Developer Tools

Contents:

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 interface 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.

On this page:

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:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:dataSet xmlns:ns2="http://www.belladati.com/api/v1.0" name="CRM - opportunities">
    <indicators>
        <indicator code="M_AMOUNT" name="Amount"/>
        <indicator code="M_PROBABILITY" name="Probability"/>
    </indicators>
    <attributes>
        <attribute code="L_CITY" name="City"/>
        <attribute code="L_ID" name="Id"/>
        <attribute code="L_NAME" name="Name"/>
        <attribute code="L_OFFICE" name="Office"/>
        <attribute code="L_PRODUCT" name="Product"/>
        <attribute code="L_STATUS" name="Status"/>
    </attributes>
    <dataSources>
        <dataSource name="SQL data source">
            <sql dbType="MYSQL">
                <columns>
                    <timeColumn localeString="cs" format="dd.MM.yyyy" index="0" name="Date"/>
                    <attributeColumn code="L_ID" index="1" name="Id"/>
                    <attributeColumn code="L_NAME" index="2" name="Name"/>
                    <attributeColumn code="L_OFFICE" index="3" name="Office"/>
                    <attributeColumn code="L_PRODUCT" index="4" name="Product"/>
                    <attributeColumn code="L_STATUS" index="5" name="Status"/>
                    <attributeColumn code="L_CITY" index="6" name="City"/>
                    <indicatorColumn code="M_AMOUNT" index="7" name="Amount"/>
                    <indicatorColumn code="M_PROBABILITY" index="8" name="Probability"/>
                </columns>
                <overridingTimeColumnDate>2010-12-17</overridingTimeColumnDate>
                <properties>
                    <entry name="SQL" value="select * from crm_opportunities"/>
                </properties>
                <connectionParameters>
                    <entry name="database" value="crm_db"/>
                    <entry name="host" value="host_address"/>
                    <entry name="password" value="password"/>
                    <entry name="user" value="user"/>
                </connectionParameters>
            </sql>
        </dataSource>
    </dataSources>
    <permissions>
        <owner>support@belladati.com</owner>
    </permissions>
</ns2:dataSet>
```

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

Creating data set

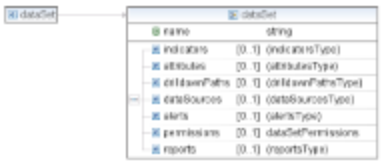
Data set is defined by `<dataSet>` tag.

Example

```
<ns2:dataSet xmlns:ns2="http://www.belladati.com/api/v1.0" name="CRM - opportunities">
   ...
</ns2:dataSet>
```

Tag attributes

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



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

```
<indicators>
       <indicator code="M_AMOUNT" name="Amount" unit="USD"/>
       <indicator code="M_COUNT" name="Count"/>
   </indicators>
```

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

```
<indicators>
       <indicator code="M_AMOUNT" name="Amount" unit="USD"/>
       <indicator code="M_COUNT" name="Count"/>
       <indicator code="M_RATE" name="Rate">
           <formula>M_AMOUNT / M_COUNT</formula>
       </indicator>
   </indicators>
```

Creating indicator's translation

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

Example

```
<indicators>
       <indicator code="M_AMOUNT" name="Amount" unit="USD">
            <translations>
                <translation locale="de" value="Translated value"/>
            </translations>
       </indicator>
</indicators>
```

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

```
<indicators>
        <indicatorGroup name="group">
            <indicator code="M_INDICATOR" name="indicator"/>
            <indicatorGroup name="sub-group">
                <indicator code="M_SUBGROUP_INDICATOR" name="sub-group indicator"/>
            </indicatorGroup>
        </indicatorGroup>
    </indicators>
```

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.

Example

```
<attributes>
        <attribute code="L_CITY" name="City"/>
        <attribute code="L_ID" name="Id"/>
        <attribute code="L_NAME" name="Name"/>
        <attribute code="L_OFFICE" name="Office"/>
        <attribute code="L_PRODUCT" name="Product"/>
        <attribute code="L_STATUS" name="Status"/>
    </attributes>
```

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

```
<attributes>
        <attribute code="L_CITY" name="City"/>
            <translations>
                <translation locale="de" value="Translated value"/>
                <translation locale="cs" value="Another translated value"/>
            </translations>
       </attribute>
    </attributes>
```

Tag attributes

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

Creating member's translation

Example

```
<attribute code="L_CITY" name="City">
   <memberTranslations>
       <memberTranslation memberValue="Prague" locale="cs" value="Praha"/>
       <memberTranslation memberValue="Prague" locale="de" value="Prag"/>
   </memberTranslations>
</attribute>
```

Defining attribute's appearance

Example

```
<attribute code="L_CITY" name="City">
   <memberSettings>
       <memberSetting icon="cs" backgroundColor="#c41019" color="#ffffff" memberValue="Prague"/>
   </memberSettings>
</attribute>
```

Defining drilldown paths

Example

```
<drilldownPaths>
   <pathAttribute code="L_ATTRIBUTE_1">
       <pathAttribute code="L_ATTRIBUTE_2"/>
       <pathAttribute code="L_ATTRIBUTE_2"/>
    </pathAttribute>
</drilldownPaths>
```

Transformation scripting

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

Transfornation scripting is an advanced features for processing data during import. Thanks to transformations cripts 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 enabled to enter transformation scripts in the column detail dialog while setting or editting import settings.

On this page:

Transformation scripts syntax resembles the Groovy and Java syntax. It is designed to be readable and effective but not strict. Users dfamiliar 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:

```
val = value()
return trim(val)
```

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:

`String value(int columnIndex)`	returns the value of column at `columnIndex`
`String name()`	returns the name of the current column
`String name(int columntIndex)`	returns the name of the column a `colimnIndex`
`format()`	returns format of the current column

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:

```
a = value()
b = 12
c = 100.56
bool = false
e = "Hello"
x = b + c
return e + " world"
```

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

```
double b = value() as double
```

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:

```
val = value() as double
result = "Success"
if (val > 100) result = "Too high"
return result
```

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:

```
val = value() as double
result = "Success"
if (val > 100) {
   result = "Too high"
   result += " (" + val.toString() + ")"
}
return result;
```

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:

```
val = value() as double
if (val > 100) {
    result = "Too high"
} else {
    result = "Success"
}
return result;
```

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

```
val = value() as double;
return val > 100 ? "Too high" : "Success";
```

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

```
switch (x) {
    case "Specific string value":
         result = "Contains specified string value"
         break
    case [4, 5, 'a', 'b']:
         result = "Is 4, 5, a, or b."
         break
    case 12..30:
         result = "In range"
         break
    case Number:
         result = "Is number"
         break
    default:
         result = "Default"
}
```

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

```
hodnota = value() as double;
result = "Lower than 100";
if (hodnota > 100) result = "Greater than 100";
return result;
```

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

```
if (value(6) == 'Credit') { return value(5)
} else if (value(6) == 'Debit') { return -value(5)
} else { return 0
}
```

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

On this page:

Function	Description

`LocalDate date(String value)`	Converts the passed **`value`** to **`LocalDate`** object. The **`value`** must be in on of the following formats: **`dd.MM.yyyy`**, **`dd/MM/yyyy`**, **`yyyy-MM-dd`**. Example:
	`def a = date('2011-01-01')`
	`def b = date(value(1))`

`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:
	`def a = time('11:00:23')`
	`def b = time(value(1))`

`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 on of the following formats: **`dd.MM.yyyy HH:mm:ss`**, **`dd/MM/yyyy HH:mm:ss`**, **`yyyy-MM-dd HH:mm:ss`**. Example:
	`def a = datetime('2011-01-01') //creates date time object with 2011-01-01 00:00:00 value`
	`def b = datetime('2011-01-01 23:32') //creates date time object with 2011-01-01 23:32:00 value`
	`def c = datetime('11:21:33') //creates the date time object with 1970-01-01 11:21:33 value`

`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.

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:
	`def a = date('2011-01-01')`
	`return year(a);//returns 2011`

`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:
	`//returns number of seconds between 00:30:32 and time value in column 1.`
	`secondsBetween(time('00:30:32'), time(value(1))`

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** instances. Example:
	`//returns number of days between 2011-01-01 and value in column 1.`
	`daysBetween(date('2011-01-01'), date(value(1))`
	`//returns number of days between the actual date and value in current column`
	`daysBetween(date(actualDate()), date(value())`

`Integer daysBetween(DateTime dt1, DateTime dt2)`	Returns the number of days between the two specified **`DateTime`** instances.
`Integer weeksBetween(LocalDate d1, LocalDate d2)`	Returns the number of weeks between the two specified **`LocalDate`** instances.
`Integer weeksBetween(DateTime dt1, DateTime dt2)`	Returns the number of weeks between the two specified **`DateTime`** instances.
`Integer monthsBetween(LocalDate d1, LocalDate d2)`	Returns the number of months between the two specified **`LocalDate`** instances.
`Integer monthsBetween(DateTime dt1, DateTime dt2)`	Returns the number of months between the two specified **`DateTime`** instances.
`Integer yearsBetween(LocalDate d1, LocalDate d2)`	Returns the number of years between the two specified **`LocalDate`** instances.
`Integer yearsBetween(DateTime dt1, DateTime dt2)`	Returns the number of years between the two specified **`DateTime`** instances.

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` value.
`DateTime minus(LocalDate date, String part, int count)`	Subtracts the amount of specified `part` from the passed `LocalDate` value.
`DateTime minus(DateTime dt, String part, int count)`	Subtracts the amount of specified `part` from the passed `DateTime` value.

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: `return actualDate('yyyy-MM-dd');//returns date as 2011-12-15`
`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: `return actualDate('HH:mm:ss');//returns time as 13:44:22 string`
`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.

```
def a1 = split(value(1), '/');
def date1 = [a1[0] as int, a1[1] as int, a1[2] as int] as Date;
def a2 = split(value(2), '/');
def date2 = [a2[0] as int, a2[1] as int, a2[2] as int] as Date;
return date2 - date1;
```

External sources functions

Functions for loading data from external sources (url, database etc.) will be released in next version of BellaDati BI. You can use [Work with lists and maps] instead.

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()`:

```
def stringList = [ "java", "perl", "python", "ruby", "c#", "cobol",
                   "groovy", "jython", "smalltalk", "prolog", "m", "yacc" ];

stringList.each() { print " ${it}" };
```

Math, formulas, expressions

Always check which decimal separator 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:

```
x = 100
a = a + 14 * 5 / (1 + x) + ("0.3" as double)
```

Equations (returning boolean values):

```
b = a == 15
```

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

```
a = isBlank("")
b = isBlank("not blank string")
c = b && a || false
```

Is the same as:

```
c = isBlank("") && isBlank("not a blank string")
```

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 concated together by a + oeprator:

```
a = "first part" + "second part" + "third part"
```

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

```
return (((value() as double) / 3600000) * 60)
```

Rounding

```
number = replace(value(), ',', '.') as double
return round(number)
```

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 "."

```
return replace(value(), ".", ",")
```

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:

```
return trim(value());
```

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.

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().
`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:

```
return capitalize(substringBefore(value(1), ' '))
```

Script for surname name:

```
return capitalize(substringAfter(value(1), ' '))
```

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:

```
emptyList = []
myList = [1776, -1, 33, 99, 0, 928734928763]
return [0]
```

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:

```
emptyMap = [:]
scores = [ "Brett":100, "Pete":"Did not finish", "Andrew":86.87934 ]
return scores["Pete"]
```

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:

```
scores["Pete"] = 3
```

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

```
scores.size()
```

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

```
myList.each() {
  if (it > 0) {
    return it;
  }
}
```

Samples

Convert bank account numver 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:

```
def bankCodes = [ '0100' : 'KB', '0800' : 'CSOB'
];
code = right(value(), 4)
return bankCodescode ? bankCodescode : 'Other'
```