ETICS 2

JOB SUBMISSION INTERFACE

TECHNICAL NOTE

Document identifier:	**ETICS-job-submission-interface**
Date:	***31/03/2008***
Workpackage:	***XXX: WP Name***
Lead Partner:	***SHORT NAME***
Document status:	***Draft or FINAL***
Document link:	**https://edms.cern.ch/document/yyyyyy**

Abstract:

The purpose of this document is to present a specification for the Job Submission Service used by the ETICS system: a service to which clients like the ETICS Web Service can send build/test job requests. The specification defines the interface together with its classes.

Delivery Slip

	Name	WP/partner	Date	Signature
From	Lorenzo Dini Elisabetta Ronchieri, Valerio Venturi	SA1/CERN SA2/INFN		
Reviewed by	TC/PMB	TC/PMB		
Approved by				

Document Log

Issue	Date	Comment	Author

Document Change Record

Issue	Item	Reason for Change

CONTENT

1. INTRODUCTION

The Job Submission Service (JSS) specification defines Web Service interface for submitting build/test job requests to a submitter, which can be based on different architectures, like Metronome, Unicore, and glite. The JSS Web Service is agnostic to the underlying execution environment but characterizes the environment through a set of requirements and environments.

The document is composed of five Chapters as follows:

Chapter 1: Introduction

Chapter 2: Specification

Chapter 3: Basic Concepts

Chapter 4: Classes

Chapter 5: Interface

1.1. REFERENCES

[R1]	ETICS User Manual

2. SPECIFICATION

This document aims to describe the JSS interface that serves as the basis for further analysis.

2.1. STATE MODEL

The JSS specification addresses the definition of a basic state model (see Figure 1) traversed by the job during its life-cycle. The job status can be summarized as follows:

- Pending – The service has not enabled the job to start execution on such a platform.
- Running – The job is running on some platforms.
- Finished – The job has terminated successfully.
- Terminated – The client has requested to cancel the job.
- Failed – The job has failed due to some system error/failure event such as failure of a platform where the job was running on.

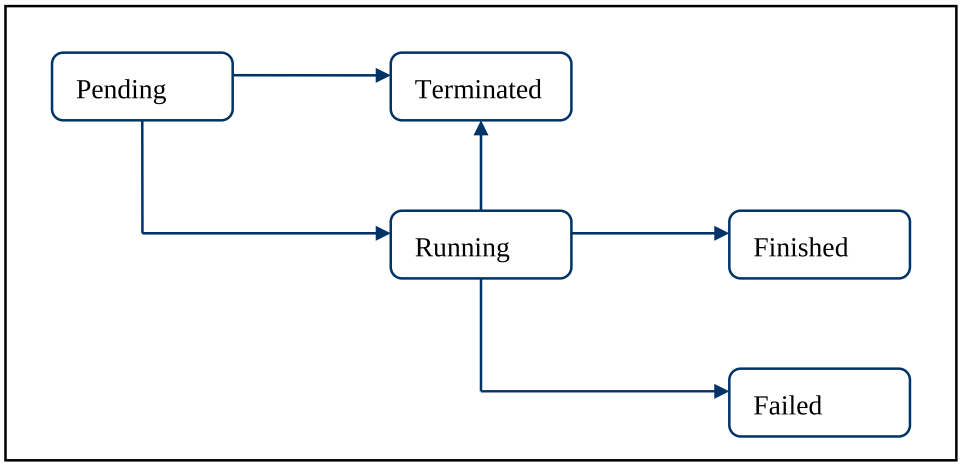

Figure 1: Basic State Model

3. BASIC CONCEPTS

In ETICS a software metadata model has been defined in order to capture the configuration information of software developed by projects. It is based on Common Information Model (CIM), characterized by formal entities able to express software project structure, build/test/checkout configurations, security information, dependencies, environment variables, properties, users and roles. The metadata model describes explicitly the objects and the relationships between objects (as in the CIM model).

3.1. PROJECT SOFTWARE METADATA

Software projects are known in ETICS as *projects*. A project can be considered as a high level container for software components and it's a good place to define properties and policies that must be applied by default to all components, as will be explained later.

A project can be further split in *subsystems* and *components* as shown in Figure 1. A subsystem represents a logical partition of the software architecture providing some subset of the project functionality. As for the project, subsystems can normally be considered containers for software components with special properties and policies that must be applied by default to all components of the subsystem. Components represent even smaller unit of functionality in the project architecture. A component can belong to the project or to an individual subsystem in the project. Normally they are used to generate individual software packages or small families of packages.

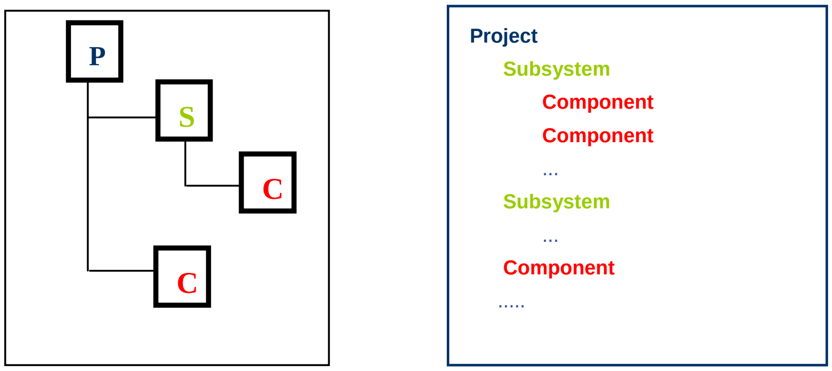

Figure 1: Project structure example.

Figure 1 shows the relationships among project, subsystem and component objects. A project has subsystems and components as children, whilst a subsystem only has components as children. A subsystem must belong to one and only one project and a component must belong to one and only one project or subsystem. Project, subsystem and component objects are generically referred to as *module*, each of which has the *name* entry.

A *configuration* in the ETICS metadata model is the set of information representing a specific version of a module. The configuration entity has three characteristics:

1. A configuration must be attached to one and only one module (i.e., project, subsystem or components);
2. Each module can have one or many configurations.
3. Each new module has a default configuration, called <*module-name*>.*HEAD*, like in CVS where each module always has the HEAD branch.

The configuration object like module object has the *name* entry.

The configurations of project and subsystem objects may contain a set of configurations related to project and subsystem children. In other words, project and subsystem configurations may be organized hierarchically. Figure 2 shows a project that contains a subsystem with a component; therefore the project configuration contains the subsystem configuration that contains the component configuration.

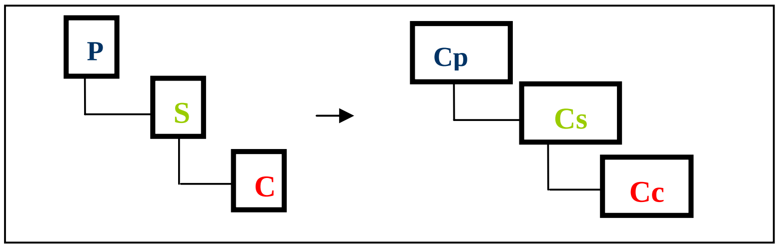

Figure 2: Example of mapping of project structure into configuration structure.

The ETICS metadata model defines also *platform* object representing particular combinations of OS types, architectures and compilers. Each platform is identified as a string of the form:

```
<os>_<arch>_<compiler>
```

For example the following platforms are currently defined:

```
slc3_ia32_gcc323
slc4_ia32_gcc346
```

The ETICS metadata model defines also *command*, *environment*, *property*, *dependency* objects that are attached to configuration by one or more platforms. Each configuration can have separate sets of the previous listed objects for each platform. Figure 3 shows a component configuration structure, while Figure 4 details a project configuration structure.

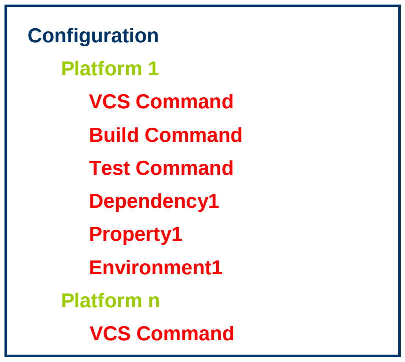

Figure 3: Component configuration structure example.

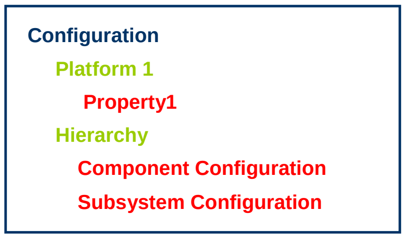

Figure 4: Project configuration structure example.

3.2. MULTI-NODE METADATA

The ETICS metadata defines three objects used by the multi-node functionality:

1. *deployment* that contains a set of nodes defining the test contest.
2. *node* that contains a set of services
3. service that represents the service to test

Each multi-node object has a configuration. The deployment configuration is handled as a subsystem configuration. The node configuration is handled as a component configuration.

3.3. SECURITY METADATA

The ETICS metadata model defines a user object. It contains user information such as name, email address and Distinguished Name of the standard x.509 certificate.

4. CLASSES

4.1. CONFIGURATION

The `Configuration` class represents a module configuration. Table 1 describes all `Configuration` class attributes.

Attribute	Type	Description
projectName	String	It is the name of the project to which the parent module belongs.
moduleName	String	It is the name of the parent module.
name	String	It is the name of the configuration set, which is unique in the ETICS metadata store.
deploymentType	String	It represents the type of deployment. It defines whether the job is normal or a multi-node.
description	String	It contains the description of the given configuration.

Table 1: Description of `Configuration` class attributes.

4.2. USER

The `User` class represents a user of the ETICS services. Table 2 describes all `User` class attributes.

Attribute	Type	Description
firstName	String	It is the first name of a user.
lastName	String	It is the last name of a user.
userDN	String	It is the distinguished name of a user.
emailAddress	String	It is the email address of a user, used for notification purposes. Therefore it muse be a valid address.

Table 2: Description of `User` class attributes.

4.3. JOB STATUS

The `JobStatus` class represents the status of a building / testing job. Table 3 describes all `JobStatus` class attributes.

Attribute	Type	Description
platform	String	It specifies the platform name, which the job is running on.
id	String	It is the unique identifier of the job.
status	String	It is the state of job.
details	String	It specifies some details of the submission.

Table 3: Description of `JobStatus` class attributes.

4.3.1. Basic States

The job life-cycle traverses the following set of states:

- Pending
- Running
- Finished that is a terminal state
- Terminated that is a terminal state
- Failed that is a terminal state

4.4. SUBMISSION

The `Submission` class contains all the needed information about module configuration, children configurations, requirements, options, user and platforms for the Submission Service interface. Table 4 describes all `Submission` class attributes.

Attribute	Type	Description
`user`	`User`	It contains information about user who has performed the submission.
`checkoutOptions`	`String`	It contains checkout options specified in the job.
`buildTestOptions`	`String`	It contains build and test options specified in the job.
`configuration`	`Configuration`	It specifies the module configuration.
`configurations`	`List<Configuration>`	It specifies children configurations, if present.
`requirements`	`Map<String, String>`	It contains submission requirements.
`environments`	`Map<String, String>`	It contains variables that must be available while the etics client is running into the worker node.
`platforms`	`List<String>`	It specifies the platforms where the job has been built / tested.

Table 4: Description of `Submission` class attributes.

The checkoutOptions and buildTestOptions are used by the ETICS client during the checkout and build/test operations respectively.

4.4.1. Checkout Options

In order to get the configuration information, source code and packages downloaded and installed onto the workspace, the command `etics-checkout` is used. It has many options, some of which may be included into the param `checkoutOptions` of the `Submission` class. A brief description of the command `etics-checkout` options is detailed in the following list (for more details look at [R1]:

```
-c, --config <configuration-name>
```

Define a specific configuration to be used instead of the default <module-name>.HEAD.

`--project <project-name>`

Specify a project name to be used instead of the current project.

`--project-config <project-configuration-name>`

Define a specific project configuration for the metadata.

`-p, --property <property=value>`

If already define, override existing properties, or define new ones.

`--platform <platform-name>`

Overwrite the local platform (useful for testing).

`--nocheckout`

Do not perform the actual VCS checkout and/or respository download.

`--nodeps`

Only checkout the currently specified module (do not checkout children and dependencies)

`--shallowbindeps`

When checking out using the **--**frombinary or –frombinaryonly options, dependencies of binary packages are not checked out.

`--runtimedeps`

When checking out configurations, dependencies of run-time dependencies are also processed.

`--ignorelocking`

If a configuration is locked, ignore its status and check it out agains the current project configuration configuration if specified.

`--volatile <namespace>`

Use a named volatile repository to look for packages.

`--defaultvolatile`

Use the default volatile repository to look for packages.

`--volatileonly`

Use only the specified volatile repository to look for packages.

`--force`

Force the configuration checkout or download even if no changes are detected.

`--forcevcscheckout`

Forces the checkout of the configuration from VCS, even if the corresponding module hasn't changed with respect to the workspace version.

`--forcedownload`

Forces the download of the configuration from the repository, even if the corresponding artefact hasn't changed with respect to the repository version.

`--noask`

Doesn't ask user confirmation and assumes YES to all questions

`--fromsource`

When possible, check out source code instead of downloading binaries.

`--frombinary`

When possible, download binaries instead of checking out source code. This implies that packages will not be built.

`--fromsourceonly`

Check out source code only.

`--frombinaryonly`

Checkout binaries only.

`--continueonerror`

Continue when checkout and/or download errors are encountered.

`--verbose`

Print verbose messages.

4.4.2. Build and Test Options

To submit remote build and test jobs the command `etics-submit` can be used. It has many options, some of which may be included into the param `buildTestOptions`. A brief description of the command `etics-submit` options is detailed in the following list (for more details look at [R1]):

`-c, --config <configuration-name>`

Define a specific configuration for the requested module.

`--project  <project-name>`

Define a specific project to build.

`--project-config <configuration-name>`

Define a specific project configuration for the metadata.

`-p, --property <property=value>`

Pass the specified property to the build process, if the property is already defined, it is overridden.

`-e, --env <env=value>`

Pass the specified variable to the build process, if the variable is already defined, it is overridden.

`-t, --target <target-name>`

Stop the build at the specified target.

`--platforms`

The list of platforms where to build.

`--continueonerror`

Do not stop building if an error is found.

`--force`

Force the build of unmodified modules.

`--cache`

Use cached checkout information when building to speed up the build (this is the default).

`--nocache`

Do not use cached checkout information when building, but recalculate build list and properties.

`--verbose`

Print verbose messages.

`--ipv6`

Run the remote build on IPv6-enabled nodes.

`--runasroot`

Run the remote build as super user (root).

`--freeze <time>`

Freeze the remote node after execution of the build.

`--shallowbindeps`

When checking out using the --frombinary or --frombinaryonly options, dependencies of binary packages are not checked out

`--runtimedeps`

When checking out configurations, dependencies of run-time dependencies are also processed.

`--ignorelocking`

If a configuration is locked, ignore its status and check it out agains the current project configuration configuration if specified.

`--volatile <namespace>`

Use a named volatile repository to look for packages.

`--defaultvolatile`

Use the default volatile repository to look for packages.

`--volatileonly`

Use only the specified volatile repository to look for packages.

`--register`

Register the build packages and reports in the permanent ETICS repository.

`--register-volatile <namespace>`

Register the build packages and reports in the named volatile repository instead of the default one.

`--private-resource`

Run the remote build on a private project node.

`--urlname <report-name>`

Specifies a custom string to be used in the URL for the published build and test reports.

`--fromsource`

When possible, check out source code instead of downloading binaries.

`--frombinary`

When possible, download binaries instead of checking out source code.

`--fromsourceonly`

Check out source code only.

`--frombinaryonly`

Checkout binaries only.

`--requirements`

Specify custom requirements for the matchmaking of the remote resources.

`--nobuild`

When <operation> is 'build', do not perform it, just print the sequence.

`--notest`

When <operation> is 'test', do not perform it.

Amongst the build and test options there are also packaging options:

`--createsource`

Create source tarball and RPMS.

`--nodistname`

Do not append the distribution name to the RPM revision

`--usetimestamp`

Append a timestamp to the RPM revision number

`--versioneddeps`

Use version information when setting package dependencies in RPMS

`--strictversioneddeps`

Use strict version information when setting package dependencies in RPMS

`--userspec`

Use a user-defined spec file.

`--usercontrol`

Use a user-defined control file for deb.

`--userchangelog`

Use a user-defined changelog file for deb.

`--userrules`

Use a user-defined rules file for deb.

4.4.3. Requirements

Requirements are needed for different behaviours to be done in the submitter, some of which are specified below:

`RUN_AS_ROOT`

It means that root is enabled onto the worker node (sudo client)

`IPV6`

It means that IPV6 is enabled onto the worker node.

`PRIVATE_RESOURCE`

e.g., `projectName` string as CLASSAD

`REGISTER`

`FREEZE`

It does not scratch the WN for n hours for debugging purposes.

`PRIVATE_RESULT`

It is not yet used. It should tell the repository how to set the visibility.

`URLNAME`

It has to be passed to the repository client – custom name to access reports

`VOLATILE`

It has to be passed to the repository client.

`CLIENT_RELEASE`

It specifies what version of the client to install (i.e., it needs to modify the scripts submitted).

`CLIENT_VOLATILE`

It specifies from what volatile to get the client.

4.4.4. Environments

5. EXCEPTIONS

The SubmissionException class contains three types of exceptions:

1. SubmissionTypeException

   Errors in the syntax or format of the inputs
2. SubmissionErrorException

   Errors during the submission
3. SubmitterErrorException

   Cannot submit this job in the underlying infrastructure due to a not supported feature

6. INTERFACE

The SubmitterService interface contains two methods:

`List<JobStatus> submit(Submission submission)`

- It gets `submission` as param
- It returns a list of job status param

`JobStatus getStatus(String platform, String id)`

- It gets `platform` and `id` as params
- It returns job status param

`void cancel(String platform, String id)`

- It gets `platform` and `id` as params

Faults are:

SubmissionException

CancelException

A Java Classes

B WSDL