An IBM Document from

IBM Z DevOps Acceleration Program

February, 2021 - Release 1.1

Managing the build scope in IBM DBB builds with IBM zAppBuild

Brice Small
brice@ibm.com

Mathieu Dalbin
mathieu.dalbin@fr.ibm.com

Dennis Behm
dennis.behm@de.ibm.com

Abstract
Documentation of different approaches to manage a build scope across multiple repositories

Table of content

Table of Contents

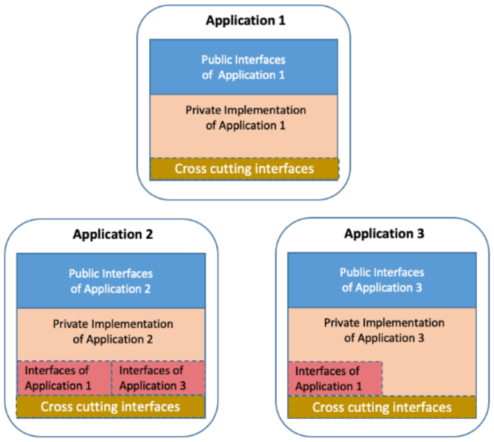
1 Introduction

For open and modern CI/CD pipeline implementations for Mainframe, existing material (whitepapers and Techdocs) have already been released to discuss about the scope of an application component, the classification of private and public elements, as well as the different adoption strategies for shared interfaces defining the communication area for a service[1].

1.1 The application scope

Each application is typically related to a business component. For Mainframe development, naming conventions are used to help to define the boundary of an application, as well as the ownership within a centralized version control system approach. These information are important for developers to navigate within the code base or during a root cause analysis.

Applications certainly interact – some applications provide a service to other applications, like the management of an account. Others will use that service and are therefore called the consumer.


Application 1
Public Interfaces of Application 1
Private Implementation of Application 1
Cross cutting interfaces
Application 2
Public Interfaces of Application 2
Private Implementation of Application 2
Interfaces of Application 1
Interfaces of Application 3
Cross cutting interfaces
Application 3
Public Interfaces of Application 3
Private Implementation of Application 3
Interfaces of Application 1
Cross cutting interfaces

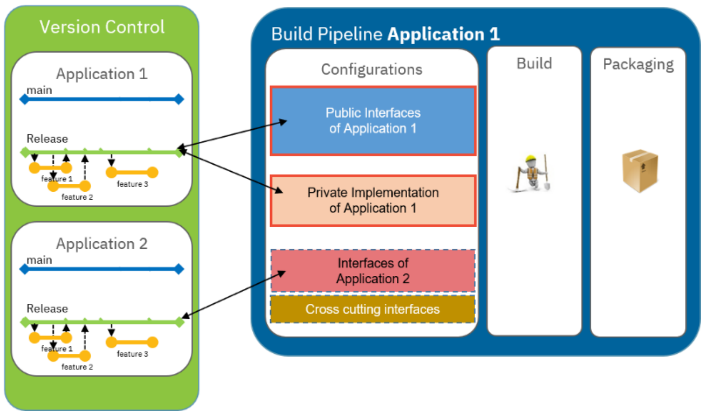
In a Cobol environment, a copybook is typically used to describe the communication structure and is provided by the application implementing the service. The provider needs to share the interface definition (the copybook) with the consumers, so they know how to use the service. Consuming programs typically include the published copybook to invoke the corresponding service.

Additionally, to successfully build the consuming application, the copybooks describing the communication structures must be available to the Cobol compiler.

Certainly, building the application includes managing the build scope of an application.

1.2 The build scope

The build scope defines the list of application programs and copybooks to be included in the build process. It can also be called the build configuration, as it is linked to the specific properties used in the build process and stored in the git repositories.



In this diagram, Application 1 uses a service provided by Application 2. The build scope for building Application 1 requires access to the copybooks of Application 2.

[1] https://ibm.github.io/mainframe-downloads/DevOps_Acceleration_Program/resources.html

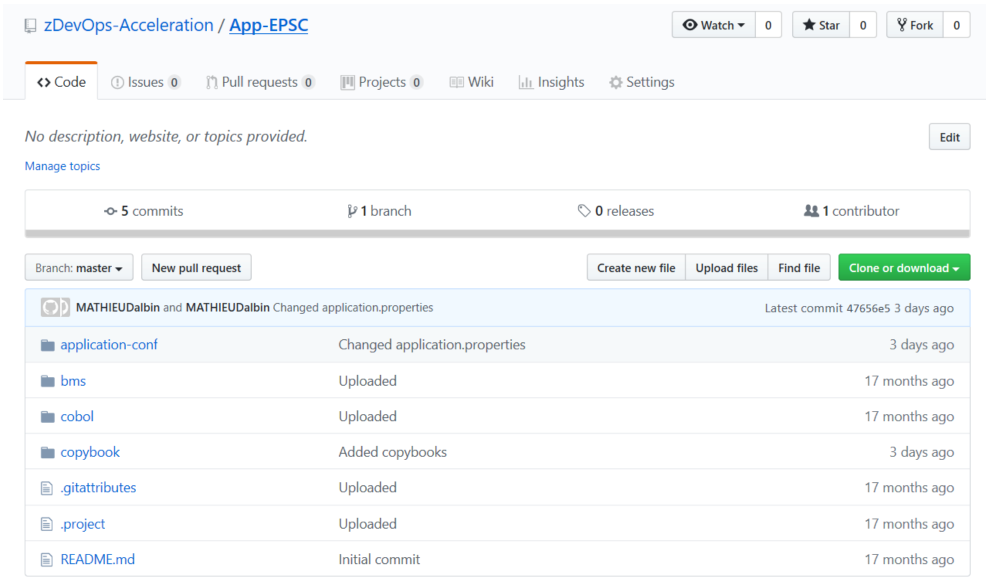
Considering Git as the version control system, this document will discuss several strategies to include the copybooks in the build scope for building the applications that required them. We will relate them to the different adoption strategies of interfaces.

1.3 Base scenario

For this document, the existing Mortgage sample application shipped with zAppBuild has been separated into two application repositories.

The two repositories are assumed to represent two isolated applications, each with their own release planning and release cycles. Each application has its own CI/CD pipeline definition.

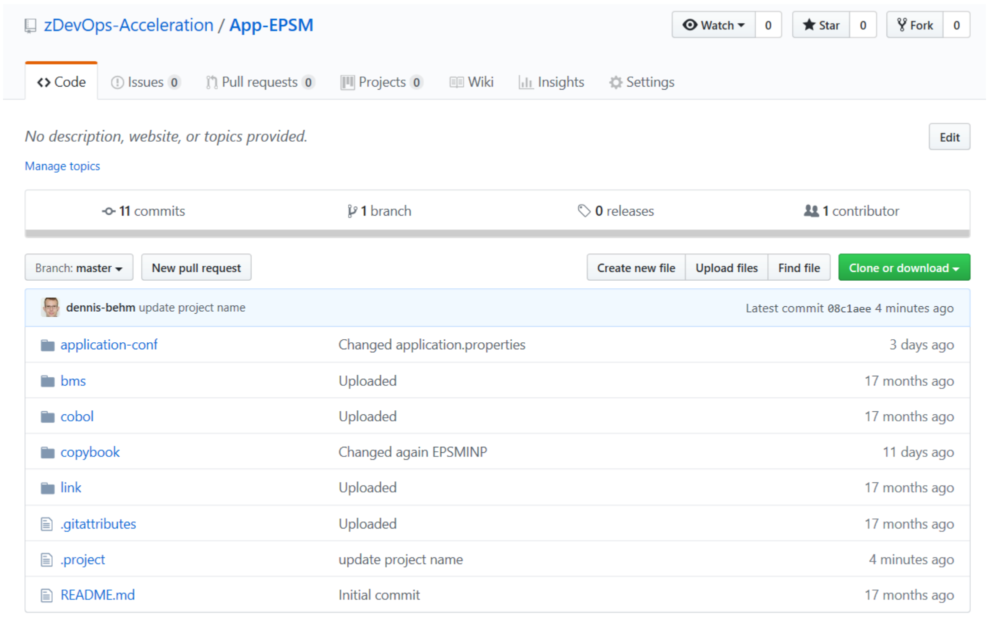
We will reference the required configuration for pipelines based on Jenkins and GitLab CI/CD in this document. However, depending on your CI orchestrator the configuration will look different, but the same strategies can be applied.



MATHIEUDalbin and MATHIEUDalbin Changed application.properties — Latest commit 47656e5 3 days ago

application-conf	Changed application.properties	3 days ago
bms	Uploaded	17 months ago
cobol	Uploaded	17 months ago
copybook	Added copybooks	3 days ago
.gitattributes	Uploaded	17 months ago
.project	Uploaded	17 months ago
README.md	Initial commit	17 months ago

Application Repository of EPSC



dennis-behm update project name — Latest commit 08c1aee 4 minutes ago

application-conf	Changed application.properties	3 days ago
bms	Uploaded	17 months ago
cobol	Uploaded	17 months ago
copybook	Changed again EPSMINP	11 days ago
link	Uploaded	17 months ago
.gitattributes	Uploaded	17 months ago
.project	update project name	4 minutes ago
README.md	Initial commit	17 months ago

Application Repository of EPSM

For all the subsequent scenarios described in this document, these common rules are in effect:

- Application 1 (EPSC) consumes services provided by Application 2 (EPSM).
- To successfully build Application 1 (EPSC), it requires access to the interfaces of Application 2 (EPSM).

1.4 Overview of scenarios

For the different scenarios, the purpose of the discussion will be to highlight the necessary configuration of the build process. The main differences in these scenarios will reside in the build configuration to pull in the necessary copybooks from EPSM and include them in the build scope of the consuming EPSC application.

*In **Scenario 1** the build configuration directly accesses the application repositories providing the service.*

*The build configuration in **Scenario 2** includes the copybooks through a dedicated repository for interfaces.*

***Scenario 3** addresses some limitations of Scenario 1 and Scenario 2 by introducing a publishing mechanism for shared interfaces.*

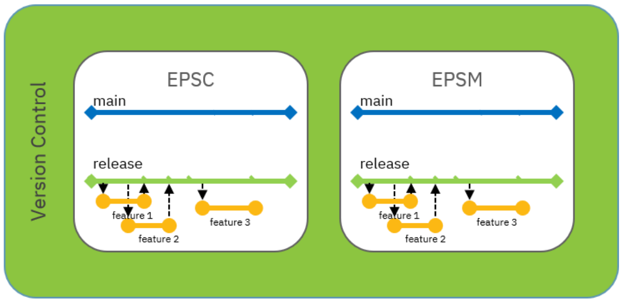
*While the build configuration for **Scenario 4** includes the copybooks through the file system – dataset libraries.*

Scenarios 1, 2 and 3 can leverage the impact build setup of zAppBuild, to automatically identify the files changed between 2 versions of the applications' repositories. The impact build feature will calculate the impact of the changes within the build scope: impacted programs will be automatically be added to the build list.

2 Scenario 1 - Build configuration references dependent applications repositories directly

In this scenario, the build configuration directly references the git repositories of the dependent application(s).

Each application (EPSC and EPSM) is managed in dedicated git repositories, containing the application programs and copybooks.



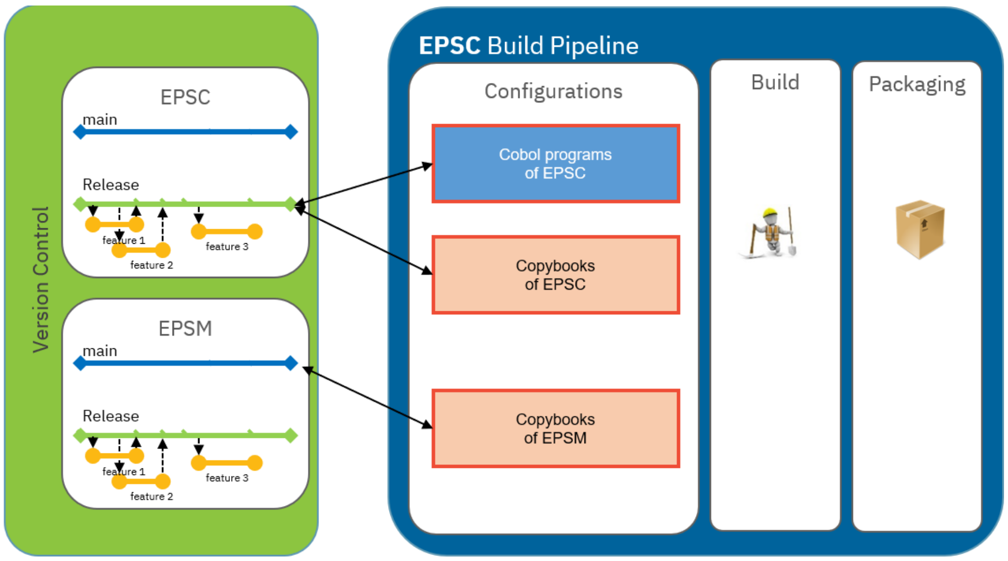
In a GitLab setup, each application (EPSC and EPSM) resides in its own GitLab project, which belong to a unique group *Multiple-repo* in this setup used for illustrating the scenario. The zAppBuild framework is hosted in its own project in GitLab and possess its own pipeline, primarily used for updating the build workspace.

2.1 Simple scenario

2.1.1 Build scope configuration

For building the EPSC application, we need to make sure that the source files are available for the build process, and that we correctly configure the build framework zAppBuild:

- the EPSC pipeline configuration requires to pull in the copybooks of the EPSM repository,
- Application-related configuration in *application-conf* properties for zAppBuild.



2.1.1.1 Build workspace with Jenkins

In a Jenkins based build setup, the Jenkins pipeline is configured to check out the application repositories of EPSC and EPSM to the build workspace for the EPSC build pipeline.

Within a Jenkinsfile, this translates into a checkout for each repository:

```
stage('Git Checkout') {
        dir (AppEPSC) {
            checkout([$class: 'GitSCM', branches: [[name: GitBranch]],
doGenerateSubmoduleConfigurations: false, submoduleCfg: [], userRemoteConfigs:
[[credentialsId: gitCredId, url: 'git@github.ibm.com:zDevOps-Acceleration/App-EPSC.git']]])
        }
        dir (AppEPSM) {
            checkout([$class: 'GitSCM', branches: [[name: GitBranch]],
doGenerateSubmoduleConfigurations: false, submoduleCfg: [], userRemoteConfigs:
[[credentialsId: gitCredId, url: 'git@github.ibm.com:zDevOps-Acceleration/App-EPSM.git']]])
        }
    }
```

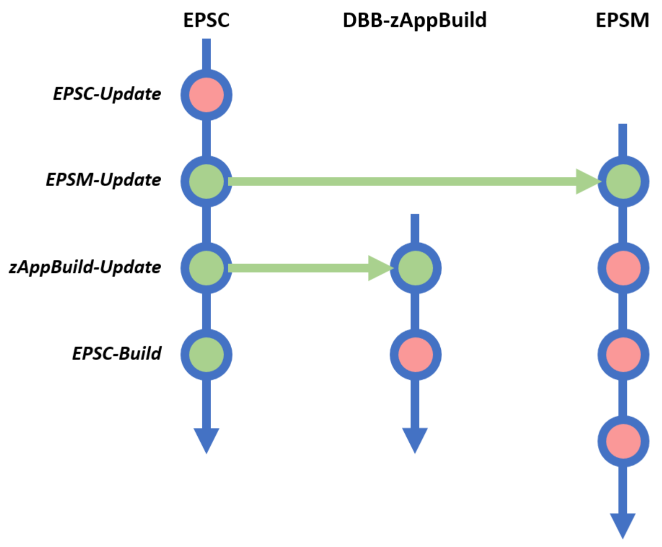
2.1.1.2 Build workspace with GitLab CI/CD

In this configuration, the multiple-project feature of GitLab CI/CD is leveraged, to trigger pipelines of other projects. Executing a pipeline in GitLab CI/CD implicitly contains the checkout step. So, triggering a downstream pipeline helps to update the build workspace with content from other projects.

The pipeline configuration of the project *EPSC* needs to reference and depends on the pipelines of *zAppBuild* and *EPSM*. Leveraging this behavior, EPSC's pipeline will trigger the execution of zAppBuild's and EPSM's pipelines to checkout zAppBuild and EPSM repositories into the build workspace.

However, these downstream pipelines should not execute all their steps, as they contain their own build stages. This requires the use of conditions for some steps, which can be achieved with the *except* and *only* keywords[2] in the stages in a pipeline.

The execution process of EPSC's pipeline is described in the following diagram, while the stages highlighted in green are performed. Please note that the EPSC-Update stage is skipped, as it is a dummy stage, that is executed only when the pipeline if triggered from other upstream pipelines.



[2] GitLab CI reference: https://docs.gitlab.com/ee/ci/yaml/

The pipeline of *EPSC* is defined in the following *.gitlab-ci.yml* file:

```
variables:
    DBB_HOME: "/usr/lpp/dbb/v1r0"
    ZAPPBUILD: "dbb-zappbuild"
    WORKDIR: "/u/gitlab/workspace/$CI_PROJECT_NAME"
    WORKSPACE: "/u/gitlab/${CI_BUILDS_DIR}/${CI_RUNNER_SHORT_TOKEN}/${CI_CONCURRENT_ID}"

EPSC-Update:
  stage: Preparation
  only:
    variables:
      - $CI_PIPELINE_SOURCE == 'pipeline'
  script:
    - echo "Updating EPSC"

zAppBuild-Update:                          // <<<--- Update Workspace with zAppBuild
  stage: Preparation
  except:
    variables:
      - $CI_PIPELINE_SOURCE == 'pipeline'
  trigger:
    project: DAT/dbb-zappbuild
    strategy: depend

EPSM-Update:                               // <<<--- Update Workspace with EPSM Repo
  stage: Preparation
  except:
    variables:
      - $CI_PIPELINE_SOURCE == 'pipeline'
  trigger:
    project: multiple-repo/App-EPSM
    branch: master
    strategy: depend

EPSC-Build:
  stage: Build
  except:
      variables:
      - $CI_PIPELINE_SOURCE == 'pipeline'
  script:
      - cd /u/gitlab/${CI_BUILDS_DIR}/${CI_RUNNER_SHORT_TOKEN}/${CI_CONCURRENT_ID}/multiple-
repo/
      - $DBB_HOME/bin/groovyz -Djava.library.path=$DBB_HOME/lib:/usr/lib/java_runtime64
$WORKSPACE/dat/${ZAPPBUILD}/build.groovy --workspace $WORKSPACE/multiple-repo --workDir
$WORKDIR/BUILD-$CI_PIPELINE_ID --hlq GITLAB.EPSC.MASTER --logEncoding UTF-8 --application App-
EPSC --verbose --impactBuild

stages:
  - Preparation
  - Build
```

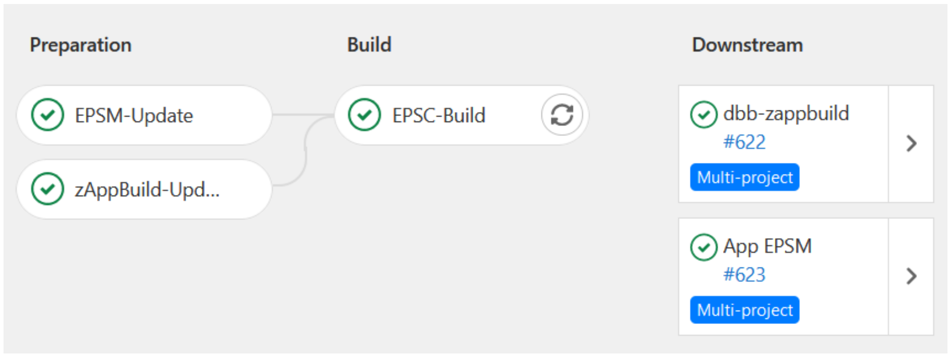
In a typical setup using GitLab, the zAppBuild framework is contained in its own project and possess a pipeline definition. The pipeline configuration of the *dbb-zappbuild* project (described in its *.gitlab-ci.yml* file) contains steps that are only executed when the pipeline is triggered from an upstream pipeline. In this case, the pipeline stage *Preparation* is executed:

```
stages:
    - Preparation

Preparation:
    stage: Preparation
    only:
      variables:
        - $CI_PIPELINE_SOURCE == 'pipeline'
    script:
       - echo "Updating zAppBuild Framework"
```

In the pipeline definition file *.gitlab-ci.yml* for EPSM, the process will only execute the dummy stage (*EPSM-Update*), because the pipeline is triggered from an upstream pipeline. All the other stages will be skipped.

```
..
EPSM-Update:
  stage: Preparation
  only:
    variables:
      - $CI_PIPELINE_SOURCE == 'pipeline'
  script:
    - echo "Updating EPSM"
..
```

For EPSC, the following execution diagram shows the dependencies to the other downstream pipelines:



2.1.1.3 zAppBuild configuration

The build framework zAppBuild is invoked using the parameter *--application App-EPSC*, to indicate to build the application configuration for App-EPSC. This name will also be used for the DBB collections, build results etc.

Within the App-EPSC repository, the *application-conf* directory contains additional directives for the build framework. In *application.properties*, the build property *applicationSrcDirs* is set to include the copybooks of EPSM to the build scope of EPSC.

```
#
# Comma separated list all source directories included in application build. Supports both
absolute
# and relative paths.  Relative assumed to be relative to ${workspace}.
# ex: applicationSrcDirs=${application},/u/build/common/copybooks
applicationSrcDirs=${application},App-EPSM/copybooks
```

If your application has additional dependencies, you will need to list these within the *applicationSrcDirs* property to make sure, that the files are available in the build configuration as well.

Additionally, we need to specify the copybooks folder of EPSM in the *copybookRule* build property:

```
# Rule to locate Cobol copybooks
copybookRule = {"library": "SYSLIB", \
                  "searchPath": [ \
                    {"sourceDir": "${workspace}", "directory": "App-EPSC/copybooks"}, \
                    {"sourceDir": "${workspace}", "directory": "App-EPSM/copybooks "} \
                 ] \
}
```

2.1.2 Build scenario

For both Jenkins and GitLab CI/CD, the next build of EPSC will update the build workspace including the EPSM repository and verify, if any files in scope of the build configuration have changed. Based on the changes the impacted files are calculated:

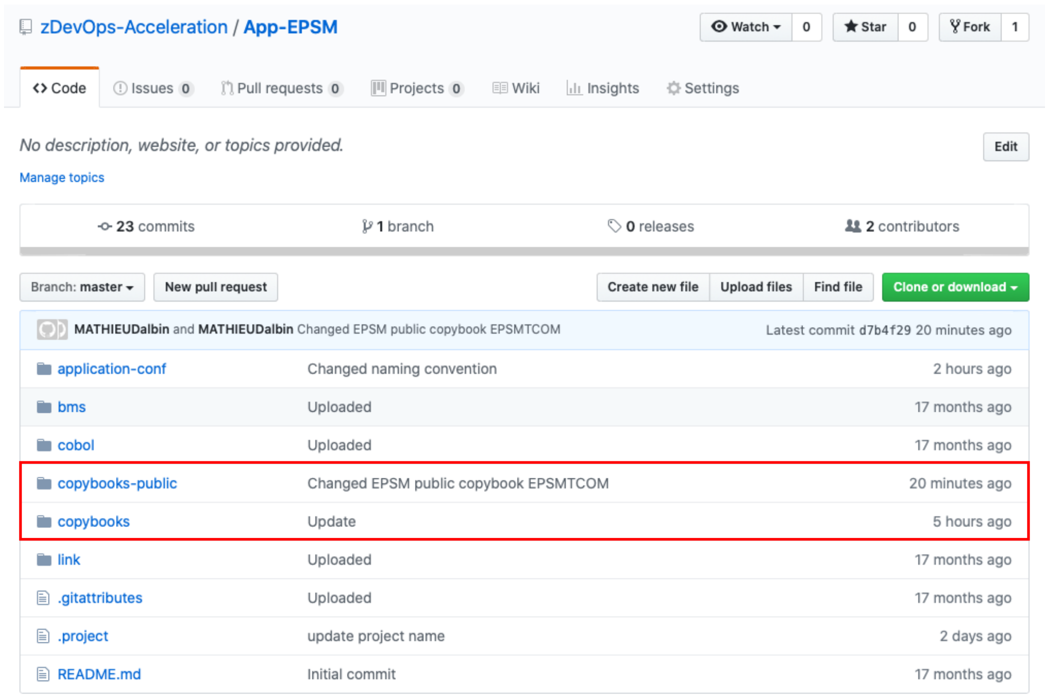
```
..
*** Changed files for directory /var/jenkins/workspace/App-EPSC/App-EPSM/copybook:
*** App-EPSM/copybook/epsmtcom.cpy
..
..
** App-EPSC/cobol/epscmort.cbl is impacted by changed file App-EPSM/copybook/epsmtcom.cpy.
Adding to build list.
** Writing build list file to /var/jenkins/workspace/App-
EPSC/outputs/build.20201201.020815.008/buildList.txt
App-EPSC/cobol/epscmort.cbl
..
```

2.2 Advanced scenario

The basic scenario described above is based on a simple architecture, where no distinction is made for copybooks: all the copybooks owned by an application are stored in the *copybooks* subfolder.

As observed in typical Mainframe applications' architectures, the application's interfaces are usually described in copybooks that can be qualified as *public*. These public copybooks are used to access services proposed by applications and are required by other applications in their build process. On the contrary, copybooks that are not used to define interfaces can be qualified as *private* and are only consumed by the owning application.

To reflect this classification, the applications' repositories for EPSC and EPSM were changed accordingly. For each application, the *copybooks* subfolder only contains the private copybooks and the *copybooks-public* subfolder contains the copybooks which can be used by other applications to use the provided services.



zDevOps-Acceleration / App-EPSM
Watch 0 Star 0 Fork 1
Code Issues 0 Pull requests 0 Projects 0 Wiki Insights Settings
Edit
Manage topics
Branch: master New pull request
Create new file Upload files Find file Clone or download

No description, website, or topics provided.

23 commits | 1 branch | 0 releases | 2 contributors

MATHIEUDalbin and MATHIEUDalbin Changed EPSM public copybook EPSMTCOM — Latest commit d7b4f29 20 minutes ago

application-conf	Changed naming convention	2 hours ago
bms	Uploaded	17 months ago
cobol	Uploaded	17 months ago
copybooks-public	Changed EPSM public copybook EPSMTCOM	20 minutes ago
copybooks	Update	5 hours ago
link	Uploaded	17 months ago
.gitattributes	Uploaded	17 months ago
.project	update project name	2 days ago
README.md	Initial commit	17 months ago

2.2.1 Build scope configuration

For a successful build, we need to make sure that the necessary source files are available for the build process.

- The build process for EPSC requires the public copybooks of EPSM to be available in the build workspace. The EPSC pipeline configuration requires to pull in the copybooks-public folder of the EPSM repository.
- The build configuration must reference the public copybooks of EPSM. This configuration must be reflected in the *application-conf* properties of EPSC when using zAppBuild.

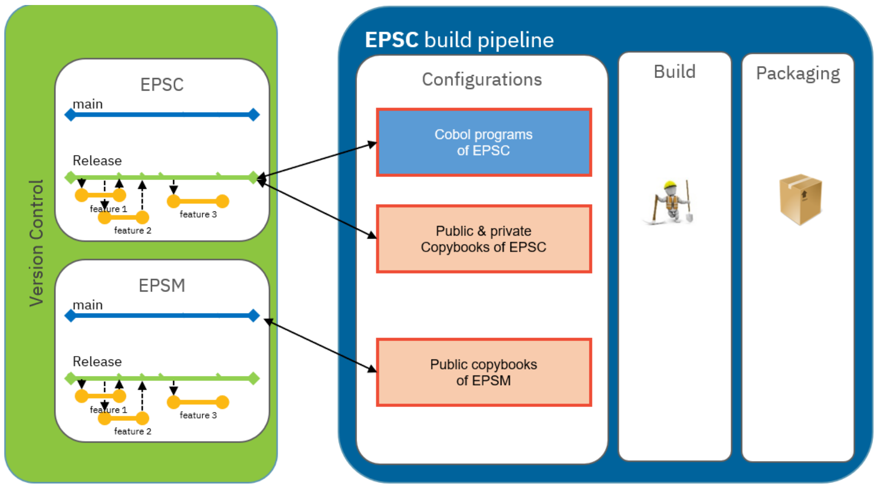
To reduce the scope of sources when pulling EPSM's repository and only retrieve the *copybooks-public* folder, a facility called *SparseCheckout* will be used in the pipeline.

This feature helps to limit the pull process and collect only the necessary artifacts from a git repository.

Unfortunately, this git feature does not exist in Gitlab – it will check out the entire repository for a project.

2.2.1.1 Build workspace

In this scenario, the Jenkins pipeline is configured to check out EPSC's repository and a subset of EPSM's repository: using the *SparseCheckout* feature, only the *copybooks-public* folder of EPSM will be transferred to the build workspace.



Compared to the previous scenario, the corresponding Jenkinsfile has changed to reflect the usage of SparseCheckout:

```
stage('Git Checkout') {
       dir (AppEPSC) {
              checkout([$class: 'GitSCM', branches: [[name: GitBranch]],
doGenerateSubmoduleConfigurations: false, submoduleCfg: [], userRemoteConfigs:
[[credentialsId: gitCredId, url: 'git@github.ibm.com:zDevOps-Acceleration/App-EPSC.git']]])
       }

       dir (AppEPSM) {
       sh(script: 'rm -f .git/info/sparse-checkout', returnStdout: true)
       checkout([$class: 'GitSCM', branches: [[name: GitBranch]],
doGenerateSubmoduleConfigurations: false,
extensions: [[$class: 'SparseCheckoutPaths',
  sparseCheckoutPaths:[ [path:'copybooks-public/']]]]],
submoduleCfg: [], userRemoteConfigs: [[credentialsId: gitCredId,
url: 'git@github.ibm.com:zDevOps-Acceleration/App-EPSM.git',]]])
       }
}
```

2.2.1.2 zAppBuild configuration

Like in the simple scenario, the build framework zAppBuild is invoked using the parameter *--application App-EPSC*. However, the build configuration must be adapted to reflect the change about copybooks subfolders.

In the *application.properties* file of EPSC's *application-conf* folder, the property *applicationsSrcDirs* must be modified to include the *copybooks-public* subfolder of EPSM for the build configuration.

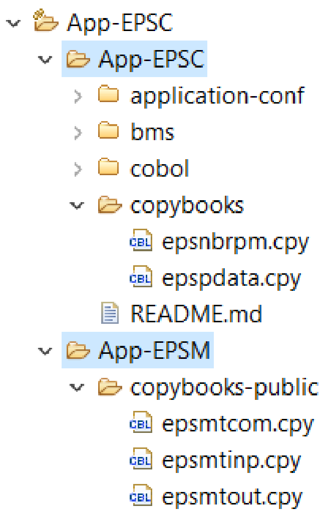
```
#
# Comma separated list all source directories included in application build. Supports both
absolute
# and relative paths.  Relative assumed to be relative to ${workspace}.
# ex: applicationSrcDirs=${application},/u/build/common/copybooks
applicationSrcDirs=${application},App-EPSM/copybooks-public
```

The property *copybookRule* needs to be updated accordingly:

```
# Rule to locate Cobol copy books
copybookRule = {"library": "SYSLIB", \
                  "searchPath": [ \
                    {"sourceDir": "${workspace}", "directory": "App-EPSC/copybook"}, \
                    {"sourceDir": "${workspace}", "directory": "App-EPSM/copybooks-public"} \
                 ] \
}
```

2.2.2 Build scenario

The next build of EPSC will update the build workspace including the EPSM's *copybooks-public* folder and verify, if any file in scope of the build configuration has changed.



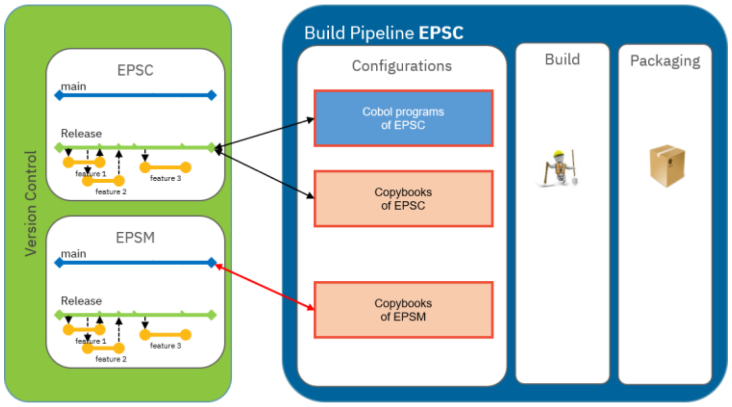
```
App-EPSC
  App-EPSC
    application-conf
    bms
    cobol
    copybooks
      epsnbrpm.cpy
      epspdata.cpy
    README.md
  App-EPSM
    copybooks-public
      epsmtcom.cpy
      epsmtinp.cpy
      epsmtout.cpy
```

Based on the changes the impacted files are calculated:

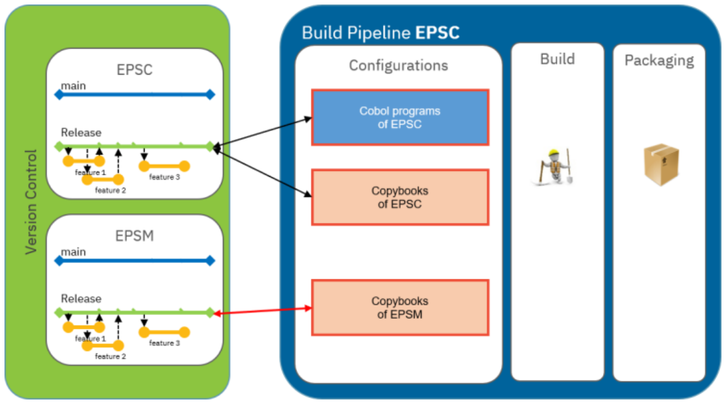
```
..
*** Changed files for directory /var/jenkins/workspace/App-EPSC/App-EPSM/copybooks-public:
*** App-EPSM/copybooks-public/epsmtcom.cpy
..
..
** App-EPSC/cobol/epscmort.cbl is impacted by changed file App-EPSM/copybooks-
public/epsmtcom.cpy. Adding to build list.
** Writing build list file to /var/jenkins/workspace/App-
EPSC/outputs/build.20201202.034639.046/buildList.txt
App-EPSC/cobol/epscmort.cbl
..
```

2.3 Adoption strategies for applications copybooks

Depending on type of change which the development teams are implementing, the build configuration can reference different branches that can be used for the required copybook structures. In the majority of the scenarios the changes will not be dependent, so, the *main branch* of EPSM (representing production) will be used to build EPSC:



In case the service provider EPSM has implemented a new functionality, that the consumer EPSC will immediately use, the build process for EPSC can be configured to pull in a release version of the copybooks. Both changes need to be released at the same time and are dependent. When the copybook changes are compatible, EPSC can also use the release branch to build their application.



2.4 Additional considerations

In the above scenario, the build configuration of EPSC would access the entire repository of EPSM. Naming conventions and rules need to be established to reduce access for the build configuration of EPSC.

For the adoption of copybooks, the consuming applications are immediately impacted as soon the provider commits the modified copybook to the release or main branch.

3 Scenario 2 - Build configuration includes dependent interfaces through dedicated repositories

Picking up the copybooks directly from the providing application repositories relies on a set of rules, which all systems need to follow.

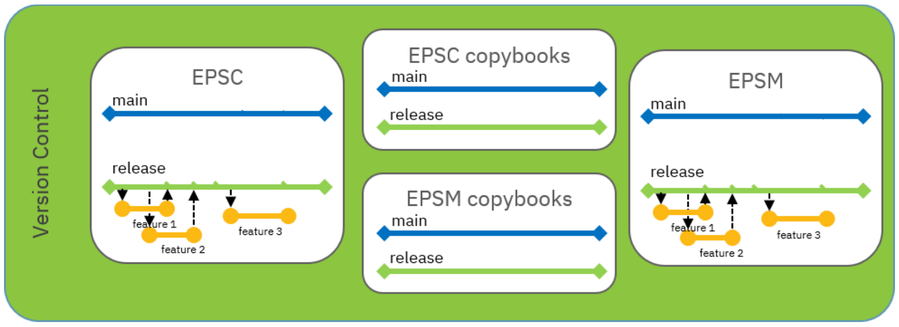
As already discussed in the previous section (2.2 Advanced scenario), a classification of the copybooks can be used to define the visibility of interfaces and private definitions. However, performing this classification is not a pre-requisite to implement the strategy of this scenario.

In this section, we will discuss the strategy of using dedicated repositories for copybooks. Compared to the first strategy, each application maintains two repositories:

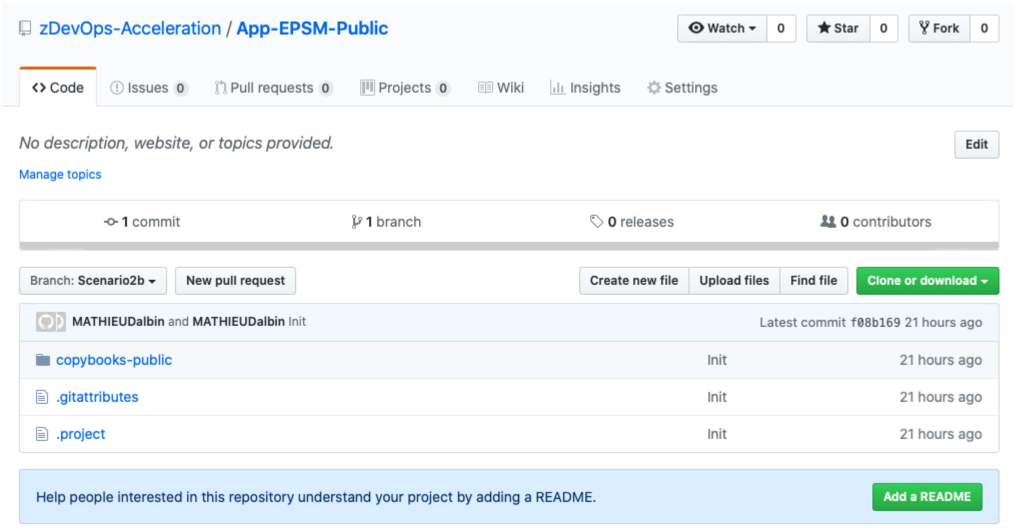
- A repository managing elements relevant to their implementation,
- A repository managing copybooks, which can be consumed by other applications.

3.1 Repository definitions

A dedicated repository for copybooks is maintained for each application. To extend this concept, copybooks can be classified: the private copybooks would be stored in the application repository along with the implementation, and the public copybooks which define the interfaces to the services would be stored in a separate repository:



To illustrate this setup, the EPSC's *public copybooks* identified in the previous scenarios were moved to a new, dedicated repository called App-EPSC-Public. The same change was applied to EPSM's *public copybooks*, which were moved to a dedicated repository called App-EPSM-Public:



zDevOps-Acceleration / App-EPSM-Public

Watch 0 | Star 0 | Fork 0

Code | Issues 0 | Pull requests 0 | Projects 0 | Wiki | Insights | Settings

No description, website, or topics provided.

Manage topics

1 commit | 1 branch | 0 releases | 0 contributors

Branch: Scenario2b | New pull request | Create new file | Upload files | Find file | Clone or download

MATHIEUDalbin and MATHIEUDalbin Init — Latest commit f08b169 21 hours ago

copybooks-public	Init	21 hours ago
.gitattributes	Init	21 hours ago
.project	Init	21 hours ago

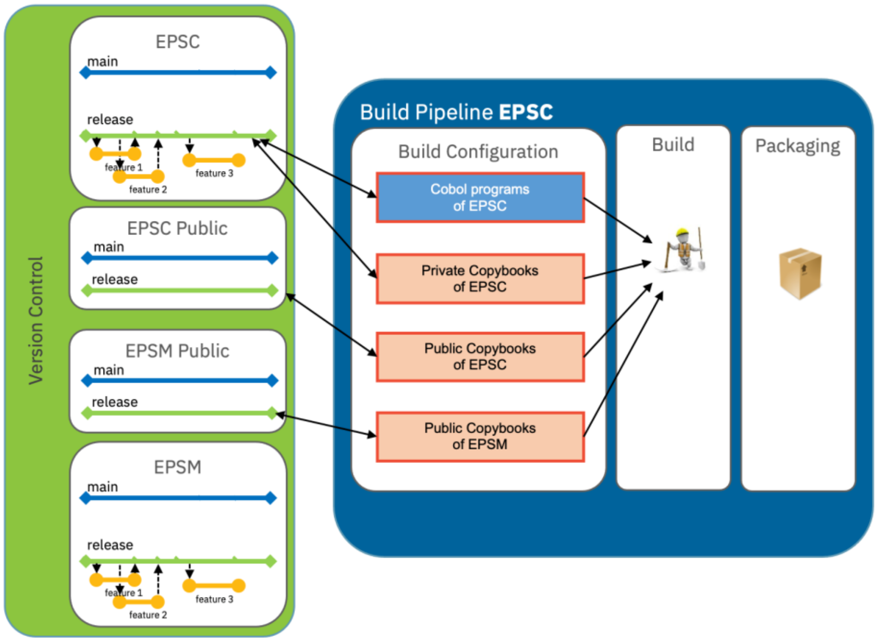
Help people interested in this repository understand your project by adding a README. Add a README

3.2 Build scope configuration

As for the previous scenarios, the necessary source files must be available for the build process to ensure a successful execution:

- The build process for EPSC requires the *public copybooks* of EPSM to be available in the build workspace. The EPSC pipeline configuration requires to pull in the copybooks of the EPSM public copybook repository.
- The build configuration must reference the copybooks of EPSM. This configuration must be reflected in the *application-conf* properties of EPSC when using zAppBuild.

By using a dedicated repository for public copybooks, consuming applications only have access to this subset of the providing application, and not the entire source repository.



3.2.1 Build workspace with Jenkins

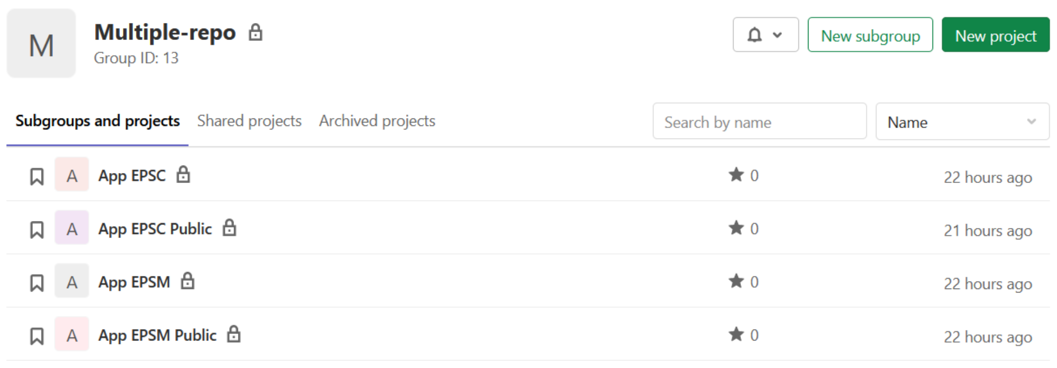
In a Jenkins based build setup, the Jenkins pipeline for EPSC is configured to check out the two repositories of EPSC and the public copybook repository of EPSM, providing the copybooks required for EPSC.

Within a Jenkinsfile, this translates into a checkout for each repository, while only the *App-EPSM-Public* repository is referenced in the build scope:

```
stage('Git Checkout') {
       // EPSC Application repository
       dir (AppEPSC) {
              checkout([$class: 'GitSCM', branches: [[name: GitBranch]],
doGenerateSubmoduleConfigurations: false, submoduleCfg: [], userRemoteConfigs:
[[credentialsId: gitCredId, url: 'git@github.ibm.com:zDevOps-Acceleration/App-EPSC.git']]])
       }
       // EPSC Public Copybook repository
       dir (AppEPSCPublic) {
              checkout([$class: 'GitSCM', branches: [[name: GitBranch]],
doGenerateSubmoduleConfigurations: false, submoduleCfg: [], userRemoteConfigs:
[[credentialsId: gitCredId, url: 'git@github.ibm.com:zDevOps-Acceleration/App-EPSC-
Public.git']]])
       }
       // EPSM Public Copybook repository
       dir (AppEPSMPublic) {
              checkout([$class: 'GitSCM', branches: [[name: GitBranch]],
doGenerateSubmoduleConfigurations: false, submoduleCfg: [], userRemoteConfigs:
[[credentialsId: gitCredId, url: 'git@github.ibm.com:zDevOps-Acceleration/App-EPSM-
Public.git']]])
       }
}
```

3.2.2 Build workspace with GitLab CI/CD

Again, the multiple-project pipelines feature will be used to reproduce this configuration. In this situation, EPSC's and EPSM's public copybooks are grouped into new projects, respectively called *App-EPSC-Public* and *App-EPSM-Public*, and are part of the same GitLab Group:



App EPSC	0	22 hours ago
App EPSC Public	0	21 hours ago
App EPSM	0	22 hours ago
App EPSM Public	0	22 hours ago

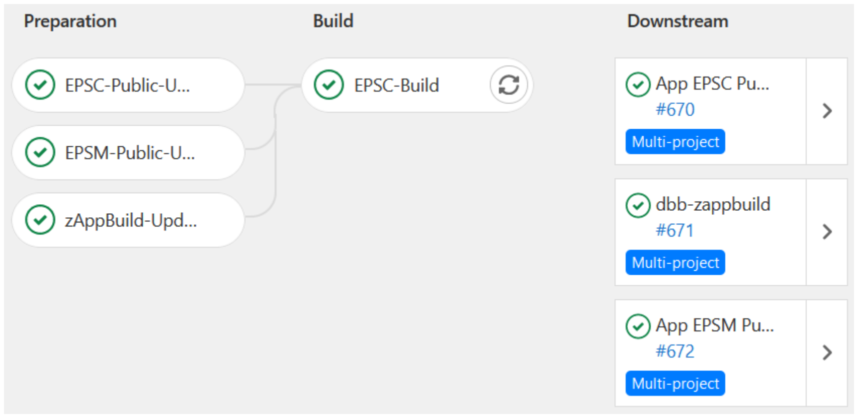
As the *App-EPSC-Public* and *App-EPSM-Public* repositories don't contain artifacts to be built, their respective pipelines only contain a dummy stage, which is executed only when triggered from another pipeline. The pipeline definition for *App-EPSC-Public* is as follows:

```
EPSC-Public-Update:
  stage: Preparation
  only:
    variables:
      - $CI_PIPELINE_SOURCE == 'pipeline'
  script:
    - echo "Updating EPSC-Public"

stages:
  - Preparation
```

The pipeline of EPSC is modified to integrate the checkouts of EPSC-Public and EPSM-Public repositories. Compared with the first scenario of the pipeline of EPSC, the invocation of the EPSM pipeline is replaced by the invocation of the pipelines of *EPSC-Public* and *EPSM-Public*, also specifying which branch to checkout:

```
EPSC-Public-Update:
  stage: Preparation
  except:
    variables:
      - $CI_PIPELINE_SOURCE == 'pipeline'
  trigger:
    project: multiple-repo/App-EPSC-Public
    branch: Scenario2b
    strategy: depend

EPSM-Public-Update:
  stage: Preparation
  except:
    variables:
      - $CI_PIPELINE_SOURCE == 'pipeline'
  trigger:
    project: multiple-repo/App-EPSM-Public
    branch: Scenario2b
    strategy: depend
```

The following diagram shows the structure for the EPSC pipeline:



The same modification is applied to the EPSM's pipeline, to follow this strategy.

3.2.3 zAppBuild configuration

Like in the previous scenarios, the build framework zAppBuild is invoked using the parameter *--application App-EPSC*.

In the *application.properties* file of EPSC's *application-conf* folder, the property *applicationsSrcDirs* must be modified to include the *copybooks-public* subfolder of EPSM for the build configuration.

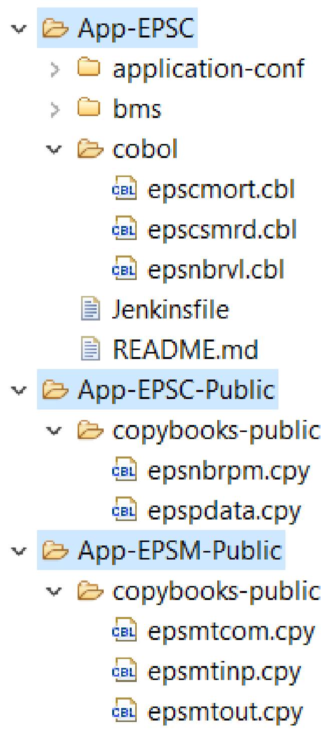
```
#
# Comma separated list all source directories included in application build. Supports both
absolute
# and relative paths.  Relative assumed to be relative to ${workspace}.
# ex: applicationSrcDirs=${application},/u/build/common/copybooks
applicationSrcDirs=${application},App-EPSC-Public/copybooks-public,App-EPSM-Public/copybooks-
public
```

The property *copybookRule* was updated accordingly:

```
# Rule to locate Cobol copy books
copybookRule = {"library": "SYSLIB", \
              "searchPath": [ \
              {"sourceDir": "${workspace}", "directory": "App-EPSC/copybook"}, \
              {"sourceDir": "${workspace}", "directory": "App-EPSC-Public/copybooks-public"} \
              {"sourceDir": "${workspace}", "directory": "App-EPSM-Public/copybooks-public"} \
         ] \
}
```

3.3 Build scenario

Either with Jenkins or GitLab CI/CD, the next build of EPSC will update the build workspace with all 3 referenced repositories and verify, if any file in scope of the build configuration has changed.



```
App-EPSC
  application-conf
  bms
  cobol
    epscmort.cbl
    epscsmrd.cbl
    epsnbrvl.cbl
  Jenkinsfile
  README.md
App-EPSC-Public
  copybooks-public
    epsnbrpm.cpy
    epspdata.cpy
App-EPSM-Public
  copybooks-public
    epsmtcom.cpy
    epsmtinp.cpy
    epsmtout.cpy
```

Based on the changes that can occur either in App-EPSC, App-EPSC-Public or App-EPSM-Public repositories, the impacted files are calculated:

```
..
** Diffing baseline 06812fa992d9dc4e019fea712fea5d9775f70e0d -> current
06812fa992d9dc4e019fea712fea5d9775f70e0d
*** Changed files for directory /var/jenkins/workspace/Scenario2-EPSC/App-EPSC:
*** Deleted files for directory /var/jenkins/workspace/Scenario2-EPSC/App-EPSC:
** Calculating changed files for directory /var/jenkins/workspace/Scenario2b-EPSC/App-EPSC-
Public/copybooks-public
** Diffing baseline 6c31b38b193c92a7995d93e96a85ae748f554e74 -> current
6c31b38b193c92a7995d93e96a85ae748f554e74
*** Changed files for directory /var/jenkins/workspace/Scenario2-EPSC/App-EPSC-
Public/copybooks-public:
*** Deleted files for directory /var/jenkins/workspace/Scenario2-EPSC/App-EPSC-
Public/copybooks-public:
** Calculating changed files for directory /var/jenkins/workspace/Scenario2-EPSC/App-EPSM-
Public/copybooks-public
** Diffing baseline f08b16975e41b7f04526e77fca9581b7a7d48395 -> current
b00e67646867be700670636042d89921c11b4d8b
*** Changed files for directory /var/jenkins/workspace/Scenario2-EPSC/App-EPSM-
Public/copybooks-public:
** Testing if fixed file path exists : copybooks-public/epsmtcom.cpy
*** App-EPSM-Public/copybooks-public/epsmtcom.cpy
..
..
** Found impacted file App-EPSC/cobol/epscmort.cbl
** App-EPSC/cobol/epscmort.cbl is impacted by changed file App-EPSM-Public/copybooks-
public/epsmtcom.cpy. Adding to build list.
** Writing build list file to /var/jenkins/workspace/Scenario2b-
EPSC/outputs/build.20201204.091628.016/buildList.txt
App-EPSC/cobol/epscmort.cbl
..
```

3.4 Additional considerations

This chapter has described how the build scope can include several repositories which contain the copybooks (interface descriptions), while "hiding" the actual implementation of the service-providing applications.

For the build configuration, the choice is still given to either include the work-in-progress version from a release branch, or to include the copybooks currently being in production (as described in section `2.3 Adoption strategies for applications copybooks`).

However, development teams now need to manage and coordinate changes in several git repositories.

4 Scenario 3 - Build configuration includes a single repository of all shared copybooks

This scenario will address some limitations and drawbacks of the previous scenario for the developer like contributing to several git repositories or coordinating pull requests on several repositories. Additionally, managing the build scope might be challenging when having a large number of dependencies on other applications.

Quickly, the concept of having a common shared repository appears: all the applications would then store their copybooks in this common repository, to simplify the management of the build configuration.

The advantage is that applications consuming any service will just need to reference this shared repository to include all the necessary copybooks. It is also possible to perform a specific retrieval only for the required copybooks using Git's SparseCheckout feature (please note, that this feature is not available in GitLab CI/CD pipelines).

The major drawback of this organization is that all application teams would still need to commit and work against at least two repositories: they would need to consistently maintain their application repository and the shared repository where they would store their public copybooks, like all shared interfaces.

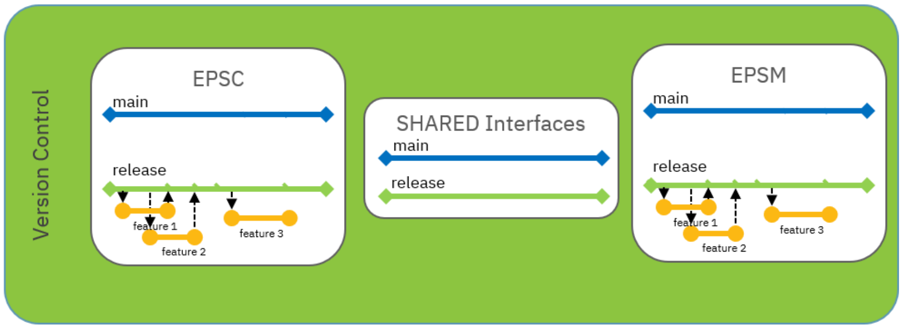
To address the above-mentioned limitations, a new organization can be set up:

- Using a single application repository (as depicted in scenario 1),
- Introducing an automated publishing process for public copybooks to replicate a copybook from the application repository to a repository for shared interfaces.

4.1 Repository definitions

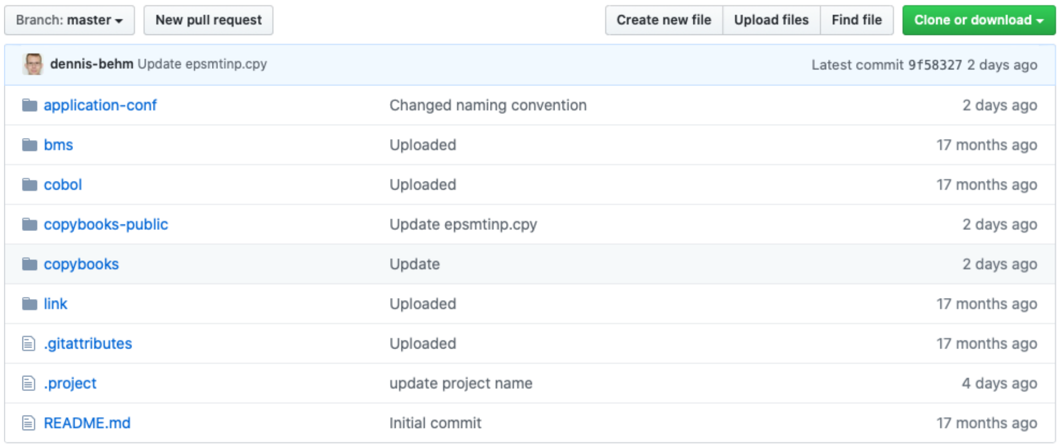
The repository setup is similar to the one described in the advanced scenario 1 (2.2 Advanced scenario). Each application team works in their application repository, which contains all the private and public copybooks.

A new repository for the shared interfaces is established, as a container for the publishing mechanism of the public copybooks.



The build framework requires a new step for the public copybooks, to automatically publish them to the repository of the shared interfaces.

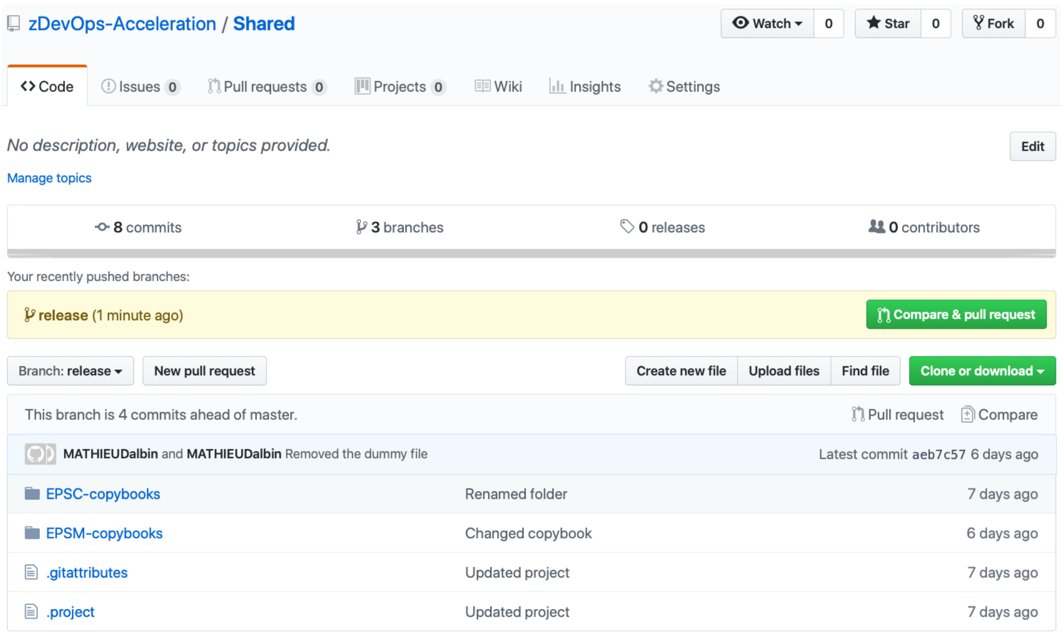
In this setup, the application repository of EPSM contains subfolders for the classified public copybooks (*copybooks-public*) and the private copybooks (*copybooks*). The application team exclusively operates on this repository.


Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

dennis-behm Update epsmtinp.cpy — Latest commit 9f58327 2 days ago

application-conf	Changed naming convention	2 days ago
bms	Uploaded	17 months ago
cobol	Uploaded	17 months ago
copybooks-public	Update epsmtinp.cpy	2 days ago
copybooks	Update	2 days ago
link	Uploaded	17 months ago
.gitattributes	Uploaded	17 months ago
.project	update project name	4 days ago
README.md	Initial commit	17 months ago

The repository for the shared interfaces will contain a copy of all shared interfaces of all applications. As highlighted in the above diagrams, the shared interfaces repository manages two branches - a release and a main branch. The publishing mechanism will add the public copybooks into a subfolder for each application – in the below sample `EPSC-copybooks` and `EPSM-copybooks`.


zDevOps-Acceleration / Shared — Watch 0 | Star 0 | Fork 0

Code | Issues 0 | Pull requests 0 | Projects 0 | Wiki | Insights | Settings

No description, website, or topics provided. Edit

Manage topics

8 commits | 3 branches | 0 releases | 0 contributors

Your recently pushed branches:

release (1 minute ago) — Compare & pull request

Branch: release | New pull request | Create new file | Upload files | Find file | Clone or download

This branch is 4 commits ahead of master. Pull request | Compare

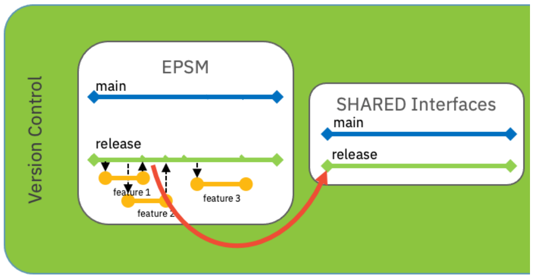
MATHIEUDalbin and MATHIEUDalbin Removed the dummy file — Latest commit aeb7c57 6 days ago

EPSC-copybooks	Renamed folder	7 days ago
EPSM-copybooks	Changed copybook	6 days ago
.gitattributes	Updated project	7 days ago
.project	Updated project	7 days ago

4.2 Publishing mechanism

The build framework implements an automated process for publishing the public copybooks from the application repository to the shared interfaces repository.

When the provider application has modified a public copybook in the release branch (or any other branch used for integration), the build framework will commit and push a copy of the public copybook to the shared interfaces repository.



Options to implement this publishing mechanism are:

- a post processing action in the zAppBuild build framework, that performs the above steps to propagate the public copybook to the shared repository.
- or a dedicated build definition to publish the modified public interfaces to the shared repository.

You can find a sample implementation of the publishing script at https://github.com/IBM/dbb/tree/master/Pipeline

In both scenarios, rules need to be established about the source branch for the publishing mechanism. When the modified copybook is released to production, it is crucial to update the application repository as well as the repository of the shared interfaces.

Other adoption strategies have already been discussed in the whitepaper[3].

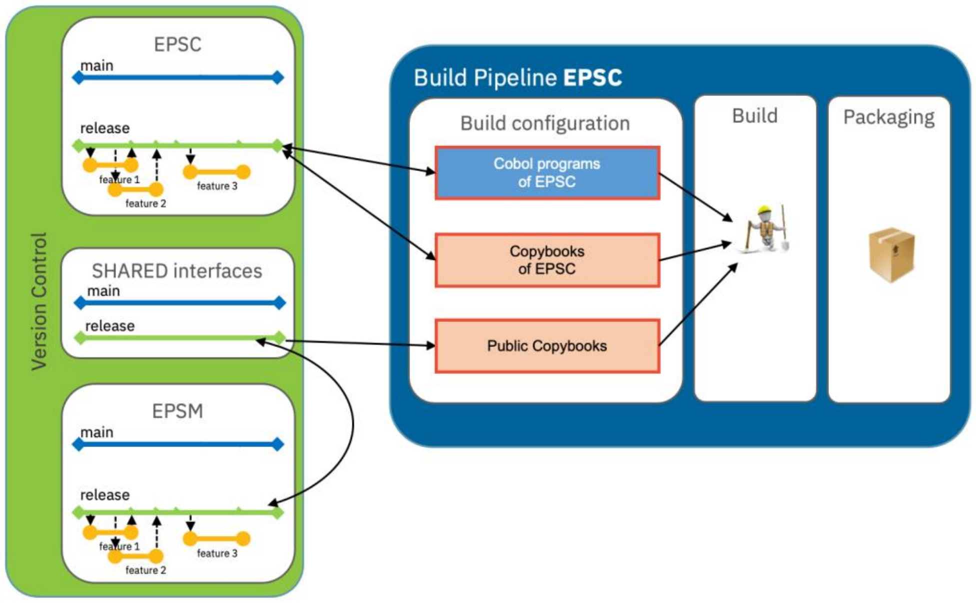
An alternate option is to publish the public copybooks to a binary artifact repository like Nexus or Artifactory.

[3] Whitepaper Develop Mainframe Software with Opensource Source Code Managers and IBM Dependency Based Build (https://www.ibm.com/support/pages/node/6355719)

4.3 Build configuration

4.3.1 Build workspace with Jenkins

In a Jenkins based build setup, the Jenkins pipeline for EPSC is configured to check out the EPSC repository and the shared repository.



Within a Jenkinsfile, this translates into a checkout for the application repository and the repository of the shared interfaces. Again, a SparseCheckout can be applied here for the shared repository, to limit the scope of the checkout process.

```
stage('Git Checkout') {
       // EPSC Application repository
       dir (AppEPSC) {
              checkout([$class: 'GitSCM', branches: [[name: GitBranch]],
doGenerateSubmoduleConfigurations: false, submoduleCfg: [], userRemoteConfigs:
[[credentialsId: gitCredId, url: 'git@github.ibm.com:zDevOps-Acceleration/App-EPSC.git']]])
       }
       // Shared Interfaces repository
       dir (Shared) {
              checkout([$class: 'GitSCM', branches: [[name: GitBranch]],
doGenerateSubmoduleConfigurations: false, submoduleCfg: [], userRemoteConfigs:
[[credentialsId: gitCredId, url: 'git@github.ibm.com:zDevOps-Acceleration/Shared.git']]])
       }
}
```

4.3.2 Build workspace with GitLab CI/CD

In this configuration, the multiple-project feature is also used to invoke pipelines of other projects.

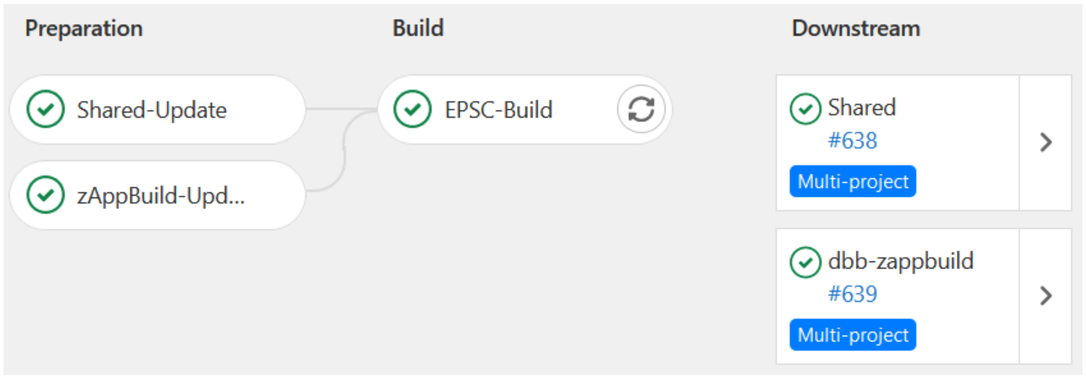
The *Shared* project in GitLab contains a pipeline definition, which is very similar to the pipeline of *App-EPSC-Public* or *App-EPSM-Public*. It only defines a dummy stage that is executed when the pipeline is triggered from an upstream pipeline:

```
Shared-Update:
  stage: Preparation
  only:
    variables:
      - $CI_PIPELINE_SOURCE == 'pipeline'
  script:
    - echo "Updating Shared"

stages:
  - Preparation
```

The pipeline of *EPSC* is configured accordingly to reference this new project and invoke the pipeline of *Shared* project:

```
Shared-Update:
  stage: Preparation
  except:
    variables:
      - $CI_PIPELINE_SOURCE == 'pipeline'
  trigger:
    project: multiple-repo/Shared
    branch: Scenario2a
    strategy: depend
```

The result of this configuration is shown in the execution diagram:



4.3.3 zAppBuild configuration

Like in the previous scenarios, the build framework zAppBuild is invoked using the parameter *--application App-EPSC*.

In the *application.properties* file of EPSC's *application-conf* folder, the property *applicationsSrcDirs* must be modified to include the *copybooks-public* of the shared interfaces repository for the build configuration.

```
#
# Comma separated list all source directories included in application build. Supports both
absolute
# and relative paths.  Relative assumed to be relative to ${workspace}.
# ex: applicationSrcDirs=${application},/u/build/common/copybooks
applicationSrcDirs=${application},Shared
```

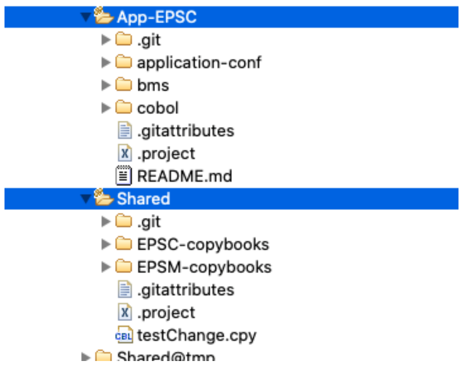
In this scenario, it is sufficient to just point to the Shared folder, as this repository contains the published copybooks.

The property *copybookRule* was updated accordingly:

```
# Rule to locate Cobol copy books
copybookRule = {"library": "SYSLIB", \
              "searchPath": [ \
              {"sourceDir": "${workspace}", "directory": "App-EPSC/copybook"}, \
              {"sourceDir": "${workspace}", "directory": "Shared/EPSC-copybooks"} \
              {"sourceDir": "${workspace}", "directory": "Shared/EPSM-copybooks "} \
         ] \
}
```

4.4 Build scenario

The next build of EPSC will update the build workspace with the two referenced repositories and verify, if any file in scope of the build configuration has changed.



Based on the changes that can occur either in App-EPSC or the shared repository, the impacted files are calculated:

```
..
** Calculating changed files for directory /var/jenkins/workspace/Scenario2a-EPSC/App-EPSC
** Diffing baseline 2ce55a5dd734d35e9e4d7ac662db4ab2d9c7c1ed -> current
2ce55a5dd734d35e9e4d7ac662db4ab2d9c7c1ed
*** Changed files for directory /var/jenkins/workspace/Scenario2a-EPSC/App-EPSC:
*** Deleted files for directory /var/jenkins/workspace/Scenario2a-EPSC/App-EPSC:
** Calculating changed files for directory /var/jenkins/workspace/Scenario2a-EPSC/Shared
** Diffing baseline 0e0556dd74f4a15bd1f75aa620f5247e8af644cc -> current
e7acd19d49c2f4e1cd355992a24159931bc0c7b2
*** Changed files for directory /var/jenkins/workspace/Scenario2a-EPSC/Shared:
** Testing if fixed file path exists : EPSM-copybooks/epsmtinp.cpy
*** Shared/EPSM-copybooks/epsmtinp.cpy
..
..
** App-EPSC/cobol/epscmort.cbl is impacted by changed file Shared/EPSM-copybooks/epsmtinp.cpy.
Adding to build list.
** Writing build list file to /var/jenkins/workspace/Scenario2a-
EPSC/outputs/build.20201203.115757.057/buildList.txt
App-EPSC/cobol/epscmort.cbl
..
```

4.5 Considerations / remarks

This scenario addresses the limitations of the previous scenarios:

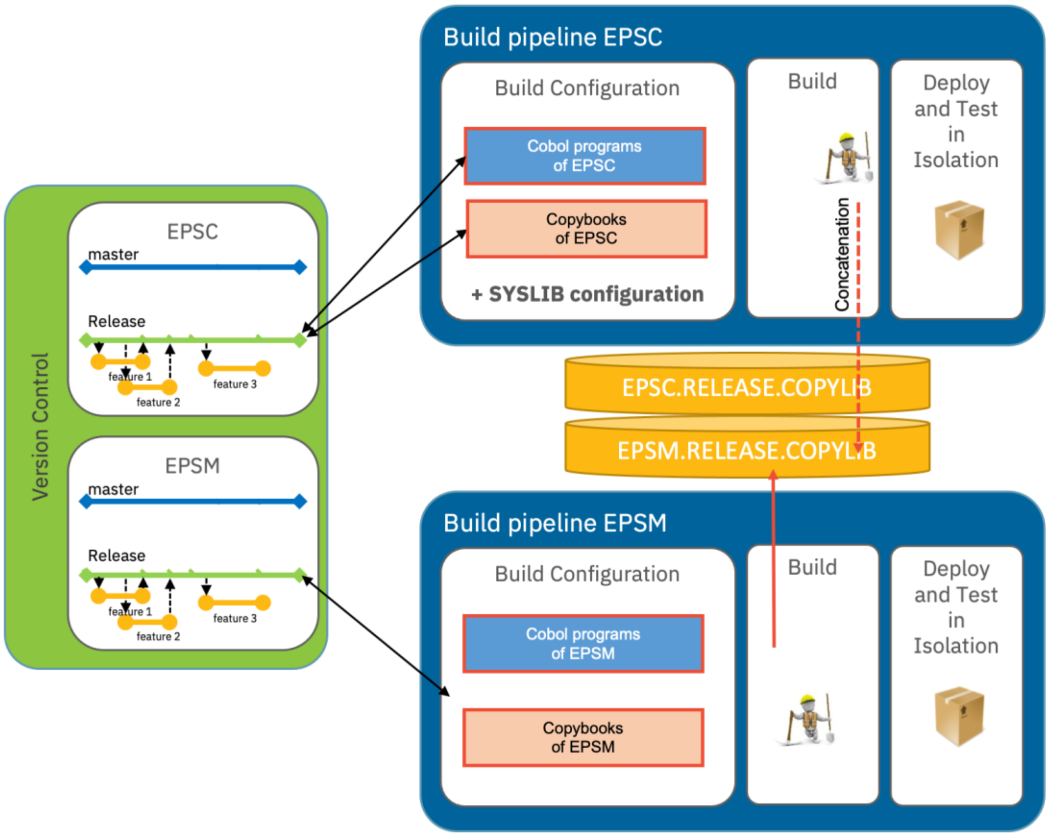
- Compared to the first scenario, this implementation limits what the consuming applications can see from the providing application,
- Compared to the second scenario, it helps to simplify the build configuration as only one repository needs to be referenced to access all the published interfaces of all applications. Additionally, development teams only operate against a single git repository for their application artifacts.

5 Scenario 4 - Build configuration to include copybooks through SYSLIB concatenation

An alternate way to include dependent copybooks is simply by referencing an existing MVS library, which contain the copybooks. This library would be added to the SYSLIB concatenation, which usually lists all the libraries where the compiler needs to look for copybooks.

One drawback of this technique is that the zAppBuild framework does not recognize changes happening in these libraries. Consequently, impact builds with zAppBuild will not be able to calculate the list of impacted files and programs that are using the copybooks in these libraries will not automatically be rebuilt. This might be the simplest way to configure the build framework, but it requires more manual coordination between teams to synchronize changes.

These copybook libraries can be the libraries used in the individual application builds or a shared library, where the application builds *publish* their copybooks, like the publishing mechanism described in the previous section.



5.1 Build scope configuration

5.1.1 Build workspace

To implement this strategy, the build workspace simply includes the elements stored in the application repository. The following definition can be used in a Jenkins pipeline:

```
stage('Git Checkout') {
       // EPSC Application repository
       dir (AppEPSC) {
              checkout([$class: 'GitSCM', branches: [[name: GitBranch]],
doGenerateSubmoduleConfigurations: false, submoduleCfg: [], userRemoteConfigs:
[[credentialsId: gitCredId, url: 'git@github.ibm.com:zDevOps-Acceleration/App-EPSC.git']]])
       }
```

For a GitLab CI/CD pipeline, no specific configuration is required, as no downstream pipeline needs to be executed.

5.1.2 zAppBuild configuration

Like in the previous scenarios, the build framework zAppBuild is invoked using the parameter *--application App-EPSC*. In the *application.properties* file of EPSC's *application-conf*, the property *applicationsSrcDirs* just includes the application directories of EPSC itself.

```
#
# Comma separated list all source directories included in application build. Supports both
absolute
# and relative paths.  Relative assumed to be relative to ${workspace}.
# ex: applicationSrcDirs=${application},/u/build/common/copybooks
applicationSrcDirs=${application}
```

The property *copybookRule* points to the copybooks owned by the EPSC application

```
# Rule to locate Cobol copybooks
copybookRule = {"library": "SYSLIB", \
              "searchPath": [ \
              {"sourceDir": "${workspace}", "directory": "App-EPSC/copybook"}, \
         ] \
}
```

The latest version of zAppBuild allows specifying additional libraries (provided as a comma-separated list), which will be added to the SYSLIB for the compiler and linker. In this scenario, the copybooks provided by EPSM and required for a successful of EPSC are stored in the *EPSM.RELEASE.COPYLIB* dataset on MVS:

```
# additional libraries for compile SYSLIB concatenation, comma-separated
cobol_compileSyslibConcatenation=EPSM.RELEASE.COPYLIB
```

5.2 Adoption strategies

In this configuration, assuming the EPSM application maintains several versions of its copybooks, the EPSC application would still be able to select which version to adopt. For instance, including the MAIN version (stored in the *EPSM.MAIN.COPYLIB* dataset) would allow EPSC to use the production version of the copybooks, while picking the RELEASE version of copybooks (*EPSM.RELEASE.COPYLIB*) would permit to immediately integrate the last versions of EPSM's copybooks.

6 Conclusion

The above scenarios are not an exhaustive list of ways to configure the build scope to resolve dependencies across multiple repositories. It can vary depending on the selected git provider or CI orchestrator.

Additionally, the focus was set on the management of dependencies at the source level, like copybooks or include files, and the integration of binary dependencies, like submodules, was not discussed.

To meet the different development workflows, certainly the scenarios can be mixed: for example, during the migration period, one of the first 3 scenarios can be combine with scenario 4 to include dependent copybooks, that are still managed though a legacy solution.