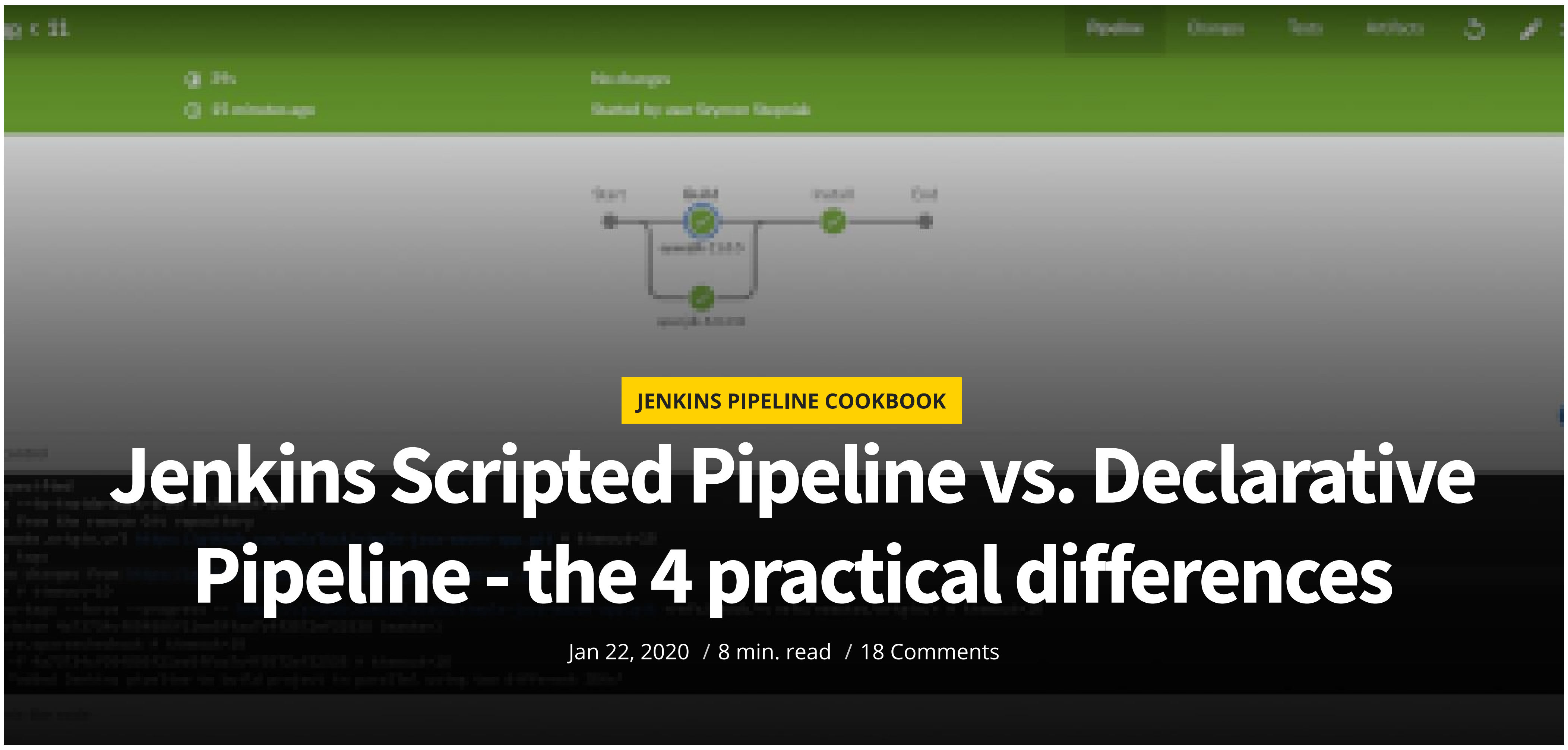


JENKINS PIPELINE COOKBOOK

Jenkins Scripted Pipeline vs. Declarative Pipeline - the 4 practical differences

Jan 22, 2020 / 8 min. read / 18 Comments

If you read this blog post, there is a high chance you're looking for information about practical differences between scripted and declarative pipeline, correct? You couldn't find a better place then. I'm going to show you the four most practical differences between those two. Stay with me for a few minutes and enjoy the ride!

Table of Contents

The introduction

But let's start with the following question - why are there two pipeline types in the first place? The scripted pipeline was the first implementation of the pipeline as a code in Jenkins. Even though it uses the underlying pipeline subsystem, it was designed more or less as a general-purpose DSL built with Groovy.[1] It means that it does not come with a fixed structure, and it is up to you how you will define your pipeline logic. The declarative pipeline, on the other hand, is more opinionated, and its structure is well-defined.[2] It may look a bit limiting, but in practice, you can achieve the same things using the scripted or declarative pipeline. So which one to choose?

If you ask me this question and expect an answer different from "*it depends*," I would say use the declarative pipeline. And here's why.

1. Pipeline code validation at startup

Let's consider the following pipeline code.

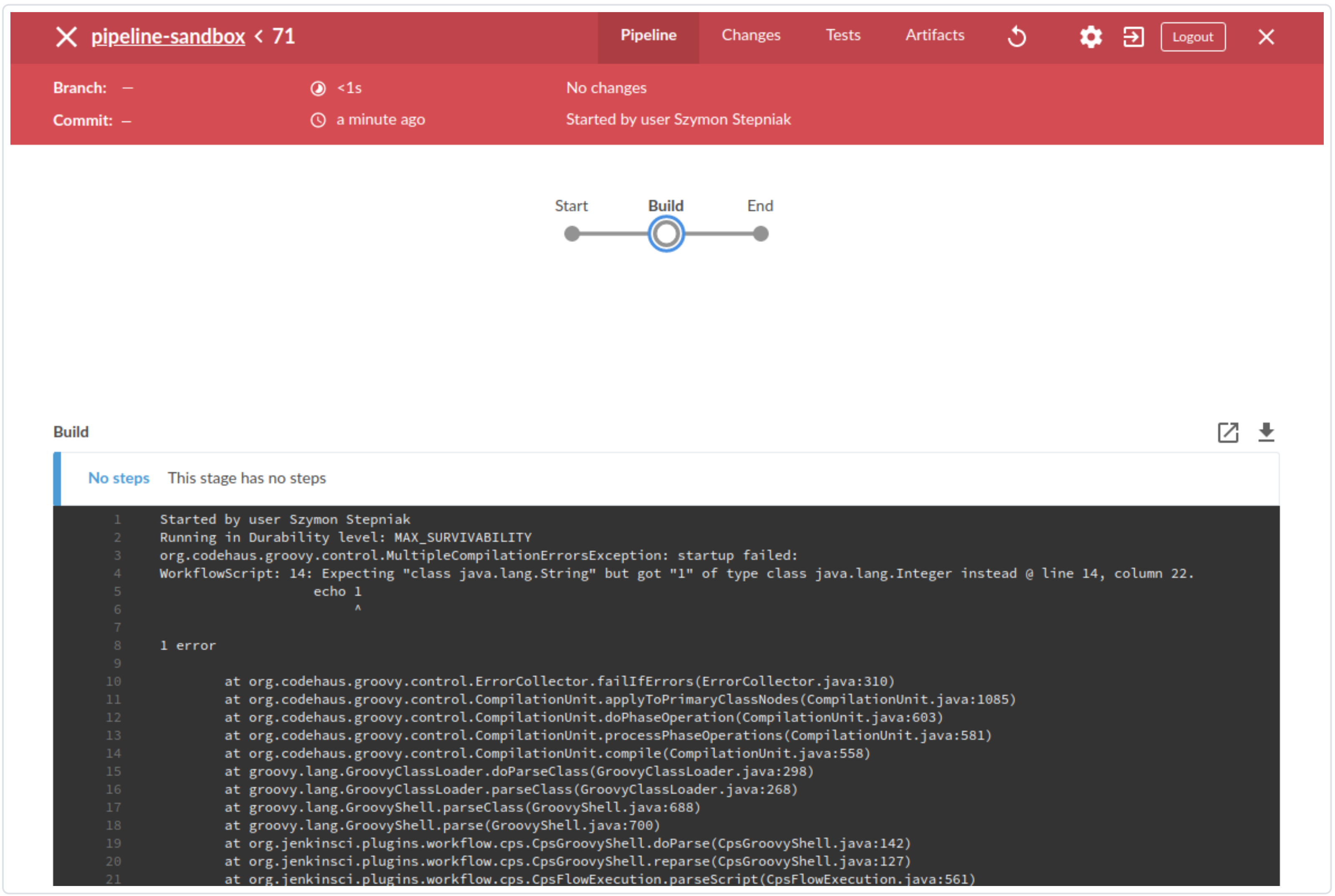
Listing 1. Jenkinsfile

```
pipeline {
    agent any

    stages {
        stage("Build") {
            steps {
                echo "Some code compilation here..."
            }
        }

        stage("Test") {
            steps {
                echo "Some tests execution here..."
                echo 1
            }
        }
    }
}
```

If we try to execute the following pipeline, the validation will quickly fail the build. The echo step can be triggered only with the `String` parameter, so we get the error like.



```
Started by user Szymon Stepniak
Running in Durability level: MAX_SURVIVABILITY
org.codehaus.groovy.control.MultipleCompilationErrorsException: startup failed:
WorkflowScript: 14: Expecting "class java.lang.String" but got "1" of type class java.lang.Integer instead @ line 14, column 22.
                   echo 1
                        ^


1 error

        at org.codehaus.groovy.control.ErrorCollector.failIfErrors(ErrorCollector.java:310)
        at org.codehaus.groovy.control.CompilationUnit.applyToPrimaryClassNodes(CompilationUnit.java:1085)
        at org.codehaus.groovy.control.CompilationUnit.doPhaseOperation(CompilationUnit.java:603)
        at org.codehaus.groovy.control.CompilationUnit.processPhaseOperations(CompilationUnit.java:581)
        at org.codehaus.groovy.control.CompilationUnit.compile(CompilationUnit.java:558)
        at groovy.lang.GroovyClassLoader.doParseClass(GroovyClassLoader.java:298)
        at groovy.lang.GroovyClassLoader.parseClass(GroovyClassLoader.java:268)
        at groovy.lang.GroovyShell.parseClass(GroovyShell.java:688)
        at groovy.lang.GroovyShell.parse(GroovyShell.java:700)
        at org.jenkinsci.plugins.workflow.cps.CpsGroovyShell.doParse(CpsGroovyShell.java:142)
        at org.jenkinsci.plugins.workflow.cps.CpsGroovyShell.reparse(CpsGroovyShell.java:127)
        at org.jenkinsci.plugins.workflow.cps.CpsFlowExecution.parseScript(CpsFlowExecution.java:561)
```

Notice that the pipeline didn't execute any stage and just failed. This might save us a lot of time - imagine executing the Build stage for a couple of minutes, just to get the information that the echo step expects to get `java.lang.String` instead of `java.lang.Integer`.

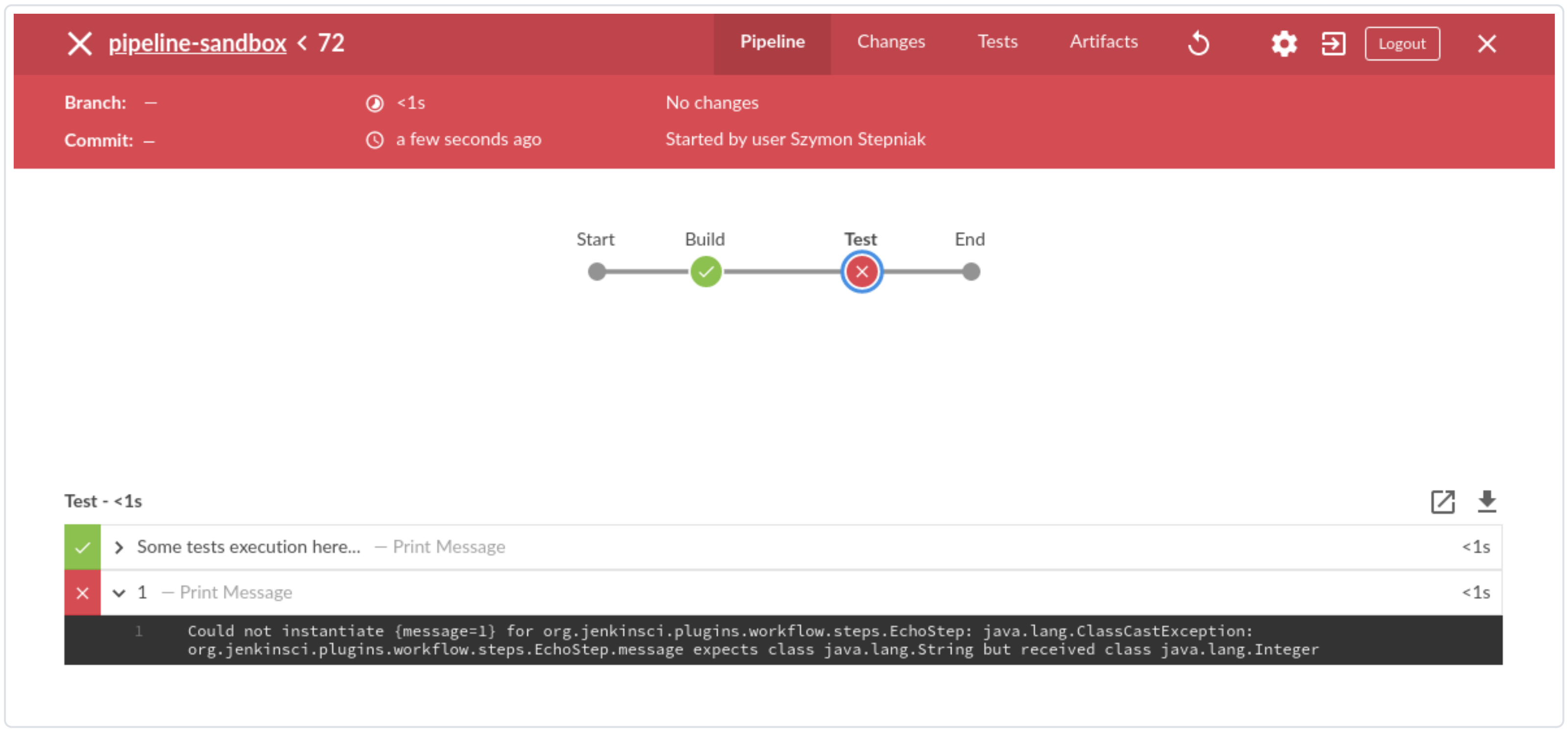
Now let's take a look at the scripted pipeline equivalent of that example.

Listing 2. Jenkinsfile

```
node {
    stage("Build") {
        echo "Some code compilation here..."
    }

    stage("Test") {
        echo "Some tests execution here..."
        echo 1
    }
}
```

This pipeline executes the same stages, and the same steps. There is one significant difference however. Let’s execute it and see what result it produces.



Test - <1s

Some tests execution here... — Print Message <1s

1 — Print Message <1s

```
Could not instantiate {message=1} for org.jenkinsci.plugins.workflow.steps.EchoStep: java.lang.ClassCastException:
org.jenkinsci.plugins.workflow.steps.EchoStep.message expects class java.lang.String but received class java.lang.Integer
```

It failed as expected. But this time the Build stage was executed, as well as the first step from the Test stage. As you can see, there was no validation of the pipeline code. The declarative pipeline in this case handles such use case much better.

2. Restart from stage

Another cool feature that only declarative pipeline has is "Restart from stage" one. Let’s fix the pipeline from the previous example and see if we can restart the Test stage only.

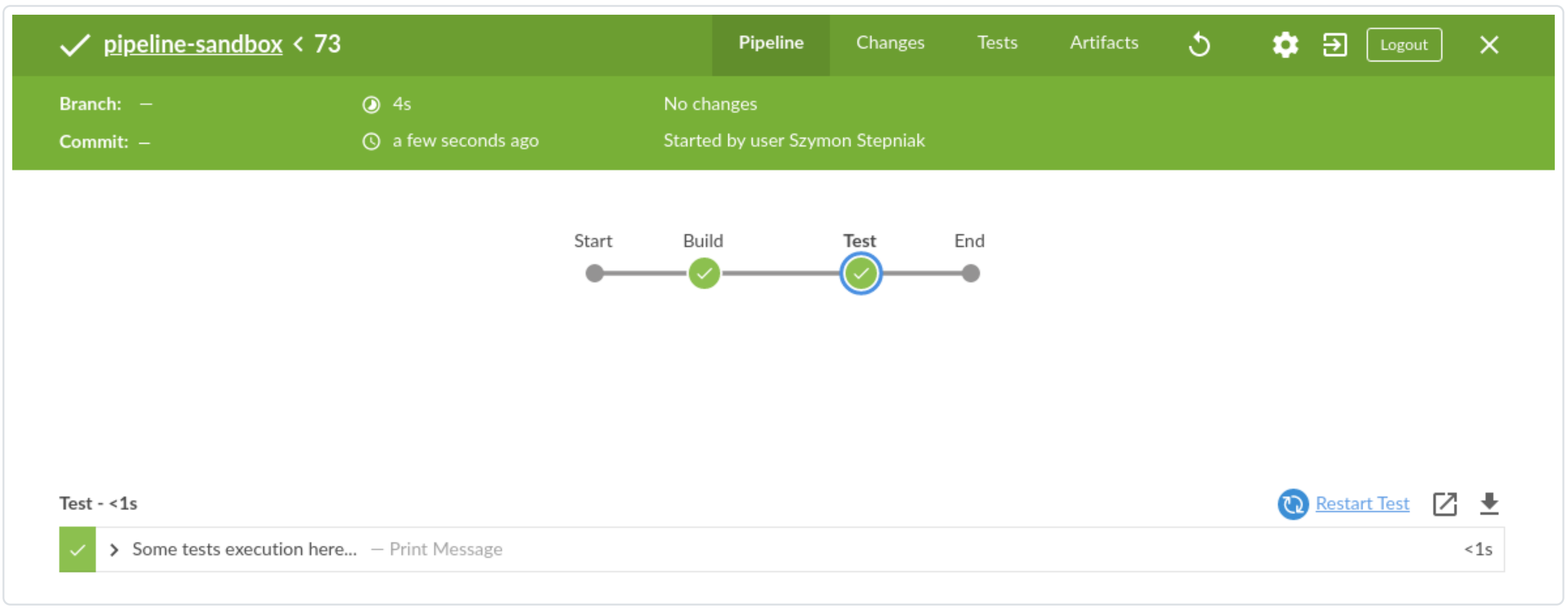
● ● ● Listing 3. Jenkinsfile

```
pipeline {
    agent any

    stages {
        stage("Build") {
            steps {
                echo "Some code compilation here..."
            }
        }

        stage("Test") {
            steps {
                echo "Some tests execution here..."
            }
        }
    }
}
```

Let’s execute it.

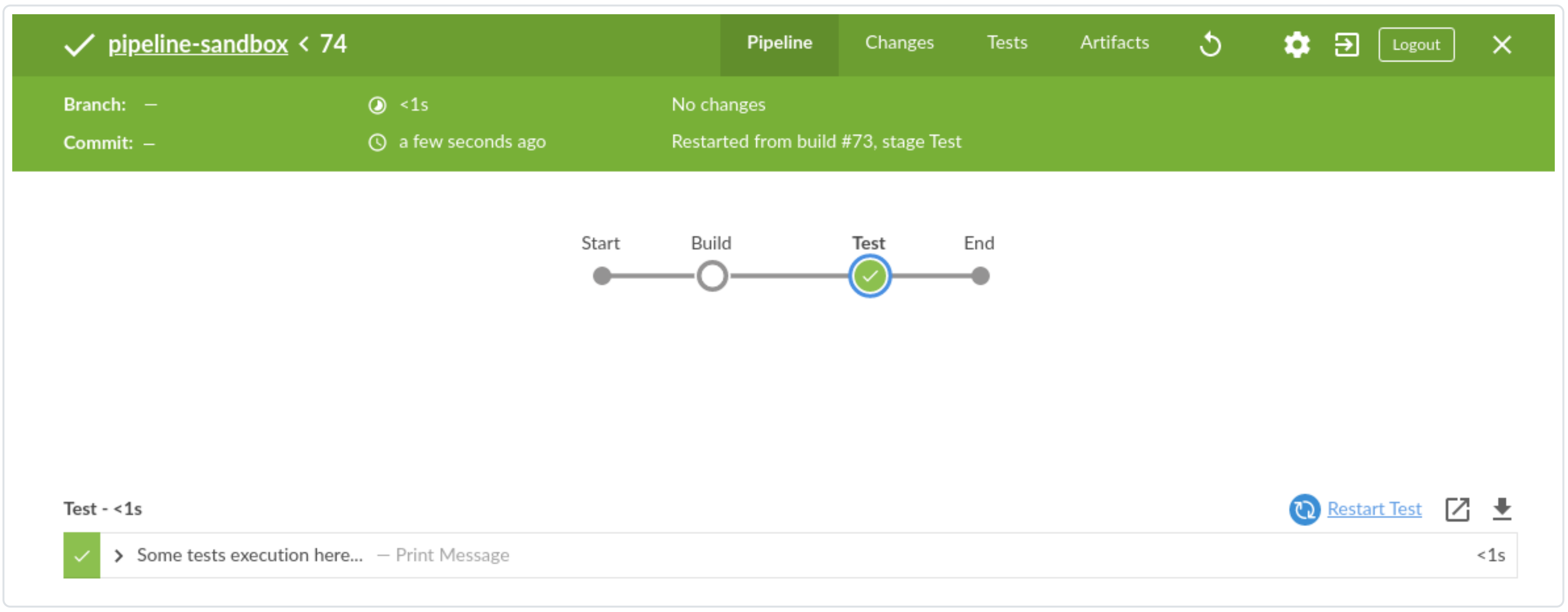

pipeline-sandbox < 73

Branch: –
Commit: –
4s
a few seconds ago
No changes
Started by user Szymon Stepniak

Test - <1s

Some tests execution here... — Print Message <1s

Here you can see that the Test stage is selected. There is an option called "Restart Test" right above the steps list on the right side. Let’s click on it and see the result.


pipeline-sandbox < 74

Branch: –
Commit: –
<1s
a few seconds ago
No changes
Restarted from build #73, stage Test

Test - <1s

Some tests execution here... — Print Message <1s

As you can see, Jenkins skipped the Build stage (it used the workspace from the previous build) and started the next pipeline execution from the Test stage. It might be useful when you execute some external tests and they fail because of some issues with the remote environment. You can fix the problem with the test environment and re-run the stage again, without the need to rebuild all artifacts. (The code of the app hasn’t change in such case.)

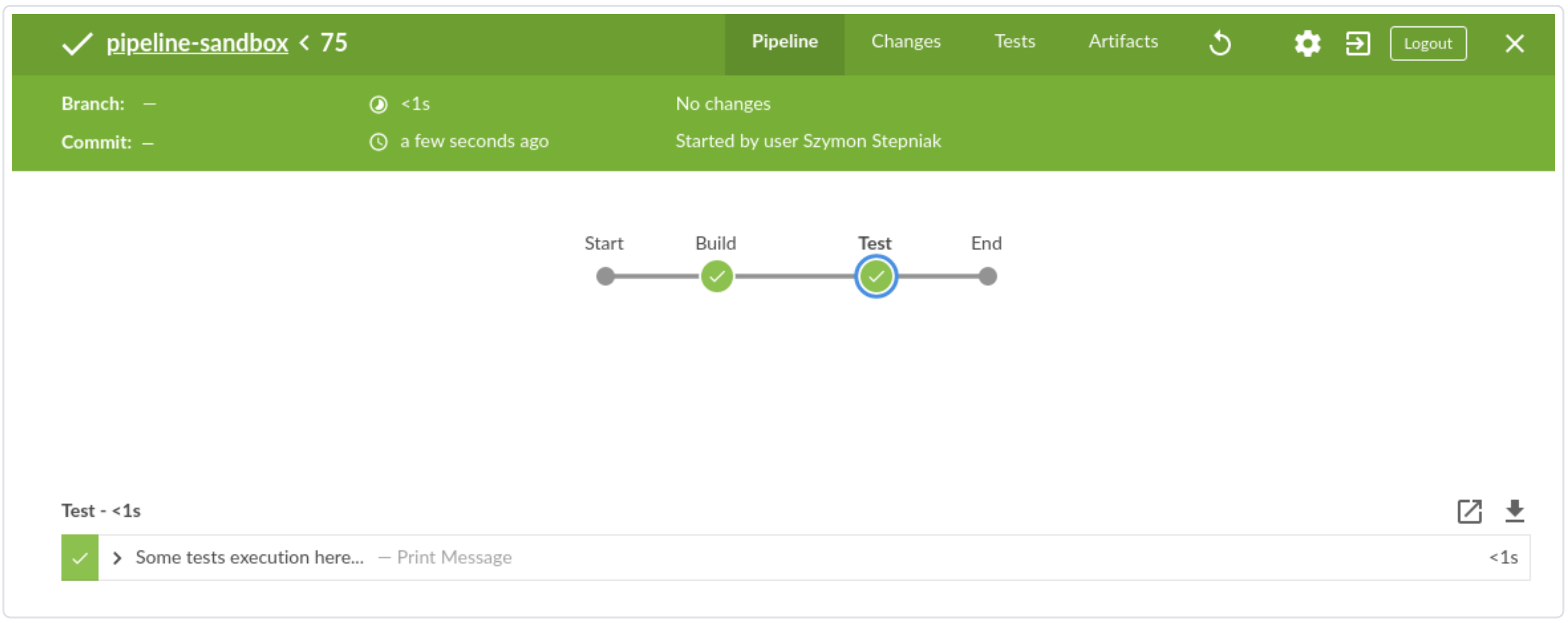
Now, let's look at the scripted pipeline example.

Listing 4. Jenkinsfile

```
node {
    stage("Build") {
        echo "Some code compilation here..."
    }

    stage("Test") {
        echo "Some tests execution here..."
    }
}
```

Let's execute it.



pipeline-sandbox < 75

Pipeline Changes Tests Artifacts Logout

Branch: — <1s No changes

Commit: — a few seconds ago Started by user Szymon Stepniak

Start Build Test End

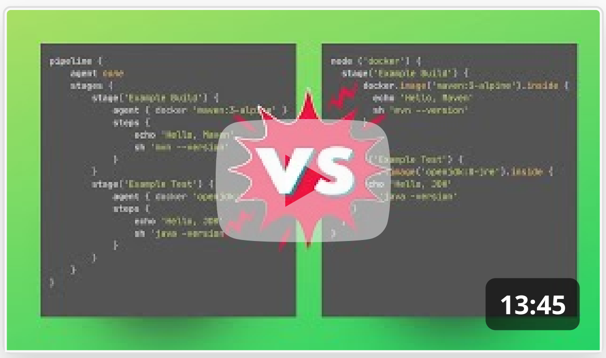
Test - <1s

Some tests execution here... — Print Message <1s

No restart option as you can see. The declarative pipeline vs. scripted pipeline - 2:0.

Jenkins Declarative Pipeline vs Scripted Pipeline - 4 key differences | #jenkinspipeline

Jenkins Pipeline as a code is a new standard for defining continuous integration and delivery pipelines in Jenkins. The scripted pipeline was the first implementation of the pipeline as a code standard. The later one, the declarative pipeline, slowly becomes a standard of how Jenkins users define their pipeline logic. In this tutorial video, I explain what are four major differences between Jenkins declarative pipeline and the scripted one. Enjoy and don't forget to like and comment on the video!



Watch now

3. Declarative pipeline `options` block

The third feature is supported by both pipeline types, however the declarative pipeline handles it a bit better in my opinion. Let's say we have the following features to add to the previous pipeline.

- The timestamps in console log.
- The ANSI color output.

- The 1-minute timeout for the Build stage, and 2 minutes timeout for the Test stage.

Here is what does the declarative pipeline look like.

Listing 5. Jenkinsfile

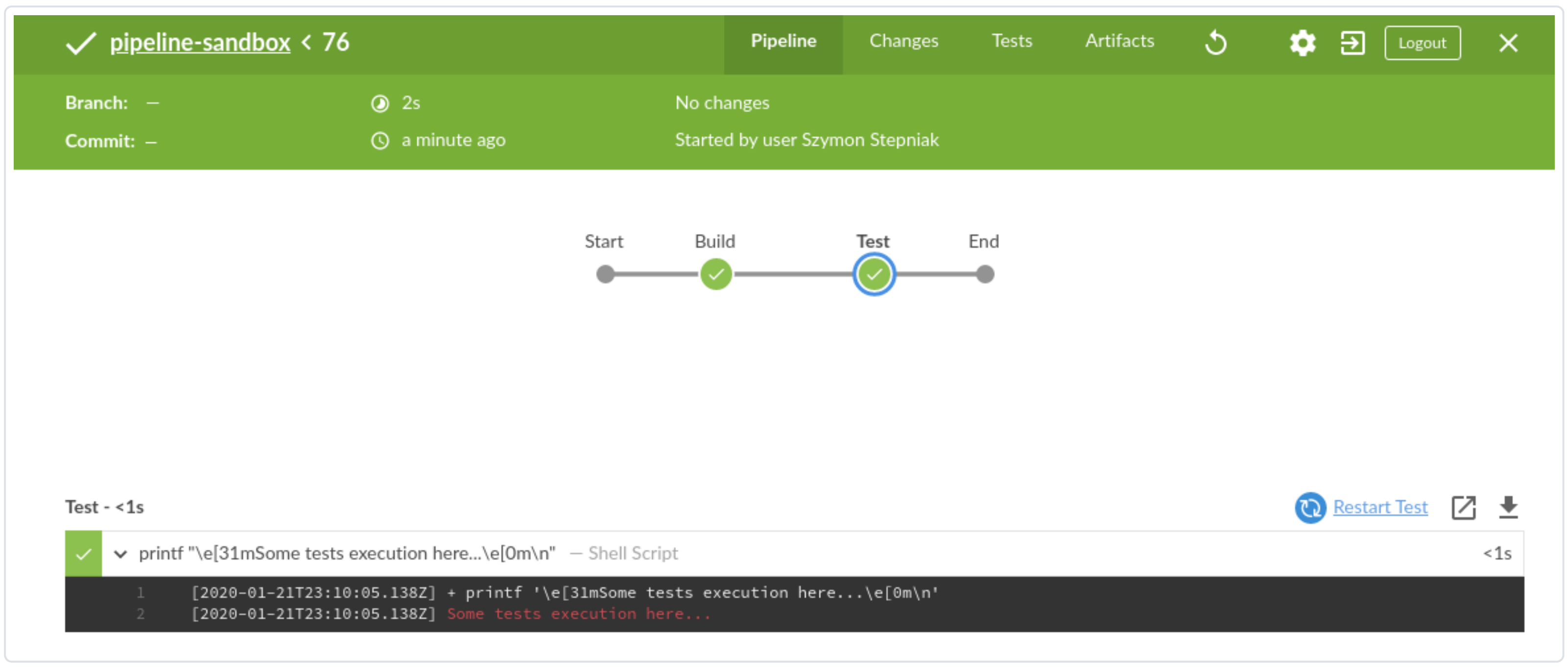
```
pipeline {
    agent any

    options {
        timestamps()
        ansiColor("xterm")
    }

    stages {
        stage("Build") {
            options {
                timeout(time: 1, unit: "MINUTES")
            }
            steps {
                sh 'printf "\\e[31mSome code compilation here...\\e[0m\\n"'
            }
        }

        stage("Test") {
            options {
                timeout(time: 2, unit: "MINUTES")
            }
            steps {
                sh 'printf "\\e[31mSome tests execution here...\\e[0m\\n"'
            }
        }
    }
}
```

Let's run it.



printf "\e[31mSome tests execution here...\e[0m\n" — Shell Script

```
[2020-01-21T23:10:05.138Z] + printf '\e[31mSome tests execution here...\e[0m\n'
[2020-01-21T23:10:05.138Z] Some tests execution here...
```

Here is the console log.

```
Started by user Szymon Stepniak
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] Start of Pipeline
[Pipeline] node
Running on Jenkins in /home/wololock/.jenkins/workspace/pipeline-sandbox
[Pipeline] {
[Pipeline] timestamps
[Pipeline] {
[Pipeline] ansiColor
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Build)
[Pipeline] timeout
15:10:04  Timeout set to expire in 1 min 0 sec
[Pipeline] {
[Pipeline] sh
15:10:04  + printf '\e[31mSome code compilation here...\e[0m\n'
15:10:04  Some code compilation here...
[Pipeline] }
[Pipeline] // timeout
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Test)
[Pipeline] timeout
15:10:04  Timeout set to expire in 2 min 0 sec
[Pipeline] {
[Pipeline] sh
15:10:05  + printf '\e[31mSome tests execution here...\e[0m\n'
15:10:05  Some tests execution here...
[Pipeline] }
[Pipeline] // timeout
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // ansiColor
[Pipeline] }
[Pipeline] // timestamps
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

In the declarative pipeline, options are separated from the pipeline script logic. The scripted pipeline also supports `timestamps`, `ansiColor` and `timeout` options, but it requires a different code. Here is the same pipeline expressed using the scripted pipeline.

Listing 6. Jenkinsfile

```
node {
    timestamps {
        ansiColor("xterm") {
            stage("Build") {
                timeout(time: 1, unit: "MINUTES") {
                    sh 'printf "\\e[31mSome code compilation here...\\e[0m\\n"'
                }
            }
            stage("Test") {
                timeout(time: 2, unit: "MINUTES") {
                    sh 'printf "\\e[31mSome tests execution here...\\e[0m\\n"'
                }
            }
        }
    }
}
```

I guess you see the problem. Here we used only `timestamps` and `ansiColor` Jenkins plugins. Imagine adding one or two more plugins. Declarative vs. scripted, 3:0.

4. Skipping stages with `when` block.

The last thing I would like to mention in this blog post is the `when` block that the declarative pipeline supports. Let's improve the previous example and add a following condition:

- Execute Test stage only if `env.FOO` equals `bar`.

Here is what the declarative pipeline code looks like.

Listing 7. Jenkinsfile

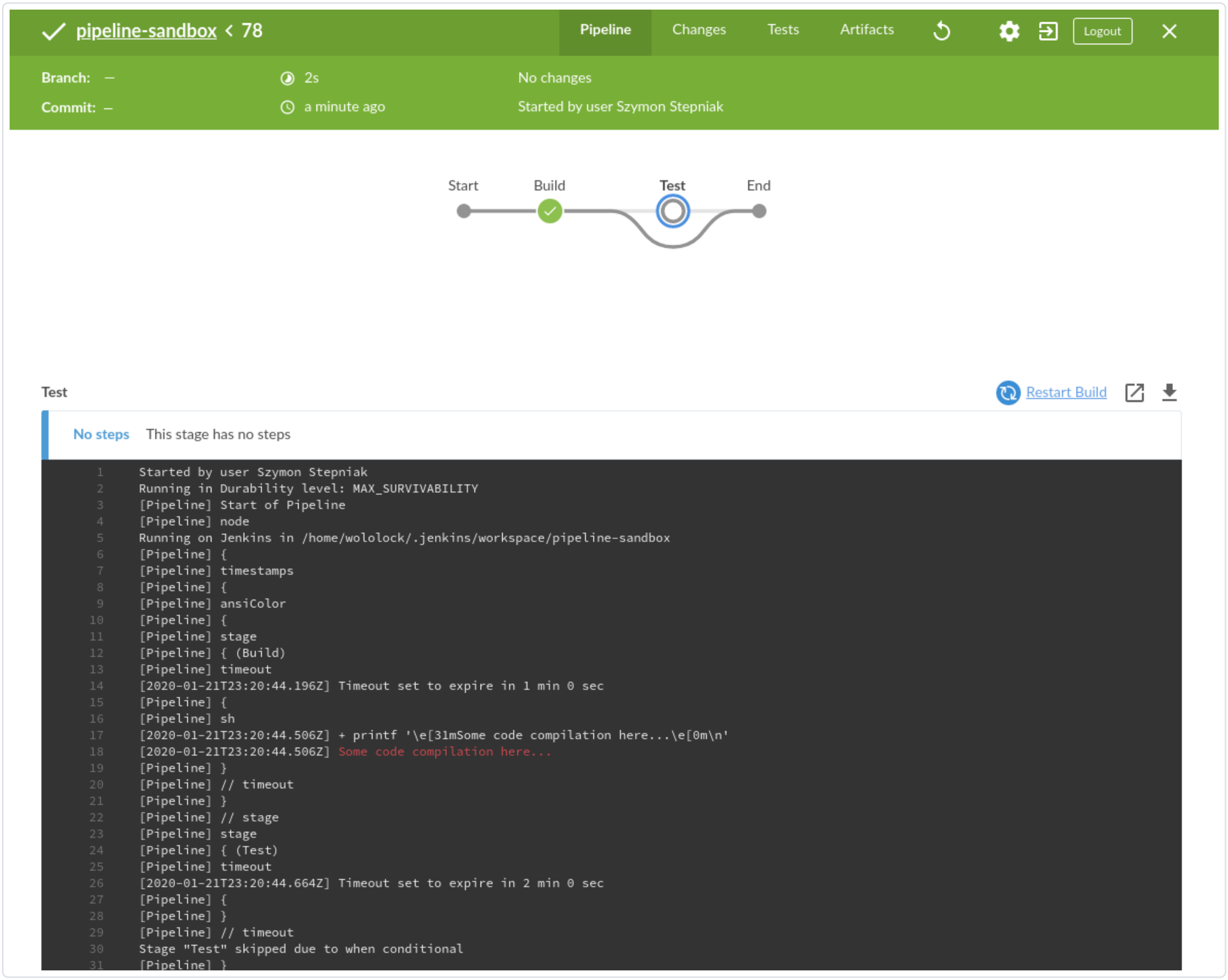
```
pipeline {
    agent any

    options {
        timestamps()
        ansiColor("xterm")
    }

    stages {
        stage("Build") {
            options {
                timeout(time: 1, unit: "MINUTES")
            }
            steps {
                sh 'printf "\\e[31mSome code compilation here...\\e[0m\\n"'
            }
        }

        stage("Test") {
            when {
                environment name: "FOO", value: "bar"
            }
            options {
                timeout(time: 2, unit: "MINUTES")
            }
            steps {
                sh 'printf "\\e[31mSome tests execution here...\\e[0m\\n"'
            }
        }
    }
}
```

And let's execute it.



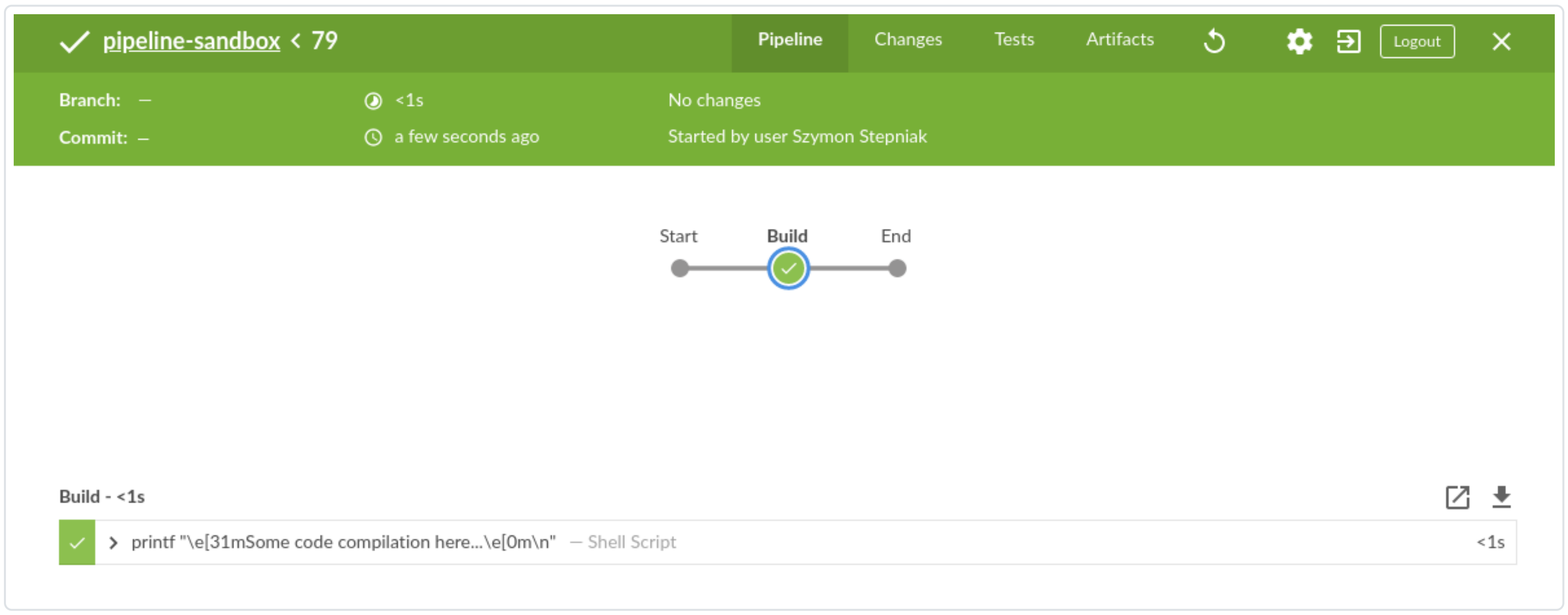
```
Started by user Szymon Stepniak
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] Start of Pipeline
[Pipeline] node
Running on Jenkins in /home/wololock/.jenkins/workspace/pipeline-sandbox
[Pipeline] {
[Pipeline] timestamps
[Pipeline] {
[Pipeline] ansiColor
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Build)
[Pipeline] timeout
[2020-01-21T23:20:44.196Z] Timeout set to expire in 1 min 0 sec
[Pipeline] {
[Pipeline] sh
[2020-01-21T23:20:44.506Z] + printf '\e[31mSome code compilation here...\e[0m\n'
[2020-01-21T23:20:44.506Z] Some code compilation here...
[Pipeline] }
[Pipeline] // timeout
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Test)
[Pipeline] timeout
[2020-01-21T23:20:44.664Z] Timeout set to expire in 2 min 0 sec
[Pipeline] {
[Pipeline] }
[Pipeline] // timeout
Stage "Test" skipped due to when conditional
[Pipeline] }
```

The Test stage was skipped as expected. Now let's try to do the same thing in the scripted pipeline example.

Listing 8. Jenkinsfile

```
node {
    timestamps {
        ansiColor("xterm") {
            stage("Build") {
                timeout(time: 1, unit: "MINUTES") {
                    sh 'printf "\\e[31mSome code compilation here...\\e[0m\\n"'
                }
            }
            if (env.FOO == "bar") {
                stage("Test") {
                    timeout(time: 2, unit: "MINUTES") {
                        sh 'printf "\\e[31mSome tests execution here...\\e[0m\\n"'
                    }
                }
            }
        }
    }
}
```

As you can see, we had to use if-condition to check if `env.FOO` equals `bar`, and only then add the Test stage. (It's not a real skipping in this case unfortunately.) Let's run it and see what is the result.



```
printf "\e[31mSome code compilation here...\e[0m\n" — Shell Script   <1s
```

This is not the same result. In the scripted pipeline use case, the Test stage is not even rendered. This might introduce some unnecessary confusion, the declarative pipeline handles it much better in my opinion. Declarative vs. scripted, 4:0.

Conclusion

Here are my top 4 differences between the declarative and scripted Jenkins pipeline. These are not the only differences, and I guess your list may look a little different. What is your choice? Do you prefer the declarative pipeline, or the scripted one? Please share your thoughts in the section down below.

Share this blog post on or LinkedIn to help me spread the word. Thanks! ☺

1. https://jenkins.io/doc/book/pipeline/syntax/#scripted-pipeline
2. https://jenkins.io/doc/book/pipeline/syntax/#declarative-pipeline

Published: Jan 22, 2020 in Jenkins Pipeline Cookbook

cicd, groovy, jenkins, jenkins-pipeline, pipeline

SZYMON STEPNIAK

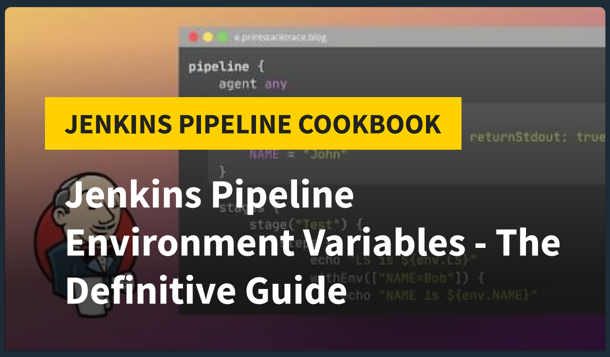
Groovista, Upwork's Top Rated freelancer, Toruń Java User Group founder, open source contributor, Stack Overflow addict, bedroom guitar player. I walk through e.printStackTrace() so you don't have to. More about me »

Related posts

JENKINS PIPELINE COOKBOOK
Building Java and Maven docker images using parallelized Jenkins Pipeline and SDKMAN!

Nov 11, 2019 / 6 min. read /
0 Comments

In the last article, I have shown you how you can build a docker image for Jenkins Pipeline using SDKMAN! command-line tool. Today I will show you how you can build multiple different docker images using parallelized Jenkins Pipeline.


JENKINS PIPELINE COOKBOOK
Jenkins Pipeline Environment Variables - The Definitive Guide

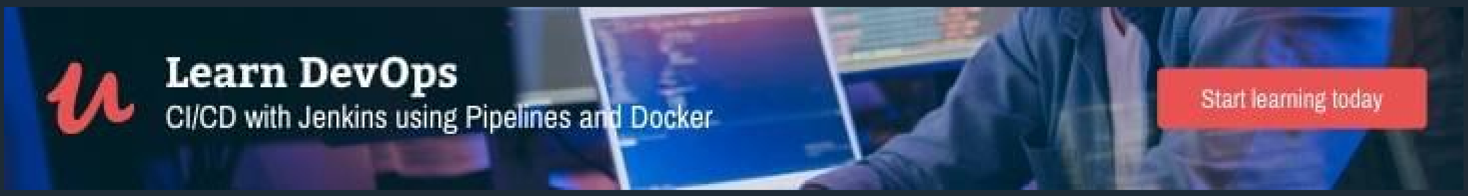
Nov 2, 2019 / 6 min. read /
16 Comments

Welcome to the first blog post of the "Jenkins Pipeline Cookbook" series. Today we are focus on working with Jenkins Pipeline environment variables effectively. You will learn how to define env variables, how to updated them, and how to use them in boolean expressions correctly.

JENKINS PIPELINE COOKBOOK
How to time out Jenkins Pipeline stage and keep the pipeline running?

Apr 16, 2020 / 4 min. read /
0 Comments

The declarative Jenkins Pipeline allows us to define timeout either at the pipeline level or the specific stage. This feature prevents Jenkins's job from getting stuck. However, in some cases, we want to accept that one stage may timeout, but we want to keep the remaining stages running.



More posts from
Jenkins Pipeline Cookbook

Want to paste code in the comment and make the formatting look good? Read this - Syntax highlighting.

18 Comments e.printstacktrace.blog Disqus' Privacy Policy Login

Recommend Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Wesley A Chambers • 20 days ago • edited

This is really helpful, Szymon! As someone very new to Jenkins, I do have a question. Suppose I have a **declarative** pipeline and I want to have the option to restart stages. However, I have a `script { ... }` tag that contains an iterator that dynamically builds stages (I can say why I have to do this if necessary). Will that preserve the restart from stage functionality or do I lose it? In the example below, I would rather not have to copy and paste a stage for each item. But I still need to restart at stage. Any thoughts?

```
pipeline {
agent any
environment {
SOME_VARIABLE = 'this,that,etc'
...
}
stages {
stage('a stage') {
when {
not {
branch 'master'
}
}
steps {
script {
sh 'shell script'
}
}
}
stage('another stage') {
when {
not {
branch 'master'
}
}
steps {
nodejs(nodeJSInstallationName: 'Node 10.5.0') {
sh 'some more shell scripts'
}
}
}
script { // does this mean the deploy stages won't restart at last fail??
SOME_VARIABLE.tokenize(',').each { item ->
stage("deploy ${item}") {
when {
branch 'master'
}
steps {
nodejs(nodeJSInstallationName: 'Node 10.5.0') {
sh "do things ${item}"
}
}
}
}
}
}
...
}
```

1 | • Reply •

Wesley A Chambers → Wesley A Chambers • 20 days ago • edited

Ugh, sorry for how ugly it looks! The comments do not allow my indents to stay.

^ | ⌄ • Reply •

Szymon Stepniak Mod ➦ Wesley A Chambers • 19 days ago

Hi, Wesley! You can edit your previous comment and add syntax highlighting (https://help.disqus.com/en/.... Regarding your question - when you add "stage" inside the "script" block, you are adding a scripted stage. This is not an equivalent of a stage in the declarative pipeline (you can't even use "when" or "steps" inside of this scripted "stage", and Jenkins will fail trying to compile your script.) What will happen in such a case is the following - when you select any of the dynamically added stages in the Blue Ocean UI, the "Restart {stage}" link will use the stage name of the declarative stage that added those scripted stages. And if you click on that restart, Jenkins will restart from that declarative stage. So if your declarative stage is "A" and you add two stages, "B" and "C" inside the script block of that stage, selecting "C" stage will say "Restart A", and clicking on it will restart the "A" stage.

PS: Check this video I recorded some time ago - https://www.youtube.com/wat... It explains in more details why there is such a difference between declarative stages and the scripted ones. I hope it helps you.

^ | ⌄ • Reply •

Wesley A Chambers ➦ Szymon Stepniak • 18 days ago

Very insightful! This is what I needed to know. Thank you!

1 ^ | ⌄ • Reply •

Szymon Stepniak Mod ➦ Wesley A Chambers • 18 days ago

I'm glad I could help you. Good luck, Wesley!

^ | ⌄ • Reply •

Ruta Borkar • 2 months ago

Amazing post. A very good read. Have 1 doubt. Is it like we mention pipeline as first word in declarative type and node as first word in scripted type??
And if this is the case, 'Listing 7' is the scripted one or declarative one?
I am a learner please don't mind this basic question.

1 ^ | ⌄ • Reply •

Szymon Stepniak Mod ➦ Ruta Borkar • a month ago

Hi Ruta! Don't be afraid to ask questions. Listing 7 shows the declarative pipeline as you suspect. The declarative pipeline always starts with the `pipeline` block at the top level.

```
pipeline {
    // ....
}
```

The scripted pipeline on the other hand, starts with the `node` block at the top level.

```
node {
    // ...
}
```

When you use the declarative `pipeline` block, Jenkins will validate its syntax and it will check if there is only one `pipeline` defined in the Jenkinsfile. (If it detects two or more, it will fail instantly with a proper message.) In case of the `node` block, there is no such validation, and you can mix multiple nodes in the single Jenkinsfile. (Mixing multiple nodes in the scripted pipeline is the way to run specific stages or blocks of code on the different worker nodes - the declarative pipeline uses `agent` configuration for that.)

^ | ⌄ • Reply •

Sudarshan T • 2 months ago

Nice post.

1 ^ | ⌄ • Reply •

Szymon Stepniak Mod ➦ Sudarshan T • 2 months ago

Thanks, Sudarshan!

^ | ⌄ • Reply •

colinbut • 2 months ago

Great post! I definitely learned something new that i hadn't knew before

1 ^ | ⌄ • Reply •

Szymon Stepniak Mod ➜ colinbut • 2 months ago
Thanks, colinbut! I'm happy to hear I was able to help you 😁
^ | ˅ • Reply •

Adam Yao • 3 months ago
Hey Szymon,
Great post, clear an concise!
1 ^ | ˅ • Reply •

Szymon Stepniak Mod ➜ Adam Yao • 3 months ago
Thanks, Adam! I'm glad to hear you found it useful! 🙂
^ | ˅ • Reply •

Mayank Singh Rathore • 3 months ago
Hey! Thanks for beautifully taking up this issue. I lived in confusion for a long time. Your posted sorted it out for me.

I'd love it if you could check out my space sometime too! :)
MY BLOG
1 ^ | ˅ • Reply •

Szymon Stepniak Mod ➜ Mayank Singh Rathore • 3 months ago
Thanks, Mayank! Happy to hear it was helpful! 🙂
^ | ˅ • Reply •

Kamil Jed • 2 months ago

Useful links

Start here
About
Archives
Privacy Policy
My Kit
RSS
Support me ❤

Popular categories

Groovy Cookbook 26
Jenkins Pipeline Cookbook 7
Programmer's Bookshelf 6
Ratpack Cookbook 4
Learning Java 4
Micronaut Cookbook 3
How to 3