Converting Conditional Build Steps to Jenkins Pipeline

Published on 2017-01-19 by Liam Newman Tweet

- pipeline
- freestyle
- plugins
- conditional-build-step
- tutorial

This is a guest post by Liam Newman, Technical Evangelist at CloudBees.

Introduction

With all the new developments in Jenkins Pipeline (and Declarative Pipeline on the horizon), it's easy to forget what we did to create "pipelines" before **Pipeline**. There are number of plugins, some that have been around since the very beginning, that enable users to create "pipelines" in Jenkins. For example, basic job chaining worked well in many cases, and the Parameterized Trigger plugin made chaining more flexible. However, creating chained jobs with conditional behavior was still one of the harder things to do in Jenkins.

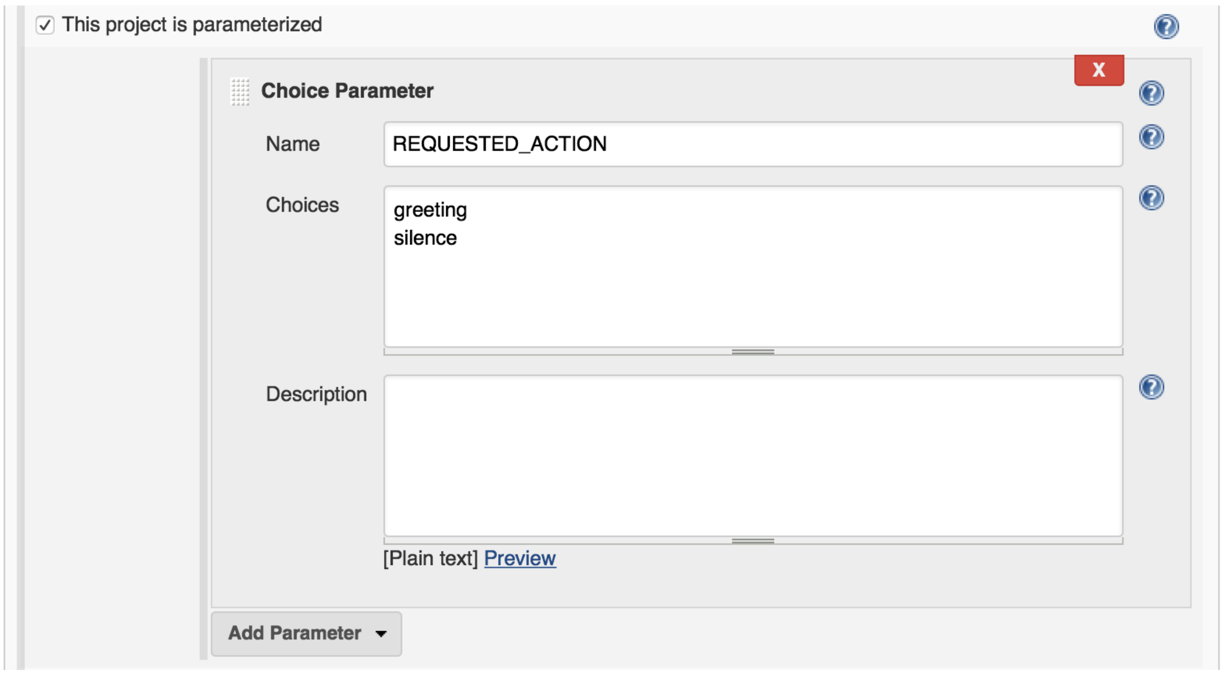
The Conditional BuildStep plugin is a powerful tool that has allowed Jenkins users to write Jenkins jobs with complex conditional logic. In this post, we'll take a look at how we might converting Freestyle jobs that include conditional build steps to Jenkins Pipeline. Unlike Freestyle jobs, implementing conditional operations in Jenkins Pipeline is trivial, but matching the behavior of complex conditional build steps will require a bit more care.

Graphical Programming

The Conditional BuildStep plugin lets users add conditional logic to Freestyle jobs from within the Jenkins web UI. It does this by:

- Adding two types of Conditional BuildStep ("Single" and "Multiple") - these build steps contain one or more other build steps to be run when the configured condition is met
- Adding a set of Condition operations - these control whether the Conditional BuildStep execute the contained step(s)
- Leveraging the Token Macro facility - these provide values to the Conditions for evaluation

In the example below, this project will run the shell script step when the value of the `REQUESTED_ACTION` token equals "**greeting**".



☑ This project is parameterized

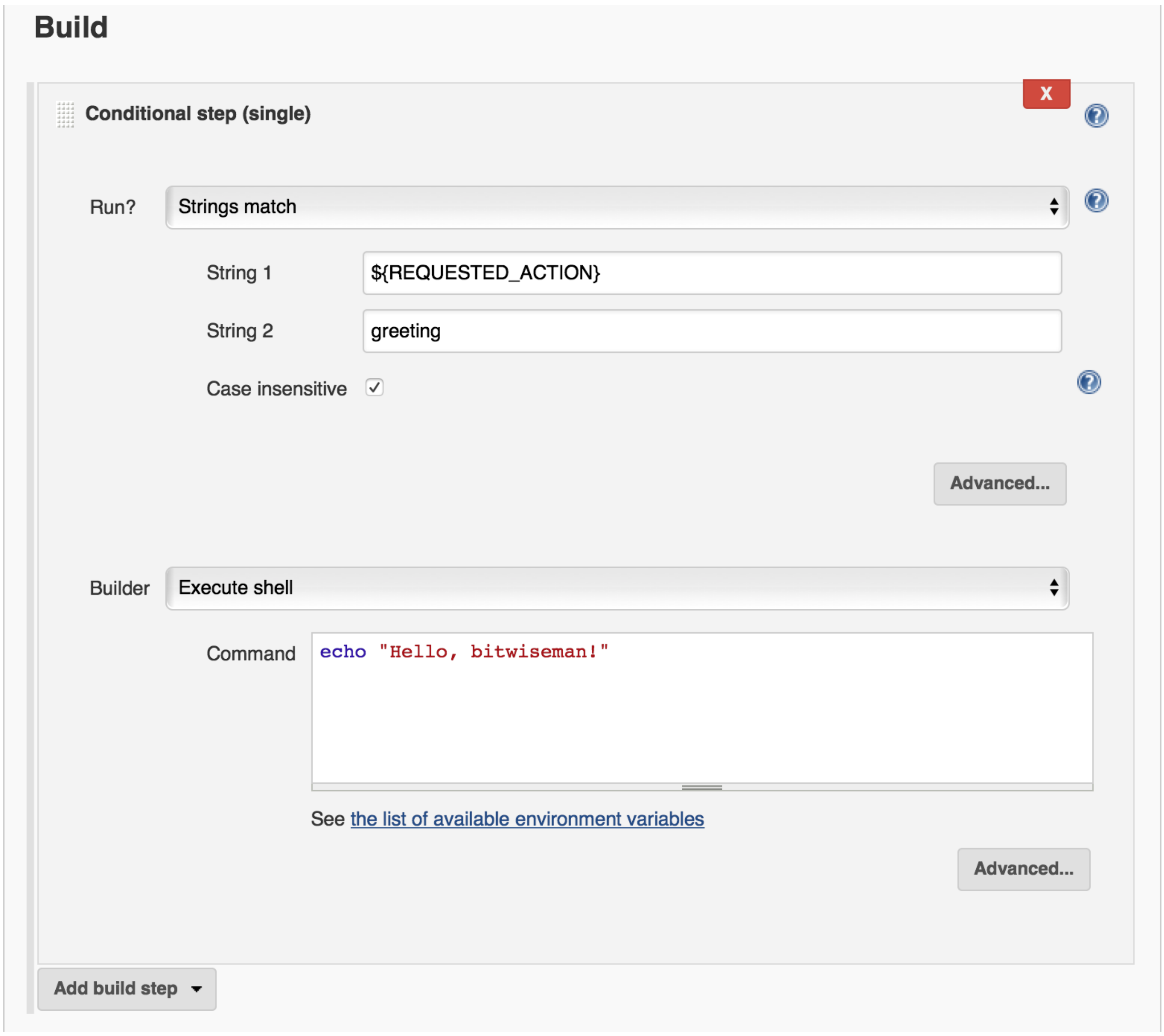
Choice Parameter

Name: REQUESTED_ACTION

Choices:
greeting
silence

Description:

[Plain text] Preview


Build

Conditional step (single)

Run? Strings match

String 1 ${REQUESTED_ACTION}

String 2 greeting

Case insensitive ✓

Builder Execute shell

Command
```
echo "Hello, bitwiseman!"
```

See the list of available environment variables

Here's the output when I run this project with `REQUESTED_ACTION` set to "greeting":

```
Run condition [Strings match] enabling prebuild for step [Execute shell]
Strings match run condition: string 1=[greeting], string 2=[greeting]
Run condition [Strings match] enabling perform for step [Execute shell]
[freestyle-conditional] $ /bin/sh -xe /var/folders/hp/f7yc_mwj2tq1hmbv_5n10v2c0000gn/T/hudson5963233933358491209.sh
+ echo 'Hello, bitwiseman!'
Hello, bitwiseman!
Finished: SUCCESS
```

And when I pass the value "silence":

```
Run condition [Strings match] enabling prebuild for step [Execute shell]
Strings match run condition: string 1=[silence], string 2=[greeting]
Run condition [Strings match] preventing perform for step [Execute shell]
Finished: SUCCESS
```

This is a simple example but the conditional step can contain any regular build step. When combined with other plugins, it can control whether to send notifications, gather data from other sources, wait for user feedback, or call other projects.

The Conditional BuildStep plugin does a great job of leveraging strengths of the Jenkins web UI, Freestyle jobs, and UI-based programming, but it is also hampered by their limitations. The Jenkins web UI can be clunky and confusing at times. Like the steps in any Freestyle job, these conditional steps are only stored and viewable in Jenkins. They are not versioned with other product or build code and can't be code reviewed. Like any number of UI-based programming tools, it has to make trade-offs between clarity and flexibility: more options or clearer presentation. There's only so much space on the screen.

Converting to Pipeline

Jenkins Pipeline, on the other hand, enables users to implement their pipeline as code. Pipeline code can be written directly in the Jenkins Web UI or in any text editor. It is a full-featured programming language, which gives users access to much broader set of conditional statements without the restrictions of UI-based programming.

So, taking the example above, the Pipeline equivalent is:

Jenkinsfile (Declarative Pipeline)

```
pipeline {
    agent any
    parameters {
        choice(
            // choices are a string of newline separated values
            // https://issues.jenkins-ci.org/browse/JENKINS-41180
            choices: 'greeting\nsilence',
            description: '',
            name: 'REQUESTED_ACTION')
    }

    stages {
        stage ('Speak') {
            when {
                // Only say hello if a "greeting" is requested
                expression { params.REQUESTED_ACTION == 'greeting' }
            }
            steps {
                echo "Hello, bitwiseman!"
            }
        }
    }
}
```

Toggle Scripted Pipeline *(Advanced)*

When I run this project with `REQUESTED_ACTION` set to "greeting", here's the output:

```
[Pipeline] node
Running on osx_mbp in /Users/bitwiseman/jenkins/agents/osx_mbp/workspace/pipeline-conditional
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Speak)
[Pipeline] echo
Hello, bitwiseman!
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

When I pass the value "silence", the only change is "Hello, bitwiseman!" is not printed.

Some might argue that the Pipeline code is a bit harder to understand on first reading. Others would say the UI is just as confusing if not more so. Either way, the Pipeline representation is considerably more compact than the Jenkins UI presentation. Pipeline also lets us add helpful comments, which we can't do in the Freestyle UI. And we can easily put this Pipeline in a `Jenkinsfile` to be code-reviewed, checked-in, and versioned along with the rest of our code.

Conditions

The previous example showed the "Strings match" condition and its Pipeline equivalent. Let's look at couple more interesting conditions and their Jenkins Pipeline equivalents.

Boolean condition

You might think that a boolean condition would be the simplest condition, but it isn't. Since it works with string values from tokens, the Conditional BuildStep plugin offers a number of ways to indicate true or false. Truth is a case insensitive match of one of the following: `1` (the number one), `Y`, `YES`, `T`, `TRUE`, `ON` or `RUN`.

Pipeline can duplicate these, but depending on the scenario we might consider whether a simpler expression would suffice.

Jenkinsfile (Declarative Pipeline)

```
when {
    // case insensitive regular expression for truthy values
    expression { return token ==~ /(?i)(Y|YES|T|TRUE|ON|RUN)/ }
}
steps {
    /* step */
}
```

Toggle Scripted Pipeline *(Advanced)*

Logical "OR" of conditions

This condition wraps other conditions. It takes their results as inputs and performs a logical "or" of the results. The `AND` and `NOT` conditions do the same, performing their respective operations.

Jenkinsfile (Declarative Pipeline)

```
when {
    // A or B
    expression { return A || B }
}
steps {
    /* step */
}
```

Toggle Scripted Pipeline *(Advanced)*

Tokens

Tokens can be considerably more work than conditions. There are more of them and they cover a much broader range of behaviors. The previous example showed one of the simpler cases, accessing a build parameter, where the token has a direct equivalent in Pipeline. However, many tokens don't have direct equivalents, some take a parameters (adding to their complexity), and some provide information that is simply not exposed in Pipeline yet. So, determining how to migrate tokens needs to be done on case-by-case basis.

Let's look at a few examples.

"FILE" token

Expands to the contents of a file. The file path is relative to the build workspace root.

${FILE,path="PATH"}

This token maps directly to the `readFile` step. The only difference is the file path for `readFile` is relative to the current working directory on the agent, but that is the workspace root by default. No problem.

Jenkinsfile (Declarative Pipeline)

```
when {
    expression { return readFile('pom.xml').contains('mycomponent') }
}
steps {
    /* step */
}
```

Toggle Scripted Pipeline *(Advanced)*

GIT_BRANCH

Expands to the name of the branch that was built.

Parameters (descriptions omitted): *all*, *fullName*.

This information may or may not be exposed in Pipeline. If you're using the Pipeline Multibranch plugin `env.BRANCH_NAME` will give similar basic information, but doesn't offer the parameters. There are also several issues filed around `GIT_*` tokens in Pipeline. Until they are addressed fully, we can follow the pattern shown in pipeline-examples, executing a shell to get the information we need.

Pipeline

```
GIT_BRANCH = sh(returnStdout: true, script: 'git rev-parse --abbrev-ref HEAD').trim()
```

CHANGES_SINCE_LAST_SUCCESS

Displays the changes since the last successful build.

Parameters (descriptions omitted): *reverse*, *format*, *changesFormat*, *showPaths*, *pathFormat*, *showDependencies*, *dateFormat*, *regex*, *replace*, *default*.

Not only is the information provided by this token not exposed in Pipeline, the token has ten optional parameters, including format strings and regular expression searches. There are a number of ways we might get similar information in Pipeline. Each have their own particular limitations and ways they differ from the token output. Then we'll need to consider how each of the parameters changes the output. If nothing else, translating this token is clearly beyond the scope of this post.

Slightly More Complex Example

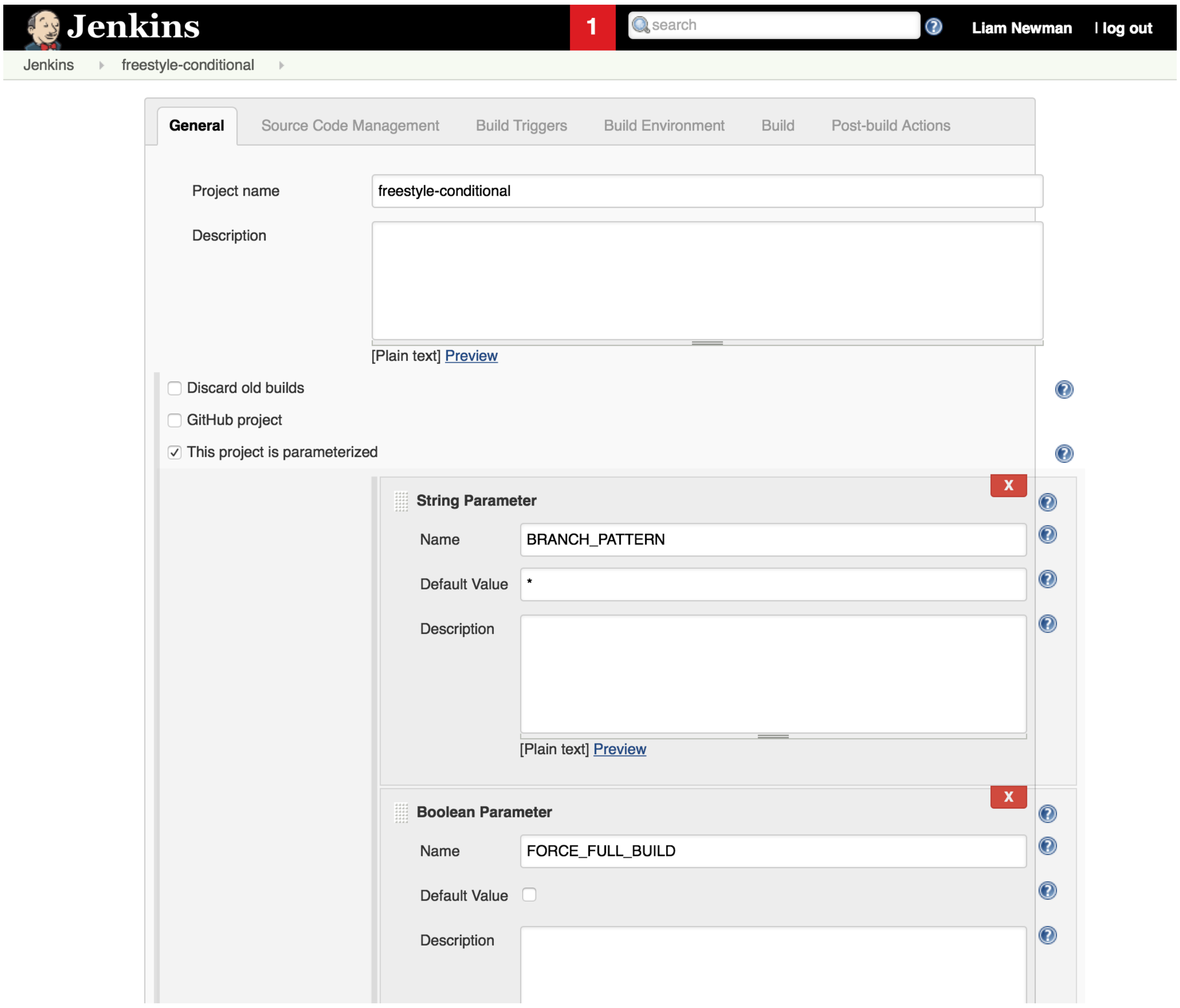
Let's do one more example that shows some of these conditions and tokens. This time we'll perform different build steps depending on what branch we're building. We'll take two build parameters: `BRANCH_PATTERN` and `FORCE_FULL_BUILD`. Based on `BRANCH_PATTERN`, we'll checkout a repository. If we're building on the `master` branch or the user checked `FORCE_FULL_BUILD`, we'll call three other builds in parallel (`full-build-linux`, `full-build-mac`, and `full-build-windows`), wait for them to finish, and report the result. If we're not building on the `master` branch and the user did not check `FORCE_FULL_BUILD`, we'll print a message saying we skipped the full builds.

Freestyle

Here's the configuration for Freestyle version. (It's pretty long. Feel free to skip down to the Pipeline version):

The Pipeline version of this job determines the `GIT_BRANCH` branch by running a shell script that returns the current local branch name. This means that the Pipeline version must checkout to a local branch (not a detached head).

Freestyle version of this job does not require a local branch, `GIT_BRANCH` is set automatically. However, to maintain functional parity, the Freestyle version of this job includes "Checkout to Specific Local Branch" as well.



Project name: freestyle-conditional

Description:

- [ ] Discard old builds
- [ ] GitHub project
- [x] This project is parameterized

String Parameter

Name: BRANCH_PATTERN

Default Value: *

Description:

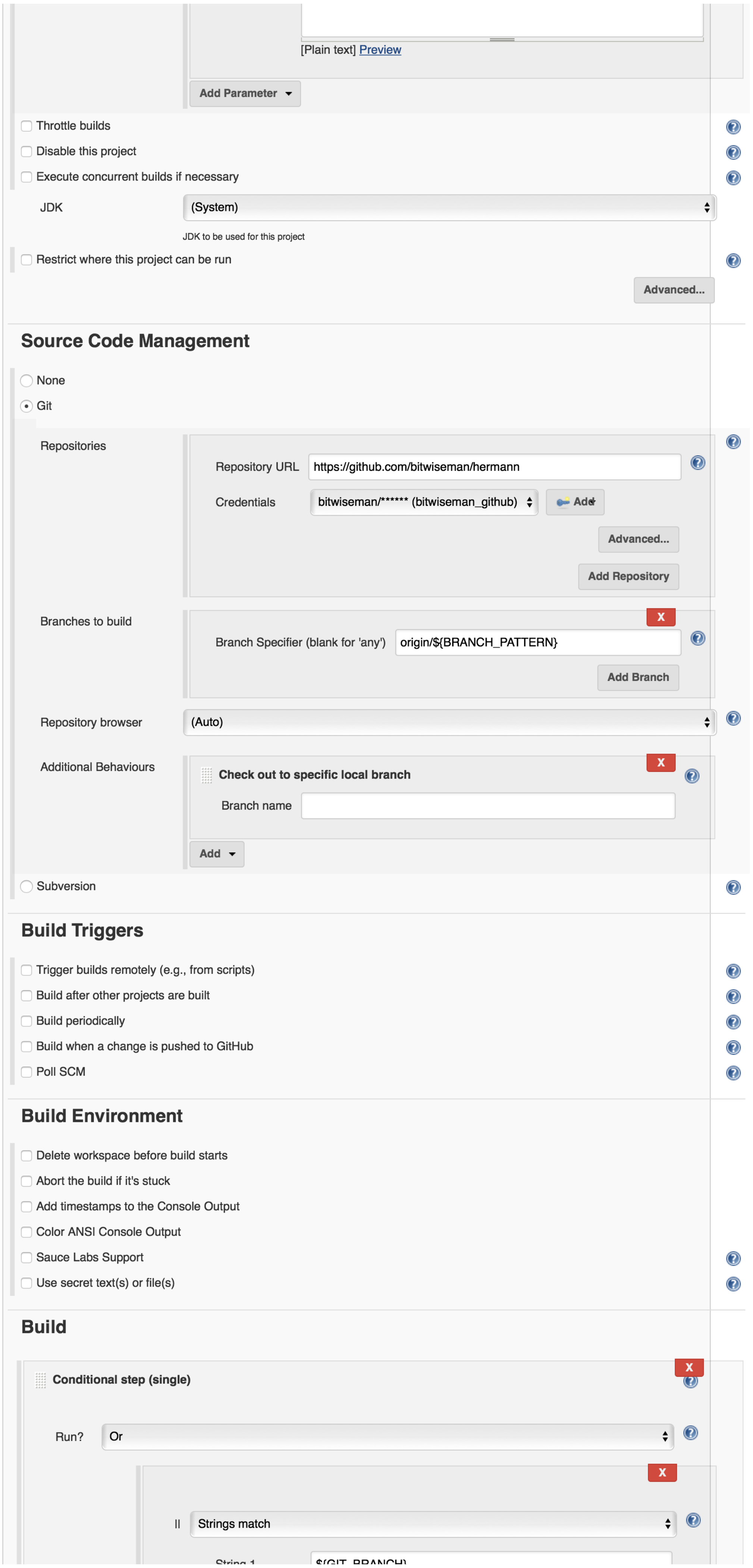
[Plain text] Preview

Boolean Parameter

Name: FORCE_FULL_BUILD

Default Value: [ ]

Description:



[Plain text] Preview

Add Parameter

- [ ] Throttle builds
- [ ] Disable this project
- [ ] Execute concurrent builds if necessary

JDK (System)

JDK to be used for this project

- [ ] Restrict where this project can be run

Advanced...

Source Code Management

- ○ None
- ● Git

Repositories

Repository URL https://github.com/bitwiseman/hermann

Credentials bitwiseman/****** (bitwiseman_github) Add

Advanced... Add Repository

Branches to build

Branch Specifier (blank for 'any') origin/${BRANCH_PATTERN}

Add Branch

Repository browser (Auto)

Additional Behaviours

Check out to specific local branch

Branch name

Add

- ○ Subversion

Build Triggers

- [ ] Trigger builds remotely (e.g., from scripts)
- [ ] Build after other projects are built
- [ ] Build periodically
- [ ] Build when a change is pushed to GitHub
- [ ] Poll SCM

Build Environment

- [ ] Delete workspace before build starts
- [ ] Abort the build if it's stuck
- [ ] Add timestamps to the Console Output
- [ ] Color ANSI Console Output
- [ ] Sauce Labs Support
- [ ] Use secret text(s) or file(s)

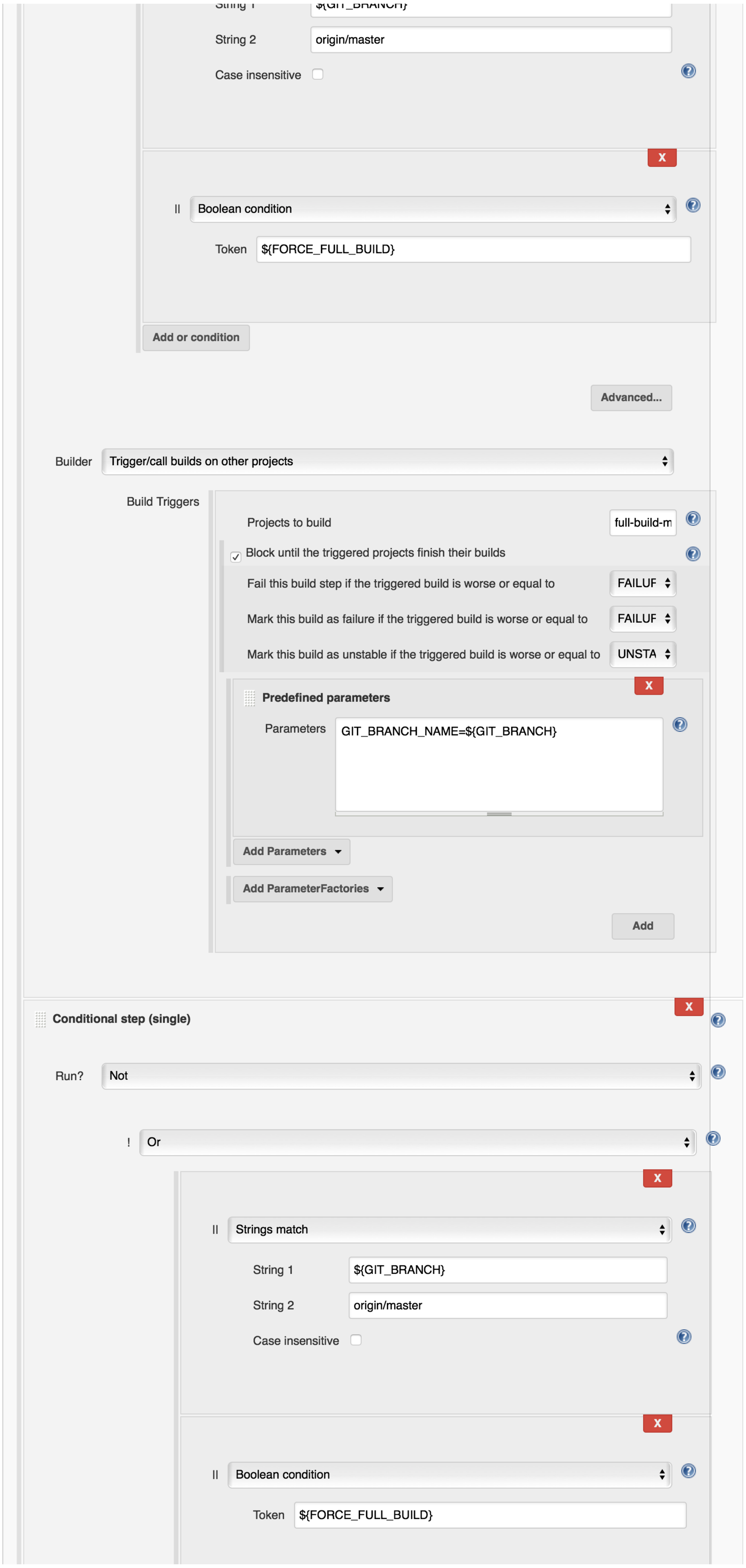
Build

Conditional step (single)

Run? Or

|| Strings match

String 1 ${GIT_BRANCH}


String 1 ${GIT_BRANCH}

String 2 origin/master

Case insensitive ☐

|| Boolean condition

Token ${FORCE_FULL_BUILD}

Builder Trigger/call builds on other projects

Build Triggers

Projects to build full-build-m

✓ Block until the triggered projects finish their builds

Fail this build step if the triggered build is worse or equal to FAILUR

Mark this build as failure if the triggered build is worse or equal to FAILUR

Mark this build as unstable if the triggered build is worse or equal to UNSTA

Predefined parameters

Parameters GIT_BRANCH_NAME=${GIT_BRANCH}

Conditional step (single)

Run? Not

! Or

|| Strings match

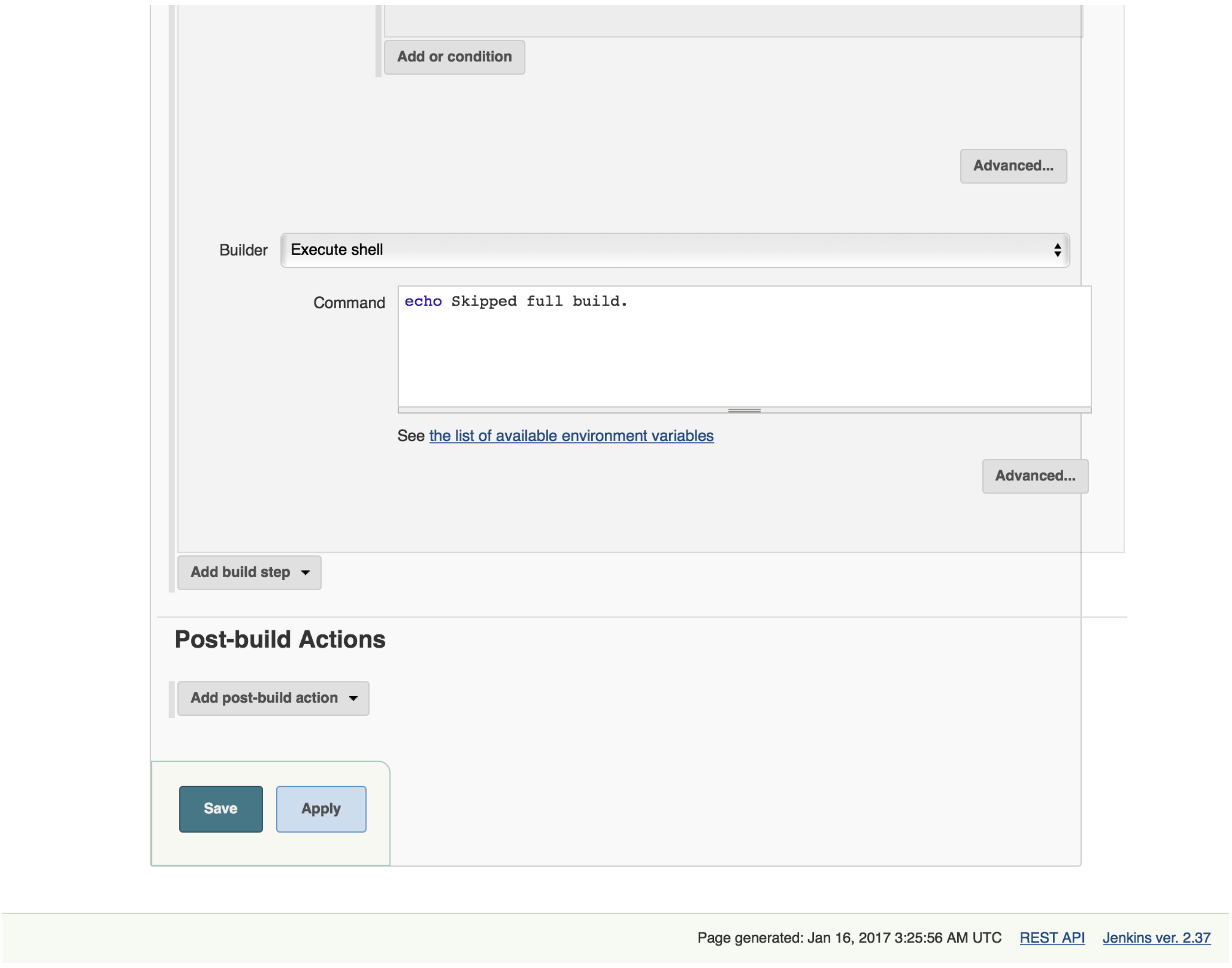
String 1 ${GIT_BRANCH}

String 2 origin/master

Case insensitive ☐

|| Boolean condition

Token ${FORCE_FULL_BUILD}



Builder: Execute shell

Command:
```
echo Skipped full build.
```

See the list of available environment variables

Page generated: Jan 16, 2017 3:25:56 AM UTC

Pipeline

Here's the equivalent Pipeline:

Freestyle version of this job is not stored in source control.

In general, the Pipeline version of this job would be stored in source control, would `checkout scm`, and would run that same repository. However, to maintain functional parity, the Pipeline version shown does a checkout from source control but is not stored in that repository.

Jenkinsfile (Declarative Pipeline)

```
pipeline {
    agent any
    parameters {
        string (
            defaultValue: '*',
            description: '',
            name : 'BRANCH_PATTERN')
        booleanParam (
            defaultValue: false,
            description: '',
            name : 'FORCE_FULL_BUILD')
    }

    stages {
        stage ('Prepare') {
            steps {
                checkout([$class: 'GitSCM',
                    branches: [[name: "origin/${BRANCH_PATTERN}"]],
                    doGenerateSubmoduleConfigurations: false,
                    extensions: [[$class: 'LocalBranch']],
                    submoduleCfg: [],
                    userRemoteConfigs: [[
                        credentialsId: 'bitwiseman_github',
                        url: 'https://github.com/bitwiseman/hermann']]])
            }
        }

        stage ('Build') {
            when {
                expression {
                    GIT_BRANCH = 'origin/' + sh(returnStdout: true, script: 'git rev-parse --abbrev-ref HEAD').trim()
                    return GIT_BRANCH == 'origin/master' || params.FORCE_FULL_BUILD
                }
            }
            steps {
                // Freestyle build trigger calls a list of jobs
                // Pipeline build() step only calls one job
                // To run all three jobs in parallel, we use "parallel" step
                // https://jenkins.io/doc/pipeline/examples/#jobs-in-parallel
                parallel (
                    linux: {
                        build job: 'full-build-linux', parameters: [string(name: 'GIT_BRANCH_NAME', value: GIT_BRANCH)
                    },
                    mac: {
                        build job: 'full-build-mac', parameters: [string(name: 'GIT_BRANCH_NAME', value: GIT_BRANCH)]
                    },
                    windows: {
                        build job: 'full-build-windows', parameters: [string(name: 'GIT_BRANCH_NAME', value: GIT_BRAN
                    },
                    failFast: false)
            }
        }
        stage ('Build Skipped') {
            when {
                expression {
                    GIT_BRANCH = 'origin/' + sh(returnStdout: true, script: 'git rev-parse --abbrev-ref HEAD').trim()
                    return !(GIT_BRANCH == 'origin/master' || params.FORCE_FULL_BUILD)
                }
            }
            steps {
                echo 'Skipped full build.'
            }
        }
    }
}
```

Toggle Scripted Pipeline *(Advanced)*

Conclusion

As I said before, the Conditional BuildStep plugin is great. It provides a clear, easy to understand way to add conditional logic to any Freestyle job. Before Pipeline, it was one of the few plugins to do this and it remains one of the most popular plugins. Now that we have Pipeline, we can implement conditional logic directly in code.

This is blog post discussed how to approach converting conditional build steps to Pipeline and showed a couple concrete examples. Overall, I'm pleased with the results so far. I found scenarios which could not easily be migrated to Pipeline, but even those are only more difficult, rather than impossible.

The next thing to do is add a section to the Jenkins Handbook documenting the Pipeline equivalent of all of the Conditions and the most commonly used Tokens. Look for it soon!

Links

- Conditional BuildStep plugin