Jenkins Pipelines and their dirty secrets

An image you'll not be happy to encounter...

I just had started my first job as a software developer, when the first news about the deprecation of the Build Flow Plugin came around. At that time the company had already had a huge, complex CI, with a lot of workflow organized Jenkins jobs. It was clear that a quick fix will be to simply *replace the build flow jobs with pipelines, and keep the party going as it is*. And of course who would be the best to fix this minor issue, if not the new intern! The stage was set, and I was sent to wrap the job up.
…but there was a catch

As I realized no one knew the syntax of this esoteric pipeline scripts yet, so I had to figure out how to use them to exactly replace the build flow scripts with it. There came a lot of interesting information about pipelines, that made me rethink how it has to be used to replace certain elements in build flows. During my research, I had not found any comprehensive source on the web about specific use-cases of certain pipeline steps. Here I'm planning to collect and share all my findings. I will specifically concentrate on how to replace certain solutions done before by build flows, or other old deprecated plugins like Active Choice.

Anyway I know you didn't come for the story so let's see the code!

Pipeline Basics:

The first thing you should do if you are getting started with pipelines is to read the official pipeline documentation. Here you can learn what a pipeline is, what the Jenkinsfile is, and all the basics, that I'm not going to explain in detail.

This documentation, however, has a dirty little secret, a piece of information you can easily disregard. ***Pipeline scripts can be written in two ways: Declarative Style and Scripted Style.***

Example of a Declarative Pipeline script:

Luckily Declarative pipelines are well documented, you can find a lot of examples and documentation on this page. Anyhow, here is an example by me. Notice that here **I specifically created a script which is**

implementing a job scheduling solution, just as a build flow scripts did before.

Also notice that a Declarative Pipeline always has a `pipeline {}` block with an `agent any` part in it, and is separated into stages.

```
#!/usr/bin/env groovy

pipeline {
    agent any
    stages {
        stage("build") {
            steps {
                retry(3) { build("build-job") }
            }
        }
        stage("tests") {
            steps {
                parallel (
                    "unit test" : {
                        build("unit-test-job")
                    },
```

Example of a Scripted Pipeline script:

Scripted Pipeline syntax, on the other hand, offers a general purpose DSL syntax. Scripted Pipelines always are enclosed within a `node {}` block. Notice that I used a `stage` but it is not necessary. You can read about specific Scripted Pipeline steps, at the documentations page, but why would you do that? From any Jenkins you can reach a pipeline snippet generator, which helps you figure out the syntax. Anyway here is an example.

```
#!/usr/bin/env groovy

node {
    def workspace = pwd()
    echo "Building Job at ${workspace}"
    build 'builder-job'

    stage('Shell') {
        try {
            sh returnStdout: true, script: 'demo.sh'
        }
```

Usefull to know, that—***yes!***—You can have both syntaxes in a Jenkinsfile.

Scripted Pipeline IN Declarative Pipeline:

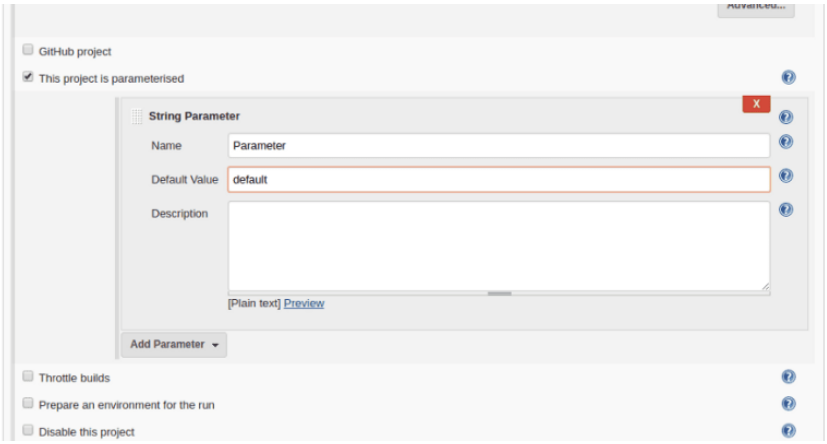
In case it is necessary you can actually create an embedded Scripted Pipeline block IN a Declarative Pipeline step! We use the `script{}` block for this. Here's how it is:

```
#!/usr/bin/env groovy

pipeline {
    agent any
    stages {
        stage("robot test") {
            steps {
                script {
                    MYLIST = []
                    MYLIST += "param-one"
                    MYLIST += "param-two"
                    MYLIST += "param-three"
                    MYLIST += "param-four"
                    MYLIST += "param-five"

                    for (def element = 0; element < MYLIST
                        build(
                            job: 'parameterized-job',
                            parameters: [
                                [
                                    $class: 'StringParamet
```

Notice that here I used the scripted block to be able to use for loop.

Run parameterized jobs:

When you create a pipeline, you can set it to be parameterized:



If you want to use these parameters in the pipeline script you can refer to them as `params.your_parameter`. Now I'm going to confuse you: **You can declare parameters in your pipeline script too!**

```
#!/usr/bin/env groovy
pipeline {
    agent any
        parameters {
            string(
                name: 'MyString',
                defaultValue:"default",
                description: "Any")
...
```

Indeed nothing stops you from putting a parameters block in your pipeline script. So what will happen if you build the job now? Here's the answer: In this case, when your pipeline job builds for the first time, **it will clear all the previous config set in your job, parameterize it according to the parameters block of the pipeline script, and run it with its default parameters**. If you do this and have a look at your job's Console Output, you will see the following message, which warns you that your config in the Jenkins job will be lost, and replaced with the one in the pipeline script: `WARNING: The properties step will remove all JobProperty currently configured in this job, either from the UI or from an earlier properties step.`

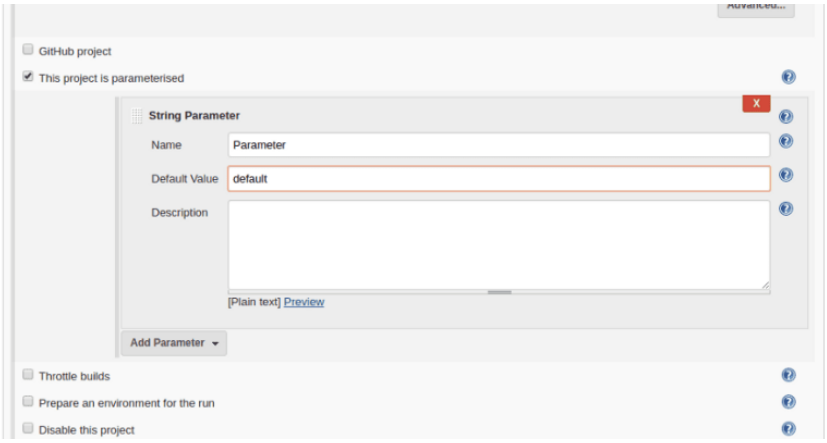
Anyway, this is how you set parameters in your Pipeline Script, and use it, for example to run another job with parameters:

```
#!/usr/bin/env groovy

pipeline {
    agent any
        parameters {
            choice(
                name: 'Nodes',
                choices:"Linux\nMac",
                description: "Choose Node!")
            choice(
                name: 'Versions',
                choices:"3.4\n4.4",
                description: "Build for which version?" )
            string(
                name: 'Path',
                defaultValue:"/home/pencillr/builds/",
                description: "Where to put the build!")
    }
    stages {
        stage("build") {
            steps {
```

Initialize parameterized jobs:

One important thing to know: It is handy to create the pipeline job as a parameterized job, with all these parameters in it, because if you don't do it, you will *not be able to choose parameters when you build the job the first time!*

Set your job to be parameterized:



Anyway, there is another way *(explained in the above section too!)*:
When a job, -that has parameters block in it's Jenkinsfile—runs, **it**

clears all the previous parameters set in the job's config and overwrites it with the ones in the Jenkinsfile. This is a kind of first *dry run*, which sets the parameters, and at the second run, you can run the job as a parameterized job **even if you didn't create it as one**. Unfortunately, in this case your job basically will just start running with its default parameters. This can be bad, as you may not want that.
For this scenario I offer a workaround:

Initialize a parameterized job on the run:

Create an Initializer choice parameter in the jenkinsfile with a "Yes" default.

```
choice(name: 'Invoke_Parameters', choices:"Yes\nNo", description:
"Do you whish to do a dry run to grab parameters?" )
```

Then create a first `stage`, which checks if this is an initial run. If it is, it will abort the build.

```
1       stages {
2           stage("parameterizing") {
3               steps {
4                   script {
5                       if ("${params.Invoke_Parameters}" == "
6                           currentBuild.result = 'ABORTED'
7                           error('DRY RUN COMPLETED. JOB PARA
8                       }
```

As this is after the `parameters` block, your job will be parameterized when you run it the second time, and you can unset the Invoke_Parameters choice parameter.

Inheriting job variables from previous jobs:

Previously in build flows, it was a common thing, that you could pass parameters to jobs, and get the resulting AbstractBuild when required.

```
b = build( "job1", param1: "foo", param2: "bar" ) build(
"job2", param1: b.build.number )
```

There is a way for this in Pipeline. You can reach these as `buildVariables`:

```
#!/usr/bin/env groovy

pipeline {
    agent any
    stages {
        stage("build") {
            steps {
                script {
                    def b = build(job: "parameter-source-j
                    build(
                        job: "analyzer-job",
                        parameters: [
                            [
                                $class: 'StringParameterVa
                                name: 'PARAM_ONE',
                                value: b.buildVariables.PA
                            ],
                            [
                                $class: 'StringParameterVa
                                name: 'PARAM_TWO',
                                value: b.buildVariables.PA
```

Pipeline asking for input during build:

Pipeline scripts offer a syntax which is useful, in case you need to ask for permissions or data during the build.
Imagine a scenario in which **I need to choose some parameters for the build depending on some other parameter I previously have chosen**.
This is where the user input choice comes handy.

- Here first a list of nodes (strings) are generated by a bash script in the node block, during an initial run, stopped by the parameterizing stage.
- During the second run, we choose a Node parameter from this list of nodes, as we run the job as a parameterized build.
- After this, in the `choose version` stage, another bash script is run with the `node` parameter as its first parameter, which generates the list of node versions for that node.
- At the `input message` part the job stops, and we can choose from these versions.
- In the end the job runs another job with our parameters.

```
#!/usr/bin/env groovy

def nodes
def versions

node {
    dir('/home/pencillr/workspace') {
        nodes = sh (script: 'sh list_nodes.sh', returnStdo
    }
}

pipeline {
agent any
    parameters {
            choice(name: 'Invoke_Parameters', choices:"Yes
            choice(name: 'Nodes', choices:"${nodes}", desc
    }
    stages {
        stage("parameterizing") {
            steps {
                script {
                    if ("${params.Invoke_Parameters}" == "
                        currentBuild.result = 'ABORTED'
                        error('DRY RUN COMPLETED. JOB PARA
                    }
                }
            }
        }
        stage("choose version") {
            steps {
                script {
                    def version_collection
                    def chosen_node = "${params.Nodes}"
                    dir('/home/pencillr/workspace') {
```

Handling build dependencies:

Now before we start, let's keep in mind that pipeline scripts can be formulated in either *Declarative* or *Scripted* syntax. The first example code in this section is formulated in Declarative Pipeline syntax. If you don't know what the difference is, read up the first chapter. Let's see the following problems:

1. You have two jobs following each other:

```
build(job: 'job-1', propagate: false)
// This step will run even if job-1 failed
build(job: 'job-2')
```

In this case, the first build step will be executed first, but the whole job will stop and fail if the first job fails. But let's say that you don't want this. **Instead, you want the following**:

- You want both of them to run, even if the first fails, and only fail if the second job fails
- You want the jobs to run after each other but not parallel.

Here's your solution: Use the `propagate` attribute, and set it to `false`. If disabled, then this step succeeds even if the downstream build is unstable, failed.

```
build(job: 'job-1', propagate: false)
// This step will run even if job-1 failed
build(job: 'job-2')
```

This case the job will start the second build step even if the first one failed. *But there is a catch*. If the first one failed, you will not know this from the pipeline's result. The pipeline job's result will evaluate to be a `SUCCESS` even if the first job fails. This is not necessarily a problem, but the third example will offer a solution for this issue.

2. You want to run parallel builds:

```
#!/usr/bin/env groovy

pipeline {
    agent any
    stages {
        stage("test") {
            steps {
                parallel (
                    "Unit Test" : {
                        build("unit-test-job")
                    },
                    "Component Test" : {
                        build("component-test-job")
                    },
                    "Build" : {
```

Yes but is there a catch? Yes there is:
For one, you can't have anything else, but that parallel block in that stage. If you would add a separate build step, like this:

```
//    !!!!!!!!!!!!!!!!!!!!!!!!
//    DONT USE! BAD CODE

            steps {
                parallel (
                    "Unit Test" : {
                        build("unit-test-job")
                    },
                    "Component Test" : {
                        build("component-test-job")
                    },
                    "Build" : {
                        build("build-job")
```

…you would get the following ERROR:

`WorkflowScript: 6: Invalid step "parallel" used - not allowed in this context - The parallel step can only be used as the only top-level step in a stages step block.`

A quick solution is to reformulate your script in Scripted pipeline syntax:

```
node {
// HEADS UP: this is Scripted Pipeline syntax
    stage("Build") {
        parallel (
            "First Build" : {
                build("first-build-job")
            },
            "Second Build" : {
                build("second-build-job")
            }
        )
        build("test-job")
        parallel (
            "Third Build" : {
                build("third -build-job")
```

3. You want to examine the build

The build step has a hidden attribute: **`wait`** (just like `propagate`, which you have seen before in this chapter).

```
build(job: 'example-job', wait: true)
```

WARNING: I've seen online that many people confuse `wait` with `propagate`, when they want to ignore the failure of a job in the pipeline. Let's see the differences:

> `build(job: 'example-job', wait: false)` : This means that the pipeline **will not wait for the result of this build step, just starts it and jumps to the next step**. If you put blocks like this after each other, they will be started one after another, without waiting for each other to finish (Actually almost like a parallel block).
> `build(job: 'example-job', propagate: false)` : This means that the pipeline will **first execute this step**, and **continue to the next step even if this step failed**.

`wait` is **true as default**, so normally you don't have to add it to the code. In case you don't turn it false, the return value of the build step will be an object on which you can obtain read-only properties: so you can inspect its .result and so on.

```
node { stage("example") { echo build("example-job").result }
```

With this, you can actually achieve to **set the build's result even if you ignored failing builds with setting** `propagate: false` : Basically what you have to do is just to set the jobs result by hand after examining given conditions with an `if` block.
This way the pipeline will keep running even if the first job fails, but you will see from the result of the pipeline if something was wrong.

```
1  node {
2      stage("example") {
3          b = build(job: "example-job", propagate: false).re
4          if(b == 'FAILURE') {
5              echo "First job failed"
6              currentBuild.result = 'UNSTABLE' // of FAILURE
7          }
8      }
9      stage("test") {
```

Originally published at dev.to.