Contents

[Basic Concepts 1](#_Toc264383165)

[Overview 1](#_Toc264383166)

[Additional Features 3](#_Toc264383167)

[Built in 3](#_Toc264383168)

[EuPathDB specific 3](#_Toc264383169)

[Information flow in the graph 4](#_Toc264383170)

[Creating a workflow 4](#_Toc264383171)

[Graph XML files 4](#_Toc264383172)

[**Compiling a graph** 5](#_Toc264383173)

[**Global graphs** 6](#_Toc264383174)

[**Workflow home directory** 7](#_Toc264383175)

[**Config files** 7](#_Toc264383176)

[**Changing a graph after the workflow has begun** 8](#_Toc264383177)

[**Grafting** 8](#_Toc264383178)

[**Subgraph references** 8](#_Toc264383179)

[**Nested data directory structure** 8](#_Toc264383180)

[**Resource steps** 9](#_Toc264383181)

[**Tracking Analysis Methods** 10](#_Toc264383182)

[**Best Practices** 11](#_Toc264383183)

[**Testing a workflow** 11](#_Toc264383184)

[**Running a workflow** 13](#_Toc264383185)

[**GUS dependencies or assumptions** 13](#_Toc264383186)

# Basic Concepts

## Overview

ReFlow is a simple workflow system based on a dependency graph.  It runs on UNIX, and has been tested on Linux.  Its primary user interface is textual (command line and log files).  While it does not have a GUI, its textual tools do a good job managing very large workflows.  
  
In overview:

* A ReFlow workflow is a directed acyclic graph
  + The nodes are Steps
  + The edges are dependencies.  A step is run when all the steps it depends are done.
* Steps
  + execute an atom of work, such as:
    - creating a shared directory
    - acquiring data from the internet
    - loading data into a database
    - running a local analysis program
    - starting a job on a compute cluster
  + are an instance of a step class.
  + declare dependencies on other steps
* Step classes
  + are instantiated by individual steps
  + perform a particular type of work
  + are written in Perl, and are a subclass of [ReFlow::Controller::WorkflowStepInvoker](https://www.cbil.upenn.edu/svn/gus/ReFlow/trunk/Controller/lib/perl/WorkflowStepInvoker.pm)
  + are parameterized.  When a step is invoked by the controller, the step class instance receives a package of key-value pairs, much like a subroutine is given argument values.
  + can do any kind of work that is compatible with the WorkflowStepInvoker API, in general launching one or a series of processes on the controller machine that will give a correct exit status code on completion.
* A Graph
  + is specified in a graph XML file
    - the [schema](https://www.cbil.upenn.edu/svn/gus/GusAppFramework/trunk/Workflow/lib/rng/workflow.rng) is defined in RNG
  + calls one or more Steps in the order specified by their dependencies
  + declares input parameters
  + declares constants
  + defines a local scope in which steps can
    - depend on other steps
    - have access to graph input parameters
    - have access to graph constants
  + may call subgraphs
    - these are similar to Steps but invoke a nested graph instead of a step class instance
    - the nesting of subgraphs forms a containment hierarchy that is orthogonal to the dependency graph.
    - Subgraphs are passed input parameter values
  + may be called as a root or sub-graph
  + may be called multiple times within the full graph
  + may be thought of as a subroutine
* The Root Graph
  + is the top of the graph nesting hierarchy.
  + is no different than any other graph, except that it is the top of the nesting
  + is passed its input parameter values from the rootParams.prop file
* The Full Graph
  + is the dependency graph obtained by expanding all the nested subgraphs.

Here is a powerpoint representation of the [PlasmoDB build workflow](https://www.cbil.upenn.edu/svn/apidb/ApiCommonWorkflow/trunk/Main/doc/IntegratedWorkflow.ppt), a large workflow developed by EuPathDB to build the PlasmoDB.org database.  Each slide in the presentation is a graph.  The first slide is the root graph.    
  
Here is a sample graph:  [getAndAnalyzeSNPs.xml](https://www.cbil.upenn.edu/svn/apidb/ApiCommonWorkflow/trunk/Main/lib/xml/workflow/getAndAnalyzeSNPs.xml)  
  
Here is a sample step class:

## Additional Features

### Built in graph features

* conditional execution of steps
* [changing the graph after the workflow has begun](http://docs.google.com/a/apidb.org/Doc?docid=0AYEDs79fcBkZZGQ5OTZqeGdfMTk4ZjczcjdicDc&hl=en#Changing_a_graph_after_the_wor)
* support for incremental builds
  + [global steps](http://docs.google.com/a/apidb.org/Doc?docid=0AYEDs79fcBkZZGQ5OTZqeGdfMTk4ZjczcjdicDc&hl=en#Global_graphs_6851842810721177)
  + Global constants
  + [subgraph references](http://docs.google.com/a/apidb.org/Doc?docid=0AYEDs79fcBkZZGQ5OTZqeGdfMTk4ZjczcjdicDc&hl=en#Subgraph_references)
  + [grafting](http://docs.google.com/a/apidb.org/Doc?docid=0AYEDs79fcBkZZGQ5OTZqeGdfMTk4ZjczcjdicDc&hl=en#Grafting)

### Built in runtime features

* test mode
* offline steps
* undo
* failure recovery
* load balancing

### EuPathDB specific features

* [resource steps](http://docs.google.com/a/apidb.org/Doc?docid=0AYEDs79fcBkZZGQ5OTZqeGdfMTk4ZjczcjdicDc&hl=en#resource_steps_2664083237482203)
* [tracking analysis methods](http://docs.google.com/a/apidb.org/Doc?docid=0AYEDs79fcBkZZGQ5OTZqeGdfMTk4ZjczcjdicDc&hl=en#analysis_methods)

## Information flow in the graph

Information is shared among steps in the following ways:

* constant parameter values within the local scope of a graph
* parameter values passed down through subgraph calls
* global constants
* a target directory structure
* a target database

In addition, step class perl code gets a handle on these two property files:

* steps.prop
* stepsShared.prop

# Creating a workflow

Creating a workflow is an iterative process. In the early stages you define the root graph and subgraphs, (in XML) and also develop step classes (in Perl). Once you have a simple graph defined you set up a workflow home directory, do a little configuring and then compile the graph. This is a quick process that reports to you syntax errors in your graph. (It does not run the graph.) You can compile a graph without having written any step classes. Once you have a graph that compiles and you have written step classes for each step you run the graph in test mode. Test mode exercises all the step classes and checks correctness of data file passing across steps.

## Graph XML files

A graph XML file describes a set of steps that form a dependency graph. The XML file defines a local scope in which steps can refer to each other and in which parameter and constant values are visible. There are two types of steps: <step> and <subgraph>. The former calls a step class to do an atom of work. The latter invokes a subgraph, which is itself a graph defined in a graph XML file.

**Constructing a single graph file**

**Input parameters**

**Constants**

**Steps**

**IncludeIf/ExcludeIf**

**load balancing**

**Calling subgraphs**

**Subgraph references**

**Compiling a graph**

Checks that:

* the graph xml is valid XML
* that dependencies are correct
* that constants are correct
* that macros are correct
* that graph names are correct
* that variables are correct

You can "compile" your graph to test that all the variables, constants and subgraph references are valid:

$ workflow -h /files/cbil/data/cbil/TrypDB/wftest -c

An alternative way to do this, which also provides a detailed list of all the steps in the graph is:

$ workflowXml -h /files/cbil/data/cbil/TrypDB/wftest

**Types of errors**

In the following example, a variable has a problem.  Go to the step mentioned and see what is wrong with the variable.

$ workflow -h /files/cbil/data/cbil/TrypDB/wftest -c

Parameter 'genomeExtDbRlsSpecList' in step 'common.InsertGenegenomicsequenceWithSql' includes an unresolvable variable reference: '$$genomeExtDbRlsSpecList$$'

In this example, the mentioned step is calling a subgraph that expects the genomeExtDbRlsSpecList parameter, but the caller is not providing a value for it

$ workflow -h /files/cbil/data/cbil/TrypDB/wftest -c  
Graph "compilation" failed.  The following subgraph parameter values are missing:  
  
  File giardiadb/giardiaWorkflow.xml:  
      step: common  
          > genomeExtDbRlsSpecList

**Global graphs**

Use a *global graph* to make steps visible for dependency to steps anywhere in the workflow.    
  
A typical application would be a step that many steps throughout the graph depends on --and which may need to be undone regularly for iterative build-- but which cannot be local to their subgraph because they are in reused subgraphs.  (If the global step were in that subgraph it would be executed multiple times, each time the graph is reused.)    
  
Any step in the workflow may use <dependsGlobal> to declare a direct dependency on any step in a global graph, with these rules:

* all <dependsGlobal> elements in a <step> or <subgraph> must come after any <depends> elements.
* In <dependsGlobal name="xxxx"> xxxx must refer to the name of a step in the global graph

In the root graph, a subgraph may be declared to be global by using <globalSubgraph> instead of <subgraph>, with these rules:

* <globalSubgraph> is only allowed in the root graph
* there may be only one <globalSubgraph> and it must be the first step in the root graph.
* Unlike the <subgraph> element, the <globalSubgraph> element does not support <depends> or <dependsGlobal>.  That is, the step that calls the global subgraph may not have any dependencies.  (Inside the global graph there may be dependencies, like any other graph.)
* in the root graph, no step can depend on the <globalSubgraph>, ie, its name may not be referred to in a depends=
* A global graph XML has the same rules as a regular graph, except that a global graph may include <globalConstant> elements, with the following rules:
  + While a <constant> is visible only within the local XML file, a <globalConstant> is visible throughout the workflow
  + Use global constants sparingly, as they can lead to spaghetti.  Typically, only steps that have a <dependsGlobal> should refer to them.
  + In a graph XML file, all <globalConstant> elements must come before all <constant> elements.
  + <constant>, <param> and <globalConstant> elements may embed variables referencing <globalConstant> elements.
  + but <globalConstants> may not reference <contstants>
  + it is strongly recommended that globalConstants are given names that begin with global\_, eg, global\_NRDBOutputFile.  This way they will be recognizable where they are used.

**Workflow home directory**

**Config files**

The config files live in your\_workflow\_home/config  
  
This directory contains much valuable information.  You should consider having it under version control (SVN, CVS, etc).  If you do, be sure that the repository is secure, as these files may contain sensitive information such as login/password info and file paths.

**gus.config**

The workflow gets its database connection from the gus.config file ($GUS\_HOME/config/gus.config).  This connection is used for both the application data and for the workflow's internal database.

**initOfflineSteps**

   (steps to take offline at startup)

**loadBalance.prop**

   (configure load balancing)

**rootParams.prop**

    (root parameter values)

**stepsShared.prop**

Values in this files are visible in three types of files:

* graph xml files.  Here they are visible using the @@property\_name@@ syntax
* resource xml files.  Here they are visible using the @property\_name@ syntax
* step classes, using the getSharedConfig() method

**steps.prop**

         (steps config)

**workflow.prop**

      (meta config)

**Changing a graph after the workflow has begun**

A graph may be changed after a workflow has begun, with the following rules:

1. the graph must not be running
2. an existing step may be changed (or deleted) if it is in the READY or ON\_DECK state
3. new steps can be added to a graph in arbitrary ways with these rules:
   1. for all calls to the graph anywhere in the workflow, the return step must be in the READY or ON\_DECK state.  (if it is not, UNDO it)
   2. existing steps that are made to depend on new steps must comply with rule 2
4. the input parameters of a graph can be changed or deleted, and new ones added with these rules:
   1. calls to the subgraph *may* be in the DONE state
   2. all callers of the subgraph must be changed to correctly match the new expected parameters
   3. no changed or deleted parameters are consumed by a step that does not comply with rule 2

**Grafting**

**Subgraph references**

A <subgraph> element has an xmlFilename= attribute.  The value of this attribute is typically a hard-coded xml file name.  If the value of this attribute is a variable, then this <subgraph> element is a "subgraph reference."  This is analogous to a *method reference* in a programming language.  The pattern of its use is to embed in a standard graph a call to a graph that varies depending upon the context in which the standard graph is called.  It is particularly powerful if elements of the standard graph depend on the subgraph reference.  In this case, using a subgraph reference is critical.

**Nested data directory structure**

Files produced and consumed by steps are stored in a common directory structure.  A preferred design for this directory is to have its structure mirror the nesting of subgraphs.  This is accomplished by:

* each graph defining an input parameter called parentDataDir
* for the root graph, the value of parentDataDir is provided by the rootParams.prop file.  This is the root of the directory structure
* each graph calls a makeDataDir step that creates a dataDir which is a subdirectory of its parentDataDir, named for this graph.
* recursively, each graph passes its dataDir to its subgraph as their parentDataDir

Steps within the same graph can use their graphs dataDir variable to refer to each other's files.  A best practice is to use a constant for these file names to avoid file name mistakes  
  
The root of this tree is in workflow\_home/data.  The WorkflowStepInvoker superclass makes this available to all steps, via the getDataDir() method, which is relative to the root.

**Resource steps**

A set of steps in the workflow called *resources* acquire data from outside the workflow and make it available to other steps in the flow either as files or in the database.  Or they may simply load the data into the database for use after the workflow is complete.  The design pattern for a resource step is:

* a resource is defined using a <resource> element in a resource XML file.  For example, [datasources/plasmodb/pvivax.xml](https://www.cbil.upenn.edu/svn/apidb/ApiCommonData/trunk/Load/lib/xml/datasources/plasmodb/pvivax.xml)
* a resource XML file may contain many <resource> elements.
* a <resource> has a unique name that is an index into the resource XML file.
* a <resource> has a version
* the resource name and version are used in the database to populate the GUS tables SRes.ExternalDatabase and SRes.ExternalDatabaseRelease
* in the workflow graph, each resource is processed by a call to a subgraph called LoadResources.
  + LoadResource steps that are in a subgraph should all be in one resources XML file
* each step in the LoadResources subgraph is parameterized by
  + the resources XML file containing the resource definition
  + the name of the resource
  + the subgraph's dataDir
  + the globalDataDir
* the <unpack> and <pluginArgs> elements of a <resource> can use these macros:
  + @dataDir@
    - allows <unpack> and <pluginArgs> elements to share files
  + @globalDataDir@
    - allows <unpack> and <pluginArgs> elements to see files produced by global steps
  + and properties from the stepsGlobal.prop file.  (eg @soVer@)
* if <resource> elements within a resources XML need to share files they must navigate the nested directory structure to find the needed file, for example @dataDir@/../IEDB/iedb.fasta.  This is a workaround and should be avoided.
* with the exception of version= always use macros for a version, eg, %RESOURCE\_VERSION% or @goVer@.  The latter can be defined in stepsGlobal.prop.
* use a the parentResource= attribute of the <resource> to indicate that this resource is adding more data into a parent resource's dataset.  In this case, the child resource does not get written into the DataSource table, and its data uses the parent's external\_database\_release\_id.  The <pluginArgs> element gets access to the parent's extDbRlsSpec info by using the %PARENT\_RESOURCE\_NAME% and %PARENT\_RESOURCE\_VERSION% macros.
* the get and unpack phases of a resource might produce files that are consumed by dependent analysis steps in the graph XML.  In order for test mode to work properly, the resource step must produce dummy files for these files.  force that by using a <getAndUnpackOutput> element

**Displaying Data Sources on a website**

When the workflow loads a resource it records minimal information describing it:  only its name and version.  The <resource> element contains extensive provenance and descriptive information.  This is stored in the ApiDB.DataSource table (and its friends).  The information is placed there by the Tuning Manager, which reads the resources XML file from SVN (trunk).  It only includes information for resources that are found in SRes.ExternalDatabase.name.  This allows the resources XML file to accrue new resources after the flow is complete without those being seen on the website.

**Testing the syntax of a resources xml file**

Use the validateResourceXml program to compare your xml against the RNG schema definition ApiCommonData/Load/lib/rng/resources.rng.  
  
Here is its usage:  
  
usage: validateResourceXml -f resources\_xml\_file

**Tracking Analysis Methods**

Methodology information describing tools used in the workflow (eg, BLAST, Mummer) is stored by the workflow in the [ApiDB.AnalysisMethod](https://www.cbil.upenn.edu/svn/apidb/ApiCommonData/trunk/Load/lib/sql/apidbschema/createAnalysisMethod.sql) table (and friends).  It is placed there by an [InsertAnalysisMethod](https://www.cbil.upenn.edu/svn/apidb/ApiCommonWorkflow/trunk/Main/lib/perl/WorkflowSteps/InsertAnalysisMethod.pm) step.  This step typically depends on the completion of the step that loads the results of the analysis.  Any tool argument values that are recorded should be placed in a constant so they can be shared between the step that actually calls the tool and the step that records the methodology.

**Best Practices**

* if you change the input parameters in a subgraph, be sure to update all steps that call that subgraph, in all workflow XML files in the code base

**Testing a workflow**

Once your workflow compiles, run it in test mode.

1. set up gus.config file to talk to correct instance
2. open SQL Developer to see the workflow tables
3. reset the workflow, if there is any old one of this name in there
   1. you can reset a test workflow (but not a real one)
   2. CAUTION: this will WIPE OUT your test workflow's home dir (not config/) and the workflow tables
   3. (does reset only work in -t mode??)
   4. might need to manually clean up cluster dirs
   5. might need to manually clean up download dirs
4. run with -t flag
5. messages about Exclude indicate that steps excludeIf = true
6. ignore this error
   1. /EuPathDB/workflows/apitest/1.0-test  
      could not find ParserDetails.ini in /usr/lib/perl5/site\_perl/5.8.5/XML/SAX
7. handling initial failures
   1. likely to be systemic, ie, errors in stepsShared.prop
      1. fix these and -reset
      2. if you fix property files, you still need to change the individual FAILED step to READY or reset the whole test flow
8. if you change the graph.xml file (eg, correct a step class name), you need to bld and then restart the flow engine
9. workflowstep -h /files/cbil/data/cbil/EuPathDB/workflows/apitest/simple-test -p '%\_RSRC' ready
10. What happens if i kill the engine while my test is running?
    1. steps that are running continue to run safely
    2. they successfully update the workflow engine database
    3. just restart your workflow!
11. instead of grepping the controller.log for FAILED steps, use this command (the log may contain old info):
    1. workflow -h your\_workflow\_home -s FAILED
12. once you fix a step, you can change its status from FAILED to READY with this command.  (Use the full step name path):
    1. workflowstep -h your\_workflow\_home -p step\_name ready
13. if you made a fix that will correct a set of steps you can set them all to ready by using a pattern for the step name.  use % as a wildcard.  The following example finds any step with Nrdb anywhere in its full step name path
    1. workflowstep -h your\_workflow\_home -p %Nrdb% ready
    2. note that it is ok if your pattern finds steps that are not FAILED.  The output of the workflowstep command will show you some warnings like this, which you are free to ignore:
       1. Warning: Can't change PbergheiPostLoadGenome.genomeAnalysis.blastxGenomicSeqsNrdb from 'DONE' to 'READY'
14. To find out if the workflow is still processing steps, or if it is blocked by FAILED steps, run this command to find ON\_DECK steps.  (If there are none, then it is stalled, and you must fix FAILED steps to give it steps to process):
    1. workflow -h your\_workflow\_home -s ON\_DECK
15. if fixing a FAILED step involves correcting the step's graph XML, then you will need to restart the controller so it can pick up the new XML files (don't forget to do a bld first).  Just kill the controller and then restart it.
16. if fixing a FAILED step involves correcting a step that the FAILED step depends on, then you will need to UNDO the dependee.
17. to delete a step from the graph (or exclude it using includeIf or excludeIf):
    1. use the workflowstep -l option to find the state of the step
    2. if the step is DONE, undo it
    3. stop the controller
    4. if the step is FAILED, use workflowstep to set the step to ready
    5. delete the step from the graph (don't forget to do a build)
    6. start the controller
18. rules for changing the XML graph
    1. no steps may be in FAILED or RUNNING state
       1. for RUNNING steps, wait till they complete
       2. for FAILED steps
          1. stop the controller
          2. do any cleanup of the step that is needed
          3. use workflowstep to set the step to ready
    2. you may not change steps that are DONE
       1. use UNDO to set DONE steps to READY
    3. if may not change the graph after entering UNDO mode
       1. you may change the graph and then enter UNDO mode, but after the first time you run with -u, you may make no changes until the UNDO is complete

**Running a workflow**

Load balancing  
Taking steps offline  
Handling Step failure  
Undo  
Changing a graph  
Grafting

**GUS dependencies or assumptions**

* in ApiCommonData
  + schema definition
    - done - remove WorkflowStepDependency
    - done - add comment to keep ReFlow's version up to date
    - done - add new workflowInstallSchema command
  + done - add new properties for Workflow and WorkflowStep table names
  + done - upgrade java code to read those properties
  + done - upgrade perl code to read those properties
* in GUS/Workflow
  + Base
    - CBIL::Util::Properties
      * make copy in ReFlow?
    - hard codes location of db config to be gus.config
      * keep as is.  document carefully.  make modular in future
  + WorkflowStepInvoker
    - move to WorkflowStep.pm
      * getCluster
      * runCmdOnCluster
      * copyToCluster
      * copyFromCluster
      * runAndMonitorClusterTask
      * clusterTaskRunning
      * runPlugin
      * getUndoPlugin
      * getAlvInvIds
* in ApiCommonWorkflow
  + in Workflow.pm
    - move to WorkflowStepInvoker
      * getWorkflowDataDir
      * testInputFile
      * getInputFiles
      * getDataSource
      * DataSource.pm
  + resource acquisition
    - resource XML
    - manual delivery
    - plugins
    - external\_database
    - compute cluster