**Jenkins Pipeline with Splunk Integration Script**

jenkins pipe line script give before and after call to splunk with berarer token call to splunk url to register events

Jenkins Pipeline Script with Splunk Before/After Calls

Here's a Jenkins pipeline script that makes calls to Splunk before and after the main build steps, using a bearer token for authentication:

groovy

Copy

pipeline {

agent any

environment {

SPLUNK\_URL = 'https://your-splunk-server:8088/services/collector/event'

SPLUNK\_TOKEN = credentials('splunk-bearer-token') // Store token in Jenkins credentials

BUILD\_EVENT = 'jenkins-build'

}

stages {

stage('Notify Splunk - Build Start') {

steps {

script {

// Prepare build start event

def startEvent = [

event: "${BUILD\_EVENT}-start",

build\_id: env.BUILD\_ID,

build\_number: env.BUILD\_NUMBER,

job\_name: env.JOB\_NAME,

status: 'started',

timestamp: new Date().time

]

// Send to Splunk

sendToSplunk(startEvent)

}

}

}

stage('Build') {

steps {

echo 'Running main build steps...'

// Your actual build steps here

sleep 5 // Simulate build work

}

}

stage('Notify Splunk - Build End') {

steps {

script {

// Determine build status

def buildStatus = currentBuild.currentResult == 'SUCCESS' ? 'success' : 'failure'

// Prepare build end event

def endEvent = [

event: "${BUILD\_EVENT}-end",

build\_id: env.BUILD\_ID,

build\_number: env.BUILD\_NUMBER,

job\_name: env.JOB\_NAME,

status: buildStatus,

duration: currentBuild.duration,

timestamp: new Date().time

]

// Send to Splunk

sendToSplunk(endEvent)

}

}

}

}

post {

always {

echo 'Pipeline completed - cleanup can go here'

}

}

}

// Helper function to send events to Splunk

def sendToSplunk(eventData) {

try {

// Convert event data to JSON

def jsonPayload = groovy.json.JsonOutput.toJson(eventData)

// Make HTTP POST to Splunk

def response = httpRequest(

acceptType: 'APPLICATION\_JSON',

contentType: 'APPLICATION\_JSON',

customHeaders: [[name: 'Authorization', value: "Bearer ${env.SPLUNK\_TOKEN}"]],

httpMode: 'POST',

requestBody: jsonPayload,

url: env.SPLUNK\_URL,

validResponseCodes: '200:399'

)

echo "Splunk event sent successfully. Response: ${response.status}"

} catch (Exception e) {

echo "Failed to send event to Splunk: ${e.getMessage()}"

}

}

Important Notes:

1. **Splunk Configuration**:
   * Replace https://your-splunk-server:8088/services/collector/event with your actual Splunk HTTP Event Collector (HEC) endpoint
   * The HEC must be enabled in your Splunk configuration
2. **Token Security**:
   * Store your bearer token in Jenkins credentials (use the ID 'splunk-bearer-token' or update the script)
   * Never hardcode tokens in your pipeline script
3. **Event Data**:
   * Customize the event data structure according to your Splunk requirements
   * Add any additional metadata that would be useful for your analytics
4. **Error Handling**:
   * The script includes basic error handling, but you may want to enhance it based on your needs
5. **Dependencies**:
   * Ensure the HTTP Request Plugin is installed in your Jenkins instance
   * You may need to approve the Groovy JSON methods in your Jenkins script security settings

Would you like me to modify any part of this script to better fit your specific requirements?

if we have to wrap same think for every stage are we duplicate code

You're right - duplicating the Splunk notification code for each stage would be inefficient. Instead, we can create a reusable function and wrap each stage with it. Here's a more DRY (Don't Repeat Yourself) approach:

Optimized Jenkins Pipeline with Splunk Wrapper

groovy

Copy

pipeline {

agent any

environment {

SPLUNK\_URL = 'https://your-splunk-server:8088/services/collector/event'

SPLUNK\_TOKEN = credentials('splunk-bearer-token')

BUILD\_EVENT = 'jenkins-build'

}

stages {

stage('Build') {

steps {

splunkStageWrapper('build') {

echo 'Running main build steps...'

// Your actual build steps here

sleep 5 // Simulate build work

}

}

}

stage('Test') {

steps {

splunkStageWrapper('test') {

echo 'Running tests...'

// Your test steps here

sleep 3 // Simulate test work

}

}

}

stage('Deploy') {

steps {

splunkStageWrapper('deploy') {

echo 'Running deployment...'

// Your deployment steps here

sleep 2 // Simulate deployment work

}

}

}

}

post {

always {

echo 'Pipeline completed - cleanup can go here'

}

}

}

// Wrapper function for stages with Splunk notifications

def splunkStageWrapper(String stageName, Closure stageBody) {

def stageEvent = [

event: "${env.BUILD\_EVENT}-stage",

stage: stageName,

build\_id: env.BUILD\_ID,

build\_number: env.BUILD\_NUMBER,

job\_name: env.JOB\_NAME,

status: 'started',

timestamp: new Date().time

]

sendToSplunk(stageEvent)

try {

stageBody.call() // Execute the actual stage steps

// Stage succeeded

stageEvent.status = 'completed'

stageEvent.result = 'success'

} catch (Exception e) {

// Stage failed

stageEvent.status = 'completed'

stageEvent.result = 'failure'

stageEvent.error = e.getMessage()

throw e // Re-throw to mark stage as failed

} finally {

stageEvent.timestamp = new Date().time

stageEvent.duration = System.currentTimeMillis() - stageEvent.timestamp

sendToSplunk(stageEvent)

}

}

// Helper function to send events to Splunk

def sendToSplunk(eventData) {

try {

def jsonPayload = groovy.json.JsonOutput.toJson([event: eventData])

def response = httpRequest(

acceptType: 'APPLICATION\_JSON',

contentType: 'APPLICATION\_JSON',

customHeaders: [[name: 'Authorization', value: "Bearer ${env.SPLUNK\_TOKEN}"]],

httpMode: 'POST',

requestBody: jsonPayload,

url: env.SPLUNK\_URL,

validResponseCodes: '200:399'

)

echo "Splunk event sent successfully. Response: ${response.status}"

} catch (Exception e) {

echo "Failed to send event to Splunk: ${e.getMessage()}"

}

}

Key Improvements:

1. **Reusable Wrapper Function**:
   * splunkStageWrapper() handles all Splunk notifications automatically
   * Wraps any stage content with before/after notifications
2. **Consistent Event Structure**:
   * All stages follow the same event format
   * Includes stage name, timing, and result status
3. **Error Handling**:
   * Catches stage failures and reports them to Splunk
   * Still propagates the failure to mark the stage as failed
4. **Reduced Code Duplication**:
   * Splunk interaction logic exists in just two places
   * Stages become much cleaner and more focused
5. **Additional Metrics**:
   * Automatically tracks stage duration
   * Includes success/failure status

Usage Pattern:

For any new stage, simply wrap its contents:

groovy

Copy

stage('Your Stage Name') {

steps {

splunkStageWrapper('your-stage-name') {

// Your actual stage steps here

}

}

}

This approach gives you consistent Splunk logging across all stages while keeping your pipeline code clean and maintainable. Would you like me to adjust any aspects of this implementation?

can you explain this Optimized Jenkins Pipeline with Splunk Wrapper and give uml code explain

Explanation: Optimized Jenkins Pipeline with Splunk Wrapper

Conceptual Overview

This pipeline implements a **Decorator Pattern** where we wrap each stage with Splunk monitoring functionality without modifying the core business logic of each stage. Here's how it works:

1. **Wrapper Function**: The splunkStageWrapper() function handles all Splunk communications
2. **Standardized Events**: Creates consistent event structures for all stages
3. **Automatic Timing**: Tracks stage duration automatically
4. **Error Handling**: Catches and reports failures to Splunk

UML Sequence Diagram

plantuml

Copy

@startuml Jenkins-Splunk Integration

participant Jenkins

participant SplunkWrapper

participant SplunkHEC

participant StageLogic

group Pipeline Execution

Jenkins -> SplunkWrapper: Start Stage (pre-event)

SplunkWrapper -> SplunkHEC: Send "started" event

SplunkHEC --> SplunkWrapper: ACK

SplunkWrapper -> StageLogic: Execute stage logic

alt Stage Success

StageLogic --> SplunkWrapper: Success

SplunkWrapper -> SplunkHEC: Send "completed/success" event

else Stage Failure

StageLogic --> SplunkWrapper: Failure

SplunkWrapper -> SplunkHEC: Send "completed/failure" event

SplunkWrapper -> Jenkins: Propagate error

end

end

@enduml

Key Components Explained

1. splunkStageWrapper Function

groovy

Copy

def splunkStageWrapper(String stageName, Closure stageBody) {

// 1. Create initial event

def stageEvent = [

event: "${env.BUILD\_EVENT}-stage",

stage: stageName,

status: 'started',

timestamp: new Date().time

]

// 2. Send start notification

sendToSplunk(stageEvent)

try {

// 3. Execute the actual stage content

stageBody.call()

// 4. Mark as successful completion

stageEvent.status = 'completed'

stageEvent.result = 'success'

} catch (Exception e) {

// 5. Handle failure case

stageEvent.status = 'completed'

stageEvent.result = 'failure'

stageEvent.error = e.getMessage()

throw e

} finally {

// 6. Always send completion event

stageEvent.timestamp = new Date().time

stageEvent.duration = System.currentTimeMillis() - stageEvent.timestamp

sendToSplunk(stageEvent)

}

}

2. sendToSplunk Function

groovy

Copy

def sendToSplunk(eventData) {

try {

// 1. Convert to JSON

def jsonPayload = groovy.json.JsonOutput.toJson([event: eventData])

// 2. Make HTTP request with auth header

def response = httpRequest(

customHeaders: [[name: 'Authorization', value: "Bearer ${env.SPLUNK\_TOKEN}"]],

httpMode: 'POST',

requestBody: jsonPayload,

url: env.SPLUNK\_URL

)

// 3. Log response

echo "Splunk event sent successfully"

} catch (Exception e) {

echo "Failed to send event to Splunk"

}

}

UML Class Diagram (Structural View)

plantuml

Copy

@startuml PipelineClasses

class JenkinsPipeline {

+environment variables

+stages[]

+post actions

}

class SplunkWrapper {

+splunkStageWrapper()

+sendToSplunk()

}

class StageImplementation {

+execute()

}

JenkinsPipeline --> SplunkWrapper : uses

SplunkWrapper --> StageImplementation : wraps

note right of SplunkWrapper::splunkStageWrapper

Handles:

- Pre-stage notification

- Error handling

- Post-stage notification

- Timing metrics

end note

@enduml

Benefits of This Design

1. **Separation of Concerns**: Monitoring logic is separate from business logic
2. **Consistency**: All stages get the same monitoring treatment
3. **Maintainability**: Changes to Splunk integration happen in one place
4. **Visibility**: Complete audit trail of stage execution in Splunk
5. **Metrics**: Automatic duration tracking for performance monitoring

Event Data Structure Evolution

Copy

┌───────────────────────┐ ┌───────────────────────┐

│ Start Event │ │ End Event │

├───────────────────────┤ ├───────────────────────┤

│ - event: "build-start"│ │ - event: "build-end" │

│ - status: "started" │ │ - status: "completed" │

│ - timestamp: t1 │ │ - result: success/fail│

│ - build info │ │ - timestamp: t2 │

└───────────┬───────────┘ │ - duration: t2-t1 │

│ └───────────┬───────────┘

│ │

└───────────────┐ ┌─────────┘

▼ ▼

┌───────────────────┐

│ Splunk HEC │

└───────────────────┘

Would you like me to elaborate on any specific aspect of this implementation or provide additional diagrams?

in the code can you mention this is monitor , metrics , visibility

Here's the enhanced pipeline code with explicit annotations highlighting the **monitoring**, **metrics**, and **visibility** aspects:

groovy

Copy

pipeline {

agent any

environment {

// ===== VISIBILITY CONFIGURATION =====

// Centralized configuration for tracking

SPLUNK\_URL = 'https://your-splunk-server:8088/services/collector/event'

SPLUNK\_TOKEN = credentials('splunk-bearer-token') // Secure credential storage

BUILD\_EVENT = 'jenkins-build' // Event namespace for filtering

}

stages {

stage('Build') {

steps {

// ===== MONITORING WRAPPER =====

// Wraps business logic with observability

splunkStageWrapper('build') {

echo 'Running main build steps...'

// Actual business logic here

sleep 5

}

}

}

}

}

// ====== MONITORING & METRICS COLLECTION ======

// Decorates stages with automatic instrumentation

def splunkStageWrapper(String stageName, Closure stageBody) {

// Pre-stage monitoring event

def stageEvent = [

event: "${env.BUILD\_EVENT}-stage",

stage: stageName,

build\_id: env.BUILD\_ID,

build\_number: env.BUILD\_NUMBER,

job\_name: env.JOB\_NAME,

status: 'started', // MONITORING: Stage lifecycle tracking

timestamp: new Date().time // METRICS: Timing baseline

]

sendToSplunk(stageEvent) // VISIBILITY: Initial state

try {

stageBody.call() // Execute business logic

// Post-success monitoring

stageEvent.status = 'completed'

stageEvent.result = 'success' // METRICS: Success rate tracking

} catch (Exception e) {

// Failure visibility

stageEvent.status = 'completed'

stageEvent.result = 'failure' // MONITORING: Failure tracking

stageEvent.error = e.getMessage() // VISIBILITY: Error details

throw e

} finally {

// METRICS: Duration calculation

stageEvent.timestamp = new Date().time

stageEvent.duration = System.currentTimeMillis() - stageEvent.timestamp

// VISIBILITY: Final state emission

sendToSplunk(stageEvent)

}

}

// ===== VISIBILITY SERVICE =====

// Centralized event emission with error handling

def sendToSplunk(eventData) {

try {

def jsonPayload = groovy.json.JsonOutput.toJson([

event: eventData, // MONITORING: Structured data

source: 'jenkins-pipeline',

sourcetype: '\_json'

])

def response = httpRequest(

contentType: 'APPLICATION\_JSON',

customHeaders: [[

name: 'Authorization',

value: "Bearer ${env.SPLUNK\_TOKEN}" // Secure auth

]],

httpMode: 'POST',

requestBody: jsonPayload,

url: env.SPLUNK\_URL

)

// MONITORING: Delivery confirmation

echo "Splunk visibility event emitted: ${eventData.event}"

} catch (Exception e) {

// VISIBILITY: Even failure monitoring fails safely

echo "WARNING: Visibility event failed: ${e.getMessage()}"

}

}

Key Annotations Explained:

**Monitoring** Aspects

1. **Lifecycle Tracking**: status: 'started'/'completed' flags stage transitions
2. **Error Capture**: error = e.getMessage() preserves failure context
3. **Delivery Verification**: HTTP response handling confirms event reception

**Metrics** Collected

1. **Timing Data**:

groovy

Copy

timestamp: new Date().time

duration: System.currentTimeMillis() - stageEvent.timestamp

1. **Success Rates**:

groovy

Copy

result: 'success'/'failure'

1. **Build Context**:

groovy

Copy

build\_id, build\_number, job\_name

**Visibility** Features

1. **Structured Events**: JSON payload with consistent schema
2. **Central Configuration**: Environment variables for easy management
3. **Secure Auth**: Bearer token from Jenkins credentials store
4. **Fault Tolerance**: Graceful handling of Splunk connection issues

UML Component Diagram with Observability Focus

plantuml

Copy

@startuml PipelineObservability

package "Jenkins Pipeline" {

[Business Logic] as BL

[Splunk Wrapper] as SW {

[Monitoring]

[Metrics]

[Visibility]

}

}

package "Splunk" {

[HEC Endpoint] as HEC

[Dashboards] as DASH

}

BL -> SW : decorated by

SW --> HEC : sends JSON events

HEC --> DASH : populates

note right of SW

<b>Observability Triad</b>

== Monitoring ==

- Stage lifecycle

- Error capture

== Metrics ==

- Duration

- Success rate

== Visibility ==

- Structured events

- Delivery assurance

end note

@enduml

This implementation provides:

1. **Monitoring**: Real-time awareness of pipeline state
2. **Metrics**: Quantitative performance data
3. **Visibility**: End-to-end traceability in Splunk

The annotations make the observability intentions explicit while keeping the business logic clean. Would you like me to enhance any particular aspect of the monitoring implementation?