Integration Pipelines

Raúl Estrada

Octubre 2020

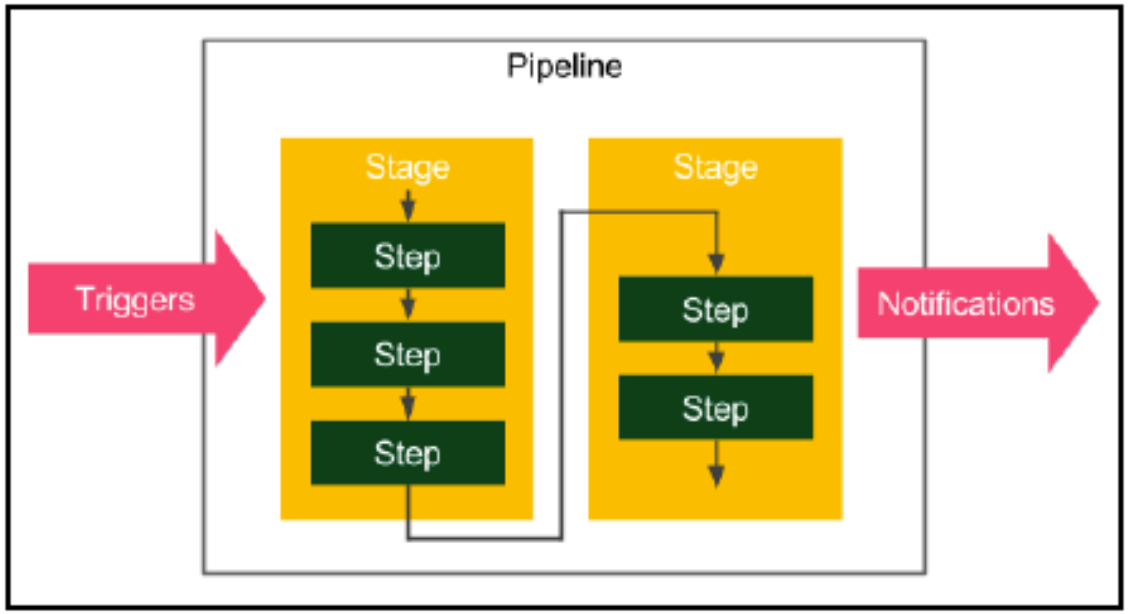
Introducing pipelines

A pipeline is a sequence of automated operations that usually represents a part of software delivery and the quality assurance process. It can be simply seen as a chain of scripts providing the following additional benefits:

- **Operation grouping**: Operations are grouped together into stages (also known as **gates** or **quality gates**) that introduce a structure into the process and clearly defines the rule: if one stage fails, no further stages are executed
- **Visibility**: All aspects of the process are visualized, which help in quick failure analysis and promotes team collaboration
- **Feedback**: Team members learn about any problems as soon as they occur, so they can react quickly

Pipeline structure

A Jenkins pipeline consists of two kinds of elements: stages and steps. The following figure shows how they are used:



The following are the basic pipeline elements:

- **Step**: A single operation (tells Jenkins what to do, for example, checkout code from repository, execute a script)
- **Stage**: A logical separation of steps (groups conceptually distinct sequences of steps, for example, **Build**, **Test**, and **Deploy**) used to visualize the Jenkins pipeline progress

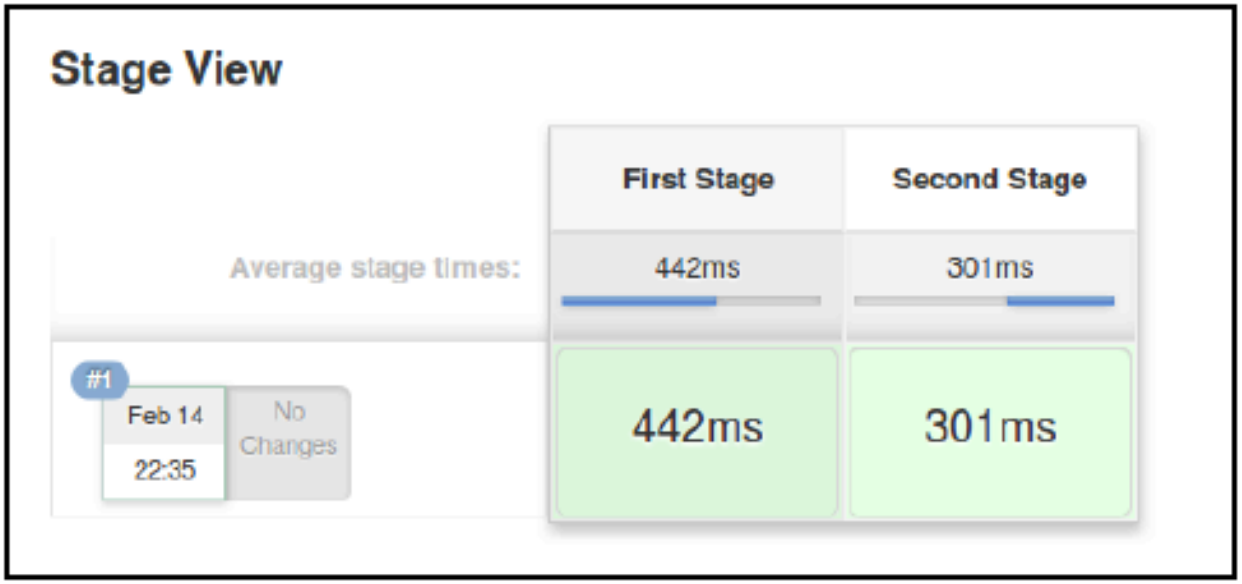
Technically, it's possible to create parallel steps; however, it's better to treat it as an exception when really needed for optimization purposes.

Multi-stage Hello World

As an example, let's extend the `Hello World` pipeline to contain two stages:

```
pipeline {
      agent any
      stages {
            stage('First Stage') {
                  steps {
                        echo 'Step 1. Hello World'
                  }
            }
            stage('Second Stage') {
                  steps {
                        echo 'Step 2. Second time Hello'
                        echo 'Step 3. Third time Hello'
                  }
            }
      }
}
```

The pipeline has no special requirements in terms of environment (any slave agent), and it executes three steps inside two stages. When we click on **Build Now**, we should see the visual representation:



The pipeline succeeded, and we can see the step execution details by clicking on the console. If any of the steps failed, the processing would stop and no further steps would run. Actually, the entire reason for a pipeline is to prevent all further steps from execution and visualize the point of failure.

Pipeline syntax

We have discussed the pipeline elements and already used a few of the pipeline steps, for example, `echo`. What other operations can we use inside the pipeline definition?

The declarative syntax was designed to make it as simple as possible to understand the pipeline, even by the people who do not write code on a daily basis. This is why the syntax is limited only to the most important keywords.

Let's prepare an experiment and, before we describe all the details, read the following pipeline definition and try to guess what it does:

```
pipeline {
    agent any
    triggers { cron('* * * * *') }
    options { timeout(time: 5) }
    parameters {
        booleanParam(name: 'DEBUG_BUILD', defaultValue: true,
        description: 'Is it the debug build?')
    }
    stages {
        stage('Example') {
            environment { NAME = 'Rafal' }
            when { expression { return params.DEBUG_BUILD } }
            steps {
                echo "Hello from $NAME"
                script {
                    def browsers = ['chrome', 'firefox']
                    for (int i = 0; i < browsers.size(); ++i) {
                        echo "Testing the ${browsers[i]} browser."
                    }
                }
            }
        }
    }
    post { always { echo 'I will always say Hello again!' } }
}
```

Hopefully, the pipeline didn't scare you. It is quite complex. Actually, it is so complex that it contains all possible Jenkins instructions. To answer the experiment puzzle, let's see what the pipeline does instruction by instruction:

1. Use any available agent.
2. Execute automatically every minute.
3. Stop if the execution takes more than 5 minutes.
4. Ask for the Boolean input parameter before starting.
5. Set `Rafal` as the environment variable NAME.
6. Only in the case of the `true` input parameter:
    - Print `Hello from Rafal`
    - Print `Testing the chrome browser`
    - Print `Testing the firefox browser`
7. Print `I will always say Hello again!` no matter if there are any errors during the execution.

Sections

Sections define the pipeline structure and usually contain one or more directives or steps. They are defined with the following keywords:

- **Stages**: This defines a series of one or more stage directives
- **Steps**: This defines a series of one or more step instructions
- **Post**: This defines a series of one or more step instructions that are run at the end of the pipeline build; marked with a condition (for example, always, success, or failure), usually used to send notifications after the pipeline build (we will cover this in detail in the *Triggers and notifications* section.)

Directives

Directives express the configuration of a pipeline or its parts:

- **Agent**: This specifies where the execution takes place and can define the `label` to match the equally labeled agents or `docker` to specify a container that is dynamically provisioned to provide an environment for the pipeline execution
- **Triggers**: This defines automated ways to trigger the pipeline and can use `cron` to set the time-based scheduling or `pollScm` to check the repository for changes (we will cover this in detail in the *Triggers and notifications* section)

- **Options**: This specifies pipeline-specific options, for example, `timeout` (maximum time of pipeline run) or `retry` (number of times the pipeline should be rerun after failure)
- **Environment**: This defines a set of key values used as environment variables during the build

- **Parameters**: This defines a list of user-input parameters
- **Stage**: This allows for logical grouping of steps
- **When**: This determines whether the stage should be executed depending on the given condition

Steps

Steps are the most fundamental part of the pipeline. They define the operations that are executed, so they actually tell Jenkins **what to do.**

- **sh**: This executes the shell command; actually, it's possible to define almost any operation using `sh`
- **custom**: Jenkins offers a lot of operations that can be used as steps (for example, `echo`); many of them are simply wrappers over the `sh` command used for convenience; plugins can also define their own operations
- **script**: This executes a block of the Groovy-based code that can be used for some non-trivial scenarios, where flow control is needed