Introducing a New Way to Define Jenkins Pipelines

Andrew Bayer
Introducing myself

• Hi!
• I’m Andrew Bayer
• Long-time Jenkins contributor
• Now a software engineer at CloudBees, working on Jenkins
• Particular focus on Pipeline
A little Jenkins job history

• Originally, there was Freestyle.
  • Configured through the web UI.
  • Choose your SCM, your build steps, your post-build actions...
  • Run on one node, check out one SCM...
A little Jenkins job history

• And then there was Pipeline
  • DSL scripting!
  • Pipeline-as-code - checked into your SCM
  • Full control over your SCM checkouts, what node you run on, conditionals, you name it
  • Durable, with running jobs surviving master restarts/disruption
But Pipeline isn’t perfect...yet.

• Coming from Freestyle (or other CI tools like Travis), Pipeline scripts can be very unfamiliar.
  • You don’t *really* need to know Groovy to write great Pipeline scripts, but it can feel that way.

• Behavior we’ve come to expect from Freestyle isn’t there automatically.
  • What, you mean I’ve got to do a try/catch to make sure I send build emails even if the build fails?

• Pipeline scripting without any additional structure is hard to represent in a visual editor.
  • Which is something else people miss!
And so, here’s what I'm doing about that...
Declarative Pipelines!
Declarative Pipelines!

• Pipelines can now be defined with a simpler syntax.
• Declarative “section” blocks for common configuration areas, like...
  • stages
  • tools
  • post-build actions
  • notifications
  • environment
  • build agent or Docker image
  • and more to come!
• All wrapped up in a `pipeline { ... }` step, with syntactic and semantic validation available.
Declarative Pipelines!

• This is not a separate thing from Pipeline. It’s part of Pipeline.
  • In fact, it’s actually even still Groovy. Sort of. =)
• Configured and run from a Jenkinsfile.
• Step syntax is valid within the **pipeline** block and outside it.

• But this does make some things easier:
  • **notifications** and **postBuild actions** are run at the end of your build even if the build has failed.
  • **agent** provides simpler control over where your build runs.
  • You’ll see more as we keep going!
What does this look like?

```groovy
pipeline {
  agent none
  stages {
    stage("foo") {
      echo "hello"
    }
  }
}
```
So what goes in the **pipeline** block?

- What we’re calling “sections”
  - Name of the section and the value for that section
- Current sections:
  - stages
  - agent
  - environment
  - tools
  - postBuild
  - notifications
Stages

- The stages section contains one or more stage blocks.
  - stage blocks look the same as the new block-scoped stage step.
  - Think of each stage block as like an individual Build Step in a Freestyle job.
- There must be a stages section present in your pipeline block.
- Example:

```groovy
stages {
    stage("build") {
        timeout(time: 5, units: 'MINUTES') {
            sh './run-some-script.sh'
        }
    }
    stage("deploy") {
        sh './deploy-something.sh'
    }
}
```
**Agent**

- **agent** determines where your build runs.

  - **Current possible settings:**
    - **agent label:''** - Run on any node
    - **agent docker:'ubuntu'** - Run on any node within a Docker container of the “ubuntu” image
    - **agent docker:'ubuntu’, label:'foo’** - Run on a node with the label “foo” within a Docker container of the “ubuntu” image
    - **agent none** - Don’t run on a node at all - manage node blocks yourself within your stages.

  - We are planning to make this extensible and composable going forward.

- **There must be an agent section in your pipeline block.**
The `tools` section allows you to define tools to autoinstall and add to the PATH.

- Note - this doesn’t work with `agent docker:'ubuntu'`.
- Note - this will be ignored if `agent none` is specified.

The `tools` section takes a block of tool name/tool version pairs, where the tool version is what you’ve configured on this master.

- Example:

  ```
  tools {
      maven "Maven 3.3.9"
      jdk "Oracle JDK 8u40"
  }
  ```
Environment

- **environment** is a block of key = value pairs that will be added to the environment the build runs in.

- Example:

  ```python
  environment {
      FOO = "bar"
      BAZ = "faz"
  }
  ```
Notifications and postBuild

• Much like Post Build Actions in Freestyle
• `postBuild` and `notifications` both contain blocks with one or more build condition keys and related step blocks.
• The steps for a particular build condition will be invoked if that build condition is met. More on this next page!
• `postBuild` checks its conditions and executes them, if satisfied, after all stages have completed, in the same node/Docker container as the stages.
• `notifications` checks its conditions and executes them, if satisfied, after `postBuild`, but doesn’t run on a node at all.
• **BuildCondition is an extension point.**

• Implementations provide:
  • A condition name
  • A method to check whether the condition has been satisfied with the current build status.

• Built-in conditions are listed on the right.

<table>
<thead>
<tr>
<th>Name</th>
<th>Satisfied When…</th>
</tr>
</thead>
<tbody>
<tr>
<td>success</td>
<td>The build is successful</td>
</tr>
<tr>
<td>failure</td>
<td>The build has failed</td>
</tr>
<tr>
<td>unstable</td>
<td>The build is unstable</td>
</tr>
<tr>
<td>changed</td>
<td>The build’s status is different than the previous build</td>
</tr>
<tr>
<td>always</td>
<td>Always true</td>
</tr>
</tbody>
</table>
Notifications and postBuild examples

```
notifications {
    success {
        hipchatSend "Build passed"
    }
    failure {
        hipchatSend "Build failed"
        mail to: "me@example.com",
        subject: "Build failed",
        body: "Fix me please!"
    }
}
```

```
postBuild {
    always {
        archive "target/**/**"
        junit 'path/to/**.xml'
    }
    failure {
        sh './cleanup-failure.sh'
    }
}
```
More sections are coming

• Some common use cases aren't covered by the sections we have right now.
• We know that!
• So more sections are in the works - we'll cover them later in the presentation!
pipeline {
    // Make sure that the tools we need are installed and on the path.
    tools {
        maven "Maven 3.3.9"
        jdk "Oracle JDK 8u40"
    }

    // Run on any executor.
    agent label:""

    stages {
        // While there's only one stage here, you can specify as many stages as you like!
        stage("build") {
            sh 'mvn clean install -Dmaven.test.failure.ignore=true'
        }
    }
}

#JenkinsWorld
A real-world example with tools, postBuild and notifications

```groovy
class MY Class {

    postBuild {
        always {
            archive "target/**/*";
            junit 'target/surefire-reports/*' xml;
        }
    }

    notifications {
        success { mail(to:"abayer@cloudbees.com", subject:"SUCCESS: ${currentBuild.fullDisplayName}",
            body: "Yay, we passed.");
        }
        failure { mail(to:"abayer@cloudbees.com", subject:"FAILURE: ${currentBuild.fullDisplayName}",
            body: "Boo, we failed.");
        }
        unstable { mail(to:"abayer@cloudbees.com", subject:"UNSTABLE: ${currentBuild.fullDisplayName}",
            body: "Huh, we're unstable.");
        }
    }
}
```
Parallel execution on multiple OSes

```groovy
pipeline {
    agent none
    stages {
        stage("distribute") {
            parallel {
                "windows" : {
                    node('windows') {
                        bat "print from windows"
                    }
                },
                "mac" : {
                    node('osx') {
                        sh "echo from mac"
                    }
                },
                "linux" : {
                    node('linux') {
                        sh "echo from linux"
                    }
                }
            }
        }
    }
}
```
Docker and environment

```groovy
pipeline {
    agent docker:'ubuntu'
    environment {
        KITTENS = "furry"
        BANANAS = "great"
    }
    stages {
        stage("testing 123") {
            sh 'echo "Kittens are \${KITTENS}"'
            echo "Bananas are \${env.BANANAS}"
        }
    }
}
```
Validation!
Error reporting in Pipeline - ow.

- A common complaint with Pipeline: reporting/handling errors in the script itself.
  - Long obscure stacktraces
  - Fix one typo, re-run build, get to the next typo, rinse, repeat
  - No way to verify your Jenkinsfile without running the build!
So let's make that better!

- Declarative Pipelines has an entirely new validation system!
- Validation of semantics, syntax, argument types, and more.
- Run at the very beginning of build execution - reports all issues from the entire definition at once, not just one at a time.
- Errors show up in “compilation” phase, with useful error messages pointing to where the problem is in the configuration.
- There's still a stacktrace, but you can ignore it. =)
What is the validation doing?

• Makes sure all required sections and/or fields are present.

• Checks for required step parameters.

• Verifies parameter types are correct.

• Errors out if a tool or tool version isn't installed.
pipeline {
    agent none
}

Console Output

Started by user Andrew Bayer
org.codehaus.groovy.control.MultipleCompilationErrorsException: startup failed:
WorkflowScript: 1: Missing required section 'stages' @ line 1, column 1.
    pipeline {
        ^

1 error
Example - duplicate fields

```groovy
pipeline {
    environment {
        FOO = "BAR"
        FOO = "BAZ"
    }
    agent label: "some-label"
    stages {
        stage("foo") {
            sh 'echo "FOO is $FOO"'
        }
    }
}
```

**Console Output**

Started by user Andrew Bayer
org.codehaus.groovy.control.MultipleCompilationErrorsException: startup failed:
WorkflowScript: 4: Duplicate environment variable name: 'FOO' @ line 4, column 9.
    FOO = "BAZ"

1 error
Example - unavailable or unknown tool

Console Output

Started by user Andrew Bayer
org.codehaus.groovy.control.MultipleCompilationErrorsException: startup failed:
WorkflowScript: 4: Invalid tool type 'gradle'. Valid tool types: [ant, hudson.tasks.Ant$AntInstallation,
org.jenkinsci.plugins.docker.commons.tools.DockerTool, git, hudson.plugins.git.GitTool, jdk, hudson.model.JDK, jgit,
  gradle "gradle-2.14.1"

1 error

pipeline {
    agent label:"some-label"
    tools {
        gradle "gradle-2.14.1"
    }
    stages {
        stage("foo") {
            sh "gradle --version"
        }
    }
}
Example - unavailable or unknown tool
Linter coming soon!

- API endpoint on Jenkins master already present for validating your Jenkinsfile using Declarative Pipelines.
- We'll be adding a command-line tool available that just needs to be pointed to your Jenkins master and your Jenkinsfile to validate and report any errors!
- Aiming to have this available in the next couple weeks.
The future!
More sections, more functionality

- Currently you can't do things like add a timeout for the entire build, or use build parameters when using the declarative syntax.
- That will be changing!
- New sections already planned for:
  - "Wrappers" around the entire build
  - Build parameters
  - Build triggers
  - Other job properties
  - Shared library loading
Extensibility!

• Build conditions are already extensible.
• Sections will soon be extensible so other plugins can contribute their own!
• Agent backends (like the current "run in this Docker image" or "run on this label") will soon be extensible, and we'll be adding more bundled backends.
  • First on the list: "Build the Dockerfile in this repo root and run in the image that gets built"!
Stage Dependency Graph

• For a given stage, specify what other stages need to be run before or after.

• At runtime, these dependencies will be inspected and a dependency graph will be constructed.

• Result: stages will be executed in dependency order, run in parallel when possible.

• Coming soon!
Eclipse/IntelliJ integration?

• A "schema" for the declarative syntax is available via the Jenkins REST API, as is validation.

• So...we could write plugins for IDEs that:
  • Do autocomplete for sections and their fields based on the declarative syntax schema.
  • Validate on demand (or maybe even on the fly!).

• I've never written an Eclipse or IntelliJ plugin, so no promises, but I'll try!
What's still missing?

• One big chunk of Freestyle functionality we still don't have in Pipeline...

• Visual editor!
Visual editor plans

- Will be part of Blue Ocean
- Takes advantage of structured form of declarative Pipelines
- Reads the Jenkinsfile from source control
- Saves the Pipeline back to that same Jenkinsfile in source control
- Very, very early work right now, should ramp up in the next couple months
Editor - getting started
Editor - adding steps
Editor - adding stages
Want to try Declarative Pipelines?
Install Blue Ocean!
Thanks!
Questions?