
## JENKINS

- **Jenkins** is an open-source automation tool for **Continuous Integration (CI)** and **Continuous Delivery (CD)**.
- It helps automate parts of software development, such as building, testing, and deploying code.
- Written in **Java**, it has a vast plugin ecosystem, allowing integration with numerous third-party tools.

## JENKINS ARCHITECTURE

## MASTER SLAVE ARCHITECTURE

- **Master**: The central server that manages jobs, monitors slaves, and provides the user interface.
- **Slave**: Executes jobs assigned by the master. Slaves can run on different machines to perform distributed builds, optimizing resource usage and reducing build times.

BENEFITS-

- **Scalability**: Distributed builds on multiple nodes (slaves).
- **Load Balancing**: Tasks can be distributed across multiple machines.

## CONTINUOUS INTEGRATION/CONTINUOUS DELIVERY (CI/CD) WITH JENKINS

- **CI**: Jenkins automatically builds and tests the code every time a developer commits to a version control system.
- **CD**: Jenkins ensures that the code is always deployable. Deployments can be automatically triggered after a successful build.

**Key Benefits**:

- **Faster Development**: Reduces the time between writing code and getting feedback.
- **Bug Detection**: Early detection of errors through automated testing.

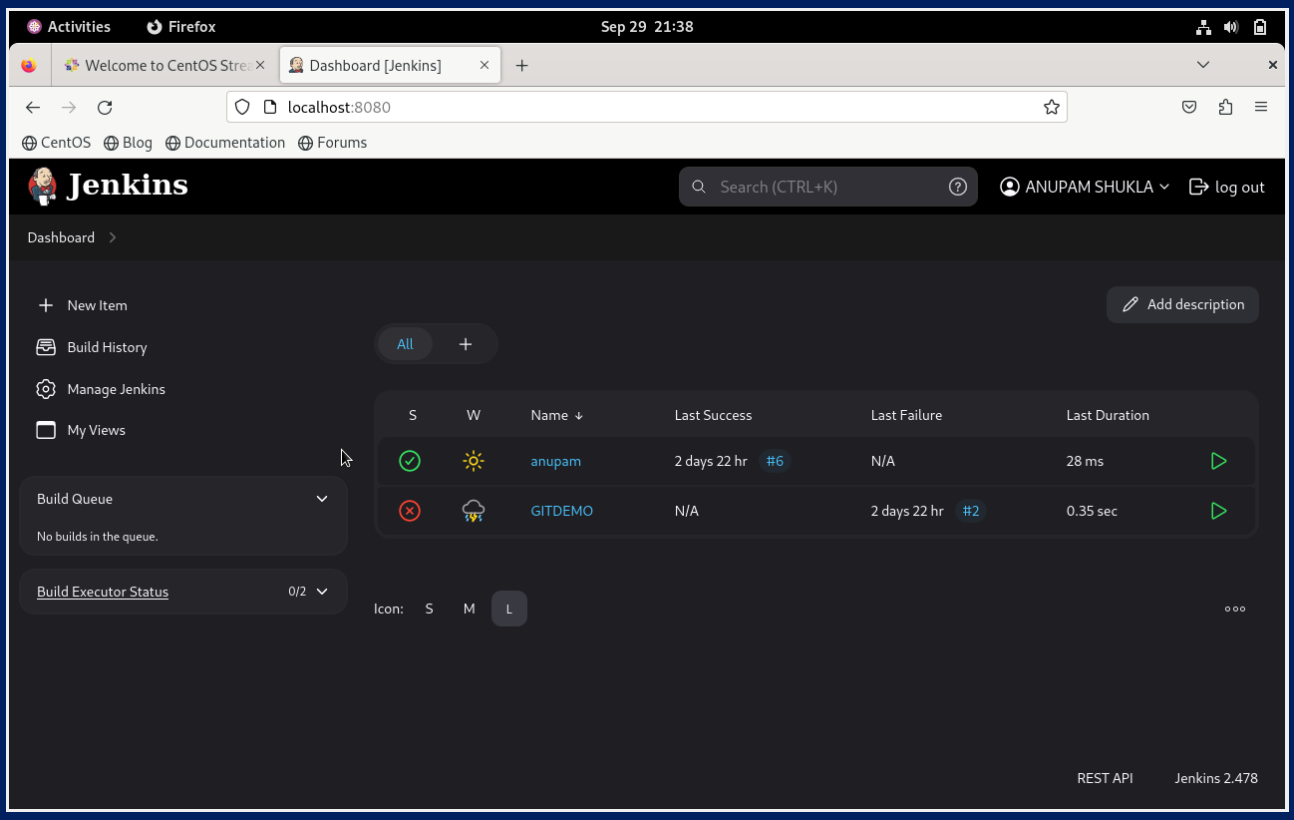
ANUPAM SHUKLA

## JENKINS COMPONENTS

Jenkins is made up of several components that work together to enable Continuous Integration (CI) and Continuous Delivery (CD). Understanding each of these components will help you effectively set up, configure, and use Jenkins.

### Jenkins Dashboard

- **Overview**: The Jenkins dashboard is the user interface where you can manage jobs, builds, and configurations.
- **Features**:
  - Create and view jobs.
  - Monitor the status of running builds.
  - View recent build history.
  - Access system configurations.




### Jobs/Projects

- **Definition**: A **job** or **project** in Jenkins is a task or series of steps to be executed, such as building, testing, or deploying code.
- **Types of Jobs**:
  - **Freestyle Project**: Simple jobs with custom build steps.
  - **Pipeline**: Allows you to define complex job sequences in code.
  - **Multibranch Pipeline**: Automatically creates pipelines for branches in version control systems like Git.

## Builds

- **Definition**: A build in Jenkins refers to the actual execution of a job, such as compiling code or running tests.
- **Features**:
  - Builds can be triggered manually, automatically via SCM polling, or through a webhook.
  - Jenkins stores the build history, logs, and artifacts for each build.

## Pipeline

- **Definition**: A **pipeline** is a series of automated steps (build, test, deploy) defined as code using either **Declarative** or **Scripted** syntax.
  - **Declarative Pipeline:** More structured, simpler to use.
  - **Scripted Pipeline:** More flexible but requires Groovy scripting.

## Nodes (Master/Slave Architecture)

- **Master**: The main Jenkins server responsible for orchestrating jobs and builds.
- **Slave**: A machine connected to the master, where jobs are executed. Slaves help distribute workload by running jobs in parallel or on specific platforms.

**Master Responsibilities:**

- Scheduling jobs.
- Monitoring nodes (slaves).
- Recording and presenting build results.

**Slave Responsibilities:**

- Performing jobs assigned by the master.
- Communicating back to the master with build results.

## Plugins

- **Definition**: Jenkins' functionality can be extended through plugins.
- **Types of Plugins**:
  - **Source Code Management Plugins:** Git, SVN.
  - **Build Tools:** Maven, Gradle, Ant.
  - **Notification Tools:** Slack, Email, SMS.
  - **Test Report Plugins**: JUnit, TestNG.
  - **Containerization:** Docker.

**How to Manage Plugins**:

- Go to **Manage Jenkins > Manage Plugins**.
- **Install**: Search for plugins in the **Available** tab and install them.
- **Update/Remove**: Use the **Installed** tab to manage current plugins.

## Source Code Management (SCM)

- Jenkins integrates with SCM tools to track and fetch changes in code repositories.
  - **Popular SCMs**: Git, Subversion (SVN), Mercurial.
- **Poll SCM**: Jenkins can periodically check SCM for changes. If any changes are detected, it triggers a build.

## Poll SCM and Webhooks

- **Poll SCM**: A method where Jenkins periodically checks for code changes in the SCM repository.
  - Useful if you don't want to rely on webhooks.
- **Webhooks**: When a change is made to the repository (push event), the repository sends a request to Jenkins to trigger the job. Webhooks are preferred for real-time builds.

## Build Triggers

- **Definition**: Build triggers are mechanisms to automatically start a build based on certain conditions.
- **Types**:
  - **Manual Trigger**: A user manually triggers the job from the Jenkins dashboard.
  - **SCM Polling**: Jenkins periodically checks the source code repository for changes.
  - **Webhook Trigger**: SCM automatically triggers the job on code push (e.g., GitHub webhook).
  - **Scheduled Build**: Jobs scheduled using Cron syntax to run at specified intervals.

## Post-Build Actions

- **Definition**: Tasks that run after the job finishes.
- **Examples**:
  - **Email Notifications**: Notify team members of build status (success, failure, etc.).
  - **Archive Artifacts**: Store build results like WAR files or JAR files.
  - **Trigger Other Jobs**: Launch dependent jobs if the build succeeds.

## Credentials

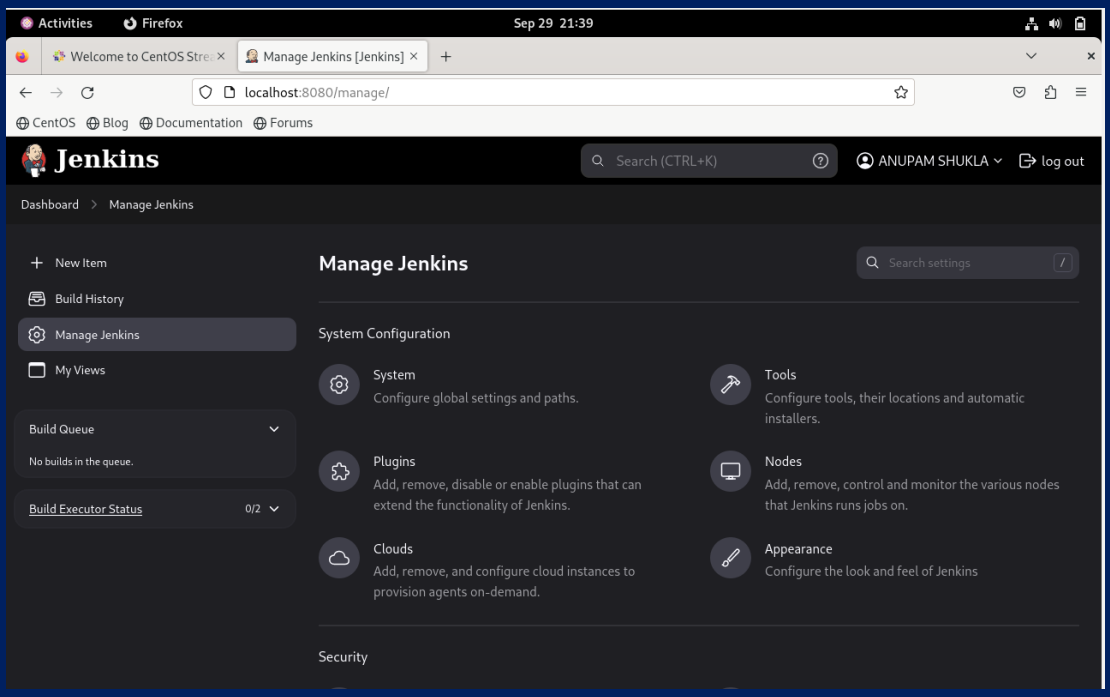
- **Definition**: Securely store and manage sensitive data (passwords, SSH keys, API tokens).
- **Usage**: Credentials are often used in jobs to connect to external services like SCM, databases, or deployment servers.

**Types of Credentials**:

- **Username and Password**: For basic authentication.
- **Secret Text**: For API tokens.
- **SSH Key**: For secure shell access.

## Manage Jenkins

- **Overview**: This section allows for global configuration and system management tasks.
-




  - **Configure Global Settings**: General settings like URL, environment variables, timeouts.
  - **Manage Plugins**: Install and manage plugins.
  - **System Configuration**: Define global tools (JDK, Maven, etc.).
  - **Security Setup**: Role-based access control (RBAC), configure user authentication.

## Global Tool Configuration

- **Definition**: Jenkins allows you to configure tools that will be used across various jobs, such as:
  - **JDK**: Configure Java versions.
  - **Maven**: Setup Maven installations for builds.
  - **Git**: Configure Git installations.

**Steps**:

- Go to **Manage Jenkins > Global Tool Configuration**.
- Define installation paths or let Jenkins automatically install tools.

## Notifications

- **Definition**: Jenkins can send email notifications or messages to communication tools (like Slack) to report build statuses.

**Common Notification Types**:

- **Email Notifications**: Use the **Email Extension Plugin** to send email updates on build success, failure, or any other condition.
- **Slack Notifications**: Use the **Slack Plugin** to integrate Jenkins with Slack and send build notifications to specific channels.

## Ansible Integration

- **Definition**: Jenkins can trigger Ansible playbooks as part of a deployment pipeline.
- **Usage**:
  - Install the **Ansible Plugin**.
  - Configure the **Ansible executable** under **Global Tool Configuration**.
  - Use ansiblePlaybook step in the pipeline to trigger Ansible automation tasks.

## Maven Integration

- **Definition**: Maven is a popular build automation tool, especially for Java projects. Jenkins integrates with Maven to run the build lifecycle.
- **Steps**:
  - Install the **Maven Integration Plugin**.
  - Configure Maven in **Global Tool Configuration**.
  - Set up a Jenkins job for a Maven project and specify the POM.xml file.

## Artifacts

- **Definition**: Artifacts are the outputs of the build process, such as JARs, WARs, or compiled code.
- **Usage**:
  - Jenkins allows archiving these artifacts for future reference.
  - Artifacts can also be deployed to other environments or stored in artifact repositories.

## Testing and Reporting

- **Definition**: Jenkins can run automated tests after each build using test frameworks such as **JUnit** or **TestNG**.
- **Test Reports**:
  - After the test runs, Jenkins can display the results in the form of reports (success, failure rates).
  - Plugins like **JUnit Plugin** or **TestNG Plugin** help parse and visualize test results.

## Build History

- **Definition**: Jenkins stores information about all previous builds (logs, artifacts, environment variables).
- **Usage**:
  - The build history allows developers to check logs and trace back to the cause of build failures or regressions.

## Jenkins Pipeline

- **Overview**: A Jenkins Pipeline is a sequence of stages and steps defining the automated process of building, testing, and deploying code. Pipelines can be written using **Declarative** or **Scripted** syntax.

### 1) Declarative Script

The **Declarative Pipeline** syntax is designed to be more readable and easier to use. It is the recommended approach for most Jenkins users.

```
pipeline {
    agent any                 // Define where the pipeline runs (can be "none" or specific agents)
    stages {                  // Define stages of the pipeline
        stage('Build') {       // Name of the stage
            steps {            // Steps to perform within this stage
                echo 'Building...'
            }
        }
        stage('Test') {
            steps {
                echo 'Testing...'
            }
        }
        stage('Deploy') {
            steps {
                echo 'Deploying...'
            }
        }
    }
    post {                    // Actions to take after the pipeline runs
        always {
            echo 'This will always run!'
        }
        success {
            echo 'This will run on success!'
        }
        failure {
            echo 'This will run on failure!'
        }
    }
}
```

**Key Components:**

- **pipeline**: Top-level block that defines the entire pipeline.
- **agent**: Specifies where the pipeline or a specific stage should run.
  - **any**: Run on any available agent.
  - **none**: No global agent (used when specifying individual agents per stage).
- **stages**: A block that contains one or more **stage** sections.
- **steps**: Inside a **stage**, the actual tasks to perform (commands, scripts, etc.).
- **post**: Defines actions (such as notifications or cleanup) that run at the end of the pipeline.

## Example: Complete Pipeline with Shell Commands

```
pipeline {
    agent any
    environment {
        MY_ENV_VAR = 'production'
    }
    stages {
        stage('Build') {
            steps {
                echo 'Building...'
                sh 'mvn clean package'
            }
        }
        stage('Test') {
            steps {
                echo 'Running tests...'
                sh 'mvn test'
            }
        }
        stage('Deploy') {
            steps {
                echo "Deploying to ${MY_ENV_VAR} environment..."
                sh 'ansible-playbook deploy.yml'
            }
        }
    }
    post {
        success {
            echo 'Pipeline completed successfully!'
        }
        failure {
            echo 'Pipeline failed!'
        }
    }
}
```

## 2) Scripted Script

The **Scripted Pipeline** is a more powerful and flexible way of defining pipelines, but it requires Groovy knowledge.

```
node {
    stage('Build') {
        echo 'Building...'
        sh 'mvn clean package'
    }

    stage('Test') {
        echo 'Testing...'
        sh 'mvn test'
    }

    stage('Deploy') {
        echo 'Deploying...'
        sh 'ansible-playbook deploy.yml'
    }
}
```

**Key Components:**

- **node**: Defines where the pipeline will run (equivalent to **agent** in Declarative Pipeline).
- **stage**: Used to define each phase of the pipeline (build, test, deploy).
- **sh**: Executes a shell command.
- **echo**: Prints a message to the console.

Example: **Parallel Stages in Scripted Pipeline**

```
node {
    stage('Build') {
        echo 'Building...'
        sh 'mvn clean package'
    }

    stage('Test') {
        parallel(
            'Unit Tests': {
                echo 'Running unit tests...'
                sh 'mvn test'
            },
            'Integration Tests': {
                echo 'Running integration tests...'
                sh 'mvn verify'
            }
        )
    }

    stage('Deploy') {
        echo 'Deploying...'
        sh 'ansible-playbook deploy.yml'
    }
}
```

## Declarative vs. Scripted Pipelines

| Feature | Declarative Pipeline | Scripted Pipeline |
|---|---|---|
| Syntax | Structured, easier to read | Free-form, more flexible |
| Usage | Recommended for most users | For advanced scenarios, Groovy knowledge needed |
| `agent` block | Required | Optional |
| `post` actions | Supports multiple conditions (e.g., success, failure) | Can be manually implemented with Groovy |
| Parallel execution | Easier to implement with the `parallel` keyword | Can be done, but more manual configuration |
| Error handling | Built-in error handling in `post` | Needs manual try-catch implementation |

## KEY CONCEPTS IN JENKIN PIPELINE

### Agent

Defines where the pipeline will run (on a specific node or any available node). For example:

```
pipeline {
    agent any    // Runs on any available agent
}
```

### Environment

Used to define environment variables that can be used in the pipeline. For example:

```
pipeline {
    environment {
        MY_ENV_VAR = 'value'
    }
    stages {
        stage('Example') {
            steps {
                echo "Environment variable is ${MY_ENV_VAR}"
            }
        }
    }
}
```

### Tools

Allows specifying tools such as JDK, Maven, or NodeJS to use within the pipeline:

```
pipeline {
    agent any
    tools {
        jdk 'JDK11'
        maven 'Maven3'
    }
    stages {
        stage('Build') {
            steps {
                sh 'mvn clean package'
            }
        }
    }
}
```

## Parameters

Used to define build parameters, allowing users to input values before starting the pipeline:

```
pipeline {
  parameters {
      string(name: 'BRANCH', defaultValue: 'master', description: 'Branch to build')
  }
  stages {
      stage('Build') {
          steps {
              echo "Building branch: ${params.BRANCH}"
          }
      }
  }
}
```

## Post Conditions

Actions that are executed after the pipeline (or a specific stage) finishes. Common conditions are:

- always: Runs regardless of the build result.
- success: Runs only if the build succeeds.
- failure: Runs only if the build fails.

```
pipeline {
  post {
      always {
          echo 'This will always run!'
      }
      success {
          echo 'The pipeline succeeded!'
      }
      failure {
          echo 'The pipeline failed!'
      }
  }
}
```

## Parallel Execution in Declarative Pipeline

run multiple stages or steps in parallel:

```
pipeline {
    agent any
    stages {
        stage('Parallel Stage') {
            parallel {
                stage('Build 1') {
                    steps {
                        echo 'Building Project 1...'
                    }
                }
                stage('Build 2') {
                    steps {
                        echo 'Building Project 2...'
                    }
                }
            }
        }
    }
}
```

## Retry

Automatically retry a stage upon failure.

```
stage('Build') {
    steps {
        retry(3) {
            sh 'mvn clean package'
        }
    }
}
```

## Timeout

Enforce a time limit on a stage.

```
stage('Build') {
    steps {
        timeout(time: 10, unit: 'MINUTES') {
            sh 'mvn clean package'
        }
    }
}
```

## Jenkins User Management

Jenkins supports role-based access control (RBAC), allowing you to define permissions for different user roles. Key concepts:

- **User accounts**: Individuals can log in and manage jobs.
- **Role-based permissions**: Admins can assign specific roles, such as read-only, job administrator, or global admin.
- **For this Role-Based Strategy Plugin needed**
- Admin can assign roles to other user like build,read and so on.

Q- Create a job to execute your name in shell and build takes place every minute.

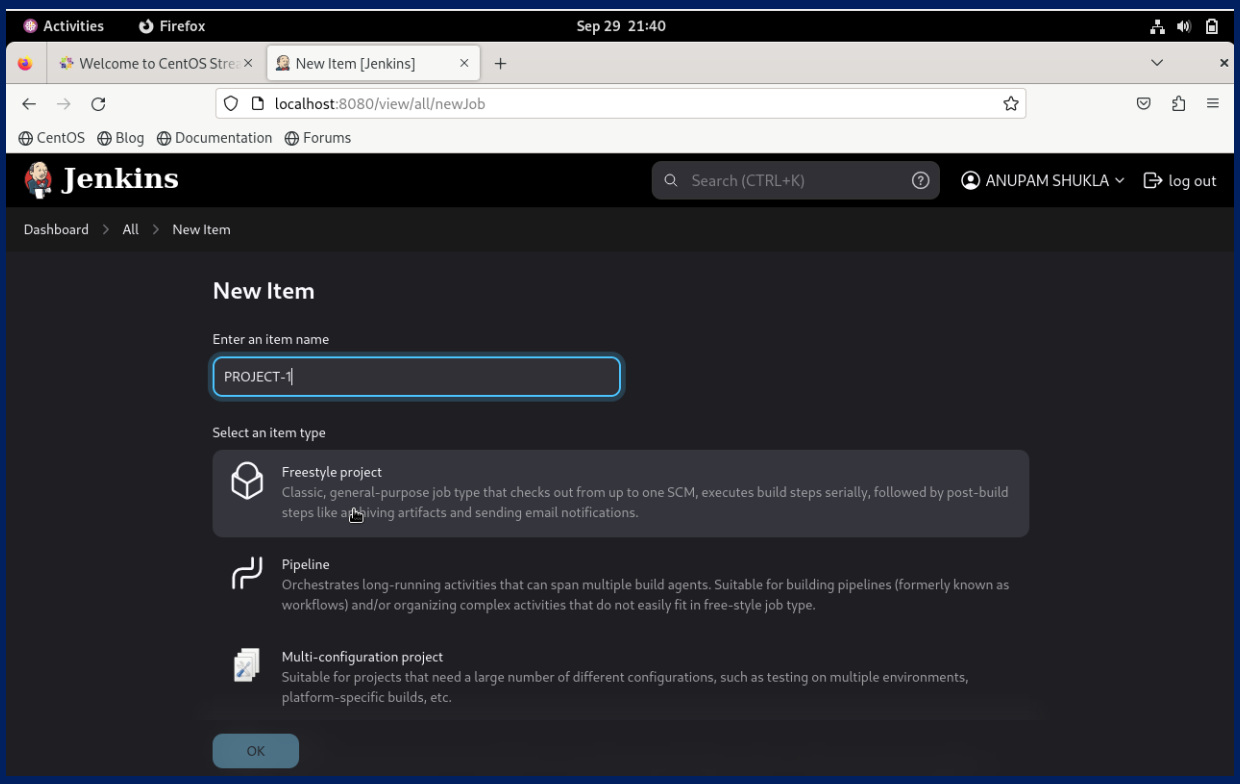
ANSWER-

STEP-1) Log in to Jenkins

Go to the Jenkins dashboard by entering the Jenkins URL in your browser.

STEP-2) Create a New Job

- Click on the **“New Item”** in the Jenkins dashboard.
- Enter a name for the job (e.g., “Print Name Job”).
- Select **“Freestyle Project”** as the project type
- Click **OK** to create the job.



## STEP-3) Configure the job

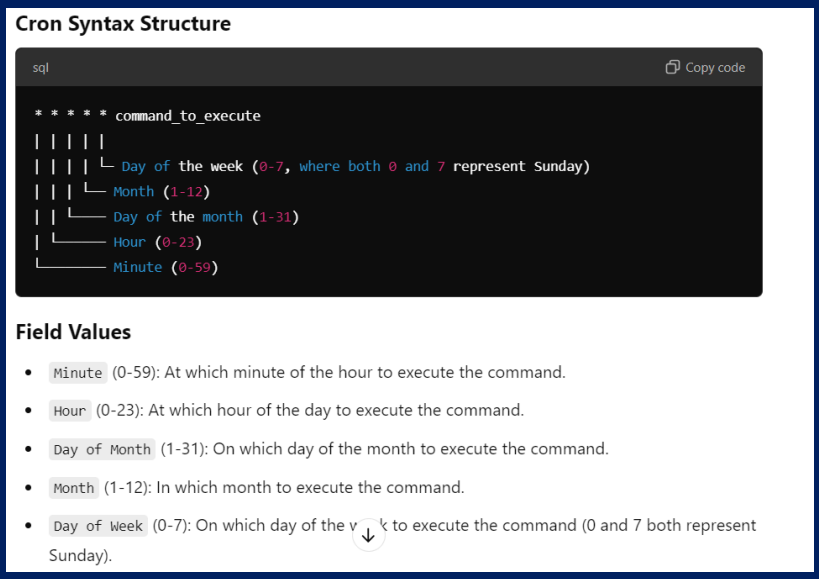
Once the job is created, you will be taken to the configuration page.

### General Configuration

- Add a brief description for the job (optional).

### Build Triggers

- Scroll down to the **“Build Triggers”** section.
- Check the **“Build periodically”** option.
- * * * * *= for every minute

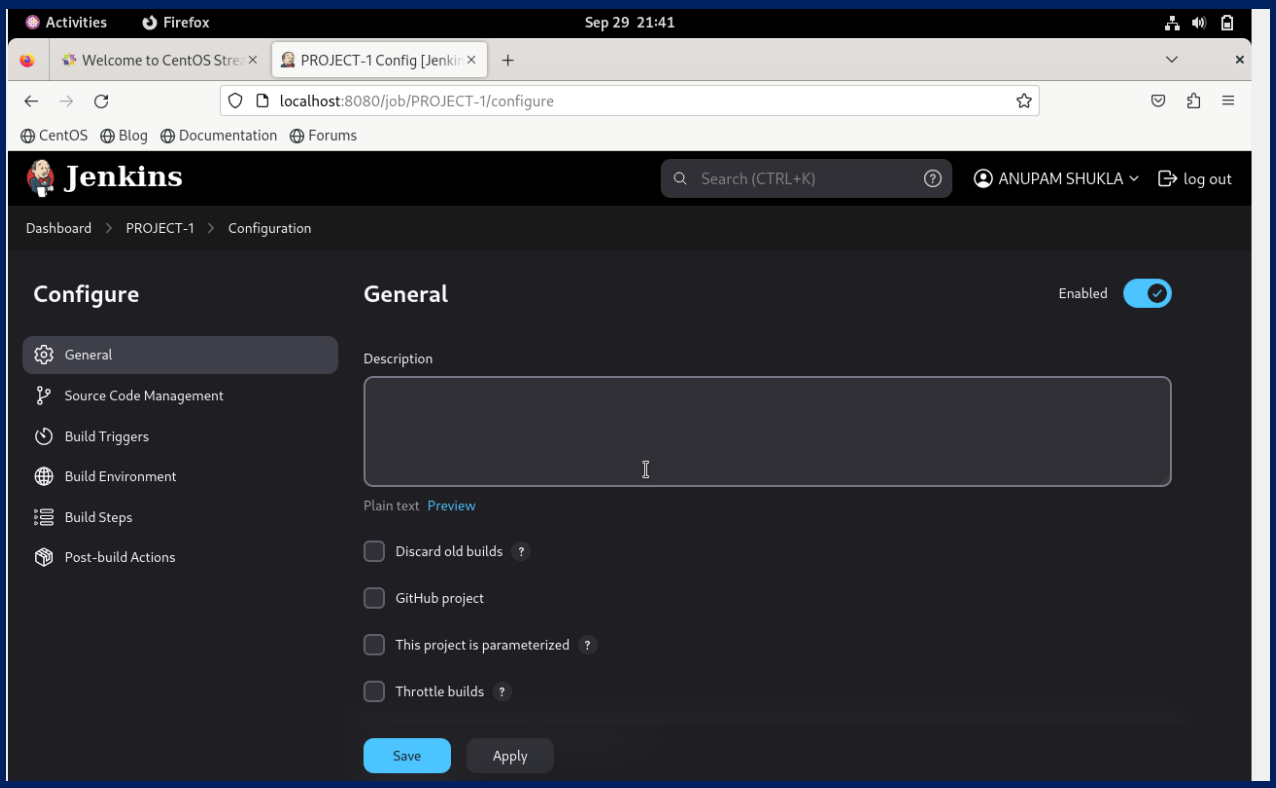


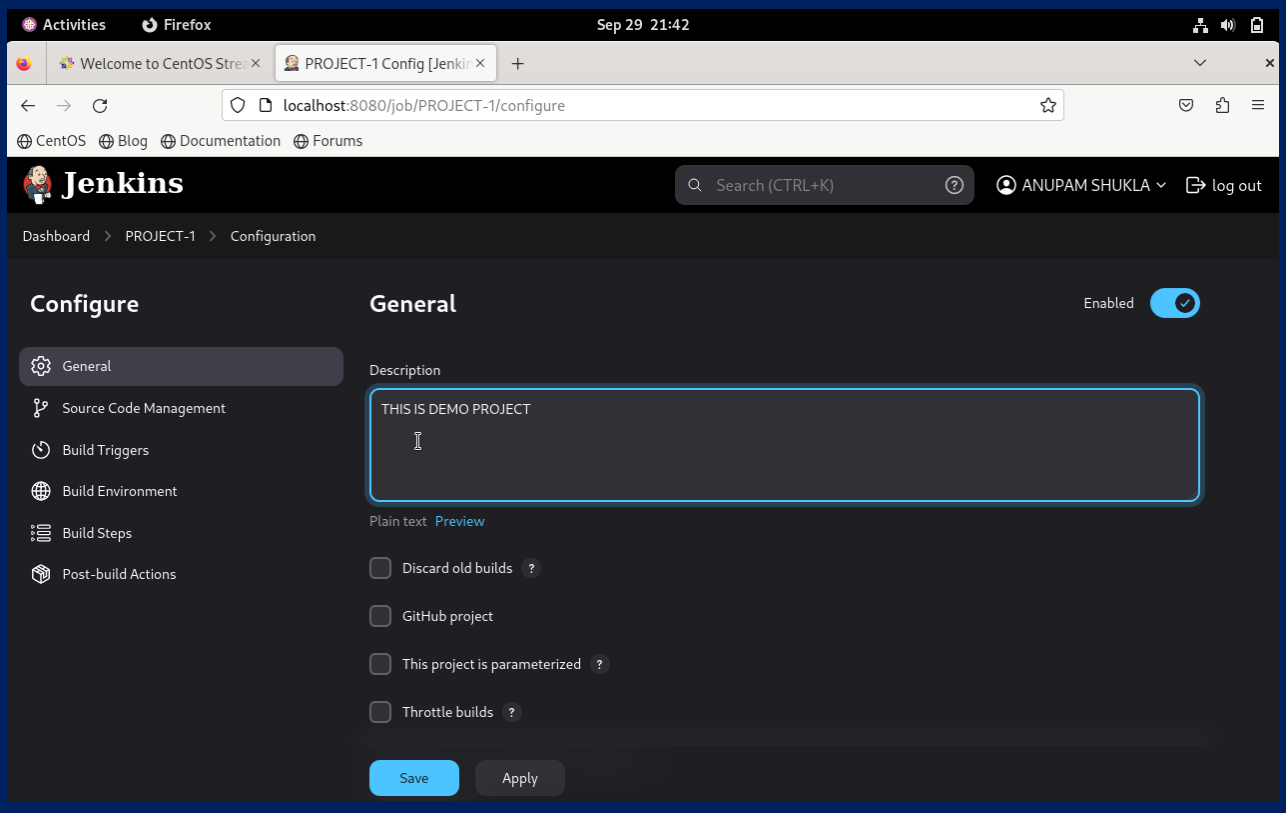
**Cron Syntax Structure**

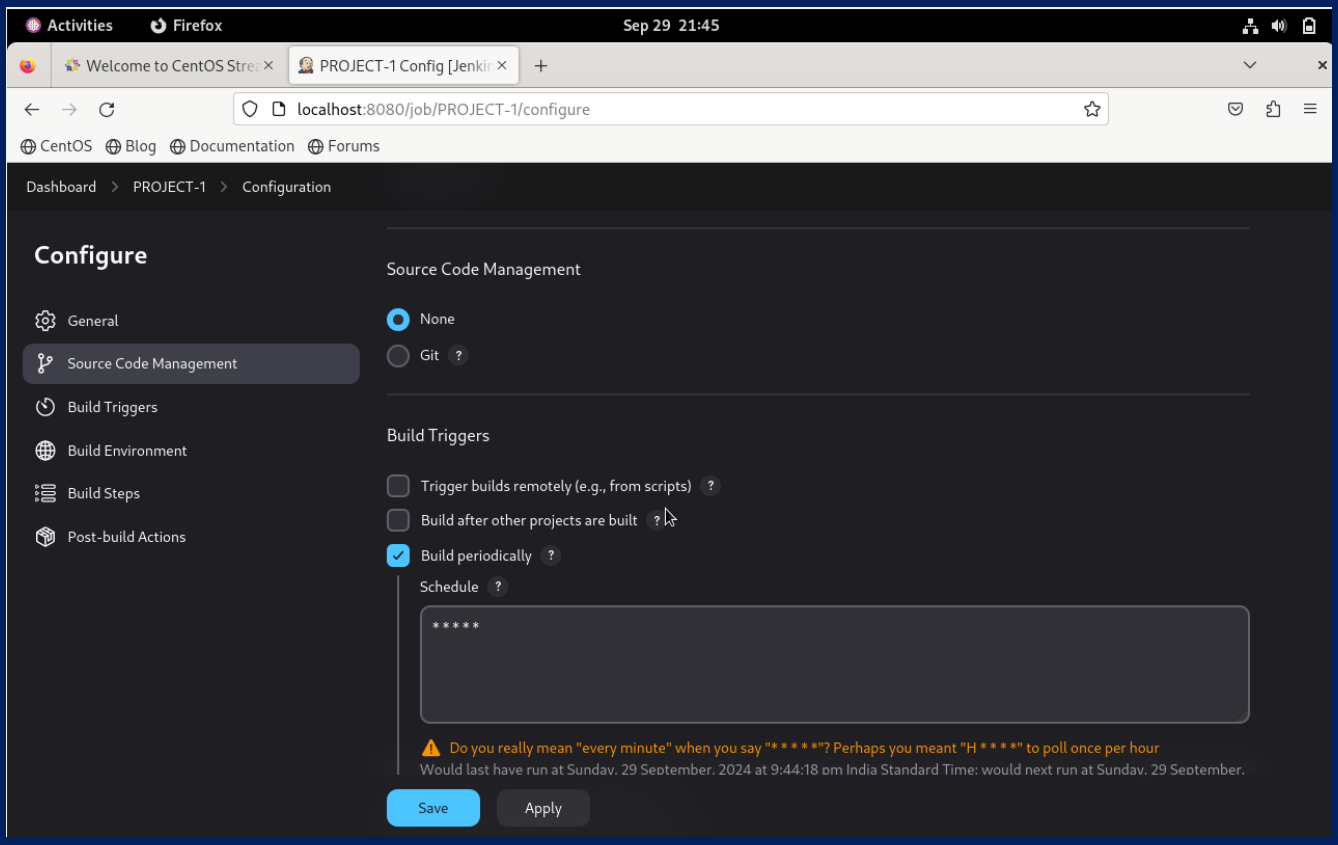
```
sql                                                    Copy code

* * * * * command_to_execute
| | | | |
| | | | └ Day of the week (0-7, where both 0 and 7 represent Sunday)
| | | └── Month (1-12)
| | └──── Day of the month (1-31)
| └────── Hour (0-23)
└──────── Minute (0-59)
```

**Field Values**

- `Minute` (0-59): At which minute of the hour to execute the command.
- `Hour` (0-23): At which hour of the day to execute the command.
- `Day of Month` (1-31): On which day of the month to execute the command.
- `Month` (1-12): In which month to execute the command.
- `Day of Week` (0-7): On which day of the w k to execute the command (0 and 7 both represent Sunday).

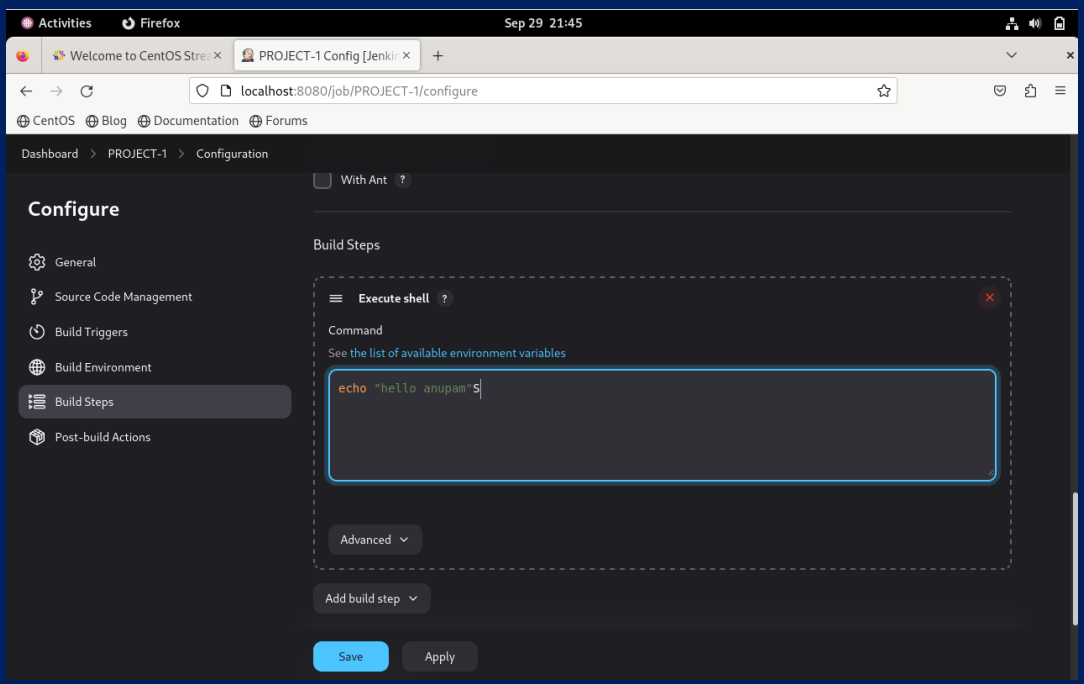





## Build Section

- Scroll to the “Build” section and click “Add build step”.
- Choose “Execute shell” from the dropdown.
- Type the script echo “hello anupam”



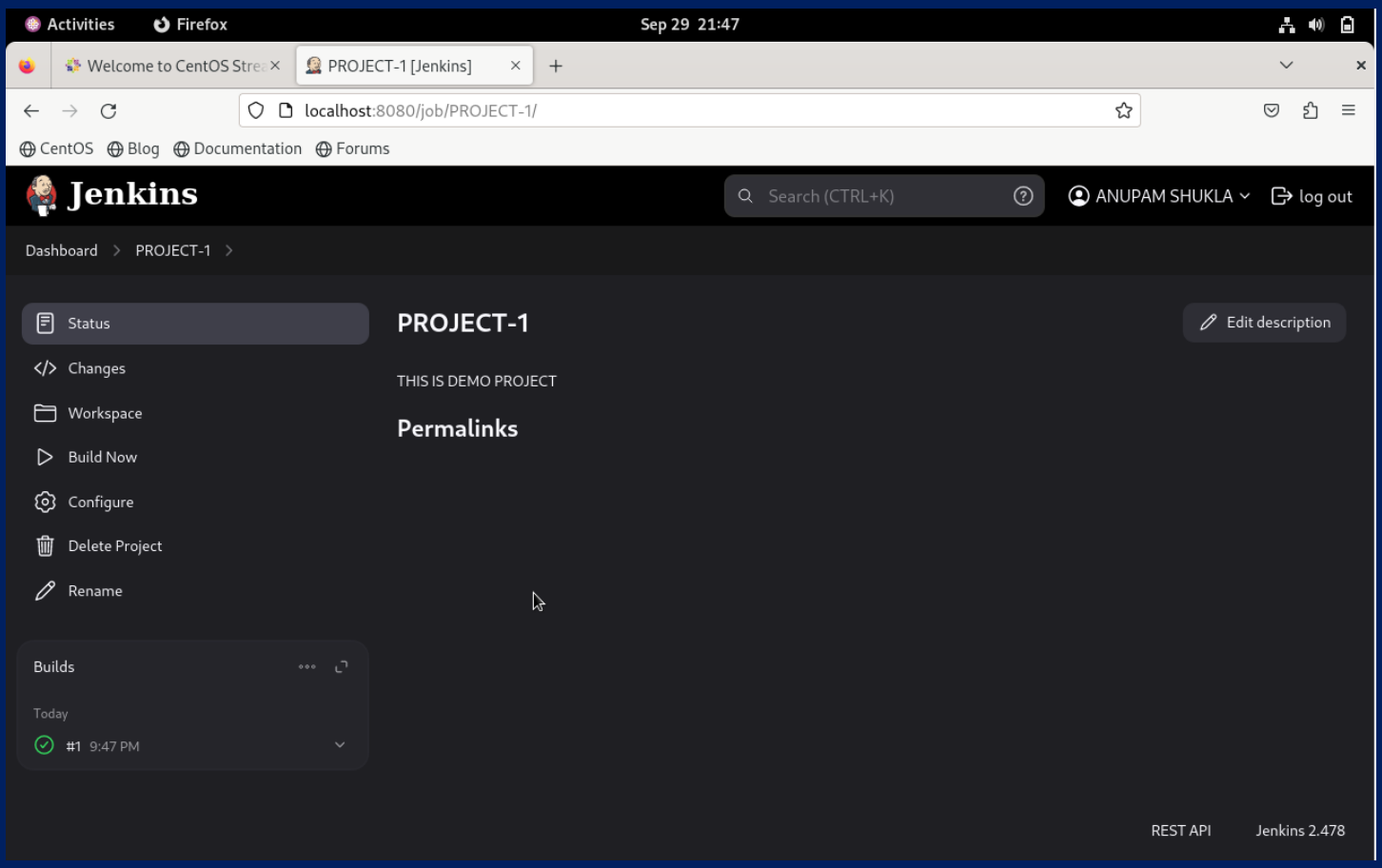

ANUPAM SHUKLA

## STEP-4) Save the Configuration

- After adding the shell command and configuring the build triggers, scroll to the bottom of the page and click **Save**.

## STEP-5) Build the job

- Once the job is configured, you can manually trigger the first build by clicking “Build Now” on the job dashboard.
- Jenkins will run the shell script, printing your name in the console output.




## STEP-6) Build history

- Go to the **Build History** on the job page and click on the latest build number.(Now you will new build will automatically created every minute)
- Click on **“Console Output”** to view the result.
- You should see your name ("anupam") printed in the console.