



Blaise



Blaise 5

The Kami sample

“The fashionable way to style your meta-data”

How did this come about?

- A user request came in to get some meta data into a compliant format that could be used in a Python program
 - This was achieved by taking the JSON export from Blaise and reformat it in a Python program into the desired format
- A later request came in to expand the export function to likewise export meta data
 - The current procedure involved using the SAS/SPSS/Stata sample and some programming
 - Turns out they weren't the only ones doing this
- After a teleconf, some decisions were made and a program written to fit the requirements



What sort of thing did they want?

```
"AppointType": {
```

```
  1: "No preference",
```

```
  2: "Appointment for date and time",
```

```
...
```

```
"WeekDay": "Weekday of slice dial",
```

```
"DialTime6": "Slice dial time",
```

```
...
```

```
"AppointType": "CatiMana.CatiAppoint.AppointType",
```

```
"AssignModeParam": "offlinecapi.AssignModeParam",
```



The bump in the road

- Just before work was completed on the sample, another request came in to export the metadata and use it in a Python program
- The problem with this request was that the output format desired was slightly different from what had already been programmed
- What to do, what to do...?



The solution

- The decision was made to finish the original code with the (mostly) hardcoded layout and dual-path a set of more flexible layout options
- These code paths are called:
 - Basic
 - Extended
- You tell the sample which path you want by setting the appropriate option in the (compulsory) options file



Basic implementation

- Layout in Basic is mostly hard-coded
- There are a few options that give some space to modify some things a little bit
 - sometimes you can filter on specific types
 - sometimes you can map a Blaise type to another string
 - sometimes you can add an extra string into certain places
 - sometimes you can sort the output
 - sorted on localname
 - restrictions apply in certain cases
- **Not everything is implemented in every place – usually it's only where requested!**
- For anything that really diverges from the layout model used in Basic, use Extended



Basic .kifx sample

[enumsandsets as variable_value specialattributes yes]

[FieldTexts as variable_text expand ELM]

[FieldTypes as dtype filter STRING,DATETYPE mappings STRING,str,DATETYPE,str]

[FieldNames as variable_name sort yes expand ELM]

[SetNamesDict as set_off_dict]

[verbatim]

#program to do stuff

....

[everbatim]



Basic .kifx sample output - 1

[enumsandsets as variable_value specialattributes yes]

```
variable_value = {
```

```
  "AppointType": {
```

```
    1: "No preference",
```

```
    2: "Appointment for date and time",
```

```
    3: "Preference for a period",
```

```
    4: "Preference for days of the week"
```

```
  },
```

```
...
```

AppointType - > ENUM (length 4)



Basic .kifx sample output - 2

```
"WeekDays_1": {  
  1: "Sunday",  
  2: "Monday",  
  3: "Tuesday",  
  4: "Wednesday",  
  5: "Thursday",  
  6: "Friday",  
  7: "Saturday"  
},
```

```
"WeekDays_2": {  
  1: "Sunday",
```

...

Weekdays -> SET OF TCatiMana.TWeekDay



Basic .kifx sample output - 3

```
Special_attributes = {  
  "nrlangs": {  
    8: "Rather not answer",  
    9: "Don't know"  
  },  
  "Status": {  
    98: "Rather not answer",  
    99: "Don't know"  
  },  
  ...  
}
```



Basic .kifx sample output - 4

[FieldTexts as variable_text expand ELM]

```
variable_text = {  
  "AppointType": "When can we call you back ?",  
  "DateStart": "Start date",  
  "TimeStart": "Start time",  
  "DateEnd": "End date",  
  "TimeEnd": "End time",  
  "WeekDays_1": "Selected weekdays",  
  ...
```

Open/Close dictionary and quotes are specified in the options file



Basic .kifx sample output - 5

...

"WeekDays_6": "Selected weekdays",

"WeekDays_7": "Selected weekdays",

"WeekDays_ELM_1": "Sunday (Code 1)",

"WeekDays_ELM_2": "Monday (Code 2)",

"WeekDays_ELM_3": "Tuesday (Code 3)",

"WeekDays_ELM_4": "Wednesday (Code 4)",

...



Basic .kifx sample output - 6

```
[FieldTypes as dtype filter STRING,DATETYPE mappings STRING,str,DATETYPE,str]
```

```
dtype = {
```

```
    "WhoMade": "str",
```

```
    "WhoMade2": "str",
```

```
    "WhoMade3": "str",
```

```
    "passorder": "str",
```

```
...
```



Basic .kifx sample output - 7

[FieldNames as variable_name sort yes expand ELM]

```
variable_name = {  
  "Address": "ioblock.Address",  
  "AdrRet": "ioblock.AdrRet",  
  "AppointType": "CatiMana.CatiAppoint.AppointType",  
  "BAKG1a": "schema.spmBackground.BAKG1a",  
  "BAKG1b": "schema.spmBackground.BAKG1b",  
  "LIBACCESS_ELM_1": "schema.spmAccess.LIBACCESS_ELM_1",  
  "LIBACCESS_1": "schema.spmAccess.LIBACCESS[1]",  
  "LIBACCESS_ELM_2": "schema.spmAccess.LIBACCESS_ELM_2",  
  "LIBACCESS_2": "schema.spmAccess.LIBACCESS[2]",  
  ...  
}
```



Basic .kifx sample output - 8

```
[SetNamesDict as set_off_dict]
set_off_dict = {
  "CatiMana.CatiAppoint.WeekDays" : {
    "CatiMana.CatiAppoint.WeekDays[1]":1,
    "CatiMana.CatiAppoint.WeekDays[2]":2,
    ...
    "CatiMana.CatiAppoint.WeekDays[6]":6,
    "CatiMana.CatiAppoint.WeekDays[7]":7},
  "schema.spmCinema.CINEMATYP" : {
    "schema.spmCinema.CINEMATYP[1]":1,
    ...
  }
}
```



Extended implementation

- Layout in Extended is heavily influenced by the user
- The layout file (mostly) dictates how items are formatted (see below)
- Each type gets its own formatting instructions
 - CLASSIFICATION, DATETYPE, DATETIMETYPE, ENUMERATION, INTEGER, OPEN, REAL, SET, STRING, TIMETYPE
- List type items have their own placeholder and a format for each item
 - ENUMs, SETs, SPECIALANSWERS
 - The code for these items keeps track of whether the last comma is needed
 - There is an option to influence whether the last comma is to be kept which allows two list templates to be placed after one another



Sample type block

[dateformat]

["]*fieldname*[":]

{"qu_text": "][*questiontext*[" ,]

["fieldsize":]][*width*][,]

["val_labels": {]

[*specialattributesvaluesandlabels*]

[}]

[},]

[edateformat]

Items in italics are keywords



Sample list type blocks

[specialattributesvaluesandlabelsformat]

[specialattributesvaluesandlabelsitems]

[especialattributesvaluesandlabelsformat]

[specialattributesvaluesandlabelsitemsformat]

[*specialanswervalue*][: "][*specialanswertext*]["]

[especialattributesvaluesandlabelsitemsformat]



Sample type block results

"Main.PIIPage.BornOn":

```
{"qu_text": "^{\resp} What is your date-of-birth? <newline>(DD-MM-YYYY)",
```

```
"fieldsize": 8,
```

```
"val_labels": {
```

```
  99999998 : "Rather not answer"
```

```
}
```

```
},
```

```
[ "[fieldname]":  
  [{"qu_text": ""}[questiontext][",]  
  ["fieldsize": ][width][",]  
  ["val_labels": {}  
  [specialattributesvaluesandlabels]  
  []  
  [},]
```



keeplastcomma

- Remember: list items are programmed to exclude the last comma in a list
- If you have a template that has two comma-separated list items after each other, the system has to know that the first list item has to keep the last comma
 - This to avoid syntax errors
- List item templates have an option:
 - `keeplastcomma yes|no|sa`
- `yes` & `no` do what they say on the tin
- `sa` means “keep the last comma if this field has specialattributes”
 - This is catered for in the controlling if statements in the Manipula code



Sample .kifx files

- Basic
 - Python formatted output
- Extended
 - Python formatted output
- R-Extended
 - R formatted output
- SQL-Extended
 - This gives out SQL commands using enum and set info
 - These commands can be used for example, to create tables that cross-ref the integer values with the text values of the items in enum fields



Datamodel level items

- Whilst the original usage for Kami was at field level, meta information is also available for the datamodel
- Extra code has been added to extract meta information at datamodel level
- Datamodel level items (generally) begin with *dm* by convention
- .kifx files at the datamodel level follow the same syntactical concepts as those at field level
- The XML-Extended.kifx file shows the possibilities



XML-Extended.kifx snippet - 1

[datamodelformat]

[<DataModel name="[dmname]">]

[dmroletextscollection]

[dmlanguagescollection]

[dmspecialanswersettingscollection]

[dmparalleldefinitionscollection]

[dmkeydefinitionscollection]

[dmrolescollection]

[dmmodescollection]

[</DataModel>]

[edatamodelformat]



XML-Extended.kifx snippet - 2

[dmlanguagescollectionformat]

[<LanguageCollection>]

[dmlanguagesitems]

[</LanguageCollection>]

[edmlanguagescollectionformat]

[dmlanguagesitemsformat]

[<Language Name="[*name*]" Description="[*desc*]" IsActive="[*active*]" />]

[edmlanguagesitemsformat]



XML-Extended.kifx snippet - 3

[verbatim]

```
<!-- Generated by Kami -->
```

[everbatim]

[datamodel]

[verbatim]

```
<!-- End of generation by Kami -->
```

[everbatim]



XML-Extended.kifx sample output - 1

```
<!-- Generated by Kami -->  
<DataModel name="FlightSurvey">  
  <RoleTextsCollection>  
    <RoleTexts Role="Description">  
      <Text>  
        <Expression Text="Flight Survey" />  
      </Text>  
      <Text>  
        <Expression Text="Vuelo Encuesta" />  
      </Text>
```



XML-Extended.kifx sample output - 2

<Text>

<Expression Text="飞行调查" />

</Text>

<Text>

<Expression Text="טיסה סקר" />

</Text>

</RoleTexts>

<RoleTexts Role="Title">

<Text>

<Expression Text="Flight Survey" />



Substitution from outside

- How can we influence things like filenames which may differ between runs?
- We need some way of providing the information
- We need some way of substituting it into the correct place in the template

- Pass the required values through Manipula's `-V` command as array members
- Use a `[parse]` block and parameter indicators `[IP1]...[IP10]`
- The `[parse]` block functions similarly to the `[verbatim]` block but parameter indicator terms are substituted with the substitution value via a regular expression



Substitution from outside - sample .kifx definition

[verbatim]

x := 1

[everbatim]

[parse]

datafile := [IP1]

[eparse]

[verbatim]

input.OPEN(datafile)

...

commandline option -> -V:IP[1]=C:\TEMP\myfile.txt



Options file

- There used to be a list of options passed via the command-line
- This very quickly became unwieldy
- Options are now supplied via an options file



Options - 1

- Package=Basic | Extended
 - which code path to use
- Language=language code
 - a language code from the datamodel to use when retrieving meta
 - may be left blank (uses the first language as there is always at least one language)
- Role=role name
 - a role name from the datamodel to use when retrieving meta
 - may be left blank
- Mode=mode name
 - a mode name from the datamodel to use when retrieving meta
 - may be left blank (the code uses what the system returns when no mode is specified)
 - e.g. SpecialAnswers contains all of them, ignoring any mode behaviours



Options - 2

- `Outputfile`=fully-qualified name of output file
- `Templatefile`=fully-qualified name of the (.kifx) file describing the output
- `DoAddZeroToIndex`=Yes | No
 - do you want `Fred[1]` or `Fred[01]`?
- `CloseDict`=}
- `OpenDict`={
- These last two allow specification of the delimiters for dictionaries
 - the fields are defined as length 10
 - this allows extended delimiters such as `[[` to be used



Options - 3

- CloseQuote=""
- OpenQuote=""
- These two allow specification of the delimiters for quoted strings
 - the fields are defined as length 10
 - this allows extended delimiters such as ``` to be used
- Indent=n
 - how many spaces to indent each new level in the output
 - where the indent is incremented/decremented is hard-coded in the Manipula code
 - you cannot influence where this happens unless you change the code



Options - 4

- `UniqueNameBasedOn=1|2|3`
 - these values are the values of the enum:
 - `fullname` "Full name",
 - `_localname` "Local name",
 - `localname_indexed` "Local name and array index"
 - this influences various items such as certain output formats and various reading orders in the code that analyzes the field level meta data
 - brought over from the SPSS/SAS/Stata sample and proved useful in Basic
- `ListComma=NONE|character`
 - Extended specific; allows the user to specify what character they want at the end of an item that is handled by a list placeholder
 - defined as `STRING[10]`; it can also be set to `NONE` which will set the auxfield to "" (an empty value); the default is `,` (a comma)



Options - 5

- `DispMsg = DISPLAY | MESSAGE | BOTH | NONE`
 - `DISPLAY` - display error and completed messages on the console and wait
 - `MESSAGE` - write error and completed messages to the message file
 - `BOTH` - do both of the above
 - `NONE` – do none of the above
- `ShowDone = DISPLAY | MESSAGE | BOTH | NONE`
 - `DISPLAY` - display Done at the end of the run on the console
 - `MESSAGE` - write Done at the end of the run to the message file
 - `BOTH` - do both of the above
 - `NONE` – do none of the above
- These settings were introduced to enable finer control over silent running



Options - 6

- Different systems expect different formats for date, time and datetime types
- These three options allow you to override the width of the date, time and datetime fields that the system returns
- Leaving them blank or setting them to ≤ 0 will take the system returned information
- DateSize=
- TimeSize=
- DateTimeSize=



Unit tests project

- Given the amount of code in this project and the complexity of some of it, considerable testing needs to be done after each change
- One way to check if any unwanted or unexpected changes have occurred is to isolate each routine, surround it with supporting code that sets up the input required for the routine, call the routine and then check the answer the routine gives back
- The unit tests project sets up a framework to do this and is a starting point for unit testing



Arrange, Act, Assert

- Each unit test is written using the Arrange, Act, Assert pattern
- Step 1 is to arrange all the artifacts that the routine being called will need
- Step 2 is to call the routine using the appropriate artifacts
- Step 3 is to assert the correctness or not of the result of the routine



Arrange, Act, Assert infrastructure - 1

- To enable an Arrange, Act, Assert approach to the unit tests for the Kami code, a (minimal) framework was created
- This infra is held in Assertions.incx in the KamiTests project
- There are two INPUTFILE definitions both of which take their definition from entries in Models.incx – AsciiMeta and dmRecordDescription
- There is a PRODEDURE which is used to output the test name and a passed/failed/unknown message to the MESSAGE file
- There are two FUNCTIONS to do comparison
 - one compares two INTEGER parameters
 - one compares two STRING parameters
 - the convention is that if the parameters are equal, the value returned is 1, otherwise the value returned is 0



Arrange, Act, Assert infrastructure - 2

- The unit tests controller includes the various files that it needs:
 - Models.incx Files.incx Auxfields.incx
 - FileUtils.incx General_routines.incx Extended_routines.incx
 - Basic_routines.incx Assertions.incx
- Two auxfields are defined:
 - gPassedTests : INTEGER // used to track the number of passed tests
 - aTotalTests : INTEGER // how many tests we currently have
- Various functions are defined that test specific routines in the codebase and add their result to gPassedTests
- The MANIPULATE section displays progress messages, calls functions, totals the number of tests, writes results to the message file & displays a summary at the end



Unit tests controller

```
aTotalTests := 0  
DISPLAY('Running unit tests for TokenCount')  
OutputInfo('Kt_TokenCount1 result: ', Kt_TokenCount1)  
...  
OutputInfo('Kt_TokenCount9 result: ', Kt_TokenCount9)  
aTotalTests := aTotalTests + 9
```

```
DISPLAY('Running unit tests for Token')  
OutputInfo('Kt_Token1 result: ', Kt_Token1)  
...  
OutputInfo('Kt_Token10 result: ', Kt_Token10)  
aTotalTests := aTotalTests + 10
```



Sample unit test

```
FUNCTION Kt_LocalName1: INTEGER
```

```
AUXFIELDS
```

```
  aResult1 : STRING
```

```
  aString1 : STRING
```

```
  aString2 : STRING
```

```
INSTRUCTIONS
```

```
  // Arrange
```

```
  aString1 := 'Fred.Arthru.Benny'
```

```
  aString2 := 'Benny'
```

```
  // Act
```

```
  aResult1 := LocalName(aString1)
```

```
  // Assert
```

```
  Result := Assert_String_Equals(aResult1,aString2)
```

```
  gPassedTests := gPassedTests + Result
```

```
ENDFUNCTION
```

LocalName is the name of a FUNCTION
in General_routines.incx

LocalName returns the last part of a
fully-qualified name



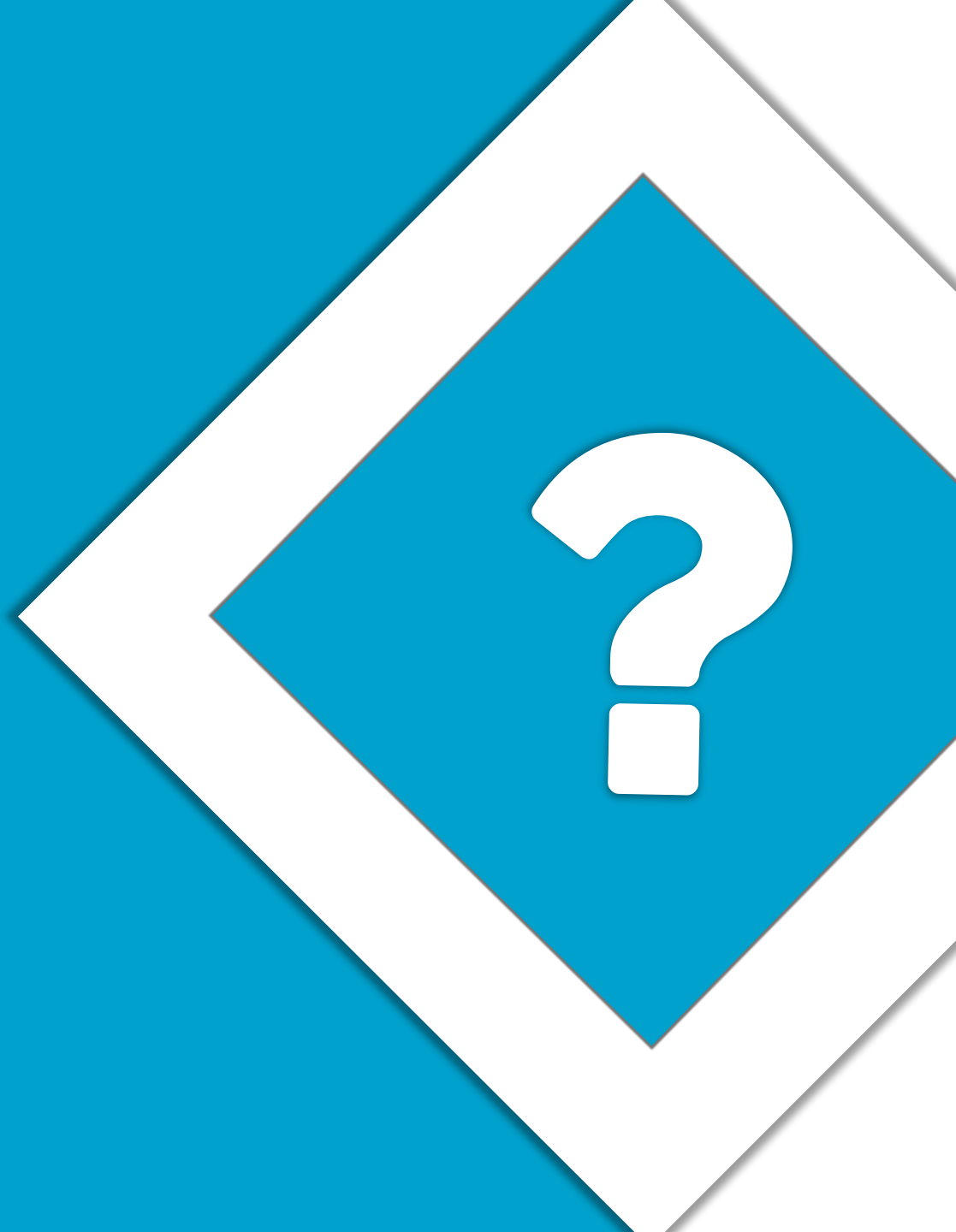
And finally...

Why is it called Kami?

Because I can't call it Cameleon...



Questions



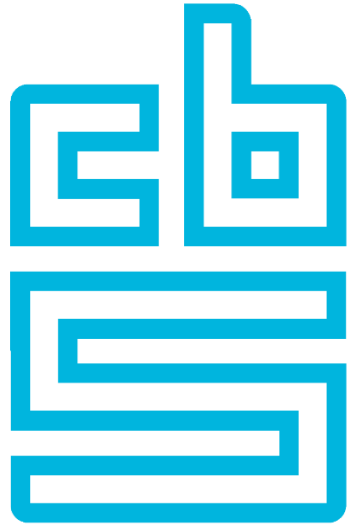
The CBS logo is a stylized white 'C' inside a blue square, which is itself inside a white diamond shape. The background is a solid blue color.

Thank you for your time

Support available at:

e-mail: Blaise Support -> blaise@cbs.nl

JIRA: portal -> <https://blaisesupport.cbs.nl>



Blaise

Gaining deeper understanding



www.blaise.com



blaise@cbs.nl



[@blaiseCBS](https://twitter.com/blaiseCBS)



[@Blaise5](https://www.youtube.com/@Blaise5)