ASAP system Administration Guide

CONTENT

Web Administration

Basics

To start administrating the ASAP system the first thing you need is to make sure the system is installed (see InstallationGuide) and operating. The second thing is to make sure you have administrator rights. Once you checked the conditions you can actually do something with the system. Just go to http://address_of_the_ system/admin/admin_index.pl (or by the link to admin page on regular users page of the system – you should see it if you have administrator rights).

Web-structure

Administration page consists of two panels. Left one is for links the content of which are reflected on the right one. See ***DETAILS*** section to see what each of the links is for.

Errors

Errors are occurring from time to time (database malfunction or unexpected input errors). If the error keeps popping up just fix the reason, the code or tell someone to do it.

Details

GENERAL/Configuration

Change a configuration of the system any time you want (think twice before doing anything). The parameters are (lots of them):

APPEARANCE

Site name	- name of the system to show in the header
Site description	- description of the system to show in the header
Default skin	- skin of site to use by default

BASICS

Server name	- domain name like asap.fccc.edu where system installed
Absolute path to script folder	- file system path to script (cgi) folder of the system
Absolute path to document folder	- file system path to document (html) folder of the system
Relative URL to script folder	- relative path (relative to domain name) of script(cgi) folder
Relative URL to document folder	- relative path (relative to domain name) of document (html) folder

INTERFACE

Jobs per page	- number of jobs to show per page on status or JOB/Management page
Job updation time (launcher waiting)	- when launcher passes the job to ASAP how frequently to say that everything is ok
Job updation time (ASAP execution)	- when ASAP perfoms a job how frequently to say that everything is ok
Keep results for (days)	- for how long to keep the results of jobs on server (in days)
Number of lines to view	- for status page, for view of result file content, how many first strings to show
Top # users and plans for statistics page	- admin statistics page. Top users statistics. # of top entries.
Top # plans for user statistics	- admin statistics page. Top plans statistics. # of top entries.

SESSIONS

Session length (sec)	- terminate session when user is inactive for that number of seconds
Cookie name	- name for cookies to store session information
Cookie path	- cookies path
Cookie domain	- cookies domain

LDAP

Use LDAP check to allow login without registration	- check yes to allow first login through LDAP
LDAP server	- if use LDAP then the name of the server
LDAP port (if differs from 389)	- LDAP port if different from default
LDAP base to perform search	- LDAP base (see configuration of your LDAP server)
Username is a synonym of	- synonym of username in your LDAP

E-MAILS

Administrator's e-mail	- main administrator e-mail
E-mail signature	- signature to add in mass e-mails
Path to a sendmail program	- path to mail program to use to send e-mails
Use SMTP server for e-mailing	- pick yes if you want to use SMTP server to send e-mails
Name of SMTP server	- if use SMTP to send e-mails, specify the name of SMTP server
Login to SMTP server (if nesessary)	- SMTP login if required by server
Password to SMTP server (if nesessary)	- SMTP password if required by server

ACTIVATION

Make the system	- activate or deactivate system (user will be told that system is inactive) the system
Make the ASAP	- activate of deactivate ASAP (can be done also on ASAP/Configuration page)

GENERAL/Update
Update the system to the next version by submitting the updating zip package. You will probably be prompted to install new perl modules. If not and everything was ok, the system will be updated. If you getting some errors and you are updating from version let's say 5 to 9, try to update the system with version 6 patch, then 7 then 8 and only after that 9.

GENERAL/Export
The export of the system is used to create a file that can be imported by any other system to transport all the data (all means ALL except configuration information that is unique for a system and main administrator data). Just select options you want and press 'Export' button. The options are: ignore or not inactive plans (the plans that are not active right now in the system) and include or not information about agent-created tables – just the table structure information - (custom created tables by agents or other plans). The name of output file contains the date of creation and if no agent-created tables info were included – the _woACT suffix.

GENERAL/Import
System can replace the existing plans and database info from the import file (exported by another system you want to copy). Submit the file you received by exporting from another system. You probably will need to update your system to the version that the file to import is (you will be notified if you need to update the system). Select the options that include the choice to import the agent-created tables or not. Since such tables can be huge (gigabytes of information) you may want to import them manually (through file system if you know how to transfer database files from one location to another) or allow the system to do it. The process can take hours – so if you choose to import agent-created tables – be patient.

To suck the information for agent-created tables you need to provide the database location to suck it from. That includes knowing of server/port/database name and of coarse, the username and password for that database.

There could be different errors in the process of importing from remote resource, so some actions were taken to allow beginning the process from interrupted point. You can choose to retain the temporary tables populated before the interruption and start from the table the error triggered at.

GENERAL/Statistics
The most useless feature – allows to see various kinds of statistics for ASAP system, including distribution of number of jobs run over the day (hourly), over the week (dayly) and over the year (monthly). Here you can find statistics for grant reports under 'plain text' link. Any date interval can be specified in form of exact date (YYYY MM DD) or in form of date range (YYYY MM DD – YYYY MM DD). If you are skipping year, month or day the following rules are applied.
For exact date:
2004 __ __ will show statistics for year 2004
2004 10 __ will show statistics for October of 2004
2004 10 03 will show statistics for October 03 of 2004
____ 10 03 will show statistics for October 03 of current year
____ 10 __ will show statistics for October of current year
2004 __ 03 will show statistics for current month, day 03 of 2004
____ __ 03 will show statistics for current month, day 03 of current year
____ __ __ will show statistics for all time
For a date range:
It pretty much the same rules as for exact date except for 'from' range if something is missing from beginning (year, year and month or everything) then the year, month and day missed are taken from the beginning of the ASAP system existence.

GENERAL/Files
Reflects file structure for the cgi-bin folder of ASAP system. Allow to browse the file tree, download (in form of zip) or update (also by zip archive) folders and separate files. The names of uploaded files (inside zip) for updating should match the names of updated files.

GENERAL/Vocabulary
You can add/modify/delete the vocabulary terms here. The insertion of term can be tree-like. That means if you want to insert term x that has two subterms y and z then you need to submit three strings of data:
x
[space]y
[space]z

where [space] is any space character (space, tab, etc.)
The descriptions to each term should be specified after a tab character after the term, i.e:
x[tab]descripton of x

USERS/Management
Search a user by name (use * char to get a list of all users), pick one and you can update users information.

USERS/Mass E-mail
The feature is to send email to all users of the ASAP. You can choose the groups of users to send the email to. Subject and text message are to be specified.

USERS/Banner
Allows to sets the text for a banner (administrator message that appeared on every page, so every user can see it). You are allowed to use html tags in it (so play it safe). To remove the message just empty the text.

JOBS/Management
Search for a certain job by different criteria, pick a job, modify or view its details.

ASAP/Configuration
The first thing to do right after the ASAP system installation. The parameters to configure are ASAP database name, login, password, tables prefix, the result folder path that for ASAP to use to store results. The last thing is the ASAP activation checkbox – use it when you update ASAP and don't want users to use it that time – they'll get message like 'ASAP is being updated' if they try do it.

ASAP/Installation
The feature is used right after the ASAP system installation (well, right after ASAP configuration) and it installs (put information in database and stuff) all plans that preexist in installation package. Or another usage of that is when you (by some reason) put some new plans directly through the file system not through the web-interface – then the feature helps to officially install the plans into the system.

ASAP/Download
The feature is useful for plan writers – it allows to download ASAP package and/or PLANS existing at the time. Pick what you want to get and the system will give you link(s) for download.

PLANS/Installation
First screen – specify zip file with plan(s) or not-zipped plan file. Specify relative path of the plan to install, i.e. agent/downloaders/ could be a relative path to install a plan that does some downloading and plan is to be uploaded as plan.pm file of zip archive, containing plan.pm. Or the plan could be archived to zip into downloaders/plan.pm name and then the plan offset would be just agent/.
Second screen tells you if the plans you are about to install already exist and if the new plan is NEWER, OLDER, SAME compared to existing. If you pass installation plans in zip archive, system uses the dates of the files to make a comparison. If simple file is passed then the size is used as a measure. Then possible statuses could be NEWER or the SAME (yes, the

plans could be the same size but different in content - deal with it. If you don’t want to – pass the installation plans in zip archive). Pick a plans you want to install anyway and proceed to next screen.
Third screen tells the success status of installation (or not). All plans are activated right away. If you want to deactivate – use PLANS/Management.

PLANS/Management
Search for a certain plan by different criteria, pick a plan, modify or delete the plan’s information.

PLANS/Types
Allows you create/modify/delete new plan types (one by one). The types are then can be assigned to plans (see PLANS/Management), but no more than 7 types for one plan.

PLANS/Download
See ASAP/Download.

NEVER/Uninstall
You may never want to use this, since it uninstalls the system from file system completely (database are kept untouched though). You will be prompted three times to make sure you didn’t hit it accidentally.

Plan writing

Basics

Any plan is an independent perl script that uses ASAP package. It is supposed to have defined structure and use functions from ASAP package to make the plan a part of the system, so the system could keep track of the plan, create input interface to it and allow other plans to use the plan.

Requirements

The best tactics to write a plan would be first to get a perl installed on your computer and second to get ASAP core downloaded, so you can create and debug your plans locally, being comfortable in your desktop environment. To get a local copy of ASAP system (it includes main ASAP package, its includes and bunch of plans created to the moment) you need to go to the administration interface of ASAP system to ASAP/Download link and download the ASAP. The new plans should be created and run the way they could find the ASAP package. That is it. You are ready to write the new plan. The new plans should be created and run the way they could find the ASAP package. That is it. You are ready to write the new plan.

One more thing that can be done to ease your life in plan writing: the first thing that is needed to be done is to create ASAPcfg.pm file in the same directory where you put ASAP.pm package. The file should contain basic configuration parameters for ASAP to run with full power (e.g. to use database). You can skip the creation of the config file and specify the parameters each time with the creation of ASAP object (see ***Initialization***).

The config file looks like this:

```
package ASAP;

$ASAP::config{'db_type'} =    XXX;  # database type (mysql, oracle, etc.)
$ASAP::config{'db_name'} =    XXX;  # database name
$ASAP::config{'db_username'} = XXX; # database login
$ASAP::config{'db_password'} = XXX; # database password
$ASAP::config{'db_server'} =  XXX;  # server where database is (or localhost)
$ASAP::config{'db_port'} =    XXX;  # port of the database
$ASAP::config{'db_prefix'} =  XXX;  # database tables prefix for ASAP

$ASAP::config{'path_result'} = XXX; # path where to put result files to

$ASAP::config{'job_id'} = XXX;      # 32 long unique job id
$ASAP::config{'job_update'} = XXX;  # seconds for job progress update

$ASAP::config{'mode'} = XXX;        # '' - regular, DEBUG – for debug

$ASAP::config{'result_ext'} = XXX;  # ASAP output results files extension

1;
```

All variables are not mandatory as long as you don't want any special behavior from the ASAP.

Plan name notation

Each plan name is a relative path to the plan's location. Try to keep plans in structured fashion, so they could make sense and be easier to remember. The first level structure is already suggested. The plans are divided by the special role they play or the kind of sources they are using for annotations. So far, we have

agent/*	plans that play special role of doing something on regular basis (e.g. download GO to make it local every month)
db/*	plans that use local database for annotations (quick ones)
Exec/*	plans that use local programs for annotations (could be slow)
http/*	plans that use remote databases through http protocol for annotations (definitely slow)

So, if there is a plan in ASAP/PLAN/agent/GO/GO.pm then the conventional name for the plan would be ***agent/GO***

Structure

Here is an example of the very simple plan

```
use ASAP;                              # use ASAP package

my $asap = ASAP->new();                # create ASAP object

$asap->format(                         # define format of the plan by sections:

      DESCRIPTION,                     # DESCRIPTION
      ...,                             # To give little description of what plan does

      PLAN,                            # PLAN
      ...,                             # To specify other plans the plan is going to use

      INPUT,                           # INPUT
      ...,                             # Format of input parameters for the plan

      OUTPUT,                          # OUTPUT
      ...,                             # Type of output the plan is providing

);

my @input = $asap->in();               # Receive input parameters from user

# do something with input parameters here

$asap->out(...);                       # Output something as a result
$asap->outE(...);                      # Output errors if any happened

1;                                     # Always finish plans with 1; string
```

So, the structure of each plan should contain ASAP object creation and format specification. Other functions are not mandatory, but it is hard to imagine a plan that doesn't use them (actually, some agents are that kinds of plans. They don't need any input, they don't provide direct output – just fill some database tables or create some work files, so they don't need input or output functions).

Just for an example – the smallest plan possible:

```
use ASAP;
my $asap = ASAP->new();
$asap->format();
1;
```

Please, note that ***format*** function is better to be called right after ASAP object creation. That is because the ***format*** function is used by system to acquire information about a plan, and the earlier the function is called, the better.

Initialization

Initialization here means how to create ASAP object (or in other words what is the special environment the plan and all other plan it uses will be run). The options to specify here are:

```
my $asap = ASAP->new(
      NOT_PLAN,
      'db_type' =>      'mysql',
      'db_name' =>      'database name',
      'db_username' =>  'database login name',
      'db_password' =>  'database password name',
      'db_server' =>    'database server',
      'db_port' =>      'database port',
      'db_prefix' =>    'prefix for tables in the database',
      'path_result' =>  'folder for output results',
      'job_id' =>       'unique id for every new plan querying',
      'job_update' =>   'time period to update information about the job',
);
```

Special keyword here is NOT_PLAN that says to ASAP that the topmost plan is not actually a plan, but something that uses ASAP plans, suggesting the environment. That is used by web part of ASAP to launch plans. You can use this to create you own perl script that uses power of ASAP plans.

(Note: the topmost plan overrides the environment for all sub plans)

Format

Format function is the one that specifies all the information about a plan that ‘outside’ world could and should acquire. The format gives the description of the plan, specifies the other plans the plan is going to use, informs about input parameters the plan is willing to take (ASAP uses it to compose web-page for user to actually input the input parameters), fixes the output parameters the plan is going to provide. Those subformats correspond to different sections in format function: DESCRIPTION, PLANS, INPUT, OUTPUT. Each section is separated from each other by commas.

DESCRIPTION

The section gives the idea of the plan for users. Since it isn’t necessary for plan internals – it is up to plan writer not to forget to describe the plan. Each separate string in the section will be a new string in the description. Of course, you always can give the description by one long string, inserting “\n” new line character where you need it.

PLANS

The section specifies what other plans are going to be used in the plan. Plans should be separated by commas, and satisfy the plan name notation. The ASAP system will warn you if the plan use the plan that wasn’t declared in format.

INPUT
The section is the one that defines the interface of the input query web-page for the plan. Parameters can be separated into different input groups. To do it specify name of the group and its description (optional) before the group of input parameters like this:

```
INPUT, ['group name', 'group description'],
... input parameter specifications ...
INPUT, ['another group name', 'another group description'],
... other input parameters ...
```

(Note: default group name is INPUT constant)

That makes comfortable to separate the parameters visually on the web-page. E.g. if you have mandatory and optional input parameters, you can separate them in two groups and user will be happy to see that.

Each parameter can have following specifications:

```
'name' => 'unique name',
'type' => 'one of the terms from vocabulary',
'desc' => 'some description to show',
'restr'=> 'restriction of the output',
'default' => 'some default value',
```

'***name***' should be unique among all input and output parameters

'***type***' specifies the type of the input parameter, defined in vocabulary hierarchy. That is (will be) used to connect different plans together or to make searches of plans based on these types. If the parameter accepts batch entities, you should emphasize it by appending |m at the end (e.g ‘accession|m’ means that the input parameter accepts accession numbers, many accession numbers at ones)

'***desc***' specifies description to show on the web-page against the field of the input parameter. User understanding of what to give as this input parameter highly based on what you specify as ‘***desc***’

'***restr***' specifies the element to use on web-page for the input parameter. Options are: text, textarea, file, checkbox, select, radio. For radio and select elements there are additional to be specified (separated by ‘|’) – e.g ‘select|All|a|b|c|d’ specifies to use select box for the input parameter with options: All, a, b, c and d

'***default***' specifies default value for the input parameter

Here is an example of INPUT section:

```
INPUT, ['MAIN', 'MAIN'],
'name' => 'acc',
'type' => 'accession|m',
'desc' => 'Various accession numbers',
'restr'=> 'textarea',

INPUT, ['OPTIONAL', 'OPTIONAL'],
'name' => 'part',
'type' => 'text',
'desc' => 'Get answers that contain the word',
'restr'=> 'text',

'name' => 'org',
'type' => 'organism',
'desc' => 'Organism',
'restr'=> 'select|All|Hs|Mm|Rn',
'default' => 'Hs',
```

OUTPUT

Like INPUT section, OUTPUT can be separated into different groups. The output of the topmost plan then will result in different output files. To make such groups specify name of the group, its description (optional) and subdescription (optional) before the group of output parameters like this:

```
OUTPUT, ['group name', 'group description', 'sub description'],
... output parameter specifications ...
OUTPUT, ['another group name', 'another group description'],
... other output parameters ...
```

(Note: default group name is OUTPUT constant)

Also, some of the input or output parameters could be rejected from suggesting for user to output (from web-page) by initializing '**reject**' field with separated by '|' names of input or output parameters (see example below)

Each parameter can have following specifications:

```
'name' => 'unique name',
'type' => 'one of the terms from vocabulary',
'desc' => 'some description to show',
'restr'=> 'restriction of the output',
```

'***name***' should be unique among all input and THE GROUP output parameters

'***type***' specifies the type of the output parameter, defined in vocabulary hierarchy. That is (will be) used to connect different plans together or to make searches of plans based on these types.

'***desc***' specifies description to show on the web-page against the checkbox of whether to output the output parameter or not (user chooses it)

'***restr***' not actually used presently

Here is an example of OUTPUT section:

```
OUTPUT, ['MAIN', 'MAIN file with important stuff', "Very important"],
'name' => 'GO',
'type' => 'GO',
'desc' => 'GO term',
'restr'=> 'text',

'name' => 'GOtype',
'type' => 'GO type',
'desc' => 'Type of GO term (1-process, 2-function, 3-component)',
'restr'=> 'text',

OUTPUT, ['op2', 'OUTPUT2', 'Something different'],
'reject' => 'part|org',

'name' => 'pGO',
'type' => 'GO:process',
'desc' => 'Process GO',
'restr' => 'text',
```

Input

If you declared some input parameters in format, you can acquire the values passed to your plan by using ***in*** function. There are lots of options how to use it. For the INPUT example above:

To retrieve all input parameters values:

`my @params = $asap->in();` or `my ($accession, $part_var, $orgm) = $asap->in();`

Note, that the elements in the array returned are in the same order as in format definition.

or explicitly (more safe - doesn't depend on format order)

```
my ($part_var, $acc) = $asap->in('part', 'acc');
```

or just for one variable:

```
my $acc = $asap->in('acc');
```

or all parameters from some group:

```
my @opt_group = $asap->in('OPTIONAL');
```

Using other plans

You plan probably wants to use some other plans. There is a fairly simple way to do it. First, you need to declare your intention to do so in ***format*** function (see PLANS section of the format for details). Then, you need to submit input parameters and choose what output parameters you need. The ***plan*** function is the one you need to carry it out.

Here are two examples of using a plan:

ONE:

```
my %output = $asap->plan( $input_file, FROM_FILE );
```

You stuff all the data (plan name, input parameters, desired output parameters) in the xml file (see XML input) and pass the file to the **plan** function along with keyword FROM_FILE and receive results in complex data (see format below)

TWO:

```
my %output = $asap->plan(

'db/some/plan',

ERROR_HANDLE,

JOB_SIZE,
...

INPUT,
'acc' => 'P123123   O098574   R432523',
'part' => 'kinase',
'org' => 'Mm',

OUTPUT, ['MAIN'],
'acc',
'GO',

)
```

Here, you specify plan name, input parameters, desired output parameters explicitly (the order of the sections doesn't matter). Another thing you need to specify is how to handle errors if occur during the plan execution. There are two options: ERROR_HANDLE and ERROR_DIE. ERROR_HANDLE makes you to handle the error by yourself (in the case of error the output will be empty). That is useful in case if you are ready to not receive results from some internet sites and can continue further. Or you want to do something before make your plan die – for example report for user that the annotation didn't work out as it supposed to. Another option is ERROR_DIE, that simply gives the permission to the plan you are using to die and exit. For details on errors handling see **Errors**

There is an optional parameter JOB_SIZE to pass in plan calling. It sets the size of the progress for the called plan and allows that plan to change the progress only within those intervals. Note, that you should define the job size for the current plan already to make things work.

Once you get the results from file you can access the output parameters you requested. The structure of the output hash is following:

```
output_hash{'output group'}[entry index]{'output parameter name'}
```

and in terms of the example above to access first result for output group MAIN for GO you need to do something like this:

```
$want_this = $output{'MAIN'}[0]{'GO'}
```

Or if you want to get all such entries from the output, you could do something like:

```
foreach my $subhash_ref ( @{$output{'MAIN'}} ){
      print $subhash_ref->{'GO'};
}
```

If you are using the plan from NOT A PLAN then the results from will be but not just to the return value but also to files (into directory specified during ASAP initialization)

Progress

This is not a mandatory part of any plan, BUT it definitely a necessary feature to keep user satisfied. If the plan takes quite amount of time to finish user definitely get nervous if he/she doesn't see any change in progress. There is a way incorporated in ASAP to handle this issue. There are four functions that do it – ***job_size***, ***job_step***, ***job_jump***, ***job_prog***. First, you pick the size of what you need to do (you decide!), then during execution of the plan you make a step by some amount of progress or you just jump to some level of progress. Let's say you have 300 time consuming tasks to perform in a plan then the plan should probably look like this:

with ***job_step***:

```
# ... beginning ...
$asap->job_size(300);
for( @tasks ){
      # do your task
      $asap->job_step(1);
}
# ... end ...
```

or with ***job_jump***

```
# ... beginning ...
$asap->job_size(300);
$i = 1;
for( @tasks ){
      # do your task
      $asap->job_jump($i);
      $i++;
}
# ... end ...
```

To get a progress of the job with ***job_prog***:

```
$progress = $asap->job_prog();
```

Output

Once you declared some output parameters in format, you should provide this output. The way to do it is to use ***out*** function. For any output group you can use something like:

```
$asap->out( 'MAIN',
      'acc' => 'P32141',
      'GO' => 'GO:123546',
      'GOtype' => 'process',
);
```

Note, that you can skip output of input parameters as long as they are the same through the whole plan. That is not the case for the plans that accept batch queries (i.e. perform separate analysis for parts of some input parameter) – then you better to specify the part of the input you giving output result for.

Sometimes your plan could face problems with some sources of information (some internet site went down for instance) – and you can not perform correct annotation. In that case you better report user about the error by using ***outE*** function. In that function you should report the combination of input parameters that happened to fail. Again, you don't need to submit input parameters as long as they are the same through the whole plan. As an example, if error happen during annotation of accession '`P32141`', your error output should look like:

```
$asap->outE( 'MAIN',
      'acc' => 'P32141',
);
```

Error handling

To make ASAP system up-to-date we should always control constantly changing outer information sources formats. In order to do it, every plan should check the data it is trying to process and if anything doesn't meet the expectations report about it to ASAP. The ASAP itself will send the report to the administrator of ASAP. Errors should be reported by function ***error*** a giving error code and error message, like

```
error( 11000,"Unexpected format" )
```

Here are the rules for error codes to report:

```
1xxxx – general ASAP errors (e.g ASAP initialization failure or configuration is not full, etc)
2xxxx – plan errors (e.g when plan doesn't define format or uses undefined plan, etc.)

x1xxx – general unspecified errors
x2xxx – database errors (e.g connection failure or query errors, etc.)
x3xxx – http connections errors (e.g server is not responding, or query failed)
x4xxx – file system errors (e.g program failed or file doesn't exist or failed to be copied, etc)

DB errors
000 – general error
001 – connection error
002 – INSERT error
```

```
003 – UPDATE error
004 – SELECT error
005 – CREATE error

FILE system errors
001 – doesn't exist
002 – protected from reading
003 – protected from writing
004 – can't be created
005 – plan execution error
006 – directory changing errors


SPECIAL cases
21000 – unexpected format
21111 – report message
```

If you don't want to die after the error, but just want to record the error message – call 'report' function instead with the message as a parameter.

Database functions

As a part of functionality that ASAP gives to any plan is a database operating routines. You can use database object $ASAP::DB exported by ASAP to query database. You can submit any sql query to database and receive a result in form of hash. Here is a little example that acquires names of all tables from database:

```
my $sth = $ASAP::DB->query("show tables");      # make a query
error(12000, "Database error") unless($sth);    # check for database failure
while( my %opa = $ASAP::DB->fetchrow($sth) ){   # for each result instance
      print "$opa{'Table_in_db'}\n";            # print the data
}
```

After making a query you can check number of instances (rows) that appeared in the result:

```
$ASAP::DB->rows();
```

There is one more useful function to use here. It is ***dbesc*** (you can call it without object reference). It escapes any unsafe database characters and returns the string embraced in apostrophes. E.g if you have string ***smth'else*** then ***dbesc*** give the string ***'smth\'else'*** or whatever is appropriate for the type of database ASAP is using.

Web-Downloading

In two words web-downloading can be interpreted as sending some request and receiving some response from some source. Sometimes it is not just one request but many, depending on kind of responses a source gives. ASAP provides tools to deal with such requests and responses.

To get a web-page content or some file from ftp or by any protocol, you should make a request. Something like this:

```
my $response = $asap->request(
      uri => "http://fccc.edu",     # URI to request
```

```
      method => "GET",              # method to use
      download => "/path/where/to/save",  # says to put the content in a file
      callback => \&callback,             # callback function
      chunk => 100000,  # call callback function each 100 kb downloaded
      timeout => 180,   # set a timeout for downloading (180 seconds)
      redirect => 1,    # allow redirect a request

);

if( $response->is_success ){  # if response is a success one
      ...                     # do what you want here
}
```

The request function actually incorporates three basic perl packages - LWP::UserAgent, HTTP::Headers, HTTP::Request and returns a HTTP::Response object (Yeees, you need to read all about theses packages to be aware of all advantages the request function gives you). In a nutshell, the ***request*** function implements following algorithm. It takes the input parameters, creates headers for a request using HTTP::Headers-specific parameters (mentioned below), then forms a request with the headers and HTTP::Request-specific parameters (also mentioned below), then passes the request to LWP::UserAgent object using LWP::UserAgent-specific parameters. The output is a HTTP::Response object that contains as a 'content' the content of requested URI or a file name that contains the content of requested URI. So, you can specify all the parameters you need in the ***request*** function and receive a response you can use later.

Here is a list of parameters that can be used:

```
ASAP-specific:
redirect    - use redirect or not
download    - 1 if we need to download the content into file (tmp file will
              be created unless instead of one you specify the file to put the
              content in). Response will contain the file name the content was
              downloaded to.
callback    - callback routine
chunk       - number of downloaded bytes after which call callback
headers     - HTTP::Headers object directly

HTTP::Request:
uri, method

LWP::UserAgent:
'proxy', 'no_proxy'
agent, from, timeout, max_size, max_redirect, env_proxy, cookie_jar,
conn_cache, keep_alive, protocols_allowed, protocols_forbidden,
requests_redirectable

HTTP::Headers (or headers object can be passed in ASAP-specific attributes)
(since 'from' attribute is shared with LWP - let it belong to LWP)
'content'
Cache-Control, Connection, Date, Pragma, Trailer, Transfer-Encoding, Upgrade
Via, Warning, Accept, Accept-Charset, Accept-Encoding, Accept-Language
Authorization, Expect, From, Host, If-Match, If-Modified-Since, If-None-
Match, If-Range, If-Unmodified-Since, Max-Forwards, Proxy-Authorization,
Range, Referer, TE, User-Agent, Accept-Ranges, Age, ETag, Location, Proxy-
Authenticate, Retry-After, Server, Vary, WWW-Authenticate, Allow, Content-
Encoding, Content-Language, Content-Length, Content-Location, Content-MD5,
Content-Range, Content-Type, Expires, Last-Modified
```

Usually, the example provided above covers most of cases.
Here are some basic HTTP::Response package methods that save you a lot of reading:

```
->code            # HTTP code of response
->message         # message of the HTTP response
->is_success      # use it to check if request succeed
->is_error        # use it to check if request failed
->content         # source content or a file name where the content stored
```

The information given here should be sufficient for writing all kinds of requests. Once you have something special to do – perl documentation reading is an answer.

XML input

You can use some plan by passing all input in XML file (see **Using other plans**). Here is a description of the file format defined by general example:

```
<?xml version="1.0"?>
<ASAP>
<QUERY plan="plan name">
<input group="INPUT GROUP" name="INPUT NAME">INPUT VALUE</input>
...
<output group="INPUT GROUP" name="INPUT NAME">INPUT ALIAS</output>
...
<output group="OUTPUT GROUP" name="OUTPUT NAME">OUTPUT ALIAS</output>
...
</QUERY>
</ASAP>
```

The strings with bolded parts are the one you need to change appropriate to the query you want to submit. The '...' strings are the one that will contain modified entries of strings with bolded parts. Each such string will describe usage of the input or output parameter. Now, the descriptions of bolded text.

plan name	- plan name in plan notation format (see **Plan notation**)
for <input> tag	- the tag specifies input parameters values for the plan
INPUT GROUP	- input *group name* that the input variable belongs to
INPUT NAME	- *name* of the input (for input) variable
INPUT VALUE	- the *value* of the input variable to pass to plan
for <output>	- the tag specifies input and/or output parameters to output
INPUT GROUP	- input *group name* that the input variable (for output) belongs to
INPUT NAME	- *name* of the input variable to output
INPUT ALIAS	- how to name the variable when output to file
OUTPUT GROUP	- output *group name* that the output variable (for output) belongs to
OUTPUT NAME	- *name* of the output variable to output
OUTPUT ALIAS	- how to name the variable when output to file

That is the only file you need to submit to a plan to run it. In case some of the input parameters (for input) are omitted – the plan will use default values for it.

Constants reference

Constants exported by ASAP

DESCRIPTION
To use as a marker in ***format*** method of ASAP object. It starts the section of defining a description for the plan. Separate the description given from the DESCRIPTION constant (and between each string of description) by commas.

ERROR_DIE
To use as a parameter in ***plan*** method of ASAP object. It allows requested plan to terminate the whole annotation in case of error (see ERROR_HANDLE for opposite effect)

ERROR_HANDLE
To use as a parameter in ***plan*** method of ASAP object. It defines that in case if requested plan is failed by unexpected error the calling plan would take care of handling the error (see ERROR_DIE for opposite effect)

FROM_FILE
To use as a parameter in ***plan*** method of ASAP object. If passed as a second parameter to the method then the first parameter is treated as a file name with all input data for the requested plan.

INPUT
To use as a marker in ***format*** or ***plan*** method of ASAP object.
For ***format*** method it starts the section of defining format for input parameters of the plan. Separate the input parameters specifications from the INPUT constant (and the input parameters specifications from each other) by commas.
For ***plan*** method it starts the section of specifying values for input parameters passed to the requested plan for different input groups.

NOT_PLAN
To use as a parameter in ***new*** constructor of ASAP object. It defines that the caller of the ASAP object is not actually a plan. It is supposed to be used from independent perl scripts to run ASAP plans.

OUTPUT
To use as a marker in ***format*** or ***plan*** method of ASAP object.
For ***format*** method it starts the section of defining format for output parameters of the plan. Separate the output parameters specifications from the OUTPUT constant (and the output parameters specifications from each other) by commas.

For ***plan*** method it starts the section of specifying output and input parameters (and aliases) asked to return from requested plan for different output groups.

PLANS
To use as a marker in ***format*** method of ASAP object. It starts the section of plans declaration that are going to be used by parent plan. Separate the list of plans from the PLANS constant (and between each other) by commas.

JOB_SIZE
For ***plan*** method it starts the section of specifying the size of delegated job to show the progress within that size.

Variables reference
Special variables provided through package name by ASAP

$ASAP::config{'mode'}
Set the variable to 'DEBUG' to enable ***debug*** function.

$ASAP::DB
Database object reference.

$ASAP::PRFX
Prefix value for tables in the current database given for ASAP. Can be NULL.

Functions reference
Methods for ASAP object

new(parameters)
Function creates ASAP object for further using in the plan (see ***Initialization***). There are multiple initialization parameters that could be passed to the construction:

```
new(
      NOT_PLAN,
      'db_type' =>      'mysql',
      'db_name' =>      'database name',
      'db_username' =>  'database login name',
      'db_password' =>  'database password name',
      'db_server' =>    'database server',
      'db_port' =>      'database port',
      'db_prefix' =>    'prefix for tables in the database',
      'path_result' =>  'folder for output results',
      'job_id' =>       'unique id for every new plan querying',
      'job_update' =>   'time period to update information about the job',
);
```

Special keyword here is NOT_PLAN (see ***Constants reference***) that says to ASAP that the topmost plan is not actually a plan, but something that uses

ASAP plans, suggesting the environment. That is used by web part of ASAP to launch plans. You can use this to create you own perl script that uses power of ASAP plans.

(Note: the topmost plan overrides the environment for all sub plans)

input

Function is used to initialize input parameters values from XMLfile (name is passed as a parameter)

format

Function is used to define different formats for the plan, preferably to be called right after ASAP object creation (see ***Format***). The input parameters to the function are:

```
format(                              # define format of the plan by sections:

      DESCRIPTION,                   # DESCRIPTION
      ...,                           # To give little description of what plan does

      PLAN,                          # PLAN
      ...,                           # To specify other plans the plan is going to use

      INPUT,                         # INPUT
      ...,                           # Format of input parameters for the plan

      OUTPUT,                        # OUTPUT
      ...,                           # Type of output the plan is providing

);
```

DESCRIPTION
The section gives the idea of the plan for users. Since it isn't necessary for plan internals – it is up to plan writer not to forget to describe the plan. Each separate string in the section will be a new string in the description. Of course, you always can give the description by one long string, inserting "\n" new line character where you need it.

PLANS
The section specifies what other plans are going to be used in the plan. Plans should be separated by commas, and satisfy the plan name notation. The ASAP system will warn you if the plan use the plan that wasn't declared in format.

INPUT
The section is the one that defines the interface of the input query web-page for the plan. Parameters can be separated into different input groups. To do it specify name of the group and its description (optional) before the group of input parameters like this:

```
INPUT, ['group name', 'group description'],
... input parameter specifications ...
INPUT, ['another group name', 'another group description'],
... other input parameters ...
```

(Note: default group name is INPUT constant)

That makes comfortable to separate the parameters visually on the web-page. E.g. if you have mandatory and optional input parameters, you can separate them in two groups and user will be happy to see that.

Each parameter can have following specifications:

```
'name' => 'unique name',
'type' => 'one of the terms from vocabulary',
'desc' => 'some description to show',
'restr'=> 'restriction of the output',
'default' => 'some default value',
```

'***name***' should be unique among all input and output parameters

'***type***' specifies the type of the input parameter, defined in vocabulary hierarchy. That is (will be) used to connect different plans together or to make searches of plans based on these types. If the parameter accepts batch entities, you should emphasize it by appending |m at the end (e.g 'accession|m' means that the input parameter accepts accession numbers, many accession numbers at ones)

'***desc***' specifies description to show on the web-page against the field of the input parameter. User understanding of what to give as this input parameter highly based on what you specify as '***desc***'

'***restr***' specifies the element to use on web-page for the input parameter. Options are: text, textarea, checkbox, select, radio. For radio and select elements there are additional to be specified (separated by '|') – e.g 'select|All|a|b|c|d' specifies to use select box for the input parameter with options: All, a, b, c and d

'***default***' specifies default value for the input parameter

Here is an example of INPUT section:

```
INPUT, ['MAIN', 'MAIN'],
'name' => 'acc',
'type' => 'accession|m',
'desc' => 'Various accession numbers',
'restr'=> 'textarea',

INPUT, ['OPTIONAL', 'OPTIONAL'],
'name' => 'part',
'type' => 'text',
'desc' => 'Get answers that contain the word',
'restr'=> 'text',

'name' => 'org',
'type' => 'organism',
'desc' => 'Organism',
'restr'=> 'select|All|Hs|Mm|Rn',
'default' => 'Hs',
```

OUTPUT

Like INPUT section, OUTPUT can be separated into different groups. The output of the topmost plan then will result in different output files. To make such groups specify name of the group, its description (optional) and subdescription (optional) before the group of output parameters like this:

```
OUTPUT, ['group name', 'group description', 'sub description'],
... output parameter specifications ...
OUTPUT, ['another group name', 'another group description'],
... other output parameters ...
```

(Note: default group name is OUTPUT constant)

Also, some of the input or output parameters could be rejected from suggesting for user to output (from web-page) by initializing '**reject**' field with separated by '|' names of input or output parameters (see example below)

Each parameter can have following specifications:

```
'name' => 'unique name',
'type' => 'one of the terms from vocabulary',
'desc' => 'some description to show',
'restr'=> 'restriction of the output',
```

'***name***' should be unique among all input and THE GROUP output parameters

'***type***' specifies the type of the output parameter, defined in vocabulary hierarchy. That is (will be) used to connect different plans together or to make searches of plans based on these types.

'***desc***' specifies description to show on the web-page against the checkbox of whether to output the output parameter or not (user chooses it)

'***restr***' not actually used presently

Here is an example of OUTPUT section:

```
OUTPUT, ['MAIN', 'MAIN file with important stuff', "Very important"],
'name' => 'GO',
'type' => 'GO',
'desc' => 'GO term',
'restr'=> 'text',
```

```
'name' => 'GOtype',
'type' => 'GO type',
'desc' => 'Type of GO term (1-process, 2-function, 3-component)',
'restr'=> 'text',

OUTPUT, ['op2', 'OUTPUT2', 'Something different'],
'reject' => 'part|org',

'name' => 'pGO',
'type' => 'GO:process',
'desc' => 'Process GO',
'restr' => 'text',
```

in

Function is used to initialize plan's local variables with input parameters values passed to the plan (see ***Input***).
If function takes no parameters then it returns array of all plans input parameters values.
If function takes name(s) of the input group(s) then it returns array of plans input parameters values of the specified group(s).
If function takes name(s) of plans input parameter(s) it returns the list of plans input parameters values.
More specific:

To retrieve all input parameters values:
`my @params = $asap->in();` or `my ($accession, $part_var, $orgm) = $asap->in();`
Note, that the elements in the array returned are in the same order as in format definition.

or explicitly (more safe - doesn't depend on format order)
```
my ($part_var, $acc) = $asap->in('part', 'acc');
```

or just for one variable:
```
my $acc = $asap->in('acc');
```

or all parameters from some group:
```
my @opt_group = $asap->in('OPTIONAL');
```

out

Function stuffs the parameters passed to it into plans output (see ***Output*** for details). The parameters are passed in form of hash, where keys are the names of the plans parameters to output and values are the values of that parameters

outE

Function stuffs the parameters passed to it into plans ERROR output (see ***Output*** for details). The parameters are passed in form of hash, where keys are the names of the plans INPUT parameters which caused the error and values are the values of that parameters.

is_requested

Function checks if a specified parameter of the output group is requested by user for output. Useful to check if certain parameter needs significant amount of time to retrieve – you can skip it if it is not necessary.

```
$asap->is_requested( 'output_group',
      'output_var'
);
```

plan
Function is used to call another plan from current plan. There are two possible inputs to the function:

```
plan( $input_file, FROM_FILE );
```

The function takes an XML file name (defined in XML file format) and FROM_FILE constant. It returns the hash containing the results.

The second option is to specify everything in input parameters:

```
plan(

  'plan name',

  ERROR_HANDLE,

  JOB_SIZE, size_value,

  INPUT,
  variable => value,
  ...

  OUTPUT, ['output group name'],
  variable => value,
  ...

)
```

It returns the hash containing the results.
The structure of the output hash is following:
`output_hash{'output group'}[entry index]{'output parameter name'}`

(See ***Using other plans*** for details)

job_size
Function sets the size for the overall progress (passed as input parameter) It should be used in progress handling (see ***Progress***).

job_step
Function increases progress level on passed as input parameter value. It should be used in progress handling (see ***Progress***).

job_jump
Function sets progress level to passed as input parameter value. It should be used in progress handling (see ***Progress*** for details).

job_prog
Function gets the progress level of the job. (see ***Progress*** for details).

request
The request function actually incorporates three basic perl packages - LWP::UserAgent, HTTP::Headers, HTTP::Request and returns a HTTP::Response object. It takes the input parameters, creates headers for a request using HTTP::Headers-specific parameters (mentioned below), then forms a request with the headers and HTTP::Request-specific parameters (also mentioned below), then passes the request to LWP::UserAgent object using LWP::UserAgent-specific parameters. The output is a HTTP::Response object that contains as a 'content' the content of requested URI or a file name that contains the content of requested URI.

Here is a list of parameters that can be used:

```
ASAP-specific:
redirect    - use redirect or not
download    - 1 if we need to download the content into file (tmp file will
              be created unless instead of one you specify the file to put the
              content in). Response will contain the file name the content was
              downloaded to.
callback    - callback routine
chunk       - number of downloaded bytes after which call callback
headers     - HTTP::Headers object directly

HTTP::Request:
uri, method

LWP::UserAgent:
'proxy', 'no_proxy'
agent, from, timeout, max_size, max_redirect, env_proxy, cookie_jar,
conn_cache, keep_alive, protocols_allowed, protocols_forbidden,
requests_redirectable

HTTP::Headers (or headers object can be passed in ASAP-specific attributes)
(since 'from' attribute is shared with LWP - let it belong to LWP)
'content'
Cache-Control, Connection, Date, Pragma, Trailer, Transfer-Encoding, Upgrade
Via, Warning, Accept, Accept-Charset, Accept-Encoding, Accept-Language
Authorization, Expect, From, Host, If-Match, If-Modified-Since, If-None-
Match, If-Range, If-Unmodified-Since, Max-Forwards, Proxy-Authorization,
Range, Referer, TE, User-Agent, Accept-Ranges, Age, ETag, Location, Proxy-
Authenticate, Retry-After, Server, Vary, WWW-Authenticate, Allow, Content-
Encoding, Content-Language, Content-Length, Content-Location, Content-MD5,
Content-Range, Content-Type, Expires, Last-Modified
```

timeout
Function sets timeout for portion of time-consuming code in the perl. Pass the timeout value (in seconds) in input parameter. You should remove the effect of timeout by calling the function without parameters after the chunk of the code you want to set limits of execution to.

Functions exported by ASAP

dbesc(string)
Function takes one parameter and escapes any database unsafe characters existing in input parameter and returns the string embraced in apostrophes.

E.g if you have string ***smth'else*** then ***dbesc*** gives the string ***'smth\'else'*** or whatever is appropriate for the type of database ASAP is using.

debug(message)
Function takes one parameter and prints it (also in the ASAP_debug.txt file). Use it for debugging purposes. Function are disabled (i.e. does nothing) unless ***$ASAP::config{'mode'}*** is set to 'DEBUG'.

error(code, message)
Function takes two parameters: error code and error message to report the error to ASAP. See ***Error handling*** for details.

info(plan_name)
Function takes one parameter – plan name (see ***Plan name notation***) and returns hash containing format details of the requested plan if possible and undef otherwise. The returned hash has following keys:

done	should be 1
desc	plan description
plans	plans that are used by the plan
igroups	array of input group names
ogroups	array of output group names
input	hash with keys of input groups names, each value is a hash with two keys – 'descr' (contains the input group description) and 'vars' is an array with the input parameters names of the input group. For example: $returned{'input'}{'MAIN'}{'descr'} gives a description of input group 'MAIN' and @{returned{'input'}{'MAIN'}{'vars'}} contains all the names of the input group 'MAIN' parameters.
output	hash with keys of output groups names, each value is a hash with two keys – 'descr' (contains the output group description) and 'vars' is an array with the output parameters names of the output group For example: $returned{'output'}{'OUT1'}{'descr'} gives a description of the output group 'OUT1' and @{returned{'output'}{'OUT1'}{'vars'}} contains all the names of the output group 'OUT1' parameters.
igroup	hash with keys of all input parameters names, each value is a hash containing input parameters specifications (see INPUT section of ***Using other plans***) For example: %{ $returned{'input'}{'var1'} } gives a hash of input parameter 'var1' specifications
ogroup	hash with keys of output groups names, each value is a hash with keys of all output group parameter names and values are the hash again containing output parameters specifications (see OUTPUT section of ***Using other plans***) For example: %{ $returned{'output'}{'OUT1'}{'var1'} } gives a hash of output parameter 'var1' (group 'OUT1') specifications.

report(message)
Function takes a report message to report it to ASAP. See ***Error handling*** for details.