GUS Developer's Guide

GUS Developer's Guide

Table of Contents

List of Tables

List of Examples

Chapter 1. Developing GUS Plugins

Steve Fischer <sfischer@pcbi.upenn.edu>

GUS Plugins

GUS plugins are Perl programs that load data into GUS. They are written using the Plugin API (the section called “The Plugin API”). You may use plugins that are bundled with the GUS distribution or you may write your own.

The standard GUS practice is to use only plugins, not straight SQL or bulk loading, to load the database. The reason is that plugins:

- track the data that is loaded
- copy any updated or deleted rows to "version" tables that store a history of the changes
- are known programs that can be scrutinized and used again
- have a standard documentation process so that they are easily understood
- use the Plugin API and so are easier to write than regular scripts.

Supported versus Community Plugins

The distribution of GUS comes with two types of plugins:

- *Supported* plugins:
  - are confirmed to work
  - are portable
  - are useful to sites other than the site that developed the plugin
  - meet the Plugin Standard described below
- *Community* plugins:
  - are contributed by the staff at CBIL and any other plugin developers
  - have not been reviewed with respect to the criteria for being supported

When you begin writing your plugin, use as a guideline or as a template an existing supported plugin. They are found in `$PROJECT_HOME/GUS/Supported/plugin/perl`.

The Plugin API

`Plugin.pm`: The Plugin Superclass

GUS plugins are subclasses of `GUS::PluginMgr::Plugin`. The public subroutines in `Plugin.pm` (private ones begin with an underscore) constitute the Plugin API. GUS also provides Perl objects for each table and view in the GUS schema. These are also part of the API. (the section called “The Plugin API”)

The plugin's package and @ISA statements

All plugins must declare their package, using Perl's `package` statement. The package name of a plugin is derived as follows:

`ProjectName``::`*`ComponentName`*`::Plugin::`*`PluginName`*

Plugins must also declare that they are subclasses of `Plugin.pm`, using Perl's `@ISA` array. The first lines of a plugin will look like this:

```
package GUS::Supported::Plugin::SubmitRow

@ISA = qw(GUS::PluginMgr::Plugin)
```

Plugin Initialization

Plugins are objects and so must have a constructor. This constructor is the `new()` method. The `new()` method has exactly two tasks to accomplish: constructing the object (and returning it), and initializing it. Construction of the object follows standard Perl practice. Initialization is handled by the `Plugin.pm` superclass method `initialize()`. the section called “The Plugin API” for details about that method.

Example 1.1. A Sample `new()` method

```
sub new {
    my ($class) = @_;
    my $self = {};

    bless($self,$class);

    $self->initialize({
        requiredDbVersion => 3.5,
        cvsRevision => '$Revision: 3009 $',
        name => ref($self),
        argsDeclaration => $argsDeclaration,
        documentation => $documentation
    });

    return $self;
}
```

The `$Revision: 3009 $` string is CVS or Subversion keyword. When the plugin is checked into source control, the repository substitutes the file's revision into that keyword. The keywords must be in single quotes to prevent Perl from interpreting `$Revision: 3009 $` as a variable.

Keeping your Plugin Current as GUS Changes

If you follow the pattern used by supported plugins, you will only ever need to change one line in the `new()` method. As you can probably tell, `initialize()` takes one argument, a reference to a hash that contains a set of parameter values. The one you will need to change is `requiredDbVersion`. As

the GUS schema evolves, you will need to review your plugin to make sure it is compatible with the latest version of GUS, upgrading it if not. When it is compatible with the new version of GUS, update `requiredDbVersion` to that version of GUS.

Declaring the plugin's command line arguments

In the example above (Example 1.1, "A Sample `new()` method"), the line

```
argsDeclaration => $argsDeclaration,
```

provides to the `initialization()` method a reference to an array, `$argsDeclaration`, that declares what command line arguments the plugin will offer. When you look at a supported plugin you will see the `$argsDeclaration` variable being set like this:

Example 1.2. Defining Command Line Arguments

```
my $argsDeclaration = [
   tableNameArg({name  => 'tablename',
                 descr => 'Table to submit to, eg, Core::UserInfo',
                 reqd  => 1,
                 constraintFunc=> undef,
                 isList =>0,
   }),

   stringArg({name  => 'attrlist',
              descr => 'List of attributes to update (comma delimited)',
              reqd  => 1,
              constraintFunc => undef,
              isList = >1,
  }),

  enumArg({name  => 'type',
           descr => 'Dimension of attributes (comma delimited)',
           reqd  => 1,
           constraintFunc => undef,
           enum => "one, two, three",
           isList => 1,
  }),

  fileArg({name  => 'matrixFile',
           descr => 'File containing weight matrix',
           reqd => 1,
           constraintFunc=> \&checkFileFormat,
           mustExist=>0,
           isList=>0,
  }),
];
```

If you look carefully at the list above you will notice that each element of it is a call to a method such as `stringArg()`. These are methods of `Plugin.pm` and they all return subclasses of `GUS::PluginMgr::Args::Arg`. In the case of `stringArg()`, it returns `GUS::PluginMgr::Args::StringArg`. All you really need to know is that there are a set of methods available for you to use when declaring your command line arguments. That is, the `argsDeclaration` parameter of the `initialize()` method expects a list of `Arg` objects. You can learn about them in detail in the Plugin API (the section called "The Plugin API")

The `Arg` objects are very powerful. They parse the command line, validate the input, handle list values,

deal with optional arguments and default values and provide for documentation of the arguments. There are two ways the `Arg` objects validate the input. First, it applies its standard validation. For example, a `FileArg` confirms that the input is a file, and throws an error otherwise. Second, if you provide a `constraintFunc`, it will run that as well, throwing an error if the plugin value violates the constraints.

Declaring the Plugin's Documentation

In a way that parallels the declaration of command line arguments, the initialize method also expects a reference to a hash that provides standardized fields that document the plugin: (Example 1.1, “A Sample `new()` method”)

```
documentation => $documentation,
```

Here is a code snippet that demonstrates the standard way `$documentation` is set:

Example 1.3. Defining Plugin Documentation

```
my $purposeBrief = <<PURPOSE_BRIEF;
Load blast results from a condensed file format into the DoTS.Similarity table.
PURPOSE_BRIEF

my $purpose = <<PLUGIN_PURPOSE;
Load a set of BLAST similarities from a file in the form generated by the blastSim
PLUGIN_PURPOSE

my $tablesAffected =
    [ ['DoTS::Similarity', 'One row per similarity to a subject'],
      ['DoTS::SimilaritySpan', 'One row per similarity span (HSP)'],
    ];

my $tablesDependedOn =
    [
    ];

my $howToRestart = <<PLUGIN_RESTART;
Use the restartAlgInvs argument to provide a list of algorithm_invocation_ids that
previous runs of loading these similarities. The algorithm_invocation_id of a run
plugin is logged to stderr. If you don't have that information for a previous run
you will have to poke around in the Core.AlgorithmInvocation table and others to f
runs and their algorithm_invocation_ids.
PLUGIN_RESTART

my $failureCases = <<PLUGIN_FAILURE_CASES;
PLUGIN_FAILURE_CASES

my $notes = <<PLUGIN_NOTES;
The definition lines of the sequences involved in the BLAST (both query and subjec
begin with the na_sequence_ids of those sequences. The standard way to achieve tha
first load the sequences into GUS, using the InsertFastaSequences plugin, and then
extract them into a file with the dumpSequencesFromTable.pl command. That command
the na_sequence_id of the sequence as the first thing in the definition line.
PLUGIN_NOTES

my $documentation = { purpose=>$purpose,
                      purposeBrief=>$purposeBrief,
                      tablesAffected=>$tablesAffected,
                      tablesDependedOn=>$tablesDependedOn,
                      howToRestart=>$howToRestart,
                      failureCases=>$failureCases,
```

```
 notes=>$notes
};
```

When you look at this example, you will see that a bunch of variables, such as `$purposeBrief` and `$tablesAffected`, are being set. They are used as values of the hash called `$documentation`. `$documentation` is in turn passed as a value to the `initialize()` method. You will also notice that Perl's HEREDOC syntax is used. The setting of the variables begins with, eg, `<<PLUGIN_PURPOSE` and ends with, eg, `PLUGIN_PURPOSE`. This is Perl's way of allowing you to create paragraph-style strings without worrying about quoting or metacharacters such as `\n`.

The documentation is shown to the user when he or she uses the `help` flag, or when he or she makes a command line error.

The documentation is formatted using Perl's documentation generation facility, pod. This means that you can include simple pod directives in your documentation to, say, emphasize a word. Run the command `perldoc perlpod` for more information

The `run()` Method

Plugins are run by a command called `ga` (which stands for "GUS application"). `ga` constructs the plugin (by calling its `new()` method) and then runs the plugin by calling its `run()` method.

The purpose of the `run()` method is to provide at a glance the structure of the plugin. It should be very concise and under no circumstances be longer than one screen. A good practice, when reasonable, is for the `run()` method to call high level methods that return the objects to be submitted to the database, and then to submit them in the `run()` method. This way, a reader of the `run()` method will know just what is being written to the database, which is the main purpose of a plugin.

The `run()` method is expected to return a string describing the result of running the plugin. An example would be "`inserted 3432 sequences`".

The Pointer Cache

The pointer cache is a somewhat infamous component of the GUS object layer. It is a memory management facility that was designed to steer around poor garbage collection in Perl (in 2000). Whether or not is still needed is another matter because it is part of the object layer for now. The pointer cache is a way for the plugin to re-use objects that have been allocated but are no longer in active use. Because Perl was not properly garbage collecting objects when they were no longer referred to, the memory footprint of plugins was getting huge.

As a plugin developer what you need to know is that at points in your code where you no longer need any of the GUS objects that you have created (typically at the bottom of your outermost loop, you should call the `Plugin.pm` method `undefPointerCache()`. This method clears out the cache.

If the default capacity (10000) is not enough to hold all the objects you are creating in one cycle through your logic, you can augment its size with the `Plugin.pm` method `setPointerCacheSize()`.

The Plugin Standard

Portability

A supported plugin must be useful to sites other than the site that developed it. It also must run at other sites without modification.

Plugin Naming

- Plugin names begin with one of four verbs:
  - *insert* if the plugin inserts only
  - *delete* if the plugin deletes only
  - *update* if the plugin updates only
  - *load* if the plugin does any two or more of insert, delete or update
- Plugin names are concise
  - for example, a plugin named InsertNewSequences is not concise because Insert and New are redundant
- Plugin names are precise
  - for example, a plugin named InsertData is way too general. The name should reflect the type of data inserted
  - if a Plugin expects exactly one file type, that file type should be in the name. For example, InsertFastaSequences.
- Plugin names are accurate
  - for example, a plugin named InsertExternalSequences is inaccurate if it can also insert internally generated sequences. A better name would be InsertSequences.

GUS Primary Keys

Plugins never directly use (hard-code) GUS primary keys, either in the body of the code or for command line argument values. Instead they use semantically meaningful alternate keys. The reason that plugins cannot use primary keys in their code is that doing so makes the plugin site specific, not portable. The reason they cannot use primary keys as values in their command line arguments is that plugins are often incorporated as steps in a pipeline (using the GUS Pipeline API described elsewhere). The pipelines should be semantically transparent so that people both on site and externally who look at the pipeline will understand it.

Application Specific Tables

Some sites augment GUS with their own application specific tables. These are not permitted in supported plugins.

Command Line Arguments

- The name of the argument should be concise and precise
- The Plugin API provides a means for you to declare arguments of different types, such integers, strings and files (the section called “Declaring the plugin's command line arguments”). Use the most appropriate type. For example, don't use a string for a file argument.
- Use camel caps (eg matrixFile) not underscores (eg matrix_file) in the names of the arguments.

Documentation

The Plugin API provides a means for you to document the plugin and its arguments. Be thorough in your documentation. the section called “Declaring the Plugin's Documentation”

Use of GUS Objects

The GUS object layer assists in writing clean plugin code. The guidelines for their use are:

- When writing data to the database, use GUS objects when possible. Avoid using SQL directly.
- When forming a relationship between two objects, use the `setParent()` or `setChildren()` method. Do not explicitly set the foreign keys of the objects.

Database Access

The GUS objects are good at writing data to the database. That is because they allow you to build up a tree structure of objects and then to simply submit the root. However they are not as useful at reading the database. You can only read one object at a time (more on this in the Guide to GUS Objects). For this reason, you will need to use SQL to efficiently read data from the database as needed by your plugin.

This is how a typical database access looks:

Example 1.4. Typical Database Access

```
my $sql =
  "SELECT $self->{primaryKeyColumn}, $self->{termColumn}
   FROM $self->{table}";

my $queryHandle = $self->getQueryHandle();
my $statementHandle = $queryHandle->prepareAndExecute($sql);

my %vocabFromDb;

while (my ($primaryKey, $term) = $sth->fetchrow_array()) {
    $vocabFromDb{$term} = $primaryKey;
}
```

The SQL is formatted on multiple lines for clarity (Perl allows this), and the SQL keywords are upper case. The Plugin API provides a method to easily get a query handle, returning a `GUS::ObjRelP::DbiDbHandle`. That object provides an easy-to-use method that prepares and executes the SQL.

Logging

The Plugin API offers a set of logging methods. They print to standard error. Use these and no other means of writing out logging messages.

Standard Output

Do not write to standard output. If your plugin generates data (such as a list of IDs already loaded, for restart) write it to a file.

Commenting

Less is more with commenting. Comment only the non-obvious. For example, do not comment a method called `getSize()` with a comment `# gets the size`. Most methods should need no commenting, as they should be self-explanatory. In many cases, if you find that you need to comment because something non-obvious needs explaining, that is a red flag indicating that your code might need simplification.

Handling Errors

There is only one permissible way to handle errors: call `die()`. Never log errors or write them to standard error or standard out. Doing that masks the error (the logs are not read reliably) so that what is really happening is the plugin is failing silently. Causing the plugin to die forces the user of the plugin or its developer to fix the problem.

When you call die, give it an informative message, including the values of the suspicious variables. Surround the variables in single quotes so that white space errors will be apparent. Provide enough information so that the user can track down the source of the problem in the input files.

If you would like your program to continue past errors, then dedicate a file or directory which will house describing the errors. The user will know that he or she must look there for a list of inputs that caused problems. Typically you use this strategy if you expect the input to be huge, and don't want to abort it because of a few errors. You may want to include as a command line argument the number of errors a user will tolerate before giving up and just aborting.

Failure Recovery and Restart

Plugins abort. They do so for many reasons. When they do, the user must be able to recover from the failure, one way or another.

A few strategies you could adopt are:

- If the plugin is inserting data (rather than inserting and updating) the plugin can check if an object that is about to be written to the database is already there. If so, it can skip that object. Because this checking will slow the plugin down, the plugin should offer a `restart` flag on the command line that turns that check on.
- If the plugin is updating it can include a command line argument that takes a list of `row_alg_invocation_id`s, one per each run of the plugin with this dataset. (Each table in GUS has a `row_alg_invocation_id` column to store the identifier of the particular run of a plugin that put data there. This is part of the automatic tracking that plugins do.) The plugin can take the same approach as the previous strategy, but, must additionally check that the object has one of the provided `row_alg_invocation_id`s.
- The plugin can store in dedicated file the identifiers of the objects it has already loaded. In this case, the plugin should offer a command line argument to ask for the name of the file.

Opening Files

A very common error is to open files without dying if the open fails. The proper way to open a file is like this:

Example 1.5. Properly Opening a File

```
open(FILE, $myFile) || die "could not open file '$myFile'\n");
```

Caching to Minimize Database Access

One of the most time consuming operations in a plugin is accessing the database. The typical flow of a plugin is that it reads the input and as it goes it constructs and submits GUS objects to the database. Some plugins additionally need to read data from the database to do their work. While it is often impossible to avoid writing to the database with each new input value, it is often possible to avoid reading it.

If most of the values of a table (or tables) will be needed then the plugin should read the table (or tables) outside the loop that processes the input. It should store the values in a hash keyed on a primary or alternate key. Storing multiple megabytes of data this way in memory should not be a problem. Gigabytes may well be a problem.

If only a few values from the table will be needed then an alternative caching strategy may be appropriate. Wrap the access to the values in a getter method, such as `getGeneType()`. This method stores values it gets in a hash. When the method is called, it first looks in the hash for the value. If the hash does not have it, then the method reads the database and stores the value in the hash to optimize future accesses.

Regular Expressions

Complicated regular expressions should be accompanied by a comment line that shows what the input string looks like. It is otherwise often very difficult to figure out what the regular expression is doing. Long regular expressions should be split into multiple lines with embedded whitespace and comments using the `/x` modifier. See the "Readability" section of Maintaining Regular Expressions [http://www.perl.com/pub/a/2004/01/16/regexps.html]

Variable and Method Names

Choosing good names for your variables and methods makes your code much more understandable. To make your code clear:

- Variable and method names should start with a lower case letter.
- Use "camel caps" (`$sequenceLength`) for variable names and method names, not underscores (`$sequence_length`).
- Variable names should be named after the type of data they hold (unless there are more than one variable for a given type, in which case they are qualified). For example a good name for a sequence would be `$sequence`
- In plugins, there are typically:
  - strings parsed from the input
  - objects created from the input (if you are using an object based parser such as Bioperl)
  - GUS object layer objects
- Input objects or strings should be named with 'input' as a prefix. For example: `$inputSequence`
- Object layer objects are named for their type, for example `$NASequence`

- Method names should be self-explanatory. A bad method name would be `process()` (what is being processed?). Don't "save keystrokes" with short names. If being self-explanatory requires using a long name, then use a long name.

Methods

Use "structured programming" when you create your methods:

- No method should ever be longer than one screen. If it is, refactor part of into its own method.
- Never repeat code. Repeated code must be in a method.

Some methods in the API are marked as deprecated. Do not use them. They are for backward compatibility only.

Syntax

- Use C and Java like syntax. Do not use weird Perl specific syntax.
- Indenting must be spaces not tabs. Two or four spaces are acceptable
- Use `$self` to refer to the object itself
- Declare method arguments using this syntax:

  ```
  my ($self, $sequence, $length) = @_;.
  ```

  Do not use `shift`

Application Specific Controlled Vocabularies

A controlled vocabulary (CV) is a restricted set of terms that are allowed values for a data type. They may be simple lists or they may be complex trees, graphs or ontologies. In GUS the CVs fall into two categories: standard CVs such as the Gene Ontology, and small application specific CVs such as ReviewStatus.

The complete list of application specific CVs in the GUS 3.5 schema is:

- DoTS.BlatAlignmentQuality
- DoTS.GOAssociationInstanceLOE
- DoTS.GeneInstanceCategory
- DoTS.InteractionType
- DoTS.MotifRejectionReason
- DoTS.ProteinCategory
- DoTS.ProteinInstanceCategory

- DoTS.ProteinProteinCategory
- DoTS.ProteinPropertyType
- DoTS.RNACategory
- DoTS.RNAInstanceCategory
- DoTS.RNARNACategory
- DoTS.RepeatType
- SRes.BibRefType
- SRes.ReviewStatus

Acquiring a standard CV typically involves downloading files from the CV provider and running a plugin to load it.

Application specific CVs are handled by the plugin that will use the CV. For example, a plugin that inserts bibliographic references will use the SRes.BibRefType CV. It is these plugins that are responsible for making sure that the CV they want to use is in the database.

Plugins that use CVs fall into two categories:

1. those that hard code the CV
2. those that do not hard code the CV, but, rather, get it from the input

In case 1, the plugin hard codes the CV in the Perl code.

In case 2, the plugin hard codes only a default. It also offers an optional command line argument that takes a file that contains the CV. If the user of the plugin determines that the input has an different CV than the default, the user will provide such a file.

In both cases, the plugin reads the table in GUS that contains the CV and compares it to the CV it expects to use. If the expected vocab is not found, the plugin updates the table.

Assigning an External Database Release Id

GUS is a data warehouse so it is very common for plugins to load into GUS data from another source. Whether the source is external or in-house, tracking its origin is often required. The tables in GUS that handle this are SRes.ExternalDatabase and SRes.ExternalDatabaseRelease. The former describes the database, eg, PFam, and the latter describes the particular release of the database that is being loaded, eg, 1.0.0. The data loaded will have a foreign key to the database release, which in turn has a foreign key to the database.

In order to create that relationship, the plugin must know the primary key of the external database release. To accomplish this, the plugin takes as command line arguments the name of the database and its release. It does not take the primary key of the external database release (that violates the plugin standard). The plugin passes that information to the API subroutine `getExtDbRlsId($dbName, $dbVersion)`.

If the plugin is inserting the dataset as opposed to updating it, create new entries for the database and the release by using the plugins `GUS::Supported::Plugin::InsertExternalDatabase` and `GUS::Supported::Plugin::InsertExternalDatabaseRls`.

Chapter 2. Extending the Schema

Michael Saffitz <msaffitz@pcbi.upenn.edu>

The GUS Schema may be extended by adding new columns to existing tables, or adding new tables and views. For the time being, adding new Schemata to GUS is not supported.

Important

Extensions to the GUS may interfere with your ability to upgrade to future releases of GUS.

Creating New Objects in the Database

The first step to extending the Schema is to create the objects within the database.

Adding New Columns to Existing Tables

When adding new columns to existing tables, it is important to maintain the existing order of the columns, and only add new columns between the existing columns and the housekeeping columns (eg before the `modification_date` column). For this reason, it will likely be necessary to rename the existing table; create the modified table; and then migrate the date from the existing table to the newly created table. As you perform this process, you should ensure that all constraints (including both "incoming" and "outgoing" foreign key constraints and primary key constraints) and that indexes on the original table are created and applied on the newly created table.

Adding New Tables

When creating new tables, it is important to include all housekeeping columns at the "end" of the definition in the proper order. All new tables should have a corresponding sequence created, with the naming convention of: *`TableSchemaName.TableName`*`_SQ` . All new tables must have a single column primary key constraint defined.

Adding New Views

Only views created as "subclass" views against an implementation table are supported. When creating new views, it is important to include all superclass columns in the view definition, including the housekeeping columns. Proper column ordering should be observed in the views.

Updating GUS Version Objects

If you've changed an existing GUS table, or wish to have GUS audit changes to your new tables and views, you must make the corresponding changes and/or additions to the version ("ver") tables and views.

Updating `Core.TableInfo`

GUS stores metadata for all tables and views in the `Core.TableInfo` table. Whenever you create a new table or view, you must add a corresponding row in this table. The column descriptions are:

Table 2.1. `Core.TableInfo` Description

Column Name	Description
table_id	The ID of this row, provided by the com Core.TableInfo_SQ sequence.
name	The name of the table. The case used here will be used at object-generation time.
table_type	Standard or Version, depending on whether this is a normal table or a version table
primary_key_column	The name of the primary key column
database_id	The id of the schema, found in Core.DatabaseInfo
is_versioned	1 if the table has a corresponding version table. 0 otherwise.
is_view	1 if the "table" is a subclass view. 0 otherwise.
view_on_table_id	If this "table" is a subclass view, the table id of the implementation table that this view is against.
superclass_table_id	If this "table" is a subclass view, the table id of the superclass view.
is_updatable	1 if the table is read/write, 0 if it is read only.

Rebuilding Objects

After you have completed the steps above, you must rebuild your objects. First, use the command below to signal that your table definitions have changed:

```
$ touch $PROJECT_HOME/Schema/gus_schema.xml
```

Then, reinstall GUS:

```
$ build GUS install -append
```