GNUe Forms: A Developer's Introduction

A Guide to Programming with GNUe Forms

Version 0.4.0

Written by Jason Cater

Table of Contents

Introduction

This section briefly introduces the process of designing an application using GNUe Forms.
BLAH, BLAH, BLAH...

Before designing an application for Forms, the developer should be somewhat familiar with a few key concepts:

- *Database Design* - This guide does not delve into database design. It is assumed the developer can either create his own tables, or has an existing set of tables to work with.
- *Python Scripting* - GNUe uses Python for scripting/event support. Any level of serious applications programming will require some level of Python. There is a short section entitled "A Brief Introduction to Python" in this guide that can serve as a starting point.
- *XML* - GNUe extensively uses XML for its internal storage format. While it is possible to create GNUe applications via Designer without interacting with the XML formats, a good, solid understanding of XML basics would definitely be useful.

Structuring the Database

TODO

Designing the Form

TODO

Planning for Security

TODO

Basic Concepts

TODO

Data Sources

TODO

Blocks and Entries

TODO

Pages and Visual Elements

TODO

Designing for Multiple Architectures

TODO

Creating your First Form

In this chapter, we will create our first full-featured form -- a postal code lookup table.
[TODO: EXPAND]

Preliminary Steps

[TODO: INTRO]

First, we need to create a test table. Using your database of choice, create a table named `zipcodes`, with three appropriately sized fields, `zipcode`, `city`, and `state`. In PostgreSQL, SAP-DB, and others, the following statement will work:

```
create table zipcodes (
  zipcode varchar(5),
  city varchar(30),
  state varchar(2) );
```

For other databases, create a similarly structured table.

Next, if you have not done so yet, setup your connections.conf file to point to your database. Our examples will use a connection called `tutorial`. An example entry:

```
[tutorial]
comment = Tutorial Database
provider = psycopg
host = localhost
dbname = mydb
```

Of course, your entry will probably look different. This example is using the PostgreSQL `psycopg` driver, connecting to a database named `mydb` running on the local machine. See the *Forms Installation Guide* for information on the location and syntax of this file.

If you already have a connection for the database you will be using, simply add an `alias = tutorial` line to the appropriate section. Example:

```
[dev]
comment = Foobar Development
provider = psycopg
host = dbserver
dbname = mydb
alias = tutorial
```

We are going to create a form that looks something like:

```
01234567890123456789012345678901234567890
|---------+---------+---------+---------|

  Zip Code:   [___]

  City Name:  [_____________________]

  State:      [__]
```

The ruler is simply for our layout reference. We will come back to it later.

[TODO: EXPAND]

For the sake of completeness, each step involved in creating the form will be repeated, once using GNUe Designer, and once using hand-edited XML. GNUe Designer also has several feature-complete "Builder wizards" that can automatically build a form identical to the one we are describing. However, the point of this section is to learn the basics of form creation, and such a wizard isn't going to help accomplish that goal. [TODO: POINT TO A GUIDE ON WIZARDS?]

Creating the Empty Form

Forms are stored natively in an XML format. [TODO: EXPAND]

Given the layout we did above, we see that our form will be 40 characters wide and 7 lines high. For simplicity's sake, we are using a simple character based layout. In the future, forms will support other layout styles.

Via GNUe Designer:

By default, GNUe Designer will start up with an empty form. You may also create a new form by selecting `File | New | Form`.

[TODO: CHANGE THE TITLE AND SIZE OF THE FORM]

Via a Text Editor:

We need to create an XML shell for our form. Using your favorite text editor, create a file called firstform.gfd with a basic XML shell:

```
<?xml version="1.0"?>
<form title="My First Form" width="40" height="7">
</form>
```

The first line identifies this as an XML file conforming to XML version 1.0. This is needed in order to conform to XML standards.

The `<form>` tag denotes the start of our form definition. Our entire form definition will end up between the `<form ...>` and `</form>` lines.

Creating the Data Sources

Our sample form will use a single table - `zipcodes`. To access this table in Forms, we need to associate it with a datasource. We will call this datasource `MyDS`.

Via GNUe Designer:

From the menu, select `Tools | Insert | Datasource...` to run the datasource wizard. The first step of the datasource wizard asks for your connection. Select tutorial. This is the connection we setup in the first part of this section. Click `Next...` You may be asked to log in. Provide a valid database username and password.

Via a Text Editor:

After the `<form ...>` tag, add a `<datasource>` tag. It will need three attributes: `name`, `connection`, and `table`.

- `name`: All referencable objects in GNUe need a unique name. We will name our datasource `MyDS`.
- `connection`: This is the name of the connection we set up in the first part of this section. If you used our example entry, then this is called `tutorial`.
- `table`: This should be the name of the table we will be referencing. Once again, if you used our example table create script, this will be `zipcodes`.

We can now add our datasource tag:

```
<?xml version="1.0"?>
<form title="My First Form" width="40" height="7">

  <!-- A datasource for the tutorial.zipcodes table -->
  <datasource name="MyDS" connection="tutorial"
              table="zipcodes"/>

</form>
```

Creating the Pages and Blocks

For historical reasons, Forms contains its layout logic in units called *pages*. Only a single page is normally seen any given time by the end user. Our simple form will need one page. We will call it `MyPage`.

Blocks are the display equivalent of datasources. Since we are working with a single datasource, we will correspondingly have a single block. Since this is tied to the `zipcodes` table, we will call this block `ZipBlock`.

Via GNUe Designer:

An empty form created by GNUe Designer automatically has a single, empty page. If we were to add more pages, this could be accomplished by selecting `Tools | Insert | Page`. However, this example uses a single page, so we do not need to do anything for this step.

Via a Text Editor:

After our recently added `<datasource>` tag, we need a `<page>` tag. It will have a single attribute, `name`, that will be our name for the page object.

```
<?xml version="1.0"?>
<form title="My First Form" width="40" height="7">

  <!-- A datasource for the tutorial.zipcodes table -->
  <datasource name="MyDS" connection="tutorial"
              table="zipcodes"/>

  <!-- Start of our layout code -->
  <page name="MainPage">

    <!-- The base block for our zipcodes table -->
    <block name="ZipBlock" source="MyDS">

    </block>
  </page>
</form>
```

We will be adding objects to the page, so create both a beginning and ending tag.

Creating the Entries and Labels

Looking back at our earlier layout grid, we have three labels and three entries. Each label starts in column two and each entry starts in column 13. Each pair of label/entry skips a row, with the first pair being on row 1.

This gives us enough information to create our entries and labels.

Via GNUe Designer:

TODO

Via a Text Editor:

Add a `<label>` tag for each label with the three attributes `x`, `y`, and `text`. Also add an `<entry>` tag for each entry. An <entry> tag will have the `x`, `y`, `field`, and `width` attributes, with our `state` `<entry>` having a `case` attribute.

Label and Entry attributes:

✂ `x`, `y`: These are coordinates relative to the current page.

Label-specific attributes:

✂ `text`: This is the text to display as a label.

Entry-specific attributes:

✂ `field`: Set this to the name of the corresponding database field.

✂ `case`: This can be set to any one of `mixed`, `upper`, or `lower` to force case convention. For example, if set to `upper`, then all lower case input characters will be converted to uppercase. Note that this only applies to data input from the user. Setting this field will not convert existing data that is queried. The default value is `mixed`.

```
<?xml version="1.0"?>
<form title="My First Form" width="40" height="7">

  <!-- A datasource for the tutorial.zipcodes table -->
  <datasource name="MyDS" connection="tutorial"
              table="zipcodes"/>

  <!-- Start of our layout code -->
  <page name="MainPage">

    <!-- The base block for our zipcodes table -->
    <block name="ZipBlock" source="MyDS">

      <label x=2 y=1 text="Zip Code:"/>
      <label x=2 y=3 text="City Name:"/>
      <label x=2 y=5 text="State:"/>

      <entry x=13 y=1 field="zipcode" width=5/>
      <entry x=13 y=3 field="city" width=30/>
      <entry x=13 y=5 field="state" width=2
             case="upper"/>
    </block>
  </page>
</form>
```

The labels, since they are not bound to any data, can be a child of either the `<page>` or the `<block>` tags.

Running the Form

TODO

[TODO: Insert pic of Win32 form]

[TODO: Insert pic of GTK form]

[TODO: Insert pic of Curses form]

Where To Go Next

TODO

Understanding Data Sources

A Data Source links data to our form. Usually, a data source points to a table if using a relational database, or a data object if using an object database. A form can have several data sources if pulling data from multiple locations, or no data sources at all if the form does not reference outside data.

If a form does not have a data source, a *virtual data source* is created. The *commit*, *rollback*, and *query* functions do not serve a purpose against virtual data sources. This is particularly useful for *action forms* that simply cause actions to occur, but do not directly manipulate data.

Data Sources can be linked to each other in a *master/detail* fashion via a foreign key. In essence, each time the *master* data source changes, the *detail* data source is automatically requeried to bring up records related to the *master*. See the section on master/detail relationships for more information.

Table-Bound Data Sources

The most common data source is one that is bound to an individual table or view.

Static Data Sources

TODO

```
<datasource name="AvailDS" type="static">
  <staticset fields="id,descr">
    <staticsetrow>
      <staticsetfield name="id" value="A"/>
      <staticsetfield name="descr"
                      value="Available"/>
    </staticsetrow>
    <staticsetrow>
      <staticsetfield name="id" value="N"/>
      <staticsetfield name="descr"
                      value="Not Available"/>
    </staticsetrow>
    <staticsetrow>
      <staticsetfield name="id" value="B"/>
      <staticsetfield name="descr"
                      value="Backordered"/>
    </staticsetrow>
  </staticset>
</datasource>
```

TODO

Defining Conditions

Form's data sources support conditions. Conditions place restrictions on the records returned by a data source. For those familiar with SQL, a condition translates directly into a `WHERE` clause.

```
<datasource connection="test" table="reps">
  <condition>
    <or>
      <eq>
        <cfield name="active"/>
        <cconst value="Y"/>
      </eq>
      <gt>
        <cfield name="sales_ytd"/>
        <cconst value="0"/>
      </gt>
    </or>
  </condition>
</datasource>
```

In this example, we are basically only allowing records from the `reps` table where the representative is either active (`active = 'Y'`) or had sales this year.

Linking Data Sources via Master/Detail

TODO

Defining Master/Detail Data Sources

TODO

Master/Detail Considerations

TODO

Advanced Relationships

TODO

Master/Detail/Detail

TODO

Reverse Master/Detail

TODO

Understanding Events and Triggers

TODO

Form's Event Model

TODO

Named Triggers verses Embedded Triggers

TODO

The Trigger Namespace

Global Names

GNUe Forms supports the Python `global` construct, which can be used by the developer to define global variables and methods, or import modules globally. For example, assume the following code chunk is a form's `On-Startup` trigger:

```
##
## On-Startup [ Form]
##

# We want to give our other triggers
# access to these three objects.
global math, myfunc, DEBUG

# We will use the math module a lot
# in our other triggers
import math

# A handy function
def myfunc(n1,n2):
  return n1+n2

# Are we in DEBUG mode?
# Enquiring triggers want to know...
DEBUG = 1

# This is an example of a non-global name.
# Only our On-Startup trigger sees this.
test = 2
```

Because Forms executes `On-Startup` before any other triggers, all other triggers within this form can now see `math`, `myfunc`, and `DEBUG`.

For example, an `On-Change` trigger could now do:

```
##
## On-Change (MyEntry)
##
if DEBUG:
  print "Starting value: %s" % self.get()

computed = myfunc(self.get(), 12)

AnotherEntry.set(math.floor(computed))
```

Note that if another trigger wanted to globally change the values of `math`, `myfunc`, or `DEBUG`, they would also have to use the `global` construct. The following section of code would only change `DEBUG` for this single execution of `On-Change`:

```
##
## On-Change (MyEntry)
##
DEBUG = 0

# ... other code ...
```

The `other code` in this example would see `DEBUG` as being `0`, but once the trigger was completed, `DEBUG` would return to being `1` for all future triggers. Now, suppose the trigger had instead looked like:

```
##
## On-Change (MyEntry)
##
global DEBUG
DEBUG = 0

# ... other code ...
```

Now, this trigger and all future triggers will see `DEBUG` as `0`.

Form-level Triggers

A Form-level trigger is defined as an object that is activated at the form-level and is defined as a child of the form object.

There are currently two form-level triggers: `On-Startup` and `On-Exit`.

On-Startup

The `On-Startup` trigger is executed once during the lifetime of a Form's instance. This happens after all the objects have been initialized and initially populated.

The `On-Startup` trigger is useful for .....

On-Exit

The `On-Exit` trigger is executed when either the user or a trigger requests that a form closes.

Page-level Triggers

TODO

Pre-Focus-In

TODO

Post-Focus-In

TODO

Pre-Focus-Out

TODO

Post-Focus-Out

TODO

Block-level Triggers

TODO

Pre-Focus-In

TODO

Post-Focus-In

TODO

Pre-Focus-Out

TODO

Post-Focus-Out

TODO

Pre-Modify

TODO

Pre-Commit

TODO

Pre-Insert

TODO

Pre-Update

TODO

Pre-Delete

TODO

Entry-level Triggers

TODO

Pre-Focus-In

TODO

Post-Focus-In

TODO

Pre-Focus-Out

TODO

Post-Focus-Out

TODO

Post-Change

TODO

Buttons and Triggers

Buttons have a special relationship with triggers. A button is associated with a single trigger and is otherwise useless if not associated with this single trigger.

Buttons have an attribute, `trigger`, that points to a named trigger.

Working with Entries

TODO

Typecasting Fields

TODO

Default Values

TODO

Formatting Fields with Masks

[NOTE: Format Masks are not yet completely functional! This section reflects the intended support]

Forms supports two types of format masks: display masks and input masks. A display mask defines how the field data will be formatted for display. An input mask defines how the user will edit a field's value. Input mask elements are a subset of display mask elements -- in other words, all input masks can also be used as display masks, but not all display masks can be used as input masks.

Note: if first character of a format is '&', then rest of date defines a preset format (settable by developer? in gnue.conf or geas?).

e.g., in gnue.conf:

```
FormatDate_longdate = "A, b d, Y"
```

Then, in the client, the format string could be: `&longdate`

This allows reuse of common format masks throughout the application.

Formatting Numeric Fields

TODO

Formatting Date/Time Fields

Element	Input?	Description
\	Yes	Next character is a literal
a	Yes	Abbreviated weekday name (Sun..Sat)
A		Full weekday name (Sunday.. Saturday)

b	Yes	Abbreviated month name (Jan..Dec)
B		Full month name (January.. December)
c		Century (20,21)
d	Yes	Day of month, left padded with zeros (01..31)
D		Day of month, non-padded (1.. 31)
h	Yes	Hour (24-hour format), left padded with zeros (00..23)
H		Hour (24-hour format), non-padded (0..23)
g	Yes	Hour (12-hour format), left padded with zeros (01..12)
G		Hour (12-hour format), non-padded (1..12)
j	Yes	Day of year, left padded with zeros (001..366)
J		Day of year, non-padded (1.. 366)
m	Yes	Month, left padded with zeros (01..12)
M		Month, non-padded (1..12)
i	Yes	Minute, left padded with zeros (01..59)
I		Minute, non-padded (1..59)
p	Yes	am/pm designation (lowercase)

P		AM/PM designation (uppercase)
s	Yes	Seconds, left padded with zeros (00..59)
S		Seconds, non-padded (0..59)
u	Yes	Week number of year with Sunday as first day of week, left padded with zeros (01..52)
U		Week number of year with Sunday as first day of week, non-padded (1..52)
v	Yes	Week number of year with Monday as first day of week, left padded with zeros (01..52)
V		Week number of year with Monday as first day of week, non-padded (1..52)
w	Yes	Day of week with Sunday as first day of week (0=Sunday) (0..6)
W	Yes	Day of week with Monday as first day of week (0=Monday) (0..6)
y	Yes	Year (1900..2100)

Y	Yes	Year, using 2-digit notation (00..99) When used as an input mask, forms tries to reasonably guess the century. (TODO: Elaborate)

Predefined literals: "/-.:, "

Examples: 01/01/2001: "m/d/y" Friday, June 1, 2001: "A, b d, Y"

Formatting Text Fields

TODO

Dropdown Fields

TODO

Check boxes

TODO

Creating and Using Libraries

Overview

TODO

Integration with GNUe Tools

Running Reports from Forms

TODO

Running Forms from Navigator

TODO

Advanced Topics

This section describes advanced forms concepts. [TODO: EXPAND]

Runtime Parameters

Forms supports runtime parameters that can be passed to a form instance at startup. Parameters are mainly useful for specifying conditions in datasources, but can also be accessed via triggers. This allows the perceived behavior of a form to be altered only by passing a parameter.

A good example is a form designed to service two divisions of a company. While you could offer an opening dialog that asks the user which division he wants to work on, an alternative is to modify his startup script to tell the form which division he works with. This especially works well when the worker only belongs in one division and will never need access to any others.

Note that in the above example, though, parameters are not a good substitute for access security. This example would strictly be for convenience, not security.

Parameters must be defined, with a default value, using the `<parameter>` tag:

```
<form>
  <parameter name="division" default="101"/>
  ...
</form>
```

Once defined, parameters can be passed to forms in one of two ways. The first is via the command line. Parameters can be passed in the format `parameter=value` on the command line appearing after the name of the form. For example:

```
gnue-forms myform.gfd division=101
```

Alternately, if the form is being called from another form, the trigger would look like:

```
## Run "myform.gfd"
form.runForm( 'myform.gfd', { 'division' : 101 } )
```

That is, the parameters would be passed to runForm as a Python dictionary. Once passed to the form, parameters can be used in one of two ways: via trigger code or as a parameter to a datasource condition.

First, triggers can access parameters using the form.getParameter() method. This method takes one argument, the case-insensitive parameter name. It returns the requsested parameter, or the default value if no parameter was passed on startup.

```
## Get the "company" parameter
division = form.getParameter('division')
```

Conditions are also a good place for parameters. Take the following fragment:

```
<datasource name="dtsExsample" table="sales"
            connection="sales"/>
  <condition>
    <eq>
      <cfield name="division"/>
      <cparam name="division"/>
    </eq>
  </condition>
</datasource>
```

With this in place, whenever the table `sales` is queried, the only records returned are the ones where the field `division` matches the parameter `division`. Note that if this datasource will also be used for inserting new rows, a `Pre-Insert` trigger is needed to set the `division` field:

```
##
## Pre-Insert [ SalesBlock]
DivisionEntry.set(form.getParameter('division'))
```

External Python Modules

Python triggers have full access to your installed Python modules. For example, if your project needs the `twofish` cryptographic module, you can install it normally and do an `import twofish` in yourtriggers.

Alternately, GNUe's `gnue.conf` file supports an ImportPath directive. You can have this point to a directory containing your custom python modules.

Designing for Multiple Interfaces

A form definition, when designed within reasonable guidelines, can be run on a plethora of system architectures and a wide variety of user interfaces. By using the approach taken in this guide, most of your forms will, by default, run on a graphical workstation (X11, Windows, Mac), in a text-based session (telnet or ssh), or via a web browser (HTML). This section highlights a few key compatibility issues.

This list, while not exhaustive, should give you a good idea of common portability pitfalls. As with all things in GNUe, you will always have a choice on how to implement your application. GNUe is not about forcing rules on developers, but about providing viable options. There will be instances where the following suggestions simply are not feasible or practical. In any event, these are simply suggestions on getting the most out of Forms.

✂ **Images**
Do not make your application dependent on displayed images. It would be acceptable,

and appropriate, to display pictures for informational purposes. For example, when doing parts lookups, it would be appropriate to display a picture of the part for reference use. However, it would be normally be inappropriate to prevent the form from working if this image could not be displayed. (TODO: Better example?)

- **OS-specific trigger code**
  Python, the default trigger language, provides an extensive library of cross-platform functions. For example, it provides a library of file-access routines that work on all its supported platforms. This is really a broad category as trigger code has all the power of Python behind it.

- **Custom widgets**
  It is often tempting to use a new whiz-bang widget available on a certain platform/widget set. This will surely make your application hard to migrate to other platforms/interfaces, as well as restrict your ability to upgrade to a newer Forms version. Form's widget-set was carefully selected to be as multi-platform-friendly as possible, while still providing all the functionality most forms will need. If your application widgets are not supported by forms, there's a good chance that your form could be more functional with a slight rethinking of its design. Remember: a goal of Forms is to be usable on as many platforms as possible, not to exploit all the features of a particular platform.

A Brief Introduction to Python

While GNUe Forms will eventually support a plethora of scripting languages, the default, and best-supported, language will always be Python. Python is a .....

If you do not know Python, don't worry! Python is one of the simplest languages to pick up. .....

Once multi-language support is added, the developer will be able to write triggers in Python, Perl, Ruby, Scheme, or possibly even Basic. .....

While Python is easy to learn, this section assumes that you know at least one programming language. It is beyond the scope of this guide to cover basic programming concepts. There are several excellent Python tutorials for those beginning programming available on the Web. Go to http://www.python.org/doc/ for a listing of available tutorials. They have docs for every stage of python programming, from new-to-programming to seasoned veteran.

The Basics

The first thing most people notice about Python is its reliance on whitespace for grouping.

Variables and Expressions

```
x = 1
```

Control Structures

```
if x == 1:
  print "Yip"
```

```
for f in (1,2,3):
  print f
```

```
for f in range(4):
  print f
```

```
n = 1
while n < 10:
  n += 1
```

Tuples, Lists, and Dictionaries... oh, my!

TODO

Trigger Recipes

Over the coarse of writing a complex application, you will encounter a few situations where you will need a trigger to perform a common task. This section lists several

Timestamping a Record prior to a Commit

To automatically fill an entry with a timestamp retrieved from the database, you can use the datasource extension getTimestamp(). create a `Pre-Commit` trigger on that block. For example,

```
##
## Pre-Commit [ MyBlock]
##

self.MyTimeField.set(MyDS.extensions.getTimestamp())
```

This example assumes your entry is named `MyTimeField and` your datasource is called `MyDS`.

As noted elsewhere in this guide, `Pre-Commit` is run prior to saving changes to the database regardless of whether the record in question is being inserted, updated, or deleted. If you want to timestamp only new records, you can use the same code listed above, only inside a `Pre-Insert` trigger. Similarly, if you only want to timestamp modifications, you can use a `Pre-Update` trigger.

At this point, you may be asking why did we go through `MyDS` to get to a database timestamp. After all, `MyDS` corresponds to a table, not to a database. [TODO: Provide an explanation]

By using a timestamp retrieved from the database server, you do not have to worry about differences in the client machines' times. If you would prefer to have the client's time, you can use python's time module.

Auto-Populating an Entry from a Sequence

To automatically fill an entry with a value from a sequence, you can create a Pre-Insert trigger on that entry. For example,

```
##
## Pre-Insert [ MyEntry]
##

self.set(MyDS.extensions.getSequence("MySequence"))
```

This example assumes your entry is named `MyEntry`, your datasource is called `MyDS`, and the sequence name as stored in the database is `MySequence`. Note that `MyEntry` and `MyDS` are both names originating in your form, whereas `MySequence` is a name originating in your database.

At this point, you may be asking why did we go through `MyDS` to get to a database sequence. After all, `MyDS` corresponds to a table, not to a database. [TODO: Provide an explanation]

Appendix A: Trigger Hierarchy

Forms supports

Appendix B: Form Elements

TODO

Form Tags

form

No description provided

Attributes

Attribute	Values	Default	Description
height	*number*	`12`	The height of the object in text rows.
readonly	`Y`, `N`	`N`	No description provided
tabbed	`bottom`, `left`, `right`, `top`		Allows a form to convert it's pages as notebook tabs. Allowed values are `left`, `right`, `bottom`, `top`.
title	*text*	`Untitled Form`	The title of the form. Will be displayed on About Screen.
width	*number*	`40`	The width of the object in text columns.

Child Nodes

`database`, `datasource`, `import-datasource`, `import-page`, `import-trigger`, `menu`, `options`, `page`, `parameter`, `trigger`

Database Tags

database

No description provided

Attributes

Attribute	Values	Default	Description
name	*text*		No description provided
provider	*text*		No description provided
comment	*text*		No description provided
dbname	*text*		No description provided
host	*text*		No description provided
service	*text*		No description provided

Datasource Tags

datasource

No description provided

Attributes

Attribute	Values	Default	Description
name	*text*		No description provided
cache	*number*	5	No description provided
database	*text*		No description provided
detaillink	*text*		No description provided
distinct	Y, N	N	No description provided
explicitfields	*text*		No description provided
master	*text*		No description provided
masterlink	*text*		No description provided
order_by	*text*		No description provided
prequery	Y, N	N	No description provided
primarykey	*text*		No description provided
table	*text*		No description provided
type	*text*	object	No description provided

Child Nodes

condition, staticset

add

No description provided

Child Nodes

add, cconst, cfield, cparam, div, mul, sub

and

No description provided

Child Nodes

and, between, conditions, eq, ge, gt, le, like, lt, ne, negate, not, notbetween, notlike, or

between

No description provided

Child Nodes

add, cconst, cfield, cparam, div, mul, sub

cconst

No description provided

Attributes

Attribute	Values	Default	Description
value	*text*		No description provided

cfield

No description provided

Attributes

Attribute	Values	Default	Description
name	*text*		No description provided

condition

No description provided

Child Nodes

`and`, `between`, `eq`, `ge`, `gt`, `le`, `like`, `lt`, `ne`, `negate`, `not`, `notbetween`, `notlike`, `or`

cparam

No description provided

Attributes

Attribute	Values	Default	Description
name	*text*		No description provided

div

No description provided

Child Nodes

`add`, `cconst`, `cfield`, `cparam`, `div`, `mul`, `sub`

eq

No description provided

Child Nodes

`add`, `cconst`, `cfield`, `cparam`, `div`, `mul`, `sub`

ge

No description provided

Child Nodes

`add`, `cconst`, `cfield`, `cparam`, `div`, `mul`, `sub`

gt

No description provided

Child Nodes

`add`, `cconst`, `cfield`, `cparam`, `div`, `mul`, `sub`

le

No description provided

Child Nodes

`add`, `cconst`, `cfield`, `cparam`, `div`, `mul`, `sub`

like

No description provided

Child Nodes

`add`, `cconst`, `cfield`, `cparam`, `div`, `mul`, `sub`

lt

No description provided

Child Nodes

`add`, `cconst`, `cfield`, `cparam`, `div`, `mul`, `sub`

mul

No description provided

Child Nodes

`add`, `cconst`, `cfield`, `cparam`, `div`, `mul`, `sub`

ne

No description provided

Child Nodes

`add`, `cconst`, `cfield`, `cparam`, `div`, `mul`, `sub`

negate

No description provided

Child Nodes

`and`, `between`, `conditions`, `eq`, `ge`, `gt`, `le`, `like`, `lt`, `ne`, `negate`, `not`, `notbetween`, `notlike`, `or`

not

No description provided

Child Nodes

`and`, `between`, `conditions`, `eq`, `ge`, `gt`, `le`, `like`, `lt`, `ne`, `negate`, `not`, `notbetween`, `notlike`, `or`

notbetween

No description provided

Child Nodes

`add`, `cconst`, `cfield`, `cparam`, `div`, `mul`, `sub`

notlike

No description provided

Child Nodes

`add`, `cconst`, `cfield`, `cparam`, `div`, `mul`, `sub`

or

No description provided

Child Nodes

`and`, `between`, `conditions`, `eq`, `ge`, `gt`, `le`, `like`, `lt`, `ne`, `negate`, `not`, `notbetween`, `notlike`, `or`

staticset

No description provided

Attributes

Attribute	*Values*	*Default*	*Description*
fields	*text*		No description provided

Child Nodes

`staticsetrow`

staticsetfield

No description provided

Attributes

Attribute	Values	Default	Description
name	*text*		No description provided
value	*text*		No description provided

staticsetrow

No description provided

Child Nodes

`staticsetfield`

sub

No description provided

Child Nodes

`add`, `cconst`, `cfield`, `cparam`, `div`, `mul`, `sub`

Menu Tags

menu

No description provided

Attributes

Attribute	Values	Default	Description
name	*text*		No description provided
enabled	`Y`, `N`	`N`	No description provided
event	*text*		No description provided
label	*text*		No description provided
leader	*text*		No description provided
location	*text*		No description provided
trigger	*text*		No description provided
type	*text*		No description provided

Options Tags

options

No description provided

Child Nodes

`author`, `description`, `name`, `option`, `tip`, `title`, `version`, `width`

author

No description provided

Attributes

Attribute	Values	Default	Description
name	`author`	`author`	No description provided
value	*text*		No description provided

description

No description provided

Attributes

Attribute	Values	Default	Description
name	`description`	`description`	No description provided
value	*text*		No description provided

name

No description provided

Attributes

Attribute	Values	Default	Description
name	`name`	`name`	No description provided
value	*text*		No description provided

option

No description provided

Attributes

Attribute	Values	Default	Description
name	*text*		No description provided
value	*text*		No description provided

tip

No description provided

Attributes

Attribute	Values	Default	Description
name	`tip`	`tip`	No description provided
value	*text*		No description provided

version

No description provided

Attributes

Attribute	Values	Default	Description
name	`version`	`version`	No description provided
value	*text*		No description provided

Page Tags

page

No description provided

Attributes

Attribute	Values	Default	Description
caption	*text*		No description provided
name	*text*		A unique ID for the widget. This is only useful when importing pages from a library.

Child Nodes

`block`, `box`, `button`, `import-block`, `import-button`, `label`, `scrollbar`

block

No description provided

Attributes

Attribute	Values	Default	Description
name	*text*		A unique ID for the widget.The name of the widget. No blocks or datasources can share the same name without causing namespace collisions in user triggers.
datasource	*text*		The name of a datasource (defined in by a `<datasource>` tag.) that provides this block with it's data.
restrictDelete	`Y`, `N`	`N`	If set then the user will be unable to request that a record be deleted via the user interface.
restrictInsert	`Y`, `N`	`N`	If set then the user will be unable to request that new records be inserted into the block
rowSpacer	*number*		Adjusts the vertical gap of this number of rows between duplicated widgets. Serves the same purpose as some of the gap attributes on individual widgets.

Attribute	Values	Default	Description
rows	*number*		Any widgets inside the block will display this number of copies in a verticle column. Simulates a very crude grid entry system. Serves the same purpose as the visibleCount attribute on some widgets.
transparentBlock	`Y`, `N`	`N`	If set then the you can tab out of the block via next or previous field events. Makes navigation in mutliblock forms easier.

Child Nodes

`box`, `button`, `entry`, `import-button`, `import-entry`, `label`, `scrollbar`

box

No description provided

Attributes

Attribute	Values	Default	Description
height	*number*		The height of the box in text rows.
width	*number*		The width of the box in text columns.
x	*number*		The text column starting position of the widget. Based upon leftmost column of screen being 0.
y	*number*		The text row starting position of the widget. Based upon the top row of the screen being 0.
focusorder	*number*		No description provided
label	*text*		An optional text label that will be displayed on the border.
name	*text*		No description provided

Child Nodes

`button`, `entry`, `import-button`, `import-entry`, `scrollbar`

button

No description provided

Attributes

Attribute	Values	Default	Description
height	*number*		The height of the entry in text rows.
width	*number*		The width of the entry in text columns.
x	*number*		The text column starting position of the widget. Based upon leftmost column of screen being 0.

Attribute	Values	Default	Description
y	*number*		The text row starting position of the widget. Based upon the top row of the screen being 0.
focusorder	*number*		No description provided
label	*text*		The text that should appear on the button
name	*text*		A unique ID for the widget. Useful for importable buttons.
trigger	*text*		The name of a named trigger that this button will fire when pressed.

entry

No description provided

Attributes

Attribute	Values	Default	Description
name	*text*		The unique ID of the entry. Referenced in master/detail setups as well as triggers.
x	*number*		The text column starting position of the widget. Based upon leftmost column of screen being 0.
y	*number*		The text row starting position of the widget. Based upon the top row of the screen being 0.
case	`lower`, `mixed`, `upper`	`mixed`	No description provided
default	*text*		The default value for any new records created. If the field is visible the user can override the value.
displaymask	*text*		No description provided
editOnNull	`Y`, `N`	`N`	No description provided
field	*text*		The name of the field in the datasource to which this widget is tied.
fk_description	*text*		No description provided
fk_key	*text*		No description provided
fk_source	*text*		No description provided
focusorder	*number*		No description provided
formatmask	*text*		No description provided
height	*number*	`1`	The height of the entry in text rows.
hidden	`Y`, `N`	`N`	If defined the entry widget will not be displayed on the form. This is usefull for fields the user doesn't need to know about that you wish to update via triggers.

Attribute	Values	Default	Description
ignoreCaseOnQuery	`Y`, `N`	`N`	If defined the entry widget ignores the case of the information entered into the query mask.
inputmask	*text*		No description provided
max_length	*number*		The maximum number of characters the user is allowed to enter into the entry.
no_ltrim	`Y`, `N`	`N`	No description provided
no_rtrim	`Y`, `N`	`N`	No description provided
queryDefault	*text*		The form will be populated with this value automatically when a query is requested. If the field is visible the user can still override the value.
readonly	`Y`, `N`	`N`	It defined the user will be unable to alter the contents of this entry. Triggers can still alter the value.
required	`Y`, `N`	`N`	This object cannot have an empty value prior to a commit.
rowSpacer	*number*		No description provided
rows	*number*		No description provided
sloppyQuery	*text*		When set, whatever value the user enters for the query mask is rewritten with % between each character. Thus `example` would be queried as `%e%x%a%m%p%l%e%`
style	`checkbox`, `default`, `dropdown`, `label`	`default`	The style of entry widget requested. Currently either `text`, `label`, `checkbox`, or `dropdown`. To use `dropdown` you are required to use both the `fk_source`, `fk_key`, and `fk_description` attributes. The `label` style implies the `readonly` attribute.
typecast	`date`, `number`, `text`	`text`	The type of data the entry widget will accept. Possible values are `text`, `number`, `date`.
value	*text*		No description provided
width	*number*		The width of the entry in text columns.

label

No description provided

Attributes

Attribute	Values	Default	Description
text	*text*		The text to be displayed.

Attribute	Values	Default	Description
x	*number*		The text column starting position of the widget. Based upon leftmost column of screen being 0.
y	*number*		The text row starting position of the widget. Based upon the top row of the screen being 0.
alignment	`center`, `left`, `right`	`left`	The justification of the label. Can be one of the following: `left`, `right`, or `center`. Requires that the `width` attribute be set.
name	*text*		The unique ID of the label.
rowSpacer	*number*		Overriders the rowSpace setting defined at the block level.
rows	*number*		Overrides the rows setting defined at the block level.
width	*number*		The width of the label in text columns. Defaults to the width of the text. Only really useful when used with the `alignment` attribute.

scrollbar

No description provided

Attributes

Attribute	Values	Default	Description
height	*number*		The height of the box in text rows.
width	*number*		The width of the box in text columns.
x	*number*		The text column starting position of the widget. Based upon leftmost column of screen being 0.
y	*number*		The text row starting position of the widget. Based upon the top row of the screen being 0.

Parameter Tags

parameter

No description provided

Attributes

Attribute	Values	Default	Description
name	*text*		No description provided
default	*text*		No description provided
description	*text*		No description provided

Attribute	Values	Default	Description
required	Y, N	N	No description provided
type	*text*	char	No description provided

Trigger Tags

trigger

No description provided

Attributes

Attribute	Values	Default	Description
language	python	python	No description provided
name	*text*		No description provided
src	*text*		No description provided
type	*text*		No description provided

Import Tags

import-datasource

No description provided

Attributes

Attribute	Values	Default	Description
library	*text*		No description provided
name	*text*		No description provided
cache	*number*	5	No description provided
database	*text*		No description provided
detaillink	*text*		No description provided
distinct	Y, N	N	No description provided
explicitfields	*text*		No description provided
master	*text*		No description provided
masterlink	*text*		No description provided
order_by	*text*		No description provided
prequery	Y, N	N	No description provided
primarykey	*text*		No description provided
table	*text*		No description provided
type	*text*	object	No description provided

import-block

No description provided

Attributes

Attribute	Values	Default	Description
library	*text*		No description provided
name	*text*		A unique ID for the widget.The name of the widget. No blocks or datasources can share the same name without causing namespace collisions in user triggers.
datasource	*text*		The name of a datasource (defined in by a `<datasource>` tag.) that provides this block with it's data.
restrictDelete	`Y`, `N`	`N`	If set then the user will be unable to request that a record be deleted via the user interface.
restrictInsert	`Y`, `N`	`N`	If set then the user will be unable to request that new records be inserted into the block
rowSpacer	*number*		Adjusts the vertical gap of this number of rows between duplicated widgets. Serves the same purpose as some of the gap attributes on individual widgets.
rows	*number*		Any widgets inside the block will display this number of copies in a verticle column. Simulates a very crude grid entry system. Serves the same purpose as the visibleCount attribute on some widgets.
transparentBlock	`Y`, `N`	`N`	If set then the you can tab out of the block via next or previous field events. Makes navigation in mutliblock forms easier.

import-button

No description provided

Attributes

Attribute	Values	Default	Description
height	*number*		The height of the entry in text rows.
library	*text*		No description provided
width	*number*		The width of the entry in text columns.
x	*number*		The text column starting position of the widget. Based upon leftmost column of screen being 0.
y	*number*		The text row starting position of the widget. Based upon the top row of the screen being 0.
focusorder	*number*		No description provided
label	*text*		The text that should appear on the button

Attribute	Values	Default	Description
name	*text*		A unique ID for the widget. Useful for importable buttons.
trigger	*text*		The name of a named trigger that this button will fire when pressed.

import-entry

No description provided

Attributes

Attribute	Values	Default	Description
library	*text*		No description provided
name	*text*		The unique ID of the entry. Referenced in master/detail setups as well as triggers.
x	*number*		The text column starting position of the widget. Based upon leftmost column of screen being 0.
y	*number*		The text row starting position of the widget. Based upon the top row of the screen being 0.
case	`lower`, `mixed`, `upper`	`mixed`	No description provided
default	*text*		The default value for any new records created. If the field is visible the user can override the value.
displaymask	*text*		No description provided
editOnNull	`Y`, `N`	`N`	No description provided
field	*text*		The name of the field in the datasource to which this widget is tied.
fk_description	*text*		No description provided
fk_key	*text*		No description provided
fk_source	*text*		No description provided
focusorder	*number*		No description provided
formatmask	*text*		No description provided
height	*number*	`1`	The height of the entry in text rows.
hidden	`Y`, `N`	`N`	If defined the entry widget will not be displayed on the form. This is usefull for fields the user doesn't need to know about that you wish to update via triggers.
ignoreCaseOnQuery	`Y`, `N`	`N`	If defined the entry widget ignores the case of the information entered into the query mask.
inputmask	*text*		No description provided

Attribute	Values	Default	Description
max_length	*number*		The maximum number of characters the user is allowed to enter into the entry.
no_ltrim	`Y`, `N`	`N`	No description provided
no_rtrim	`Y`, `N`	`N`	No description provided
queryDefault	*text*		The form will be populated with this value automatically when a query is requested. If the field is visible the user can still override the value.
readonly	`Y`, `N`	`N`	It defined the user will be unable to alter the contents of this entry. Triggers can still alter the value.
required	`Y`, `N`	`N`	This object cannot have an empty value prior to a commit.
rowSpacer	*number*		No description provided
rows	*number*		No description provided
sloppyQuery	*text*		When set, whatever value the user enters for the query mask is rewritten with % between each character. Thus `example` would be queried as `%e%x%a%m%p%l%e%`
style	`checkbox`, `default`, `dropdown`, `label`	`default`	The style of entry widget requested. Currently either `text`, `label`, `checkbox`, or `dropdown`. To use `dropdown` you are required to use both the `fk_source`, `fk_key`, and `fk_description` attributes. The `label` style implies the `readonly` attribute.
typecast	`date`, `number`, `text`	`text`	The type of data the entry widget will accept. Possible values are `text`, `number`, `date`.
value	*text*		No description provided
width	*number*		The width of the entry in text columns.

import-page

No description provided

Attributes

Attribute	Values	Default	Description
library	*text*		No description provided
caption	*text*		No description provided
name	*text*		A unique ID for the widget. This is only useful when importing pages from a library.

import-trigger

No description provided

Attributes

Attribute	Values	Default	Description
library	*text*		No description provided
language	`python`	`python`	No description provided
name	*text*		No description provided
src	*text*		No description provided
type	*text*		No description provided

Appendix C: Form Objects

TODO

Form

getParameter()

Syntax:

`getParameter(`*parameter*`)`

Description:

Returns a runtime parameter, or the default value of such if the user did not pass the requested runtime parameter. See **Runtime Parameters** on page 25 for more information.

Example:

```
# Get the runtime parameter "company"
company = form.getParameter("company")
```

setParameter()

Syntax:

`setParameter(`*parameter*`,`*value*`)`

Description:

Changes the value of a runtime parameter. See **Runtime Parameters** on page 25 for more information.

Example:

```
# Set the runtime parameter "company" to "101"
form.setParameter("company","101")
```

getFocus()

Syntax:

`getFocus(`*object*`)`

Description:

Request that the current focus be given to `object`. If `object` is a block or a page, then focus will be given to the first navigable entry on that page or block. All appropriate `Pre-FocusOut`, `Pre-FocusIn`, `Post-FocusOut`, and `Post-FocusIn` triggers will be

executed. If `setFocus` is called on a non-navigable item (such as a label), the call is ignored and focus does not change.

Example:

```
# Request that MyEntry gets the current focus
form.setFocus(MyEntry)
```

setFocus()

Syntax:

`setFocus(`*object*`)`

Description:

Request that the current focus be given to `object`. If `object` is a block or a page, then focus will be given to the first navigable entry on that page or block. All appropriate `Pre-FocusOut`, `Pre-FocusIn`, `Post-FocusOut`, and `Post-FocusIn` triggers will be executed. If `setFocus` is called on a non-navigable item (such as a label), the call is ignored and focus does not change.

Example:

```
# Request that MyEntry gets the current focus
form.setFocus(MyEntry)
```

setStatusText()

Syntax:

`setStatusText(`*text*`)`

Description:

For user interfaces that support a status bar, or some textual equivalent, set the displayed text. For interfaces without a status bar equivalent, this function is meaningless.

Example:

```
# Tell the user how great they are
form.setStatusText("Dude, you are the best user ever!")
```

showMessage()

Syntax:

`showMessage(`*text*`)`

Description:

Description goes here.

Example:

```
# Code Sample
```

commit()

Syntax:

```
commit()
```

Description:

Description goes here.

Example:

```
# Code Sample
```

close()

Syntax:

```
close()
```

Description:

Description goes here.

Example:

```
# Exit the current form
form.close()
```

Datasource

createResultSet()

Syntax:

```
createResultSet([ conditions] , [ readOnly] )
```

Description:

Description goes here.

Example:

```
# Code Sample
```

simpleQuery()

Syntax:

`simpleQuery(`*dictionary*`)`

Description:

Description goes here.

Example:

```
# Code Sample
```

delete()

Syntax:

`delete()`

Description:

Description goes here.

Example:

```
# Code Sample
```

Block

clear()

Syntax:

`clear()`

Description:

Clears the current block with an empty result set.

Example:

```
# Clear out MyBlock
MyBlock.clear()
```

gotoRecord()

Syntax:

`gotoRecord(`*index*`)`

Description:

Move to the record indicated by `index`. If `index` is negative, then move relative to the last record. Records are numbered beginning with `0`.

Example:

```
# Go to the second record in this block
MyBlock.gotoRecord(1)

# Go to the last record in this block
MyBlock.gotoRecord(-1)
```

newRecord()

Syntax:

`newRecord()`

Description:

Inserts a new record immediately following the current record. This new record will then become the current record. The `On-NewRecord` trigger is executed for the newly created record and any default values are recorded.

Example:

```
# Code Sample
MyBlock.newRecord()
```

nextRecord()

Syntax:

`nextRecord()`

Description:

Navigate to the next record. If the block is currently on the last record, then this method returns `0` (false). Otherwise it returns `1` (true).

Example:

```
# Move to the next record
MyBlock.nextRecord()
```

prevRecord()

Syntax:

```
prevRecord()
```

Description:

Navigate to the previous record. If the block is currently on the first record, then this method returns `0` (false). Otherwise it returns `1` (true).

Example:

```
# Move to the previous record
MyBlock.prevRecord()
```

deleteRecord()

Syntax:

```
deleteRecord()
```

Description:

Mark the current record as *deleted*. On the next save, this record will be permanently removed.

Example:

```
# Code Sample
MyBlock.deleteRecord()
```

parent

Description:

This read-only property contains the parent container of this block. The parent container is usually a page.

Example:

```
# Get MyBlock's parent page
page = MyBlock.parent
```

Entry

allowedValues()

Syntax:

```
allowedValues()
```

Description:

Returns a tuple containing valid values for this entry. This call will only return a set when a `fk_source` has been specified for the entry.

Example:

```
# Code Sample
if 'Test' not in MyEntry.allowedValues():
  MyEntry.set(None)
```

autofillBySequence()

Syntax:

`autofillBySequence(`*sequence*`)`

Description:

Description goes here.

Example:

```
# Code Sample
```

isEmpty()

Syntax:

```
isEmpty()
```

Description:

Returns true if the current entry is considered empty. Empty is usually associated with a blank, or null, value.

Example:

```
# Set MyEntry to 0 if it has no other value.
if MyEntry.isEmpty():
  MyEntry.set(0)
```

set()

Syntax:

```
set(value)
```

Description:

Description goes here.

Example:

```
# Code Sample
```

get()

Syntax:

```
get()
```

Description:

Description goes here.

Example:

```
# Code Sample
```

resetForeignKey()

Syntax:

```
resetForeignKey()
```

Description:

Description goes here.

Example:

```
# Code Sample
```

parent

Description:

This read-only property contains the parent container of this entry. The parent container will typically be a block, unless container boxes are used.

Example:

```
# Get this entry's parent block
block = self.parent
```

readonly

Description:

Description goes here.

Example:

```
# Set MyEntry to be readonly if not already
if not MyEntry.readonly:
  MyEntry.readonly = 1
```

Appendix D: Data Objects

TODO

Result Set

xxxx()

Syntax:

`getParameter(`*parameter*`)`

Description:

Description....

Example:

```
# Get the runtime parameter "company"
company = form.getParameter("company")
```

Record Set

xxxx()

Syntax:

`getParameter(`*parameter*`)`

Description:

Description....

Example:

```
# Get the runtime parameter "company"
company = form.getParameter("company")
```

Alphabetical Index