THE JFROG PLATFORM
CI/CD FROM CODE TO EDGE WITH JFROG PIPELINES
INTRODUCING THE JFROG PLATFORM 2
   JFROG PLATFORM ENTERPRISE+ 3
   THE ESSENTIALS OF ENTERPRISE DEVOPS 4
CI/CD WALK-THROUGH 5
   GETTING STARTED WITH PIPELINES 6
   PIPELINE 1: GRADLE BUILD 8
   PIPELINE 2: NPM BUILD 17
   PIPELINE 3: DISTRIBUTION 24
CONCLUSION 28
INTRODUCING THE JFROG PLATFORM

The JFrog Platform empowers growing companies to develop and distribute software quickly, safely, and frequently to meet the demands of today's competitive landscape. The end-to-end DevOps platform, powered by the industry-leading binary repository manager JFrog Artifactory, provides the key tools that can modernize the software development lifecycle (SDLC) and transform the way teams create, manage, secure and deploy applications at enterprise scale. The JFrog Platform enables DevOps automation best practices through tools for artifact management, security and vulnerability protection, CI/CD orchestration, analytics and release distribution.
JFROG PLATFORM ENTERPRISE+

The Enterprise+ edition of the JFrog Platform meets the demanding needs of large organizations to support many development teams and distribute software releases to an unlimited number of endpoints across the globe.

Enterprise+ includes the full complement of solutions that compose the JFrog Platform:

**JFROG ARTIFACTORY** is the industry-leading universal artifact repository manager that powers the JFrog Platform. Through REST API’s and a CLI, Artifactory readily integrates with all major CI/CD and DevOps tools for an end-to-end, automated solution to track artifacts from development to production.

**JFROG XRAY** provides universal artifact analysis, increasing visibility and performance of your software components by recursively scanning all layers of your organization's binary packages to provide radical transparency and unparalleled insight into your software architecture.

JFrog Distribution empowers DevOps teams to distribute and continuously update remote locations with release-ready binaries.

**JFROG ARTIFACTORY EDGE** accelerates and provides control of release-ready binary distribution through a secure distributed network and edge nodes.

**JFROG MISSION CONTROL** is your DevOps dashboard solution for managing multiple services of Artifactory, Xray, Edge and Distribution.

**JFROG ACCESS** with Federation provides governance to the distribution of artifacts by managing releases, permissions and access levels.

**JFROG PIPELINES** helps automate the non-human part of the whole software development process with continuous integration and empowers teams to implement the technical aspects of continuous delivery.
THE ESSENTIALS OF ENTERPRISE DEVOPS

This walk-through will illustrate how these essential building blocks of successful enterprise DevOps are accomplished in the JFrog Platform.

UNIVERSAL ARTIFACT MANAGEMENT

JFrog Artifactory is a universal artifact repository manager, supporting 30 package types plus generic binaries to act as a “single source of truth” for all binaries. Artifactory annotates artifacts with metadata, to provide full traceability of builds.

SECURITY VULNERABILITIES AND LICENSE COMPLIANCE

JFrog Xray provides deep integration with Artifactory repository management to perform deep recursive software component analysis (SCA) scanning on all designated repositories, and alert for dangerous vulnerabilities in open source dependencies. Xray can also screen builds to make sure licenses are in compliance with policy and regulatory requirements.

CI/CD AUTOMATION

JFrog Pipelines is a powerful, enterprise-ready automation tool that is naturally integrated with the core solutions of the JFrog Platform. Pipelines is a truly cloud native solution that provides central management of your automated workflows, to help scale your enterprise-level CI/CD with elastic, concurrent execution.

DISTRIBUTION TO THE EDGE

JFrog Distribution empowers teams to release multi-component applications securely as signed bundles, and distribute them to JFrog Edge nodes with validation for delivery to end user devices.
CI/CD WALKTHROUGH

This walk-through will guide you through an example JFrog Platform workflow using JFrog Pipelines to demonstrate how it can automate your CI/CD processes from code to production delivery. We'll build a containerized microservice with an example Java and NPM two-tier web application which requires third party dependencies and Docker images used for deployment. We'll package those images into a release bundle, then securely distribute it to an edge node where it can be deployed to remote Kubernetes clusters.

Our CI/CD walk-through consists of a sequence of three pipelines, which can be seen in the My Pipelines view of the JFrog Platform.

Together, these pipelines build a multi-component web application for distribution:

1. “tutorial-gradle-build”
   Creates a java web app which displays some images.

2. “tutorial-npm-build”
   Builds an NPM user interface that connects to the Java web app.

3. “tutorial-distribution”
   Creates a Release Bundle that holds the 2 previously created applications

We'll walk through each one of the example pipelines and do the following:

- **Review**
  A general overview of the example

- **Understand**
  The pipeline implementation and how to use the platform

- **Run**
  Execute the pipeline and review the results
Before any of our example pipelines can be run, some key elements of JFrog Pipelines must be understood and prepared by a JFrog Platform user with administrator or resource manager permissions:

**BUILD NODE POOL**
Build nodes are the external VMs where your pipeline steps will execute. VMs of the same operating system type are grouped in node pools, from which Pipelines distributes build nodes as pipelines need them to run. A pool can be composed of a fixed number of static build nodes, or can be configured to dynamically create and destroy build nodes in the cloud or in Kubernetes as needed.

**INTEGRATIONS**
Pipelines comes with the right variety of out-of-the-box integrations for the tools you’re likely to use most, so connecting to services is a snap. Adding an integration is often just giving it a friendly name, providing an API endpoint, and entering user credentials. Integrations ready to connect include GitHub, Bitbucket, Docker, Kubernetes, and Slack, as well as cloud services like AWS, GCP, and Azure.
PIPELINE SOURCES
The Pipelines DSL files that define your pipelines are stored in Git source control repositories, which you must connect to Pipelines as a pipeline source. Pipelines syncs the DSL files from all pipeline sources and loads them into My Pipelines.
Review
The first pipeline pulls a Gradle project from GitHub and builds it using the GradleBuild native step. It then uses the DockerBuild step to create a Docker image for this app and performs a DockerPush step to store the image in the local Docker repository in Artifactory. An Xray scan of the build assures there are no critical vulnerabilities or license compliance issues.

We can see the pipeline diagram in the Run History view for tutorial_gradle_build.

Understand
Let’s take a look at the YAML-based Pipelines DSL. You can click the “View YAML” button to see the implementation’s YAML file.
You can switch between the Pipeline and Resources tabs to view the YAML for each set of definitions.

The first step uses the GradleBuild native step to build the Gradle application:

```
  pipelines:
    - name: tutorial_gradle_build
      configuration:
        environmentVariables:
          readOnly:
            Version: 1.1-$run_number
      steps:
        - name: gradle_build_app
          type: GradleBuild
          configuration:
            runtime: image
            custom:
              # The docker image is used to support a java 8 image for the gradle build
              name: docker.bintray.io/jfrog/pipelines-8j
              tag: "8"
              gradleCommand: clean artifactFactoryPublish -D buildProfile -->stacktrace
              sourceLocation: tutorial/step3-create-gradle-app
              configFileLocation: .
              configFileName: jfrog-gradle.yml
              autoPublishBuildInfo: true
              inputResources:
                - name: tutorial_gitRepo_code
              integrations:
                - name: artifactory
              outputResources:
                - name: tutorial_gradleBuildInfo

  resources:
    - name: tutorial_gitRepo_code
      type: GitRepo
      configuration:
        path: repositories/project-examples
        gitProvider: sys\_github
        branches:
          include: eplus-v2-06biters
        #---------------------------------- FileSpec ----------------------------------
        - name: tutorial_gradle_fileSpec
          type: FileSpec
          configuration:
            sourceArtifactory: artifactory
            pattern: gradle-demo-dev-local/con/[jfrog/frogs\_\~\_1.0-$\(run_number\)\_frogs\_\~\_1.0-$\(run_number\)].jar
            target: gradle\_results/
        #---------------------------------- FileSpec ----------------------------------
        - name: tutorial_npm_fileSpec
          type: FileSpec
          configuration:
            sourceArtifactory: artifactory
            pattern: npm-demo/frogs\~\_1.2.$\(run_number\).tgz
            target: npm\_results/
        #---------------------------------- Gradle BuildInfo ----------------------------------
        - name: tutorial_gradleBuildInfo
          type: BuildInfo
          configuration:
            sourceArtifactory: artifactory
            buildName: tutorial_gradle\_build
            buildNumber: $\(run_number\)
```
• We also specify the Artifactory integration named “artifactory”

The second step creates a Docker image of the Gradle application:

```yaml
- name: gradle_docker_build
  type: DockerBuild
  configuration:
    dockerfileLocation: tutorial/step1-create-gradle-app
    dockerfileName: Dockerfile
    dockerImageName: $(ArtIF)/docker-demo-dev-local/gradle-app
    dockerImageTag: $(Version)
  inputSteps:
    - name: gradle_build_app
  inputResources:
    - name: tutorial_gitRepo_code
      trigger: false
    - name: tutorial_gradle_fileSpec
      trigger: false
    - name: tutorial_gradleBuildInfo
      trigger: false
  integrations:
    - name: artifactory
  execution:
    onStart: # export artifactory id from internal build env var to tag the docker image
      - export ArtIF=$(echo $artifactory_url | awk -F '/' '{print $3}')
```

• It will use the Docker file from the relevant location to generate the Docker image with specific name and tag based on the value of the environment variables that were configured on the pipeline level

• It will use the Git repo input resource for finding the Dockerfile

• It will use the BuildInfo input resource from the previous step

• It will also use an additional input resource for downloading the Gradle-created artifact into the workspace of the running step, to be used as part of the Docker build

The next step pushes the Docker image to the target Docker repository in Artifactory:

```yaml
- name: gradle_docker_push
  type: DockerPush
  configuration:
    targetRepository: docker-demo-dev-local
    autoPublishBuildInfo: true
    integrations:
      - name: artifactory
    inputSteps:
      - name: gradle_docker_build
    outputResources:
      - name: tutorial_docker_gradleBuild_Info
```
• It will be triggered by the previous gradle_docker_build step
• It will publish the updated BuildInfo

The following step is scanning the build Docker image using Xray, and generating a “scanned” build info

```json
- name: gradle_docker_scan
type: XrayScan
configuration:
  inputResources:
    - name: tutorial_docker_gradleBuild_Info
trigger: true
outputResources:
  - name: tutorial_scanned_gradle_dockerBuild_Info
```

The last step will promote the build to its “final” phase after all steps were completed

```json
- name: npm_docker_promote
type: PromoteBuild
configuration:
  targetRepository: docker-demo-prod-local
  includeDependencies: true
  status: Passed
  comment: Artifact passed Xray Scan
copy: false
inputResources:
  - name: tutorial_scanned_npm_dockerBuild_Info
    trigger: true
outputResources:
  - name: tutorial_final_docker_npmBuild_Info
```

Run
Let’s explore and run the pipeline that performs our Gradle build.

In the pipeline diagram, you can click on any resource to view its details
- You can click on any step to view its details, as well as to manually trigger it.
- Proceed to run the whole pipeline by clicking its first step and triggering it.
- You will notice an additional “run” was added to the Runs history.
To view the execution logs for detecting a problem or monitoring the execution progress, click View Logs:

Watch the log data for the different steps:

Verify that the job finished successfully:
When the job finishes successfully, switch to the Builds view in Artifactory and search for your build among the builds displayed there:

Drill down into your build and search for the specific run:

Click the desired run and see all the information that was captured as part of the published build info:
Select the published modules tab and review the modules and their dependencies:

Review the artifacts that were created.

Select the jar file and follow it to the Repository Browser:

You can see there all the relevant details on this created package:
Going back to the Build view, you can see other relevant tabs with important information.

Choose the Environment tab to see the information captured from the environment where the build ran:

Choose the Xray Data tab to see relevant information from Xray (on security vulnerabilities and license compliance issues within the created package, including some Xray actions that you can take):

Choose the Diff tab to compare your build results to a previous run:
PIPELINE 2: NPM BUILD

Review
This pipeline creates an NPM UI project that connects to the WAR file built by the previous pipeline.

Similar to the previous pipeline, we are using some native steps for the implementation: NpmBuild to compile the code, NpmPublish to create the BuildInfo and publish it to Artifactory, and again the DockerBuild and DockerPush steps.

We can see the pipeline diagram in the Run History view for tutorial_npm_build.

Understand
Before we run the pipeline, let’s take a look at its implementation YAML.

Similar to the previous pipeline, we are using some native steps for the implementation: NpmBuild to compile the code, NpmPublish to create the BuildInfo and publish it to Artifactory, and again the DockerBuild and DockerPush steps.
The first step uses the NpmBuild native step to prepare the npm environment for the build:

```
pipelines:
  - name: tutorial_npm_build
    configuration:
      environmentVariables:
        readOnly:
          - ArtIP: $[echo ${int_artifactory_url}|awk -F '/' '{print $3}']
          - Version: $[run_number]
    steps:
      - name: npm_build
        type: NpmBuild
        configuration:
          npmArgs: --no-progress --no-audit
          repositoryName: npm-demo
          sourceLocation: tutorial/step2-create-ui-pkg
          integrations:
            - name: artifactory
            - name: tutorial_gitRepo_code
        execution:
          onStart:
            - push $(resTutorial_gitRepo_code_resourcePath)/tutorial/step2-create-ui-pkg
            - npm version $(Version) --no-git-tag-version
            - popd
```

- We configure two environment variables on the pipeline level:
  - ArtIP - contains the value of the Artifactory IP that is being used in a later step for naming the Docker image we create
  - Version - contains a sequential number that is being used in a later step for tagging the Docker image we create

- In the first step we specify the relevant npm arguments, the repository name and the source location

- We use the “artifactory” integration

- We specify the relevant resource (the Git repository as the input resource) as in the previous pipeline

```
resources:
  - name: tutorial_gitRepos_code
    type: GitRepo
    configuration:
      path: roybensch/project-examples
      gitProvider: roybs_github
      branches:
        include: eplus-v2-orbitera
```
• The execution part that is added here is for populating the relevant version within the npm metadata (the version field in the packages.json file), so it will be well rendered in Artifactory's packages view.

The second step uses a generic Bash step where you can add all your logic in designated callback points, using bash commands:

```bash
- name: npm_compile
  type: Bash
  configuration:
    inputSteps:
      - name: npm_build
    integrations:
      - name: artifactory
    execution:
      onStart:
        - export tempStateLocation="$step_temp_dir/npmSourceState"
        # Restore file from previous step to current step
        - restore_run_files npmBuildInputGitRepo $stepStateLocation
        - pushd $tempStateLocation/tutorial/step2-create-ui-pkg
        - npm install shelljs
        - npm install
        # Add files to npm pipeline target
        - add_run_files $tempStateLocation/\.npmBuildInputGitRepo
        - popd
```

• The purpose of this step is compiling the npm package.

• We do that by using several bash commands.

• We also rely on the restore_run_files built-in function and the npmBuildInputGitRepo environment variable that is created implicitly on the previous step.

The third step will publish the npm package to Artifactory:

```yaml
- name: npm_publish
  type: NpmPublish
  configuration:
    repositoryName: npm-demo
    integrations:
      - name: artifactory
    inputSteps:
      - name: npm_compile
    execution:
      onStart:
        - export inputNpmBuildStepName="npm_build"
```
• We specify the repositoryName
• We specify the previous step as the trigger for this one
• The step uses the “artifactory” Artifactory integration

The fourth step will create a Docker image of the npm application:

```yaml
- name: npm_docker_build
  type: DockerBuild
  configuration:
    dockerFileLocation: tutorial/step2-create-ui-pkg
    dockerFileName: Dockerfile
    dockerImageName: $(ArtIF)/docker-demo-dev-local/npm-app
    dockerImageTag: ${Version}
    inputSteps:
      - name: npm_publish
    inputResources:
      - name: tutorial_gitRepo_code
      - name: tutorial_npm_fileSpec
    integrations:
      - name: artifactory
    execution:
      onStart:
        - pushd $res_tutorial_gitRepo_code_resourcePath/tutorial/step2-create-ui-pkg
        # Creating a Folder for the fileSpec Target
        - mkdir -p npm_results
        - popd
      onSuccess:
        - echo "Congrats The Docker image was build!"
```

• It will use the Docker file from the relevant location to generate the Docker image with specific name and tag based on the value of the environment variables that were configured on the pipeline level.
• It will use the Git repo input resource for finding the Dockerfile
• It will use the BuildInfo input resource from the previous step
• It will also use an additional input resource for downloading the npm created artifact into the workspace of the running step, to be used as part of the Docker build

```yaml
- name: tutorial_npm_fileSpec
  type: FileSpec
  configuration:
    sourceArtifactory: artifactory
    pattern: npm-demo/frogsui/-/frogsui-1.2.${run_number}.tgz
    target: npm_results/
```
• You can see that we are also using lifecycle callbacks (as in the previous generic Bash step) within the DockerBuild native step for additional operations. Here it is being used for creating the relevant folder that will contain the results of the step based on what is configured in the file spec.

The next step will push the Docker image to the target Docker repository in Artifactory

```
- name: npm_docker_push
  type: DockerPush
  configuration:
    targetRepository: docker-demo-dev-local
    autoPublishBuildInfo: true
    integrations:
      - name: artifactory
    inputSteps:
      - name: npm_docker_build
    outputResources:
      - name: tutorial_docker_npmBuild_Info
```

• It will be triggered by the previous **npm_docker_build** step
• It will publish the updated BuildInfo onto Artifactory

The following step will run an Xray scan on the previously created build:

```
- name: npm_docker_scan
  type: XrayScan
  configuration:
    inputResources:
      - name: tutorial_docker_npmBuild_Info
        trigger: true
    outputResources:
      - name: tutorial_scanned_npm_dockerBuild_Info
```

• It uses BuildInfo output resource of the previous step as input to this step
• It produces another BuildInfo as output to specify the “clean” status of it after Xray scanned it.
The last step will promote the build to its final stage:

```
- name: npm_docker_promote
  type: PromoteBuild
  configuration:
    targetRepository: docker-demo-prod-local
    includeDependencies: true
    status: Passed
    comment: Artifact passed Xray Scan
    copy: false
    inputResources:
      - name: tutorial_scanned_npm_dockerBuild_Info
        trigger: true
    outputResources:
      - name: tutorial_final_docker_npmBuild_Info
```

Run
Let's run the pipeline that performs our npm build and explore the results.

As in the previous pipeline, you can run the whole pipeline by clicking its first step and manually triggering it.

When the job has finished successfully, switch to the Builds view in Artifactory and search for your build among those displayed:

Drill down into your build and search for the specific run:
Click the desired run to view all the information that was captured as part of the published build info:

- Select the job that you ran and let’s review the build information.
- Review the artifacts that were created.
- Watch the Xray tab for the build and understand the relevant information.

Here you can see the license information on the built package.

- Follow one of the files to its view in Artifactory.
PIPMENT 3: DISTRIBUTION

Review
This pipeline creates a release bundle and distributes it to an Artifactory Edge node.

Again, we are using some native steps for the implementation: CreateReleaseBundle for creating and signing the release bundle and DistributeReleaseBundle for actually distributing the signed release bundle to its destinations.

We can see the pipeline diagram in the Run History view for tutorial_distribution.

Understand
Before we run the pipeline, let’s take a look at its implementation YAML.

- The first step uses the CreateReleaseBundle native step:

```yaml
steps:
  - name: create_releaseBundle
    type: CreateReleaseBundle
    configuration:
      releaseBundleName: tutorial_releaseBundle
      releaseBundleVersion: v1.2.$(run_number)
      dryRun: false
      sign: true
      storeAtSourceArtifactory: artifactory
      inputResources:
        - name: tutorial_final_docker_gradleBuild_Info
        - name: tutorial_final_docker_npmBuild_Info
      outputResources:
        - name: tutorial_new_releaseBundle
      description: RB of gradle and npm build
      releaseNotes:
        syntax: markdown
        content: |
          ## Heading
          * Bullet
          + Points
```
• In this step we specify the name and version for the release bundle
• The step will be triggered by the creation of one of its input resources which are the BuildInfos that are created in the previous pipelines
• The output of this pipeline is the new Release Bundle output resource

```yaml
- name: tutorial_new_releaseBundle
  type: ReleaseBundle
  configuration:
    sourceDistribution: distribution
    name: tutorial_new_releaseBundle
    version: v1.0.0
    isSigned: false

- name: tutorial_signed_releaseBundle
  type: ReleaseBundle
  configuration:
    sourceDistribution: distribution
    name: tutorial_signed_releaseBundle
    version: v1.0.0
```

• We are also using the “artifactory” integration to access our source Artifactory
• Note the “sign” flag. It is used to automatically sign the release bundle upon its creation.

▶ The second step will distribute the release bundle to its destination

```yaml
- name: distribute_releaseBundle
  type: DistributeReleaseBundle
  configuration:
    dryRun: false
    inputResources:
      - name: tutorial_new_releaseBundle
        trigger: true
      - name: tutorial_distribution_rule
```

• We rely on the input resource that was created on the previous step
• We specify that it is not a dry run
• We also use an additional input resource named Distribution_Rule:
• It uses the “distribution” integration

• It uses additional configurations for specifying the destinations

Run

Let’s run the pipeline that performs our distribution and explore the results.

As in the previous pipelines, you can run the whole pipeline by clicking its first step and manually triggering it.

► When the job finishes successfully, browse the “release-bundles” repository in your source Artifactory and review the different files in the release bundle:

► Now, switch to view the Artifactory Edge node and search for your bundle in the “Distribution” menu:
Drill down into your release bundle to view its versions:

And drill down into a specific version to view its manifest information.
Finally, drill down into the relevant repositories and watch the artifacts that arrived there during the distribution of the bundle.

Alternatively, you can view the Edge's packages view:
CONCLUSION

In this walk-through, we demonstrated using the JFrog Platform to create CI/CD pipelines to build an application, manage the artifacts, scan the artifacts for security vulnerabilities and license compliance, and distribute the application to production edge nodes.

Now that you have some basic understanding of the JFrog Platform, we encourage you to use it with your existing build tools, like Maven, Gradle, Ivy and Ant and work with packaging systems of other development platforms, like NuGet, RubyGems or NPM.

In addition to JFrog Pipelines, you can leverage Artifactory with other CI servers such as Jenkins, Hudson, TeamCity, and Bamboo.

Artifactory is also a great artifact repository for your DevOps needs, since it natively supports managing Docker images, Vagrant images, as well as RPM and Debian package managers.

Additional information about the topics in this paper:

JFrog Wiki: https://www.jfrog.com/confluence/
JFrog White-Papers: https://www.jfrog.com/support-service/whitepapers/
JFrog Webinars: https://www.jfrog.com/support-service/webinars/

Contact us at sales@jfrog.com for a technical session on the JFrog Platform