of the `/api/reports/:id` response. **Example**: 1234
`filter`	Filter used in the KPI.
`dateTimeDefinition`	Date time definition used in the KPI.

Sample request/response

Using `curl`:

The JSON format response:

GET View as Image

/api/reports/views/:id/text

This method returns view (chart or table) as image in PNG format.

Resource Information

Resource URL	https://service.belladati.com/api/reports/views/:id/image
HTTP Method	`GET`

Parameters

Parameters	Detail
`id`	ID of the view. This ID is part of the `/api/reports/:id` response. **Example**: 123
`width`	Width of the image in pixels. **Example**: 600
`height`	Height of the image in pixels. **Example**: 400

Sample request/response

GET Custom Content

/api/reports/views/:id/text

This method return custom content and its metadata.

Resource Information

Resource URL	https://service.belladati.com/api/reports/views/:id/text
HTTP Method	`GET`

Parameters

Parameters	Detail
`id`	ID of the view. This ID is part of the `/api/reports/:id` response. **Example**: 123

Sample request/response

GET Table Bounds

/api/reports/views/:id/table/bounds

This method returns bounds of the table.

Resource Information

Resource URL	https://service.belladati.com/api/reports/views/:id/table/bounds
HTTP Method	`GET`

Parameters

Parameters	Detail
`id`	ID of the table. This ID is part of the `/api/reports/:id` response. **Example**: 123
`filter`	Filter applied to the table. **Example**:

Sample request/response

GET Table Left Header

/api/reports/views/:id/table/leftHeader

This method returns left header in form of HTML.

Resource Information

Resource URL	https://service.belladati.com/api/reports/views/:id/table/leftHeader
HTTP Method	`GET`

Parameters

Parameters	Detail
`id`	ID of the table. This ID is part of the `/api/reports/:id` response. **Example**: 123
`filter`	Filter applied to the table. **Example**:
`rowsFrom`	Rows from. **Example**:
`rowsTo`	Rows to. **Example**:

Sample request/response

GET Table Top Header

/api/reports/views/:id/table/topHeader

This method returns top header in form of HTML.

Resource Information

Resource URL	https://service.belladati.com/api/reports/views/:id/table/topHeader
HTTP Method	`GET`

Parameters

Parameters	Detail
`id`	ID of the table. This ID is part of the `/api/reports/:id` response. **Example**: 123
`filter`	Filter applied to the table. **Example**:
`columnsFrom`	Rows from. **Example**:
`columnsTo`	Rows to. **Example**:

Sample request/response

GET Table Data

/api/reports/views/:id/table/data

This method returns table data in HTML format.

Resource Information

Resource URL	https://service.belladati.com/api/reports/views/:id/table/data
HTTP Method	`GET`

Parameters

Parameters	Detail
`id`	ID of the table. This ID is part of the `/api/reports/:id` response. **Example**: 123
`filter`	Filter applied to the table. **Example**:
`rowsFrom`	Rows from. **Example**: 0
`rowsTo`	Rows to. **Example**: 10
`columnsFrom`	Columns from. **Example**: 0
`columnsTo`	Columns to. **Example**: 20

Sample request/response

GET Table JSON

/api/reports/views/:id/table/json

This method returns table in JSON format.

Resource Information

Resource URL	https://service.belladati.com/api/reports/views/:id/table/json
HTTP Method	`GET`

Parameters

Parameters	**Detail**
`id`	ID of the table. This ID is part of the `/api/reports/:id` response. **Example**: 123
`filter`	Filter applied to the table. **Example**:

Sample request/response

GET Map

/api/reports/views/:id/map

This method returns data and metadata of specified map.

Resource Information

Resource URL	https://service.belladati.com/api/reports/views/:id/map
HTTP Method	`GET`

Parameters

Parameters	Detail
`id`	ID of the map view. This ID is a part of the `/api/reports/:id` response. **Example**: 1234
`filter`	Filter used in the KPI.
`dateTimeDefinition`	Date time definition used in the KPI.

Sample request/response

Using `curl`:

The JSON format response (points):

Dashboards

Dashboards

Resource	URL	Method	Overview
GET Dashboards	/api/dashboards	GET	Lists all dashboards available to the user.
GET Dashboard Detail	/api/dashboards/:id	GET	Shows detailed information about the dashboard with the specified ID.
GET Dashboard Thumbnail	/api/dashboards/:id/thumbnail	GET	Returns a dashboard thumbnail image in PNG format, displaying the first view in the dashboard.

GET Dashboards

/api/dashboards

Lists all dashboards available to the user.

Resource Information

Resource URL	https://service.belladati.com/api/dashboards
HTTP Method	`GET`

Parameters

Parameters	Detail
`filter` (optional)	If set, the result only contains dashboards that contain the filter text in their name. **Example**: Sales
`offset` (optional)	Specifies the page offset if pagination is necessary. **Example**: 2
`size` (optional)	Specifies the number of entries on each page if pagination is necessary. **Example**: 15

Sample Request / Response

Using `curl`:

The JSON format response:

GET Dashboard Detail

/api/dashboards/:id

Shows detailed information about the dashboard with the specified ID.

Resource Information

Resource URL	https://service.belladati.com/api/dashboards/:id
HTTP Method	`GET`

Parameters

Parameters	Detail
`id`	ID of the dashboard to fetch. You can find the ID in the response to `/api/dashboards`. **Example**: 123

Sample Request / Response

Using `curl`:

The JSON format response:

GET Dashboard Thumbnail

/api/dashboards/:id/thumbnail

Returns a dashboard thumbnail image in PNG format, displaying the first view in the dashboard.

Resource Information

Resource URL	https://service.belladati.com/api/reports/:id/thumbnail
HTTP Method	`GET`

Parameters

Parameters	Detail
`id`	ID of the dashboard for which to fetch a thumbnail. You can find the ID in the response to `/api/dashboards`. **Example**:123

Sample Request / Response

Using `curl`:

The response is binary image data in PNG format.

Import

Import

Resource	URL	Method	Overview
GET Forms	/api/import/forms	GET	Lists all data import forms available to the user.
POST Form Data	/api/import/forms/:id	POST	Posts data to BellaDati using the form with the specified ID.

POST JSON data

/api/import/:id

Posts data to BellaDati in JSON format.

Resource Information

Resource URL	https://service.belladati.com/api/import/:id
HTTP Method	`POST`

Parameters

Parameters	Detail
`id`	ID of the data set. You can find the ID in the response to `/api/dataSets` **Example**: 10
`data`	Data content to submit, in JSON format. For each field, the key is the ID of the form element to save as the given value. **JSON structure**:

Sample Request / Response

Using `curl`:

When the form has been submitted successfully, the server replies with an HTTP status code of 200 and an empty body.

GET Forms

/api/import/forms

Lists all data import forms available to the user.

Resource Information

Resource URL	https://service.belladati.com/api/import/forms
HTTP Method	`GET`

Parameters

None.

Sample Request / Response

Using `curl`:

The JSON format response:

POST Form Data

/api/import/forms/:id

Posts data to BellaDati using the form with the specified ID.

Resource Information

Resource URL	https://service.belladati.com/api/import/forms/:id
HTTP Method	`POST`

Parameters

Parameters	Detail
`id`	ID of the form to fill. You can find the ID in the response to `/api/import/forms`. **Example**: 10
`data`	Form contents to submit, in JSON format. For each field, the key is the ID of the form element to save as the given value. **Example**: { "rN5hVdAXBJ" : "John" , "mNdZBe1rMY" : 1000 }

Sample Request / Response

Using `curl`:

When the form has been submitted successfully, the server replies with an HTTP status code of 200 and an empty body.

General

/api

Returns the current BellaDati version and build ID.

Resource Information

Resource URL	https://service.belladati.com/api
HTTP Method	`GET`

Parameters

None.

Sample Request / Response

Using `curl`:

Response:

Setup Data Model using XML

You can define the data model either via GUI during the import or in more advanced way using the XML file. The XML definition is submitted to BellaDati's REST API or uploaded via GUI.

Basic concept

XML definition can be applied to objects, the data model consists of:

- **dataset** - dataset is a virtual cube consisting of attributes and indicators
- **attributes** - attribute is descriptor of indicator typically strings like ID, name, city etc.
- **attribute members** - represents the single value of the attribute - e.g. New York, London, Berlin for attribute City.
- **indicators** - is a data column containing computational data - prices, amounts etc.

XML configuration file structure

As described above, data model contains of data set definition including attributes and indicators. Each data set can be also connected to external data source. Following example illustrates how to do it:

Content of the XML configuration file corresponds with the XSD schema belladati_1_0.xsd.

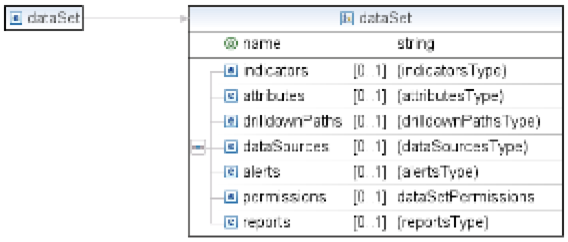
Creating data set

Data set is defined by `<dataSet>` tag.

Example

Tag attribures

name	type	required	description
`name`	string	true	Name of the data set


dataSet
dataSet
name string
indicators [0..1] (indicatorsType)
attributes [0..1] (attributesType)
drilldownPaths [0..1] (drilldownPathsType)
dataSources [0..1] (dataSourcesType)
alerts [0..1] (alertsType)
permissions [0..1] dataSetPermissions
reports [0..1] (reportsType)

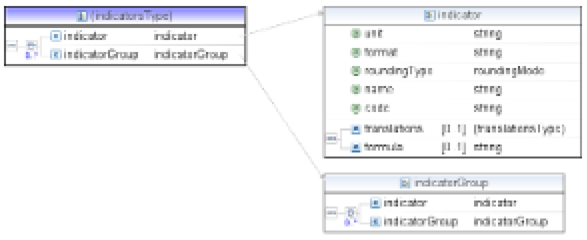
Nested tags

name	type	cardinality	description
`indicators`	complex	0..1	Encloses the indicators and indicator groups
`attributes`	complex	0..1	Encloses the attributes
`drilldownPaths`	complex	0..1	Encloses the possible drilldown paths

`dataSources`	complex	0..1	Defines the data source
`alers`	complex	0..1	Encloses the alerts
`permissions`	complex	0..1	Encloses the permissions
`reports`	complex	0..1	Encloses the reports definitions

Creating indicators and indicators groups

Indicator is represented by single object or is a part of a group of indicators. Indicators and groups are enclosed by `<indicators>` tag.



Creating indicator

Single indicator is represented by `<indicator>` tag.

Example

Tag attribures

name	type	required	description
`code`	string	true	Indicator's code. Must start with "M_" prefix and is unique within the dataset.
`name`	string	true	Name of the indicator
`unit`	string	false	Unit of the indicator
`roundingType`	string	false	Rounding mode - see http://download.oracle.com/javase/6/docs/api/java/math/RoundingMode.html for more information
`format`	string	false	Defines how the value of the indicator should be formatted

Nested tags

`translations`	complex	0..1	Encloses the indicator's translation
`formula`	CDATA value	0..1	Defines the formula, if the indicator's value is evaluated

Creating indicator with formula

Example

Creating indicator's translation

Translation for indicator is supported by nested tag `<translation>`. Translations are compliant with indicators and indicator groups as well.

Example

Tag attribures

name	type	required	description
`locale`	string	true	Locale of the translation
`value`	string	true	Translated value

Creating indicator group

If you want more structured and hierarchical organized indicators, you can define it within the indicator group. Each indicator group consists of single indicators or nested indicators groups. Indicator group definition is enclosed in tag `<indicatorGroup>`.
Example

Tag attribures

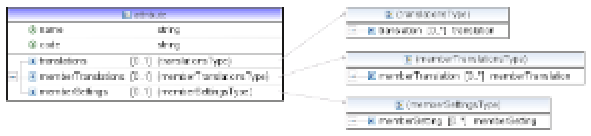
name	type	required	description
`code`	string	true	Indicator's code. Must start with "M_" prefix and is unique within the dataset.
`name`	string	true	Name of the indicator
`unit`	string	false	Unit of the indicator
`roundingType`	string	false	Rounding mode - see http://download.oracle.com/javase/6/docs/api/java/math/RoundingMode.html for more information
`format`	string	false	Defines how the value of the indicator should be formatted

Nested tags

`indicator`	complex	0..1	Nested indicator
`indicatorGroup`	complex	0..1	Nested indicator group

Creating attributes

Attributes entries `<attribute>` are defined within the `<attributes>` tag.


attribute
name string
code string
translations (0..1) (translationsType)
memberTranslations (0..1) (memberTranslationsType)
memberSettings (0..1) (memberSettingsType)
(translationsType)
translation (0..*) translation
(memberTranslationsType)
memberTranslation (0..*) memberTranslation
(memberSettingsType)
memberSetting (0..*) memberSetting

Example

Tag attribures

name	type	required	description
`code`	string	true	Attribute's code. Must start with "L_" prefix and is unique within the dataset.
`name`	string	true	Name of the attribute

Nested tags

`translations`	complex	0..1	Encloses the attribute's translation (see indicator's translation)

`memberTranslations`	complex	0..1	Encloses the attribute member's translation
`memberSettings`	complex	0..1	Holds the attribute member's settings, like appearance

Creating attribute's translation

Translation for attributes is supported by nested tag `<translation>`.

Example

Tag attribures

name	type	required	description
`locale`	string	true	Locale of the translation
`value`	string	true	Translated value

Creating member's translation

Example

Defining attribute's appearance

Example

Defining drilldown paths

Example

Transformation scripting

The transformation scripting engine's base concept is based on the Groovy scripting language.

Transformation scripting is an advanced feature for processing data during import. Thanks to transformation scripts you can:

- **cleanup** values
- **change** values
- **create new columns** with values computed from other column(s)
- mark values with flags and notes
- build intelligence for handling values according to column names

You are allowed to enter transformation scripts in the column detail dialog while setting or editting import settings.

Transformation scripts syntax resembles the Groovy and Java syntax. It is designed to be readable and effective but not strict. Users familiar with scripting and programming should know, that some of the common features (like loops) are absent because of security and performance reasons. Script can be applied to and import column. The basic function is *transforming* the input value of a column row to *another* value.

Transformation scripts basics

- You can use **variables** in scripts for storing values and working with them
- You can use **conditions** and **branching**
- You can use wide pallete of **functions**

Our first script using variable and function:

This script trims whitespaces from the beginning and end of the column value of each data import cell. You can see, that the last line contains command *return.* The function of the script can be described as "pick a value from the current cell, apply function trim and return the result".

How to access the column values?

Accessing the value we want to process is a key issue. Scripts provide a function value() which returns the current value. There are more advanced possibilities to access values:

`Object value()`	returns the value (String, Number, LocalDate, LocalTime - according to column type) of current column.
`Object value(int columnIndex)`	returns the value (String, Number, LocalDate, LocalTime - according to column type) of column at `columnIndex`
`Object value(String indicatorCode)`	returns the value (String, Number, LocalDate, LocalTime - according to column type) of column specified by `indicatorCode. Applicable in existing data sets only.`
`Object value(String attributeCode)`	returns the value (String, Number, LocalDate, LocalTime - according to column type) of column specified by `attributeCode. Applicable in existing data sets only.`
`LocalDate dateValue()`	returns the value as LocalDate of current column.
`LocalDate dateValue(int columnIndex)`	returns the value as LocalDate of column at `columnIndex`
`LocalDate dateValue(String attributeCode)`	returns the LocalDate value of column specified by `attributeCode. Applicable in existing data sets only.`
`LocalTime timeValue()`	returns the value as LocalTime of current column.
`LocalTime timeValue(int columnIndex)`	returns the value as LocalDate of current column at `columnIndex.`

`LocalTime timeValue(String attributeCode)`	returns the LocalTime value of column specified by `attributeCode. Applicable in existing data sets only.`
`Integer integerValue()`	returns the value as Integer of current column.
`Integer integerValue(int columnIndex)`	returns the value as Integer of current column at `columnIndex.`
`Integer integerValue(String code)`	returns the Integer value of column specified by `attributeCode. Applicable in existing data sets only.`
`Double doubleValue()`	returns the value as Double of current column.
`Double doubleValue(int columnIndex)`	returns the value as Double of current column at `columnIndex.`
`Double doubleValue(String code)`	returns the Double value of column specified by `attributeCode. Applicable in existing data sets only.`
`String name()`	returns the name of the current column
`String name(int columntIndex)`	returns the name of the column a `colimnIndex`
`String format()`	returns format of the current column
`String defaultValue()`	returns default value of the current column, which is the value in "Replace empty cells with value"
String getFileName()	returns file name of imported file
String getPath()	returns path to the imported file. Useful for zip archives, ftp folders or http connections.

Variables

You can store values into variables and use them into expressions or as a script result. The typing of variables is **dynamic** thus in most of the cases there is no need to think of a variable type.

Example of declaring universal variables which are dynamically typed. You can assign values to variables for later use:

You can retype a variable or function return by adding *as nameOfType*:

Type name	
`boolean`	stores boolean value True / False, used usually as a result of functions in branching expressions
`String`	stores a string of characters
`int`	stores an integer number
`double`	stores an decimal number
`Date`	represents an Date object in gregorian format
`LocalDate`	represents the Date part of the date for using in date and time functions
`LocalTime`	represents the Time part of the date for using in date and time functions

`DateTime`	represents the DateTime object for using in date and time functions

The values function returns a String value. String cannot be handled as a number unless it is properly converted. The conversion usually consists of **normalization** (ex.: round all numbers to 2 decimal places) or **cleanup** (ex.: replace all "," with ".").

More scripting language basics

- Math, formulas, expressions
- Conditions and branching
- Work with lists and maps

Transformation scripts topics

- Date and time
- String cleaning and normalization functions
- String manipulation

Advanced script tools

- Transformations over multiple columns

Conditions and branching

You can use *if* and *case* commands for implementing conditions and branching in transformation scripts.

If command

The if command is the most basic of all the control flow statements. It tells script to execute a certain section of code only if a particular test (condition) evaluates to true. Condition can be any expression containing boolean, integer and strings logic. Integer expression is evaluated to true when greather than 0 and string expression is evaluated to true when returns non blank value. **It is recommended to use boolean expressions to maintain script readability.**
Example of basic if command usege:

The *result* value is set to "Too high" only if source value of the row is greater than 100. Otherwise the result in row after transformation is "Success".
You can use multiple commands after if command when you use brackets:

If comman can contain an *else* part to provide a secondary path of execution when an "if" clause evaluates to false. We can rewrite our first example as:

In case of simple conditions you can also use a "?, :" *if* notation for branching single commands:

Switch command

Unlike if and if else commands, the switch statement allows for any number of possible execution paths. You can use more sophisticated conditions. Following example ilustrates how to evaluate value *x* three different ways:

1. Equals a specified string value
2. Is one of the values from list
3. Is number whithin an range

Another point of interest is the break statement after each case. Each break statement terminates the enclosing switch statement. Control flow continues with the first statement following the switch block. The break statements are necessary because without them, case statements fall through; that is, without an explicit break, control will flow sequentially through subsequent case statements.

Technically, the final break is not required because flow would fall out of the switch statement anyway. However, we recommend using a break so that modifying the code is easier and less error-prone. The default section handles all values that aren't explicitly handled by one of the case sections.

Deciding whether to use if command or a switch statement is sometimes a judgment call. You can decide which one to use based on readability and other factors. If you have more than 2 ways branching, use *switch* command.
You can naturally **nest** the if and switch commands.

Samples

Return text according to value

Returns a negative or positive value of a column 5 according to text value in column 6

For details about accessing other columns in script, see [Transformations over multiple columns].

Date and time

Working with date and time objects

Before you can use advanced functions, the date time information must be converted to one of the following objects:

- **`LocalDate`** - represents the date without time
- **`LocalTime`** - represents time only
- **`DateTime`** - represents date and time including the time zone

Function	Description
`LocalDate date(String value)`	Converts the passed **`value`** to **`LocalDate`** object. The **`value`** must be in one of the following formats: **`dd.MM.yyyy`**, **`dd/MM/yyyy`**, **`yyyy-MM-dd`**. Example:
LocalDate date(Partial value)	Converts the passed **Partial `value`** to **`LocalDate`** object. The **Partial `value`** is a return value e.g. from **firstValue('L_DATE')** or **lastValue('L_DATE')** met hods. Example:
`LocalDate date(long value)`	Converts the passed timestamp **`value`** to **`LocalDate`** object. The **`value`** mu st be in valid timestamp format. Example:
`LocalDate date(long value, String timezone)`	Same as date(long value) but with timezone. Valid timezone IDs are: 1. offset, e.g. +01:00, -09:00 etc 2. ID, e.g. Europe/London
`LocalDate date(String value, String format)`	Works in the same way as **`date(String value)`**, additionally you can specify the format of the **`value`** parameter. See section Date and time patterns.
`LocalTime time(String value)`	Converts the passed **`value`** to **`LocalTime`** object. The **`value`** must be in **`HH :mm:ss`** format. Example:
LocalTime time(Partial value)	Converts the passed **Partial `value`** to **`LocalDate`** object. The **Partial `value`** is a return value e.g. from **firstValue('L_TIME')** or **lastValue('L_TIME')** metho ds. Example:

`LocalTime time(long value)`	Converts the passed timestamp **`value`** to **`LocalTime`** object. The **`value`** mu st be in **valid timestamp** format. Example:
	
	`def a = time(1396348391)`
	`def b = time(value(1))`

`LocalTime time(long value)`	Same as time(long value) but with timezone. Valid timezone IDs are: 1. offset, e.g. +01:00, -09:00 etc 2. ID, e.g. Europe/London
`LocalTime time(String value, String format)`	Works in the same way as **`time(String value)`**, additionally you can specify the format of the **`value`** parameter. See section Date and time patterns.
`DateTime datetime(String value)`	Converts the passed **`value`** to **`DateTime`** object. The **`value`** must be in one of the following formats: **`dd.MM.yyyy HH:mm:ss`**, **`dd/MM/yyyy HH:mm:ss`**, **`yyyy-MM-dd HH:mm:ss`**. Example:
DateTime datetime(Partial value)	Converts the passed **Partial `value`** to **`LocalDate`** object. The **Partial `value`** is a return value e.g. from **firstValue('L_DATETIME')** or **lastValue('L_DATE TIME')** methods. Example:
`DateTime datetime(long value)`	Converts the passed timestamp **`value`** to **`DateTime`** object. The **`value`** must be in valid timestamp Example:
`DateTime datetime(long value, String timezone)`	Same as datetime(long value) but with timezone. Valid timezone IDs are: 1. offset, e.g. +01:00, -09:00 etc 2. ID, e.g. Europe/London
`DateTime datetime(String value, String format)`	Works in the same way as **`datetime(String value)`**, additionally you can specify the format of the **`value`** parameter. See section Date and time patterns.
`long timestamp(DateTime value)`	Converts the passed DateTime value into timestamp.

After the date time has been converted into appropriate objects, you can use following functions:

Getting parts of date time

Function	Description
`Integer year(LocalDate date)`	Extracts the year of the **`LocalDate`** object. Example:
`Integer month(LocalDate date)`	Extracts the month from the **`LocalDate`** object.
`Integer week(LocalDate date)`	Extracts the week from the **`LocalDate`** object.
`Integer dayOfWeek(LocalDate date)`	Extracts the day of week from the **`LocalDate`** object.

`Integer dayOfMonth(LocalDate date)`	Extracts the day of month from the **`LocalDate`** object.
`Integer dayOfYear(LocalDate date)`	Extracts the day of year from the **`LocalDate`** object.
`Integer hourOfDay(LocalTime time)`	Extracts the hour of the day from the **`LocalTime`** object.
`Integer hourOfDay(DateTime dt)`	Extracts the hour of the day from the **`DateTime`** object.
`Integer minuteOfHour(LocalTime date)`	Extracts the minute of hour from the **`LocalTime`** object.
`Integer minuteOfHour(DateTime date)`	Extracts the minute of hour from the **`DateTime`** object.
`Integer minuteOfDay(DateTime date)`	Extracts the minute of the day from the **`DateTime`** object.
`Integer secondOfMinute(LocalTime date)`	Extracts the second of minute from the **`LocalTime`** object.
`Integer secondOfMinute(DateTime date)`	Extracts the second of minute from the **`DateTime`** object.
`Integer secondOfDay(DateTime date)`	Extracts the second of day from the **`DateTime`** object.

Comparing two instances

Basic comparison is possible by using the operators `==, =<, <, =>, >, !=`

Function	Description
`boolean isAfter(LocalTime t1, LocalTime t2)`	Determines whether the **`t1`** is after the **`t2`** instance.
`boolean isAfter(LocalDate d1, LocalDate d2)`	Determines whether the **`d1`** is after the **`d2`** instance.
`boolean isAfter(DateTime dt1, DateTime dt2)`	Determines whether the **`dt1`** is after the **`dt2`** instance.
`boolean isBefore(LocalTime t1, LocalTime t2)`	Determines whether the **`t1`** is before the **`t2`** instance.
`boolean isBefore(LocalDate d1, LocalDate d2)`	Determines whether the **`d1`** is before the **`d2`** instance.
`boolean isBefore(DateTime dt1, DateTime dt2)`	Determines whether the **`dt1`** is before the **`dt2`** instance.
`boolean isEqual(LocalTime t1, LocalTime t2)`	Determines whether the **`t1`** is equal to **`t2`** instance.
`boolean isEqual(LocalDate d1, LocalDate d2)`	Determines whether the **`d1`** is equal to **`d2`** instance.
`boolean isEqual(DateTime dt1, DateTime dt2)`	Determines whether the **`dt1`** is equal to **`dt2`** instance.
`Integer secondsBetween(LocalTime t1, LocalTime t2)`	Returns the number of seconds between the two specified **`LocalTime`** instances. Example:
`Integer secondsBetween(DateTime dt1, DateTime dt2)`	Returns the number of seconds between the two specified **`DateTime`** instances.
`Integer minutesBetween(LocalTime t1, LocalTime t2)`	Returns the number of minutes between the two specified **`LocalTime`** instances.
`Integer minutesBetween(DateTime dt1, DateTime dt2)`	Returns the number of minutes between the two specified **`DateTime`** instances.
`Integer hoursBetween(LocalTime t1, LocalTime t2)`	Returns the number of hours between the two specified **`LocalTime`** instances.
`Integer hoursBetween(DateTime dt1, DateTime dt2)`	Returns the number of hours between the two specified **`DateTime`** instances.

`Integer daysBetween(LocalDate d1, LocalDate d2)`	Returns the number of days between the two specified **`LocalDate`** inst ances. Example:
`Integer daysBetween(DateTime dt1, DateTime dt2)`	Returns the number of days between the two specified **`DateTime`** insta nces.
`Integer weeksBetween(LocalDate d1, LocalDate d2)`	Returns the number of weeks between the two specified **`LocalDate`** ins tances.
`Integer weeksBetween(DateTime dt1, DateTime dt2)`	Returns the number of weeks between the two specified **`DateTime`** inst ances.
`Integer monthsBetween(LocalDate d1, LocalDate d2)`	Returns the number of months between the two specified **`LocalDate`** in stances.
`Integer monthsBetween(DateTime dt1, DateTime dt2)`	Returns the number of months between the two specified **`DateTime`** ins tances.
`Integer yearsBetween(LocalDate d1, LocalDate d2)`	Returns the number of years between the two specified **`LocalDate`** inst ances.
`Integer yearsBetween(DateTime dt1, DateTime dt2)`	Returns the number of years between the two specified **`DateTime`** insta nces.

Date time manipulation

Manipulation with **`DateTime`**, **`LocalDate`** and **`LocalTime`** objects is supported via the `plus(...)` and `minus(...)` functions. Both functions are changing the passed object and are returning it's changed instance. The mandatory **`part`** parameter specifies, which part of the date/time value should be changed. Following parts are supported depending on the instance type:

- `LocalDate` and `DateTime` - days, weeks, months, years
- `LocalTime` and `DateTime` - hours, minutes, seconds

Function	Description
`DateTime plus(LocalTime time, String part, int count)`	Adds the amount of specified `part` to the passed `LocalTime` value.
`DateTime plus(LocalDate date, String part, int count)`	Adds the amount of specified `part` to the passed `LocalDate` value.
`DateTime plus(DateTime dt, String part, int count)`	Adds the amount of specified `part` to the passed `DateTime` value.
`DateTime minus(LocalTime time, String part, int count)`	Subtracts the amount of specified `part` from the passed `LocalTime` v alue.
`DateTime minus(LocalDate date, String part, int count)`	Subtracts the amount of specified `part` from the passed `LocalDate` v alue.
`DateTime minus(DateTime dt, String part, int count)`	Subtracts the amount of specified `part` from the passed `DateTime` val ue.

Formatting date time output

Function	Description
`String toString(DateTime dt, String format)`	Outputs the **`DateTime`** object in passed **`format`**
`String toString(LocalDate date, String format)`	Outputs the **`LocalDate`** object in passed **`format`**
`String toString(LocalTime time, String format)`	Outputs the **`LocalTime`** object in passed **`format`**

`String` based date time functions

Basic date time functions works with date time as **`Strings`**.

Function	Description
`String actualDate()`	Returns the current date in **`dd.MM.yyyy`** format. This function is suitable also for inducting an current time for periodic import from data source which has no time column.
`String actualDate(String format)`	Works like the **`actulaDate()`** function extended with custumizable output date format. See section Date and time patterns. Example:
`String actualTime()`	Returns the current time in **`HH:mm:ss`** format. This function is suitable also for inducting an current date for periodic import from data source which has no time column.
`String actualTime(String format)`	Works like the **`actulaDTime()`** function extended with custumizable output time format. See section Date and time patterns. Example:
`String datetimePart(String value, String part)`	Returns the part of the passed date or time string. This string should be in format **`yyyy-MM-dd HH:mm:ss`** or **`yyyy-MM-dd`**. The **`part`** parameters defines one of the following: **`year`**, **`monthOfYear`**, **`weekOfYear`**, **`dayOfMonth`**, **`dayOfWeek`**, **`dayOfYear`**, **`hourOfDay`**, **`minuteOfHour`**, **`secondOfMinute`**

Date and Time Patterns

Date and time formats are specified by date and time pattern strings. Within date and time pattern strings, unquoted letters from 'A' to 'Z' and from 'a' to 'z' are interpreted as pattern letters representing the components of a date or time string. Text can be quoted using single quotes (') to avoid interpretation. "''" represents a single quote. All other characters are not interpreted; they're simply copied into the output string during formatting or matched against the input string during parsing.

The following pattern letters are defined (all other characters from 'A' to 'Z' and from 'a' to 'z' are reserved):

Letter	Date or Time Component	Examples
G	Era designator	AD
y	Year	1996; 96
M	Month in year	July; Jul; 07
w	Week in year	27
W	Week in month	2
D	Day in year	189
d	Day in month	10
F	Day of week in month	2
E	Day in week	Tuesday; Tue
a	Am/pm marker	PM
H	Hour in day (0-23)	0
k	Hour in day (1-24)	4
K	Hour in am/pm (0-11)	0

h	Hour in am/pm (1-12)	12
m	Minute in hour	30
s	Second in minute	55
S	Millisecond	978
z	Time zone	Pacific Standard Time; PST; GMT-08:00
Z	Time zone RFC 822	-0800

Another way how to compare two dates

Consider following example:
We have an import, which contains two date columns and we want the day difference of these two dates. Both date columns contains date in format yyyy/MM/dd.

Looping

BellaDati doesn't support the usual `while {...}` and `for {...}` loops like Java. Instead of this, you can use closures in place of most for loops using `each()`:

Looping over splitted string

Math, formulas, expressions

⚠ Always check which decimal separator does your data contain. Other separators than dot "." must be replaced first, otherwise wrong or empty result may be calculated. See replacing options for details.

Expressions

You can use numeric variables, functions and values in regular math expressions:

Equations (returning boolean values):

You can use boolean variables, functions and values in boolean logic expressions:

Is the same as:

Boolean operators

|	the OR operator
&	the AND operator
^	the XOR operator
!	the NOT operator
||	the short-circuit OR operator
&&	the short-circuit AND operator
==	the EQUAL TO operator
!=	the NOT EQUAL TO operator

String can be concatenated together by a + operator:

Math funtions

Function	Description
`double abs(double a)`	Returns the absolute value of a double value.
`float abs(float a)`	Returns the absolute value of a float value.
`int abs(int a)`	Returns the absolute value of an int value.
`long abs(long a)`	Returns the absolute value of a long value.
`double acos(double a)`	Returns the arc cosine of a value; the returned angle is in the range 0.0 through pi.
`double asin(double a)`	Returns the arc sine of a value; the returned angle is in the range -pi/2 through pi/2.

`double atan(double a)`	Returns the arc tangent of a value; the returned angle is in the range -pi/2 through pi/2.
`double atan2(double y, double x)`	Returns the angle theta from the conversion of rectangular coordinates (x, y) to polar coordinates (r, theta).
`double cbrt(double a)`	Returns the cube root of a double value.
`double ceil(double a)`	Returns the smallest (closest to negative infinity) double value that is greater than or equal to the argument and is equal to a mathematical integer.
`double copySign(double magnitude, double sign)`	Returns the first floating-point argument with the sign of the second floating-point argument.
`float copySign(float magnitude, float sign)`	Returns the first floating-point argument with the sign of the second floating-point argument.
`double cos(double a)`	Returns the trigonometric cosine of an angle.
`double cosh(double x)`	Returns the hyperbolic cosine of a double value.
`double exp(double a)`	Returns Euler's number e raised to the power of a double value.
`double expm1(double x)`	Returns ex -1.
`double floor(double a)`	Returns the largest (closest to positive infinity) double value that is less than or equal to the argument and is equal to a mathematical integer.
`int getExponent(double d)`	Returns the unbiased exponent used in the representation of a double.
`int getExponent(float f)`	Returns the unbiased exponent used in the representation of a float.
`double hypot(double x, double y)`	Returns sqrt(x2 +y2) without intermediate overflow or underflow.
`double IEEEremainder(double f1, double f2)`	Computes the remainder operation on two arguments as prescribed by the IEEE 754 standard.
`double log(double a)`	Returns the natural logarithm (base e) of a double value.
`double log10(double a)`	Returns the base 10 logarithm of a double value.
`double log1p(double x)`	Returns the natural logarithm of the sum of the argument and 1.
`double max(double a, double b)`	Returns the greater of two double values.
`float max(float a, float b)`	Returns the greater of two float values.
`int max(int a, int b)`	Returns the greater of two int values.
`long max(long a, long b)`	Returns the greater of two long values.
`double min(double a, double b)`	Returns the smaller of two double values.
`float min(float a, float b)`	Returns the smaller of two float values.
`int min(int a, int b)`	Returns the smaller of two int values.
`long min(long a, long b)`	Returns the smaller of two long values.
`double nextAfter(double start, double dir)`	Returns the floating-point number adjacent to the first argument in the direction of the second argument.
`float nextAfter(float start, double dir)`	Returns the floating-point number adjacent to the first argument in the direction of the second argument.
`double nextUp(double d)`	Returns the floating-point value adjacent to d in the direction of positive infinity.
`float nextUp(float f)`	Returns the floating-point value adjacent to f in the direction of positive infinity.
`double pow(double a, double b)`	Returns the value of the first argument raised to the power of the second argument.
`double random()`	Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.

`double rint(double a)`	Returns the double value that is closest in value to the argument and is equal to a mathematical integer.
`long round(double a)`	Returns the closest long to the argument.
`int round(float a)`	Returns the closest int to the argument.
`double scalb(double d, int scaleFactor)`	Return d × 2scaleFactor rounded as if performed by a single correctly rounded floating-point multiply to a member of the double value set.
`float scalb(float f, int scaleFactor)`	Return f × 2scaleFactor rounded as if performed by a single correctly rounded floating-point multiply to a member of the float value set
`double signum(double d)`	Returns the signum function of the argument; zero if the argument is zero, 1.0 if the argument is greater than zero, -1.0 if the argument is less than zero.
`float signum(float f)`	Returns the signum function of the argument; zero if the argument is zero, 1.0f if the argument is greater than zero, -1.0f if the argument is less than zero.
`double sin(double a)`	Returns the trigonometric sine of an angle.
`double sinh(double x)`	Returns the hyperbolic sine of a double value.
`double sqrt(double a)`	Returns the correctly rounded positive square root of a double value.
`double tan(double a)`	Returns the trigonometric tangent of an angle.
`double tanh(double x)`	Returns the hyperbolic tangent of a double value.
`double toDegrees(double angrad)`	Converts an angle measured in radians to an approximately equivalent angle measured in degrees.
`double toRadians(double angdeg)`	Converts an angle measured in degrees to an approximately equivalent angle measured in radians.
`double ulp(double d)`	Returns the size of an ulp of the argument.
`float ulp(float f)`	Returns the size of an ulp of the argument.
`long factorial(int value)`	Returns the factorial of passed value.

Samples

Simple computation

Rounding

Notice we must normalize input to be valid number by normalizing the decimal separator.

String cleaning and normalization functions

Replacing

`replace(text, String repl, String with)`	**Replaces all occurrences of a String within another String.**
`replace(text, String repl, String with, int max)`	**Replaces a String with another String inside a larger String, for the first max values of the search String.**
`replaceOnce(text, String repl, String with)`	Replaces a String with another String inside a larger String, once.

Samples

number decimal separator to "."

Trimming and whitespaces cleaning

`strip(str, String stripChars)`	Strips any of a set of characters from the start and end of a String.
`String[] stripAll(String[] strs)`	Strips whitespace from the start and end of every String in an array.
`String[] stripAll(String[] strs, String stripChars)`	Strips any of a set of characters from the start and end of every String in an array.
`stripEnd(str, String stripChars)`	Strips any of a set of characters from the end of a String.
`stripStart(str, String stripChars)`	Strips any of a set of characters from the start of a String.
`trim(str)`	**Removes control characters (char <= 32) from both ends of this String, handling null by returning null.**
`chomp(str)`	Removes one newline from end of a String if it's there, otherwise leave it alone.
`chomp(str, String separator)`	Removes separator from the end of str if it's there, otherwise leave it alone.
`deleteWhitespace(str)`	Deletes all whitespaces from a String.

Samples

Basic trimming of all whitespaces after and before cell content:

Case manipulation

`swapCase(str)`	Swaps the case of a String changing upper and title case to lower case, and lower case to upper case.
`uncapitalize(str)`	**Uncapitalizes a String changing the first letter to title case.**
`lowerCase(str)`	**Converts a String to lower case as per String.toLowerCase().**
`upperCase(str)`	**Converts a String to upper case.**

Next steps:

- Transformation scripts use cases

String manipulation

The complete reference of string amnipulating functions

`capitalize(str)`	Capitalizes a String changing the first letter to title case as per Character.toTitleCase(char).
`center(str, int size)`	Centers a String in a larger String of size size using the space character (' ').
`center(str, int size, String padStr)`	Centers a String in a larger String of size size.
`chomp(str)`	Removes one newline from end of a String if it's there, otherwise leave it alone.
`chomp(str, String separator)`	Removes separator from the end of str if it's there, otherwise leave it alone.
`chop(str)`	Remove the last character from a String.
`boolean contains(str, String searchStr)`	Checks if String contains a search String, handling null.
`boolean containsIgnoreCase(str, String searchStr)`	Checks if String contains a search String irrespective of case, handling null.
`boolean containsNone(str, String invalidChars)`	Checks that the String does not contain certain characters.
`boolean containsOnly(str, String validChars)`	Checks if the String contains only certain characters.
`int countMatches(str, String sub)`	Counts how many times the substring appears in the larger String.
`deleteWhitespace(str)`	Deletes all whitespaces from a String.
`difference(str1, String str2)`	Compares two Strings, and returns the portion where they differ.
`boolean equals(str1, String str2)`	**Compares two Strings, returning true if they are equal.**
`boolean equalsIgnoreCase(str1, String str2)`	**Compares two Strings, returning true if they are equal ignoring the case.**
`int getLevenshteinDistance(s, String t)`	Find the Levenshtein distance between two Strings.
`int indexOf(str, String searchStr)`	**Finds the first index within a String, handling null.**
`int indexOf(str, String searchStr, int startPos)`	**Finds the first index within a String, handling null.**
`int indexOfAny(str, String searchChars)`	**Search a String to find the first index of any character in the given set of characters.**
`int indexOfAny(str, String[] searchStrs)`	**Find the first index of any of a set of potential substrings.**
`int indexOfAnyBut(str, String searchChars)`	Search a String to find the first index of any character not in the given set of characters.
`int indexOfDifference(str1, String str2)`	Compares two Strings, and returns the index at which the Strings begin to differ.
`boolean isAlpha(str)`	**Checks if the String contains only unicode letters.**
`boolean isAlphanumeric(str)`	**Checks if the String contains only unicode letters or digits.**
`boolean isAlphanumericSpace(str)`	**Checks if the String contains only unicode letters, digits or space (' ').**
`boolean isAlphaSpace(str)`	**Checks if the String contains only unicode letters and space (' ').**
`boolean isBlank(str)`	**Checks if a String is whitespace, empty ("") or null.**

`boolean isEmpty(str)`	Checks if a String is empty ("") or null.
`boolean isNotBlank(str)`	**Checks if a String is not empty (""), not null and not whitespace only.**
`boolean isNotEmpty(str)`	**Checks if a String is not empty ("") and not null.**
`boolean isNumeric(str)`	**Checks if the String contains only unicode digits.**
`boolean isNumericSpace(str)`	**Checks if the String contains only unicode digits or space (' ').**
`boolean isWhitespace(str)`	**Checks if the String contains only whitespace.**
`join(String[] array)`	Joins the elements of the provided array into a single String containing the provided list of elements.
`join(String[] array, String separator)`	Joins the elements of the provided array into a single String containing the provided list of elements.
`join(String[] array, String separator, int startIndex, int endIndex)`	Joins the elements of the provided array into a single String containing the provided list of elements.
`int lastIndexOf(str, String searchStr)`	Finds the last index within a String, handling null.
`int lastIndexOf(str, String searchStr, int startPos)`	Finds the first index within a String, handling null.
`int lastIndexOfAny(str, String[] searchStrs)`	Find the latest index of any of a set of potential substrings.
`left(str, int len)`	**Gets the leftmost len characters of a String.**
`leftPad(str, int size)`	Left pad a String with spaces (' ').
`leftPad(str, int size, String padStr)`	Left pad a String with a specified String.
`lowerCase(str)`	**Converts a String to lower case as per String.toLowerCase().**
`length(str)`	This will return the no. of characters of the string.
`mid(str, int pos, int len)`	Gets len characters from the middle of a String.
`int ordinalIndexOf(str, String searchStr, int ordinal)`	Finds the n-th index within a String, handling null.
`overlay(str, String overlay, int start, int end)`	Overlays part of a String with another String.
`remove(str, String remove)`	Removes all occurances of a substring from within the source string.
`removeEnd(str, String remove)`	Removes a substring only if it is at the end of a source string, otherwise returns the source string.
`removeStart(str, String remove)`	Removes a substring only if it is at the begining of a source string, otherwise returns the source string.
`repeat(str, int repeat)`	Repeat a String repeat times to form a new String.
`replace(text, String repl, String with)`	**Replaces all occurrences of a String within another String.**
`replace(text, String repl, String with, int max)`	**Replaces a String with another String inside a larger String, for the first max values of the search String.**
`replaceChars(str, String searchChars, String replaceChars)`	**Replaces multiple characters in a String in one go.**
`replaceOnce(text, String repl, String with)`	Replaces a String with another String inside a larger String, once.
`reverse(str)`	Reverses a String as per StringBuffer.reverse().
`right(str, int len)`	**Gets the rightmost len characters of a String.**

`rightPad(str, int size)`	Right pad a String with spaces (' ').
`rightPad(str, int size, String padStr)`	Right pad a String with a specified String.
`String[] split(str)`	**Splits the provided text into an array, using whitespace as the separator.**
`String[] split(str, String separatorChars)`	**Splits the provided text into an array, separators specified.**
`String[] split(str, String separatorChars, int max)`	**Splits the provided text into an array with a maximum length, separators specified.**
`String[] splitByWholeSeparator(str, String separator)`	Splits the provided text into an array, separator string specified.
`String[] splitByWholeSeparator(str, String separator, int max)`	Splits the provided text into an array, separator string specified.
`String[] splitPreserveAllTokens(str)`	Splits the provided text into an array, using whitespace as the separator, preserving all tokens, including empty tokens created by adjacent separators.
`String[] splitPreserveAllTokens(str, String separatorChars)`	Splits the provided text into an array, separators specified, preserving all tokens, including empty tokens created by adjacent separators.
`String[] splitPreserveAllTokens(str, String separatorChars, int max)`	Splits the provided text into an array with a maximum length, separators specified, preserving all tokens, including empty tokens created by adjacent separators.
`strip(str)`	**Strips whitespace from the start and end of a String.**
`strip(str, String stripChars)`	Strips any of a set of characters from the start and end of a String.
`String[] stripAll(String[] strs)`	Strips whitespace from the start and end of every String in an array.
`String[] stripAll(String[] strs, String stripChars)`	Strips any of a set of characters from the start and end of every String in an array.
`stripEnd(str, String stripChars)`	Strips any of a set of characters from the end of a String.
`stripStart(str, String stripChars)`	Strips any of a set of characters from the start of a String.
`substring(str, int start)`	**Gets a substring from the specified String avoiding exceptions.**
`substring(str, int start, int end)`	**Gets a substring from the specified String avoiding exceptions.**
`substringAfter(str, String separator)`	Gets the substring after the first occurrence of a separator.
`substringAfterLast(str, String separator)`	Gets the substring after the last occurrence of a separator.
`substringBefore(str, String separator)`	Gets the substring before the first occurrence of a separator.
`substringBeforeLast(str, String separator)`	Gets the substring before the last occurrence of a separator.
`substringBetween(str, String tag)`	Gets the String that is nested in between two instances of the same String.
`substringBetween(str, String open, String close)`	Gets the String that is nested in between two Strings.
`String[] substringsBetween(str, String open, String close)`	Searches a String for substrings delimited by a start and end tag, returning all matching substrings in an array.
`swapCase(str)`	Swaps the case of a String changing upper and title case to lower case, and lower case to upper case.
`trim(str)`	**Removes control characters (char <= 32) from both ends of this String, handling null by returning null.**
`uncapitalize(str)`	**Uncapitalizes a String changing the first letter to title case.**
`upperCase(str)`	**Converts a String to upper case.**

Transformations over multiple columns

From every column's script, you can access other columns by functions:
Function value(x) returns value of a x-th column of the import
Function name(x) returns name of the x-th column of the import
and include the results into the current column's script.

Samples

Separating a column containing full name into two first name and surname:

Script for first name:

Script for surname name:

Working with rows and columns

Excluding rows from import

Working with global values

Useful when you need to pass parameters over rows or columns. Following example shows how to count number of rows:

Work with lists and maps

The transformation script has built-in support for two important data types, lists and maps (Lists can be operated as arrays in Java language). Lists are used to store ordered collections of data. For example an integer list of your favorite integers might look like this:

Another native data structure is called a map. A map is used to store "associative arrays" or "dictionaries". That is unordered collections of heterogeneous, named data. For example, let's say we wanted to store names with IQ scores we might have:

To add data to a map, the syntax is similar to adding values to an list. For example, if Pete re-took the IQ test and got a 3, we might:

To get the size of the collection, you can use the size() function:

Looping is provided using the `each()` closure:

Samples

Convert bank account number to bank name

Let the column cell contains a full bank account number (eg. "54-123456789/0100") and we transform it to the bank name according to last 4 digits. The script is:

Working with XML content

ⓘ All functions with XML processing support are working with the xPath. We are using xPath as element selector.

Available functions

Function	Description
`xpathElementAttribute(String content, String xpathExpression, String attributeId)`	Returns the value of the attribute specified by *attributeId* of the element selected by *xpathExpression*.
`xpathElementAttribute(String content, String xpathExpression, String attributeId, String encoding)`	Returns the value of the attribute specified by *attributeId* of the element selected by *xpathExpression*. Content is in specified *encodin g.*
`xpathElementContent(String content, String xpathExpression)`	Returns the content of the element selected by *xpathExpression*.
`xpathElementContent(String content, String xpathExpression, String encoding)`	Returns the content of the element selected by *xpathExpression*. Content is in specified *encoding.*

Examples

Consider following XML *content*:

Results:

Use Cases

Make sure that you are familiar with Transformation scripting prior to proceeding with **Transformation Scripting Use Cases**.

Transformation scripts are very powerful and complex scripting language. You can conduct wide variety of calculations and transformations by using and combining advanced functions.

In the following sections, you can find advanced **Use cases** and detailed description of the most common tasks conducted in BellaDati simply by using **Transformation scripts**.

Current **Use cases** include:

- Change number of decimal places
- Column Splitting - Parsing Out Address
- Data cleansing
- Handling empty and non-valid date formats
- Monthly values with custom start date
- Parsing XML Content from Data Source
- Record ordering
- Sentiment Analysis

You can request help or additional use cases by contacting BellaDati analysts team at support@belladati.com

Change number of decimal places

In some cases, you might like to change number of decimal places in your data. For this case can be used following transformation script:

```
java.text.DecimalFormat df = new java.text.DecimalFormat(".00")
return df.format(value('M_VALUE'))
```

Number of decimal values is defined in definition java.text.DecimalFormat(".00"). In this particular case, number of decimal values is set to 2.

Column Splitting - Parsing Out Address

It is recommended to get familiar with functions **isEmpty()**, **substringBefore()** and **substringAfter()** before proceeding with this tutorial.

In this example you are going to learn how split columns in you data set. In this sample file we have two kind of records. First one has the address in one column and the second one has it split into city and street.

Edit Document

Address	City	Street
	TUCSON	799 E DRAGRAM SUITE 5A
200 E MAIN ST, PHOENIX		
	SEATTLE	100 MAIN ST
300 BOYLSTON AVE E, SEATTLE		
7647 DEWY PRAIRIE PASSAGE, MEXICO		

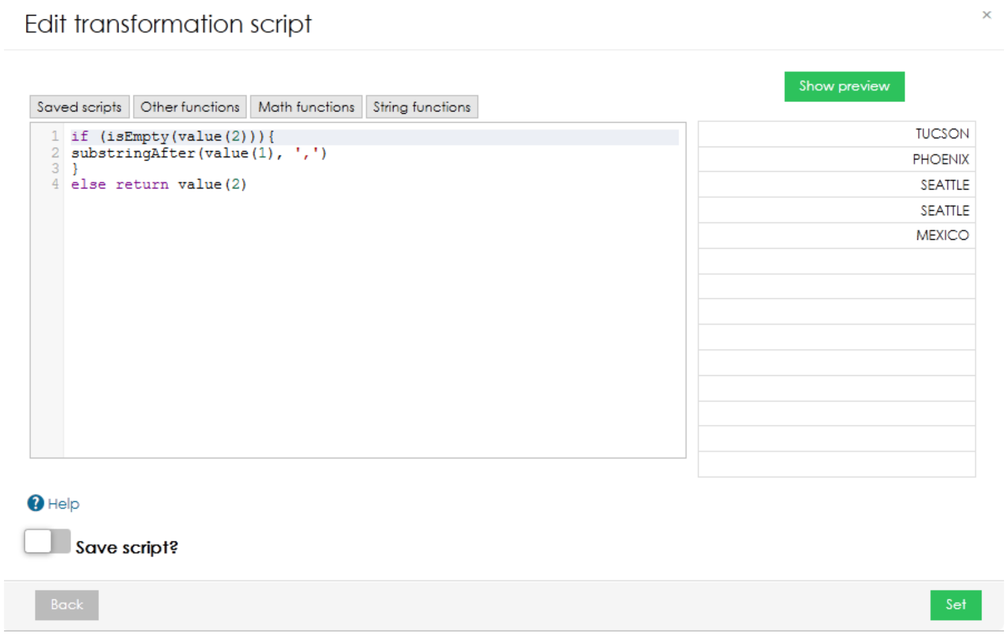
Source file: addresses.xlsx

For better analysis options it would be better to have it separated in all rows.

1. Upload sample data file into BellaDati.
2. Add transformation script to column City by hovering over column name and clicking Edit transformation script.

Extracting City Name from the Address

This script is checking whether the column City is empty. In case it is, part of first column (Address) starting after comma is used. In case it is not empty, current value is used.



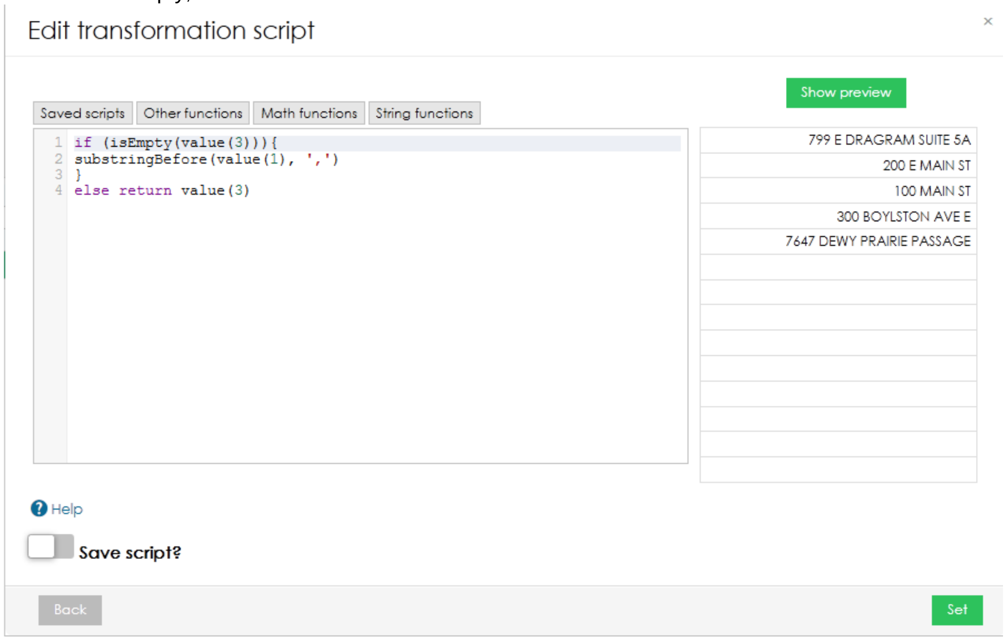
```
if (isEmpty(value(2))){
substringAfter(value(1), ',')
}
else return value(2)
```

```
TUCSON
PHOENIX
SEATTLE
SEATTLE
MEXICO
```

3. Add transformation script to column Street by hovering over column name and clicking Edit transformation script.

Extracting Street Name and Number from the Address

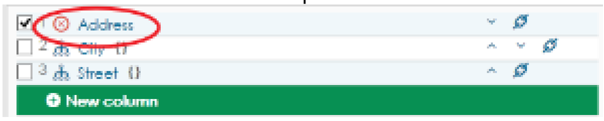
This script is checking whether the column Street is empty. In case it is, part of first column (Address) ending before comma is used. In case it is not empty, current value is used.



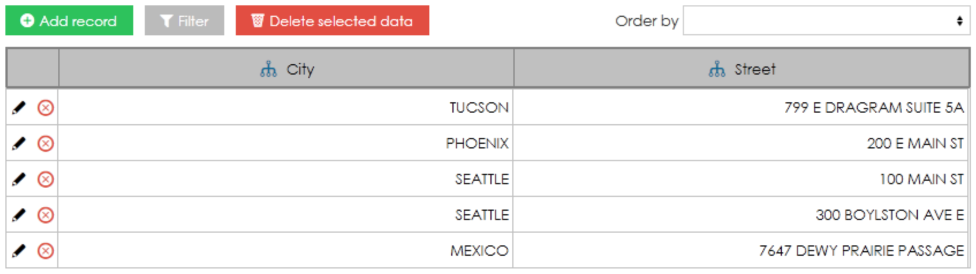
```
if (isEmpty(value(3))){
substringBefore(value(1), ',')
}
else return value(3)
```

799 E DRAGRAM SUITE 5A
200 E MAIN ST
100 MAIN ST
300 BOYLSTON AVE E
7647 DEWY PRAIRIE PASSAGE

4. Exclude column Adress from import



5. Click on Continue and Import data.
6. The result should look like this:



City	Street
TUCSON	799 E DRAGRAM SUITE 5A
PHOENIX	200 E MAIN ST
SEATTLE	100 MAIN ST
SEATTLE	300 BOYLSTON AVE E
MEXICO	7647 DEWY PRAIRIE PASSAGE

Data cleansing

It is recommended to get familiar with Transformation scripts basics before proceeding with this tutorial.

Goal of this use cases is to clean the values in imported data file.

Merging values

You can download data for this use case here.

Use **contains** function to transform values into the same value for all the rows. First you should define all the wrong appearances and then create condition for all of them. This function will update all the columns which contain selected string.

This function will be used for all the values which contain selected string. If you want to update only specific values, you can also use conditions, like following:

```
if (value(1) == 'UK') {
return "United Kingdom"
}
return value(1)
```

Removing unwanted values

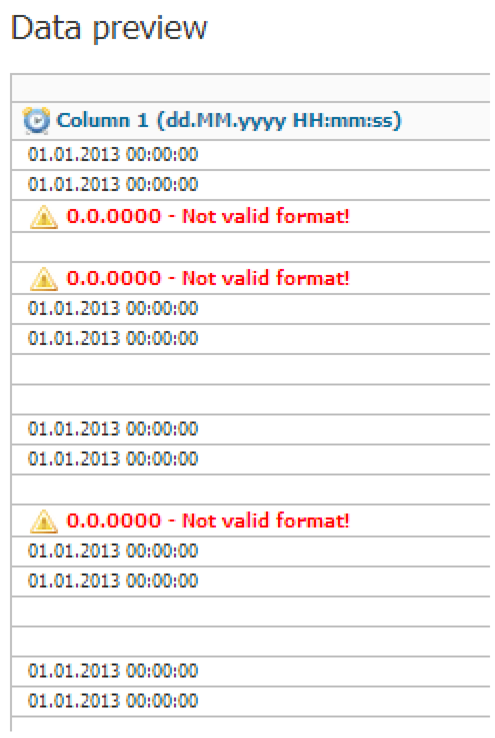
Sometimes a column should contain only specified values but thanks to some error in source system it also contains some unwanted values. In this example there is a column Gender which you should contain only values "Male" and "Female". Everything else should be replaced by "Unkown". You can use this script to get rid of them.

Similar way it is also possible to apply transformation scripts on indicators. For example you can get rid of values outside of range. In this example, we want to display all values greater than 1 as 0:

Handling empty and non-valid date formats

⚠ When importing data with date dimension, BellaDati will always check for valid date format.

If the date is missing or has wrong format, data will not be imported.



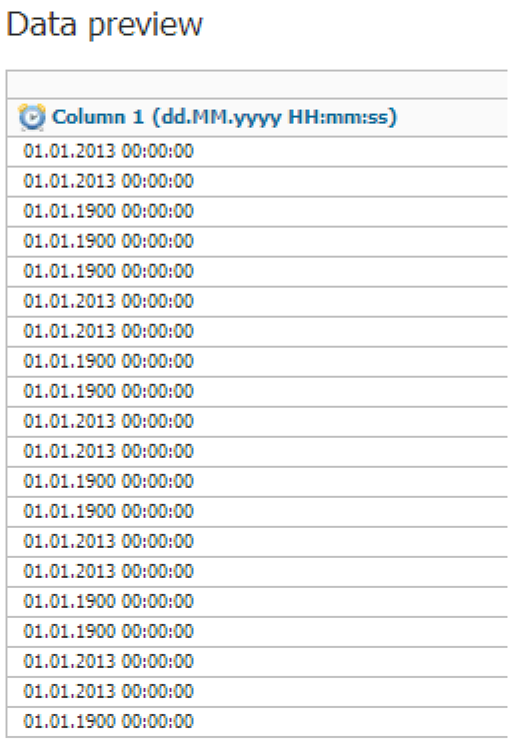
Data preview

Column 1 (dd.MM.yyyy HH:mm:ss)
01.01.2013 00:00:00
01.01.2013 00:00:00
⚠ 0.0.0000 - Not valid format!
⚠ 0.0.0000 - Not valid format!
01.01.2013 00:00:00
01.01.2013 00:00:00
01.01.2013 00:00:00
01.01.2013 00:00:00
⚠ 0.0.0000 - Not valid format!
01.01.2013 00:00:00
01.01.2013 00:00:00
01.01.2013 00:00:00
01.01.2013 00:00:00

You can use transformation script to handle these rows and modify them before being imported into BellaDati.

Following code checks if **date is missing** or is in default **00.00.0000** format.

Transformation script returns either **dummy date** or **original value**, depending on result of IF condition.


Data preview

Column 1 (dd.MM.yyyy HH:mm:ss)
01.01.2013 00:00:00
01.01.2013 00:00:00
01.01.1900 00:00:00
01.01.1900 00:00:00
01.01.1900 00:00:00
01.01.2013 00:00:00
01.01.2013 00:00:00
01.01.1900 00:00:00
01.01.1900 00:00:00
01.01.2013 00:00:00
01.01.2013 00:00:00
01.01.1900 00:00:00
01.01.1900 00:00:00
01.01.2013 00:00:00
01.01.2013 00:00:00
01.01.1900 00:00:00
01.01.1900 00:00:00
01.01.2013 00:00:00
01.01.2013 00:00:00
01.01.1900 00:00:00

After this transformation, all data will be imported. It is up to you how to report and analyze rows with dummy date.

Next Step

Transformation scripting

Monthly values with custom start date

The goal of this tutorial is to create a table that will display year over year comparison of values aggregated by months with custom start date. In this case we want to display data from 21st day of previous month to 20th day of actual month.

⚠ It is recommended to get familiar with date functions (month, date, dayOfMonth) before proceeding with this tutorial.

Data Set preparation

Two special attributes need to be created in data set:

- Create attribute “Statistical Month” in Data Set with transformation script:

Statistical Month

This is only a part of the script which is valid only for one month. The whole script can be found in an attachment: statisticalmonths.txt.

The script checks if the month is equal to 1 (January). Then it checks if the day of month is equal or bigger than 21. If it is true, then it is labelled as “Statistical February. If not, it is labelled as “Statistical January”. You can of course use different labels.

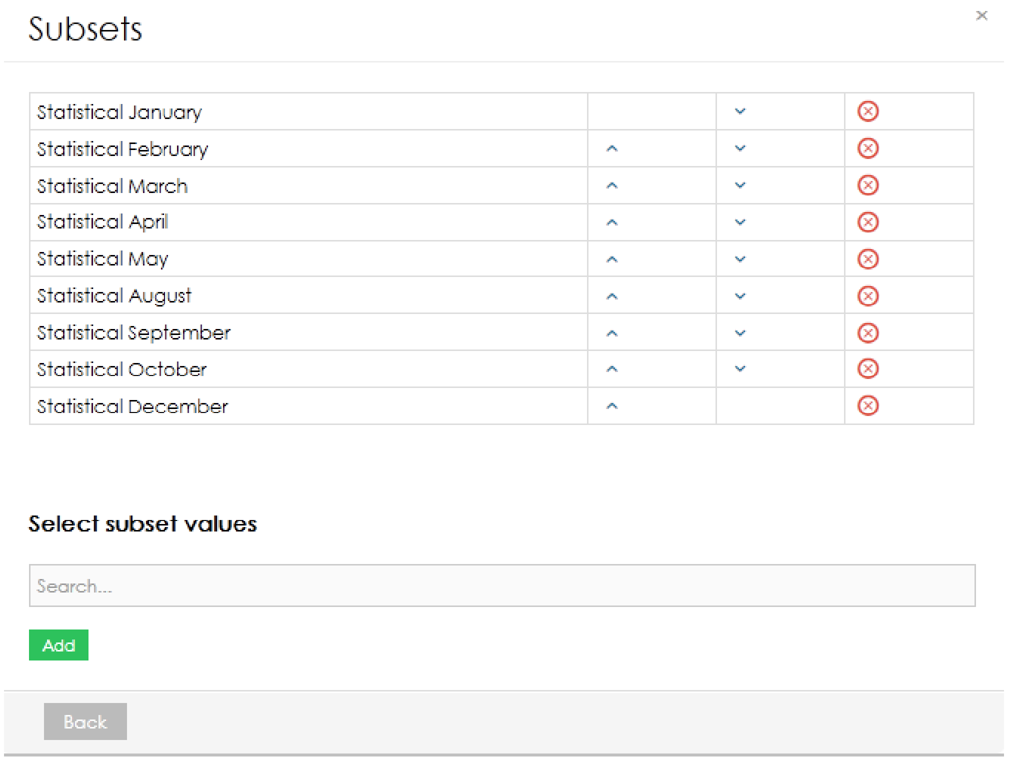
The same goes for all other months (only the month number and return value is changing).

- Create attribute “Statistical Year” in Data Set with transformation script:

Statistical Year

This script returns the year. If the date is bigger than 21st December, then it returns year +1. If not, it returns just the year. This way the “Statistical Year” lasts from last year’s 21st December to 20th December of this year (for example 2013/12/21 – 2014/12/20).

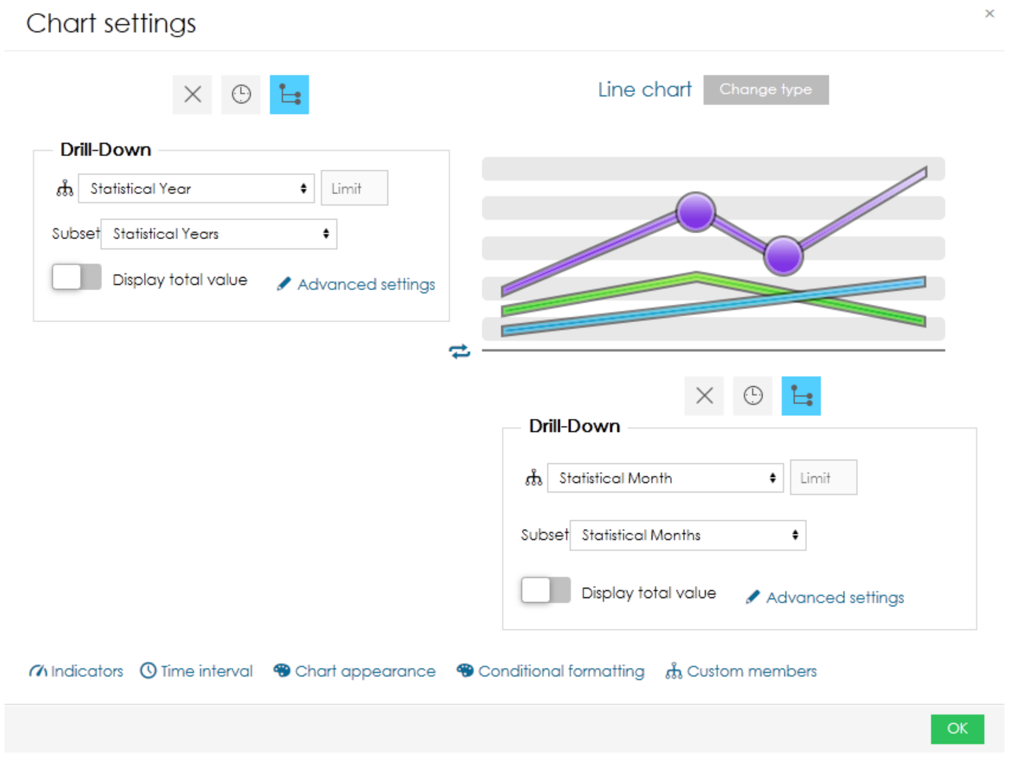
- Create Subsets in Data Set for Months and Years. The Months and Years need to be correctly sorted. Therefore you need to create subsets with correct sorting.


Subsets

Statistical January		˅	⊗
Statistical February	˄	˅	⊗
Statistical March	˄	˅	⊗
Statistical April	˄	˅	⊗
Statistical May	˄	˅	⊗
Statistical August	˄	˅	⊗
Statistical September	˄	˅	⊗
Statistical October	˄	˅	⊗
Statistical December	˄		⊗

Report

Create view with an indicator of your choice, attribute Statistical Month with selected Subset on x-axis and attribute Statistical Year with selected Subset of y-axis.



Parsing XML Content from Data Source

It is recommended to get familiar with working with XML content before proceeding with this tutorial.

In some cases it is necessary to store XML content in Data Source. In BellaDati it is possible to access and parse this data. This example shows how to parse data imported from MySQL database.

Preparation

The database table *test* contains sample data about books. For the purpose of trying this scenario, you can insert data into table with this command:

As you can see, last column contains list of copies which is stored in XML and has following structure:

Working with XML Content

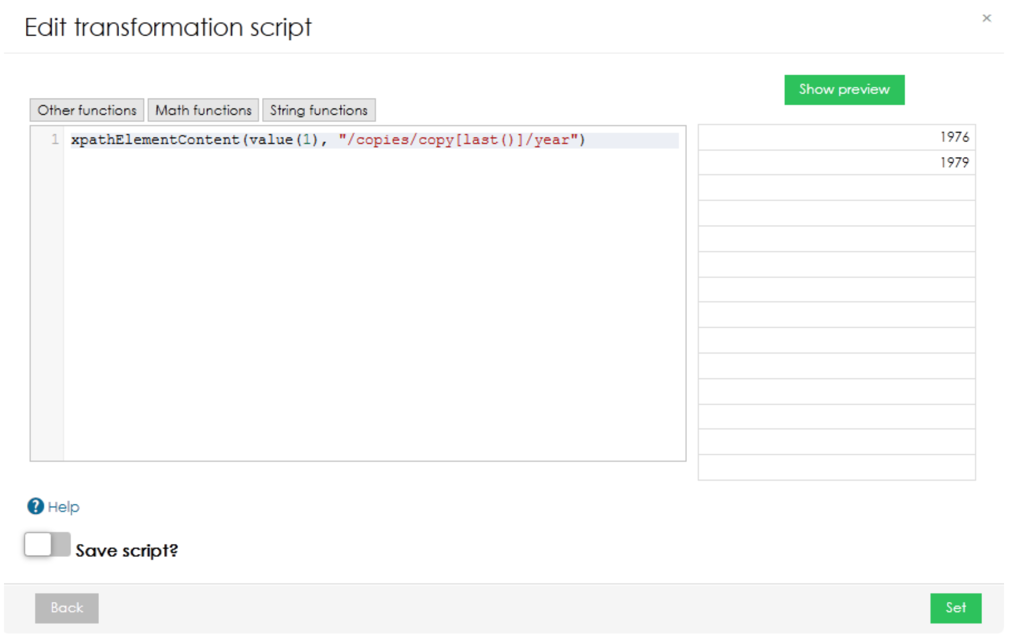
Set up an import of this table into BellaDati. On import settings page, add new columns that will contain parsed data. In BellaDati xPath is used for selecting XML elements.

Getting ID of first version

This code returns id of first copy for each book.

Getting latest year of publication

This code looks for last version and then returns its year of publication.



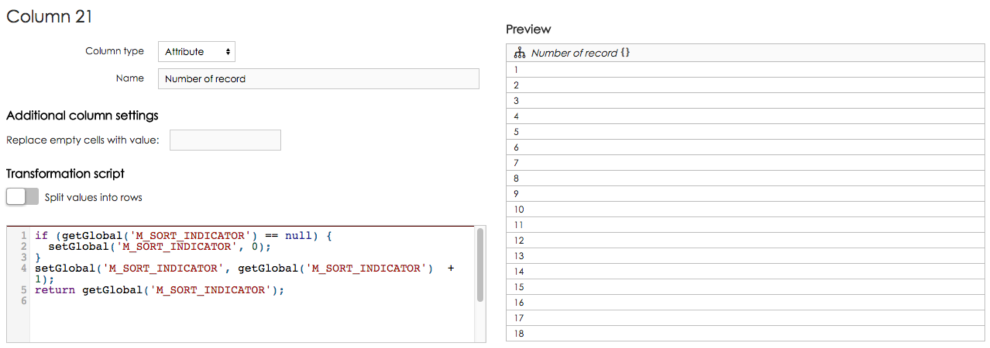
```
xpathElementContent(value(1), "/copies/copy[last()]/year")
```

```
1976
1979
```

Record ordering

In some cases, you might like to add numbers to the imported rows. In this case, there can be used following transformation script:

This transformation script will return record from 1 to the last record of imported data:



Column 21

Column type	Attribute
Name	Number of record

Additional column settings

Replace empty cells with value:

Transformation script

Split values into rows

```
if (getGlobal('M_SORT_INDICATOR') == null) {
  setGlobal('M_SORT_INDICATOR', 0);
}
setGlobal('M_SORT_INDICATOR', getGlobal('M_SORT_INDICATOR')  +
1);
return getGlobal('M_SORT_INDICATOR');
```

Preview

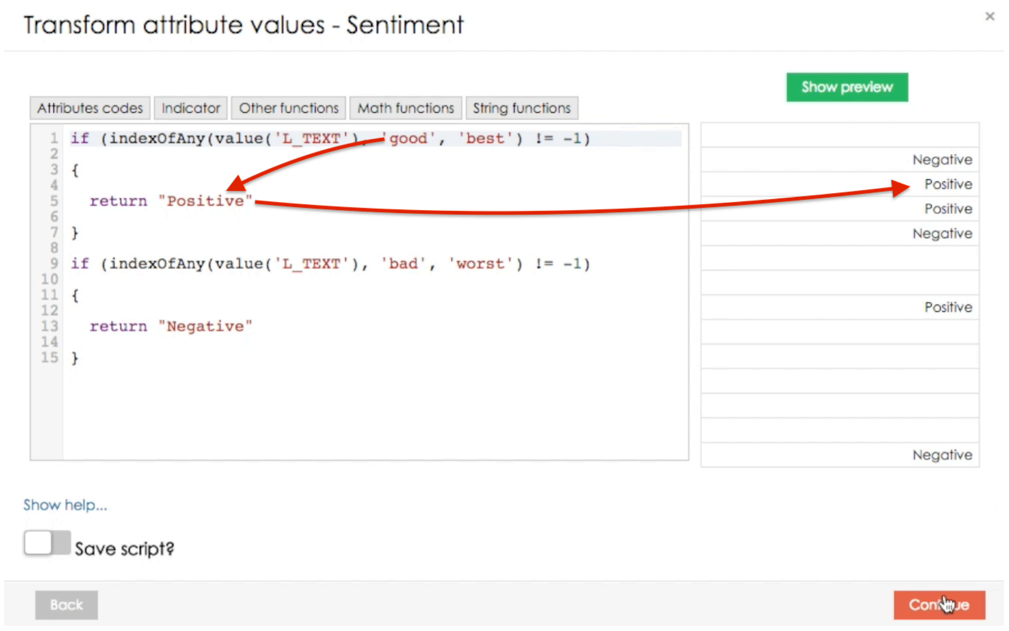
Number of record {}
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18

Sentiment Analysis

Your social network data might contain reactions from your customers. To do some basic classification, following transformation script can be used:

⚠ It is not recommended to use this transformation script on the same attribute you are analysing as the analysed text strings would be replaced by classifiers.

- 'L_ATTRIBUTE_CODE' is code of the attribute you want to analyse.
- Keywords represent string values' conditions which, when any met, return the specified classifier (in this example "Positive" or "Negative").
- Higher number of keywords raises the success rate of finding a match.



```
if (indexOfAny(value('L_TEXT'), 'good', 'best') != -1)

{

  return "Positive"

}

if (indexOfAny(value('L_TEXT'), 'bad', 'worst') != -1)

{

  return "Negative"

}
```

```
Negative
Positive
Positive
Negative


Positive





Negative
```

Whitelabeling BellaDati

You need to have **Domain administrator** role assigned to change BellaDati's design. In multi-domain configurations, you must have the **Global admin** role.

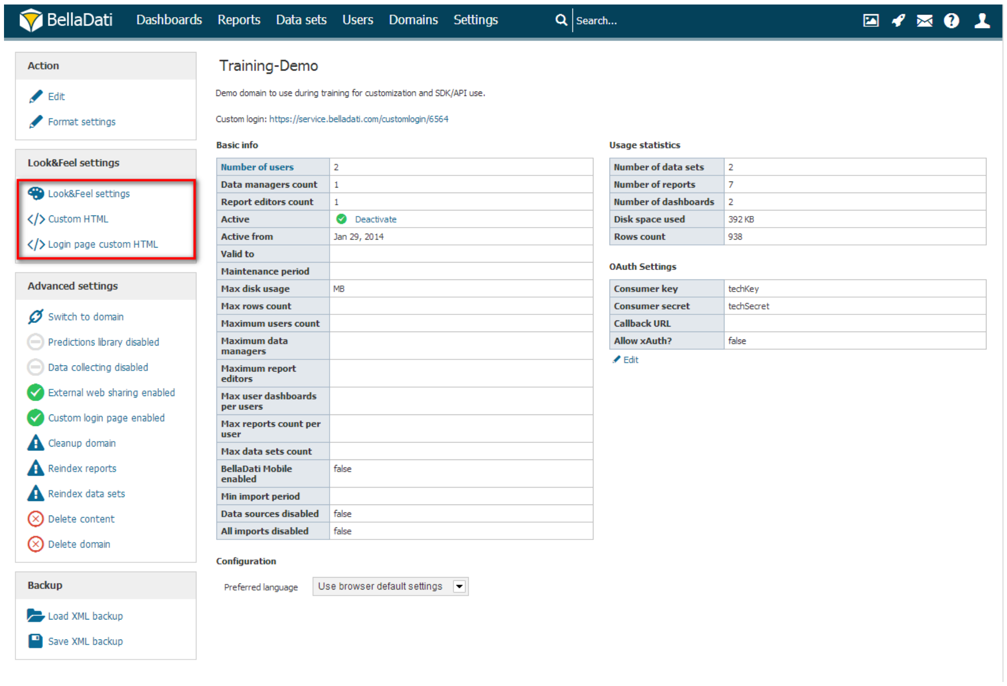
BellaDati allows you to change:

- Header logo
- Top and bottom menu background and text color
- Application background color
- Custom HTML and stylesheet
- Custom Login page

See the detailed whitelabeling manual here.

Changing look and feel

Go to **Domain** and select **Look&Feel Settings** in left navigation bar.



Training-Demo

Demo domain to use during training for customization and SDK/API use.

Custom login: https://service.belladati.com/customlogin/6564

Basic info

Number of users	2
Data managers count	1
Report editors count	1
Active	✓ Deactivate
Active from	Jan 29, 2014
Valid to	
Maintenance period	
Max disk usage	MB
Max rows count	
Maximum users count	
Maximum data managers	
Maximum report editors	
Max user dashboards per users	
Max reports count per user	
Max data sets count	
BellaDati Mobile enabled	false
Min import period	
Data sources disabled	false
All imports disabled	false

Usage statistics

Number of data sets	2
Number of reports	7
Number of dashboards	2
Disk space used	392 KB
Rows count	938

OAuth Settings

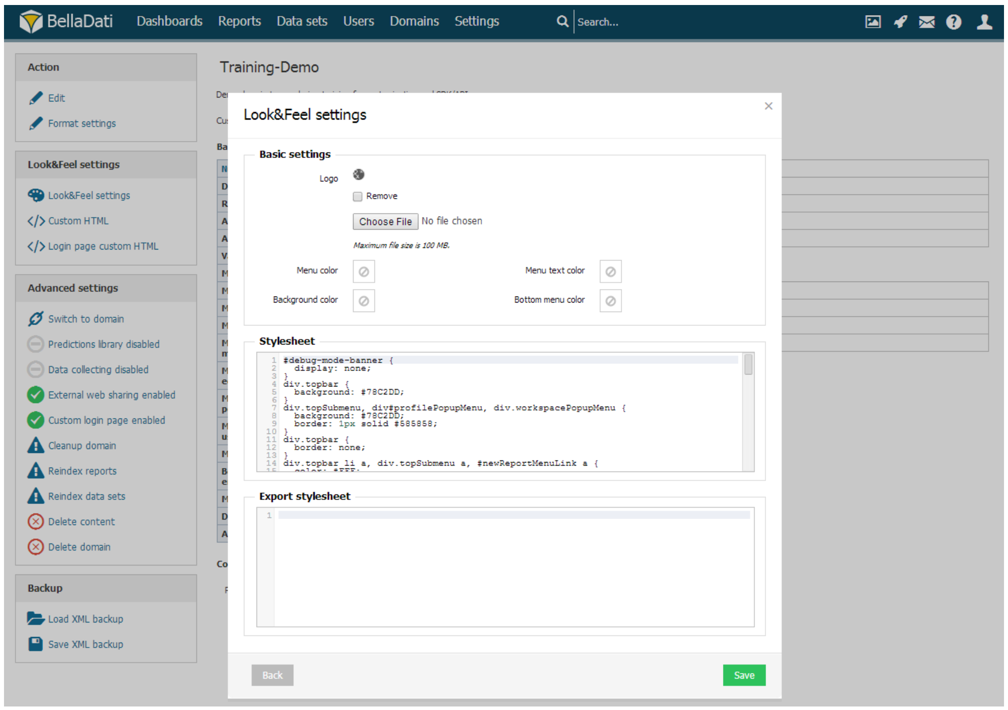
Consumer key	techKey
Consumer secret	techSecret
Callback URL	
Allow xAuth?	false

Edit

Configuration

Preferred language: Use browser default settings

Upload new logo or select from color picker to modify menu, background or footer colors.



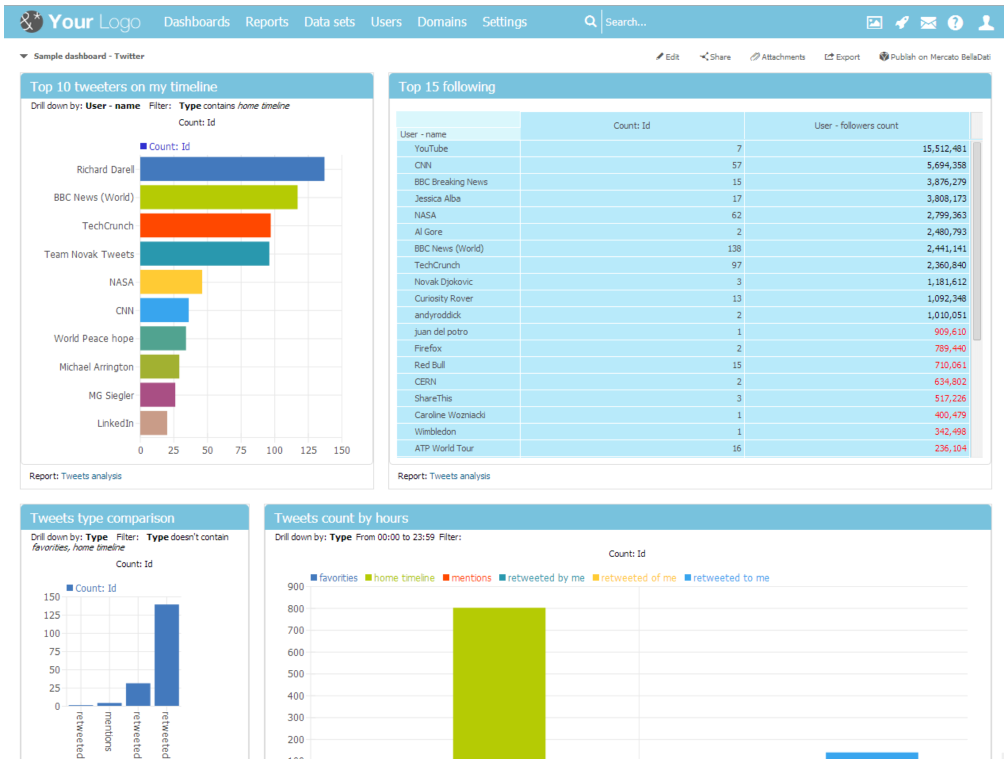
```
#debug-mode-banner {
  display: none;
}
div.topbar {
  background: #78C2DD;
}
div.topSubmenu, div#profilePopupMenu, div.workspacePopupMenu {
  background: #78C2DD;
  border: 1px solid #585858;
}
div.topbar {
  border: none;
}
div.topbar li a, div.topSubmenu a, #newReportMenuLink a {
```

While changing stylesheets we recommend to use Firefox's Firebug or Chrome's Developers Tools to identify used selectors.

Try our sample custom stylesheet

Results

Eventually, results may look like as follows:



User - name	Count: Id	User - followers count
YouTube	7	15,512,481
CNN	57	5,694,358
BBC Breaking News	15	3,876,279
Jessica Alba	17	3,808,173
NASA	62	2,799,363
Al Gore	2	2,480,793
BBC News (World)	138	2,441,141
TechCrunch	97	2,360,840
Novak Djokovic	3	1,181,612
Curiosity Rover	13	1,092,348
andyroddick	2	1,010,051
juan del potro	1	909,610
Firefox	2	789,440
Red Bull	15	710,061
CERN	2	634,802
ShareThis	3	517,226
Caroline Wozniacki	1	400,479
Wimbledon	1	342,498
ATP World Tour	16	236,104

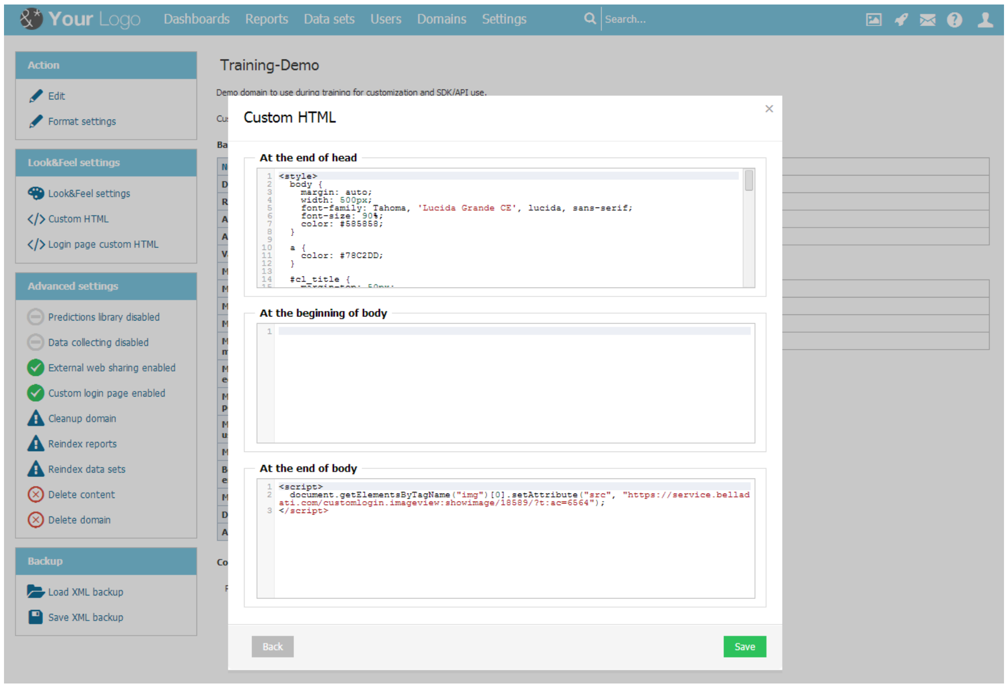
Adding Custom HTML

BellaDati allows you to place **custom HTML** at:

- the end of the head
- the beginning of the body
- the end of the body

Go to **Domain** and select **Custom HTML** in left navigation bar.

Place your HTML into appropriate text box.



Training-Demo

Demo domain to use during training for customization and SDK/API use.

Custom HTML

At the end of head

```
<style>
  body {
    margin: auto;
    width: 500px;
    font-family: Tahoma, 'Lucida Grande CE', lucida, sans-serif;
    font-size: 90%;
    color: #585858;
  }

  a {
    color: #78C2DD;
  }

  #cl_title {
```

At the beginning of body

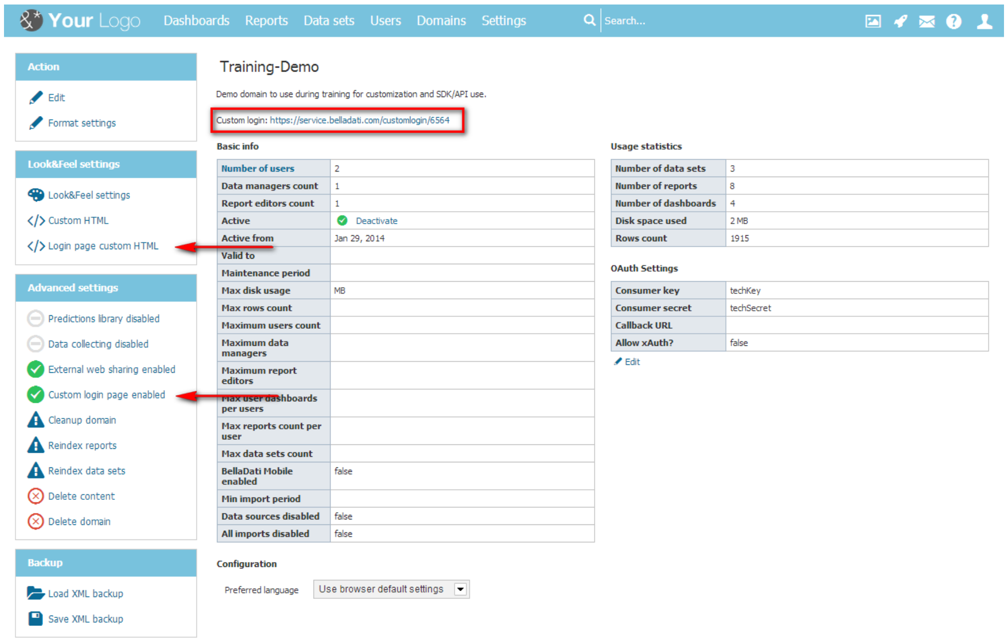
```
```

At the end of body

```
<script>
  document.getElementsByTagName("img")[0].setAttribute("src", "https://service.bellad
ati.com/customlogin.imageview:showimage/18589/?t:ac=6564");
</script>
```

Custom Login page

Enable custom login and edit the HTML for your customized login page.



Training-Demo

Demo domain to use during training for customization and SDK/API use.

Custom login: https://service.belladati.com/customlogin/6564

Basic info

Number of users	2
Data managers count	1
Report editors count	1
Active	✓ Deactivate
Active from	Jan 29, 2014
Valid to	
Maintenance period	
Max disk usage	MB
Max rows count	
Maximum users count	
Maximum data managers	
Maximum report editors	
Max user dashboards per users	
Max reports count per user	
Max data sets count	
BellaDati Mobile enabled	false
Min import period	
Data sources disabled	false
All imports disabled	false

Usage statistics

Number of data sets	3
Number of reports	8
Number of dashboards	4
Disk space used	2 MB
Rows count	1915

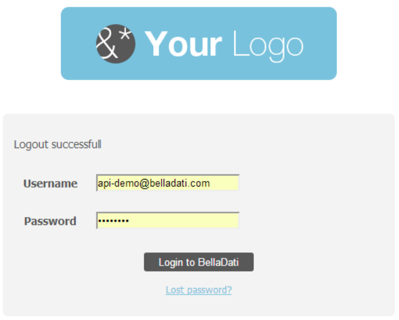
OAuth Settings

Consumer key	techKey
Consumer secret	techSecret
Callback URL	
Allow xAuth?	false

Configuration

Preferred language: Use browser default settings

Here is the result:

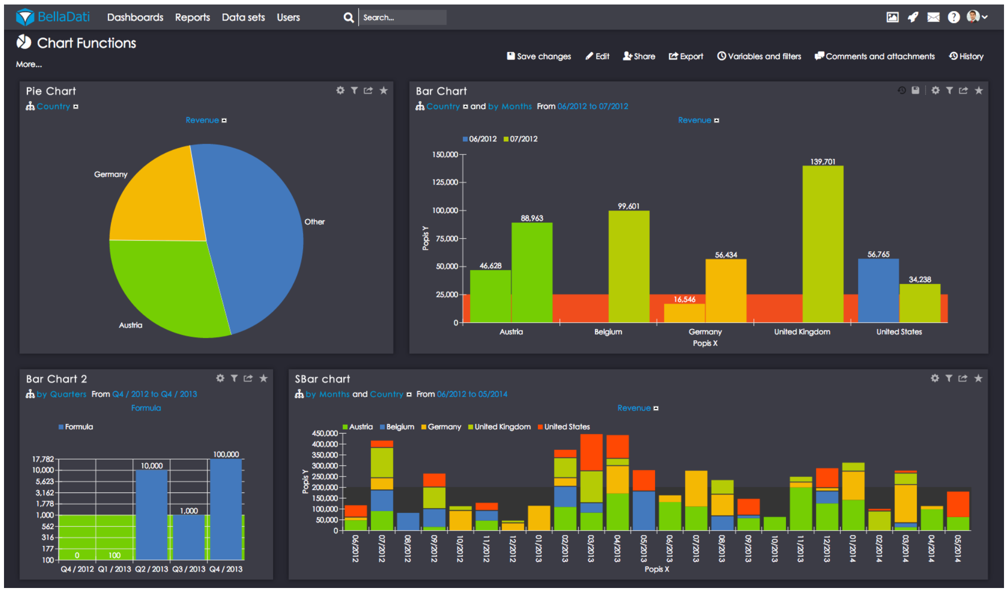

Logout successfull

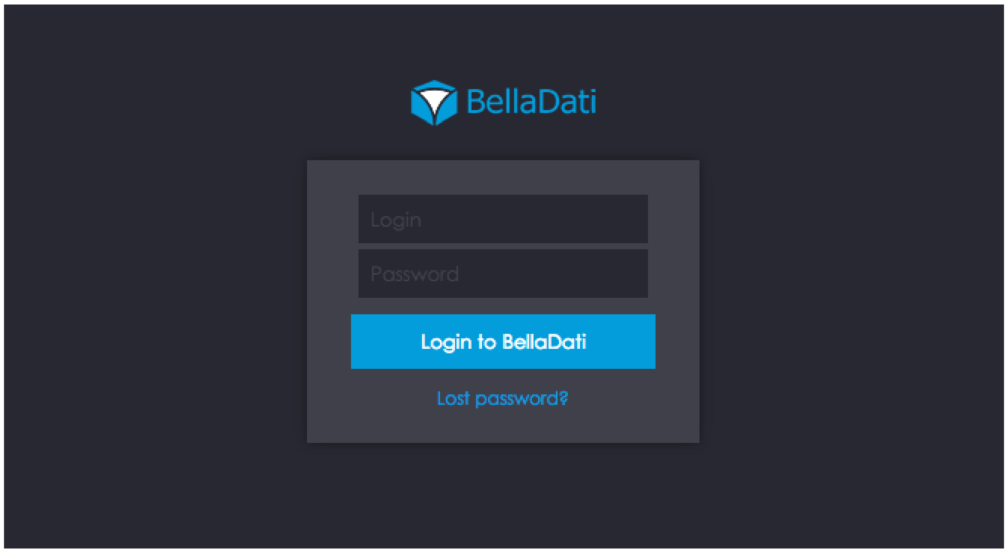
Username: api-demo@belladati.com

Password: ••••••••

BellaDati Dark theme

✓ Try our dark theme and dark login stylesheet!



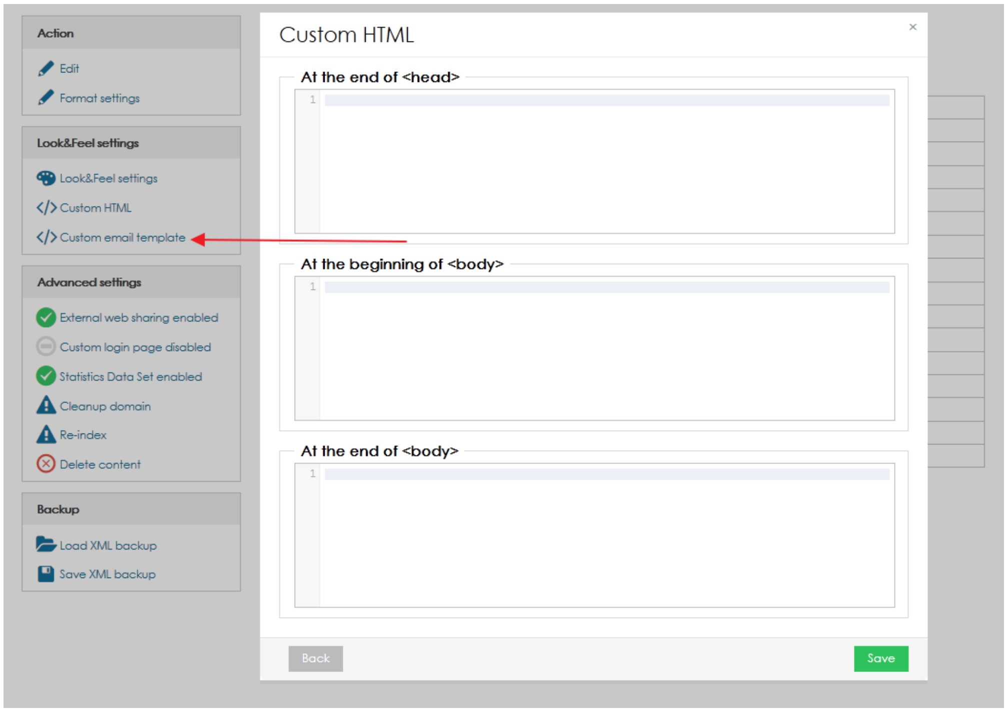


Custom email template

There are two ways how to customize email content.

1. If you want to set it for whole BellaDati installation, login as super admin and go to Settings - Configuration - Look & Feel box.

2. For one domain - login as domain admin, go to Manage Domain - Look & Feel box on left side of screen. This also overrides settings for whole installation.



For both options there are three parts which you can customize:

Email template - end of head - if you want to customize head of the template, add you code here.

Email template - beginning of body - if you want to override the default header, add you code here.

For example logo of your company can be added to header with this HTML code:

Email template - end of body - if you want to override the default footer, add you code here.

For example some information about your company can be added in footer:

Detailed Whitelabeling Manual

Below you can see a detailed list of css elements you can select and customize. That way you can very easily change a complete visual style of your Belladati domain.

See the whitelabeling tutorial here.

TOPBAR

Navigation topbar:	.topbar
Topbar menu links + icons:	.topbar ul li a
Search input:	.topbar form input
Search input typed text:	form#quickSearchForm .txt
Search icon:	#quickSearchForm .icon-search:before
Search vertical line:	.icon-search-position-fix (border-right: 1px solid #fff;)

BODY CONTENT

Be careful when modifying the global css styles, it could result in unexpected visual modifications throughout the entire application.

Body:	body.desktop
Dashboard dashlets title header:	.dashletHeader
Other links:	a

LEFT SIDEBAR

Sidebar body:	.sidebarBlock
Sidebar title:	.sidebarBlock h3
Sidebar link:	.sidebarBlock a

RIGHT SIDEBAR - HISTORY, FILTERS & VARIABLES

Sidebar body:	.setupPanel, .sidePanelPadding
Sidebar body content:	#reportView .setupPanel, #reportView .sharingPanel, #reportView .historyPanel
Icon colors:	.icon-info:before, .icon-search:before, .icon-calendar:before, .icon-attribute-date:before
Other icons “i”:	#reportView .setupPanel i, #reportView .sharingPanel i, #reportView .historyPanel i
Button colors:	.btn.primary, .btn
Sidebar links:	#reportView .setupPanel a, #reportView .sharingPanel a, #reportView .historyPanel a

REPORTS

Report title (without top banner):	.top_header_no_topbar h2
Report title (with top banner):	.topbar_bg_title
Report title icon (without top banner):	#contentColumn h2 [class^="icon-"]:before, h2 [class*=" icon-"]:before, #content h2 [class^="icon-"]:before, h2 [class*=" icon-"]:before
Report title icon (with top banner)	.topbar_bg_out [class^="icon-"]:before, .topbar_bg_out [class*=" icon-"]:before
Report top control buttons:	#reportPanelActions .btn
Report top control buttons pop-up menu:	#reportPopupMenu, #reportPopupMenu a
Report view control buttons:	.editViewMenu a, .viewMenu a
Report view title color:	.viewContentInnerLayout
Floating button - Edit:	.btn.primary.editModeStart
Floating button - Save changes:	.btn.primary.saveAllViews
Floating button - Finish:	.btn.primary.editModeEnd
Floating button - Reset:	.btn.editModeReset

TABLES

Table body:	table.list, table.t-data-grid
Table header:	table.t-data-grid thead tr th, table.t-data-grid tr th
Odd table row:	.oddRow, .oddRow td
Even table row:	.evenRow, .evenRow td

Data import table:	.dataImportTable

FORM ELEMENTS

Select	select
Text input	`input[type=text]`

BOTTOM MENU

Bottom menu:	#bottomMenu
Bottom menu links:	#bottomMenu a
Bottom menu copyright:	#bottomMenu .copyright

Bottom menu line separators	#bottomMenu .support, #bottomMenu .privacyPolicy, #bottomMenu .securityStatement, #bottomMenu .termsOfUse (border-left: 1px solid #7893ac;)

OTHER

To customize the other elements not listed here, please use the inspector feature in your browser in order to find the css selector.

Implementing simple SSO

Beside the standard SAML/Kerberos/SPNEGO implementation of SSO, BellaDati offers custom and easy-to-implement way to achieve the SSO.

Prerequisites

- installed BellaDati
- configured BellaDati with oAuth and CORS filter
- 3rd party application with option to recognize currently signed user

SSO Basic Flow

1. 3rd party application obtains the *accessToken* for the service account over the REST API
2. 3rd party application server executes “LOGIN_UNATTENDED” request REST API call and receives the *request_id* and *request_code* of the “user-login” request
3. 3rd party application generates a link pointing to the BellaDati's front-end service processing the “user-login” request
4. Execute XMLHttpRequest from the browser using the generated link
5. BellaDati process the request and do the “unattended” user-login and set appropriate session headers

SSO Operations

Only Admin user is allowed to execute the domain level operations.

Create User Request

3rd party application application is supposed to do the following:

1. obtains the *accessToken* and
2. issues the “LOGIN_UNATTENDED” request

Request structure:

URL	http://belladati_host/api/users/${username}/requests
Method	POST
Parameters	• *username* : username of the user to log-in • *request_type* : LOGIN_UNATTENDED (other types: PASSWORD_SET, PASSWORD_RESET, UNLOCK_ACCOUNT, LOGIN)
Returns	*request_id* and *request_code* of the created “user-login” request Example: 1544;RDQX1Qx9UokSf4n3KAVWgNClvrFUqncSZg7fK3gnVAfNIAOylN
Constraints	User request is valid for 30 seconds.

Refer to POST Create User Request for more details.

Process user login

3rd party application application is supposed to do the following:

- receives the *request_id* and *request_code* of the “user-login” request (previous step)
- generates a link pointing to the BellaDati's front-end service processing the “user-login” request
- Link will be called from the client side using the AJAX

Request structure:

URL	http://belladati_host/user/processRequest/{request_id}/{request_code}?redirect={redirect_url}
Method	GET
Parameters	• *request_id* : id of the “user-login” request • *request_code* : security verification code of the request • *redirect_url* (optional) : URL to redirect to
Returns	BellaDati processes the user login (sets JSESSIONID to the web browser) and returns OK or NOT_VALID;ERROR_MESSAGE. HTTP 200 in all cases.

Implementing SSO - C# example

To simplify the oAuth flow, we recommend to use the DevDefined.OAuth library.

Step 1 - obtain accessToken from BellaDati REST API

Configure the oAuth settings on your domain detail page. See REST API.

Step 2 - create unattended login request and get *request_id* and *request_code*

Step 3 - generate login link with *request_id* and *request_code*

Step 4 - execute XMLHttpRequest using the generated link

Don't forget to configure the CORS filter on BellaDati settings page.