Project Idea

A Jenkins plugin to wrap the https://github.com/cdancy/jenkins-rest library, allowing Pipeline scripts to interact easily with other (or the self) Jenkins instances via the Jenkins REST API, and eliminating the need to parse responses, making pipeline scripts more concise.

Note: there are other java libraries for the Jenkins REST. The author of this proposal was only familiar with the aforementioned library. There is another library called the java client api, hosted in jenkins, which could also be used. The student is expected to compare these libraries and make a recommendation prior to starting the work.

This proposal is not about filling gaps in the Jenkins DSL.

From the Jenkins Pipeline DSL, we can call the http request plugin to talk to other services. But this plugin does not “understand” the responses, it simply returns the header and the body of the response to the user. It is up to the user to parse the JSON and extract the relevant pieces.

Interestingly, Jenkins itself has a REST API, so a Pipeline DSL could use the http request plugin to talk to the Jenkins controller from an agent. But the user still has to parse the response.

An object oriented approach, leveraging an existing library that parses the responses into objects would be easier for the user.

As explained in the mailing list, it is essential for build steps to return only simple data types, like string, integers, lists of simple data types, maps of simple data types. Steps cannot return methods, nor objects with behaviors. Also, using the GlobalVariable is not recommended, however it might be inevitable. The student is expected to study the use of GlobalVariable in plugins that use it and ask for guidance on the Pipeline Authoring SIG gitter chat on this matter. The student should study the docker pipeline plugin source code and the pipeline loader plugin to understand.

In the following, we show how the finished plugin would look like from the user point of view in a Jenkins Pipeline DSL program. This is not a specification, it is only an example. The student is expected to study the the jenkins-rest library, the Jenkins Plugin tutorials and the Scripted Pipeline syntax, and propose a proper Jenkins Pipeline DSL syntax for this project.

Quick start

There are many technologies to use together to form this plugin. The student who wishes to get started will need to:

- Try out Jenkins REST APIs
- Try writing a small client using one of REST API Client libraries
- study plugin tutorials on how to write a Pipeline Step plugin
  - Tutorials listed on the student information page
  - Writing Pipeline compatible plugins
  - Writing Pipeline steps
  - Updating plugin for Pipeline

    - looking at existing pipeline compatible plugins will be very useful. Example:
        - External Workspace Manager (look at the steps folder, the steps themselves, and their execution classes)
- study the jenkins-rest library, try the examples in the wiki
- create a basic custom pipeline compatible plugin and load it in Jenkins (see the plugin tutorials)

Examples

The proposal for the implementation is to provide complete transparent access to all the jenkins-rest APIs by mapping them as Scripted Pipeline DSL steps, so as to make the following possible:

To create a client to a jenkins server use the jenkinsClient step:
```
def jenkinsClient = jenkinsClient url: “jenkinsurl”, username: “user”, password: “secret-or-api-token”
```

The jenkins client can set a verbosity level (see this wiki for a groovy example of the underlying implementation of the logger framework supporting the verbosity):
```
jenkinsClient.verbosity = “level”
```

The jenkins-rest client can set System properties to configure the underlying JClouds library:
```
jenkinsClient.setProperty("jclouds.so-timeout", "60000")
```

To query the system info, use the systemInfo step:
```
def resp = jenkinsSystemInfo client: jenkinsClient
```

The response is automatically parsed and the user can simply read the properties:
```
echo resp.errors
echo resp.errors[0].context
echo resp.errors[0].message
echo resp.jenkinsVersion
```

Of course, it gets more exciting when one realizes that jobs can be created and executed on other Jenkins instances:
```
String config = "... an xml file describing the job to be created..."
def response = jenkinsCreateJob client: jenkinsClient, folder: null, name: “myjob”, job: config
```

You can check the response for errors:
```
if (response.errors.size() == 0) {
   echo “Job created succesfully”
} else {
    echo response.errors[0].context
    echo response.errors[0].message
    // fail in some way
}
```

Trigger a build on the remote Jenkins server:
```
def queueId = jenkinsBuild client: jenkinsClient, folder: null, job: "myJob"
```

Deal with errors in job submission by reading `queueId.errors` (same as above in `reponse.errors`).

Poll the queue until the job runs or gets thrown out:

```
def queueItem = jenkinsQueueItem client: jenkinsclient, queueId: queueId.value()
while (true) {
    if (queueItem.cancelled) {
        error("Queue item cancelled")
    }

    if (queueItem.isExecutable) {
        echo("Build is executing with build number: " + queueItem.executable.number)
        break
    }

    sleep(10000)
    queueItem = jenkinsQueueItem client: jenkinsclient, queueId: queueId.value()
}
```

The queueItem can be converted to a buildInfo when the loop exits. This queueItem can be used to poll until the build is done:

```
def buildInfo = jenkinsBuildInfo client: jenkinsClient, folder: null, job: “myjob”, jobId:
queueItem.executable.number
while (buildInfo.result == null) {
    sleep(10000)
    buildInfo = jenkinsBuildInfo client: jenkinsClient, folder: null, job: “myjob”, jobId:
queueItem.executable.number
}
echo("Build status: " + buildInfo.result)
```

It is even possible to delete jobs:

```
def status = jenkinsDeleteJob client: jenkinsClient, folder: null, job: "myJob"
```

The polling algorithm could be embedded inside the plugin as a high level step. However, the plugin should expose all the rest apis as pipeline steps to give the user full flexibility to implement their own algorithms.

It may be possible to generalize REST APIs with:

```
withClient(....) {
  def res = restGet(“jobs/myJob/buildInfo”)
}
```

Expectations

The student is expected to compare the java libraries that implement the REST API and make a recommendation. They are 1) https://github.com/jenkinsci/java-client-api and 2) https://github.com/cdancy/jenkins-rest.

The student is expected to come up with a prototype to demonstrate the capability of the plugin, for example by implementing a single REST API method and a single response type, before proceeding with a complete implementation. This will help the student create proper step method calls and proper response objects.

Many interactions with a Jenkins server are possible via REST, see the full Javadoc for the Jenkins REST library. Having low level access to the Jenkins REST API allows users to interact with Jenkins in ways that are not provided by existing higher level plugins.

Advanced concept: it would be interesting to generate this plugin automatically simply by reading the bitbucket-rest library. This would enable automatic updates to this project each time the underlying library is updated, and it would also enable the automatic maintenance of the Jenkins-rest plugin and the Artifactory-rest plugin.

Links

- The two java rest api libraries:
  - https://github.com/jenkinsci/java-client-api
  - https://github.com/cdancy/jenkins-rest
- Mailing list discussion: https://groups.google.com/forum/#!topic/jenkinsci-dev/x-EbjnWcFqs
- Discussion on REST: https://groups.google.com/forum/#!topic/jenkinsci-dev/mYeM5qA6tGM

Open questions

- GlobalVariables have been debated on the mailing list, and their use is controversial. However they do work. Example of global variables usage in working plugins: docker, pipeline loader plugin.
- This approach has to be compared with the existing jenkins-java plugin.

Skills to study/improve

- Java
- REST API
- Jenkins Pipeline