**Introduction**

Jenkins is a popular open-source tool that helps teams with automation and implementation of continuous integration and deployment pipelines, comparable to for example Atlassian’s Bamboo, GitLab CI or to some extent Travis.

In this post, we share some practical lessons learned when integrating R applications via Jenkins for the purpose of continuous integration and regression testing on runner nodes configured using Jenkins via declarative pipelines defined in a Jenkinsfile.

**Contents**

1. [Propagating environment variables to R sessions](https://jozef.io/r918-jenkins-pipelines/#propagating-environment-variables-to-r-sessions)
2. [Checking and accessing the propagated variables](https://jozef.io/r918-jenkins-pipelines/#checking-and-accessing-the-propagated-variables)
3. [Using a per-pipeline R library](https://jozef.io/r918-jenkins-pipelines/#using-a-per-pipeline-r-library)
4. [Working with parametrized builds from R](https://jozef.io/r918-jenkins-pipelines/#working-with-parametrized-builds-from-r)
5. [References](https://jozef.io/r918-jenkins-pipelines/#references)



Example jenkins pipeline. Image credit <https://bit.ly/2fpnBWI>

**Propagating environment variables to R sessions**

When running R code on a local machine or a remote server from a user perspective, we count on a lot of configuration that is already present potentially without the user even noticing or knowing about the details of that configuration. One example of such configuration is the environment variables that configure some of R’s behavior.

When running R code on a computer that is connected to the Jenkins server as a node (a place where Jenkins sends the jobs to run), those environment variables likely need to be passed to the worker process, including configuration present for example in .Renviron files and .Rprofile files.

To propagate environment variables to all the stages of a declarative pipeline, we can use the environment directive in the pipeline definition. For example, to propagate a path to a user library, an example Jenkinsfile could look as follows:

pipeline {

environment {

R\_LIBS\_USER = '/path/to/lib'

}

// ... pipeline continues ...

}

This will ensure that the environment variables defined will be propagated to all the stages defined in the pipeline.

Note: We might be tempted to simply use EXPORT on the variables that need to be propagated to other stages. While this will likely work in a classic setup where we are running multiple R scripts under the same shell, Jenkins runs each of the stages in a separate shell, meaning that EXPORT does *not* ensure that the variables will be propagated to other stages. The same of course applies to using Sys.setenv() from R itself.

**Checking and accessing the propagated variables**

To test whether our environment variables were propagated as intended, we can use printenv, for example in a stage dedicated to showing the environment variables:

pipeline {

environment {

R\_LIBS\_USER = '/path/to/lib'

}

agent any

stages {

stage('Show env vars') {

steps {

sh 'printenv'

}

}

}

}

From R, we can access the environment variables using Sys.getenv():

# List all environment variables

Sys.getenv()

# Get a specific one

Sys.getenv("R\_LIBS\_USER")

**Using a per-pipeline R library**

For continuous integration purposes, it is useful to get our code checked out and tested on each commit. To get our packages installed into a separate library for each branch, one of the options is setting a user library path.

Doing that we can also choose the granularity of the separation we want to achieve. For example, using a library per branch in a multibranch pipeline context:

environment {

R\_LIBS\_USER = """${sh(

returnStdout: true,

script: 'echo $PWD/test-lib'

)}""".trim()

}

Using this would mean the same library is used for each build of the same branch. If we need more granularity we can use a library per both branch and build adding the BUILD\_ID variable to the path:

environment {

R\_LIBS\_USER = """${sh(

returnStdout: true,

script: 'echo $PWD/$BUILD\_ID/test-lib'

)}""".trim()

}

Note the need to apply the trim() method on the constructed paths to strip whitespaces/linebreaks that get produced when retrieving the value from standard output.

**Working with parametrized builds from R**

Jenkins also offers the option to parametrize builds, such that parameters of several types can be passed as environment variables to the shell through which the staged jobs are executed.

For usage with R applications, this means we can retrieve such parameters using the Sys.getenv() function. For example, if we create a parameter named r\_num\_cores in Jenkins, we can easily access its value within the build:

Sys.getenv("r\_num\_cores")

A small caveat to this is that all the parameters are passed as strings, so in case we want to pass R objects as parameters (for example a vector c(1, 2)), we would need to parse the string values, for example writing a wrapper function. A naive implementation of such wrapper can look as follows:

env\_get <- function(varName, parse = TRUE) {

res <- Sys.getenv(varName)

if (isTRUE(parse)) res <- eval(parse(text = res))

res

}

It is also worth noting that syntactical differences can require some further tweaking, for example, boolean Jenkins parameters are passed as "true" or "false", so would not work with the eval(parse(...) approach unless changed to uppercase first.

**References**

* Jenkins documentation on [Creating multibranch pipelines](https://jenkins.io/doc/book/pipeline/multibranch/#creating-a-multibranch-pipeline)
* Jenkins documentation on [Declarative pipelines](https://jenkins.io/doc/book/pipeline/syntax/#declarative-pipeline)
* Jenkins documentation on [Setting environment variables](https://jenkins.io/doc/book/pipeline/jenkinsfile/#setting-environment-variables)
* Jenkins documentation on [Parametrized builds](https://wiki.jenkins.io/display/JENKINS/Parameterized+Build)