# Jenkins Global Variable Reference: Env, Params, and CurrentBuild

## 1. Environment Variables (env)

**Overview**

Jenkins provides access to environmental variables via the env global variable. These include system environment variables and Jenkins-specific variables.

**Usage**

Environment variables allow access to system paths, workspace directories, job metadata, and other relevant information.

**Common env Variables**

| **Variable** | **Description** |
| --- | --- |
| env.JENKINS\_HOME | Jenkins home directory path. |
| env.WORKSPACE | Path to the workspace directory. |
| env.BUILD\_URL | URL of the current build. |
| env.GIT\_BRANCH | Name of the branch being built. |
| env.NODE\_NAME | Name of the agent executing the pipeline. |

**Example Usage**

pipeline {

agent any

stages {

stage('Show Environment Variables') {

steps {

script {

echo "Workspace Path: ${env.WORKSPACE}"

echo "Jenkins Home: ${env.JENKINS\_HOME}"

echo "Current Build URL: ${env.BUILD\_URL}"

}

}

}

}

}

## 2. Pipeline Parameters (params)

**Overview**

The params global variable is used to access parameters defined in **Parameterized Builds**.

**Usage**

This is useful for defining user-input values before executing a pipeline.

**Example: Defining Parameters**

pipeline {

agent any

parameters {

string(name: 'BRANCH\_NAME', defaultValue: 'main', description: 'Branch to build')

booleanParam(name: 'DEPLOY', defaultValue: false, description: 'Deploy after build?')

}

stages {

stage('Print Parameters') {

steps {

script {

echo "Building branch: ${params.BRANCH\_NAME}"

echo "Deployment enabled: ${params.DEPLOY}"

}

}

}

}

}

**Common Use Cases**

| **Parameter Type** | **Description** |
| --- | --- |
| string | Accepts text input for builds. |
| booleanParam | Enables toggling features in builds. |
| choice | Allows users to select from predefined options. |

## 3. Current Build (currentBuild)

**Overview**

The currentBuild variable provides information about the current pipeline execution.

**Common currentBuild Properties**

| **Property** | **Description** |
| --- | --- |
| currentBuild.number | Gets the current build number. |
| currentBuild.result | Returns the build result (SUCCESS, FAILURE, etc.). |
| currentBuild.duration | Gets the build duration in milliseconds. |
| currentBuild.displayName | Returns the display name of the build. |

**Example Usage**

pipeline {

agent any

stages {

stage('Check Build Status') {

steps {

script {

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

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

}

}

}

}

post {

always {

echo "Build completed with status: ${currentBuild.result}"

}

}

}

**Best Practices**

* Use currentBuild.result inside post blocks for conditional post-build actions.
* Combine currentBuild.duration with alerts to monitor build performance.

**Conclusion**

Jenkins' global variables **env**, **params**, and **currentBuild** are essential tools for managing CI/CD pipelines effectively. They provide easy access to **system environment data, user-defined parameters, and build status details**, ensuring smooth and customizable workflows.

# How to get the list of failure stages and error descriptions from current Jenkins build using the global variable currentBuild?

**Reference link :**

[**https://stackoverflow.com/questions/64418907/jenkins-list-stages-for-each-parallel-branch**](https://stackoverflow.com/questions/64418907/jenkins-list-stages-for-each-parallel-branch)

[**https://javadoc.jenkins.io/plugin/workflow-support/org/jenkinsci/plugins/workflow/support/steps/build/RunWrapper.html#%3Cinit%3E(hudson.model.Run,boolean)**](https://javadoc.jenkins.io/plugin/workflow-support/org/jenkinsci/plugins/workflow/support/steps/build/RunWrapper.html#%3Cinit%3E(hudson.model.Run,boolean))

[**https://javadoc.jenkins.io/plugin/blueocean-pipeline-api-impl/io/jenkins/blueocean/rest/impl/pipeline/PipelineNodeGraphVisitor.html**](https://javadoc.jenkins.io/plugin/blueocean-pipeline-api-impl/io/jenkins/blueocean/rest/impl/pipeline/PipelineNodeGraphVisitor.html)

## Extracting Failed Stages and Error Descriptions in Jenkins Pipeline

**Overview**

When running a Jenkins pipeline, it is often essential to track which stages have failed and retrieve their corresponding error messages. Jenkins provides multiple ways to achieve this using global variables like currentBuild and classes such as RunWrapper, ErrorAction, and PipelineNodeGraphVisitor.

**Key Components**

* currentBuild: Retrieves details about the ongoing Jenkins build.
* RunWrapper: Helps interact with Jenkins build information.
* ErrorAction: Identifies errors associated with failed pipeline stages.
* PipelineNodeGraphVisitor: Used to traverse and analyze pipeline execution nodes.

**Jenkins Pipeline Script**

Below is a scripted pipeline example that extracts failed stages and their error messages using PipelineNodeGraphVisitor:

import io.jenkins.blueocean.rest.impl.pipeline.PipelineNodeGraphVisitor

import io.jenkins.blueocean.rest.impl.pipeline.FlowNodeWrapper

import org.jenkinsci.plugins.workflow.support.steps.build.RunWrapper

import org.jenkinsci.plugins.workflow.actions.ErrorAction

pipeline {

agent any

stages {

stage('Initialize') {

steps {

script {

echo 'Initializing Pipeline'

}

}

}

stage('Failing Stage') {

steps {

script {

error 'Intentional failure in this stage'

}

}

}

stage('Final Stage') {

steps {

script {

echo 'Executing Final Stage'

}

}

}

}

post {

always {

script {

def run = currentBuild.rawBuild

def visitor = new PipelineNodeGraphVisitor(run)

def failedStages = []

def stages = visitor.pipelineNodes.findAll {

it.type == FlowNodeWrapper.NodeType.STAGE || it.type == FlowNodeWrapper.NodeType.PARALLEL

}

stages.each { stage ->

def errorActions = stage.getPipelineActions(ErrorAction)

if (errorActions) {

failedStages.add([stage: stage.displayName, error: errorActions.first().error.message])

}

}

if (failedStages) {

echo "Failed Stages Summary:"

failedStages.each { stageInfo ->

echo "Stage: ${stageInfo.stage}, Error: ${stageInfo.error}"

}

} else {

echo "No stages failed."

}

}

}

}

}

**Explanation of the Script**

1. Pipeline Execution:
   * Defines three stages: Initialize, Failing Stage, and Final Stage.
   * The Failing Stage contains an intentional error.
2. Retrieving Failed Stages:
   * Uses PipelineNodeGraphVisitor to traverse pipeline execution nodes.
   * Filters nodes to extract STAGE and PARALLEL types.
   * Fetches ErrorAction instances from each stage to identify failures.
3. Printing Error Summary:
   * Iterates over failed stages and prints the stage name along with its error message.

**Best Practices**

* Error Handling: Always implement error handling mechanisms to capture failures.
* Logging: Use echo statements to log error details for easier debugging.
* Parallel Processing: Ensure that failures in parallel stages are also captured.

## Expected Output (If a Stage Fails)

**Failed Stages Summary:**

Stage: Failing Stage, Error: Intentional failure in this stage

This method provides a robust way to track failed stages in a Jenkins pipeline, aiding in debugging and improving reliability.

## Details of RunWrapper

The org.jenkinsci.plugins.workflow.support.steps.build.RunWrapper class is part of Jenkins' pipeline plugin and is used to wrap a build run in the context of a pipeline. It is commonly utilized when you want to interact with another build (such as triggering a downstream build or accessing its results) from within a Jenkins pipeline.

Key Features of RunWrapper:

* Encapsulates a Build: The RunWrapper class wraps around a Jenkins build, giving you access to the build's metadata, artifacts, and other associated information. This class is particularly useful when you need to handle another job’s build in your pipeline, such as waiting for a downstream build to complete or retrieving build-related information.
* Access to Build Information: With RunWrapper, you can get details like the build status (success, failure, unstable), build parameters, artifacts, and logs. This helps in making decisions based on the results of a previous build in your pipeline.
* Downstream Build Control: It is used to control or influence downstream builds. For example, you may use it to trigger downstream builds, pass parameters to those builds, or evaluate the outcome of a downstream job.

Example Use Case:

Let's say you have a Jenkins pipeline where one job triggers another job, and you want to monitor the status or collect artifacts from that triggered job. RunWrapper would be helpful in this case. It allows you to manage the downstream job’s execution within your pipeline.

Here is an example of how it might be used:

groovy

Copy code

pipeline {

agent any

stages {

stage('Trigger Downstream Job') {

steps {

script {

// Trigger a downstream job and get the RunWrapper for that job

def downstreamJob = build job: 'downstream-job', wait: true

// Wrap the downstream build using RunWrapper

def wrappedRun = new org.jenkinsci.plugins.workflow.support.steps.build.RunWrapper(downstreamJob)

// Accessing the status of the downstream build

if (wrappedRun.getResult().toString() == 'SUCCESS') {

echo "Downstream build succeeded!"

} else {

echo "Downstream build failed!"

}

}

}

}

}

}

Common Methods in RunWrapper:

* getResult(): This returns the result of the wrapped build (e.g., SUCCESS, FAILURE, ABORTED).
* getBuild(): Access the actual Jenkins Run object (i.e., the build run) that the RunWrapper wraps.
* getArtifacts(): Fetch the artifacts associated with the build.
* getDuration(): Get the duration of the wrapped build run.

Best Practices:

* Wait for Completion: When triggering downstream jobs, it’s important to wait for them to finish using the wait: true option to ensure proper synchronization in your pipeline.
* Error Handling: Always check the result of the wrapped run to handle failures or unexpected outcomes effectively.

Summary:

RunWrapper is a useful class for managing interactions with current Jenkins builds, making it easier to retrieve build information, trigger downstream jobs, and control the execution flow based on the results of previous builds in a Jenkins pipeline.