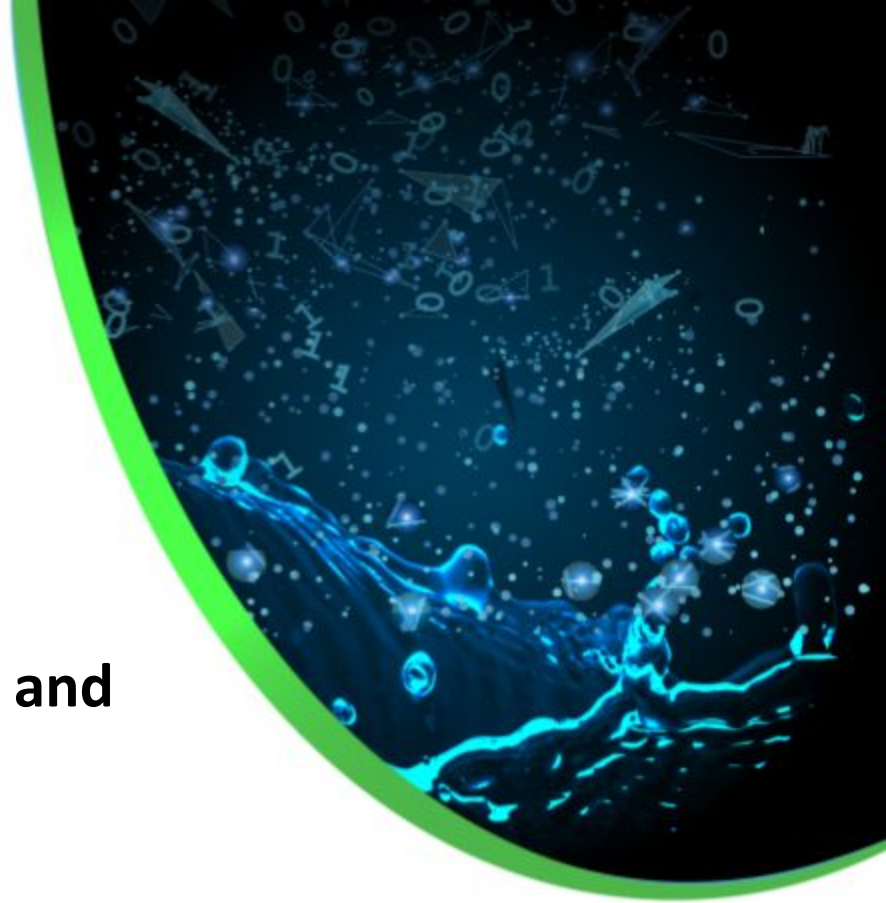


JFrog | CONAN C/C++ Package manager

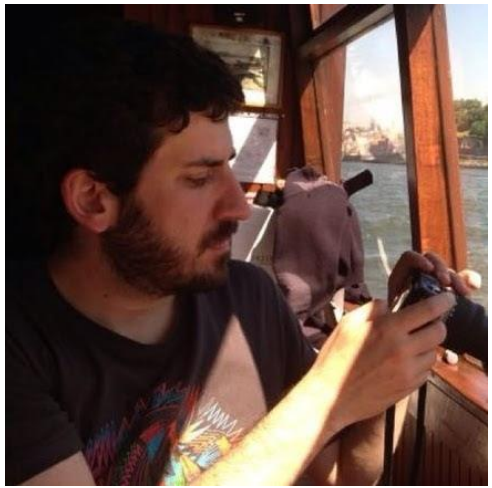
CI/CD in C/C++ Projects with Conan and Artifactory

Jerry Wiltse, Conan Developer @ JFrog
Carlos Zoido, Conan Developer @ JFrog

Coaches

Jerry Wiltse, Conan Developer

Carlos Zoido, Conan Developer



[Lab 0] Environment bootstrapping

- Artifactory
  - Create CI user ⟶ **conan/conan2020**
  - Create repositories: conan-tmp, conan-develop, conan-metadata
  - Create permissions
- Jenkins
  - Create pipelines for all libraries in Jenkins
- Conan Client
  - Preconfigured with conan remote, conan user, and custom profiles
  - All libraries and apps from the training pre-built and uploaded to Artifactory
- Custom Docker Images
  - Conan and GCC6 pre-installed

JFrog

[Lab 0] Environment bootstrapping

```
ssh conan@<orbitera-IP>
# Use password from orbitera

git clone https://github.com/conan-io/training-ci.git

cd training-ci/setup_jenkins/

./bootstrap.sh <artifactory_password> <jenkins_credential>
```

```
vm-testdriveinstance-1289-88142
-------------------


------ Outputs ------
Username:
admin

Artifactory URL:
http://34.68.29.120:8082/

Password:
WEs22tORIP

IP:
34.68.29.120

SSH Username:
conan

Jenkins Credential:
zmpoqUUj8z

Jenkins URL:
http://34.68.29.120:8080/
----------------------
```

Outline

- **Recap from Advanced Training**
- Introduction: The Story
- CI Workflow: Phase 1
- CI Workflow: Phase 2
- CI Workflow: Phase 3
- Artifactory: Build Info
- Artifactory: Promotion
- Summary
- Appendix

Recap from Advanced Training: Revisions

- 2 types of Revisions:
  - Recipe :
    - Id for tracking down any changes at the recipe level.
    - **RREV** = hash(sources, recipe, ...)
  - Package:
    - Id for tracking down any changes at the binary package level
    - **PREV** = hash(all the packaged files)
- Package ID:
  - Also a hash, but corresponds to hashing options, settings, and requirements

Recap from Advanced Training: Package ID modes

package_id = f(**settings**, **options**, **requirements**)

- **Settings**: operating systems, compilers, build types,…
- **Options**: shared, fPIC…
- **Requirements**: depending the package_id mode

Package ID modes for binary compatibility

- Can be more strict or more relaxed
- Choosing the right one is important, we will use **recipe_revision_mode** for our CI (quite strict)**, new revisions will affect package id's of dependents**

Recap from Advanced Training: Lockfiles

- A snapshot of a dependency graph at a given time.


```
{
 "version": "0.3",
 "profile_host":
"[settings]\narch=x86_64\narch_build=x86_64\nbuild_type=Release\ncompiler=gcc\ncompiler.libcxx=libstdc++11\ncompiler.version=6\nos=Linux\nos_build=Linux\n[options]\n[build_requ
ires]\n[env]\n",
 "graph_lock": {
   "nodes": {
     "0": {
       "options": "shared=False\nlibA:shared=False",
       "pref": "libB/1.0:ef4c743309e6cde478db59544c22fd8b98d6e0df",
       "path": "/var/lib/jenkins/libB/conanfile.py",
       "requires": [
         "1"
       ]
     },
     "1": {
       "options": "shared=False",
       "pref": "libA/1.0@mycompany/stable#d84a023833ae8b56bd8573d05962c937:57547fe65fffc300f05aa42ee64b3b02eeabb6d7#5bafcbf5f3eb1682dcac8e6810bf6e35"
     }
   }
 }
}
```

Recap from Advanced Training: Lockfiles use in CI

- Start by creating a lockfile, which builds with the **exact graph** of dependencies
- Use the lockfile to calculate the **build order** of a dependencies in the graph
- CI Jobs **update the initial lockfile** as the CI builds each library
- Lockfiles need to be stored somewhere:
  - In this training, we'll use conan-metadata repo on Artifactory
- Lockfiles can be used to copy groups of binaries between Conan repositories
  - In this training, we'll promote from conan-tmp to conan-develop

Recap from Advanced Training: Lockfiles cheatsheet

command	Input lockfile	Output
create / install / export / export-pkg	Yes (optional)	Update **lockfile**
graph lock	No	**lockfile** with the graph
graph build-order	Yes	**JSON** with build order
graph update-lock	Yes (requires 2 lockfiles)	Update oldest **lockfile**

Recap from Advanced Training: Two more things

- We will use SCM mode for our examples:
  - This means that commits of source code will generate new RREV

```
class LibB(ConanFile):

    scm = {"type": "git",
           "url": "https://github.com/conan-ci-cd-training/libB.git",
           "revision": "auto"}
```

- Will share the Conan configuration among developers with a git repo

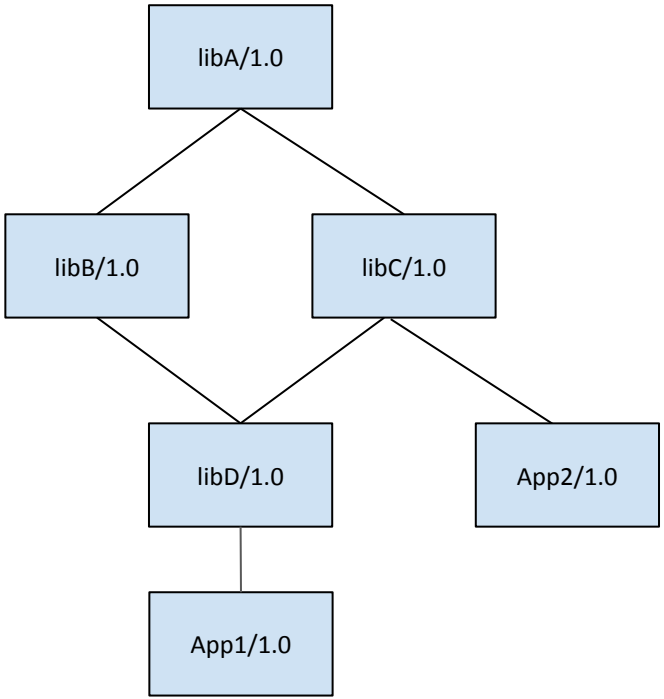
```
conan config install https://github.com/conan-ci-cd-training/settings.git
```

Outline

- Recap from Advanced Training
- **Introduction: The Story**
- CI Workflow: Phase 1
- CI Workflow: Phase 2
- CI Workflow: Phase 3
- Artifactory: Build Info
- Artifactory: Promotion
- Summary
- Appendix

The Story: Mycompany components

- 1 project providing 2 Apps which consumes libraries
- All libraries are internal to the project
- Some of them are shared by the Apps

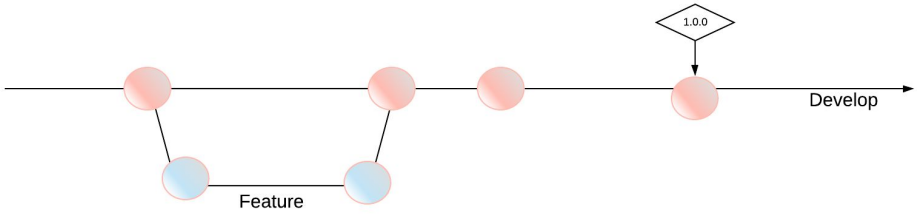


Conan Revisions Vs. Version Bumping

- Version bumping for breaking changes is a standard
- C and C++ are unique in definition of “breaking change”
- Unconditionally version bump for every change may be excessive
- Conan revisions provide an alternative
- Valid cases still exist for both revisions and version bumping
- Today’s focus is on using Conan Revisions, without version bumping
  - We’ll show one way to test consumers before promoting
  - If tests fail, version bump may be appropriate afterall
  - This strategy allows us to safely measure impact of a change

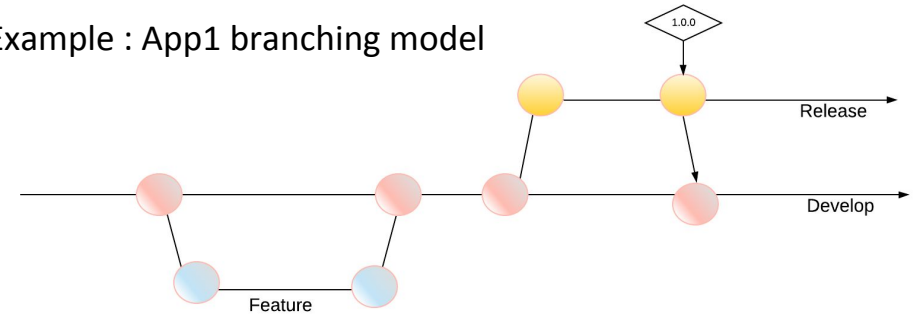
The Story: Code workflow

Example : libA branching model



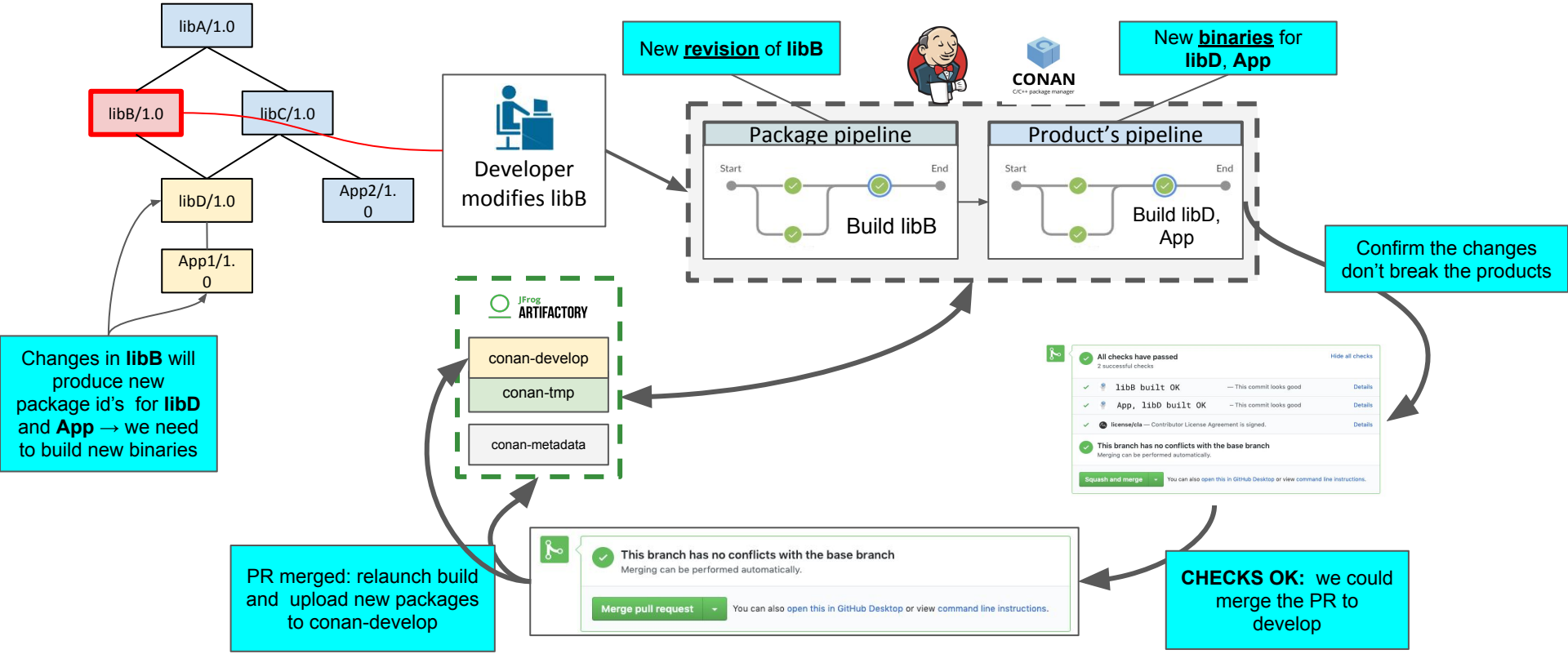
* libB, libC and libD follow the same flow and have their own code repository

Example : App1 branching model

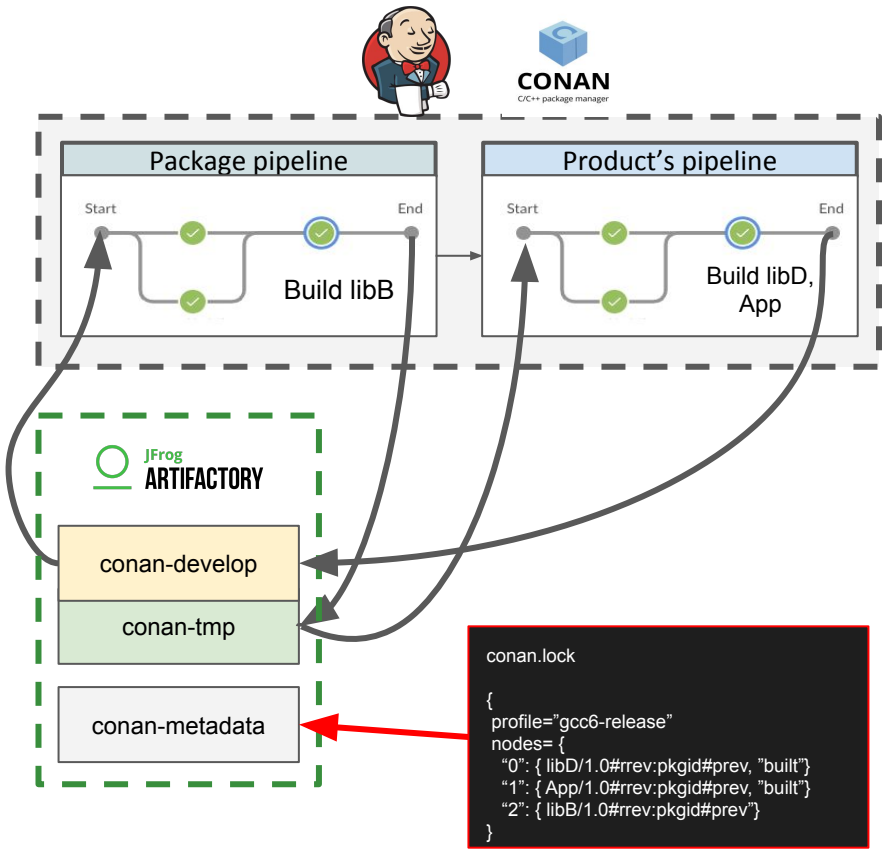


* App2 follows the same flow and has its own code repository

Mycompany development workflow



Artifactory repos



Conan repositories

conan-develop ⟶ packages that will be used by developers and other CI jobs. Packages here were usually "promoted" from conan-tmp

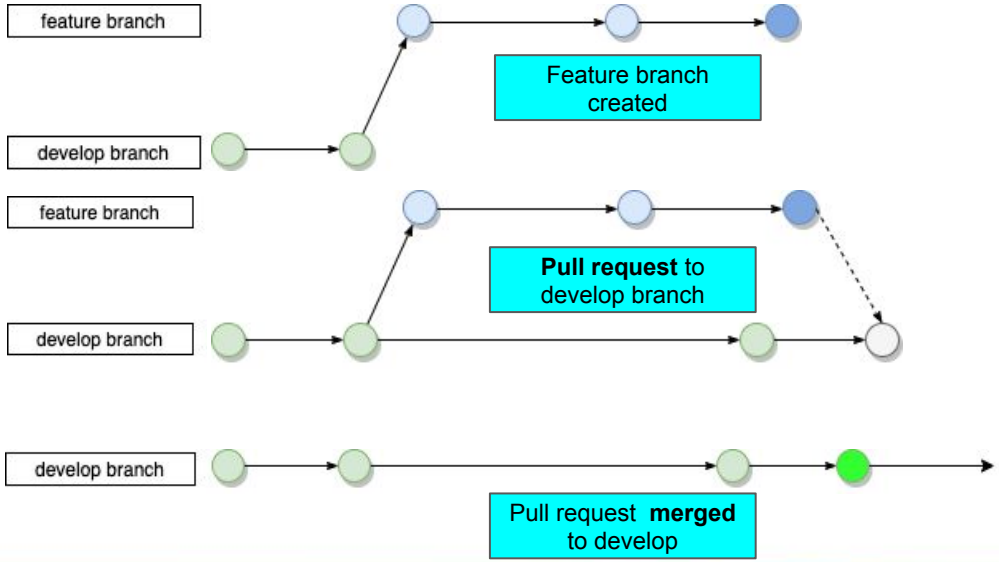
conan-tmp ⟶ packages build on the CI which are currently under development or testing, and which may be promoted in the future

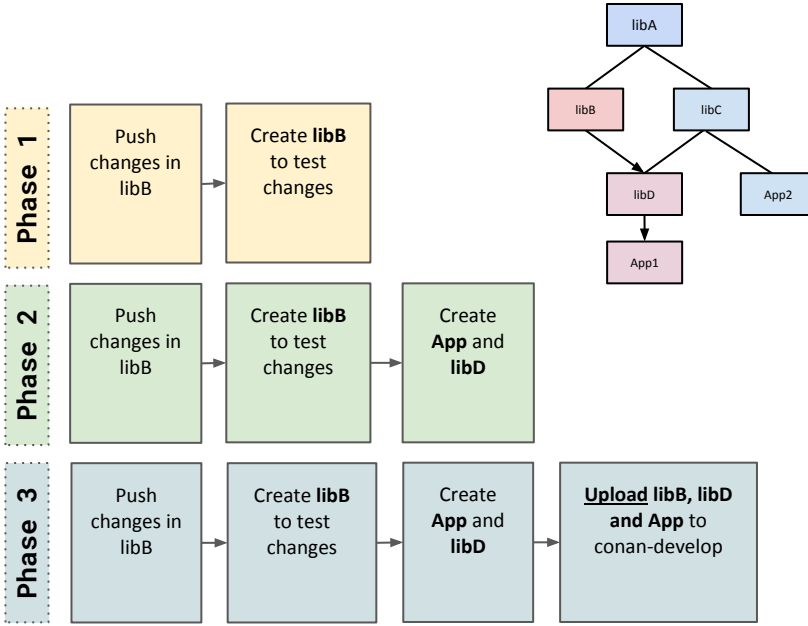
Generic repositories

conan-metadata ⟶ use to upload metadata associated with the build. We will upload the lockfiles generated creating conan packages

Phases in the workflow

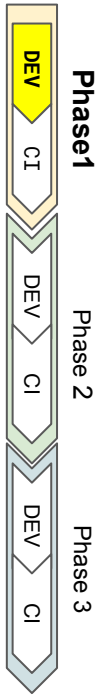
Three phases for testing changes in **libB**





Outline

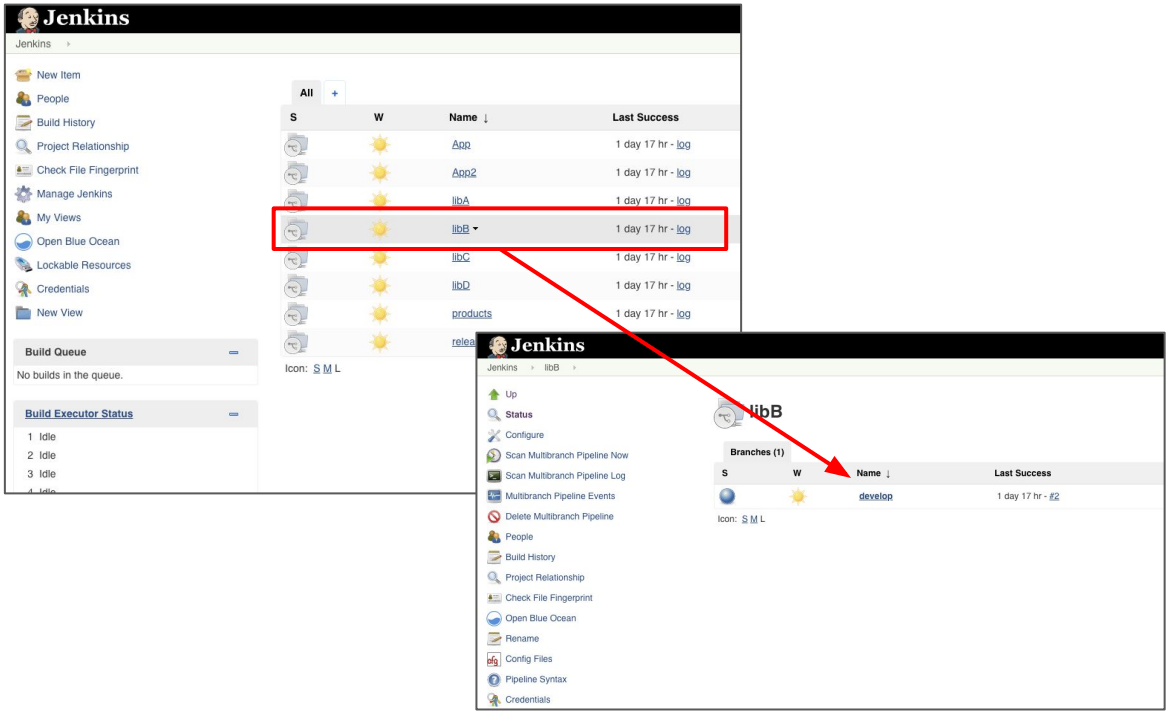
- Recap from Advanced Training
- Introduction: The Story
- **CI Workflow: Phase 1**
  - **Developer creates a feature branch**
  - Operations on CI
    - Package pipeline
- CI Workflow: Phase 2
- CI Workflow: Phase 3
- Promotion in Artifactory
- Summary
- Appendix

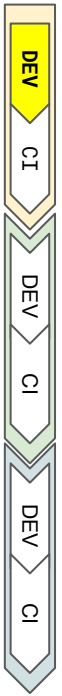


[Reminder] Access Jenkins

Username: **administrator**
Password: **<Jenkins Credential>**

In orbitera e-mail with JFrog Test Drive Details





[Lab 1] The developer creates a feature branch for libB

Goal:

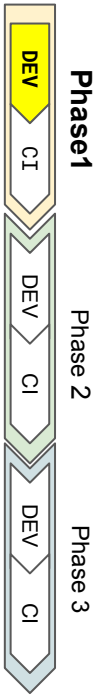
- Check and understand the series of actions that are going to be triggered in the CI when a commit is pushed to the feature branch

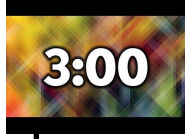
Tasks:

- Go to the developer's working folder and create a new feature branch
- Push some changes to the branch
- Jenkins: check the stages of the package pipeline being triggered by the push

Success:

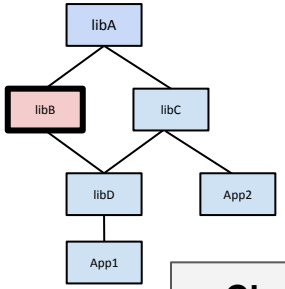
- Check the the package pipeline finishing successfully

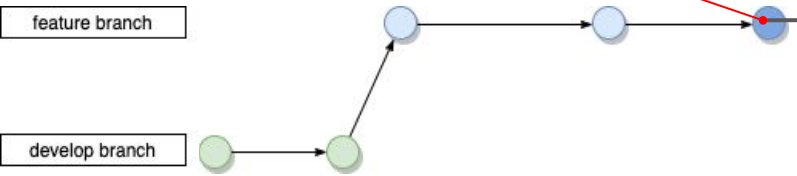




[Lab 1] The developer creates a feature branch and pushes

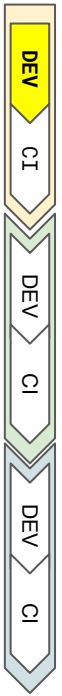
```
docker exec -it jenkins /bin/bash

cd /workdir/libB

git checkout -b cool_feature

echo "// modify libB source" >> src/libB.cpp

git commit -a -m "commit cool feature"

git push origin cool_feature
```





After the push, a **hook** triggers the Jenkins libB pipeline

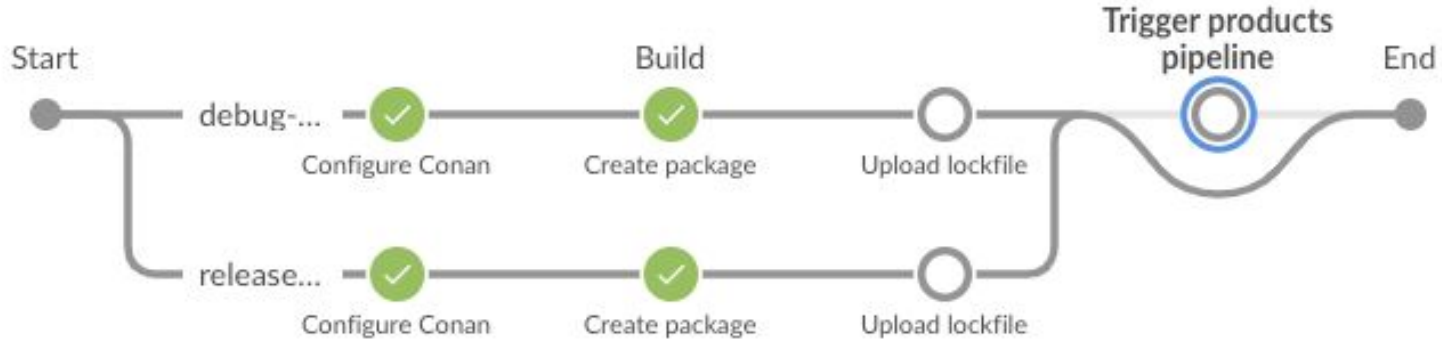
Check the results in Jenkins

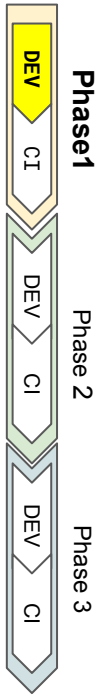


[Lab 1][Result] Check the stages run in the pipelines

Here is a link to the code for the **Package pipeline for libB**
Here we point out that:

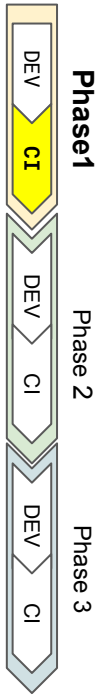
- We build multiple configurations for libB
- We choose not to trigger the products pipeline





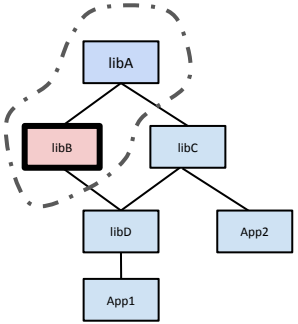
Outline

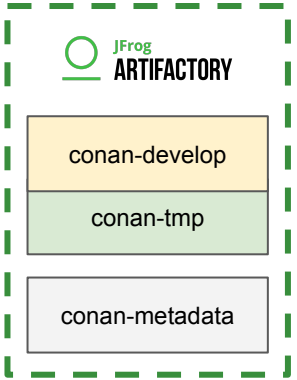
- Recap from Advanced Training
- Introduction: The Story
- **CI Workflow: Phase 1**
  - Developer creates a feature branch
  - **CI Stages**
    - **Package pipeline**
- CI Workflow: Phase 2
- CI Workflow: Phase 3
- Artifactory: Build Info
- Artifactory: Promotion
- Summary
- Appendix

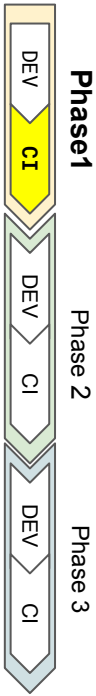


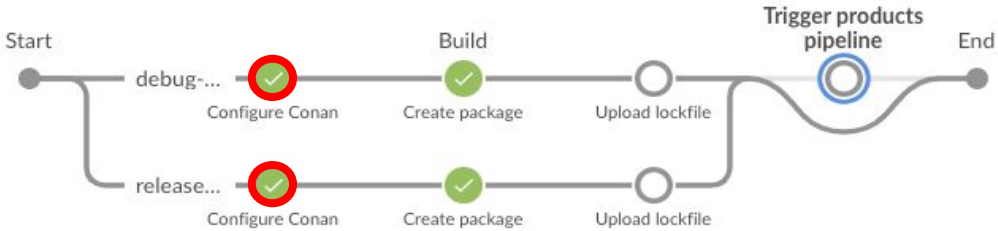
[Phase 1 - Package pipeline] [Configure Conan]

```
# set the CONAN_USER_HOME for each stage

conan config install <config_url>

conan user -p conan2020 -r conan-develop conan
```



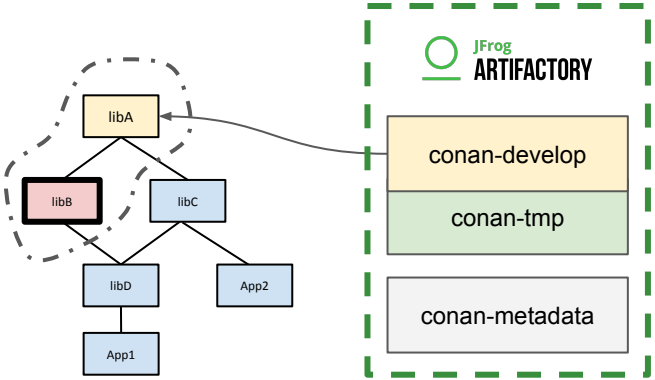


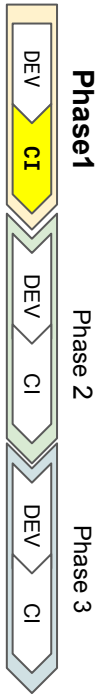


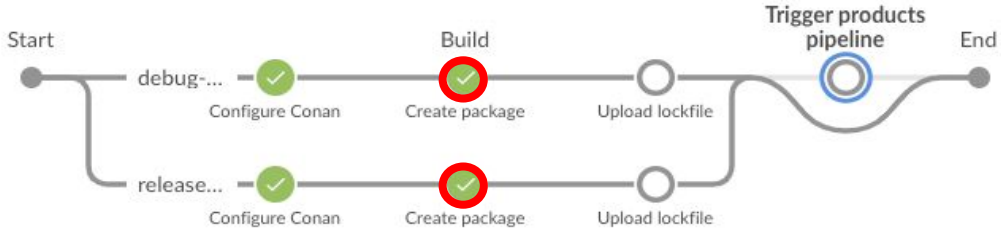


[**Phase 1 -** Package pipeline] [Create libB]

```
conan graph lock . --profile profile
--lockfile=lockfile.lock -r conan-develop

conan create . mycompany/stable --profile profile
--lockfile=lockfile.lock -r conan-develop
```

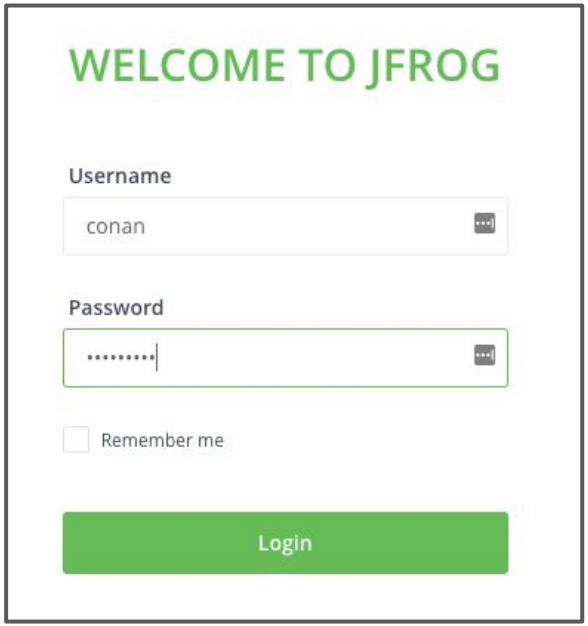




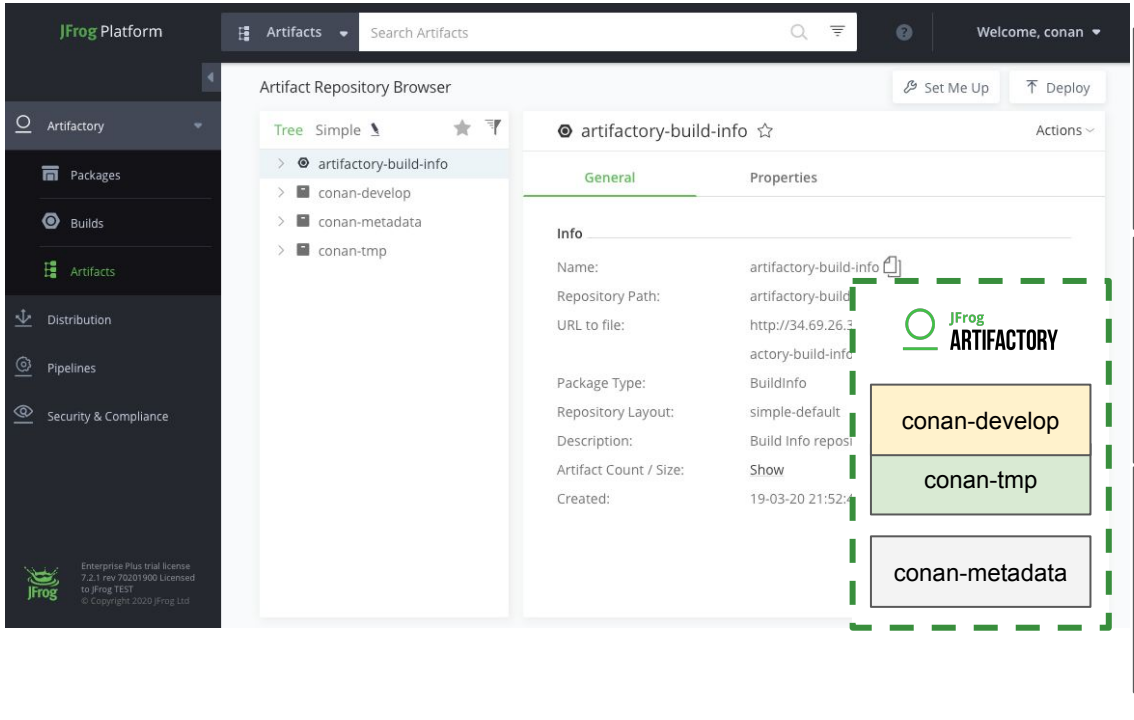


We will consume the latest revisions of the packages from conan-develop

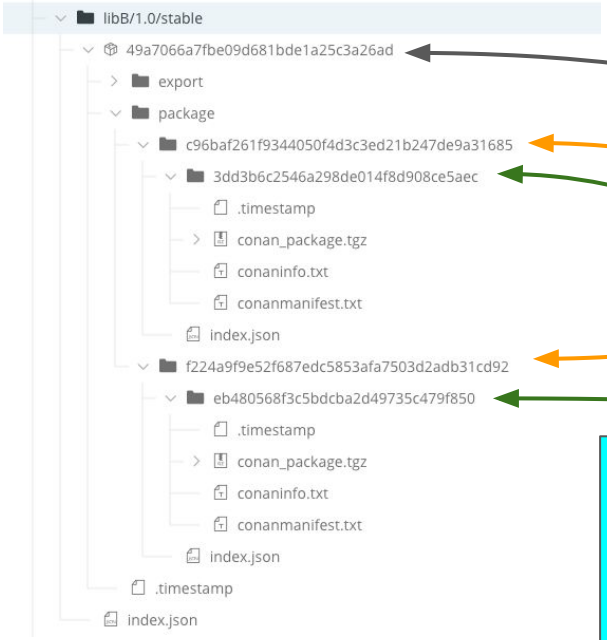
[Reminder] Access Artifactory



User: **conan**
Password: **conan2020**



Check libB in conan-tmp in Artifactory

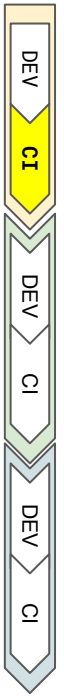


1x RECIPE REVISION(RREV)

2x PACKAGE ID

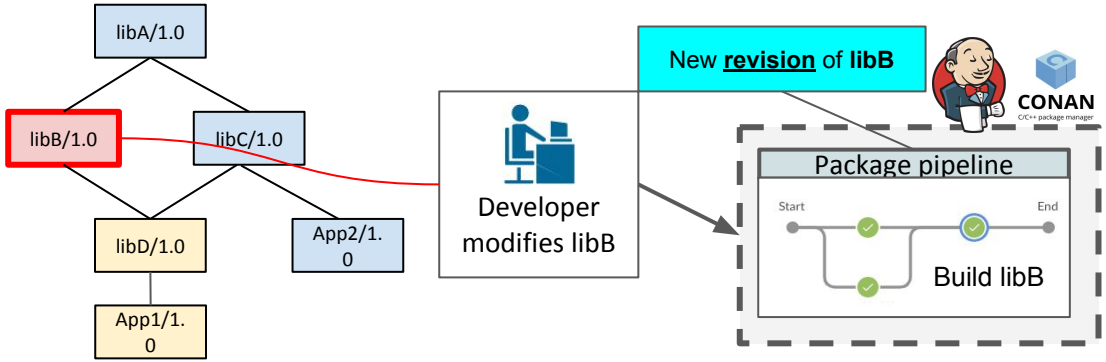
PACKAGE REVISION (PREV)

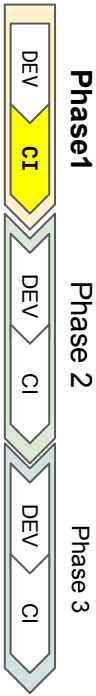
We have just one revision, that was created in the **bootstrapping.**
No packages were uploaded by the CI at this point



<name>/<version>@<user>/<channel>#<rrev>:<pkg_id>#<prev>

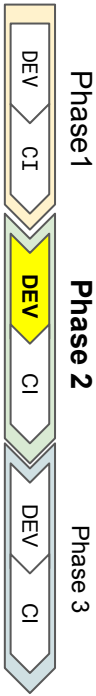
[Phase 1 - Summary]





Outline

- Recap from Advanced Training
- Introduction: The Story
- CI Workflow: Phase 1
- **CI Workflow: Phase 2**
  - **Developer opens a PR with the feature branch**
  - CI Stages
    - Package pipeline
    - Products pipeline
- CI Workflow: Phase 3
- Artifactory: Build Info
- Artifactory: Promotion
- Summary
- Appendix



[Lab 2] The developer creates a PR to libB's develop branch

Goal:

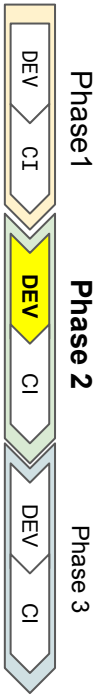
- Have a look at the set of operations that are going to be triggered in the CI when a commit is pushed to a pull request branch

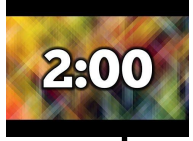
Tasks:

- Command line: Create a new branch for the PR
- Push some changes to the PR
- Check the package pipeline being triggered by the push to the repo
- Check the product's pipeline being triggered at the end of the package pipeline

Success:

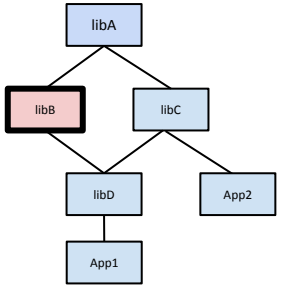
- Find the new revision of libB in conan-tmp repo in Artifactory

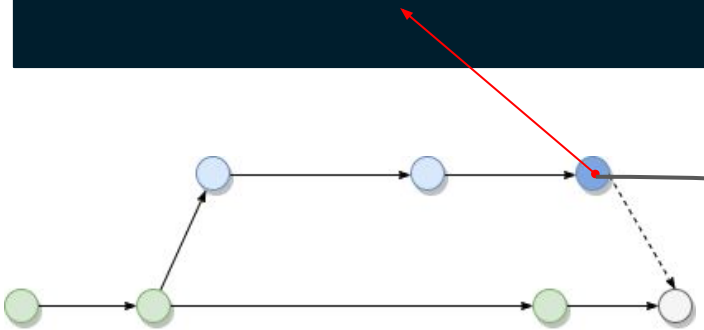




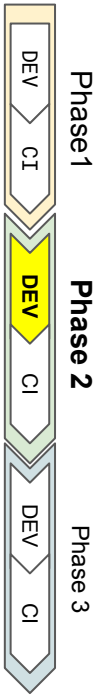
[Lab 2] The developer creates a PR to libB's develop branch

```
git checkout -b PR-01

echo "# Comments in the conanfile.py" >>
conanfile.py

git commit -a -m "simulating a PR to develop"

git push origin PR-01
```



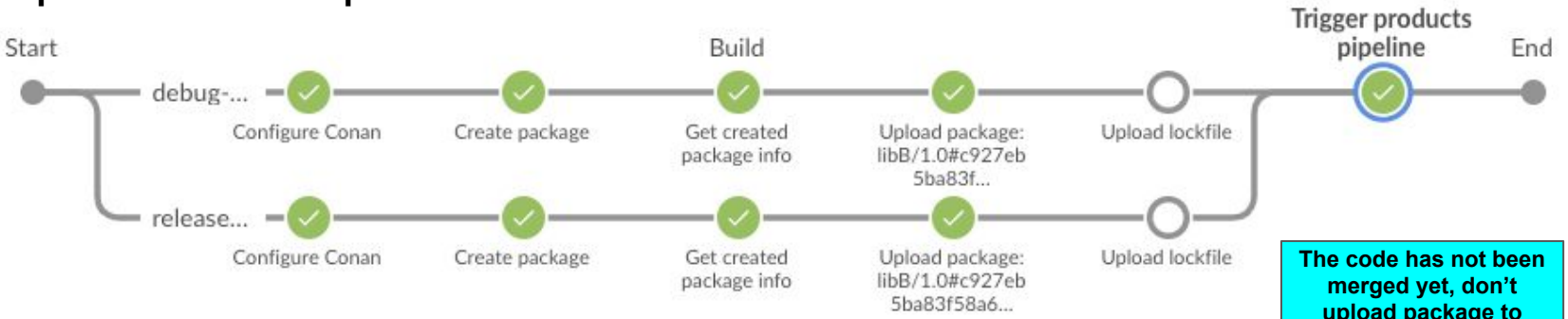


After the push a **hook** triggers the Jenkins libB pipeline



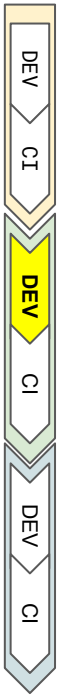
[Lab 2][Result] Check the stages run in Jenkins

Package pipeline for libB: for each configuration create the new revision and upload to conan-tmp

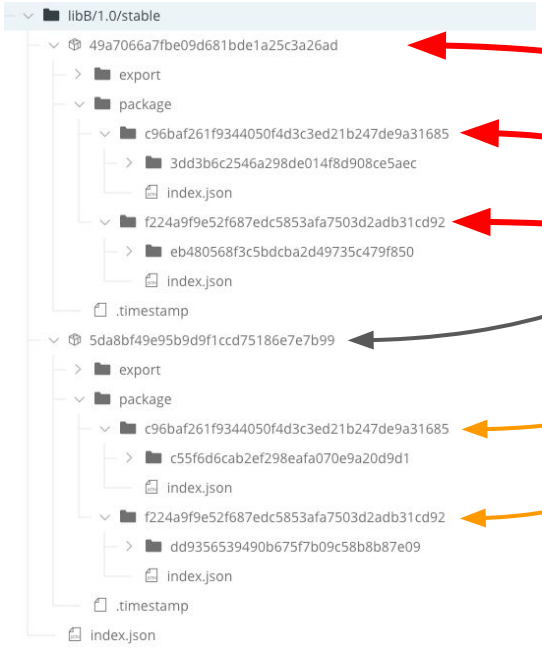


Products pipeline: check if App or App2 are affected and rebuild

The code has not been merged yet, don't upload package to conan-develop



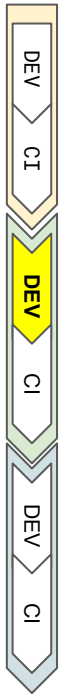
[Lab 2][Result] Check libB in conan-tmp in Artifactory



2x RECIPE REVISION(RREV)

4x PACKAGE ID (2x RREV)

The **new revision of libB** was uploaded to conan-tmp to download it from the products pipeline but now new packages are in conan-develop because the PR is not merged yet



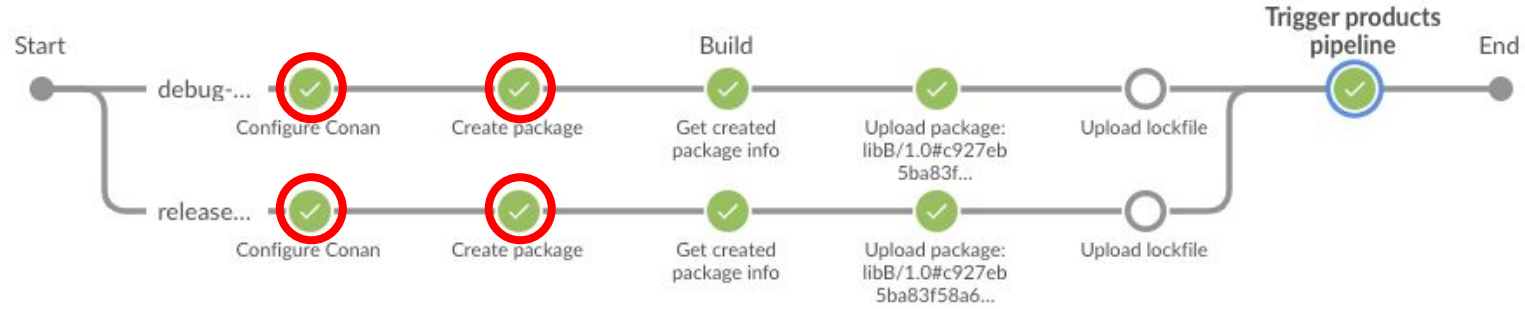
<name>/<version>@<user>/<channel>#<rrev>:<pkg_id>#<prev>

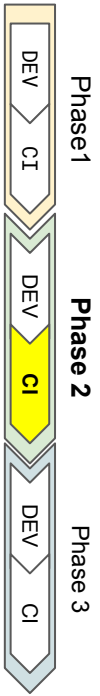
Outline

- Recap from Advanced Training
- Introduction: The Story
- CI Workflow: Phase 1
- **CI Workflow: Phase 2**
  - Developer opens a PR with the feature branch
  - **Operations on CI**
    - **Package pipeline**
    - Products pipeline
- CI Workflow: Phase 3
- Artifactory: Build Info
- Artifactory: Promotion
- Summary
- Appendix

[**Phase 2 -** Package pipeline] Stages in common with Phase 1

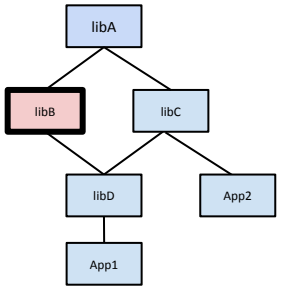
- Configure Conan
- Create new revision of libB with changes



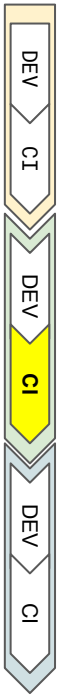


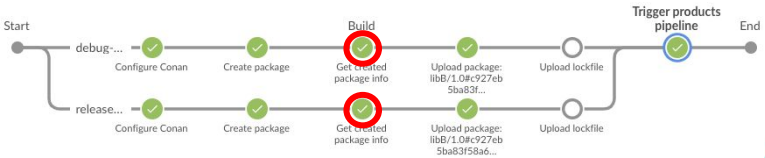
[Phase 2 - Package pipeline] [Get created package information]

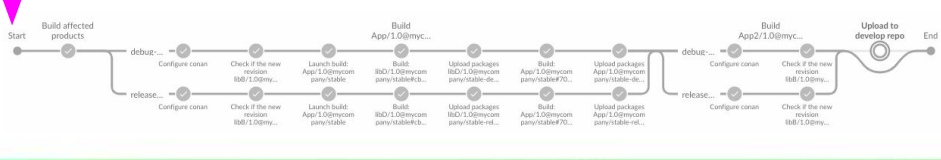
```
# we have just created the new revision of libB
in the local cache, we search for it as we only
have one revision there

conan inspect . --raw name
conan inspect . --raw version
conan search libB/1.0@mycompany/stable
--revisions --raw

# get the output of the commands
# name = libB
# version = 1.0
# revision = 5da8bf49e95b9d9f1ccd75186e7e7b99
```



We need to pass libB/1.0#rrev as a parameter to the products pipeline so that we know which libB needs to be tested against the products

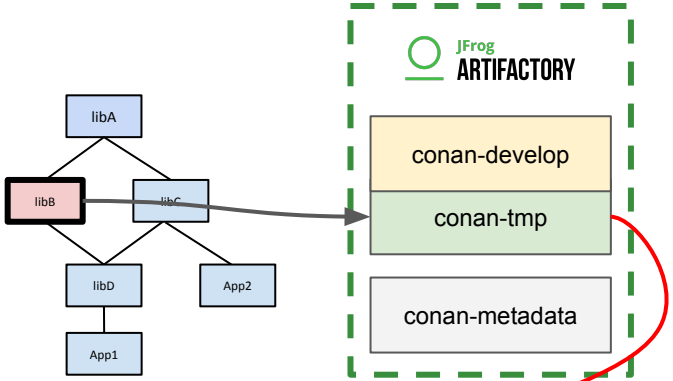




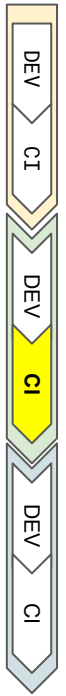


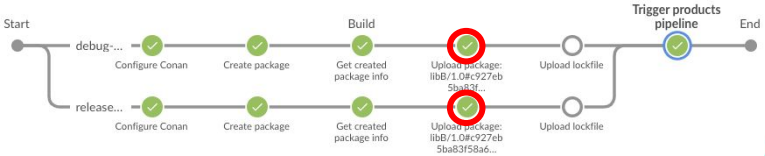
[**Phase 2 -** Package pipeline] [Upload libB to conan-tmp]

```
# we have just retrieved the name, version and
revision from the created package

conan upload 'libB/1.0' --all -r conan-tmp
--confirm
```

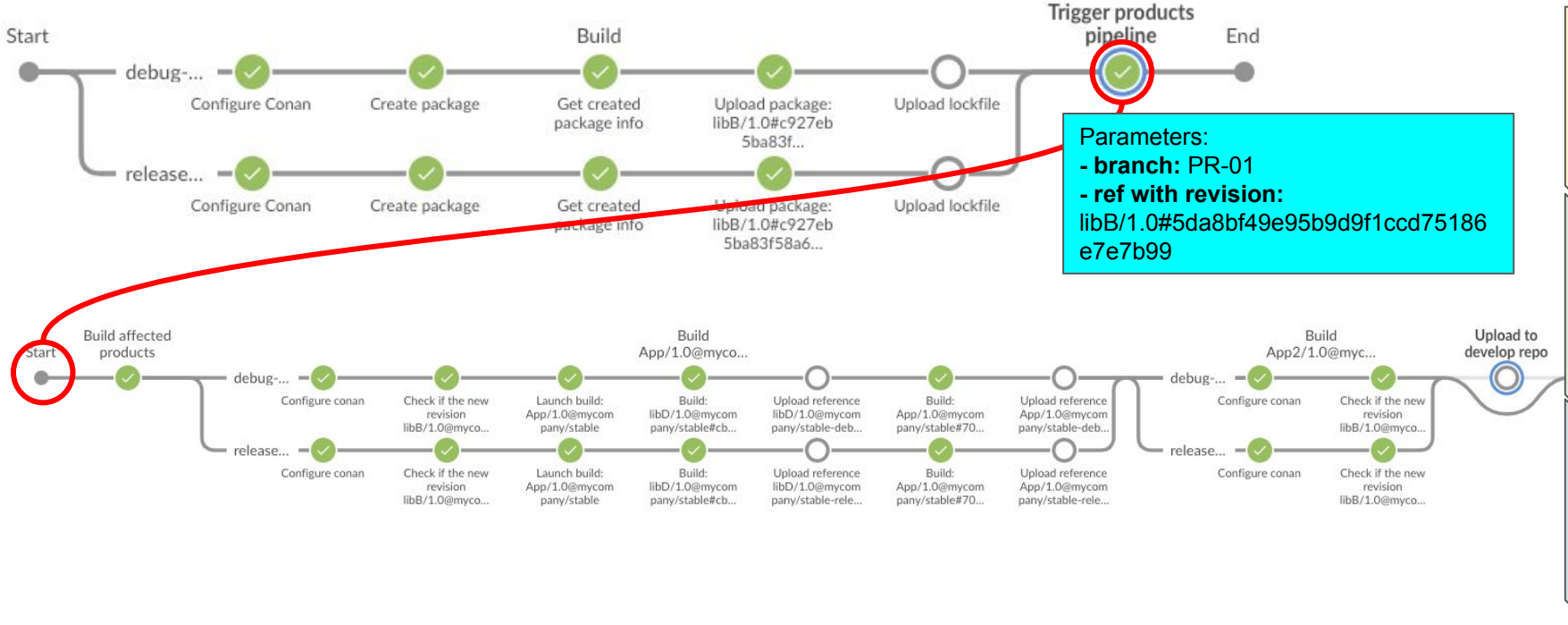


Later, in the products pipeline the CI will retrieve **libB/1.0#rrev** from conan-tmp to integrate its changes into **App**





[**Phase 2 -** Package pipeline] [Trigger the products pipeline]



Outline

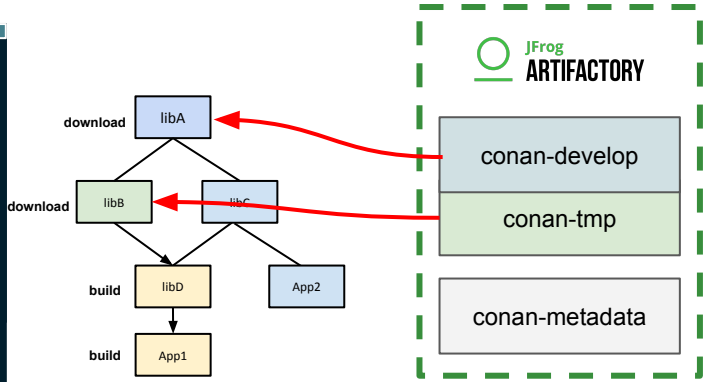
- Recap from Advanced Training
- Introduction: The Story
- CI Workflow: Phase 1
- **CI Workflow: Phase 2**
  - Developer opens a PR with the feature branch
  - **CI Stages**
    - Package pipeline
    - **Products pipeline**
- CI Workflow: Phase 3
- Artifactory: Build Info
- Artifactory: Promotion
- Summary
- Appendix

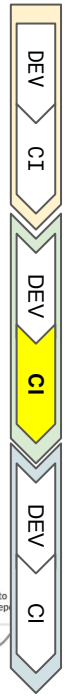
[**Phase 2** - Products pipeline] [Check if libB/1.0#rrev affects products]

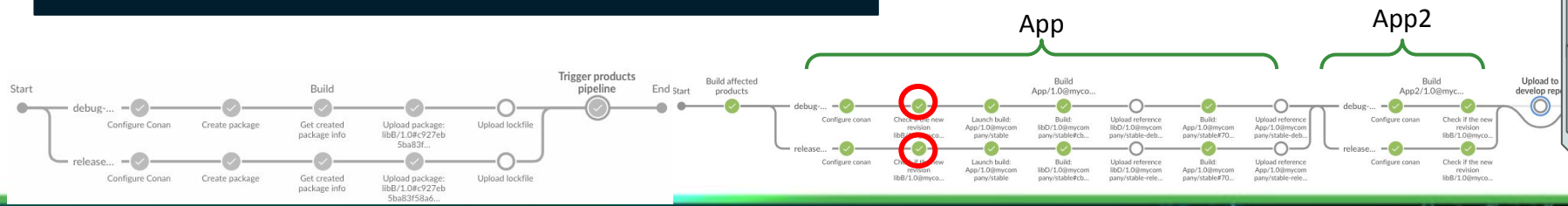
products = ["App/1.0@mycompany/stable", "App2/1.0@mycompany/stable"]

For each product:

1. **Download** the recipe of the created revision **of libB** from **conan-tmp**
2. Get the **lockfile** of the product we want to check getting the dependencies from **conan-develop**
3. Calculate the **build-order** with the lockfile: if the build-order is empty, the product is not affected



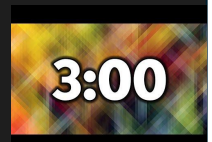


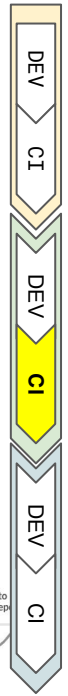


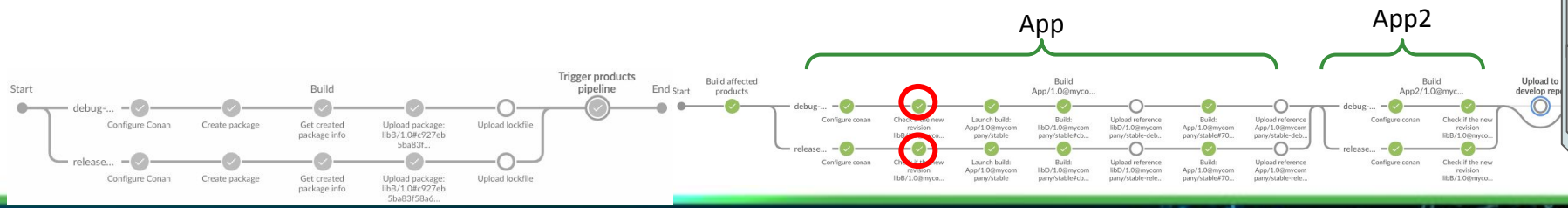
[Phase 2 - Products pipeline][Check if libB/1.0#rrev affects products]

products/Jenkinsfile

```
stage("Check if the new revision ${params.reference} is in ${product} graph") {
 sh "conan download ${params.reference} -r ${conan_tmp_repo} --recipe"
 sh "conan graph lock ${product} --profile=${profile} --lockfile=${lockfile} -r ${conan_develop_repo}"
 sh "conan graph build-order ${lockfile} --json=${bo_file} --build missing"
 build_order = readJSON(file: bo_file)
 if (build_order.size()>0) {
   affected_product = true
 }
}
```







[Lab 3] Check if a new revision of libB affects App

Goal:

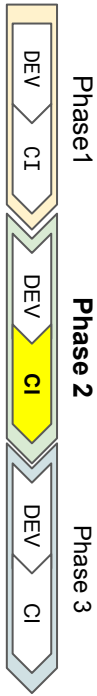
- See if the new revision of libB is affecting a product downstream so that they have to be rebuilt

Tasks:

- Search for available revisions in conan-tmp
- Download the recipe for the latest revision of libB from conan-tmp
- Do the graph lock for the product using the conan-develop remote (latest revisions of libs)
- Calculate the build order with --build missing, will tell us if the new revision of libB is affecting App

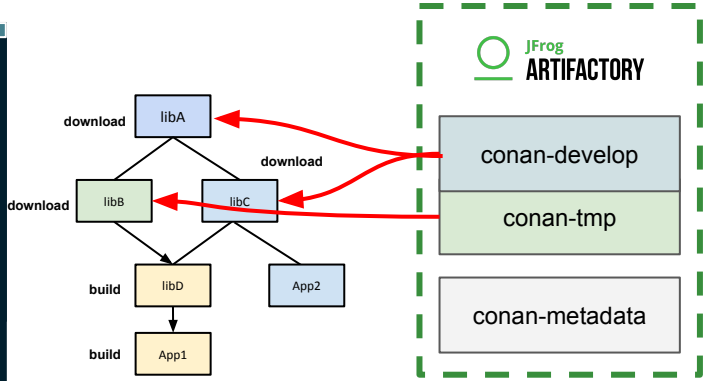
Success:

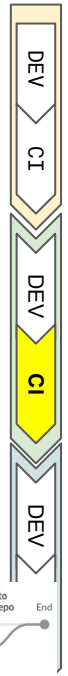
- The build order for App contains libD and App

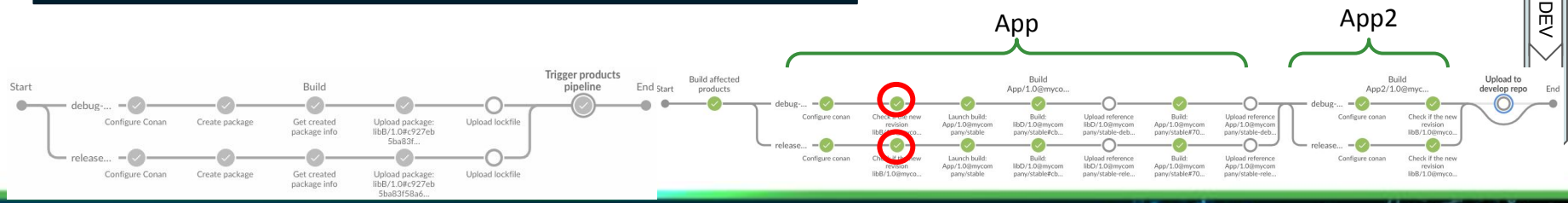


[Lab 3] Check if App is affected by libB/1.0#rrev

```
cd /ci_labs
# Get the revision list from the server (just for the lab, not made by
the CI) and get the latest revision
conan search libB/1.0@mycompany/stable -r conan-tmp --revisions

# Download the recipe of the created revision of libB from conan-tmp
conan download libB/1.0@mycompany/stable#<rrev> -r conan-tmp --recipe

# Get the lockfile of the product we want to check getting the
dependencies from conan-develop
conan graph lock App/1.0@mycompany/stable --profile=debug-gcc6
--lockfile=App.lock -r conan-develop

# Calculate the build-order with the lockfile: if the build-order is
empty, the product is not affected
conan graph build-order App.lock --json=bo.json --build missing

cat bo.json
```







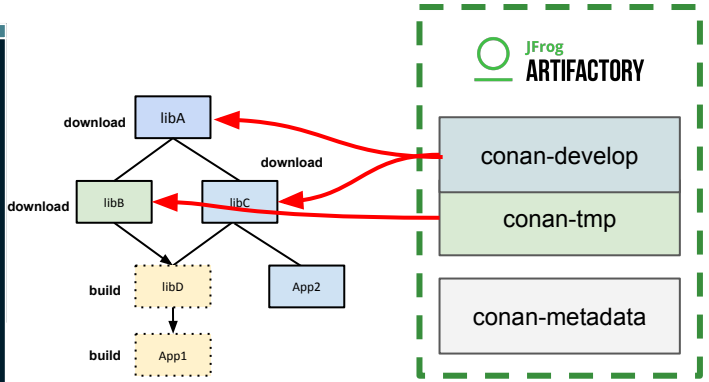
[Phase 2 - Products pipeline][App/1.0 affected → Build libD, App]

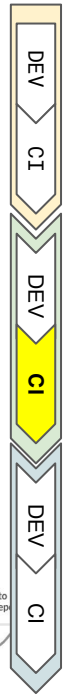
We already have calculated the lockfile for App and got the build order → iterate through the build order [libD, App]:

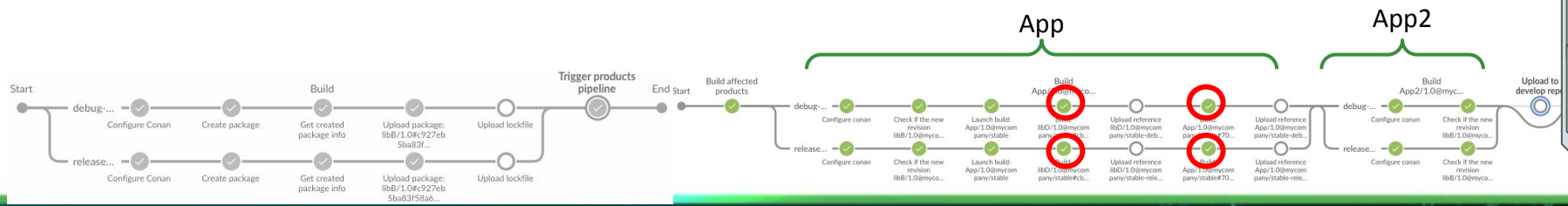
1. cp App.lock conan.lock
2. Create libD: conan install libD/1.0@... --build libD --lockfile conan.lock
3. cp conan.lock libD.lock

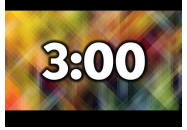
4. conan graph update-lock App.lock libD.lock

5. cp App.lock conan.lock
6. Create App: conan install App/1.0@... --build App --lockfile conan.lock





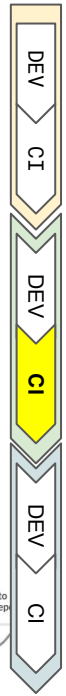


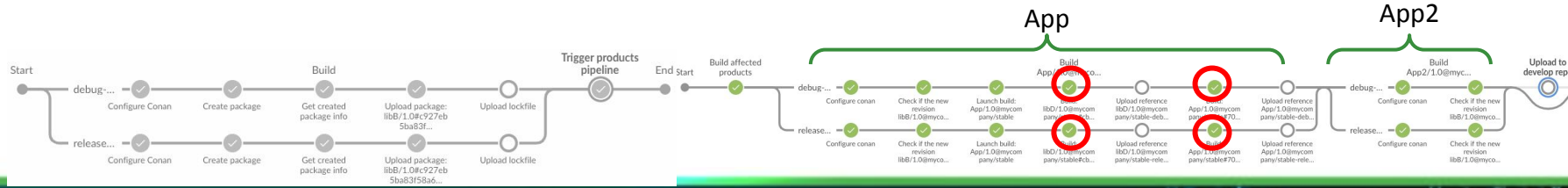


[Phase 2 - Products pipeline][App/1.0 affected → Build libD, App]

products/Jenkinsfile

```
stage("Launch build: ${product}")
{
 stash name: lockfile, includes: lockfile
 build_order.each { references_list ->
   def stage_jobs = references_list.each { index_reference ->
     def lib_name = index_reference[1].split("/")[0]
     def lib_name_profile = "${lib_name}-${profile}.lock"
     def upload_ref = (params.library_branch == "develop") ? true : false
     build_ref_with_lockfile(index_reference[1], lockfile, profile, upload_ref).call()
     unstash lib_name_profile
     sh "conan graph update-lock ${lockfile} ${lib_name_profile}"
     stash name: lockfile, includes: lockfile
   }
 }
}
```





[Lab 4] Build App using lockfiles and build order

Goal:

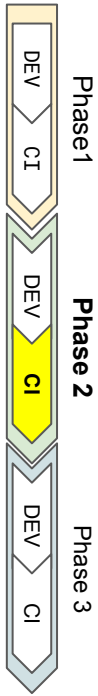
- Understand the process of building downstream packages using a lockfiles and build order

Task:

- Calculate the build order of App using the lockfile
- Build libD → update lockfile

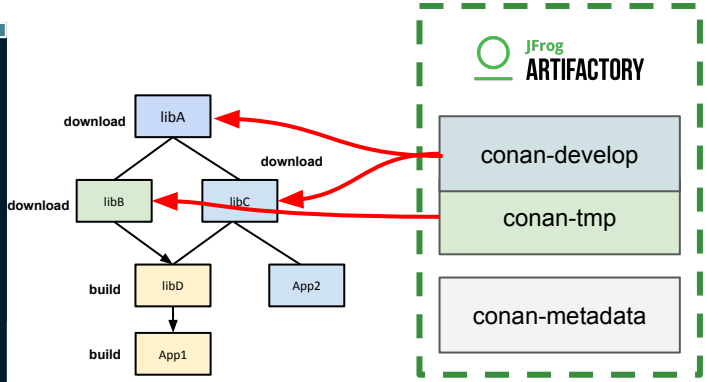
Success:

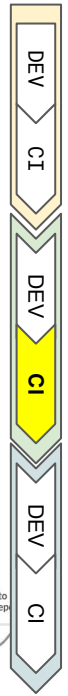
- Check libD being marked as built in libD.lock and App.lock

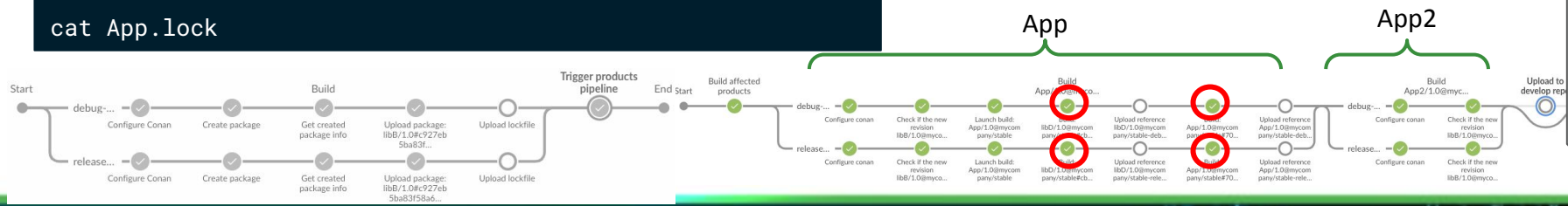


[Lab 4] Build App using lockfiles and build order

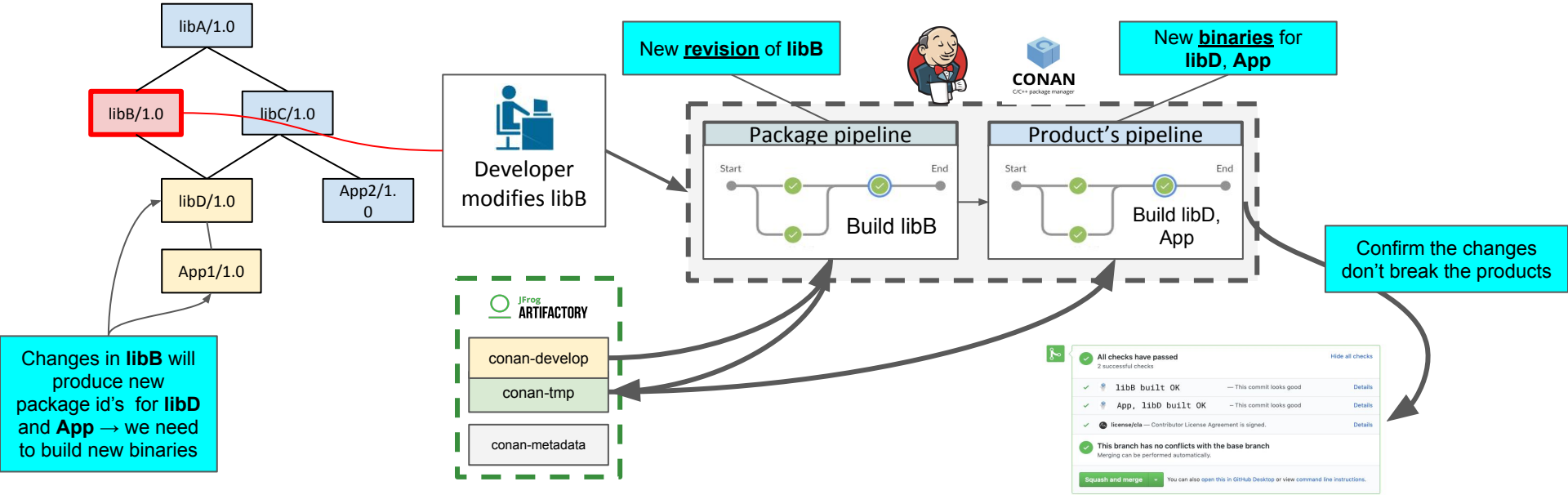
```
cat bo.json

cp App.lock conan.lock

cat conan.lock

conan install libD/1.0@mycompany/stable --build libD --lockfile
conan.lock

cp conan.lock libD.lock

# the install marked libD as modified="built" in the lockfile
cat libD.lock

# we bring those changes to App.lock and continue iterating
until we build all the nodes in bo.json
conan graph update-lock App.lock libD.lock

cat App.lock
```





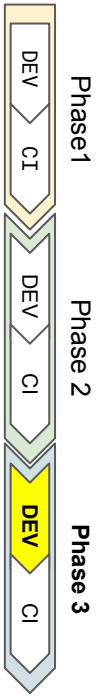


[Phase 2 - Summary]



Outline

- Recap from Advanced Training
- Introduction: The Story
- CI Workflow: Phase 1
- CI Workflow: Phase 2
- **CI Workflow: Phase 3**
  - **The PR is merged to the develop branch**
  - CI Stages
    - Package pipeline
    - Products pipeline
- Artifactory: Build Info
- Artifactory: Promotion
- Summary
- Appendix



[Lab 5] The PR is merged to develop

Goal:

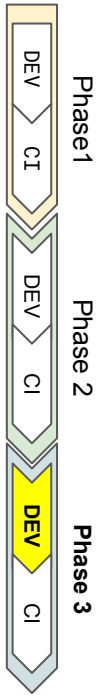
- Have a look at the set of operations that are going to be triggered in the CI when the PR-01 is merged to develop and the changes are pushed

Tasks:

- Checkout develop branch
- Merge PR-01
- Push to origin
- Check the package pipeline being triggered by the push to the repo
- Check the product's pipeline being triggered at the end of the package pipeline

Success:

- Find the new revision of libB in conan-develop repo in Artifactory
- Find the new binaries of libD and App in conan-develop repo in Artifactory
- Check conan-metadata repo

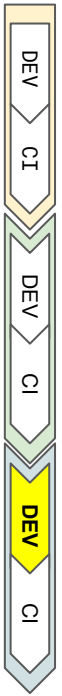


[Lab 5] The PR is merged to develop

```
cd /workdir/libB

git checkout develop

git merge PR-01 --no-ff -m "merge PR-01"

git push origin develop
```

We will have a new commit so we need to launch the pipelines again and rebuild

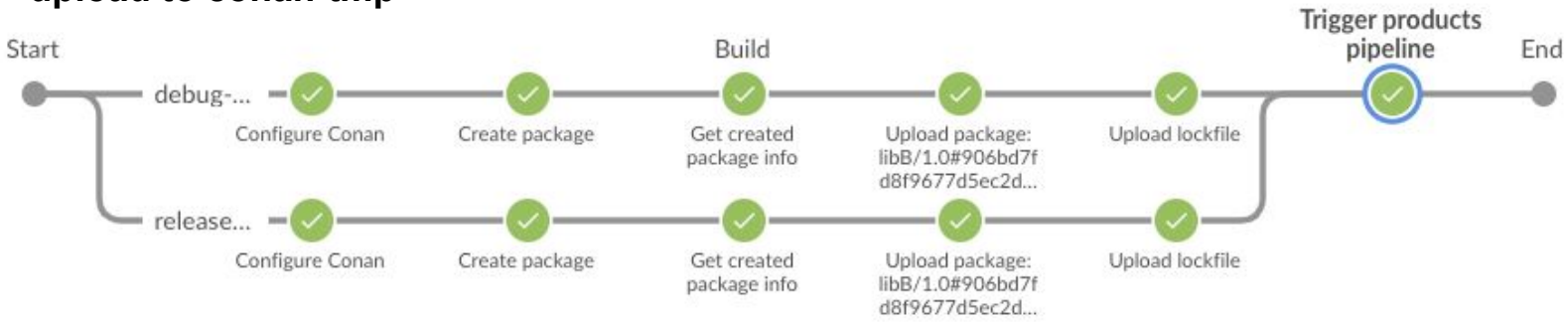
After the push a **hook** triggers the Jenkins libB pipeline and starts the job for the develop branch



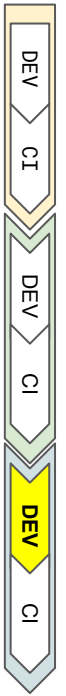
2:00

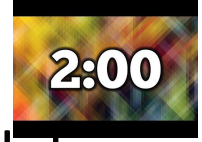
[Lab 5][Result] Check the stages run in the pipelines

Package pipeline for libB: for each configuration create the new revision and upload to conan-tmp



Products pipeline: check if App or App2 are affected and rebuild, upload artifacts and lockfiles to conan-develop





[Lab 5][Result] Check conan-develop and conan-metadata

- conan-develop
  - mycompany
    - App/1.0/stable
    - App2/1.0/stable
    - libA/1.0/stable
    - libB/1.0/stable
      - 49a7066a7fbe09d681bde1a25c3a26ad
      - 977295a4a866ca4e07b5d9ec3994677a
      - index.json
    - libC/1.0/stable
    - libD/1.0/stable
      - 3c53819b8edcf327e16dd275f7a1c103
        - export
        - package
          - 2e9e2c77ca150ea2282f1836ad7b296201ad4b7d
          - 3f683615ce650fcadb110c2378a1fc73dd473270
          - 47e8963e688336d05692d3806e5d9ea33da83d6f
          - 7f9a0d6f1593b5af70e986a9cd6ea5ab573d1793
        - .timestamp
      - index.json

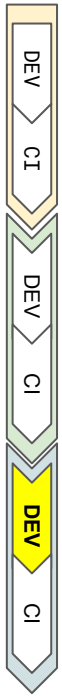
New revision of libB

New binaries for libD and App

Lockfiles for libB and App uploaded to conan-metadata for the different profiles built

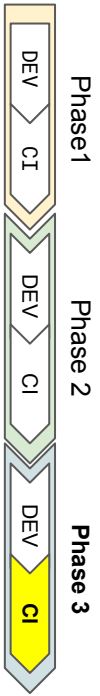
- conan-metadata
  - libB/develop/5/libB/1.0@mycompany/stable
  - products/master/21/App/1.0@mycompany/stable
    - debug-gcc6
      - App-debug-gcc6.lock
    - release-gcc6
      - App-release-gcc6.lock

Property	Value(s)
profile	debug-gcc6
name	App
build.name	products/master
build.number	21
version	1.0



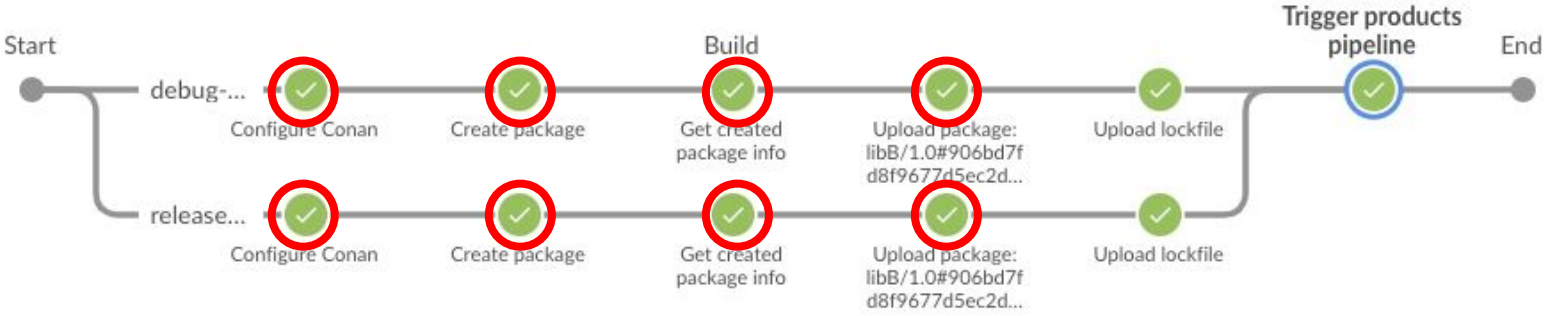
Outline

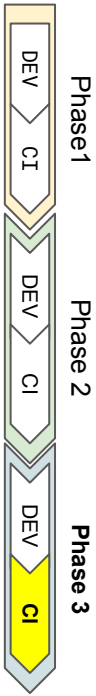
- Recap from Advanced Training
- Introduction: The Story
- CI Workflow: Phase 1
- CI Workflow: Phase 2
- **CI Workflow: Phase 3**
  - The PR is merged to the develop branch
  - **CI Stages**
    - **Package pipeline**
    - Products pipeline
- Artifactory: Build Info
- Artifactory: Promotion
- Summary
- Appendix



[**Phase 3 -** Package pipeline] Stages in common with Phase 2

- Configure Conan
- Create new revision of libB with changes
- Get libB's revision, name and version
- Upload new revision of libB to conan-tmp

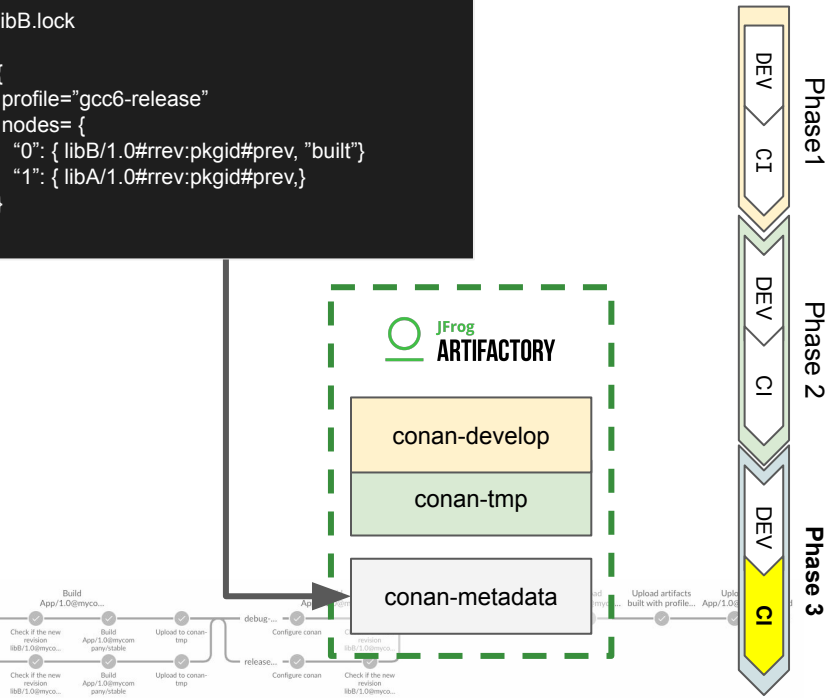


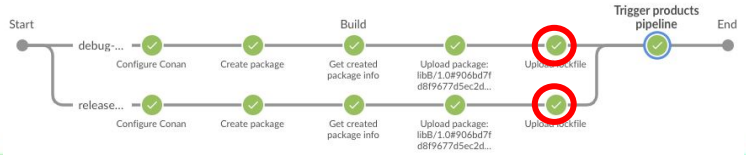


[Phase 3 - Package pipeline][Upload libB lockfile to conan-metadata]

```
# upload the lockfile to conan-metadata repo using the
artifactory REST API
# you can use the build_name and number for the path

curl -u conan:conan2020 -X PUT
http://<artifactory_url>/conan-metadata/<path>/ -T conan.lock

# you can also assign properties to the uploaded files

curl -u conan:conan2020 -X PUT
http://<artifactory_url>/api/storage/<path>/conan.lock?propertie
s=build.name=conan-app%7Cbuild.number=1%7Cprofile=gcc6-release%7
ClibB.version=1.0
```

```
libB.lock

{
 profile="gcc6-release"
 nodes= {
   "0": { libB/1.0#rrev:pkgid#prev, "built"}
   "1": { libA/1.0#rrev:pkgid#prev,}
}
```

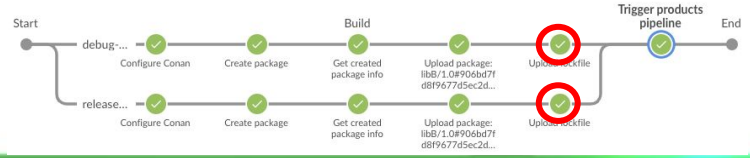


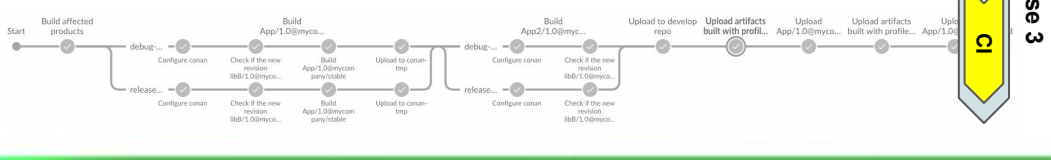


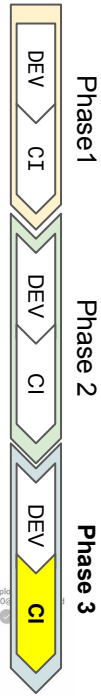
[Phase 3 - Package pipeline][Upload libB lockfile to conan-metadata]

libB/Jenkinsfile

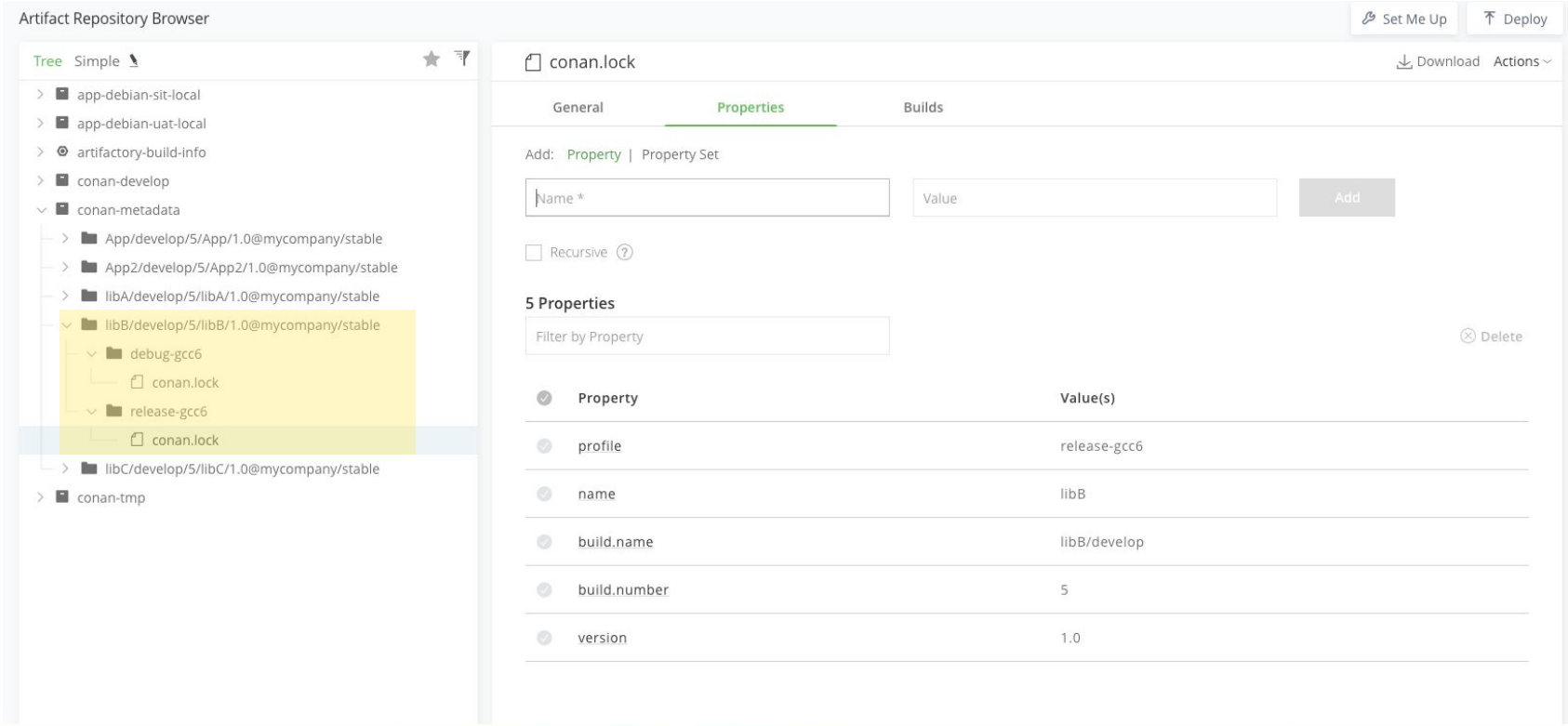
```
stage("Upload lockfile") {
   if (env.BRANCH_NAME == "develop") {
       def lockfile_path =
"/${artifactory_metadata_repo}/${env.JOB_NAME}/${env.BUILD_NUMBER}/${name}/${version}@${user_channel}/${profile}/conan.lock"
       def base_url = "http://${artifactory_url}:8081/artifactory"
       def properties =
"?properties=build.name=${env.JOB_NAME}%7Cbuild.number=${env.BUILD_NUMBER}%7Cprofile=${profile}%7Cname=${name}%7Cversion=${version}"
       withCredentials([usernamePassword(credentialsId: 'artifactory-credentials', usernameVariable: 'ARTIFACTORY_USER', passwordVariable:
'ARTIFACTORY_PASSWORD')]) {
           // upload the lockfile
           sh "curl --user \"\${ARTIFACTORY_USER}\":\"\${ARTIFACTORY_PASSWORD}\" -X PUT ${base_url}${lockfile_path} -T ${lockfile}"
           // set properties in Artifactory for the file
           sh "curl --user \"\${ARTIFACTORY_USER}\":\"\${ARTIFACTORY_PASSWORD}\" -X PUT ${base_url}/api/storage${lockfile_path}${properties}"
       }
   }
}
```

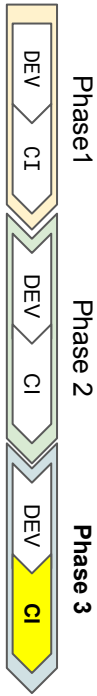






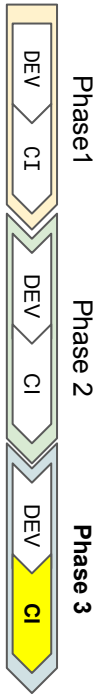
[**Phase 3 -** Package pipeline][Upload libB lockfile to conan-metadata]





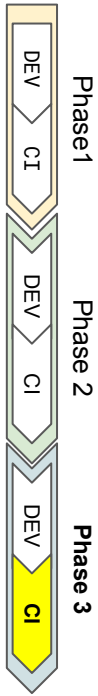
Outline

- Recap from Advanced Training
- Introduction: The Story
- CI Workflow: Phase 1
- CI Workflow: Phase 2
- **CI Workflow: Phase 3**
  - The PR is merged to the develop branch
  - **CI Stages**
    - Package pipeline
    - **Products pipeline**
- Artifactory: Build Info
- Artifactory: Promotion
- Summary
- Appendix



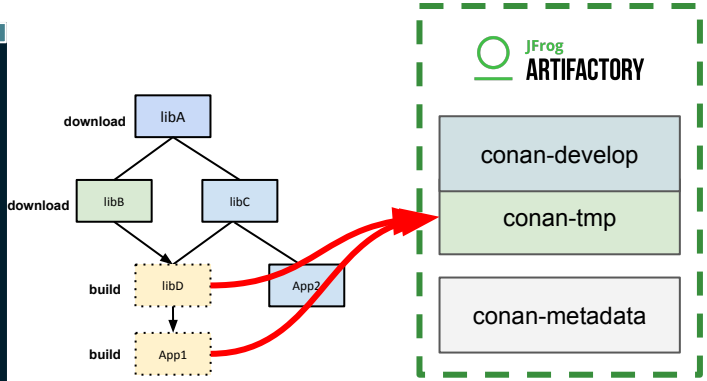
[**Phase 3 -** Products pipeline] Stages in common with Phase 2

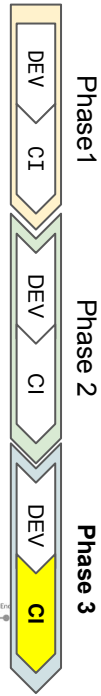
- Configure Conan
- Check if App or App2 are affected by the changes
- Build needed packages

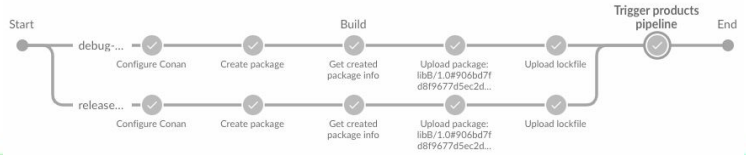


[**Phase 3 -** Products pipeline][Upload packages to conan-tmp]

```
conan upload libD --all -r conan-tmp --confirm

conan upload App --all -r conan-tmp --confirm
```







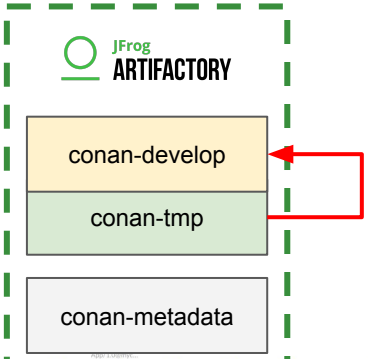
[**Phase 3 -** Products pipeline][Copy built packages from tmp to develop repo]

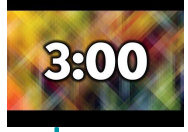
The information for all the built packages is stored in the lockfile we have used for building. We'll need code which does the following:

1. Iterate through all the nodes in the lockfile and add all those marked as modified="built" to a list. Also add the new revision of libB for promotion (it's not marked as "built" in the lockfile).
2. Copy export folder: sources, recipe, manifest…
3. Copy packages id's marked as built in the lockfile

* The copies are made using Artifactory's API

```
conan.lock

{
 profile="gcc6-release"
 nodes= {
   "0": { libD/1.0#rrev:pkgid#prev, "built"}
   "1": { App/1.0#rrev:pkgid#prev, "built"}
   "2": { libB/1.0#rrev:pkgid#prev"}
}
```

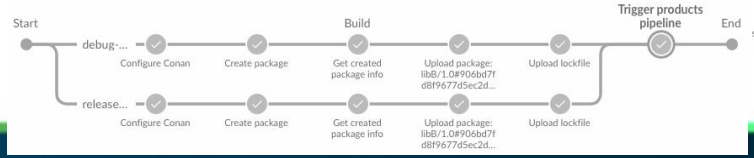


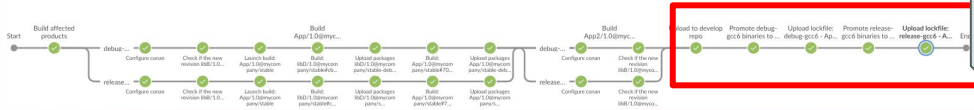


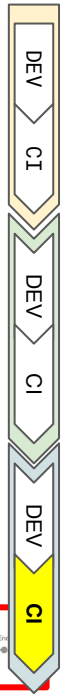
[Phase 3 - Products pipeline][Copy built packages from tmp to develop repo]

products/Jenkinsfile

```
def promote_with_lockfile(lockfile_json, source_repo, target_repo, additional_references=[]) {
 def references_to_copy = []
 def nodes = lockfile_json['graph_lock'].nodes
 nodes.each { id, node_info ->
   // iterate through the nodes and get those marked as built in the lockfile
   // add to those references the additionals (in this case libB, find the full reference and add)
   references_to_copy.add ...
 }
 references_to_copy.each { pref ->
   def recipe_source_path = "${source_repo}/${user}/${name_version}/${channel}/${rrev}/"
   def recipe_target_path = "${target_repo}/${user}/${name_version}/${channel}/${rrev}"
   def package_source_path = "${source_repo}/${user}/${name_version}/${channel}/${rrev}/package/${pkgid}/${prev}"
   def package_target_path = "${target_repo}/${user}/${name_version}/${channel}/${rrev}/package/${pkgid}/${prev}"
   withCredentials([usernamePassword(credentialsId: 'artifactory-credentials', usernameVariable: 'ARTIFACTORY_USER', passwordVariable: 'ARTIFACTORY_PASSWORD')]) {
     sh "curl -u\"\${ARTIFACTORY_USER}\":\"\${ARTIFACTORY_PASSWORD}\" -XPOST
\"http://${artifactory_url}:8081/artifactory/api/copy/${recipe_source_path}/export?to=${recipe_target_path}\""
     sh "curl -u\"\${ARTIFACTORY_USER}\":\"\${ARTIFACTORY_PASSWORD}\" -XPOST
\"http://${artifactory_url}:8081/artifactory/api/copy/${package_source_path}?to=${package_target_path}\""
   }
 }
}
```



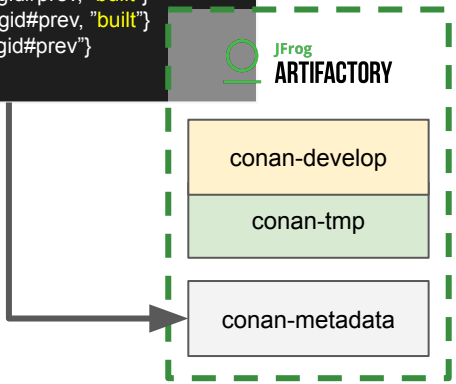


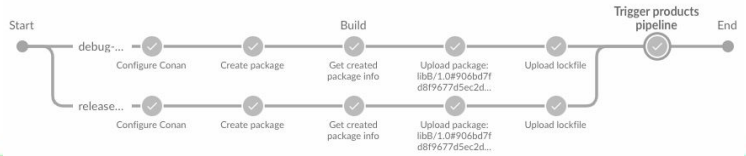


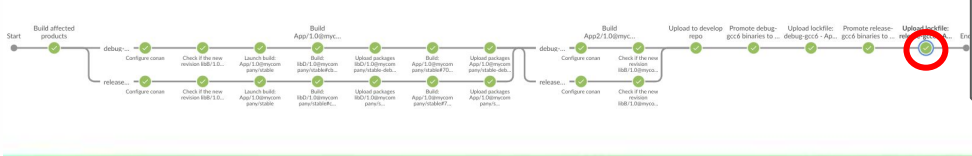
[**Phase 3 -** Products pipeline][Upload App lockfile to conan-metadata]

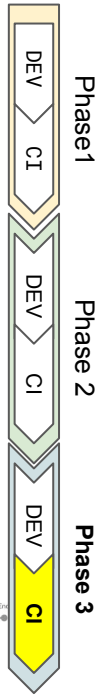
```
# upload the lockfile to conan-metadata repo using the
artifactory REST API
# you can use the build_name and number for the path

curl -u conan:conan2020 -X PUT
http://<artifactory_url>/conan-metadata/<path>/ -T conan.lock

# you can also assign properties to the uploaded files

curl -u conan:conan2020 -X PUT
http://<artifactory_url>/api/storage/<path>/conan.lock?propertie
s=build.name=...build.number=..version..
```

```
conan.lock

{
 profile="gcc6-release"
 nodes= {
   "0": { libD/1.0#rrev:pkgid#prev, "built"}
   "1": { App/1.0#rrev:pkgid#prev, "built"}
   "2": { libB/1.0#rrev:pkgid#prev"}
}
```

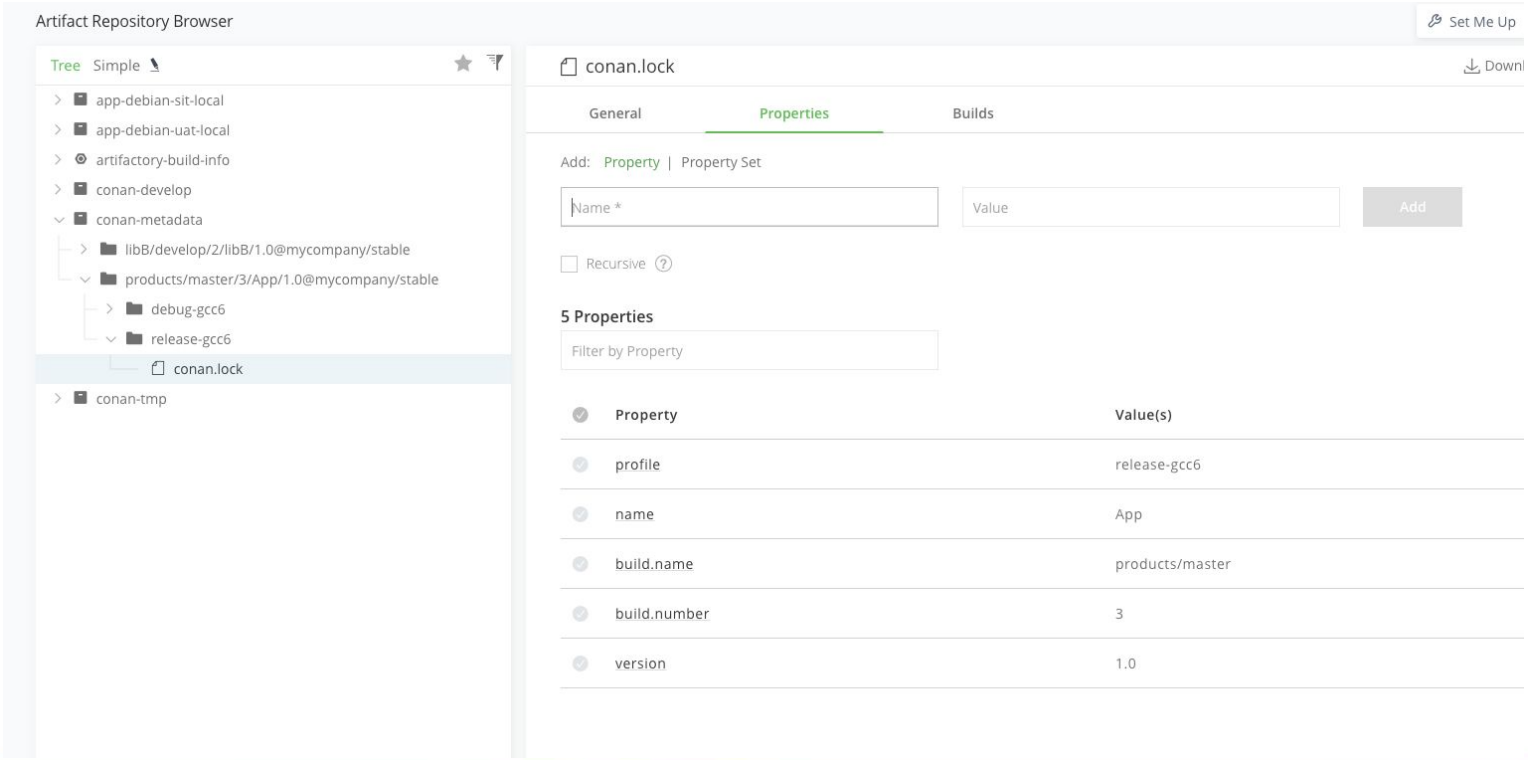


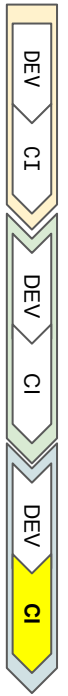




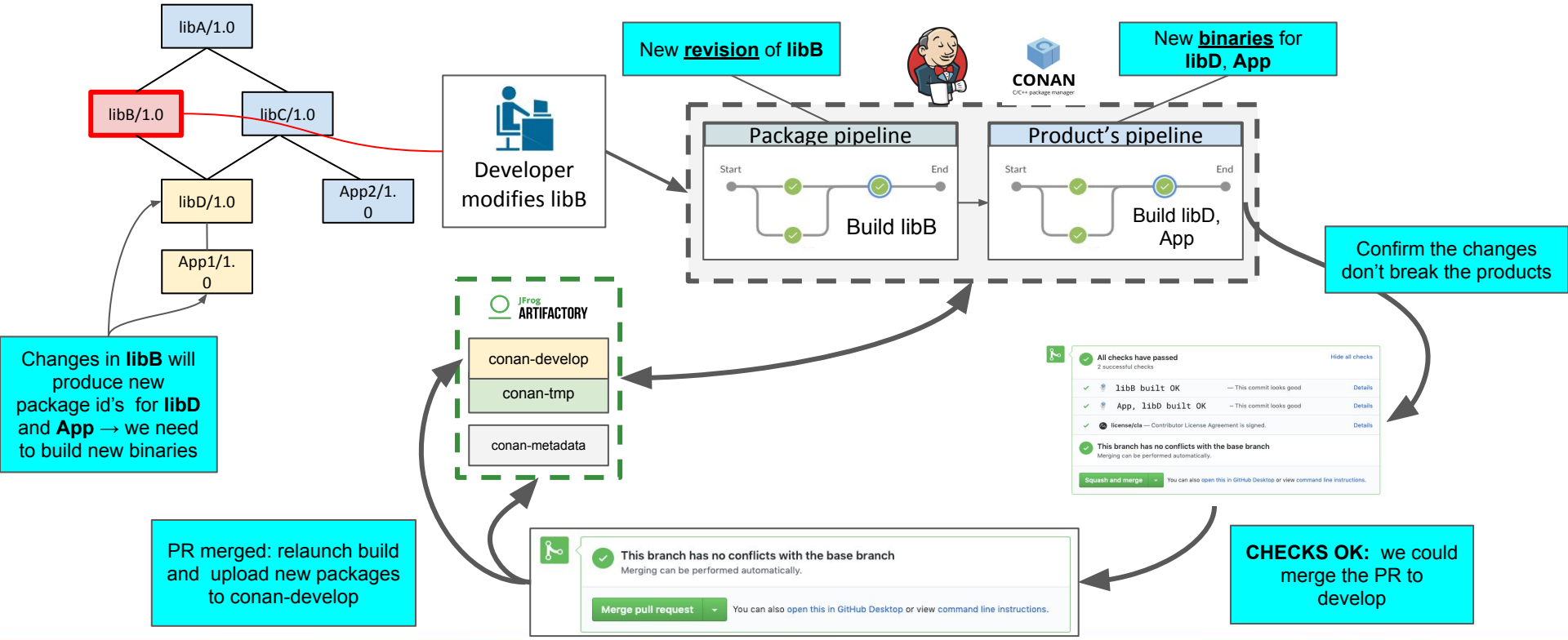


[**Phase 3 -** Products pipeline][Upload App lockfile to conan-metadata]





[Phase 3 - Summary]



But… what if libB breaks App?

- Until now we have been automatically integrating libB's changes without bumping versions, just using revisions
- If libB is introducing breaking changes then we should consider bumping the version of libB
- For the products pipeline the bump will not affect until App uses the new VERSION of libB

Outline

- Recap from Advanced Training
- Introduction: The Story
- CI Workflow: Phase 1
- CI Workflow: Phase 2
- CI Workflow: Phase 3
- **Artifactory: Build Info**
- Artifactory: Promotion
- Summary
- Appendix

Build Info

- Bill Of Material (JSON file):
  - List of generated binaries
  - Includes all consumed dependencies
  - You can use it to promote artifacts between repositories
- Can be published by CI plugins and JFrog CLI
- Conan can handle build info files (conan_build_info --v2)
  - Generate build info from a lockfile
  - Merge several build infos from different lockfiles
  - Publish generated build infos to Artifactory

Build Info - WARNING

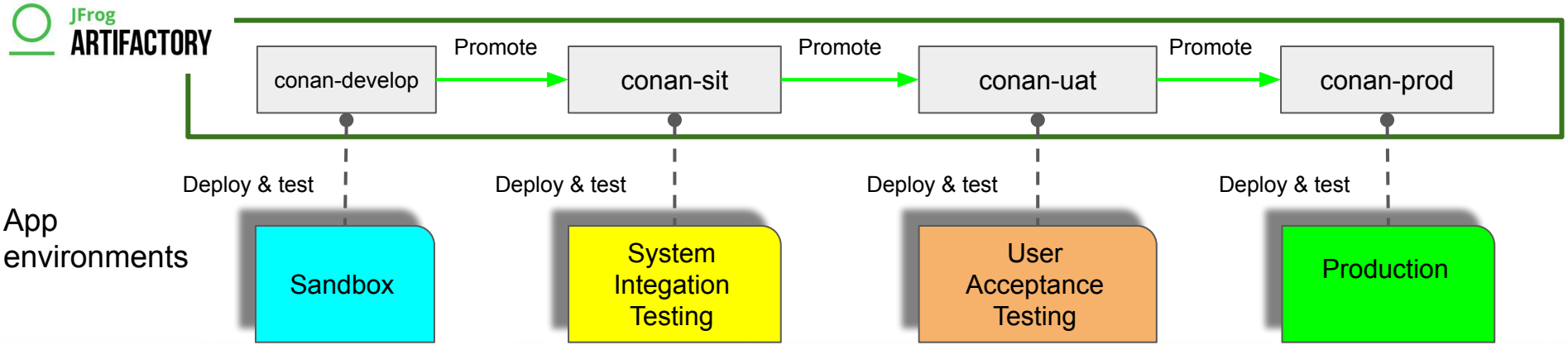
- Initially covering Java use case
- Doesn’t FULLY support some use cases including **Conan**
  - **We don’t recommend to use it for Conan for now**
  - See Appendix for :
    - How to create a Conan Build Info
    - Build Info limitation
- It’s possible to create a custom Build Info where result of a build and dependencies are specified manually

Outline

- Recap from Advanced Training
- Introduction: The Story
- CI Workflow: Phase 1
- CI Workflow: Phase 2
- CI Workflow: Phase 3
- Artifactory: Build Info
- **Artifactory: Promotion**
- Summary
- Appendix

Promotion mechanism

- Monitor your binaries during the delivery process
- The component lifecycle is represented by a chain of repositories
- Consist in copying/moving a single or group of artifacts from a source repository to a target repository



Promotion mechanism

- Triggered automatically (CI/CD tool) or manually after passing a test in the delivery process
- 2 types of promotions
  - Artifact(s) promotion = copy or move 1 or more artifact
  - Build promotion = copy or move artifacts from a Build Info
    - Promotion status
    - Promote generated artifacts with or without dependencies

Promotion process of App/1.0

- Configure the JFrog CLI
- Locate the lockfile we used to build the App we want to promote
  - Download the artifacts of the App using the lockfile
- Create a debian package for the App
- Create a build info for the debian package
- Promote the debian package from SIT to UAT using the build info

JFrog CLI

- Lightweight tool running on the following OS : linux, windows, mac
- Optimized for massive actions : upload, download, search, update, move, copy, delete
- Checksum aware on uploads and downloads:
  - Compute the checksum of the binary to upload and send it in the header request
  - Only upload binaries which checksum doesn’t exist in the Artifactory DB
- Easy way to manage Build Info

[Lab6] Configure the JFrog CLI

Goal:

- Connect the JFrog CLI to Artifactory

Task:

- Configure the JFrog CLI and search the repo content

Success:

- Test connection by listing the conan-metadata content

[Lab 6] Configure the JFrog CLI

```
cd /promotion_labs/

jfrog rt c --interactive=false  --url=http://jfrog.local:8081/artifactory
--user=conan --password=conan2020 art7

# show current art7 profile
jfrog rt c show

# test connection by listing the repo content
jfrog rt search conan-metadata/
```

[Lab 7] Download App based on properties

Goal:

- Use AQL (*) to retrieve a lockfile based on its properties (build.name, build.number, profile)
- Use the Conan Deploy Generator to deploy files locally

Task:

- Download a lockfile based on properties using AQL in a filespec

Success:

- App is deployed to the local folder and run it successfully

* Artifactory Query Language : see Appendix for more details

[Lab 7] Download App based on properties

```
# show filespec based on AQL
cat automation/filespec.json


# download lockfile based on properties + output “success”
jfrog rt download --spec=automation/filespec.json

# “deploy” the package referenced in the lockfile in the current path
conan install App/1.0@mycompany/stable --lockfile App-release-gcc6.lock -g deploy -r conan-develop
--update

ls -l App/

# execute the deployed App
./App/bin/App
```

[Lab 8] Create and upload a debian package

Goal:

- Create and upload a debian package

Task:

- Create a debian package from the App binary
- Upload the debian package to Artifactory

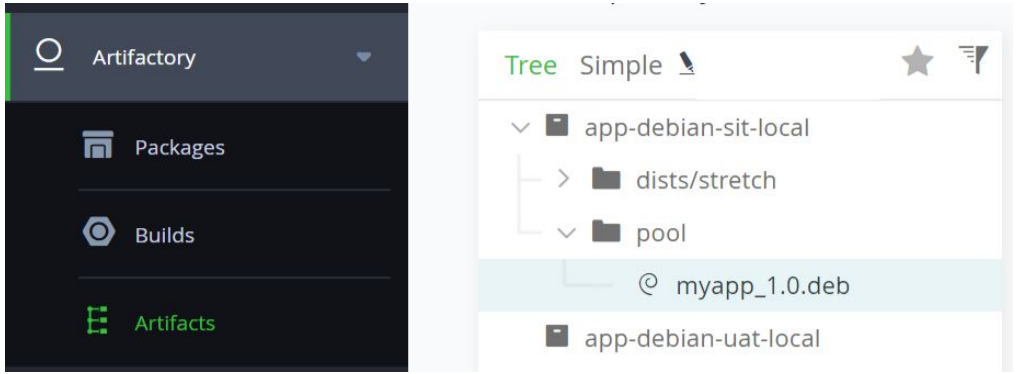
Success:

- Check the Debian package in Artifactory

[Lab 8] Create and upload a debian package

2:00

```
./generateDebianPkg.sh conan conan2020
```



[Lab 9] Create a custom Build info

Goal:

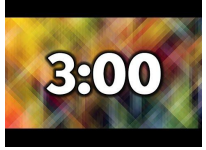
- Create a Build Info using the **JFrog CLI** which can then be promoted by the ops team

Task:

- Create and publish a **custom build info** :
  - Artifact section : debian package
  - Dependencies section : lockfile for the App
- Publish the Build Info

Success:

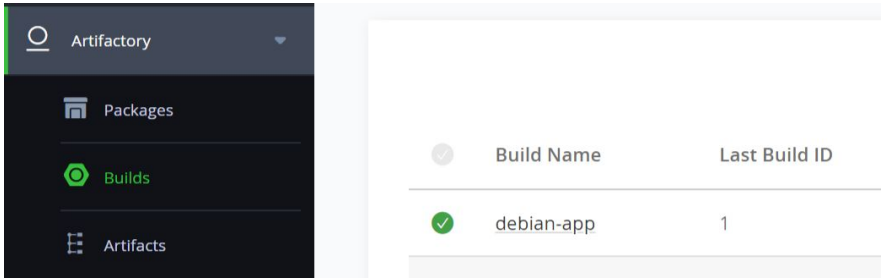
- Check the Build Info in Artifactory



[Lab 9] Create a custom Build info

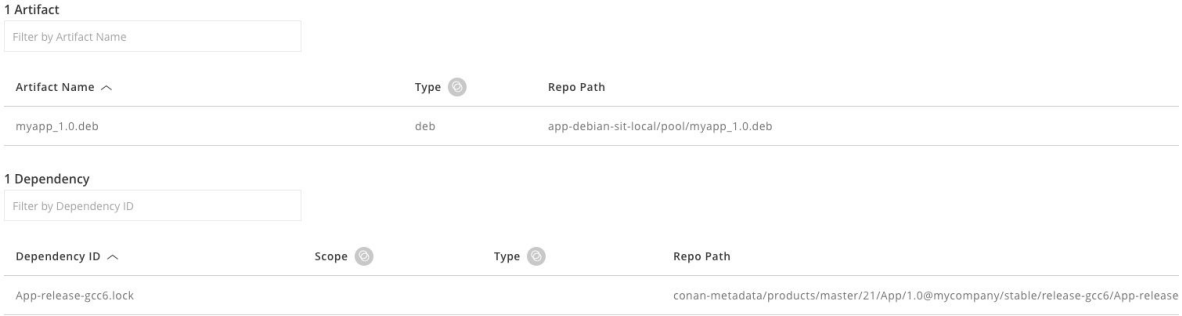
```
# define “artifact section” in build info
# won’t be re-uploaded as the JFrog CLI is checksum aware => output “status”:“success”
jfrog rt u debian_gen/myapp_1.0.deb app-debian-sit-local/pool/ --build-name=debian-app
--build-number=1

# define “dependency section” in build info => output “status”:“success”
jfrog rt bad debian-app 1 App-release-gcc6.lock

# publish build info => check result in Artifactory in the build section
jfrog rt bp debian-app 1
```

[Lab 9] Create a custom Build info



Go to “Build” section and select debian-app

Check the Build Info content



[Lab 10] Build Info Promotion

Goal:

- Promote Build Info by move without dependencies using the **JFrog CLI**

Task:

- Use jfrog rt bpr (build promote) instruction

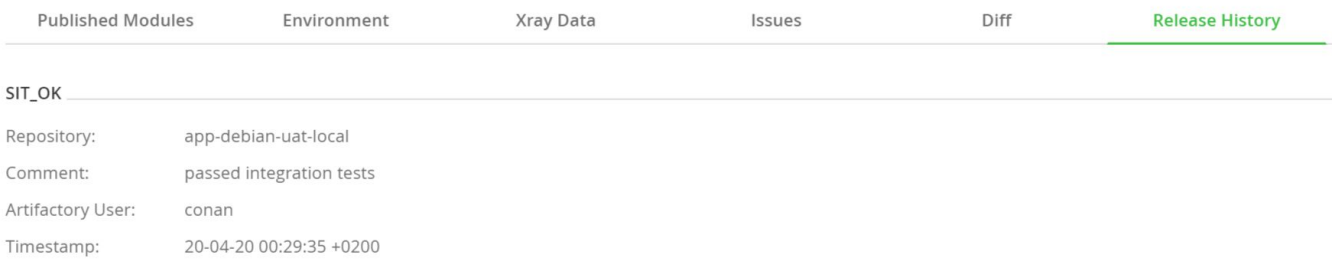
Success:

- See the Build Info Promotion in Artifactory
  - Check path in “published modules” tab
  - Check “Release history” tab

[Lab 10] Build Info Promotion

```
jfrog rt bpr debian-app 1 app-debian-uat-local --status="SIT_OK"
--comment="passed integration tests" --include-dependencies=false --copy=false
```

Check the Build Info content and Release History tab



Promotion - Good to know

- When promoting by copy :
  - This will create more artifacts (not binaries)
  - Any AQL and filespec have to target a repository name
- Build Info promotion with / without dependencies
  - Depends on your project structure and delivery process
- Limitation : A unique target repository

Outline

- Conan reminder
- Introduction: The Story
- CI Workflow: Phase 1
- CI Workflow: Phase 2
- CI Workflow: Phase 3
- Artifactory: Build Info
- Artifactory: Promotion
- **Summary**
- Appendix

[Last Lab] Homework :)

Have a look at the different Jenkinsfiles in this github organization:
https://github.com/conan-ci-cd-training

- Package pipeline:
  https://github.com/conan-ci-cd-training/libB/blob/develop/Jenkinsfile
- Products pipeline:
  https://github.com/conan-ci-cd-training/products/blob/master/Jenkinsfile
- Debian package promotion process:
  https://github.com/conan-ci-cd-training/release/blob/master/Jenkinsfile

Summary

- Use different Artifactory repos
  - conan-tmp: exchange repo
  - conan-develop: storing binaries for developers to consume and for CI builds
  - conan-metadata (generic repo): store lockfiles
- Revisions + recipe_revision_mode → “automatic versioning” to integrate your changes quickly
- Use lockfiles
  - For reproducibility: calculate the build order of a graph with fixed recipe revisions and install binaries
  - Use them to promote packages between repos
- Always use config install to have the same configuration in all Conan clients
- Properties
  - To retrieve easily artifacts based on specific criterias
- Build promotion
  - Monitor your binaries via a chain of repositories in Artifactory
  - Should reflect your own delivery process

Resources

- Docs: https://docs.conan.io/
  - Read carefully, explore.
- Issues:
  - CppLang slack (community)
  - Github issues (https://github.com/conan-io/conan) “official” support
- Following trainings:
  - conandays@jfrog.com
- Other Conan questions?
  - info@conan.io
- Twitter:
  - @conan_io

THANK YOU !

Outline

- Conan reminder
- Introduction: The Story
- CI Workflow: Phase 1
- CI Workflow: Phase 2
- CI Workflow: Phase 3
- Artifactory: Build Info
- Artifactory: Promotion
- Summary
- **Appendix**

Conan features

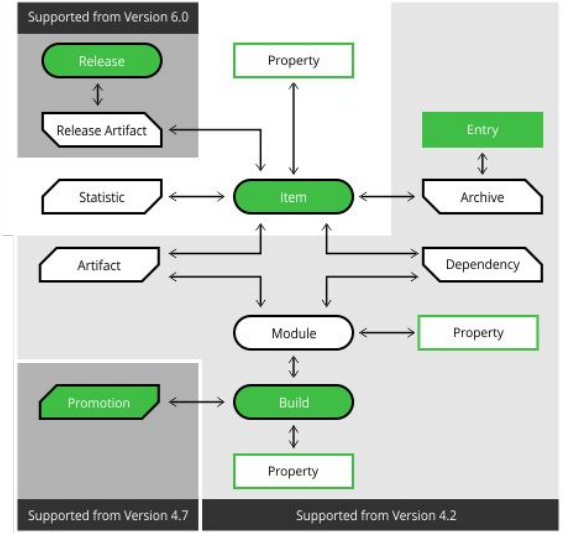
- Revisions
  - https://docs.conan.io/en/latest/versioning/revisions.html
- Package ID mode
  - https://docs.conan.io/en/latest/creating_packages/define_abi_compatibility.html#versioning-schema
- Custom Package ID
  - https://docs.conan.io/en/latest/creating_packages/define_abi_compatibility.html
- Lockfiles
  - https://docs.conan.io/en/latest/versioning/lockfiles.html#versioning-lockfiles
- Versioning
  - https://docs.conan.io/en/latest/versioning/introduction.html
- Conan Build Info client
  - https://docs.conan.io/en/latest/reference/commands/misc/conan_build_info.html

Artifactory features

- Checksum based storage
  - https://www.jfrog.com/confluence/display/JFROG/Checksum-Based+Storage
- Properties
  - https://www.jfrog.com/confluence/display/JFROG/Using+Properties+in+Deployment+and+Resolution
- Build Info
  - https://www.jfrog.com/confluence/display/JFROG/Build+Integration
- Promotion
  - https://jfrog.com/knowledge-base/how-does-build-promotion-work/
- JFrog CLI
  - https://www.jfrog.com/confluence/display/CLI

Automation with AQL

- Artifactory Query Language ~ SQL for Artifactory
- JSON formatted requests and responses
- String, Date, Time operators
- Sorting, limiting results
- Non admin can only use item domain



List artifact of a Build Info

build_info_artifacts.json

```
builds.find({
    "name": "app1",
    "number": "2",
}).include("module.artifact.item.name", "module.artifact.item.path")
```

```
# with creds or access token
curl -uadmin:<PASS> -XPOST -T build_info_artifacts.json
http//jfrog.local:8081/artifactory/api/search/aql
```

List dependencies filtered on property

build_info_deps.json

```
builds.find({
    "name": "app1",
    "number": "2",
    "module.artifact.dependency.@conan.settings.os" : "Linux"
}).include("module.dependency.item.name", "conan.settings.build_type",
"module.dependency.item.path")
```

```
# with creds or access token
curl -uadmin:<PASS> -XPOST -T build_info_deps.json
http//jfrog.local:8081/artifactory/api/search/aql
```

List artifacts based on a property value

artifact_search.json

```
items.find({
    "repo": "conan-develop",
    "name": "conaninfo.txt",
    "$or": [
         { "@conan.settings.os": "Linux" }, { "@conan.settings.os": "Windows" }
     ]
}).include("repo","path","name","@conan.settings.os","@conan.settings.arch","@cona
n.settings.build_type")
```

```
# with creds or access token
curl -uconan:conan2020 -XPOST -T artifact_search.json
http//jfrog.local:8081/artifactory/api/search/aql
```

Download a file using the CLI and filespec with AQL

automation/filespec.json

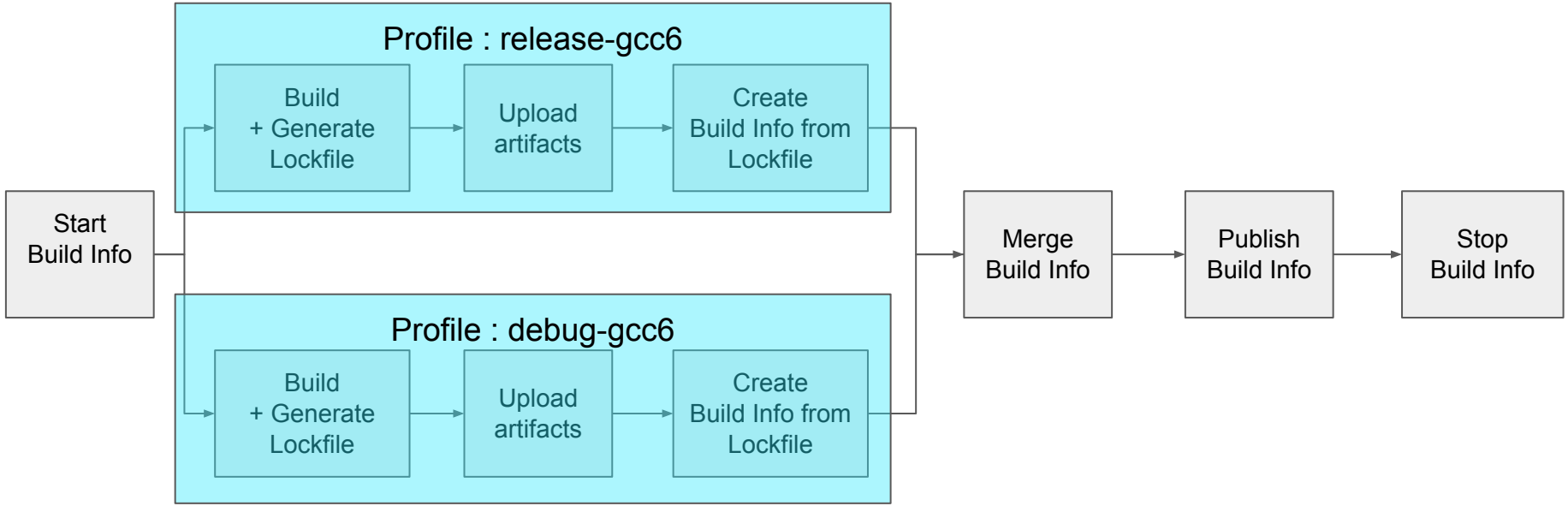
```
{
  "files": [{
    "aql": {
      "items.find": {
        "repo": "conan-metadata",
        "name" : { "$match" : "*.lock"},
        "$and": [
          { "@build.name": "conan-app" }, { "@build.number": "1" }
        ]}
  }}]
```

```
# JFrog CLI should have be configured before
jfrog rt download --spec=automation/filespec.json
```

Build Info - General explanation

- An artifact in the “**Artifacts” section** is located if the following requirements are met :
  - Checksum/hash exists in the Artifactory DB
  - Build properties set on the artifacts
- An artifact in the **“Dependencies” section** is “located” if
  - its checksum/hash exists in the Artifactory DB
- No artifact upload = no Build properties assigned to the artifact

Conan Build Info in parallel pipelines (1/3)



Conan Build Info in parallel pipelines (2/3)

```
# disable/enable build properties
conan_build_info --v2 stop && cat ~/.conan/artifacts.properties
conan_build_info --v2 start conan-app 1 && cat ~/.conan/artifacts.properties

# create build info for release from the release lockfile for App1
conan_build_info --v2 create release_bi.json --lockfile=app_release.lock --user=conan --password=conan2020 &&
cat release_bi.json

# generate libs in Debug + upload App in Debug
# current path : ~/conan_ci_cd/labs
./genAppDebug.sh

# create build info
conan_build_info --v2 create debug_bi.json --lockfile=app_debug.lock --user=conan --password=conan2020 && cat
debug_bi.json
```

Conan Build Info in parallel pipelines (3/3)

```
# create the aggregated build info
conan_build_info --v2 update --output-file app_bi.json debug_bi.json release_bi.json && cat app_bi.json

# publish the build info and remove build properties
conan_build_info --v2 publish app_bi.json --url=http://jfrog.local:8081/artifactory --user=conan
--password=conan2020

conan_build_info --v2 stop && cat ~/.conan/artifacts.properties
```

Build Info - Limitation

- MAY NOT fit the use case when :
  - An artifact is referenced by multiple Build Info (like unchanged recipe)
  - An artifact is NOT considered as a Build Info dependency
- Possible workaround :
  - All the files from the Artifact section should be packaged into an archive which will be the result of your Build Info
- Stay tuned about Build Info improvements !