An IBM Document from

IBM Z DevOps Acceleration Program

October 2022 - Release 2.0

Migrating applications to a DevOps pipeline

Migration recipes with DBB and zAppBuild

Gil Parent
gparent@fr.ibm.com

Dennis Behm
dennis.behm@de.ibm.com

Mathieu Dalbin
mathieu.dalbin@fr.ibm.com

Timothy Donnelly
donnellt@us.ibm.com

Abstract

Tips & Tricks on migrating applications to the pipeline focusing to initialize the dependency metadata

Table of Contents

1 Introduction

Managing and Building existing Mainframe applications written in COBOL, Assembler or PLI using a new SCM and build framework is not always as simple as what we would expect. Many Mainframe applications have been developed many years ago with a set of rules and also exceptions in place.

Today, new methodologies like automated impact builds, code review or automatic testing are being applied by many clients into their Mainframe development workflow. With IBM Dependency Based Build, a sample build framework implementation named zAppBuild is available at https://github.com/IBM/dbb-zappbuild and proposes a solid baseline to apply these new build practices.

However, when migrating to a new toolchain with a new version control system which includes adopting a new build framework, some specific migration considerations need to be taken into consideration to be able to benefit from the build automation.

One important aspect is the understanding of dependencies between source artifacts, which is key to enable impact builds scenarios. What do we mean by that? Understanding dependencies is a prerequisite that the build framework can automatically identify impacted artifacts which need to be built for a set of changed artifacts.

During migration to the new automated build framework, these dependencies between the different parts of the application source files need to be captured before using the new toolchain. The purpose of this document is to outline a migration checklist including discussing details about the initialization of the dependencies metadata and setting the baseline of the build result history leveraging the zAppBuild framework.

2 Migration Overview and Checklist

Migrating a mainframe application to Git as the version control system and IBM Dependency Based Build as the build framework, requires multiple basic considerations before migrating the application code.

The below checklist provides an overview of the different activities to consider when migrating an application to the new version control system with focusing on initializing the dependency metadata and setting the baseline in the build result history.

	Task	References
1	Defining the *scope of the application* including understanding the boundaries between applications. Usually this is done based on existing naming conventions including ownership / responsibilities of the existing application systems. Determine all source codes belonging to the application and define which versions are migrated to the new version control system, including to define a strategy how to deal with the existing history in the legacy system.	
2	While Git as the new version control system stores source in UTF-8 and the mainframe uses an EBCDIC code page, assess if the application source code contains non-roundtripable and non-printable characters. *Managing the code page conversion when migrating z/OS source files to git*, is an important prerequisite before migrating. Decide on the strategy to deal with the exceptions. It is recommended to clean up these in the old version control system before migrating. In some instances, cleaning up is not possible, and you might need to migrate the file in binary.	Please see the publication *Managing the code page conversion when migrating z/OS source files to Git* [1]
3	*Perform the migration* of the source code to the git repositories to the desired folder layout including all build settings (like mapping to the new build scripts, see application-conf[2]). We recommend migrating into a *Migration* branch, which allows you to validate the migrated source code with the application team for a sign-off before populating the other long-living branches according to your branching strategy.	Please see the publications on different branching strategies.[3] [4]

[1] Managing the code page conversion when migrating z/OS source files to Git - https://www.ibm.com/support/pages/node/6591177

[2] Overview of application settings in zAppBuild https://github.com/IBM/dbb-zappbuild/tree/main/samples/application-conf

[3] Implementing a Release-based Development Process for Mainframe Applications - https://www.ibm.com/support/pages/node/6619083

[4] Design and implement your own release-based development workflow - https://mediacenter.ibm.com/media/Design+and+implement+your+own+release-based+development+workflow/1_f3qpi2ld

4	Perform a scan of the Migration branch to validate that dependencies are correctly identified and stored in the DBB dependency store. Optionally, validate the migration and build framework by performing a full build of the entire application. At this stage in the migration process, the build outputs are seen as temporary. Please consider, that performing a full build of each application that you are migrating also takes significant time to validate. Also, you might be facing application artifacts which were not changed for decades and no longer compile.	See chapter ***Error! Reference source not found.*** and *Initial Build strategy after migration*
5	After the validation, populate the long-living Production/Main branch according to the branching strategy, through a Pull/Merge request depending on the selected Git provider. Run the initial build of the Production/Main branch to populate the dependency metadata for the Production/Main configuration including the creating a first build result in the build result history. Your application architecture will play a significant role on potential additional activities.	See Checklist #4 See Chapter *Initial Build strategy after migration*
6	Populate the remaining long-living branches with the code base from Production/Main, such as the *Development* branch.	
7	Run the pipeline build for any additional long-living branches, such as the *Development* branch without any changes to clone the dependency metadata information to set the baseline for future impact builds including the creating a first build result in the build result history.	See chapter *Managing pipeline builds for feature branches*
8	Consider any potential in-flight changes when the migration of an application is scheduled. Ideally, there are no or only very few ongoing changes, which need to be migrated to the new pipeline. A convenient approach is letting the developers to redo the changes in the new pipeline after completing the above migration steps.	

Please assess above checklist to your needs.

3 Populating DBB dependency metadata store

A very powerful feature of a dependency-based build is the capability to automatically identify files which have been modified or files impacted by a change. To be able to use this feature, DBB needs to know the dependencies between files. Dependencies are stored in the DBB metadata store. There are two types of dependencies:

- Dependencies which can be extracted by scanning the source code
- Dependencies which are defined in the object composition of a load module, when several objects are linked together.

Dependencies available in the source code itself are updated when the build scans the sources files, to find dependencies like a Cobol program file including a copybook file. This is performed through the default DependencyScanner [5] [6] shipped with the DBB toolkit.

Object-level dependencies appear when programs call subroutines statically or if the linker is instructed to bind several objects together. This information will be discovered by scanning the load module after the compile and link edit steps and are also stored in the DBB metadata store.

After the initial migration from your legacy version control system to git, the DBB metadata store needs to be populated with this dependency information, in order to perform the impact analysis like implemented in the sample zAppBuild framework[7].

This can be accomplished by performing a full build of the application, which will also produce a new set of load modules. This can be time-consuming and potentially some programs cannot be built due to technical constraints (languages level, missing dependencies, ...).

To address these challenges, zAppBuild provides a capability to just scan for dependencies without actually building the application.

3.1 Populating the collections for source-level and object-level dependencies

zAppBuild stores source-level dependencies in a collection following the naming conventions *<applicationName>-<branchName>*, while object-level dependencies are stored in the collection *<applicationName>-<branchName>-outputs*.

zAppBuild contains various scan options to populate the different collections for an application:

`-s,--scanOnly`	`Flag indicating to only scan source files for application without building anything (deprecated use --scanSource). Populates the collection for source-level dependencies.`
`-ss,--scanSource`	`Flag indicating to only scan source files for application without building anything. Populates the collection for source-level dependencies.`

[5] DBB DependencyScanner API - https://www.ibm.com/docs/api/v1/content/SS6T76_1.1.0/javadoc/index.html?com/ibm/dbb/dependency/DependencyScanner.html

[6] IBM Docs DBB Manage Build dependencies - https://www.ibm.com/docs/en/dbb/1.1.0?topic=scripts-how-manage-build-dependencies

[7] zAppBuild build framework implementation - https://github.com/IBM/dbb-zappbuild

-sl,--scanLoad	Flag indicating to only scan load modules for application without building anything. Populates the collection for object-level dependencies.
-sa,--scanAll	Flag indicating to scan both source files and load modules for application without building anything. Populates both collections for source-level and object-level dependencies.

With the options *--scanOnly* and *--scanSource* the build framework will only inspect the source code for dependencies, so it also only requires access to the source code.

The options *--scanLoad* and *--scanAll* require access to an existing set of load modules on z/OS and leverage the build option *--hlq* to locate the load libraries.

The option *--scanAll* can be used to scan the source and an existing set of load modules. This is the option on which we focus in our scenario:

In this scenario, the MortgageApplication sample, which includes a static linkage scenario with a link card[8], is used to demonstrate these capabilities: the source files were just migrated to a git repository so the DBB collections for this application don't yet exist at this time.
On z/OS, preexisting load modules of the MortgageApplication were made available under the high-level qualifier *DDS0690.PREVIOUS.MORTGAGE*: they correspond to the source files that were migrated to git from the legacy SCM.

For the scanning process, the build process requires read access to the dataset libraries; so, you can either point to your existing load library or a copy of it.

The inventory process is kicked off by invoking the zAppBuild framework with the *--scanAll* option that will scan the source code and the preexisting load modules:

```
$DBB_HOME/bin/groovyz /u/dds0690/userBuildRepo/zAppBuild/build.groovy --sourceDir
/u/dds0690/gitlab/dbb-zappbuild/samples --workDir /u/dds0690/gitlab/work --hlq
DDS0690.PREVIOUS.MORTGAGE --verbose --application MortgageApplication --fullBuild --scanAll
```

Please note, that the above command, using the options --fullBuild –scanAll, will only perform the scan operation and not rebuild the source.

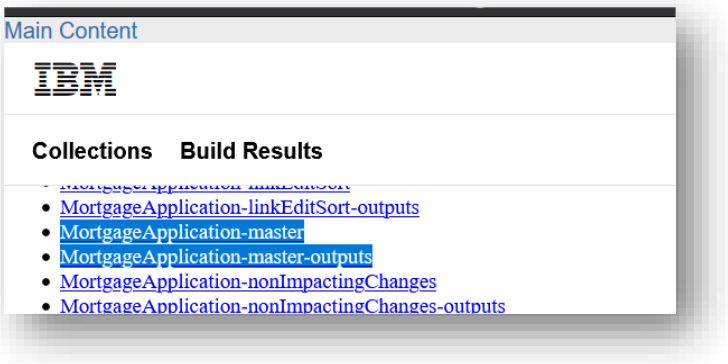
The zAppBuild console output will document that it scanned the source code and the load modules:

```
** Build start at 20210225.021541.015
** Repository client created for https://10.3.20.96:10443/dbb
** Build output located at /u/dds0690/gitlab/work/Mortgage/build.20210225.021541.015
** Build result created for BuildGroup:MortgageApplication-scanLoadModule
BuildLabel:build.20210225.021541.015 at https://10.3.20.96:10443/dbb/rest/buildResult/40154
** --fullBuild option selected. Scanning all programs for application MortgageApplication
** Writing build list file to
/u/dds0690/gitlab/work/Mortgage/build.20210225.021541.015/buildList.txt
** Scanning source code.
** Scanning load modules for static dependencies.
** Writing build report data to
/u/dds0690/gitlab/work/Mortgage/build.20210225.021541.015/BuildReport.json
** Writing build report to
/u/dds0690/gitlab/work/Mortgage/build.20210225.021541.015/BuildReport.html
** Build ended at Thu Feb 25 14:15:51 GMT+01:00 2021
** Build State : CLEAN
```

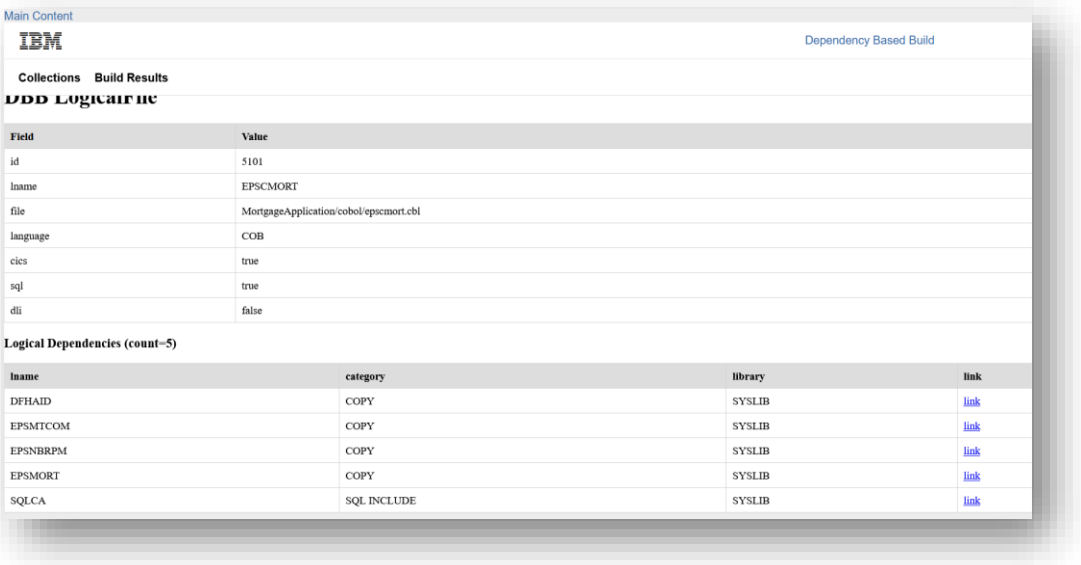
[8] zAppBuild link card sample:
https://github.com/IBM/dbb-zappbuild/blob/main/samples/MortgageApplication/link/epsmlist.lnk

```
** Total files processed : 15
** Total build time  : 9.975 seconds
```

In the DBB metadata store, two new collections are created for the MortgageApplication application:



The first collection MortgageApplication-master contains metadata information about source-level dependencies. When inspecting a file, detailed information about the file and its logical dependencies are displayed:

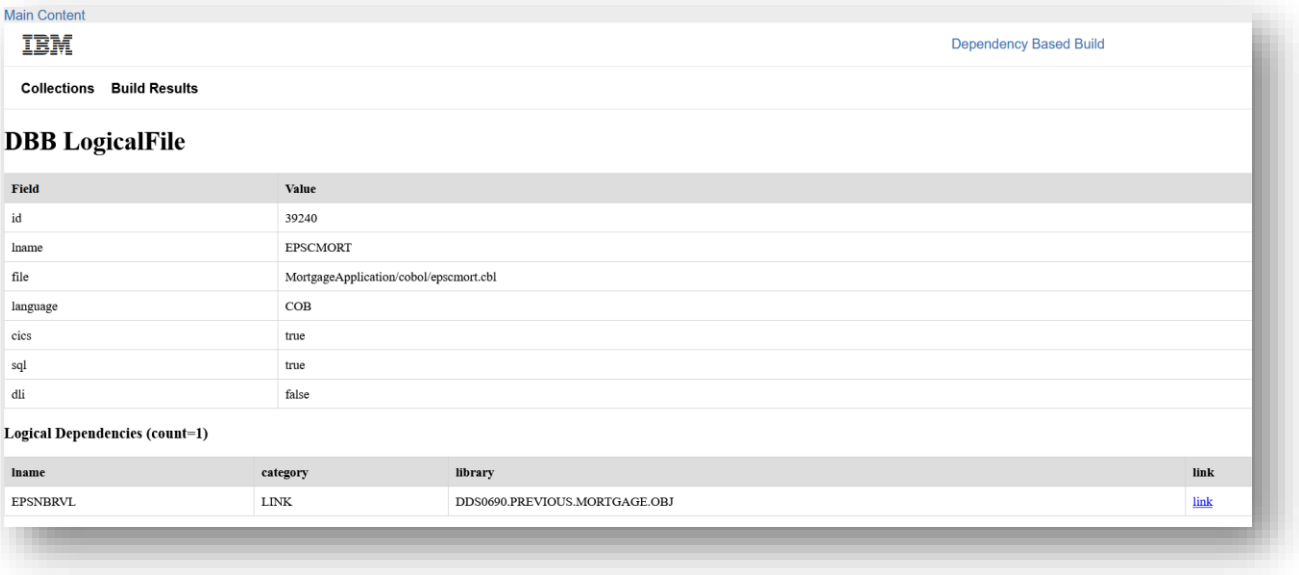


Field	Value
id	5101
lname	EPSCMORT
file	MortgageApplication/cobol/epscmort.cbl
language	COB
cics	true
sql	true
dli	false

Logical Dependencies (count=5)

lname	category	library	link
DFHAID	COPY	SYSLIB	link
EPSMTCOM	COPY	SYSLIB	link
EPSNBRPM	COPY	SYSLIB	link
EPSMORT	COPY	SYSLIB	link
SQLCA	SQL INCLUDE	SYSLIB	link

The second collection MortgageApplication-master-outputs contains object-level dependencies information. Selecting the EPSCMORT file, the identified static link dependencies are displayed:



Please validate the console output of the build for any HTTP issues in case dependencies could not be stored correctly.

Please also see further documentation on the different scanning options in the zAppBuild repository under the different invocation samples.[9]

3.2 Conclusion

With these options, both dependency types are managed in a scan-only scenario and will enable the use of Impact Build process without performing a full build.

If a copybook or a subroutine is changed, the impact build scenario will be able to find all the impacted files.

Please make sure that the build points to the right set of the libraries!

Caution - With this approach, please consider that impact builds may fail even with correctly populated DBB dependencies:

Object-Level dependencies (object decks) for build files may not be available in the configured build libraries for the linkedit step.

Either copying the existing load modules and object decks from the old system to the new build libraries is required or use a concatenation approach in the link steps should be performed. See also section *Managing pipeline builds for feature branches*

[9] zAppBuild invocation samples - https://github.com/IBM/dbb-zappbuild/blob/main/BUILD.md#invocation-samples-including-console-log

4 Initial Build strategy after migration

4.1 Existing strategies

While Chapter 3 has outlined the technical options and considerations about populating the dependency metadata for DBB without actually invoking the compile and link steps, this chapter will assess different strategies for the initial build in context of your application architecture.

We are distinguishing performing the initial build by:

- Building all programs, or
- Simply scanning all programs (and existing load modules).

Both strategies will initialize the dependency metadata in DBB and create an initial build result to the baseline to enable subsequent impact builds.

4.1.1 Building all programs

This means that you will create new load modules for all the buildable files of an application. The build is invoking the following command leveraging the --fullBuild option[10]:

```
$DBB_HOME/bin/groovyz /u/WhereYouHaveStoredYourScript/build.groovy --sourceDir
/u/YourSandBoxPath --workDir /u/YourLogLocationPath --hlq TARGETED.PDS.APPLI --application
TheApplicationName --fullBuild
```

Result :

- All the source files are built (compiled and linked).
- All the dependencies, both for source-level and object-level relationships, are recorded in the DBB metadata store.

- The build result history will have a baseline for subsequent impact builds.

- The build library for this configuration will have new binaries.

Remarks :

- A Full build ensures that the build framework works for all files of the application. However, this verification takes extra time and also resources.

- To complete this inventory process, it requires that the build completes successfully.

- New version of the loads will be created in the PDSEs. What will you do with these new loads:
    - Delete them, because you are sure that they are the same as those already running in Production. This is applicable for applications which use dynamic calls. With static calls, you might want to preserve the object decks.
    - Deploy them because you want to be sure that the source files in Git and the load files in the different runtime environments are in sync. This means additional testing will be required.

[10] zAppBuild – Execution log - Perform a Full build
https://github.com/IBM/dbb-zappbuild/blob/main/BUILD.md#perform-full-build-to-build-all-files

4.1.2 Scanning programs

This means that the build will not produce new load modules for all the files of an application, but focus on publishing dependency information in the metadata store and set a baseline in the build history of DBB.

4.1.2.1 Collect source-level dependencies

To only scan the code base, run a full build *--fullBuild*, with the *--scanSource* option[11], by submitting the following command:

```
$DBB_HOME/bin/groovyz /u/WhereYouHaveStoredYourScript/build.groovy --sourceDir
/u/YourSandBoxPath --workDir /u/YourLogLocationPath --hlq TARGETED.PDS.APPLI --application
TheApplicationName --fullBuild --scanSource
```

Result :

- Only the source files are scanned, object-level dependencies are not captured.
- All the dependencies between sources files such as a copybook used in a program, are recorded in the DBB metadata store.
- The build result history will have a baseline for subsequent impact builds.

Remarks :

- Execution time will be much faster when compared to the previous case with performing a full build since it will not update the PDSEs with a new version of the load files.
- If your application is using static linkage, or your application stored link cards in the repository, dependencies are incomplete.
- This means no extra steps of testing will be required.

4.1.2.2 Collect source-level and object-level dependencies

This means that the build will **not** produce new load modules for all the files of an application, but scan source-level and object-level dependencies using zAppBuilds *--scanAll* option[12].

The build can be run by submitting the following command:

```
$DBB_HOME/bin/groovyz /u/WhereYouHaveStoredYourScript/build.groovy --sourceDir
/u/YourSandBoxPath --workDir /u/YourLogLocationPath --hlq TARGETED.PDS.APPLI --application
TheApplicationName --fullBuild --scanAll
```

Please note, that you need to pass the HLQ to make the load modules accessible to the build. This can be a copy of your production load library.

Result :

[11] zAppBuild – Execution log - Perform a Scan Source build
https://github.com/IBM/dbb-zappbuild/blob/main/BUILD.md#perform-a-scan-source-build

[12] zAppBuild – Execution log - Perform a Scan source and outputs build
https://github.com/IBM/dbb-zappbuild/blob/main/BUILD.md#perform-a-scan-source--outputs-build

- All the source files and the load files are scanned to populate the source-level and object-level dependencies.

- All the dependencies are recorded in the DBB metadata store.

- The build result history will have a baseline for subsequent impact builds.

Remarks :

- Execution time will be much faster when compared to the previous case with full build option since it will not update the PDSEs with a new version of the load files.

- This means no extra steps of testing will be required.

- Identify a strategy to deal with object decks which need to be available for subsequent impact builds on the long-living branches as well as on feature branches.

4.2 Selecting a strategy for the initial builds

After you have migrated your application source files, you must finish the migration by running the initial zAppBuild build to populate the dependency metadata as well as setting a baseline build result for the subsequent incremental builds.

Depending on your individual requirements, you will have to decide which of the above options is the best for you:

- The outlined **build** strategy obviously implies to perform a fullBuild producing a new set of load modules. This is a viable solution to ensure you have migrated all the sources files and to verify that the entire migrated source codes can be built with the new build framework.
  Given the nature of the full build in zAppBuild, the metadata recorded in the DBB metadata store will be automatically populated for both source level and object level dependencies. The draw-back of this strategy is, that it might take more time and poses the question on how to proceed with the new version of load files.

- The strategy to only perform a **scan** for the dependencies is a faster approach to complete the initial zAppBuild build. Which scanning option to choose is dependent on the application characteristics – i.e. on the existence (or not) of static calls. In case of static linkage, it additionally requires providing the corresponding set of load modules to the version of the source code that you have migrated.
  Even with picking this strategy the metadata is created and you will be able to benefit from the impact build feature of zAppBuild and DBB. Scanning will be much faster as no sources are really built. Keep in mind that this approach does not allow a 100% validation of the new build framework. You want to run the full build for a selection of your applications.

If you are migrating by an “application by application” approach, you can select one of the above strategies for each application.

If you plan to incrementally migrate subsets of an application system to the git repository because the application is very big and does not allow for a single migration, you need to define a strategy to update the dependency metadata for each subset. While the typical impactBuild, will treat the migrated files as new files and add them to the build list, which you would like to avoid, you can leverage the *scanning* strategy. After migration of the subset, make sure, that there are no commits with delivered to the branch, then invoke the build with the appropriate configuration for scanning.

5 Managing pipeline builds for feature branches

Feature branches allow you to implement your new feature in isolation before sharing them with your team.

Pipeline builds on a feature branch is a common practice to prove build consistency across the application.

Like the other builds, which were discussed in this document so far, the first build of a features branch (also) requires the correct dependency metadata and a baseline reference to calculate the changed files.

This section will explain how this is implemented in zAppBuild and also discuss a strategy to mitigate failing builds when your application uses static linkage.

5.1 General approach for cloning collections

zAppBuild has capabilities to identify if a build is on a topic branch. It starts with setting the *mainBuildBranch* property in *application.properties*, providing a reference for determining if it is a feature branch:

```
#
# The main build branch.  Used for cloning collections for topic branch builds instead
# of rescanning the entire application.
mainBuildBranch=master
```

When performing an impact build on a newly created feature branch, zAppBuild will detect the new branch in the *verifyCollections()* method and clone the existing collections of the reference branch of the previous setting.

Additionally, zAppBuild will use the last successful build results of the *mainBuildBranch* to determine the baseline git hash for the calculation of the changed files.

Please make sure that each pipeline build configuration references a unique high-level qualifier for the build libraries. This can be accomplished by calculating a unique qualifier in the pipeline definition file (it can be a *Jenkinsfile* or a *.gitlab-ci.yml* file), for example based on the branch name.

The following processing is possible in GitLab CI/CD to obtain a unique qualifier in the build stage:

```
Build:
    stage: Build
    except :
        variables:
        - $CI_PIPELINE_SOURCE == 'pipeline'
    script:
        - if [ -n "$CI_COMMIT_BRANCH" ]; then export LLQ=`echo ${CI_COMMIT_BRANCH} | sed 's/-
//g' | sed 's/^[0-9]*//g' | cut -c 1-8 | tr '[:lower:]' '[:upper:]'`; fi
        - if [ -n "$CI_MERGE_REQUEST_SOURCE_BRANCH_NAME" ]; then export LLQ=`echo
${CI_MERGE_REQUEST_SOURCE_BRANCH_NAME} | sed 's/-//g' | sed 's/^[0-9]*//g' | cut -c 1-8 | tr
'[:lower:]' '[:upper:]'`; fi
        - if [ -z "$LLQ" ]; then export LLQ=DEFAULT; fi
        - if [ "$CI_COMMIT_BRANCH" = "MASTER" ]; then $DBB_HOME/bin/groovyz -
Djava.library.path=$DBB_HOME/lib:/usr/lib/java_runtime64 build.groovy --workspace
$CI_PROJECT_DIR/samples --hlq GITLAB.ZAPP.CLEAN.$LLQ --workDir $CI_PROJECT_DIR/BUILD-
$CI_PIPELINE_ID --application MortgageApplication --logEncoding UTF-8 --impactBuild --verbose;
fi
        - if [ "$CI_COMMIT_BRANCH" != "MASTER" ]; then $DBB_HOME/bin/groovyz -
Djava.library.path=$DBB_HOME/lib:/usr/lib/java_runtime64 build.groovy --workspace
```

```
$CI_PROJECT_DIR/samples --hlq GITLAB.ZAPP.CLEAN.$LLQ --workDir $CI_PROJECT_DIR/BUILD-
$CI_PIPELINE_ID --application MortgageApplication --logEncoding UTF-8 --impactBuild --verbose
--propFiles MortgageApplication/application-conf/featureBranchConcatenation.properties; fi
```

The *mainBuildBranch* configuration defining the baseline to populate the new collection information, will need to be **dynamically** set depending where in your development workflow you are – basically it has to be set to the branch from where the topic branch was created from.

Let's look at a sample explaining the situation: We are assuming a release-based workflow[13]. For a feature branch, which is forked from the Development branch, the *mainBuildBranch* setting needs to be *Development*; while for an emergency/hotfix branch, which is forked from the Production/Main branch, the *mainBuildBranch* setting needs to be *Prodution/Main.* This dynamic evaluation needs to be embedded in your pipeline script and is passed to zAppBuild via the *--propFiles* or *–propOverwrites* cli argument.

5.2 Avoiding build issues on initial builds for feature branches

When the development in the feature branch starts and the first zAppBuild "impactBuild" build for a feature branch is requested, a new set of z/OS libraries is created automatically. These libraries are empty, and the build will populate them with the identified changed and impacted files.

zAppBuild instructs the compiler and binder to search in the set of libraries which are allocated for your specific branch.

However, there are some cautionary notes that must be realized when applications have dependencies which are not seen as a direct dependency. This will be the case for static linkage, where the linker tries to include objects of a subprogram or when a program references copybooks that are generated from BMS Map definitions that are not part of the git repository.

In these situations, the build process potentially fails as the dependencies are not found in included libraries.

zAppBuild possess a set of properties for each language script to add existing libraries to the SYSLIB concatenation for compilation or link-edition.

This helps to point to existing libraries that contain indirect dependencies and to resolve the issue previously described.

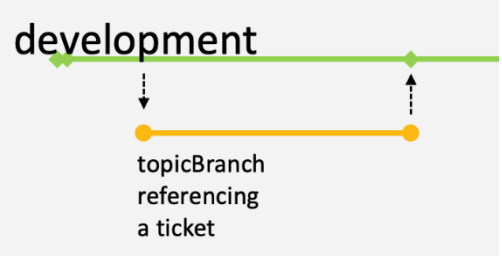
```
#
# additional libraries for compile SYSLIB concatenation, comma-separated
cobol_compileSyslibConcatenation=

#
# additional libraries for linkEdit SYSLIB concatenation, comma-separated
cobol_linkEditSyslibConcatenation=
```

5.2.1 Scenario walkthrough

In this scenario, a developer creates a new branch (called *topicBranch*) to implement the requested changes.

[13] Implementing a Release-based Development Process for Mainframe Applications
https://www.ibm.com/support/pages/node/6619083



The development branch is configured to use the high-level qualifier *GIT.APP.DEV* for the target libraries.

The feature branch requires to use a different high-level qualifier, in this case *GIT.APP.TOPIC.WI12345*.

An additional property file is created for this scenario called *featureBranchConcatenation.properties* and stored in the *application-conf* directory. This file contains overwrites for the SyslibConcatenation in the different language scripts:

```
cobol_compileSyslibConcatenation= GIT.APP.DEV.COPY, GIT.APP.DEV.BMS.COPY
cobol_linkEditSyslibConcatenation= GIT.APP.DEV.OBJ, GIT.APP.DEV.LOAD
linkedit_linkEditSyslibConcatenation= GIT.APP.DEV.LOAD
```

Additionally, zAppBuild allows you to pass additional properties file to the build framework through the *--propFiles* (or *-p*) parameter, as described in

https://github.com/IBM/dbb-zappbuild/blob/development/BUILD.md#command-line-options-summary

When running a pipeline build for a feature branch, zAppBuild must be executed with this additional property file, supplied with the *--propFiles*. The necessary libraries will then be added to the SYSLIB concatenation for the different phases of the build process for the feature branch.

When performing the first build with zAppBuild using the *--impactBuild* option, the build framework will automatically clone the collections related to the development branch (as defined by the *mainBuildBranch* property) and add the necessary concatenations.

The build is invoked with:

```
$DBB_HOME/bin/groovyz /u/dds0690/userBuildRepo/zAppBuild/build.groovy --workspace
/u/gitlab/gitlab-runner/zos/builds/GHWkdySL/0/dat/dbb-zappbuild/samples --hlq
GIT.APP.TOPIC.WI12345 --workDir /u/gitlab/gitlab-runner/zos/builds/GHWkdySL/0/dat/dbb-
zappbuild/BUILD-856 --application MortgageApplication --logEncoding UTF-8 --impactBuild --
verbose --propFiles MortgageApplication/application-conf/featureBranchConcatenation.properties
```

Then it clones the collections:

```
** Repository client created for https://10.3.20.96:10443/dbb
** Build output located at /u/gitlab/gitlab-runner/zos/builds/GHWkdySL/0/dat/dbb-
zappbuild/BUILD-856/build.20210217.111101.011
** Build result created for BuildGroup:MortgageApplication-dev-concatenation
BuildLabel:build.20210217.111101.011 at https://10.3.20.96:10443/dbb/rest/buildResult/38319
** Cloned collection MortgageApplication-dev-concatenation from MortgageApplication-master
** Cloned collection MortgageApplication-dev-concatenation-outputs from MortgageApplication-
master-outputs
** --impactBuild option selected. Building impacted programs for application
MortgageApplication
** No previous topic branch successful build result. Retrieving last successful main branch
build result.
```

Finally, it identifies the changed and impacted files:

```
** Writing build list file to /u/gitlab/gitlab-runner/zos/builds/GHWkdySL/0/dat/dbb-
zappbuild/BUILD-856/build.20210217.120051.000/buildList.txt
MortgageApplication/cobol/epsmlist.cbl
```

```
MortgageApplication/cobol/epscsmrt.cbl
MortgageApplication/cobol/epscmort.cbl
MortgageApplication/link/epsmlist.lnk
. . .
** Updating build result BuildGroup:MortgageApplication-dev-concatenation
BuildLabel:build.20210217.120051.000
*** Obtaining hash for directory /u/gitlab/gitlab-runner/zos/builds/GHWkdySL/0/dat/dbb-
zappbuild/samples/MortgageApplication
** Setting property :githash:MortgageApplication : 2ad350db6be61a6791ef7224c1420694a5231af9
** Build ended at Wed Feb 17 12:01:07 GMT+01:00 2021
** Build State : CLEAN
** Total files processed : 4
** Total build time  : 16.163 seconds
```

When checking the link listing output for MortgageApplication/link/epsmlist.lnk, we see that the SYSLIB contained also the additionally libraries:

```
DDNAME    CONCAT   FILE IDENTIFICATION

 SYSLIB      01     GIT.APP.TOPIC.WI12345.OBJ
 SYSLIB      02     GIT.APP.DEV.LOAD
 SYSLIB      04     CICSTS.V5R4.CICS.SDFHLOAD
1          *** SYMBOL REFERENCES NOT ASSOCIATED WITH ANY ADCON ***
```

Please note that by taking this approach the build is no longer fully isolated as it pulls in files from build environment of the development branch.

Be aware, that the configured concatenation is not considered in the calculation of the changed files.

This approach is most likely sufficient for a feature branch, which will be merged into the development branch anyway. It mitigates to perform a full build when you fork the new feature branch.