Writing Jenkinsfile for Defining Pipeline as Code

In the previous lessons, we learned about the key components of Declarative Pipelines in Jenkins, including stages, steps, and post actions. Now, let's put that knowledge into practice and learn how to write a complete Jenkinsfile to define our pipeline as code.

What is a Jenkinsfile?

A Jenkinsfile is a text file that contains the definition of a Jenkins Pipeline. It is written using the Declarative Pipeline syntax and is typically stored in the root directory of your source code repository. The Jenkinsfile is versioned along with your code, enabling you to track changes and maintain a history of your pipeline.

Creating a Jenkinsfile

To create a Jenkinsfile, follow these steps:

1. Create a new file named `Jenkinsfile` (without any extension) in the root directory of your source code repository.
2. Open the Jenkinsfile in a text editor.
3. Begin by defining the pipeline using the `pipeline` block:

```
pipeline {
  /* Pipeline definition */
}
```

4. Specify the agent on which the pipeline should run using the `agent` directive:

```
pipeline {
  agent any

  /* Pipeline stages */
}
```

5. Define the stages of your pipeline within the `stages` block:

```
pipeline {
  agent any

  stages {
    stage('Build') {
      steps {
        /* Build steps */
      }
    }
    stage('Test') {
      steps {
        /* Test steps */
```

```
        }
      }
      stage('Deploy') {
        steps {
          /* Deploy steps */
        }
      }
    }
}
```

6. Add steps to each stage to define the actions to be performed:

```
pipeline {
  agent any

  stages {
    stage('Build') {
      steps {
        sh 'npm install'
        sh 'npm run build'
      }
    }
    stage('Test') {
      steps {
        sh 'npm test'
      }
    }
    stage('Deploy') {
      steps {
        sh 'npm run deploy'
      }
    }
  }
}
```

7. (Optional) Include post actions to perform additional tasks based on the pipeline's outcome:

```
pipeline {
  agent any

  stages {
    /* Pipeline stages */
  }

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

```
  }
}
```

8. Save the Jenkinsfile and commit it to your source code repository.

Example Jenkinsfile

Here's an example of a complete Jenkinsfile for a Node.js application:

```
pipeline {
  agent any

  stages {
    stage('Checkout') {
      steps {
        checkout scm
      }
    }

    stage('Build') {
      steps {
        sh 'npm install'
        sh 'npm run build'
      }
    }

    stage('Test') {
      steps {
        sh 'npm test'
      }
      post {
        always {
          junit 'test-results/**/*.xml'
        }
      }
    }

    stage('Deploy') {
      when {
        branch 'main'
      }
      steps {
        sh 'npm run deploy'
      }
    }
  }

  post {
    always {
      cleanWs()
    }
    success {
      archiveArtifacts 'dist/**'
    }
    failure {
```

```
            mail to: 'team@example.com', subject: 'Pipeline Failed', body: 'The pipeline has failed. Please investigate.'
        }
    }
}
```

In this example:

- The pipeline runs on any available agent.
- The Checkout stage checks out the source code from the repository.
- The Build stage installs dependencies and builds the application.
- The Test stage runs tests and publishes the test results using the `junit` step.
- The Deploy stage deploys the application only when the branch is 'main'.
- The post actions clean the workspace, archive artifacts on success, and send an email notification on failure.

Conclusion

Writing a Jenkinsfile is a crucial step in defining your pipeline as code. By using the Declarative Pipeline syntax and following the structure of stages, steps, and post actions, you can create a comprehensive and maintainable pipeline definition.

Remember to store your Jenkinsfile in version control alongside your source code. This allows you to track changes, collaborate with team members, and ensure that your pipeline evolves with your application.

In the next lesson, we'll explore best practices for writing efficient and maintainable pipelines, including tips for organizing your Jenkinsfile, handling secrets, and optimizing pipeline performance.

Get ready to put your Jenkinsfile writing skills into practice and take your CI/CD pipelines to the next level!