General Coding Standards

Rick Cox

rick@rescomp.berkeley.edu

A description of general standards for all code generated by ResComp employees (including non-programmers), intended to make maintaince, reuse, upgrades, and trainig possible.

Table of Contents

Introduction

Coding standards are in place to guide development of ResComp code based on past experiences and general computer science best-practices. That said, this document is not the end-all/be-all of coding. See the excellent Perl books, example code, and ask programmers for more guidance. Also see other programmer documentation, including the CGI-HOWTO, Perldoc2 Standard, and recent design docs.

These standards are not meant to be imposing, simply to formalize our code enough that reusing and maintenance are easier.

General Standards

This should be better organized, but instead, it's totally random:

- No long functions (break long functions into subroutines).
- Comment every routine describing what it does (not how) and what the assumptions/invariants are.
- Write comments before code - if you can't describe what the function is doing, you won't be able to write it either.
- Live scripts are more fun.

Perl Standards

The most important two perl coding standards are: always run using the `-w` to perl, and always

```
use strict;
```

at the top of any perl file (module or script). The `Prototype` module should also be used by every perl file.

The typical header block of a script would thus be:

```
#!/usr/bin/perl -w

###
# @script Script that does nothing
# @author Name <email>
# @version $Id: Coding-Standards.sgml,v 1.4 2001/06/04 18:32:37 rick Exp $
##/

use strict;
use Prototype;
...
```

Other Perl standards include:

- No global variables.
- No database access from a script or CGI.
- Use object-oriented Perl where possible (it avoids the long function names and aids readability).

Perl Module Standards

A function in a module should never print to standard out. Instead, it should return text which the calling script can print when it wants.

Constants should be placed either inside the module, if they will not be used by other modules, or in a Constants.pm module (such as Rescomp::RFC::Constants). Placing constants besides the names of database tables in Rescomp::RFC::Tables is deprecated (though all table names should still be placed there).

Error catching, getting to know eval(), and Error Messages

In perl, make use of an eval() block to catch errors, and exit with appropriate messages.

Example: The following example is directly related to Perl's DBI.pm and Perl

```
$dbh = some_db_handle;
$sqlcmd = some_sql_statement;
eval{
  $sth = $dbh-
>prepare($sqlcmd);  #checks the statement for validity and sytanx errors.
  $sth->execute;                 #executes the sql statement.
};                               #don't forget the semi-colon.

if($@){  #if there is an error in the eval statement, $@ is defined, other-
wise eval
         #guarantees that it will be undefined.
  warn "some error message"; #for format of error message, see below.
  RETURN_DB_HANDLE;          #always do this
  return SOME_VALUE;
}
```

At all phases of error catching, a message should be printed to STDERR this way:

```
warn _SYSTEM_ERROR_MESSAGE_
```

This prints out to standard error `'ERROR_MESSAGE at _filename_ line _line_num_'`

Example: When you make a database call, something may go wrong or someone has changed the schema somehow and what was once a proper call is no longer one. Be prepared to catch this error and print the proper error message. For example code, see the above example.

Database Standards

In designing database applications, keep in mind portability. This means not taking advantage of lots of advanced features of the database, such as procedural functions, but will make everyone's life alot easier if we change databases (we did already once, and will probably do so again shortly).

Schemas

Schemas should be placed in `.sql` files under `cvs:/docs/schemas/`, and kept upto date.

Perl Specifics

Rescomp::RFC::DB is a light-weight wrapper for the perl DBI interface. It provides for the use of shared handles, as well as handling the details of connecting to whatever database we are using (e.g. Oracle on cube vs. MySQl on the ccservers vs. possible Postgres in the future). Most of the functions you'll actually use on the handles are DBI functions.

Scripts vs. Modules

All SQL should be contained in functions in .pm files (this really is important, don't ignore it, or things will break). All a script should do is init the db (see Shared Handles below), and decide whether to commit or rollback (always do that explicitly). Functions in perl modules that do database access should usually not handle errors, instead, use Perldoc2 to mark them as `@throws db error unmodified`. The user of that function will then need to use an `eval` around the module function to catch errors. See `Rescomp::RFC::Example` for a complete example of DB usage in a module.

Shared Handles

It takes a long time to create a connection to the database, and most of our scripts do many seperate database interactions. Realizing this, and that a single connection can be reused, DB.pm was created. Calling

```
use Rescomp::RFC::DB;
&Rescomp::RFC::DB::init();
```

will open this shared database handle. Any calls to `$dbh = &Rescomp::RFC::DB::get_handle()` thereafter will use the shared handle, saving a significant overhead.

The catch is that the handle must be explicitly returned prior to script exit (normal or on an error). This is done by `&Rescomp::RFC::DB::deinit()`, which must be placed before any place the script could possibly exit.

Note that under mod_perl, DB.pm ignores inits. Instead, Apache::DBI transparently manages the connection pool across scripts, automatically issuing a rollback at script termination.

Quoting

Text values in SQL must be quoted with ’. This means any single-quotes in the value must be escaped (in SQL, replaced by two single-quotes: ”). This quoting should be done in the perl module handling database access using `$dbh->quote()`. The following prints `’O”Reilly’`:

```
$quoted = $dbh->quote("O’Reilly");
print $quoted;
```

Bind Columns vs. Fetch

When working with an $sth, use bind_columns whenever possible. This binds scalar references to columns in your sql result set, and as you fetch each row from the sth, the values of the variables are updated. Be careful when using columns which may be null, as the behavior of bind_columns when a null value is encountered is to not change the bound variable. If you need an undef to represent null values, make sure all variables are set to undef at the bottom of your loop. There are a few cases where you need other functionality, but in general, bind_columns should be your ginsu.

Naming Conventions

All function, method, and variable names should be written in lower case, with underscores separating words.

Object names should conform to java style. Objects, not instances of objects, which fall under the variable naming point.

The following is a list of commonly used variable names.

- $sql - a sql statement
- $dbh - a database handle

- $sth - a statement handle.
- $xml - raw xml
- $xmldom - a full xml dom structure
- $xsl - raw xsl
- @errors - an error list to print
- $cgi - an instance of CGI
- $html - an instance of Rescomp::RFC::Html (deprecated)
- $theme - an instance of Rescomp::RFC::UI::Theme

Command Line Scripts

Commandline scripts need to use the Getopts and/or Getopts::Long packages to parse command line args. You should at least support the --version and --help arguments, and print a usage message if the command line arguments do no match what you expected.