Syntax and Structure of Declarative Pipelines

In the previous lesson, we learned about the differences between Declarative Pipelines and Scripted Pipelines in Jenkins. Now, let’s dive deeper into the syntax and structure of Declarative Pipelines and understand how to construct a pipeline using this approach.

Basic Structure of a Declarative Pipeline

A Declarative Pipeline is defined within a `pipeline` block in the `Jenkinsfile`. The basic structure of a Declarative Pipeline consists of the following elements:

```
pipeline {
  agent { /* Specify the agent to run the pipeline */ }

  stages {
    stage('Stage 1') {
      steps {
        /* Define the steps for Stage 1 */
      }
    }
    stage('Stage 2') {
      steps {
        /* Define the steps for Stage 2 */
      }
    }
    /* Add more stages as needed */
  }
}
```

Let’s break down each element:

1. `pipeline`: The outermost block that encapsulates the entire pipeline.
2. `agent`: Specifies where the pipeline should be executed. It can be any of the following:

- `any`: Run the pipeline on any available agent.
- `none`: No global agent. Each stage must specify its own agent.
- `label`: Run the pipeline on an agent with a specific label.
- `docker`: Run the pipeline inside a Docker container.

1. `stages`: Contains the stages of the pipeline. Each stage represents a logical grouping of steps.
2. `stage`: Defines a named stage within the pipeline. It typically represents a specific phase of the CI/CD process, such as Build, Test, or Deploy.
3. `steps`: Contains the individual steps to be executed within a stage. Steps can include shell commands, Jenkins built-in functions, or custom Groovy code.

Directives in Declarative Pipelines

Declarative Pipelines provide a set of directives that allow you to configure and control the behavior of your pipeline. Some commonly used directives include:

1. `environment`: Defines environment variables that can be accessed throughout the pipeline.

```
pipeline {
  agent any

  environment {
    NODE_ENV = 'production'
  }

  /* Stages */
}
```

2. `parameters`: Defines parameters that can be passed to the pipeline during execution.

```
pipeline {
  agent any

  parameters {
    string(name: 'DEPLOY_ENV', defaultValue: 'staging', description: 'Deployment environment')
  }

  /* Stages */
}
```

3. `options`: Specifies pipeline-specific options, such as timeout, retry, and more.

```
pipeline {
  agent any

  options {
    timeout(time: 1, unit: 'HOURS')
    retry(3)
  }

  /* Stages */
}
```

4. `when`: Defines conditions for executing a stage based on certain criteria.

```
pipeline {
  agent any

  stages {
    stage('Deploy') {
      when {
```

```
                branch 'main'
            }
            steps {
                /* Deployment steps */
            }
        }
    }
}
```

Parallel Stages

Declarative Pipelines also support parallel execution of stages. You can define multiple stages within a `parallel` block to run them concurrently.

```
pipeline {
    agent any

    stages {
        stage('Parallel Stages') {
            parallel {
                stage('Stage 1') {
                    steps {
                        /* Steps for Stage 1 */
                    }
                }
                stage('Stage 2') {
                    steps {
                        /* Steps for Stage 2 */
                    }
                }
            }
        }
    }
}
```

In this example, Stage 1 and Stage 2 will be executed in parallel, potentially saving time in the overall pipeline execution.

Post Actions

Declarative Pipelines provide a `post` section that allows you to define actions to be executed at the end of the pipeline, regardless of the pipeline's outcome.
The `post` section supports the following conditions:

- `always`: Run the steps regardless of the pipeline's outcome.
- `success`: Run the steps only if the pipeline succeeds.
- `failure`: Run the steps only if the pipeline fails.
- `unstable`: Run the steps if the pipeline is marked as unstable (e.g., test failures).

- `changed`: Run the steps if the pipeline's status has changed from the previous run.

```
pipeline {
  agent any

  /* Stages */

  post {
    always {
      /* Steps to run always */
    }
    success {
      /* Steps to run on success */
    }
    failure {
      /* Steps to run on failure */
    }
  }
}
```

The `post` section is useful for performing cleanup tasks, sending notifications, or triggering additional actions based on the pipeline's outcome.

Conclusion

Declarative Pipelines provide a structured and expressive way to define your CI/CD pipelines in Jenkins. By understanding the basic structure, directives, parallel stages, and post actions, you can create powerful and maintainable pipelines.

In the next lesson, we'll explore the different stages, steps, and post actions available in Declarative Pipelines and learn how to use them effectively to build a robust CI/CD pipeline.

Get ready to master the syntax and structure of Declarative Pipelines and take your Jenkins pipelines to the next level!