**CurrentBuild**

n a Jenkins Pipeline, currentBuild is a special variable that provides access to the current build's metadata and status. It's available in **Declarative** and **Scripted Pipelines**, and it's especially useful for logging, notifications, and conditional logic based on build results.

Here’s a **detailed breakdown** of how to use currentBuild in a Jenkins Pipeline:

**🔧 Basic Usage in a Scripted Pipeline**

node {

    echo "Build number: ${currentBuild.number}"

    echo "Build result: ${currentBuild.result}"

    echo "Build URL: ${currentBuild.absoluteUrl}"

    // Set build description

    currentBuild.description = "Triggered by user"

    // Set display name

    currentBuild.displayName = "#${currentBuild.number} - Custom Name"

}

**🧱 Usage in a Declarative Pipeline**

pipeline {

    agent any

    stages {

        stage('Example') {

            steps {

                script {

                    echo "Build number: ${currentBuild.number}"

                    echo "Build result: ${currentBuild.result ?: 'IN PROGRESS'}"

                    currentBuild.description = "Running tests"

                }

            }

        }

    }

    post {

        success {

            script {

                currentBuild.result = 'SUCCESS'

                echo "Build succeeded!"

            }

        }

        failure {

            script {

                currentBuild.result = 'FAILURE'

                echo "Build failed!"

            }

        }

    }

}

**📌 Common currentBuild Properties**

| **Property** | **Description** |
| --- | --- |
| currentBuild.number | The build number (e.g., 42) |
| currentBuild.result | The result of the build (SUCCESS, FAILURE, UNSTABLE, or null if still running) |
| currentBuild.duration | Duration of the build in milliseconds |
| currentBuild.displayName | Custom name shown in the Jenkins UI |
| currentBuild.description | Custom description shown in the Jenkins UI |
| currentBuild.previousBuild | Reference to the previous build |
| currentBuild.absoluteUrl | Full URL to the build |
| currentBuild.rawBuild | Gives access to lower-level APIs (advanced use) |

**Build**

In Jenkins Pipeline, the build step is used to **trigger another job (build)** from within your current pipeline. This is useful for creating **multi-job workflows**, **modular pipelines**, or **triggering downstream jobs**.

**🔧 Syntax of build Step**

build job: 'job-name', parameters: [...], wait: true, propagate: true

**🧱 Detailed Breakdown of Parameters**

| **Parameter** | **Description** |
| --- | --- |
| job | **(Required)** Name of the job to trigger. |
| parameters | List of parameters to pass to the triggered job. |
| wait | If true, waits for the triggered job to finish before continuing. |
| propagate | If true, the current build will fail if the triggered build fails. |

**✅ Example 1: Trigger Another Job Without Parameters**

pipeline {

    agent any

    stages {

        stage('Trigger Job') {

            steps {

                build job: 'My-Other-Job'

            }

        }

    }

}

**✅ Example 2: Trigger Job With Parameters and Wait**

pipeline {

    agent any

    stages {

        stage('Trigger with Params') {

            steps {

                build job: 'Deploy-App',

                      parameters: [

                          string(name: 'ENV', value: 'staging'),

                          booleanParam(name: 'RUN\_TESTS', value: true)

                      ],

                      wait: true,

                      propagate: true

            }

        }

    }

}

**✅ Example 3: Trigger Job in Scripted Pipeline**

node {

    stage('Trigger Downstream') {

        def result = build job: 'Build-Library',

                           parameters: [string(name: 'VERSION', value: '1.2.3')],

                           wait: true,

                           propagate: false

        echo "Triggered build result: ${result.result}"

    }

}

**📌 Notes**

* The build step **requires the downstream job to be configured to accept parameters** if you're passing any.
* You can use propagate: false to **continue the pipeline even if the triggered job fails**.
* The build step returns a RunWrapper object, which gives access to the result, URL, and more.

**catchError**

In Jenkins Pipeline, catchError is a very useful step that allows you to **catch errors in a block of code** without failing the entire build. It’s commonly used to **handle failures gracefully**, especially when you want to continue the pipeline even if a step fails.

**🔧 Syntax**

catchError(buildResult: 'FAILURE', stageResult: 'FAILURE') {

    // Code that might fail

}

**🧱 Parameters**

| **Parameter** | **Description** |
| --- | --- |
| buildResult | The result to set for the overall build if an error is caught (e.g., 'SUCCESS', 'UNSTABLE', 'FAILURE'). |
| stageResult | The result to set for the stage if an error is caught. |
| message | Optional message to show in the logs when an error is caught. |

**✅ Example 1: Basic Usage**

pipeline {

    agent any

    stages {

        stage('Test') {

            steps {

                catchError {

                    sh 'exit 1' // This would normally fail the build

                }

                echo 'This will still run even if the shell command fails.'

            }

        }

    }

}

**✅ Example 2: Custom Build and Stage Result**

pipeline {

    agent any

    stages {

        stage('Optional Step') {

            steps {

                catchError(buildResult: 'UNSTABLE', stageResult: 'FAILURE') {

                    sh 'exit 1'

                }

                echo 'Pipeline continues, but build is marked UNSTABLE.'

            }

        }

    }

}

**✅ Example 3: With Scripted Pipeline**

node {

    stage('Try Something Risky') {

        catchError(buildResult: 'SUCCESS', stageResult: 'FAILURE') {

            error("Something went wrong")

        }

        echo "Continuing even after error"

    }

}

**📌 When to Use catchError**

* To **prevent a stage from failing the entire pipeline**.
* To **mark builds as unstable** instead of failed.
* To **log errors and continue** with cleanup or reporting steps.
* In **post-build actions** like sending notifications regardless of success/failure.

**error 'message'**

In Jenkins Pipeline, the error 'message' step is used to **intentionally fail the build or stage** with a custom error message. It's a way to stop execution and mark the build or stage as failed.

**🔧 Syntax**

error 'Something went wrong'

This will:

* Immediately stop the current stage or script block.
* Mark the stage and build as **FAILED** (unless caught with catchError or try/catch).
* Print the message in the **console output**.

**✅ Example in a Scripted Pipeline**

node {

    stage('Validation') {

        def isValid = false

        if (!isValid) {

            error 'Validation failed: Required files are missing.'

        }

    }

}

**✅ Example in a Declarative Pipeline**

pipeline {

    agent any

    stages {

        stage('Check Condition') {

            steps {

                script {

                    def condition = false

                    if (!condition) {

                        error 'Condition not met. Aborting build.'

                    }

                }

            }

        }

    }

}

**✅ Example with catchError**

node {

    stage('Try Something Risky') {

        catchError(buildResult: 'SUCCESS', stageResult: 'FAILURE') {

            error 'Simulated failure for testing'

        }

        echo 'Pipeline continues even after error'

    }

}

**📌 Use Cases**

* **Fail fast** when a critical condition isn't met.
* **Abort builds** based on validation or test results.
* **Simulate failures** for testing pipeline behavior.
* **Custom error messages** for better debugging.

**sleep time: 10, unit: 'SECONDS'**

In Jenkins Pipeline, the sleep step is used to **pause the pipeline execution** for a specified amount of time. This can be useful for:

* Waiting for a service to start
* Introducing a delay between retries
* Throttling requests
* Debugging timing issues

**🔧 Syntax**

sleep time: 10, unit: 'SECONDS'

**🧾 Explanation**

* **time: 10** – This sets the duration of the sleep to 10 units.
* **unit: 'SECONDS'** – This specifies the unit of time. Valid values include:
  + 'SECONDS'
  + 'MINUTES'
  + 'HOURS'
  + 'DAYS'

**✅ Example in a Declarative Pipeline**

pipeline {

    agent any

    stages {

        stage('Wait a Bit') {

            steps {

                echo 'Waiting for 10 seconds...'

                sleep time: 10, unit: 'SECONDS'

                echo 'Done waiting!'

            }

        }

    }

}

**✅ Example in a Scripted Pipeline**

node {

    stage('Pause') {

        echo 'Pausing for 10 seconds...'

        sleep time: 10, unit: 'SECONDS'

        echo 'Resuming pipeline.'

    }

}