java maven

Last updated by | Salma Rais | Mar 6, 2024 at 3:23 PM GMT+5:30

Home | Pipeline Templates | Pipeline template Java Maven

Prerequisites

Before using any **Pipeline template** we assume that the application is properly onboarded and all security tooling: **Twistlock, Fortify, NexusIQ, SonarQube**.

Contents

Introduction

The Maven Pipeline Template(YAML Based) allows teams to on-board quickly with their Continuous Integration(CI) Pipeline. This all in one package which would help to Test, Analyse Code Quality, Security Scanning according to ABN AMRO best practices defined. To see how it works feel free to look at the source code in PIPE's central template repository. More information on standards and guideline for Java Development can be found on confluence .

At Present we are supporting JDK8, JDK11 & JDK17 versions. JDK11 is the default. Check template parameters.

Watch the below short video guides on

- Click here for the introduction on PITA Templates and Onboarding.
- Click here for Java-maven pipeline creation demo.
- Click here for how to solve common pipeline issues and setup of sonar and nexuslc scan in local.

Pipeline flow

Once you trigger the JavaMaven CI pipeline template the following steps (blocks) will be performed:

name	type	description
Versioning		Determines the version off the build, includes Maven Release on certain conditions, please see the chapter Building releases.
maven package and test		Junit test execution and maven build
SonarQube		Static code quality scan
LocalIntegrationtestsWithVirtualizedServices		Run your tests as part of the buiild i.e local integration tests with virtualized or dependent services running in other containers.
NexusIQ		Third party library vulnerability scanning
Fortify		Static code scanning for security issues (only on master or main or release_branches)
RESC		Secret scan using Repository Scanner(RESC) tool
Fortify		Static code scanning for security issues (only on master or main or release_branches)
Publish		Publishing artifact to Nexus and optionally to the pipeline

Getting started

In order to successfully run Java Maven pipeline template, there are few mandatory steps required:

- Have proper naming in place
- Inherit from Master POM

```
<project>
    <groupId>com.abnamro.pipe</groupId>
    <artifactId>Abnamro_Pipe_JavaMaven</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <description>Training material for Java for Testers</description>
    <parent>
        <groupId>com.abnamro.coesd</groupId>
        <artifactId>master-pom</artifactId>
        <version>X.X.X</version> <!-- Check for latest -->
    </parent>
</project>
```

- Create `azure-pipelines.yml` file.

Just follow the instructions from **Basic configuration** section.

Please review out our knowledge articles with based on frequently asked questions:

- https://aabsiampr.service-now.com/myit

Should you need any additional support or assistance do not hesitate to create ServiceNow request in **CICD Pipelines** service:

- https://aabsiampr.service-now.com/myit?id=myit_support_msg

If you're interested in more sophisticated setup please review **Advanced configuration** section.

Basic configuration

Start testing with the pipeline *on a branch*, the pipeline behaves bases on which branch is being build, for more info on this see the chapter 'branch behaviour'

An example of the Reference Java Maven pipeline can be found in the following repository: spex-java-maven

This pipeline template is tucked away in a 'Template' to make pipelines easier to use. This template in turn calls different 'building blocks' which contain the actual pipeline functionality.

Project structure

Pipeline template configuration follows common used project structure recommendations:

```
repository
  ├── README.md
  ├── src/
  │   └── main/
  │   └── test/
  ├── pom.xml
  ├── azure-pipelines.yml
  ├── sonar-project.properties # add this ONLY when you have dependency on java-8 and you can not migrate t
  └── .gitignore
```

Multimodule projects have a slightly different structure, but a pom.xml in the root directory is mandatory. For standards and guidelines for Java development please refer to this page .

YAML template

Copy the following to the azure-pipelines.yml file

```
# ---- Define a trigger if needed
trigger:
  branches:
    include:
      - master # You need this to automatically trigger a release build on commit to master
      - main # You need this to automatically trigger a release build on commit to main

  # PIPE team shares the read-only variable group: `Abnamro.Coesd.VariableGroup.GlobalVars`
  # this variable group needs to be included in every pipeline.
  # These are referenced in your pipeline yaml.
variables:
  - group: Abnamro.Coesd.VariableGroup.GlobalVars
  - group: TeamVars.<Application Acronym> # Replace with your Application ID e.g   TeamVars.SPEX

# ---- Target of which the templates are loaded, feel free to view content in that repository.
resources:
  repositories:
    - repository: templates
      type: git
      name: GRD0001045/pita-pipeline-templates

  containers:  # add this ONLY when you have dependency on java-8 and you can not migrate to java-11. java-
    - container: java-11
      image: 'pita/aab-build-maven:java-11'
      endpoint: 'nx3-build-images-sc'
stages:
  - stage: CI
    jobs:
      - template: flows/java-maven.yml@templates
```

SonarQubeScan (Dependency on JDK17 and higher)

Note that the below sonar-project.properties are NOT NEEDED if you are using java_version: '11' or '17' (Note that '11' is default option for the java-maven.yml flow). The recommendation is to migrate to java-17, if it is not possible to migrate for any good reasons, the below steps has to be followed to setup the build.

This is only needed when you use java_version '1.8' or '1.11'

1. Ensure that you are adding the below container resource to your main yaml from where java-maven.yml flow is called. See the YAML example given above.

```
containers:  # add this ONLY when you have dependency on java-8 and you can not migrate to java-17. java-1
  - container: java-17
    image: 'pita/aab-build-maven:java-17'
    endpoint: 'nx3-build-images-sc
```

2. Include sonar-project.properties file at the root of the project with the below configuration

```
sonar.sourceEncoding=UTF-8
sonar.projectKey=<sonar project key> # is nothing but groupid:artifactname , Alternatively you can copy pas
sonar.groupid=<group id of your application>

sonar.modules=<module name1>,<module name2>,<module name3> so on # comma seperated module names, this is no

sonar.sources=.flattened-pom.xml,src/main/java
sonar.tests=src/test/java
sonar.java.binaries=target/classes

sonar.exclusions=**/generated/*
```

example1 - click here:

```
sonar.sourceEncoding=UTF-8
sonar.projectKey=com.abnamro.spex:java-maven
sonar.groupid=com.abnamro.spex

sonar.sources=.flattened-pom.xml,src/main/java
sonar.tests=src/test/java

sonar.java.binaries=target/classes
sonar.exclusions=**/generated/*
```

example2 (for multi module) - click here:

```
sonar.sourceEncoding=UTF-8
sonar.projectKey=com.abnamro.spex:java-maven-test
sonar.groupid=com.abnamro.spex
sonar.modules=spex-examples,spex-exercises,spex-homework

sonar.sources=.flattened-pom.xml,src/main/java
sonar.tests=src/test/java

sonar.java.binaries=target/classes
sonar.exclusions=**/generated/*
```

This will run the pipeline with all defaults. For all available parameters to influence pipeline behaviour please see template parameters.

Local integration tests With Virtualized Services running in containers

With build-unittests-containers.yml block, it is possible to run tests i.e local integration tests by virtualizing any dependent services by spinning up those services in containers. For more on how to include this in java-maven flow and how to use it as standalone block, refer this page

Advanced configuration

Versioning

By default, this pipeline will look at the POM to determine the version. There are two other methods of providing a version by providing the parameter `versioning` .

```
versioning: 'source' or 'generate' or 'use-published'
```

In which:

- **source** - Default behaviour, version will be read from POM.xml
- **generate** - The pipeline will auto-generate a version. Before choosing this option, first read documentation on "auto-versioning" to understand how to use it. Then follow instructions to set up your pom accordingly.
- **use-published** - The pipeline will attempt to download an artifact from the current run called `auto_version.json` this is a JSON file in the below format. Based on this file the version will be set. Follow instructions to set up your pom accordingly.

auto_version.json:

```
{
  "aab_application_version": "my-version",
  "aab_version_bumped": "true/false",
  "aab_is_release_version": "true/false"
}
```

Setup pom.xml

In case of `versioning: source` (=default) nothing special: the pom.xml needs to specify the "project.version" field.

For any other versioning option (e.g. `use-published` and `generate`) you have to prepare the pom.xml for so-called "Maven CI Friendly version". Read maven documentation for details: https://maven.apache.org/maven-ci-friendly.html

Summary:

- Set de version field in POM.xml to ' `${revision}` ' . Explanation .
- For multi-module projects: also update the parent version of the child modules .
- Add the flatten-plugin to your pom.xml. Explanation .
- Include the below line within your flatten-maven-plugin configuration tag

```
<flattenedPomFilename>flattened-pom.xml</flattenedPomFilename>
```

- Add `flattened-pom.xml` to your `.gitignore` file. ("flattened-pom.xml" is a file generated by the flatten-plugin that should not be part of the source code.)

Snapshots and "auto-versioning"

When using option `generate` the "auto-versioning" for maven produces `-SNAPSHOT` versions on non-master branches. This way artifacts from non-master branch end up in 'maven-snapshots' repository instead of 'maven-releases' repository. However, the "auto-versioning" still generates unique versions for each commit, even for snapshots.

For example: A version for the fifth commit in a branch from release 1.2.3 named 'feature/my-awesome-feature' will have the following version generated: `1.3.0-my-awesome-feature.5-SNAPSHOT`

Important: Switching from "source" to "generate"

When your project is using "source" until now, and you want to start using "generate" you need to do the following one time action to make sure the "auto-versioning" will continue with your latest release version:

Do this on or main or release_branches branch as first commit after a release has been created.

```
# make sure your workspace is clean
git pull
# change azure-pipeline.yml to set the parameter versioning: 'generate'
# change the pom.xml files as described above in section "Setup pom.xml"
git commit -am "Start using auto-versioning"
git tag -am "Manually tag release version" 1.2.3
# replace 1.2.3 with your new release version where you increase the patch level from the recent release.
git push --follow-tags
```

In case you have any branches open: merge these changes from master or main into your branches as usual.

Branch behaviour

Release branches

Release by default from the master or main branch. If you want to build a release from another branch please specify the branch in the release_branches parameter. For those parameters please see template parameters. Build trigger on all other branches will result in a snapshot.

> Note: You should only have one release branch. Please keep in mind that auto-versioning will look at the last tag of the BRANCH you are on. In case you change the current custom release branch to another branch, you need to manually tag the new release branch with the new release version on REPOSITORY level (across all branches). So let's say you have a custom release branch 'custom-release-branch' with the latest tag 2.0.0 and the release branch 'new-release-branch' you are switching to is BEHIND with it's tagging, e.g. new-release-branch was last tagged with 1.0.0, then run the following to ensure that git follows the tag for the new release based on the latest tag + increment:
>
> ```
> # make sure your workspace is clean
> git checkout <new-release-branch>
> git commit -am "Changing release branch"
> git tag -am "Manually tag release version" 2.0.1
> # replace 2.0.1 with your new release version where you increase the patch level from the latest release
> git push --follow-tags
> ```

Security tooling

- Fortify only builds on a release branch! (master or main branch by default) For faster feedback please install Fortify locally. (Why? See chapter 'Building releases'.)
- Sonarqube & NexusIQ can be skipped if you're not on a release branch (master or main branch by default). For those parameters please see template parameters.

Building releases

JUST COMMIT TO MASTER!! We advocate Trunk Based Development! Alternatively you can specify a release branch

This will:

- Remove -SNAPSHOT from the version specified in pom.xml.
- Commit this change on a 'tag'
- Checkout code from the tag
- Scan with all the QA tools
- Upload released artifact to Nexus

Important

It is your (team) responsibility to consciously set the SNAPSHOT version to the next release version applying semver before committing to master or main branch.

Accessing CI Pipeline Artifacts

The pipeline will push its artifacts to Nexus 2 plus Repository Manager. Your CD pipeline or other maven pipelines can consume the artifacts from Nexus Repository Manager.

By using parameters `publish_pipeline_artifact`, `publish_directory`, `published_artifact_name` you have the option to also publish the build artifact as a "pipeline artifact". See documentation on pipeline artifact

Example use case would be where you do maven a build followed by creating a docker image using docker flow.

Use Download Pipeline Artifact task to download the pipeline artifact that you chose to publish using the mentioned parameters.

Running mutation tests

A mutation test is a form of testing that firstly modifies your software in small ways and then executes your unit tests to verify they *fail*. For JVM applications Pitest provides this feature by means of a maven plugin. You simply add it t

Getting started with pitest

See Pitest maven quickstart for how to get started. In short, simply add the plugin to your maven build:

```
<build>
    <plugins>
        <!-- Other plugins go here...  -->
        <plugin>
            <groupId>org.pitest</groupId>
            <artifactId>pitest-maven</artifactId>
            <version>VERSION_GOES_HERE</version>
        </plugin>
    </plugins>
</build>
```

and then they can be executed by running:

```
mvn test-compile org.pitest:pitest-maven:mutationCoverage
```

This will generate mutations to your application and then run all your tests. The mutation test report is stored in `target/pit-reports/` , you can check this out to see how you can improve the mutation coverage of your tests.

To incorporate this in your Azure pipeline run, simply enable them by setting the `run_mutation_tests` parameter to `true` like so:

```yaml
- stage: AppBuild
  displayName: Build Application
  jobs:
    - template: flows/java-maven.yml@pita
      parameters:
        # other params ...
        run_mutation_tests: true
```

Pitest optimizations

Since running mutation tests is quite computationally intensive you might experience longer build times for your application. In the pipeline itself parallelization (and coming soon also caching of maven resources) should make the effective time lower. You can yourself also limit the number of mutations per test, and exclude certain classes for mutations (e.g. exclude the more intensive integration tests that require the full application context to run).

Alternatively, you could create a separate pipeline that just executes the mutation tests on a schedule (e.g. every night at 3 a.m.). For this you can use the template located at `blocks/maven/mutation-test.yml` .

YAML template parameters

The Java Maven pipeline template accepts the following parameters.

```
  - template: flows/java-maven.yml@templates
    parameters:
      java_version:                  string  # Optional; Default: '11'. Options: '1.8' | '11' | '17'
      versioning:                    string  # Optional; Default: 'source'. Options: 'generate' | 'source
      release_branches:              string  # Optional; Branch that will start a 'release build'. Defaul
      use_maven-group:               boolean # Optional; Default: false. By default the maven settingsfil
      maven_phase:                   string  # Optional; maven phase to be used in the maven build. Defau
      optional_maven_parameters:     string  # Optional; extra flags. For example '-X' for Maven debug
      resc_scan_location:            string  # Optional; Default: $(Build.SourcesDirectory) (or other spe
      run_fortify_scan_prerelease:   string  # Optional; Default: true. Options: 'true' | 'false'. Only p
      fortify_project_name:          string  # Optional; by default this is generated from application ac
      fortify_scan_location:         string  # Optional; Default: $(Build.SourcesDirectory)/**/*
      fortify_excludes:              string  # Optional; Default: "**/*.yml:**/*.txt:**/*.html:**/*.js"
      run_sonarqube_scan_prerelease: string  # Optional; Default: 'true'. Options: 'true' | 'false'. Only
      test_report_results:           string  # Optional; Unit test results file. Default: '**/surefire-re
      run_nexus_lc_scan_prerelease:  string  # Optional; Default: 'true'. Options: 'true' | 'false'. Only
      nexus_lc_name:                 string  # Optional; default derived from <application acronym>_<syst
      run_release:                   boolean # Optional; if there's no use case to do a Maven release set
      execute_publishing_prerelease: boolean # Optional; To choose if prerelease publish to NXRM is neede
      publish_pipeline_artifact:     string  # Optional; Default: 'false'. Options: 'true' | 'false'
      publish_directory:             string  # Optional; Specify the location of the artifacts to be publ
      published_artifact_name:       string  # Optional; Pipeline artifact name, see paragraph 3.2 for mo
      fetch_depth:                   number  # Optional; how many commits does the git checkout fetch, De
      run_unittests_with_containers: boolean # Optional; can be set to true in combination with the below
      containers:                    list    # Required; provide the list of the containers which you wan
      docker_compose_file:           string  # Optional; should be at the root level of the repository. D
      container_to_wait_for:         string  # Required; name of the container that should wait for its e
      wait_for_log_message_in_container: string # Required; message that will appear in the console while
      run_dependency_report:         boolean # Optional; Default: 'false'. Options: 'true' | 'false'. Ena
      dependency_group_ids:          object  # Optional; If dependency reporting is enabled (run_dependen
      code_coverage_report:          boolean # Optional; If code coverage report is enabled i.e 'true' th
      code_coverage_tool:            string  # Optional; But required when code_coverage_report is true (
      code_coverage_summaryFileLocation: string # Optional; Default: '$(System.DefaultWorkingDirectory)/t
      code_coverage_reportDirectory: string  # Optional; Default: '$(System.DefaultWorkingDirectory)/targ
      maven_cache:                   boolean # Optional; Default: 'false'. Options: 'true' | 'false'. Ena
```

Dependency Report

This feature allows you to report the available versions of a specific group of dependencies.

```
- template: flows/java-maven.yml@templates
  parameters:
    publish_pipeline_artifact: 'true' # (optional) 'true' | 'false' defaults to false
    published_artifact_name: 'MavenCI' # (optional) Pipeline artifact name, see paragraph 3.2 for more info
    versioning: generate
    java_version: '1.8'
    run_sonarqube_scan_prerelease: 'false'
    run_dependency_report: true
    dependency_group_ids:
      - om.github.tomakehurst
      - com.tngtech.junit.dataprovider
      - io.rest-assured
      - org.xerial
```

- How to find nexus_lc_name
- Java-maven pipeline creation demo
- How to solve common pipeline issues and setup of sonar and nexuslc scan in local
- For more details on run_unittests_with_containers - Invoking build, 'unittests with containers' and sonar scan

Recommendations

Configure Retention Policy

It is recommended to define build retention policies for build artifacts.

Set debug mode

In case you are not confident your pipeline is going to run successfully, it is recommended to it in debug mode. This makes it easier to debug issues when they occur. To enable this, you can either:

1. Check the 'Enable system diagnostics' option in the bottom of the popup you get when triggering your pipeline manually.
2. Adding the variables System.Debug to your azure pipeline yaml file.

```
variables:
- name: System.Debug
  value: true
```

Check the Microsoft documentation for more details

FAQ / Common errors

Fortify error on METADATA.xml file

The error will look like this:

```
Approval Required: The version of the external metadata file "ExternalMetadata/externalmetadata.xml" is mor
```

To solve please follow these steps **as a fortify admin**:

- Go to your Fortify project
- On the top right there, a little below the 'search area' there is button called "Profile", push this
- Go to the third tab "Processing Rules"
- **Uncheck** the box with the title: "Check external metadata file versions in scan against versions on server."

Git checkout taking very long

To fetch less of the git history provide the 'fetch_depth' parameter. Check template parameters. on how to set it.