

Lumen Standardized DevOps CI/CD Pipeline Onboarding Guide

Updated: March 1st, 2023

LUMEN

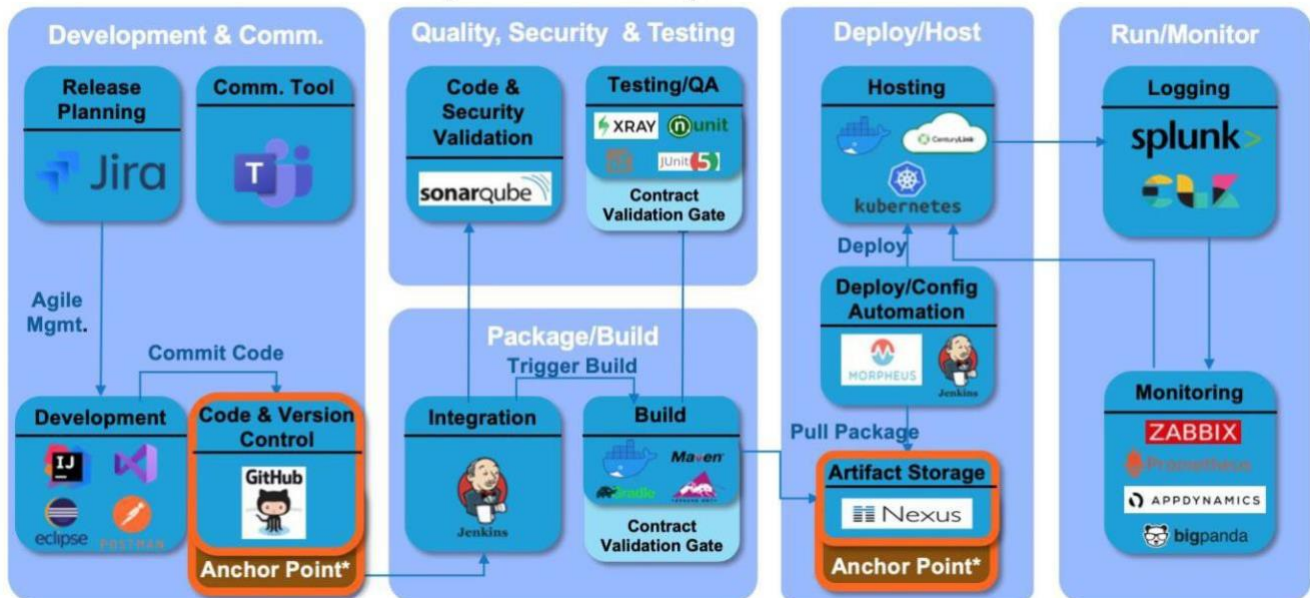
Table of contents

Overview.....	3
Jenkins Based Pipeline.....	7
Contract Validation	9
Anchor Points.....	10
Adoption Stats.....	11
DevOps Pipeline Template	13
Environments.....	29
Self-paced Onboarding Process	30
Orchestration Jobs	47
GCR and DevOps GCR	49
Best Practices.....	51

Overview

The Lumen Standardized DevOps CI/CD Pipeline is shown below:

Standardized DevOps CI/CD Pipeline



Development & Communications

Release Planning

Lumen Corporate JIRA has been chosen for the release planning. Configurations have been put in place to get automatic feedback and update from the Lumen Standardized DevOps CI/CD Pipeline different stages.

To onboard a project on JIRA please follow the Lumen Corporate JIRA onboarding documentation.

- JIRA documentation resource: <https://confluence.atlassian.com/jira>

Communication Tools

Corporate MS Teams has been chosen as the preferred communication channel for notifications and team communication. Configurations have been put in place to get automatic feedback and updates from the Lumen Standardized DevOps CI/CD Pipeline different stages through MS Teams channels.

To onboard a project on MS Teams please follow the Lumen Corporate MS Teams onboarding documentation.

- MS Teams documentation resource: <https://docs.microsoft.com/en-us/microsoftteams/teams-overview>

Development

In the Lumen Standardized DevOps CI/CD Pipeline, teams are free to choose their development environment and tools. The use of IDE's is encouraged to reduce development times.

Some of the recommended development tools are:

- Eclipse: <https://www.eclipse.org/ide/>
- JetBrains: <https://www.jetbrains.com/>
- Visual Studio: <https://visualstudio.microsoft.com/>
- Postman: <https://www.postman.com/>

Code and Version Control

Lumen organization on GitHub has been chosen for Code and Version Control.

All code must be stored in a repository belonging to the Lumen organization. Note that artifacts (binary files, third party dependencies) are not to be stored along the code.

To onboard a project on Lumen GitHub organization please follow the Lumen GitHub organization onboarding documentation.

- GitHub documentation resource: <https://docs.github.com/en>

Quality, Security and Testing

In the Lumen Standardized DevOps CI/CD Pipeline, teams are free to choose their QA and Testing tools, frameworks, and plugins. To keep the projects aligned the following recommendations are encouraged:

Code and Security Validation

Lumen Corporate SonarQube CE (Community Edition) is the Corp standard tool for static code analysis, security vulnerability scanning, and test code coverage.

To onboard an application (referred to as a "project" in SonarQube) onto SonarQube CE, please review the onboarding documentation:

- SonarQube CE documentation resource: <https://docs.sonarqube.org/latest/>

Testing and QA

To keep the projects aligned, teams should use JUnit-compatible XML and HTML reporting formats. Below is a list of recommended tools:

- NUnit: <https://docs.nunit.org/>
- JUnit: <https://junit.org/junit5/docs/current/user-guide/>
- Selenium: <https://www.selenium.dev/documentation/en/>

Package/Build

Integration

Lumen Corporate Jenkins servers have been chosen as the mandatory tool to be used for integration, job orchestration and pipelines.

To onboard a project on a Lumen Corporate Jenkins server please revise the Lumen Corporate Jenkins servers onboarding documentation.

- Jenkins documentation resource: <https://www.jenkins.io/doc/>

Build

In the Lumen Standardized DevOps CI/CD Pipeline, teams are free to choose their build tool chain, frameworks, and plugins.

Some of the recommended build tools are:

- Gradle: <https://docs.gradle.org/current/userguide/userguide.html>
- Maven: <https://maven.apache.org/guides/index.html>
- Ant: <https://ant.apache.org/manual/index.html>

Deploy/Host

Hosting

In the Lumen Standardized DevOps CI/CD Pipeline, teams are free to choose where to deploy their applications and intermediate stages.

Some of the recommendations are:

- Docker: [Docker Documentation | Docker Documentation](#)
- Kubernetes: <https://kubernetes.io/docs/home/>
- Lumen Cloud: <https://wwwctl.io/>

Deployment Configuration Automation

Lumen Corporate Morpheus has been chosen as the mandatory tool to be used for deployment configuration automation.

To onboard a project on Lumen Corporate Morpheus please review the Lumen Corporate Morpheus onboarding documentation.

- Morpheus documentation resource: <https://docs.morpheusdata.com/en/4.2.2/>

Artifact Storage

Lumen Corporate Nexus V3 has been chosen as the mandatory tool to be used artifact storage. All produced/consumed artifacts must be stored/retrieved in/from Nexus V3.

To onboard a project on Lumen Corporate Nexus V3 please follow the [Lumen Corporate Nexus V3 onboarding documentation](#).

- Nexus V3 documentation resource: <https://help.sonatype.com/repomanager3>

Run/Monitor

Logging and Indexing

In the Lumen Standardized DevOps CI/CD Pipeline teams are free to choose how to log and index their deployed applications.

Some of the recommended logging tools are:

- Splunk: <https://docs.splunk.com/Documentation>
- ELK: <https://www.elastic.co/guide/index.html>

Monitoring

In the Lumen Standardized DevOps CI/CD Pipeline, teams are free to choose how to monitor their applications.

Some of the recommended monitoring tools are:

- Zabbix: <https://www.zabbix.com/documentation/current/>
- Prometheus: <https://prometheus.io/docs/>

Jenkins Based Pipeline

Pipeline models

Jenkins provides a full range of jobs and project types for accomplishing a myriad of different tasks.

In the pipeline as a code paradigm, Jenkins offers two kinds of pipeline models:

- Declarative Pipeline
- Scripted Pipeline

Declarative Pipeline model has been chosen for the Lumen Standardized DevOps CI/CD Pipeline:

- It's structured and can be templated for homogeneous use.
- Offers the same customization levels as the Scripted Model.
- Provides mechanisms to track the teams are aligned.

For more information, please find Jenkins' official documentation resource:

- <https://www.jenkins.io/doc/book/pipeline/>

Agent models

Jenkins provides many agent types to handle distributed job handling. Dockerized agents have been chosen for the Lumen Standardized DevOps CI/CD Pipeline:

- Dockerized agents give control of the agent configuration to teams.
- Dockerized agents remove unnecessary delays in new agent deployment.
- This model allows agents with identical configuration for different purposes.

For more information, please see Jenkins' official documentation resources:

- <https://www.jenkins.io/doc/book/pipeline/syntax/#agent>
- <https://www.jenkins.io/doc/pipeline/tour/agents/>

Credentials

Code repositories are not the expected place to store sensitive information. The use of repositories to store passwords and tokens is not considered a best practice and should be avoided.

To store safely passwords and tokens the use of Jenkins credentials is encouraged.

For more information, Jenkins official documentation resource:

- <https://www.jenkins.io/doc/book/using/using-credentials/>

Jenkins Shared Libraries

Keeping pipelines as dry as possible is considered a good practice, allowing readers of Jenkinsfile to know what the pipelines do without prior knowledge of technical and implementation details. To achieve that goal Jenkins allows the use of Jenkins Shared Libraries.

For more information, Jenkins official documentation resource:

<https://www.jenkins.io/doc/book/pipeline/shared-libraries/>

Contract Validation

The Lumen Standardized DevOps CI/CD Pipeline provides a common framework, based on recommended stages, keeping the different projects aligned but allowing teams to work with freedom in solving their custom needs.

To track the adoption and compliance levels of each project in their pipelines some Contract Validation Gates have been put at disposal of the teams in the form of Jenkins Shared Library functions.

Contract Validation Gates will be used to track the compliance levels and to measure the adoption level.

Anchor Points

The Lumen Standardized DevOps CI/CD Pipeline uses two Anchor Points to pin up all the software development process. GitHub and Nexus are those Anchor Points.

In the development process, all the code produced will be stored in GitHub repositories.

In the build process, the code will be retrieved from GitHub repositories and the produced artifacts will be stored in Nexus.

In the deployment process, the artifacts will be retrieved from Nexus and deployed wherever it fits the project.

Adoption Stats

All the analysis, statistics and results produced during pipeline runs will be measured against the following KPIs:

DevOps KPIs

Common set of metrics and targets for transparency and accountability

KPI	Low (Level 0) (D)	Medium (Level 1) (C)	High (Level 2) (B)	Elite (Level 3) (A)
Deployment Frequency	Less than 1 per month	Between 1 per month and 1 per week	Between 1 per day and 1 per week	On-demand (multiple per day)
Avg Speed of Deployment	More than 2 hours	Between 30 and 120 min	Between 15 and 30 min	Less than 15 min
Avg Lead Time for Changes	More than 1 month	Between 1 week and 1 month	Between 1 day and 1 week	Less than 1 day
Mean Time to Restore Service (MTTR)	More than 1 week	Between 1 day and 1 week	Between 1 day and 1 hour	Less than 1 hour
Change Failure Rate	More than 45%	Between 30% and 45%	Between 15% and 30%	Less than 15%
Code Quality Bug Score	At least 1 Blocker or Critical Bug (E or D)	At least 1 Major Bug (C)	At least 1 Minor Bug (B)	0 Bugs (A)
Static Application Security Testing (SAST) Vuln Score	At least 1 Blocker or Critical Vuln (E or D)	At least 1 Major Vuln (C)	At least 1 Minor Vuln (B)	Info Vulns Only (A)
Dependency Vulnerabilities (Severity Level)	At least 1 Critical or High Dependency Vuln	At least 1 Moderate Dependency Vuln	At least 1 Low Dependency Vuln	0 Dependency Vulns
Test Code Coverage	Less than 20%	Between 20% and 80%	80% or More	80% or More
Functional / System / E2E Tests	N/A	All Tests Run	All Tests Pass	All Tests Pass

DevOps KPI Definitions

How to calculate each metric

KPI	Definition
Deployment Frequency	How often you release a new version of a product or service to Production as measured by number of deploys per month.
Avg Speed of Deployment	How long a single deployment takes (in minutes) from deployment approval to deployment complete and code running in Production.
Avg Lead Time for Changes	Avg time it takes to go from PR request to merge feature branch into dev (or master) branch to code successfully running in Production.
Mean Time to Restore Service (MTTR)	Avg time to restore an impacted service when the disruption is due to a defect requiring a hotfix to be developed and deployed to Production.
Change Failure Rate	A measure of how frequently Production deployments result in service issues that require an immediate remedy -- either a rollback or a hotfix.
Code Quality Bug Score	SonarQube provides a Bug Score for each repo based on the number and type of bugs discovered in the source code. Is it both a letter grade and a bug count.
Static Application Security Testing (SAST) Vuln Score	SonarQube provides a Vulnerability Score for a repo based on the number and type of source code vulnerabilities. It reports both a letter grade and vuln count by severity (Blocker, Critical, Major, Minor, Info).
Dependency Vulnerabilities (Severity Level)	GitHub evaluates a repo for dependency vulnerabilities (packages, libraries, etc) that have known CVEs and reports severity based on Common Vulnerability Scoring System (CVSS) (Critical, High, Moderate, or Low)
Test Code Coverage	A measure of Test-driven Development (TDD) adoption, the percentage of functions or methods with unit and integration tests executed during automated testing.
Functional / System / E2E Tests	A tally of the total number of Functional, System or End-to-end user action tests (via Selenium) that passed.

Current automatically generated DevOps Scorecards can be checked at [DevOps-Scorecard](#).

DevOps Pipeline Template

Currently the template of the Lumen Standardized DevOps CI/CD Pipeline is available in GitHub:

- <https://github.com/CenturyLink/jsl-jenkins-shared-library/blob/master/templates/Jenkinsfile>

The chosen pipeline type is the Declarative Pipeline model, which provides a DSL based framework.

For illustrative and training purposes, a copy of the template can be found below:

```
/*
Library declaration.
Notes:
identifier includes the version of the library (git tag / branch)
remote includes the repository git url
credentialsId needs to be of the type SSH key in Jenkins
_ at the end of the declaration loads the whole library

This section always runs in the master jenkins.
*/
library(
    identifier: 'jsl-jenkins-shared-library@VERSION',
    retriever: modernSCM(
        [
            $class: 'GitSCMSource',
            remote: "git@github.com:CenturyLink/jsl-jenkins-shared-library.git",
            credentialsId: 'SCMAUTO_SSH_DEVOPS_PIPELINE',
            extensions: [[ $class: 'WipeWorkspace' ]]
        ]
    )
) _

pipeline {
    environment {

        // Credentials:
        // GITHUB_TOKEN_CREDENTIALS github token, jenkins user password credential. SCMAUTO_GITHUB contains the GitHub token
        from SCMAuto user, which need to have access to the repository.
        // GITHUB_SSH_CREDENTIALS github ssh private key, jenkins private key credential. SCMAUTO_SSH_DEVOPS_PIPELINE contains
        the SSH key from SCMAuto user, which need to have access to the repository.
        // DOCKER_CREDENTIALS Docker access info, jenkins secret file credential with environment variables to export.
        // KUBE_CREDENTIALS Kubernetes access info, jenkins secret file credential with environment variables to export. For
        PRs.
        // KUBE_CREDENTIALS_TEST Kubernetes access info, jenkins secret file credential with environment variables to export.
        For branches.
        // AMAZON_CREDENTIALS AWS access info, jenkins secret file credential with environment variables to export
        // SONARQUBE_CREDENTIALS Sonarqube access info, jenkins secret text
        // GCP_CREDENTIALS GCP access info, jenkins secret file credential with environment variables to export
        // JIRA_CREDENTIALS Jira access info, jenkins secret file credential with environment variables to export
        // MORPHEUS_CREDENTIALS Morpheus access info, jenkins secret text
        // MSTEAMS_CREDENTIALS MS Teams access info, jenkins secret text
        // QUALITY_GATE_CREDENTIALS Credentials to gather all the contract validation gates expected to be crossed.
        // PROJECT_MAL The MAL of the project

        GITHUB_TOKEN_CREDENTIALS = 'SCMAUTO_GITHUB'
        GITHUB_SSH_CREDENTIALS = 'SCMAUTO_SSH_DEVOPS_PIPELINE'
        DOCKER_CREDENTIALS = ''
        KUBE_CREDENTIALS = ''
        KUBE_CREDENTIALS_TEST = ''
        KUBE_CREDENTIALS_PROD = ''
        AMAZON_CREDENTIALS = ''
        SONARQUBE_CREDENTIALS = ''
        GCP_CREDENTIALS = ''
    }
}
```

```

JIRA_CREDENTIALS = ''
MORPHEUS_CREDENTIALS = ''
MSTEAMS_CREDENTIALS = ''
QUALITY_GATE_CREDENTIALS = 'qg-creds'
//Deployment control credentialsId
AUTHORIZED_USERS = ''
DEPLOY_AUTH_TOKEN = ''

// Custom project variables
// Add your project name (repo name will do)
PROJECT_NAME = 'current_project'
DOCKER_REPO = 'current_project/current_project_repo'
PROJECT_MAL = "current_project mal"

BRANCH_NAME = GIT_BRANCH.split('/')[1].trim().toLowerCase()
COMMIT_ID = GIT_COMMIT.substring(0,7).trim().toLowerCase()
PULL_REQUEST="pr-${env.CHANGE_ID}"
IMAGE_NAME = "${env.PROJECT_NAME}"
IMAGE_TAG = "${env.PULL_REQUEST}"
KUBE_DOCKER_SECRET_NAME = "${env.PROJECT_NAME}-${env.PULL_REQUEST}"
KUBE_DOCKER_SECRET_NAME_TEST = "${env.PROJECT_NAME}-${env.BRANCH_NAME}"
KUBE_DOCKER_SECRET_NAME_PROD = "${env.PROJECT_NAME}-${env.BRANCH_NAME}"
}

// Add parameters if needed or if deployment control is in place.
// parameters {
//     //https://www.jenkins.io/doc/book/pipeline/syntax/#parameters
//     text(name: 'GCR', defaultValue: '', description: 'Enter the GCR description. Only used in deployment to production
stage.')
//     text(name: 'VERSION', defaultValue: '', description: 'Version to deploy. Only used in deployment to production stage.')
// }

// https://www.jenkins.io/doc/book/pipeline/syntax/#agent
//Add agent sections in stages/stage if needed.

agent {
    label 'Docker-enabled'
}

options {
    // https://www.jenkins.io/doc/book/pipeline/syntax/#options

    timestamps ()
    timeout(time: 1, unit: 'HOURS')
    buildDiscarder(logRotator(numToKeepStr: '10', daysToKeepStr: '30'))
    preserveStashes(buildCount: 10)
    disableConcurrentBuilds()
}

// https://www.jenkins.io/doc/book/pipeline/syntax/#triggers

triggers {
    issueCommentTrigger('.*test this please.*')
}

stages {
    stage('Authorize - Prod only') {
        // when {
        //     //https://www.jenkins.io/doc/book/pipeline/syntax/#when
        //     //Use it to decide when this stage runs to select the pipeline flow
        // }
        steps {
            script {
                js1DeploymentControlKnob()
            }
        }
    }
}

```

```

stage('Static Code Analysis') {
    // when {
    //     //https://www.jenkins.io/doc/book/pipeline/syntax/#when
    //     //Use it to decide when this stage runs to select the pipeline flow
    // }
    parallel {
        stage('Linting') {
            agent {
                dockerfile {
                    // Insert your code here
                }
            }
            steps {
                script{
                    // Insert your code here
                }
            }
        }
        stage('DevSecOps'){
            steps {
                script {
                    // Contract validation gate
                    jslGitHubSecurityAlert()
                }
            }
        }
    }
}

stage('Build') {
    // when {
    //     //https://www.jenkins.io/doc/book/pipeline/syntax/#when
    //     //Use it to decide when this stage runs to select the pipeline flow
    // }
    agent {
        dockerfile {
            // Insert your code here
        }
    }
    steps {
        script {
            // Insert your code here
        }
    }
}

stage('Create Images') {
    // when {
    //     //https://www.jenkins.io/doc/book/pipeline/syntax/#when
    //     //Use it to decide when this stage runs to select the pipeline flow
    // }
    steps {
        script {
            // Insert your code here
        }
    }
}

stage('Test') {
    // when {
    //     //https://www.jenkins.io/doc/book/pipeline/syntax/#when
    //     //Use it to decide when this stage runs to select the pipeline flow
    // }
    stages {
        stage('Unit Tests') {
            agent {
                dockerfile {
                    // Insert your code here
                }
            }
            steps {
                script {
                    // Insert your code here
                }
            }
        }
    }
}

```

```

        // Contract validation gate
        // call jslPublishTestResults()
        // or
        // call jslPublishHTMLTestResults()
        // Insert your code here
    }
}
}
stage('Coverage'){
    steps {
        script {
            // Insert your code here
            // Contract validation gate
            jslQualityGateCodeCoverage('./cicd/conf/sonarqube/sonar-project.properties')
        }
    }
}
stage('Sonarqube'){
    agent {
        dockerfile {
            filename 'Dockerfile'
            dir 'cicd/docker/sonarqube'
            label 'Docker-enabled'
        }
    }
    steps {
        script {
            // Insert your code here
            // Contract validation gate
            jslSonarQubeStaticAnalysis('./cicd/conf/sonarqube/sonar-project.properties')
        }
    }
}
stage('Quality Gate'){
    agent {
        dockerfile {
            filename 'Dockerfile'
            dir 'cicd/docker/sonarqube'
            label 'Docker-enabled'
        }
    }
    steps {
        script {
            // Contract validation gate
            jslQualityGate()
        }
    }
}
}
}

stage('Deploy') {
    // when {
    //     //https://www.jenkins.io/doc/book/pipeline/syntax/#when
    //     //Use it to decide when this stage runs to select the pipeline flow
    // }
    steps {
        script {
            // Insert your code here
        }
    }
}

stage('After Deployment Testing') {
    // when {
    //     //https://www.jenkins.io/doc/book/pipeline/syntax/#when
    //     //Use it to decide when this stage runs to select the pipeline flow
    // }
    agent {
        dockerfile {
            // Insert your code here
        }
    }
}

```



```

    steps {
        script {
            // Insert your code here
            // Contract validation gate
            // call jslPublishTestResults()
            // or
            // call jslPublishHTMLTestResults()
            // Insert your code here
        }
    }
}

stage('Deploy to Prod - Prod only') {
    // when {
    // //https://www.jenkins.io/doc/book/pipeline/syntax/#when
    // //Use it to decide when this stage runs to select the pipeline flow
    // }
    steps {
        script {
            // Insert your code here
        }
    }
}

stage('Adoption Stats') {
    agent {
        dockerfile {
            filename 'Dockerfile'
            dir 'cicd/docker/jira'
            label 'Docker-enabled'
        }
    }
    steps {
        script {
            // Insert your code here
            // Contract validation gate
            // call the function with a single filename / wildcard expression
            // containing all the tests reports produced
            // jslAdoptionMain('')
        }
    }
}
}
post {
    /*
    https://www.jenkins.io/doc/book/pipeline/syntax/#post

    Always post somewhere the watermark:
        - md5sum of Jenkinsfile
        - Output of the Jenkinsfile checker output
    */
    success {
        jslNotification('success')
        cleanWs()
    }
    failure {
        jslNotification('failure')
        cleanWs()
    }
    unstable {
        jslNotification('unstable')
        cleanWs()
    }
}
}
}

```

Note: The template from above can be out of date. Check the provided link for the latest version.

Credentials

All pipeline-related credentials, mainly the ones that connect the Jenkins jobs with the tools selected by Lumen to be used in the jobs, must be stored safely in the Jenkins Credentials repository. Build manager configuration files (i.e., maven settings.xml like files) must be stored in Jenkins as configuration files as normally contain credentials to connect with the tools selected by Lumen to be used in jobs.

More information on configuration file storage in Jenkins can be found in Jenkins official documentation:

- <https://www.jenkins.io/doc/pipeline/steps/config-file-provider/>

To store application-based credentials (i.e., a password to connect to a DB by the application), in case they need to be stored in the GitHub repository, git crypt is supported.

For more info check the git crypt official documentation:

- <https://www.agwa.name/projects/git-crypt/>

A brief description of the common credentials included in the template can be found below:

- **GITHUB_TOKEN_CREDENTIALS:** GitHub token. Jenkins user password credential. SCMAUTO_GITHUB is the preset credential id containing the GitHub token from SCMAuto user, which need to have access to the repository.
- **GITHUB_SSH_CREDENTIALS:** GitHub ssh private key. Jenkins private key credential. SCMAUTO_SSH_DEVOPS_PIPELINE is the preset credential id containing the GitHub token from SCMAuto user, which need to have access to the repository.
- **DOCKER_CREDENTIALS:** Docker access info. Jenkins secret file credential with environment variables to export.
- **KUBE_CREDENTIALS:** Kubernetes access info. Jenkins secret file credential with environment variables to export.
- **KUBE_CREDENTIALS_TEST:** Kubernetes access info. Jenkins secret file credential with environment variables to export.
- **KUBE_CREDENTIALS_PROD:** Kubernetes access info. Jenkins secret file credential with environment variables to export.
- **AMAZON_CREDENTIALS_AWS:** access info. Jenkins user password credential with the AWS key and secret key.
- **SONARQUBE_CREDENTIALS:** SonarQube access info. Jenkins secret text.
- **GCP_CREDENTIALS_GCP:** access info. Jenkins secret file credential with environment variables to export
- **JIRA_CREDENTIALS:** Jira access info. Jenkins secret file credential with environment variables to export.
- **MSTEAMS_CREDENTIALS:** MS Teams access info. Jenkins secret text credential, containing the webhook to the expected MS Teams channel.
- **MORPHEUS_CREDENTIALS** Morpheus access info, Jenkins secret text credential.

- **QUALITY_GATE_CREDENTIALS:** List of contract validation gates agreed to be crossed. It is a Jenkins secret file credential containing the key values for each gate. Note that Lumen Jenkins runs in a Linux server and the expected file format is Unix (LF).
- **AUTHORIZED_USERS:** List of CUIDs of users who can run the pipeline in the production environment. If there is more than one CUID they must be separated by ',' and ' ' (space) (i.e. ACXXXX, ACYYYY, AZZZZ). Jenkins secret text credential.
- **DEPLOY_AUTH_TOKEN:** Deployment token that must be inserted during the pipeline by the authorized users as M2FA. Jenkins secret text credential.

These credentials enable the connectivity of the provided Jenkins Shared Library functions to interact with other services.

Agents

By parameter: Agent selection after parameters

Fixed: Hardcode the label (by stage or global).

Dockerized: In each stage, select a docker image as base and mount the workspace as a volume. Beware of permissions.

<https://www.jenkins.io/doc/book/pipeline/syntax/#agent>

Labels available:

- Docker-enabled: Pool of agents with docker available running linux.
- Docker-compose: Pool of agents with docker and docker compose available running linux.

Parameters

Currently the pipeline is not asking for any input parameter. In case of need, click on the below link:

- <https://www.jenkins.io/doc/book/pipeline/syntax/#parameters>

Triggers

Section with standard (cron) or webhook based triggers (GitHub):

- <https://www.jenkins.io/doc/book/pipeline/syntax/#triggers>

GitHub pipeline plugin can be used to trigger jobs from comments. See linked documentation:

- <https://github.com/jenkinsci/pipeline-github-plugin>

Options

As a code:

- **timeout**: How long the pipeline can run before being considered failed.
- **buildDiscarder**: How many build results will be stored in Jenkins for the current job.
- **disableConcurrentBuild**: Prevents the same job to start concurrent builds to reduce the load in the agents.
- **preserveStashes**: It keeps the stashed of previous runs. It must be sync with the number of builds results to be stored configured in buildDiscarder.
- **timestamps**: Add timestamp to allow the debug of parallel stages. Depends on a plugin.

Details about options can be found in Jenkins official documentation:

- <https://www.jenkins.io/doc/book/pipeline/syntax/#options>

Stages

Following the design patterns of a Jenkins Declarative Pipeline, the pipeline will run a succession of stages. Each stage can contain more stages (sequential or parallel) or a series of steps.

Details about stage definition can be found in Jenkins' official documentation:

- <https://www.jenkins.io/doc/book/pipeline/syntax/#stages>

Which stage runs in which case, can be controlled using when clauses. Details about when clause can be found in Jenkins official documentation:

- <https://www.jenkins.io/doc/book/pipeline/syntax/#when>

Authorize - Prod only

This stage will run when a manually controlled action is needed to authorize. It can be used to authorize the deployment to production or can be replicated/tuned to authorize other use cases. Uses `jslDeploymentControlKnob()` Jenkins shared library and its associated credentials for a M2FA of job triggering.

Static Code Analysis

This stage will contain all stages related to simple static code analysis and static DevSecOps. Currently they are executed in parallel.

Linting

This stage is the placeholder for all the linting related steps contained in a project, the Jenkins Shared Library provides functions for most of the common build managers.

DevSecOps

This stage calls one of the Anchor Points and Contract Validation stages. It must include the call to the `jslGitHubSecurityAlert()` Jenkins Shared Library function to enable recommended GitHub Security Alert configuration.

Build

This stage is the placeholder for the build related steps contained in a project. The Jenkins Shared Library provides functions for most of the common build managers.

Create Images

This stage is the placeholder for the build related steps contained in a project. The Jenkins Shared Library provides functions to help teams to create images and push them to the Nexus 3 repository.

If the project does not create any docker image this stage can be removed.

Test

This stage is the placeholder for several sequential stages:

Unit Test

This stage is the placeholder for the unit testing related steps contained in a project. The Jenkins Shared Library provides functions for most of the common build managers.

This stage calls one of the Anchor Points and Contract Validation stages. It must include the call to the `jslPublishTestResults()` Jenkins Shared Library function to check that tests reports are uploaded to the designed place.

Coverage

This stage is the placeholder for the coverage related steps contained in a project. the Jenkins Shared Library provides functions for most of the common build managers.

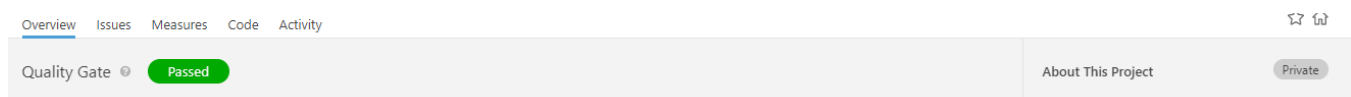
This stage calls one of the Anchor Points and Contract Validation stages. It must include the call to the `jslQualityGateCodeCoverage()` Jenkins Shared Library function to verify the SonarQube code coverage settings are the agreed ones and it's not disabling any check (i.e. code coverage).

SonarQube

This stage calls one of the Anchor Points and Contract Validation stages. It must include the call to the `jslSonarQubeStaticAnalysis()` Jenkins Shared Library function to perform and upload a SonarQube analysis to the corporate SonarQube.

Quality Gate

This stage calls one of the Anchor Points and Contract Validation stages. It must include the call to the `jslQualityGate()` Jenkins Shared Library function to check that the SonarQube analysis results are the expected ones, checking the SonarQube Quality Gate in a traffic light way to prevent regressions:



Deploy

This stage is the placeholder for the build related steps contained in a project. The Jenkins Shared Library provides functions to help teams to create images and push them to the Nexus 3 repository.

Deploy to Prod - Prod only

There are two philosophies related to what to deploy to production. The first one is to follow the same CI/CD steps (including building artifacts from code) again, and the second one is just deploy the produced artifact in the previous development phase (pre prod / test).

To cover the latter, using when clauses this stage can be used to deploy to prod in a segregated step.

After Deployment Testing

This stage is the placeholder for tests to be executed against the deployed instance contained in a project. The Jenkins Shared Library provides functions to help teams to execute tests against the deployed instance in the previous stage.

This stage calls one of the Anchor Points and Contract Validation stages. It must include the call to the `jslPublishTestResults()` or `jslPublishHTMLTestResults()` Jenkins Shared Library function to check that tests reports are uploaded to the designed place.

Adoption Stats

This stage is mandatory to get the needed statistics for the DevOps Score Card. The use of `jslAdoptionMain()` Jenkins shared library function is mandatory to publish each pipeline run statistics. It has a mandatory input parameter which is the path (wildcards accepted) to the report or reports (white spaces accepted to separate different reports) generated along previous stages.

Post

This stage is the placeholder for pipeline results notification. Every subsection available should use the provided JSL function for notification. At least success, failure, unstable and always sections are expected to be used.

This stage calls one of the Anchor Points and Contract Validation stages. It must include the call to the `jslNotification()` Jenkins Shared Library function to check that all the Contract Validation Gate steps have been executed.

The use of `cleanWS()` is recommended to clean up the leftovers of previous stages.

Lumen Jenkins Shared Library

To avoid complex code and improve the clarity of the pipeline's Jenkinsfile the use of Jenkins Shared Libraries is encouraged, as is considered a best practice in Declarative Pipelines to keep the Jenkinsfile as free of code as possible.

Jenkins Shared Libraries take advantage of groovy language. For more information on their structure and development documentation follow the link from below:

<https://www.jenkins.io/doc/book/pipeline/shared-libraries/>

The chosen pipeline type is the Declarative Pipeline model, which provides a DSL based framework. Therefore, any JSL development should be following that model.

For common steps and anchor points a common Jenkins shared library is available for its use in Lumen projects:

<https://github.com/CenturyLink/jsl-jenkins-shared-library>

For a more thorough information consult the library documentation that can be found at:

[Class-Documentation · CenturyLink/jsl-jenkins-shared-library Wiki \(github.com\)](#)

For documentation related to an specific release, please check the code branch for the release:

<https://github.com/CenturyLink/jsl-jenkins-shared-library/blob/release/20210112/Class-documentation.md>

A summary of some key functions is provided below in the next sections.

Useful Steps

jslBuildAndPushToNexus

- [jslBuildAndPushToNexus · CenturyLink/jsl-jenkins-shared-library Wiki \(github.com\)](#)

Function to build a Dockerfile and push to Lumen Nexus V3 server.

jslDeployK8s

- [jslDeployK8s · CenturyLink/jsl-jenkins-shared-library Wiki \(github.com\)](#)

Function to deploy a K8s manifest to Lumen Hyperion Kubernetes cluster.

jslGitHubMessage

- [jslGitHubMessage · CenturyLink/jsl-jenkins-shared-library Wiki](#)

Function to publish a message to a GitHub Pull Request.

jslMavenWrapper

- [jslMavenWrapper · CenturyLink/jsl-jenkins-shared-library Wiki \(github.com\)](#)

Function to wrap maven use.

jslYarnWrapper

- [jslYarnWrapper · CenturyLink/jsl-jenkins-shared-library Wiki \(github.com\)](#)

Function to wrap Yarn use.

jslNpmWrapper

- [jslNpmWrapper · CenturyLink/jsl-jenkins-shared-library Wiki \(github.com\)](#)

Function to wrap Npm use.

jslDeploymentControlKnob

- [jslDeploymentControlKnob · CenturyLink/jsl-jenkins-shared-library Wiki \(github.com\)](#)

Function control that a user belonging to a selected list of users, inserts a secret token. Useful to authorize workflows (i.e. deployment to production).

jslTriggerRemoteJob

- [jslTriggerRemoteJob · CenturyLink/jsl-jenkins-shared-library Wiki \(github.com\)](#)

Function to trigger another Jenkins job. Designed to orchestrate jobs.

jslJiraSendBuildInfo

- [jslJiraSendBuildInfo · CenturyLink/jsl-jenkins-shared-library Wiki \(github.com\)](#)

This function sends build info from the Jenkins build to the Jira Issue to which it pertains. To use, call the function in a post step within the build stage. To view data, click on a Jira story, and click on the *builds* link from the **Development** attribute.

jslJiraSendDeploymentInfo

- [jslJiraSendDeploymentInfo · CenturyLink/jsl-jenkins-shared-library Wiki \(github.com\)](#)

This function sends deployment info from Jenkins to the Jira Issue(s) to which it pertains. To use, call the function in a post step within the build stage. To view data, click on a Jira story, and under the **Releases** attribute, click on the *Environment Type* (Development, Testing, Staging, or Production) to open a modal that displays deployment info for that story.

Contract Validation Gate Steps

jslGitHubSecurityAlert

- [jslGitHubSecurityAlert · CenturyLink/jsl-jenkins-shared-library Wiki](#)

This function implements a check and configures (if needed) the GitHub Security Alerts for the project repository to be compliant with Lumen policies. It will check the status and enable it if disabled. It will leave a watermark towards Contract Validation Gate verification.

jslPublishTestResults

- [jslPublishTestResults · CenturyLink/jsl-jenkins-shared-library Wiki \(github.com\)](#)

This function implements publishes test reports in xml format to the designated places to be compliant with Lumen policies. It will leave a watermark towards Contract Validation Gate verification.

jslPublishHTMLTestResults

- [jslPublishHTMLTestResults · CenturyLink/jsl-jenkins-shared-library Wiki \(github.com\)](#)

This function implements publishes test reports in HTML format to the designated places to be compliant with Lumen policies. It will leave a watermark towards Contract Validation Gate verification.

jslQualityGateCodeCoverage

- [jslQualityGateCodeCoverage · CenturyLink/jsl-jenkins-shared-library Wiki \(github.com\)](#)

This function checks the SonarQube code coverage configuration to be compliant with Lumen policies. It will leave a watermark towards Contract Validation Gate verification.

jslQualityGate

- [jslQualityGate · CenturyLink/jsl-jenkins-shared-library Wiki \(github.com\)](#)

This function checks the code analysis results in SonarQube are compliant with Lumen policies. It will leave a watermark towards Contract Validation Gate verification.

jslSonarQubeStaticAnalysis

This function runs a code analysis against SonarQube compliant with Lumen policies. It will leave a watermark towards Contract Validation Gate verification.

This function uses sonar-scanner tool to be build manager agnostic. It needs a valid sonar-project.properties file with the scan details. More info can be found at:

<https://docs.sonarqube.org/latest/analysis/scan/sonarscanner/>

jslNotifyTeams

- [jslNotifyTeams · CenturyLink/jsl-jenkins-shared-library Wiki \(github.com\)](#)

This function reports the results of the pipeline execution compliant with Lumen policies. This functions check that all watermarks from mandatory stages are in place and puts a red or green light for the Contract Validation Gate. This Contract Validation Gate can be used to protect branches for merging.

jslAdoptionMain

- [jslAdoptionMain · CenturyLink/jsl-jenkins-shared-library Wiki \(github.com\)](#)

This function gathers all the test reports produced in each run and assembles all the statistics needed to produce the DevOps scorecard.

Environments

Currently the provided shared library steps work for both branches in the upstream repository and pull requests (from forks and branches).

Jobs are encouraged to be based on the following Jenkins project type (Multibranch Pipeline)

<https://plugins.jenkins.io/workflow-multibranch/>

<https://www.jenkins.io/doc/book/pipeline/multibranch/>

This kind of Jenkins project will scan the repositories looking for branches and pull request and automatically creates jobs for each that match the configuration.

The Jenkinsfile and the associated Jenkins Shared Library can include logic to distinguish the between branches and PRs. This capability can be used to change the deployment destination from 'production' named branch from any PR.

Self-paced Onboarding Process

Placeholders and Contract Validation Gates

All agreed Contract Validation Gates must be crossed to get the LumenContractValidation status check in GitHub.

If due to project constraints (i.e. the project does not have unit test yet), the recommended approach is to leave the stage and put a place holder to cross it (i.e. generate an empty unit test report).

Credentials

To keep credentials away from being stored with the code, the current solution for safe storing credentials in Lumen Standardized DevOps CI/CD Pipeline is the use of Jenkins credentials:

<https://www.jenkins.io/doc/book/using/using-credentials/>

To onboard in the different services used by the Lumen Standardized DevOps CI/CD Pipeline please follow each service onboarding guide.

Docker credentials

Working with docker images implies working with a docker registry. The kind of credentials expected are 'Secret file' type credentials, and they are used by some of the provided Jenkins shared library functions. Note that the Corporate Jenkins runs on Linux and credentials files are expected to be in Unix format. Please convert the text file to Unix format before uploading them to avoid issues.

The file should contain the following structure:

```
export DOCKER_REGISTRY_USR=" "  
export DOCKER_REGISTRY_PSW=" "  
export DOCKER_REGISTRY_HOST=" "  
export DOCKER_CONFIG=" "
```

An example:

```
export DOCKER_REGISTRY_USR="DOCKER_REGISTRY_USERNAME"
export DOCKER_REGISTRY_PSW="DOCKER_REGISTRY_PASSWORD"
export DOCKER_REGISTRY_HOST="nexus3-server-and-sample-port:1234"
export DOCKER_CONFIG="./.docker_config"
```

Kubernetes credentials

Working with a Kubernetes implies setting up a kubeconfig file. To avoid the complexity of the kubeconfig file, the pipeline uses a 'Secret file' type credential to deploy to your Lumen cluster with the following format.

The file should contain the following structure:

```
export KUBECONFIG=""
export KUBERNETES_NAMESPACE=
export KUBERNETES_TOKEN=
export KUBERNETES_SERVER=
export KUBERNETES_INSECURE=
export KUBERNETES_PREFIX=
export CLUSTER_DOMAIN=
```

Note that the Corporate Jenkins runs on Linux and credentials files are expected to be in Unix format. Please convert the text file to Unix format before uploading them to avoid issues. An example:

```
export KUBECONFIG=".localkube_config"
export KUBERNETES_NAMESPACE=fake-namespaces
export KUBERNETES_TOKEN=fake-token
export KUBERNETES_SERVER=https://fake-kuberentes.corp.intranet:1234/
export KUBERNETES_INSECURE=true
export KUBERNETES_PREFIX=fake-project-prefix-to-have-multiple-things-in-same-namespaces
export CLUSTER_DOMAIN=fake-domain.corp.intranet
```

SonarQube credentials

SonarQube tokens are stored as 'Secret text' type credentials and are defined as global in Jenkins.

If you are using <https://sonar.foss.corp.intranet> you should use SONARQUBE_CREDENTIALS = 'sonartokenprod'

If you are using <https://sonarent.corp.intranet> you should use SONARQUBE_CREDENTIALS = 'sonarcicdent'

Morpheus

Working with Morpheus and its associated Jenkins shared library functions a Jenkins credential must be set to safely store the connection details. The kind of credentials expected are 'Secret file' type credentials. Note that the Corporate Jenkins runs on Linux and credentials files are expected to be in Unix format. Please convert the text file to Unix format before uploading them to avoid issues.

The file should contain the following structure:

```
export MORPHEUS_SERVER=  
export MORPHEUS_INSECURE=  
export MORPHEUS_USER=  
export MORPHEUS_PASSWORD=  
export MORPHEUS_PREFIX=
```

An example:

```
export MORPHEUS_SERVER=https://morpheus-fake.fake_domain.intranet  
export MORPHEUS_INSECURE=true  
export MORPHEUS_USER=fake_user  
export MORPHEUS_PASSWORD=fake_password  
export MORPHEUS_PREFIX=fake_morpheus_prefix
```

JIRA

The Lumen Standardized DevOps CI/CD Pipeline interacts with JIRA to gather statistics during the mandatory Adoption Stats stage. The kind of credentials expected are 'Secret file' type credentials, and they are used by some of the provided Jenkins shared library functions.

Note that the Corporate Jenkins runs on Linux and credentials files are expected to be in Unix format.

Please convert the text file to Unix format before uploading them to avoid issues.

The file should contain the following structure:

```
credentials    =
cloudSite      =
myjira         =
myconfluence   =
default        =
```

An example:

```
credentials    = --user fake_account@centurylink.com --token fake_jira_token
cloudSite      = ctl
myjira         = jiracloud -s https://${cloudSite}.atlassian.net ${credentials}
myconfluence   = confluencecloud -s https://${cloudSite}.atlassian.net/wiki ${credentials}
default        = ${myjira}
```

Jira Deployments

The Jenkins Jira Plugin allows you to view Jenkins deployment and build info within Jira. **Note** that for the Jira integration to work, the Jira Project must have Jira Deployments enabled. Another **note**, the plugin can be used by itself by following the following documentation(<https://wiki.jenkins.io/display/JENKINS/Atlassian%20Jira%20Software%20Cloud%20Plugin>) in the *How to use the integration* section. For convenience, the Jenkins Shared Library has *jslJiraSendBuildInfo* and *jslJiraSendDeploymentInfo* functions that wrap the plugin. The data can be viewed for individual stories as described in the *jslJiraSendBuildInfo* and *jslJiraSendDeploymentInfo* sections of the *Lumen Jenkins Shared Library - Useful Steps above* or you can click on **Deployments** in the left side nav bar to view historical data for all stories in the project that have deployment data.

Contract Validation Gate credentials

The Contract Validation gates that are going to be crossed are setup using a credential. The kind of credentials expected are 'Secret file' type credentials.

Note that the Corporate Jenkins runs on Linux and credentials files are expected to be in Unix format. Please convert the text file to Unix format before uploading them to avoid issues.

The file must contain the following keywords (each belonging to a Contract Validation Gate):

```
testing_bread_crumb
qualitygate_bread_crumb
sonarqube_bread_crumb
qualitygate_codecoverage_bread_crumb
```

github_sa_bread_crumb

These keywords are required for standardization; you cannot leave them aside. The defined keywords are checked in the Adoption Stats mandatory stage.

To ensure pipeline maintainability you should use the `QUALITY_GATE_CREDENTIALS = 'qg-creds'`.

Authorized users

To use the deployment knob Jenkins shared library, 'Secret text' type credentials are used. The credential should contain the list of CUIDs of the authorized users to open the deployment knob, separated with commas and a space.

An example:

AB62976, AC69988, AC04119, AC69986

Deployment token

To use the deployment knob Jenkins shared library, 'Secret text' type credentials are used. The credential should contain deployment token which the authorized users must introduce to open the deployment knob from the pipeline console.

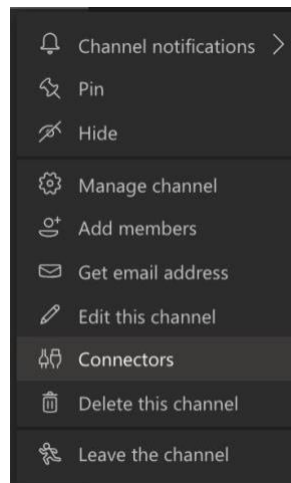
An example:

o7gFyEflmcxU3wZQMqeyUwKuVp6bAfSeDIQjt4PBBrvXkzbCKGZjX0SjD2k8dGMtT3Diurn0oMMjEWsF2xjrby99sFPe3bLoSBKUrtl9lOgwZjYM
oj1wS6xs3sG5bltEY

MS Teams Credentials

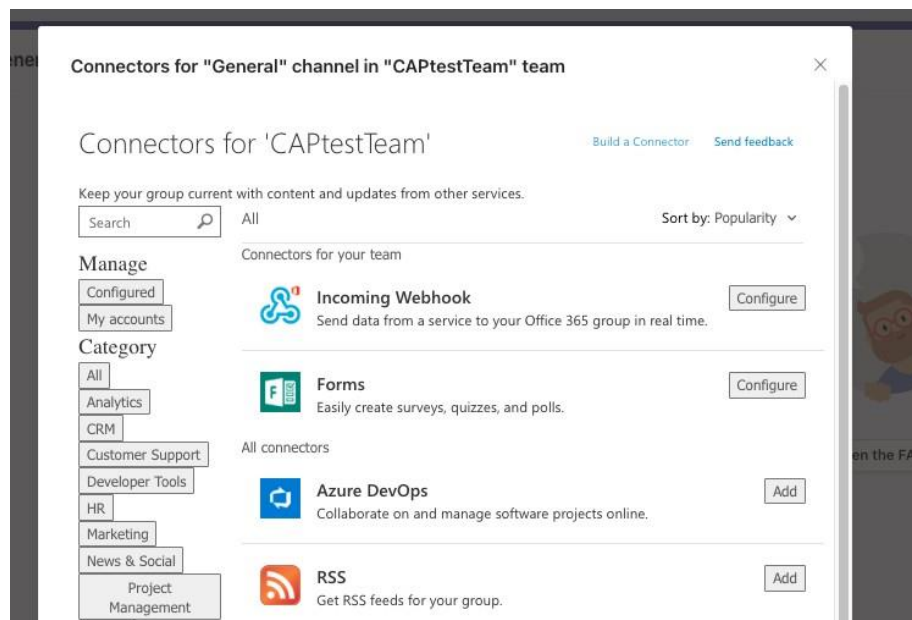
To forward pipeline result notifications, 'Secret text' type credentials are used. The credential should include the data gathered from the MS Teams channel:

(right click over the channel name)




Configure an 'Incoming Webhook' connector:

You may need to add it first. Then get back to this same screen and click on 'Configure'.



Select a name for the connector

Connectors for "General" channel in "CAPtestTeam" team

 Incoming Webhook [Send feedback](#)


The Incoming Webhook connector enables external services to notify you about activities that you want to track. To use this connector, you'll need to create certain settings on the other service, which needs to support a webhook that's compatible with the Office 365 connector format.

Fields marked with * are mandatory

To set up an Incoming Webhook, provide a name and select Create. *

Customize the image to associate with the data from this Incoming Webhook.

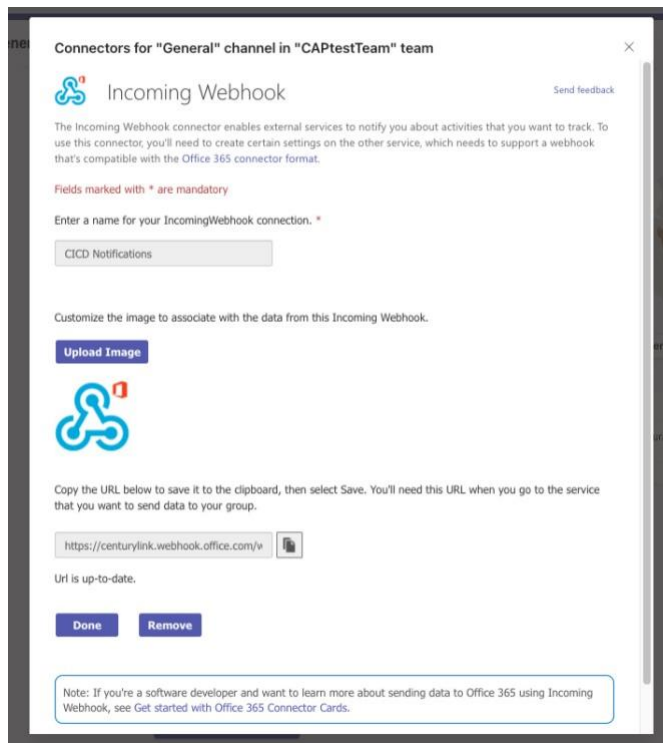
[Upload Image](#)


Default Image

[Create](#) [Cancel](#)

Note: If you're a software developer and want to learn more about sending data to Office 365 using Incoming Webhook, see [Get started with Office 365 Connector Cards](#).

Use the Webhook URL as the value for the credential (there is a copy button):



GitHub credentials

Different Jenkins plugins connect to GitHub using the two authentication's methods GitHub provides:

Token based authentication: The pipeline needs a 'Username and password' type credential for some of the plugins to connect to GitHub. SCMAuto user token is available under SCMAUTO_GITHUB credentials if no other service account is planned to be used.

SSH based authentication: The pipeline needs a 'SSH Username with private key' type credential for some of the plugins to connect to GitHub. SCMAuto ssh key is available under SCMAUTO_SSH_DEVOPS_PIPELINE credentials if no other service account is planned to be used.

Amazon Web Services credentials

The pipeline uses a 'Username and password' type credential for AWS interaction (if needed), containing the access key as user and the secret key as password.

Jenkins configuration

Once onboarded in the Lumen Jenkins server following its onboarding guide, Jenkins should be configured to contain the jobs related to our project.

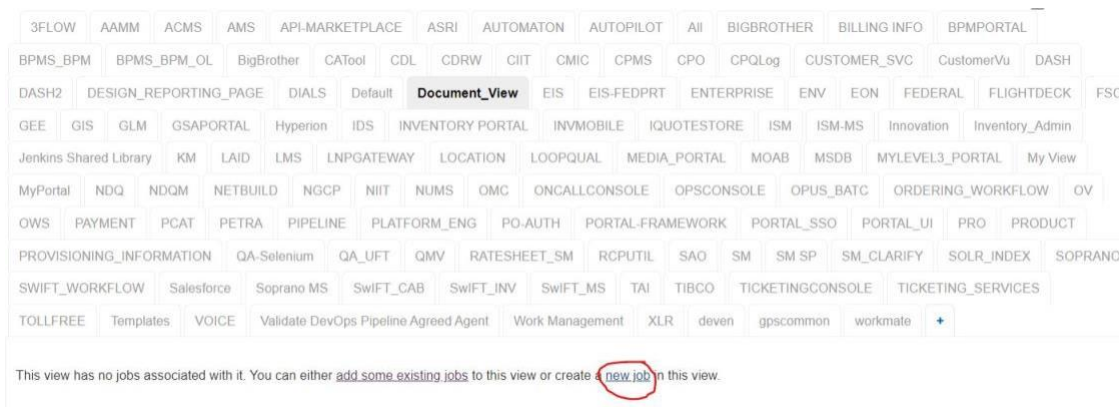
Setup a project folder

It is a best practice to create a folder to contain the projects. It can contain also more folders.

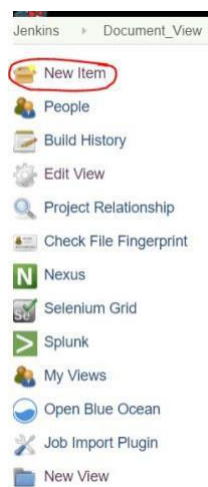
One of the advantages of using folders is that they allow to manage access rights to what they contain.

Another advantage is that credentials can be stored in folders and shared between Jenkins projects.

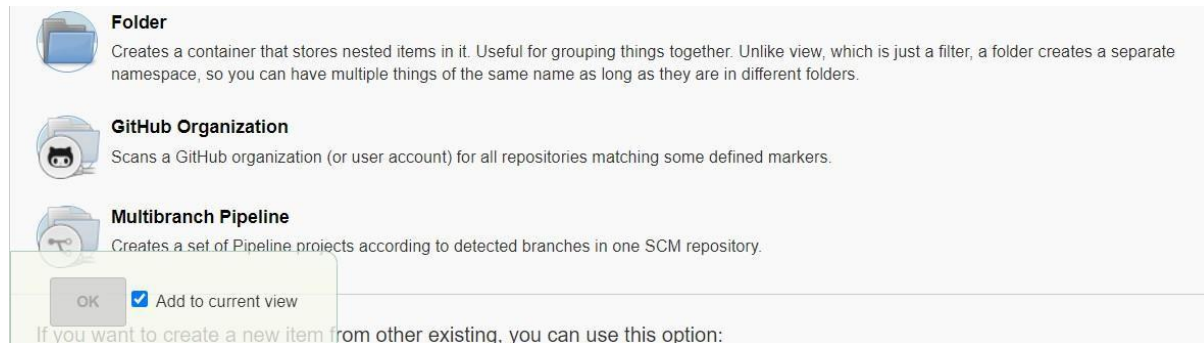
After selecting the desired view if the view has no existing jobs, the first folder can be created pressing the following button:



Or the regular 'New Item':



To create a folder, Folder item name must be introduced, and 'Folder' item type selected:



The screenshot shows the 'Add New Item' dialog in Jenkins. The 'Folder' option is selected, and the 'Add to current view' checkbox is checked. The dialog also shows options for 'GitHub Organization' and 'Multibranch Pipeline'.

Folder
Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, which is just a filter, a folder creates a separate namespace, so you can have multiple things of the same name as long as they are in different folders.

GitHub Organization
Scans a GitHub organization (or user account) for all repositories matching some defined markers.

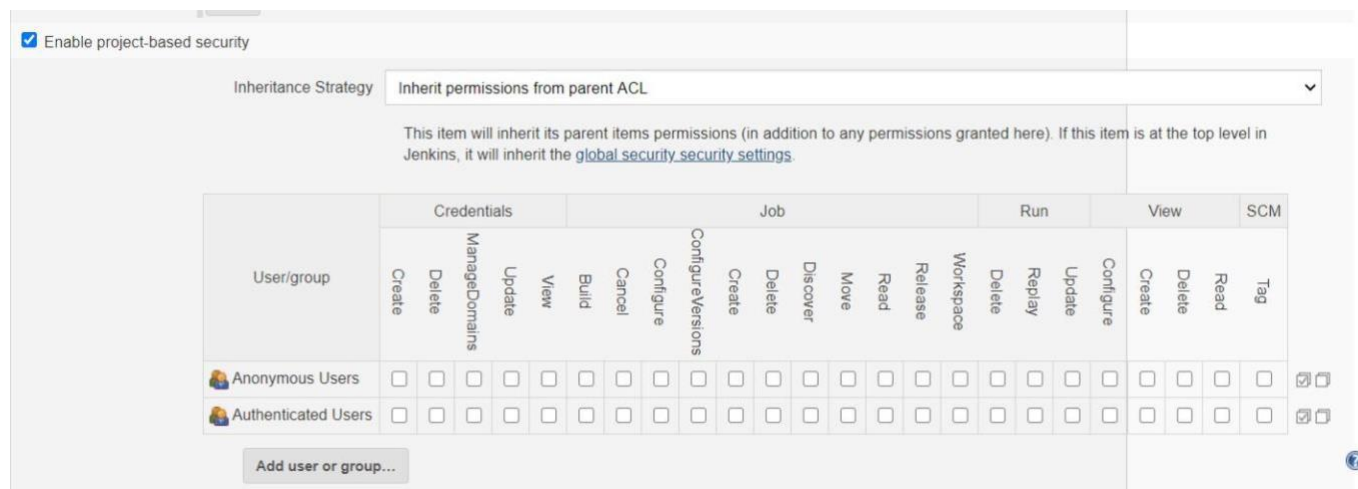
Multibranch Pipeline
Creates a set of Pipeline projects according to detected branches in one SCM repository.

OK ☒ Add to current view

If you want to create a new item from other existing, you can use this option:

Note that to get the item added to the current view the 'Add to current view' box must be checked.

In the configuration section of a folder, access rights can be managed, by clicking on 'Enable project-based security':



The screenshot shows the 'Configure' page for a folder in Jenkins. The 'Enable project-based security' checkbox is checked. The 'Inheritance Strategy' is set to 'Inherit permissions from parent ACL'. Below this, there is a table of permissions for different users and groups.

☒ Enable project-based security

Inheritance Strategy: Inherit permissions from parent ACL

This item will inherit its parent items permissions (in addition to any permissions granted here). If this item is at the top level in Jenkins, it will inherit the [global security settings](#).

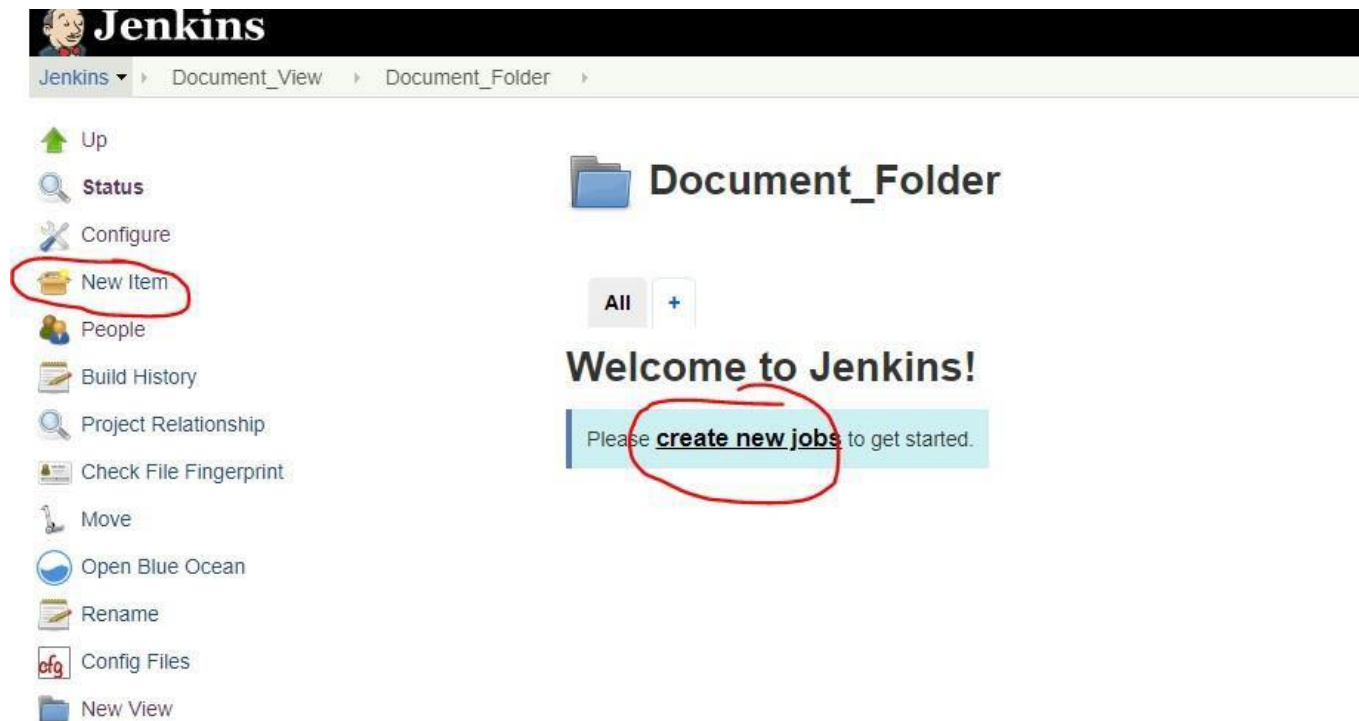
User/group	Credentials		Job						Run		View		SCM											
	Create	Delete	ManageDomains	Update	View	Build	Cancel	Configure	Create	Delete	Discover	Move	Read	Release	Workspace	Delete	Replay	Update	Configure	Create	Delete	Read	Tag	
Anonymous Users	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Authenticated Users	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

Add user or group...

Please ensure that the members of the team have the correct set of rights. Please refrain to allow anonymous users rights to perform destructive operations. Note that to add new users you should use their CUID and they should be onboarded in the Jenkins instance.

Set up a job

Jobs can be created anywhere, but the recommended way is to do so inside folder. Once inside a folder a job can be created clicking in any of the circled areas:

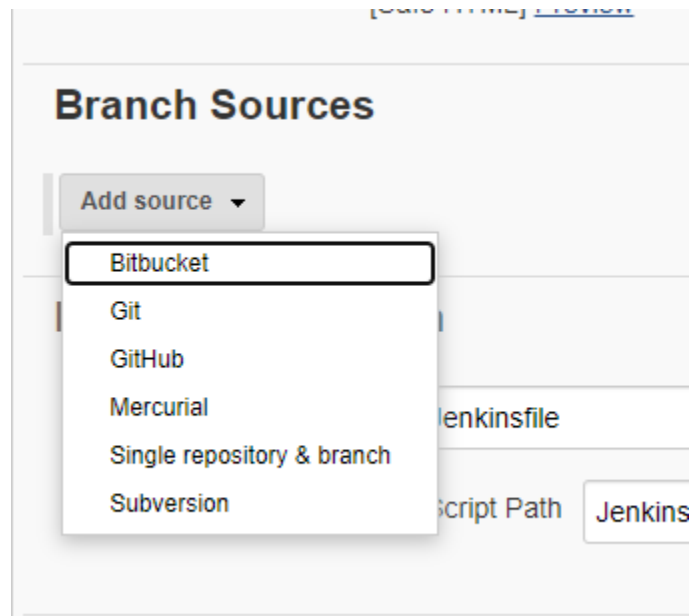


For the Lumen Standardized DevOps CI/CD Pipeline, the use of Multibranch Pipeline jobs is encouraged.

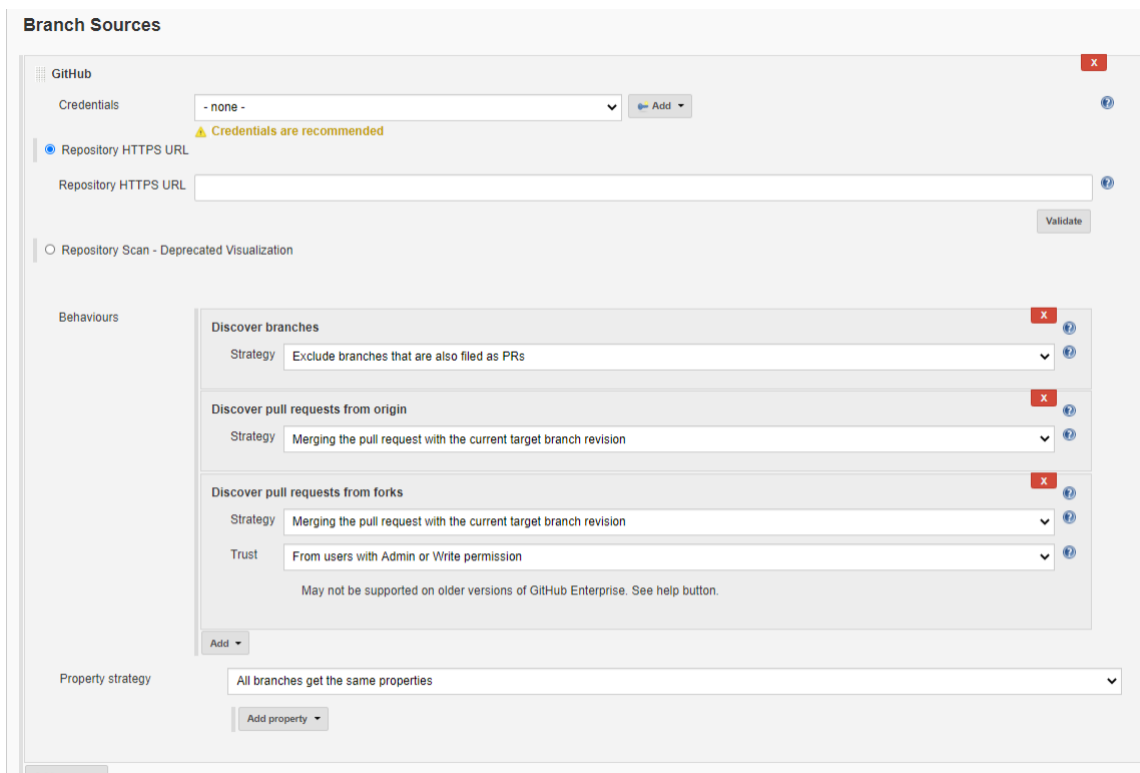


This kind of jobs scans a GitHub repository looking for Jenkinsfile files in previously configured places and automatically creates and removes jobs for branches and pull requests following the GitHub development flow.

Once created the job it should be configured to point to the project GitHub repository. To do so, a GitHub source must be added:



To configure the source, the 'Branch Sources' section has to be configured to match the GitHub repository and project development flow:



Branch Sources

GitHub

Credentials: Add ?

⚠ Credentials are recommended

☒ Repository HTTPS URL

Repository HTTPS URL: ?

Validate

☐ Repository Scan - Depreciated Visualization

Behaviours

Discover branches ?

Strategy: ?

Discover pull requests from origin ?

Strategy: ?

Discover pull requests from forks ?

Strategy: ?

Trust: ?

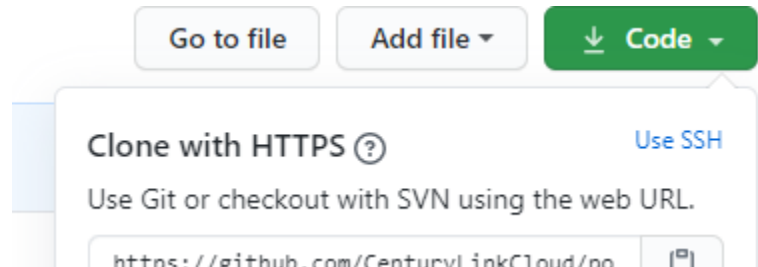
May not be supported on older versions of GitHub Enterprise. See help button.

Add

Property strategy: ?

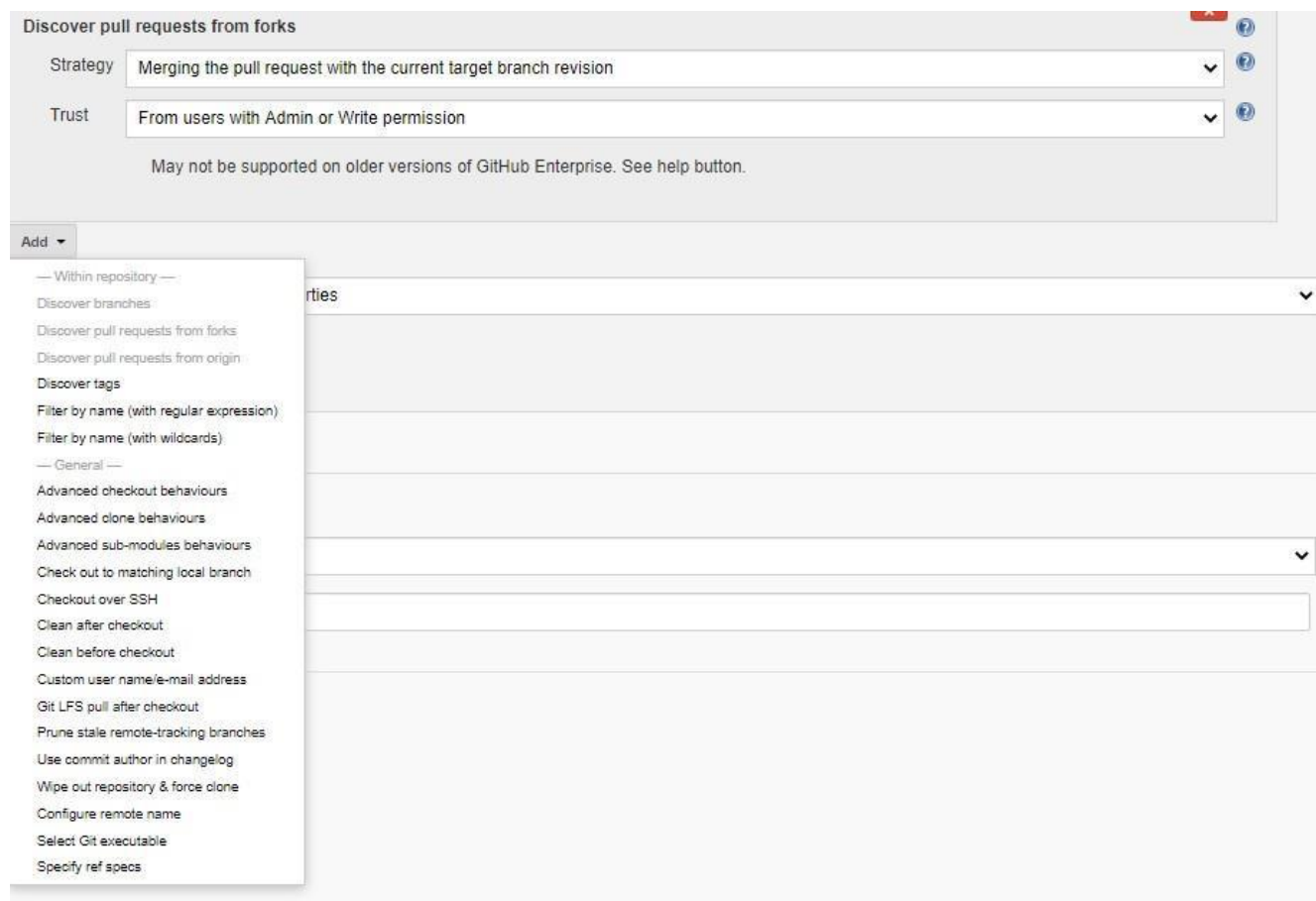
Add property

In 'Credentials', a GitHub Token with access to the repository must be selected. In 'Repository HTTPS URL' must be configured with the same address used to clone the repository through https:



Select the matching policies for “Discover Branches“, “Discover pull requests from origin” and “Discover pull requests from forks” to match the project workflow.

In case of need of filtering, or additional policies:



Filtering branches and pull request using the 'Filter by name' options are useful. Note: To filter pull requests they are named following the "PR-^[0-9]*\$" regular expression.

Filtering branches and pull requests using by 'Filter by name (with wildcards)' allows an easier management. Note: To filter default configured repositories branches and pull requests "master main PR-* pr-*" expression does the trick.

To configure the job to look for Jenkinsfile files in the repository the 'Build Configuration' must be configured:



Build Configuration

Mode: by Jenkinsfile

Script Path: cicd/jenkins/Jenkinsfile

The jobs will be triggered by webhooks but sometimes is useful to setup a periodic checking of changes in the repository:



Scan Multibranch Pipeline Triggers

- ☐ Build whenever a SNAPSHOT dependency is built
- ☐ Maven Dependency Update Trigger
- ☒ Periodically if not otherwise run
 - Interval: 5 minutes
- ☐ [FSTrigger] - Monitor files
- ☐ [FSTrigger] - Monitor folder

Once setup the Multibranch Pipeline will create jobs for all matching branches and pull request and keep doing so.

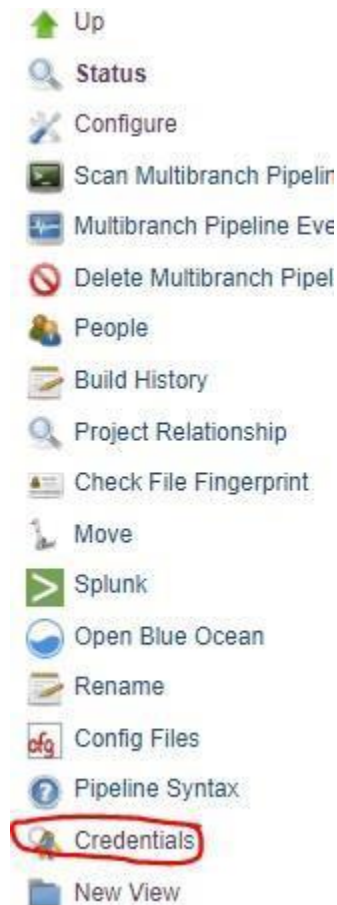
More information about Multibranch Pipeline jobs can be found on Jenkins's documentation:

<https://www.jenkins.io/doc/book/pipeline/multibranch/#creating-a-multibranch-pipeline>


Set up a credential

Credentials are useful to store secrets that cannot be stored in regular repositories (like access tokens). They can be created at job level or a folder level. A folder level credential is inherited by all the jobs inside the folder. If any job needs a different credential with the same name, a job level credential can be created to override the inherited one only for itself. This behavior is the same inside nested folders and jobs.



To manage credential the 'Credential' button can be used:





In the following picture the job level and folder level credentials can be observed:



Credentials

T	P	Store	Domain	ID	Name
Icon: S M L					
Stores scoped to Document_Folder » DELETE					
P	Store	Domains			
	Document_Folder » DELETE	 (global)			

Stores from parent

P	Store	Domains
	Document_Folder	 (global)






To manage a credential, select the 'Store':



jenkins » Document_View » Document_Folder » DELETE » Credentials » Folder » Global credentials (unrestricted) »

[Back to credential domains](#)
[Add Credentials](#)

Global credentials (unrestricted)

Credentials that should be available irrespective of domain specification to requirements matching

	Name	Kind	Description	
	test1 (test1)	Username with password	test1	
	test2-secret.txt (test2)	Secret file	test2	
	test3	Secret text	test3	

Icon:   L

Already existing credentials can be updated, but not retrieved from the UI.

More information about Credentials can be found on Jenkins's documentation:

<https://www.jenkins.io/doc/book/using/using-credentials/>

Note that the Corporate Jenkins runs on Linux and Jenkins secret file credentials are expected to be in Unix format. Please convert the text file to Unix format before uploading them to avoid issues.

Pipeline development

The main source of documentation for pipeline development can be found in Jenkins' official documentation:

<https://www.jenkins.io/doc/book/pipeline/syntax/>

The Lumen Standardized DevOps CI/CD Pipeline provides templates and examples to be customized by teams to accomplish their projects goals.

These templates provided a common framework, to keep the different projects aligned but as they can be customized and extended allows teams to work with freedom in solving their custom needs.

A set of Jenkins Shared Library functions are provided to handle with the Anchor Points and Contract Validation gates. The use of this functions is encouraged, to allow the tracking of adoption and compliance levels.

Teams are encouraged to keep their pipelines as dry as possible, to allow the readers of Jenkinsfile to know what the pipelines do without entering the details. To achieve that goal the use of custom Jenkins Shared libraries is encouraged. More information can be found at Jenkins official documentation:

<https://www.jenkins.io/doc/book/pipeline/shared-libraries/>

Orchestration Jobs

The tools described in this manual can be used (i.e. Jenkins) for tasks that are outside the scope of a CI/CD pipeline nor by the Lumen Standard DevOps Pipeline. As a rule of thumb, it is considered one repository per application with a single CI/CD job following the Lumen Standard DevOps Pipeline. In several use cases, an application is composed by several 'parts' (i.e. microservices) that must be managed in a synchronized way.

For that use cases an orchestrator job is recommended to be used. A sample template is available in GitHub:

<https://github.com/CenturyLink/jsl-orchestrator/blob/main/cicd/jenkins/Jenkinsfile>

For illustrative and training purposes, a copy of the template can be found below:

```
/*
Library declaration.
Notes:
  identifier includes the version of the library (git tag / branch)
  remote includes the repository git url
  credentialsId needs to be of the type SSH key in Jenkins
  _ at the end of the declaration loads the whole library
  This section always runs in the master jenkins.
*/
library(
    identifier: 'jsl-jenkins-shared-library@release/20210112',
    retriever: modernSCM(
        [
            $class: 'GitSCMSource',
            remote: "git@github.com:CenturyLink/jsl-jenkins-shared-library.git",
            credentialsId: 'SCMAUTO_SSH_DEVOPS_PIPELINE',
            extensions: [[ $class: 'WipeWorkspace' ]]
        ]
    )
)

pipeline {
    environment {

        /*
        Credentials:
        GITHUB_TOKEN_CREDENTIALS github token, jenkins user password credential
        GITHUB_SSH_CREDENTIALS github ssh private key, jenkins private key credential.
        */
        GITHUB_TOKEN_CREDENTIALS = 'SCMAUTO_GITHUB'
        GITHUB_SSH_CREDENTIALS = 'SCMAUTO_SSH_DEVOPS_PIPELINE'
        PROJECT_MAL = "DEVOPS-CICD"
    }

    parameters {
        // https://www.jenkins.io/doc/book/pipeline/syntax/#parameters
    }
}
```

```

    text(name: 'CICDJOB1', defaultValue: '', description: 'CICDJOB1 job to trigger. Left empty to skip.')
    text(name: 'CICDJOB2', defaultValue: '', description: 'CICDJOB2 job to trigger. Left empty to skip.')
    text(name: 'CICDJOB3', defaultValue: '', description: 'CICDJOB2 to trigger. Left empty to skip.')
}

/*
https://www.jenkins.io/doc/book/pipeline/syntax/#agent
Add agent sections in stages/stage if needed.
*/
agent {
    label 'Docker-enabled'
}

options {
    /*
    https://www.jenkins.io/doc/book/pipeline/syntax/#options
    */
    timestamps ()
    timeout(time: 2, unit: 'HOURS')
    buildDiscarder(logRotator(numToKeepStr: '10', daysToKeepStr: '30'))
    preserveStashes(buildCount: 10)
    disableConcurrentBuilds()
}

/*
https://www.jenkins.io/doc/book/pipeline/syntax/#triggers
*/
triggers {
    issueCommentTrigger('.*test this please.*')
}
stages {
    stage('Trigger CICDJOB1') {
        steps {
            script {
                js1TriggerRemoteJob(params.CICDJOB1, '/Jenkins Shared
Library/CATALOG/ORCHESTRATOR/TRIGGERED_JOBS/CICDJOB1/')
            }
        }
    }
    stage('Trigger CICDJOB2') {
        steps {
            script {
                js1TriggerRemoteJob(params.CICDJOB2, '/Jenkins Shared
Library/CATALOG/ORCHESTRATOR/TRIGGERED_JOBS/CICDJOB2/')
            }
        }
    }
    stage('Trigger CICDJOB3') {
        steps {
            script {
                js1TriggerRemoteJob(params.CICDJOB3, '/Jenkins Shared
Library/CATALOG/ORCHESTRATOR/TRIGGERED_JOBS/CICDJOB3/')
            }
        }
    }
}
}

```

GCR and DevOps GCR

ServiceNow integration

Release 20210714 and upper versions of JSL have all the related functions to integrate the ServiceNow Global Change Request workflow into your standardized CI/CD pipeline.

- For teams that plan to keep using manual GCR, there are several functions to search, read information, check approvals, or the kind of deployment you are doing based on the Jira cards included in the deployment.
- For teams that would want to start by creating standard GCR in the CI/CD Pipeline, there are functions to create a GCR based on a json payload. Even if you provide an empty JSON the JSL functions will fill as many required fields as possible with the information of the Jenkins build, Jira and SonarQube.
- For teams that accomplish all the required prerequisites and have been approved to use an automatized GCR, the GCR will be automatically created and approved with all kinds of test evidence and mandatory fields. The team only needs to take care of the production deployment result to update the “In progress” GCR status at the end. Because the DevOps GCR strategy is based on tags, there is a function to search and read these special tags.

In the 1st and 2nd cases, the Jenkins file would need to include these variables in the environment section:

```
environment {  
  
    /* SERVICE NOW Integration */  
    SERVICE_NOW_CREDENTIALS = 'your-non-interactive-account'  
    SERVICE_NOW_HOST = 'mysupportdesk.service-now.com'
```

Provided credentials are of kind Username with password and the account should be a non-interactive account in Service Now.

For DevOps GCR cases, the Jenkins file should use our specific account and all these variables at least:

```
environment {  
  
    /* SERVICE NOW Integration */  
    SERVICE_NOW_CREDENTIALS = 'devops-servicenow-credential'  
  
    SERVICE_NOW_HOST = 'mysupportdesk.service-now.com'  
  
    DEVOPS_GCR_PREPROD_BRANCH = 'your-preproduction-branch'  
  
    DEVOPS_GCR_PREPROD_PAYLOAD = '{"u_implementers": "CUID1,CUID2,CUID3", "u_testers": "CUID4"}'
```

The Adoption Stats stage will detect that the GCR preprod branch has been specified and will validate that the PROJECT_MAL is correct and allowed to use this kind of GCR before creating it.

The DEVOPS_GCR_PREPROD_PAYLOAD variable is not mandatory but is especially useful for teams that want to specify some CUIDs as implementers or testers of the DevOps GCR, or want to fix any other change request field.

The DevOps GCR workflow will create some tags with the GCR Id for Release Candidate, Promote, and Deployment. Depending on your branch strategy teams can specify tag patterns in their stages to identify all cases and run the pipeline in Release Candidate, Promote or Deploy to production mode.

In the more complex scenario, when the team's production branch is not equal to the preproduction one, there are two functions that can help to manage such scenario: jslGitHubPRCreate function to create a PR against the production branch and jslGitHubPRMerge to merge it automatically.

The shared libraries use a toolset repository for ServiceNow integration.

There are pipeline samples at: <https://github.com/CenturyLink/jsl-snow/tree/master/cicd/samples>

Best Practices

Multibranch pipeline jobs

Jenkins UI based jobs are the source of most of Jenkins management overhead. To reduce the overhead multibranch pipeline jobs can be of help. This kind of jobs allow to manage all the development lifecycle phases from a single UI job and a single Jenkinsfile, scanning branches, pull requests and tags of a single repository to create jobs dynamically.

Following the rule of thumb of one repository one CI/CD pipeline combined with multibranch pipeline jobs is the key to reduce Jenkins UI based management overhead.

Use dockerized agents

The use of project owned custom agents is discouraged. To keep the workflows as LEAN as possible, the use of dockerized agents running on top of a pool of shared agents is recommended. This best practice provides the following benefits:

- Reduce the agent management overhead.
- Reduce downtimes of agents.
- Empowers teams to take control of agent configuration through dockerfiles.
- Improves the throughput of the resources used.

Keep the pipelines clean

The Jenkinsfiles should be readable. Keeping them clean reduces the time needed to know what a pipeline does. The ideal scenario is based in the rule of thumb of “one stage one action”.

Reuse code

If one step is being issued on more than one stage is a candidate to be included in a Jenkins shared library.

Avoid writing functions in Jenkinsfiles

If a function is written inside a Jenkinsfile it cannot be reused in other Jenkinsfile. Avoiding function definition in Jenkinsfiles is considered a best practice as it will improve code reusability.

Avoid hardcoded credentials in Jenkinsfiles

Credentials should not be 'hardcoded' in Jenkinsfiles. If they are pipeline related, they are expected to be stored safely as Jenkins credentials. If they are application related, they are expected to be stored as Jenkins credentials or using git crypt. They never should be readable in an environment variable in a Jenkinsfile.

Avoid tampering with pipeline results

Jenkins provides an exception-based control in declarative pipelines. Once a stage fails, the default behavior is jump to the post section. Once a stage is considered unstable (through a plugin use), the default behavior is to continue the flow. This default behaviors are valid for most use cases, but they can be modified in the options section of a declarative pipeline.

Tampering with pipeline results to change the stage status to hide errors in what is being executed in a stage, is not a best practice. Retrying a stage to mask an error, is not a best practice (i.e. a software that only builds at the third retry).

Manual input parameters and automation do not match

Manual input parameters break automated workflows. They should be avoided. If they cannot be avoided, the only way to include them in an automated workflow is to assign them a default value and add code to test if the run of the pipeline is manual or automatic and behave accordingly.

Avoid building “Lumen third-party” code

Sometimes there is a “third party” dependency that is provided as a source code and needs to be built before using it in a pipeline.

Rebuilding software is a waste of time, and all “Lumen third-party” (i.e. Lumen test frameworks) software should be gathered from the corporate Nexus server instead of being rebuilt in every test execution.

A pipeline is as good as the underlying tools and infrastructure

A Jenkinsfile and its associated Jenkins Shared libraries can be a software masterpiece but if any of the tools used by it have performance issues or experiences frequent downtimes the CICD experience is not the optimum.

Implementing retries to cope with these issues can be a workaround but never a solution.

Use CLI's

Interacting with other services in pipelines is a common use cases in pipeline stages. Dealing with services through API's looks like the default solution (programmatically or using curls), but when an intensive use of a service is needed, and there are no resources to cope with API changes nor to develop code-based solution, using the vendor CLI to interact with the service can be considered a best practice.

Do not blocks Agents nor Executors

Agents are scarce resource. Stay lean, do not use/block if it is not necessary:

- Always add a timeout to your pipeline. As a rule of thumb, any pipeline should run in less than an hour.
- There is no need to block an agent/executor for long periods. In case of launching tests or deploy something that needs an user input, let the pipeline timeout in a reasonable time (15 minutes) and share the link to the pipeline so the authorizer can restart the timeout pipeline from that stage. There is no case in which blocking a user/agent for 48 hours waiting for an user input is justified.

Keep the traces clean

The feedback in traces must be clear and concise. When using sh() or bat() blocks, the debug output should not be shown, to avoid an excessive feedback in which real errors can hide.

Use replay to debug

In case of debugging a failing pipeline, the using the replay functionality to add more traces or uncomment them in the existing Jenkinsfile or Jenkins shared library function.

More information can be found at Jenkins official documentation:

<https://www.jenkins.io/doc/book/pipeline/running-pipelines/#replay>

Keep the workspaces clean

During stages files are generated. Some of them are going to be used in later stages and those should be stored in corporate Nexus (i.e. software built in the pipeline), stashed for next stages (i.e. a coverage report to be handled to SonarQube), or archived for accountability (i.e. test reports). But the rest of them, especially files including sensitive information, must be removed.

Keep Jenkins clean

The Unified Lumen Jenkins is shared by many teams. Please try to keep the environment clean:

- Create a folder to store all the jobs belonging to your project MAL. Please do not create jobs in the Jenkins root.
- Inside the MAL folder, store your jobs in sub folders. As a rule of thumb, each repository CICD pipeline based on multibranch pipeline plugin should be stored in its own folder.
- Store credentials at folder level. Remember that credentials at root level can be accessed by any user. Please keep the sensitive information private.
- Use folder level project security to grant access to your folder only to the expected people.
- Never store credentials at job level. That way cannot be shared between different jobs

Application Lifecycle Management Recommendations

This Section Covers recommendations, mandatory requirements and best practices provided by different teams across Lumen organization to guarantee a high standard in terms of quality, functionality, performance, and security across different phases of the ALM.

DevOps Standards might enforce in the future, if not already, the following statements as part of the CI/CD process.

Security

For more information or assistance on regards to the following recommendations please contact security at AskSecurity@lumen.com

Logging Requirements

From [Cyber Security Incident Response Team \(CIRT\)](#) has been notified that all applications need to implement the following logging requirements as part of their ALM.

For more information please check [Audit Logging policies](#).

Purpose

The purpose of log and record data is that the systems can be appropriately monitored in order to detect indications of security problems and guarantee consistency across all the organization's portfolio. With the implementation of this requirements we can consume, correlate, analyze and manage all the systems at glance and being able to respond accordingly in a incident or audit.

Please Note that all data generated is subject of the [data protection policy](#) which is mandatory to all statements across the company.

Below you will find the request and example, please note that not all might be applicable to your case.

Web application logs (IIS, Apache,nginx, Tomcat, etc.)

The below logs are used to identify access attempts (successes and failures) against the web application resource, errors in the application logs, and potentially suspicious activity. These logs should include IP address and timestamp information. Examples include, but are not limited to:

- HTTP access and error logs
- Web server logs
- Authentication log

OS and additional logs

The below logs are used to identify remote login activity, login successes and failures, user account changes, service failures, and privilege escalation attempts. These logs should include IP address and timestamp information.

- /var/log:
 - messages
 - auth.log
 - secure
 - lastlog
 - btmp
 - wtmp
 - utmp
 - faillog
 - syslog
 - httpd
- Sudo logs
- System and Security logs
- Application

Database

Logging for data access - logins, logouts, reads and writes against data Used to identify data access through usernames and timestamps.

Repository Management

As part of the usage of the Standardized Pipeline, it is mandatory to use the unified tools even for the repositories in Nexus. Below you should be able to find information on regards to how to configure the different repositories.

Maven

Current repositories (which some of them are proxied to public ones) can be integrated with any of your tools are:

- 1- Public Group (<https://nexusprod.corp.intranet:8443/repository/maven-public/>) which includes
 - a. Maven Central Proxy (<https://nexusprod.corp.intranet:8443/repository/maven-central/>)
 - b. Internal Release (<https://nexusprod.corp.intranet:8443/repository/maven-releases/>)
 - c. Internal Snapshots (<https://nexusprod.corp.intranet:8443/repository/maven-snapshots/>)
 - d. Spring Milestone (<https://nexusprod.corp.intranet:8443/repository/SpringMaven/>)
 - e. Spring Snapshot (<https://nexusprod.corp.intranet:8443/repository/SpringSnapshot/>)
- 2- Redhat (<https://nexusprod.corp.intranet:8443/repository/redhat/>)
- 3- Jboss-public (<https://nexusprod.corp.intranet:8443/repository/jboss-public/>)
- 4- Apache (<https://nexusprod.corp.intranet:8443/repository/apache/>)
- 5- Internal 3rdParty (<https://nexusprod.corp.intranet:8443/repository/3rdParty/>)

It is important to note that the 3rdParty repository is designed to publish those libraries that are not publicly available and that belong to a non-Lumen company e.g.: oracle libs

In order to upload a 3rdParty Library we recommend the use of maven deploy-file goal
<https://support.sonatype.com/hc/en-us/articles/115006744008-How-can-I-programmatically-upload-files-into-Nexus-3->

Please be sure to upload both jar and pom to the repository otherwise your project might have issues with dependencies. Please check above chapters on how to get access to the different tools.

Docker

Docker repositories are not only intended to be used for storing internal containers. In order to avoid restrictions with docker.io, it is needed to include the layers of public containers from Nexus. The reason behind is that Nexus store a cache of the images requested to docker.io and therefore reduce the frequency to request a public image and download them. Use nexusprod.corp.intranet:4567 as registry for your convenience e.g:

FROM nexusprod.corp.intranet:4567/ubuntu:focal