UNICORE WORKFLOW SERVICE MANUAL

UNICORE Team

Document Version:	1.0.0
Component Version:	8.1.0
Date:	23 02 2021

Contents

The UNICORE Workflow service provides advanced workflow processing capabilities using UNICORE resources. The Workflow service provides graphs of activities including high-level control constructs (for-each, while, if-then-else, etc), and submits and manages the execution of single UNICORE jobs.

The Workflow service offers a REST API for workflow submission and management and uses an easy-to-understand workflow description syntax in JSON format.

Thanks to a flexible internal workflow model and execution engine, the Workflow service can be in principle extended with custom workflow parsers and custom activities.

The Workflow service supports the full range of authentication options provided by UNICORE and uses JWT tokens for delegated authentication when submitting jobs to the execution sites.

For more information about UNICORE visit http://www.unicore.eu.

1 Installing and setting up the UNICORE Workflow engine

This chapter covers basic installation of the Workflow engine and the integration of the workflow services into an existing UNICORE system.

As a general note, the Workflow engine is based on a UNICORE/X instance. General UNICORE configuration concepts (such as user authentication, gateway integration, shared registry, attribute sources) fully apply, and you should refer to the UNICORE/X manual for such details.

1.1 Prerequisites

- Java 8 or later
- An existing UNICORE installation with Gateway, Shared Registry and one ore more UNICORE/X execution systems.
- (for production use) A server certificate

1.2 Updating from previous versions

If you update from 7.x, please note that it is a major update, and we suggest installing from scratch based on the template config files. The required changes are very similar to the UNICORE/X 7.x to 8.x update.

1.3 Installation

The workflow system is available either as part of the UNICORE Server Bundle, or as separate Linux packages (deb or rpm).

The basic installation procedure is completely analogous to the installation of a UNICORE/X server.

If you downloaded the UNICORE server bundle, untar the tar.gz, edit configure.properties and run configure.py

- please review the configure.properties file and edit the parameters to integrate the workflow services into your existing UNICORE environment. Then call `./configure.py` to apply your settings to the configuration files. Finally use `./install.py` to install the workflow server files to the selected installation directory.

If using the Linux packages, simply install using the package manager of your system and review the config files.

1.4 Setup

After installation, there are some manual steps needed to integrate the new server into your UNICORE installation.

- Gateway: edit `gateway/conf/connections.properties` and add the connection data for the Workflow server. For example,

```
WORKFLOW = https://localhost:7700
```

- XUUDB: if you chose to use an XUUDB for the Workflow server, you might have to add entries to the XUUDB to allow users access to the workflow engine. Optionally, you can edit the GCID used by the workflow/servorch servers, so that existing entries in the XUUDB will match.
- Registry: if the registry is setup to use access control (which is the default), you need to allow the Workflow server to register in the Registry. The exact procedure depends on how you configured your Registry, please cross-reference the section "Enabling access control" in the Registry manual. If you're using default certificates and the XUUDB, the required entries can be added as follows.

```
cd xuudb
bin/admin.sh add REGISTRY <workflow>/conf/workflow.pem nobody  ↩
    server
```

1.5 Verifying the installation

Using the UNICORE commandline client, you can check whether the new server is available and accessible:

```
ucc system-info -l
```

should include output such as

```
Checking for <Workflow submission> endpoint ...
... OK, found 1 endpoint(s)
 * https://localhost:8080/WORKFLOW/rest/workflows
  * server v8.0.0 CN=Demo Workflow,O=UNICORE,C=EU
  * authenticated as: 'CN=Demo User, O=UNICORE, C=EU' role='user'
```

The "authenticated as:" line should list you as "user".

Some more info about the server can be obtained via

```
ucc rest get https://localhost:8080/WORKFLOW/rest/workflows
```

1.5.1 Running a test job

Using UCC again, you can submit workflows

```
ucc workflow-submit /path/to/samples/date1.json
```

and get the ID of your new workflow back, e.g.

```
https://localhost:8080/WORKFLOW/rest/workflows/86686f72-b732-42e8 ↩
    -b14d-a8bd514e7edf
```

1.6 API documentation

Since version 8.0, the Workflow engine exclusively uses a RESTful API for all operation including job submission.

You can find an API reference and usage examples on the UNICORE Wiki at https://sourceforge.net/p/unicore/wiki/REST_API

2 Configuration of the Workflow server

This chapter covers configuration options for the Workflow server. Since the Workflow server is running in the same underlying environment (UNICORE Services Environment, USE), a lot of the basic configuration options are documented in the UNICORE/X manual.

NOTE

The configuration files in the distribution are commented, and contain example settings for all the options listed here.

Depending on how you installed the server, the files are located ion

- `/etc/unicore/workflow` (Linux package)
- `<basedir>/workflow/conf` (standalone installer)

2.1 Workflow processing

Some details of the workflow engine's behaviour can be configured. All these settings are made in `uas.config`.

2.1.1 Limits

To avoid too many tasks submitted (possibly erroneously) from a workflow, various limits can be set.

- `workflow.maxActivitiesPerGroup` limits the total number of tasks submitted for a single group (i.e. (sub-)workflow). By default, this limit is 1000, ie. a maximum number of 1000 jobs can be created by a single group. Note, that it is not possible to limit the total number of jobs for any workflow, it can only be applied to individual parts of the workflow (such as loops).
- `workflow.forEachMaxConcurrentActivities` limits the maximum number of tasks in a for-each group that can be active at the same time (default: 20).

2.1.2 Resubmission

The workflow engine will (in some cases) resubmit failed tasks to the service orchestrator. To completely switch off the resubmission,

```
workflow.resubmitDisable=true
```

To change the maximum number of resubmissions from the default "3",

```
workflow.resubmitLimit=3
```

2.1.3 Cleanup behaviour

This controls the behaviour when a workflow is removed (automatically or by the user). By default, the workflow engine will remove all child jobs, but will keep the storage where the files are. This can be controlled using two properties

- `workflow.cleanupStorage` remove storage when workflow is destroyed (default: false)
- `workflow.cleanupJobs` remove jobs when workflow is destroyed (default: true)

2.2 XNJS settings

The workflow engine uses the XNJS library for processing workflows. Some settings for modifying the behaviour are available, and are usually found in the workflow server's `container.properties` file

An important characteristic is the number of threads used by the workflow engine for processing.Note, this does not control the number of concurrent activities etc, since all XNJS processing is asynchronous. The default number (4) is usually fine.

This properties is set via

```
XNJS.numberofworkers=4
```

2.3 Property reference

A complete reference of the properties for configuring the Workflow server is given in the following table.

Property name	Type	Default value / mandatory	Description
`workflow.cleanupJobs`	[true, false]	`true`	Whether to remove child jobs when the workflow is destroyed.
`workflow.cleanupStorage`	[true, false]	`false`	Whether to cleanup the workflow storage when the workflow is destroyed.
`workflow.forEachMaxConcurrent`	integer >= 1	`10`	Maximum number of concurrent for-each iterations.
`workflow.maxActivitiesPerGroup`	integer >= 1	`1000`	Maximum number of workflow activities per activity group.
`workflow.pollingInterval`	integer >= 1	`600`	Interval in seconds for (slow) polling of job states.

Property name	Type	Default value / mandatory	Description
`workflow.resubmitDisable`	[true, false]	`false`	Whether to disable automatic re-submission of failed jobs.
`workflow.resubmitLimit`	integer >= 1	`3`	Maximum number of re-submissions of failed jobs.
`workflow.xnjsConfiguration`	string	`n/a`	(deprecated)

3 The workflow description language

3.1 Introduction

This chapter provides an overview of the JSON workflow description that is supported by the Workflow engine. It will allow you to write workflows "by hand", i.e. without using tools such as the Java or Python APIs.

After presenting all the constructs individually, several complete examples are given in Section 3.8.

3.2 Overview and simple constructs

The overall workflow document has the following form:

```
{
 "activities" : [],

 "subworkflows": [],

 "transitions": [],

 "variables" : [],

 "notification" : "optional_notification_url",

 "tags": ["tag1", "tag2", ... ],

}
```

Two special elements are

- "notification" (optional) denotes an URL to where UNICORE Workflow server will send a POST notification (authenticated via a JWT token signed by the Workflow server) when the workflow has finished processing. Notification messages have the following content

```
{
  "href" : "workflow_url",

  "group_id": "id of the workflow or sub-workflow",

  "status": "...",

  "statusMessage": "..."
}
```

- "tags" is an optional list of initial tags, that can later be used to conveniently filter the list of workflows.

Both of these are analogous to their conterparts for single jobs in UNICORE.

In the next sections the elements of the workflow description will be discussed in detail.

3.2.1 Activities

Activity elements have the following form

```
{
 "id" : "unique_id",
 "type": "...",

 ...

}
```

The `id` element must be UNIQUE within the workflow. There are different types of activity, which are distinguished by the "type" element.

- "START" denotes an explicit start activity. If no such activity is present, the processing engine will try to detect the proper starting activities
- "JOB" denotes a executable (job) activity. In this case, the `job` sub element holds the JSON job definition. (if a "job" element is present, you may leave out the "type")
- "ModifyVariable" allows to modify a workflow variable. An option named "variableName" identifies the variable to be modified, and an option "expression" holds the modification expression in the Groovy programming language syntax. See also the variables section later
- "Split": this activity can have multiple outgoing transitions. All transitions with matching conditions will be followed. This is comparable to an "if() ... if() ... if()" construct in a programming language.

- "Branch": this activity can have multiple outgoing transitions. The transition with the first matching condition will be followed. This is comparable to an "if() ... elseif() ... else()" construct in a programming language
- "Merge" merges multiple flows without synchronising them
- "Synchronize" merges multiple flows and synchronises them
- "HOLD" stops further processing of the current flow until the client explicitely sends continue message.

3.2.2 Subworkflows

The workflow description allows nested sub workflows, which have the same formal structure as the main workflow (without the "tags"). There is an additional "type" element that is used to distinguish the different control structure types.

```
{
 "id": "unique_id",
 "type": "...",
 "variables" : [],
 "activities" : [],
 "subworkflows": [],
 "transitions": [],
 "notification" : "optional_notification_url",
}
```

3.2.3 Job activities

Job activities are the basic executable pieces of a workflow. The embedded JSON job definition will be sent to an execution site (UNICORE/X) for processing.

```
{
 "id": "unique_id",
 "type" : "job",
 "job" : {
  ... standard UNICORE job ...
```

```
 },

 "options": { ... },

}
```

The execution site is specified by the optional "Site name" element in the job

```
{
 "id": "unique_id", "type" : "job",

 "job" : {

   "Site name": "DEMO-SITE",

    ...
 },

}
```

NOTE that there is currently no form of "brokering" in place, it is up to the user to select an execution site

The job description is covered in detail here: https://sourceforge.net/p/unicore/wiki/Job_Description

The processing of the job can be influenced using the (optional) "option" sub-element. Currently the following options (key-value) can be used

- `IGNORE_FAILURE` if set to "true", the workflow engine will ignore any failure of the task and continue processing as if the activity had been completed successfully. NOTE: this has nothing to do with the exit code of the actual UNICORE job! Failure means for example data staging failed, or no matching target system for the job could be found.
- `MAX_RESUBMITS` set to an integer value to control the number of times the activity will be retried. By default, the workflow engine will re-try three times (except in those cases where it makes no sense to retry).

For example

```
{
 "id": "unique_id",

 "job" : {

  ... standard UNICORE job ...

 },

 "options": { "IGNORE_FAILURE": "true",  },
```

```
}
```

If you need to pass on user preferences to the site, e.g. for selecting your primary group, or choosing between multiple user IDs, you can specify this in the "job" element like this

```
 ...

 "job": {

    "User prefences": {
      "uid":   "hpcuser21",
      "group": "hpc",
  }

 }
 ...

}
```

where the allowed field names are "role", "uid", "group" and "supplementaryGroups".

3.3 Files

Data management is one of the most important aspects of workflow definition and execution.

The Workflow service offers an additional tool for referencing data that is produced and consumed during workflow execution, which is workflow files.

A "workflow file" is a named reference to a physical file location, that can be generated at at any time during a workflow's lifetime.

Such a reference is an URI of the form `wf:...` which you can use in a job's "Imports" section or in For-Each file sets (see below).

In a Job you can use them as if they were a normal remote file:

```
{

 "Executable": "ls -l",

 "Imports":[
   { "From": "wf:infile1", "To": "infile1" }
   { "From": "wf:infile2", "To": "infile2" }
 ],

}
```

The Workflow engine will resolve these at runtime, and the job will thus import the file at the registered physical location.

There are several ways to define these workflow files.

3.3.1 Defining inputs

You can define inputs as part of your workflow

```
{
  "inputs": {
     "wf:infile1" : "https://remote_url_1",
     "wf:infile2" : "https://remote_url_2",
  },
}
```

(see the UCC manual for an example how a client might make use of this feature)

3.3.2 Defining workflow files as outputs

If your job / workflow step produces output, you can register these output files as workflow files, simply by "exporting" them.

```
{

 "Executable": "my_similation_code",

 "Exports":[
   { "From": "output.dat", "To": "wf:my_sim_output/output.dat" }
   { "From": "simlog.txt", "To": "wf:my_sim_output/log.txt" }
 ],

}
```

The Workflow service will not move the file, just register its location under the "wf:..." name, so you can reference it later.

3.3.3 Wildcard support

The workflow file registration and resolution mechanisms support wildcards.

For example

```
{

 "Executable": "my_similation_code",

 "Imports":[
     { "From": "wf:/stage1/*", "To": "/" }
 ],

 "Exports":[
     { "From": "output*", "To": "wf:my_sim_output/" }
```

```
 ],

}
```

3.3.4 Transitions and conditions

The basic flow of control in a workflow is handled using `transition` elements. These reference `from` and `to` activities or subflows, and may have conditions attached. If no condition is present, the transition is followed unconditionally, otherwise the condition is evaluated and the transition is followed only if the condition matches (i.e. evaluates to `true`).

The syntax for a Transition is as follows.

```
{

 "from" : "from_id",

 "to" : "to_id",

 "condition": "expression"

}
```

The `from` and `to` elements denote activity or subworkflow id's.

An activity can have outgoing (and incoming) transitions. In general, all outgoing transitions (where the condition is fulfilled) will be followed. The exception is the "Branch" activity, where only the first matching transition will be followed.

The optional `condition` element is a string-valued expression. The workflow engine offers some pre-defined functions that can be used in these expressions. For example you can use the exit code of a job, or check for the existence of a file within these expressions.

- `eval(expr)` Evaluates the expression "expr" in Groovy syntax, which must evaluate to a boolean. The expression may contain workflow variables
- `exitCodeEquals(activityID, value)` Allows to compare the exit code of the Grid job associated with the Activity identified by "activityID" to "value"
- `exitCodeNotEquals(activityID, value)` Allows to check the exit code of the Grid job associated with the Activity identified by "activityID", and check that it is different from "value"
- `fileExists(activityID, fileName)` Checks that the working directory of the Grid job associated with the given Activity contains a file "fileName"
- `fileLengthGreaterThanZero(activityID, fileName)` Checks that the working directory of the Grid job associated with the given Activity contains the named file, which has a non-zero length

- `before(time)` and `after(time)` check whether the current time is before or after the given time (in "yyyy-MM-dd HH:mm" format)
- `fileContent(activityID, fileName)` Reads the content of the named file in the working directory of the job associated with the given Activity and returns it as a string.

3.4 Using workflow variables

Workflow variables need to be declared using an entry in the `variables` array before they can be used.

```
{
 "name": "...",
 "type": "...",
 "initial_value": "..."
}
```

Currently variables of type "STRING", "INTEGER" , "FLOAT" and "BOOLEAN" are supported.

Variables can be modified using an activity of type `ModifyVariable`.

For example, to increment the value of the "COUNTER" variable, the following Activity is used

```
{
 "type": "ModifyVariable",
 "id": "incrementCounter",
 "variableName": "COUNTER",
 "expression": "COUNTER += 1;"
}
```

The "expression" contains an expression in Groovy syntax (which is very close to Java).

The workflow engine will replace variables in job data staging sections and environment definitions, allowing to inject variables into jobs. Examples for this mechanism will be given in the examples section.

3.5 Loop constructs

Apart from graphs constructed using `activity` and `transition` elements, the workflow system supports special looping constructs, for-each, while and repeat-until, which allow to build complex workflows.

3.6 While and repeat-until loops

These allow to loop a certain part of the workflow while (or until) a condition is met. A while loop looks like this

```
{
 "id": "while_example",

 "type" : "WHILE",

 "variables" : [
  {
    "name": "C",
    "type": "INTEGER",
    "initial_value": "1",
  }
 ],

 "body":
 {

   "activities":[
   {
      "id": "job",
      "job": { ... }
   },
   {
   # this modifies the variable used in the 'while'
   # loop's exit condition
  "id": "mod", "type": "ModifyVariable",
  "variableName": "C",
  "expression": "C++;",
   }
   ],

   "transitions: [
     {"from": "job", "to": "mod"}
   ]

   "condition": "eval(C&lt;5)",

}
```

The necessary ingredients are that the loop's "body" modifies the loop variable ("C" in the example), and the exit condition eventually terminates the loop.

Completely analogously, a repeat-until loop is constructed, the only syntactic difference is that the subworkflow now has a different `type` element:

```
{
 "id": "repeat_example",

 "type": "REPEAT_UNTIL",

 ...

}
```

Semantically, the repeat-loop will always execute the body at least once, since the condition is checked after executing the body, while in the "while" case, the condition will be checked before executing the body.

3.7 For-each loop

The for-each loop is a complex and powerful feature of the workflow system, since it allows parallel execution of the loop body, and different ways of building the different iterations. Put briefly, one can loop over variables (as in the "while" and "repeat-until" case), but one can also loop over enumerated values and (most importantly) over file sets.

The basic syntax is

```
{
 "id": "for_each_example",

 "type": "FOR_EACH",

 "iterator_name": "IT",

 "body": {

 },

# define range to loop over

 "values": [...],

# OR variables

 "variables": [...],

# OR files
```

```
 "file_sets": [...],

  # with optional chunking
 "chunking":

}
```

The `iterator_name` element allows to control how the "loop iterator variable" is to be called, by default it is named "IT".

3.7.1 The `values` element

Using `value`, iteration over a fixed set of strings can be defined. The main use for this is parameter sweeps, i.e. executing the same job multiple times with different arguments or environment variables.

```
"values": ["1", "2", "3", ],
```

In each iteration, the workflow variables "CURRENT_ITERATOR_VALUE" and "CURRENT_ITERATOR_INDEX" will be set to the current value and index.

3.7.2 The `variables` element

The `variables` element allows to define the iteration range using one or more variables, similar to a for-loop in a programming language.

```
"variables: [
 {
   "variable_name": "X",
   "type": "INTEGER",
   "start_value": "0",
   "expression": "Y++",
   "end_condition": "Y<2"
 },
 {
   "variable_name": "Y",
   "type": "INTEGER",
   "start_value": "0",
   "expression": "Y++",
   "end_condition": "Y<2"
 }
],
```

The sub-elements should be self-explanatory.

Note that you can use more than one variable range, allowing you to quickly create things like parameter studies,

3.7.3 The `file_sets` element

This variation of the for-each loop, allows to loop over a set of files, optionally chunking together several files in a single iteration.

The basic structure of a file set definition is this

```
"file_sets": [

 {
  "base": "...",
  "include": [ "..." ],
  "exclude": [ "..." ],
  "recurse": "true|false",
  "indirection": "true|false",
},

]
```

The `base` element defines a base of the filenames, which will be resolved at runtime, and complemented according to the `include` and/or `exclude` elements. The `recurse` attribute allows to control whether the resolution should be done recursively into any subdirectories. The `indirection` attribute is explained below.

For example to recursively collect all PDF files (except two files named "unused*.pdf") in a certain directory on a storage:

```
"file_sets": [

 {
    "base": "https://mysite/rest/core/storages/my_storage/files/pdf ↩
        /",
    "include": [ "*.pdf" ],
    "exclude": [ "unused1.pdf", "unused2.pdf", ],
    "recurse": "true"
  }

]
```

Of course you can use workflow files in the file sets as well. For example:

```
"file_sets": [

 {
    "base": "wf:/stage1/outputs/",
    "include": [ "*" ],
  }

]
```

The following variables are set where ITERATOR_NAME is the loop iterator_name defined in the for group as shown above.

- ITERATOR_NAME is set to the current iteration index (1, 2, 3, ...)
- ITERATOR_NAME_VALUE is set to the current full file path
- ITERATOR_NAME_FILENAME is set to the current file name (last element of the path)

3.7.4 Indirection

Sometimes the list of files that should be looped over is not known at workflow design time, but will be computed at runtime. Or, you wish simply to list the files in a file, and not put them all in your workflow description. The indirection attribute on a FileSet allows to do just that. If indirection is set to true, the workflow engine will load the given file(s) in the fileset at runtime, and read the actual list of files to iterate over from them. As an example, you might have a file filelist.txt containing a list of UNICORE file URLs:

```
https://someserver/file1
https://someserver/fileN
...
```

and the fileset

```
{
   "indirection": "true",
   "base": "https://someserver/rest/core/storages/mystorage/files/"
   "include": [ "filelist.txt" ],
}
```

You can have more than one file list.

3.7.5 Chunking

Chunking allows to group sets of files into a single iteration, for example for efficiency reasons.

A chunk is either a certain number of files, or a set of files with a certain total size.

```
 "chunking": {
  "chunksize": ... ,
  "type": "NORMAL|SIZE",
  "filename_format": "...,
  "chunksize_formula": "expression",
}
```

The chunksize element is either the number of files in a chunk, or (if type is set to "SIZE") the total size of a chunk in kbytes.

For example:

To process 10 files per iteration:

```
"chunking":
{
  "chunksize": "10",
}
```

To process 2000 kBytes of data per iteration:

```
"chunking":
{
  "chunksize": "2000",
  "type": "SIZE"
}
```

The chunksize can also be computed at runtime using the expression given in the optional `expression` element. In the expression, two special variables may be used. The `TOTAL_NUMBER` variable holds the total number of files iterated over, while the `TOTAL_SIZE` variable holds the aggregated size of all files in kbytes. The script must return an integer-valued result. The `type` element is used to choose whether the chunk size is interpreted as number of files or data size.

For example:

To choose a larger chunksize if a certain total file size is exceeded:

```
"chunking": {
  "expression": "if(TOTAL_SIZE>50*1024)return 5*1024 else return  ↩
     2048;"
  "type": "SIZE"
}
```

The optional `filename_format` allows to control how the individual files (which are staged into the job directory) should be named. By default, the index is prepended, i.e. "inputfile" would be named "1_inputfile" to "N_inputfile" in each chunk. The pattern uses the without extension and extension respectively. For example, if you have a set of PDF files, and you want them to be named "file_1.pdf" to "file_N.pdf", you could use the pattern

```
  "filename_format": "file_{0}.pdf"
```

or, if you prefer to keep the existing extensions, but append an index to the name,

```
  "filename_format": "{1}{0}.{2}"
```

3.8 Examples

This section collects a few simple example workflows. They are intended to be submitted using UCC.

3.8.1 Simple "diamond" graph

This example shows how to use transitions for building simple workflow graphs. It consists of four "Date" jobs arranged in a diamond shape, i.e. "date2a" and "date2b" are executed roughly in parallel. A "Split" activity is inserted to divide the control flow into two parallel branches.

All "stdout" files are staged out to the workflow storage.

```
{
"activities": [

  {
   "id": "date1",
   "job": { "ApplicationName": "Date" }
  },

  {
   "id": "date2a",
   "job": { "ApplicationName": "Date" },
  },

  {
   "id": "date2b",
   "job": { "ApplicationName": "Date" },
  },

  {
   "id": "date3",
   "job": { "ApplicationName": "Date" },
  }

],

"transitions": [
   {"from": "date1", "to": "date2a" },
   {"from": "date1", "to": "date2b" },
   {"from": "date2a", "to": "date3" },
   {"from": "date2b", "to": "date3" },
],

}
```

3.8.2 Conditional execution in an if-else construct

Transitions from one activity to another may be conditional, which allows all sorts of if-else constructs. Here is a simple example

```
{
```

```
"activities": [

  {"id": "branch", "type": "BRANCH" },

  {
   "id": "if-job",
   "job": { "ApplicationName": "Date" }
  },

  {
   "id": "else-job",
   "job": { "ApplicationName": "Date" },
  },

],

"transitions": [
   {"from": "branch", "to": "if-job", "condition": "2+2==4"},
   {"from": "branch", "to": "else-job" },
],

}
```

Here we use the "BRANCH" activity which will only follow the first matching transition.

3.8.3 While loop example using workflow variables

The next example shows some uses of workflow variables in a while loop. The loop variable "C" is copied into the job's environment. Another possible use is to use workflow variables in data staging sections, for example to name files.

```
{

"activities":[],

"subworkflows": [

  {
    "id": "while-example", "type": "WHILE",

    "variables": [
    {
       "name": "C",
       "type": "INTEGER",
       "initial_value": "0"
    }
    ],

    "condition": "C<5",
```

```
    "body": {

       "activities": [

       {
         "id": "job",
         "job": {
            "Executable": "echo",
            "Arguments": ["$TEST"],
            "Environment": ["TEST=${C}"],
            "Exports": [
              { "From": "stdout", "To": "wf:/out_${C}" }
            ]
          }
       },

       {
        "id": "mod", "type": "MODIFY_VARIABLE",
        "variable_name": "C",
        "expression": "C++"
       }

      ],

      "transitions": [
        {"from": "job", "to": "mod" }
      ],
      }

    }
}
```

3.8.4 For-each loop example

The next example shows how to use the for-each loop to loop over a set of files. The jobs will stage-in the current file. Also, the name of the current file is placed into the job environment.

```
{

"subworkflows": [

{
   "id": "for-example", "type": "FOR_EACH",
   "iterator_name": "IT",

   "body":
   {
```

```
     "activities": [

       {
         "id": "job",
         "job": {
             "Executable": "echo"
             "Arguments": ["processing: ", "$NAME"],
             "Environment": ["NAME=${IT_FILENAME}"],
             "Imports": [
               {"From": "${IT_VALUE}", "To": "infile"},
             ],
             "Exports": [
               {"From": "stdout", "To": wf:/out_${IT}},
             ],
         }
       },

      ],

    },
    "file_sets": [
      {
         "base": "https://mygateway.de:7700/MYSITE/rest/core/ ↩
             storages/my_storage/"
         "include": ["*"],
      }
    ],
}

}
```

4 Updating an existing UNICORE Workflow service

This chapter covers the steps required to update an existing workflow installation (v8.x).

4.1 Stop the server

Stop the workflow server.

4.2 Backup

You should make a backup of your existing data, and, if necessary, your config files.

4.3 Update jar files

The Java libraries have to be replaced with the new versions.

4.4 Update config files

Compare the config files from the new version to your existing one. Check the changelog for new features that might require updates to config files.

4.5 Restart the servers

Restart the workflow server, and check the logs for any suspicious error messages!