Jenkins Shared Library

Jenkins Shared Library is the way of having a common pipeline code in the version control system which is git that can be called by any number of pipeline jobs just by referring to it in the pipeline code using @Library("<library_name>")_

Steps to setup Jenkins Shared Library

1. Create a git repo for shared library

    - Create a folder by name vars

2. Create .groovy file inside vars folder and put shared library code here
   - Make sure to remember and provide a meaningful name for groovy
   - Groovy filename is used as the function call

gitCheckout.groovy

```
def call(Map config) {
    echo "URL: ${config.url}"
    echo "BRANCH: ${config.branch}"
        checkout([$class: 'GitSCM',
                    branches: [[name: config.branch]],
                    doGenerateSubmoduleConfigurations: false,
                    extensions: [],
                    userRemoteConfigs: [[url: config.url,
                    credentialsId: config.credentialsId]]])
}
```

3. **Integrated it with Jenkins**
   - Manage Jenkins > System > Global Pipeline Libraries
        Name - Can give any name, we can call it in any job using this @Library("<name>")_
        Default Version - git branch in which we keep our shared library
        Retrieval method - Modren SCM
        Source Code Management - Git

4. To call a shared library function / Groovy script
   - use the name of the script in the pipeline job (Jenkinsfile)

example:

```
gitCheckout('main',
            'itd_jenkins',
            'https://github.com/<repo_url>')
```

CICD - Prerequisites

Jenkins Machine:

1. Software requirements (Installations in jenkins machines):(All install scripts are in materials drive - Installation_scripts)

- Install Docker in jenkins server

```
sudo bash docker_instal.sh

#Provide jenkins user access to docker
add jenkins user to docker gorup
sudo usermod -aG docker jenkins
sudo systemctl restart jenkins
sudo systemctl restart docker
```

- Install trivy in jenkins server

```
sudo bash trivy_install.sh
```

- Install kubectl in jenkins server

```
curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.32/deb/Release.key |
sudo gpg --dearmor -o /etc/apt/keyrings/kubernetes-apt-keyring.gpg
echo 'deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg]
https://pkgs.k8s.io/core:/stable:/v1.32/deb/ /' | sudo tee
/etc/apt/sources.list.d/kubernetes.list
sudo apt update -y
sudo apt install -y kubectl
```

- Install argoCD cli in jenkins server

```
# Download latest Argo CD CLI release
VERSION=$(curl -s
https://api.github.com/repos/argoproj/argo-cd/releases/latest | grep
tag_name | cut -d '"' -f 4)
curl -sSL -o argocd-linux-amd64
https://github.com/argoproj/argo-cd/releases/download/$VERSION/argocd-l
inux-amd64

# Make it executable
chmod +x argocd-linux-amd64
```

```
# Move to a directory in your PATH
sudo mv argocd-linux-amd64 /usr/local/bin/argocd
```

- Install sonar-scanner in jenkins server

```
sudo bash sonar_scanner.sh
```

- make Jenkins user sudo user

```
sudo su -
usermod -aG sudo jenkins
echo 'jenkins ALL=(ALL) NOPASSWD:ALL' | sudo tee /etc/sudoers.d/jenkins
sudo systemctl restart jenkins
```

Jenkins settings

Jenkins plugins needed

```
HTML Publisher
Docker Pipeline
```

1.
2. Jenkins Credentials

Provide meaning credentials ID and use the same IDs in pipeline

- github - Kind (Username with password): <username>: github_username, <password>: github_token
- **docker - Kind (Username with password): <username>: dockerhub_username, <password>: hub_token**
- k8s-cluster - file(config): "$HOME/.kube/config" file with context, user, cluster configured
- sonarqube - Kind (Secret text): Sonarqube token
- **argocd - Kind (Username with password): <username>: admin, <password>: get it from secret in cluster**
  (if you want you can add user to argo cd and you can use it)

Install and Configure SonarQube Server:

- Download and install SonarQube Server on a server or machine accessible to your Jenkins instance.
- Configure SonarQube server, including setting up projects and quality profiles.curl -s https://raw.githubusercontent.com/jaintpharsha/install/main/sonarqube-docker | sudo bash

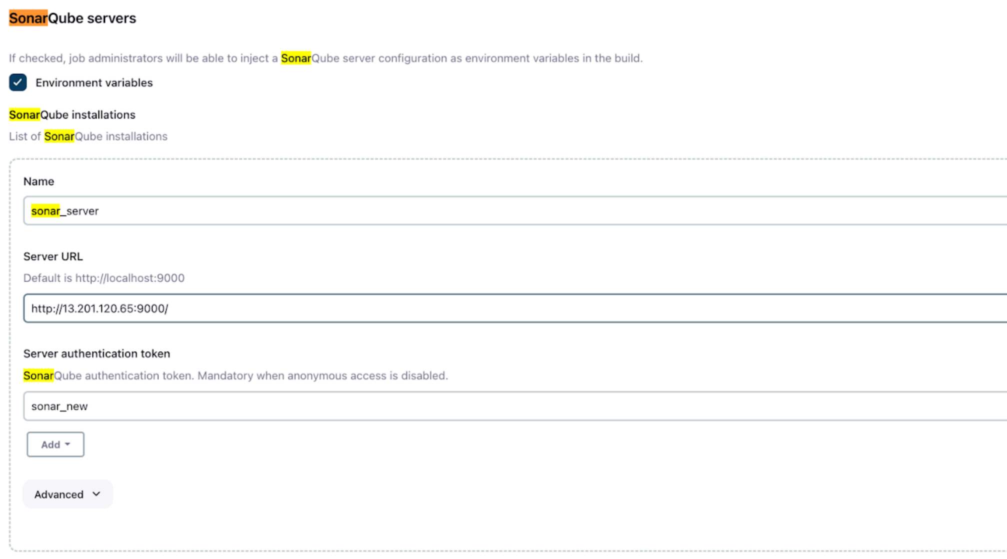
Install SonarQube Scanner:

- Download and install the SonarQube Scanner on the machine where Jenkins is installed.
- Configure the SonarQube Scanner properties, such as SonarQube server URL and authentication token. You can find the authentication token in SonarQube under My Account > Security.

Download Link:
https://docs.sonarsource.com/sonarqube/latest/analyzing-source-code/scanners/sonarscanner/

curl -s https://raw.githubusercontent.com/jaintpharsha/install/main/sonarscanner | sudo bash

1. Configure SonarQube in Jenkins using Sonarqube scanner plugin:
    - In Jenkins, go to Manage Jenkins > Global Tool Configuration.
    - Find the SonarQube Scanner section and add a new SonarQube Scanner installation. Provide the name and specify the path to the SonarQube Scanner executable.



SonarQube servers

If checked, job administrators will be able to inject a SonarQube server configuration as environment variables in the build.

☑ Environment variables

SonarQube installations
List of SonarQube installations

Name
sonar_server

Server URL
Default is http://localhost:9000
http://13.201.120.65:9000/

Server authentication token
SonarQube authentication token. Mandatory when anonymous access is disabled.
sonar_new

2. Configure Jenkins Pipeline:
    - Create or edit your Jenkins pipeline script (Jenkinsfile) to include SonarQube analysis.
    - Add the SonarQube Scanner step to your pipeline script. Here's a basic example:

Trigger Sonarqube scanning using cli

1. sonar-scanner \
   -Dsonar.projectKey=<prokect_key>\
   -Dsonar.sources=. \

   -Dsonar.host.url=http://<sonarqube_server_ip>:<port
   > \
   -Dsonar.token=<user_token>

Can also use sonar-project.properties

```
sonar.projectKey=backend
sonar.projectName=Backend API
sonar.projectVersion=1.0

# Source configuration
sonar.sources=.
#sonar.exclusions=node_modules/**,coverage/**,**/*.spec.js,**/__tests__
/**

# Language & encoding
sonar.language=js
sonar.sourceEncoding=UTF-8

# Coverage report (if available)
# sonar.javascript.lcov.reportPaths=coverage/lcov.info

# SonarQube server details
```

2. `#sonar.host.url=http://56.11.23.212:9000`

Piepline stage

```
stage('Sonar Code Quality check') {
            steps {
                script {
                    dir('./backend') {
                        withCredentials([string(credentialsId:
'SONAR_TOKEN', variable: 'SONAR_TOKEN')]) {
                            sh """
                                sonar-scanner \
                                  -Dsonar.projectKey=backend \
                                  -Dsonar.sources=. \

-Dsonar.host.url=${env.SONAR_HOST_URL} \
                                  -Dsonar.token=${SONAR_TOKEN}
```

```
                                             """
                                       }
                                 }
                           }
                     }
```

3.
   ```
             }
   ```

SonarQube sends Quality Gate status back to Jenkins via webhook.

- Go to SonarQube → Administration → Configuration → Webhooks
- Add new webhook:
  - Name: Jenkins
  - URL: http://<jenkins-server>:<port>/sonarqube-webhook/

```
stage('Quality Gate') {
                  steps {
                        timeout(time: 5, unit: 'MINUTES') {
                              waitForQualityGate abortPipeline: true
                        }
                  }
```

-
  ```
             }
  ```

ArgoCD, Reverse Proxy - nginx, HTTPS, Route53 hosting

Reverse Proxy

- A reverse proxy is a server that sits between clients (e.g., browsers) and backend infrastructure (Applications), acting as a gateway.
- It accepts client requests, forwards them to the appropriate backend server, and returns the server's response to the client.
- This architecture adds a critical layer of abstraction by enhancing security, scalability, load balancing, SSL termination, and caching while hiding backend infrastructure.

Nginx reverse proxy configuration nginx.cof (HTTP)

```
#/etc/nginx/sites-available/example.com.conf
server {
        # Listen on port 80 (HTTP) for requests to example.com
and www.example.com
        listen 80;

        # Allow uploads/files up to 250MB
        client_max_body_size 250m;

        # Handle requests to the /.well-known directory (e.g.,
SSL certificate issue and renewals)
        # Let's Encrypt validation requires HTTP access (not
HTTPS) to these files for validation.
        # If not, The redirect breaks validation, causing
certificate renewals to fail.
        server_name example.com www.example.com;
        location ^~ /.well-known {
            root /etc/nginx/ssl/bot;
        }

        # Redirect all other HTTP traffic to HTTPS
        location / {
            include proxy_params;   # Include common proxy
headers/settings (optional here)
            return 301 https://$host$request_uri; #Permanent
redirect to HTTPS
        }
}
```

$host - https://example.com/

Without $request_uri:

(browser) http://example.com/login?user=123 → (server)

https://example.com/(redirects to root, losing the path/query).

With $request_uri:

(browser) http://example.com/login?user=123 → (server)

https://example.com/login?user=123 (correct behavior).

Nginx reverse proxy configuration nginx.cof (HTTPS)

```
server {
        # Listen on standard HTTPS port with SSL/TLS encryption
        listen 443 ssl;
        # Domain names to listen for https requests
        server_name example.com www.example.com;
        client_max_body_size 250m;


        location / {
             # Preserve original request headers for backend
applications:
            proxy_set_header Host $http_host;    # Forward original
Host header
            proxy_set_header X-Real-IP $remote_addr;    # Client's real
IP address
            proxy_set_header X-Forwarded-For
$proxy_add_x_forwarded_for; # Client IP chain
            proxy_set_header X-Forwarded-Proto $scheme; # Original
protocol (http/https)

            # Forward all traffic to backend service running on port
8090
            proxy_pass http://127.0.0.1:8090;
        }

        #SSL/TLS Certificate Configuration (Let's Encrypt recommended)
        ssl_certificate
/etc/letsencrypt/live/example.com/fullchain.pem;
        ssl_certificate_key
/etc/letsencrypt/live/example.com/privkey.pem;
}
```

ArgoCD Installation in K8S cluster:

1. Install argocd

```
kubectl create namespace argocd
kubectl apply -n argocd -f
https://raw.githubusercontent.com/argoproj/argo-cd/stab
le/manifests/install.yaml
```

2. To access the ArgoCD Dashboard

```
kubectl patch svc argocd-server -n argocd -p '{"spec":
{"type": "NodePort"}}'
```

3. Login to rgoCD Dashboard

```
username: admin
passord: <run_the_below_command>
kubectl get secrets -n argocd
argocd-initial-admin-secret -o
jsonpath="{.data.password}" | base64 -d
```

Argo CD Controller:

- The Argo CD controller is a Kubernetes controller that continuously monitors the Git repository for changes to the desired state.
- When changes are detected, the controller reconciles the current state of the cluster with the desired state defined in the Git repository.
- It uses a pull-based approach, meaning it pulls the configuration from the Git repository and applies it to the cluster.

Git Repository:

- Argo CD relies on Git repositories to store and version your Kubernetes manifests and application configuration files.
- The Git repository is where you define the desired state of your applications and environments.

Custom Resource Definitions (CRDs):

- Argo CD extends Kubernetes by introducing custom resource definitions (CRDs) to define applications and application sets.
- The Application CRD is used to represent a deployed application and its desired state.

Application Set Controller:

- Argo CD introduced the notion of an "Application Set" for managing multiple similar applications in a more streamlined way.
- The AppSet controller watches for changes in Application Sets and manages the creation and deletion of individual applications.

User Interface (Web UI):

- Argo CD provides a web-based user interface that allows users to visualize the applications, compare the current and desired states, and manually synchronize or rollback deployments.

API Server:

Its The k8s API server only which handles communication between the Argo CD CLI, Web UI, and the Argo CD controller.
It exposes RESTful APIs for interacting with Argo CD.

Metrics and Monitoring:

Argo CD supports metrics and monitoring, allowing you to gather data on the performance and health of the Argo CD components.

Interview Terminologies

- **Argo CD**: The primary tool for declarative continuous delivery of Kubernetes applications. It uses Git repositories as the source of truth for application configuration.
- **Application CRD**: Custom Resource Definition in Kubernetes used by Argo CD to represent a deployed application and its desired state.
- **Sync Operation (Synchronize)**: The process by which Argo CD reconciles the desired state defined in the Git repository (yaml files) with the current state of the Kubernetes cluster.
- **Sync Policy:** Configurable settings that define how often Argo CD should check the Git repository for changes and synchronize with the Kubernetes cluster.

- **Rollback:** The action of reverting to a previous state of an application or environment, usually triggered in response to issues or failures.
- **Application Set**: A feature in Argo CD that allows managing multiple similar applications using a single definition.
- **Health Status:** The condition or state of an application, indicates whether it is running as expected or if there are issues.
- **Hooks:** Custom scripts or commands that can be executed before or after a synchronization operation. Hooks are defined in the Argo CD Application configuration.
- **Custom Resource Definition (CRD):** An extension of the Kubernetes API that represents a customization of a particular resource, such as an application (Application CR).

kubectl get app (OR) kubectl get application