An IBM Document from

IBM Z DevOps Acceleration Program

March, 2021 - Release 1.1

Integrating IBM zUnit Testing into an open and modern CI/CD pipeline

David Rice
drice@us.ibm.com

Dennis Behm
dennis.behm@de.ibm.com

Mathieu Dalbin
mathieu.dalbin@fr.ibm.com

Abstract

Configure and run zUnit tests using Git, DBB, IBM Developer for z/OS, and Jenkins or GitLab

Table of content

1 Introduction

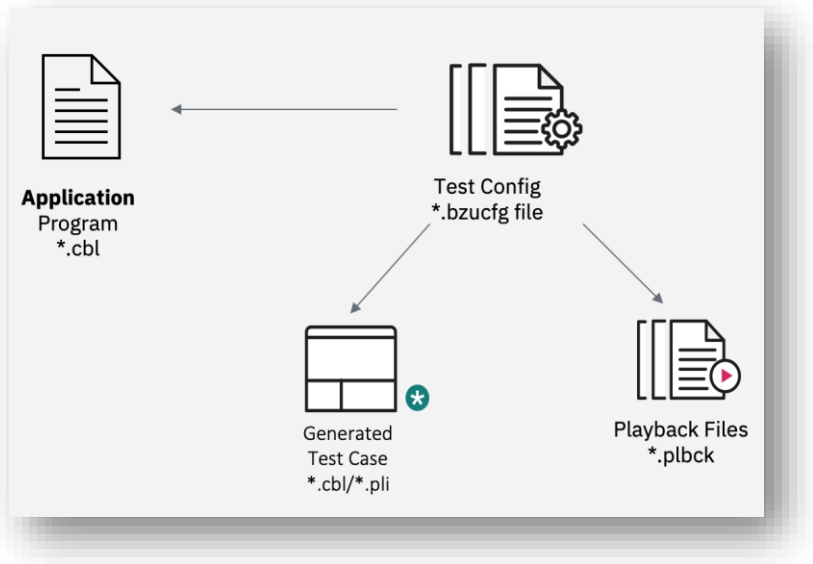
With the June/July 2020 releases of IBM Dependency Based Build, IBM Developer for z/OS (IDz) and the sample build framework zAppBuild (which leverages DBB's capabilities), it is now possible to easily integrate the execution of unit tests in an open and modern CI/CD pipeline.

The purpose of this document is to outline the steps to configure and run zUnit Test cases as part of a CI/CD pipeline.

This document will assume that the following tools have already been installed and configured:

- Rocket Software's Git,
- IBM DBB Toolkit (v1.0.9.ifix1 or newer), [1]
- IBM DBB zAppBuild framework, [2]
- Jenkins Server and Agent, or GitLab Server and Runner,
- IBM Developer for z/OS (IDz >= v14.2.3) including the z/OS Dynamic Test Runner. [3]

With the above software levels, the build framework is capable to understand the dependencies between the application and IDz zUnit test configuration artifacts.



Instructions for installing and configuring them can be found at IBM's Knowledge Center:

https://www.ibm.com/support/knowledgecenter/

Technical papers on these topics can be found in the DAP resource page:

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

[1] https://www.ibm.com/support/pages/fix-list-ibm-dependency-based-build
[2] https://github.com/IBM/dbb-zappbuild
[3] https://www.ibm.com/support/pages/fix-list-ibm-developer-z-systems-and-ibm-developer-z-systems-enterprise-edition

2 Required configuration

This section provides an overview of the necessary configuration of the application repository, which plans to integrate zUnit tests into their pipeline.

The required modifications to the global configuration will affect different areas:

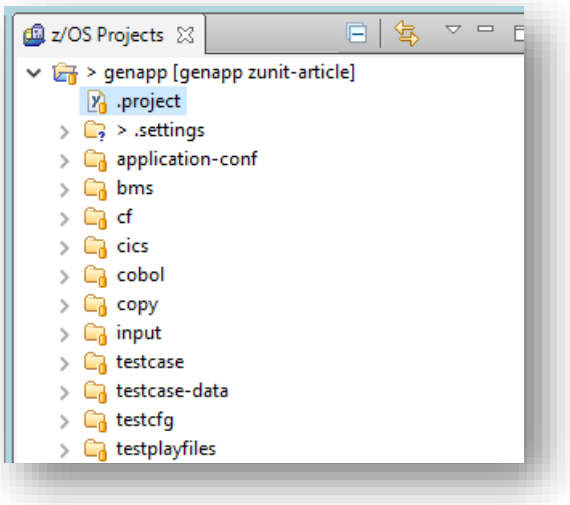
- for convenience, additional subfolders to manage the zUnit test artifacts will need to be created,
- specific settings located in the *application-conf* folder will be customized,
- finally, some IDz workstation preferences will be modified accordingly.

2.1 Overview of Application repository structure

In addition to your standard application folders, zUnit now has the ability to store test artifacts into multiple directories in the local project, which should be managed in a git repository.

With that capability, it is far easier to manage the zUnit test files for processing with DBB and zAppBuild, as everything is located in the git repository and can be shared between the developers. You can version control all the unit test related artifacts along with your source files.

The basic structure should look like the following:



The zUnit related folders are:

- *testcase* – This is where the generated zUnit test cases are stored.
- *testcase-data* – This is where the test case generation configuration (*.json) are stored.
- *testplayfiles* - This is where the zUnit playback files are stored.
- *testcfg* – This is where the zUnit runner configuration files (or *bzucfg* files) are stored. These *bzucfg* files are key to processing zUnit test cases from a pipeline as they point to the test case and *testplayfile*.

Note: The above names of these subdirectories can be customized to your naming conventions. Also, it is possible to manage the different subdirectories nested in a folder **zunit_tests** to avoid having too many folders next to the actual application code itself.

2.2 zAppBuild application-specific settings

2.2.1 Application-Conf properties

In order to make zUnit run in a zAppBuild build, you will need to set the following properties in the application configuration of the application repository.

A sample template of the property files is provided via zAppBuild at https://github.com/IBM/dbb-zappbuild/tree/development/samples/application-conf

2.2.1.1 application.properties

The following table provides a list of necessary properties to run zUnit within zAppBuild, which are configured in the *application.properties* file.

Property	Description	Additional Comments
runzTests=true	set to "true", zAppBuild will build and execute the zUnit tests found in the repository.	This is optional, there is also a command line Boolean property you can set *-zTest* or *--runzTests*
applicationPropFiles	Add a reference to the new *ZunitConfig.properties* file	
testOrder=ZunitConfig.groovy	*build.groovy* will execute the scripts defined in the *testOrder* after it has the processed the language scripts defined in the *buildOrder*.	
jobCard	The *ZunitConfig.groovy* script generates and runs a JCL. Specify the job card according to your environment specifics.	Additionally, you can use a jobname which relates to your application.
impactResolutionRules	Add the below resolution rules related to unit tests for understanding the dependencies in an Impact Build scenario	
testconfigRule	Defines the resolution rule for the test configuration files. The dependency of library "SYSPROG" will be resolved to files located in the specified folder of this rule	Dependencies are described via library SYSPROG
testcaseRule	Defines the resolution rule for the test play files. The dependency of library "SYSPLAY" will be resolved to files located in the specified folder of this rule	Dependencies are described via library SYSPLAY

For reference, see below a sample of these definitions. You have the complete set in the zAppBuild repository mentioned previously.

```
#
# Run zUnit Tests
# Defaults to "false", to enable, set to "true"
#runzTests=true

#
# Comma seperated list of the test script processing order
testOrder=ZunitConfig.groovy

#
# Job card, please use \n to indicate a line break and use \ to break the line in this
property file
# Example: jobCard=//RUNZUNIT JOB ,MSGCLASS=H,CLASS=A,NOTIFY=&SYSUID,REGION=0M
jobCard=//RUNZUNIT JOB ,MSGCLASS=H,CLASS=A,NOTIFY=&SYSUID,REGION=0M

#
# Impact analysis resolution rules (JSON format).
# Defaults to just looking for local application dependency folders
impactResolutionRules=[${copybookRule},${plincRule},${maclibRule},${testcaseRule},${testconfig
Rule}]

testconfigRule =   {"library": "SYSPROG", \
              "searchPath": [ \
                 {"sourceDir": "${workspace}", "directory": "${application}/testcfg"} \
              ] \
            }

testcaseRule =   {"library": "SYSPLAY", \
              "searchPath": [ \
                 {"sourceDir": "${workspace}", "directory": "${application}/testplayfiles"} \
              ] \
            }
```

2.2.1.2 *file.properties*

The following properties managed in the *file.properties* file define necessary mappings for processing the test artifacts during a zAppBuild build process.

Property	Description	Additional Comments
dbb.scriptMapping = ZunitConfig.groovy :: **/*.bzucfg	The *bzucfg* files in your application repository contain the test runner configuration and needs to be processed by the new *ZunitConfig.groovy* language script. You describe here the config files mapped to this script.	
dbb.scannerMapping = ZUnitConfigScanner :: **/*.bzucfg	The *bzucfg* file is not a standard enterprise source code. There is a new scanner embedded in the DBB toolkit starting v.1.0.9.ifix1 to handle it. This mapping it necessary to map the config files to the right scanner.	
cobol_testcase = true :: **/testcase/*.cbl	The standard *Cobol.groovy* language script will be used to compile and link the generated test case. This file level property allows to identify the test case programs specifically in the scripts and handle special cases for them.	Even if testcases are COBOL programs, they certainly need to be handled differently from the other COBOL programs: It is a recommendation to target separate libraries for source and build outputs. Please also see the additional library definitions in *build-conf/Cobol.properties*.[4]

[4] https://github.com/IBM/dbb-zappbuild/blob/development/build-conf/Cobol.properties

2.2.1.3 ZunitConfig.properties

The following *ZunitConfig.properties* application file properties customize your zUnit Test behavior during a zAppBuild process.

Property	Description	Additional Comments
zunit_maxPassRC	(default is 4) By default, when running a zUnit Test case, if the test has a return code of 0-4, it will mark the test as "Passed", this can be adjusted based on desired usage.	
zunit_maxWarnRC	(default is 8) By default, when running a zUnit Test case, if the test has a return code of >4 and <=8, it will mark the test as "Warning". Warnings will not stop the build process but will create a warning message. Anything beyond that max return code value will return a "Failure" and will stop the build process. this can be adjusted based on desired usage.	
zunit_playbackFileExtension	The zUnit playback file is managed as a binary file. This property indicates to load the *playbackfile* as a binary into the target dataset.	Please make sure, that the file extension is mapped to binary in the *.gitattributes* file. Note: a more general mechanism for handling binary file will be introduced in zAppBuild in the future.
zunit_resolutionRules	Links to the resolution rule in *application.properties*. This defines the rules for resolving dependent files (e.g. the zUnit playback file). These files will be loaded to the target datasets when processing the zUnit configuration file (*bzucfg*) file.	

2.2.2 Parametrization of build.groovy

zUnit test cases will not run by default. In order to tell zAppBuild to run them, you either need to set the *runzTests* property in the *application.properties* file as described earlier, or you can alternatively use the *--runzTests* flag as a parameter passed when invoking zAppBuild:

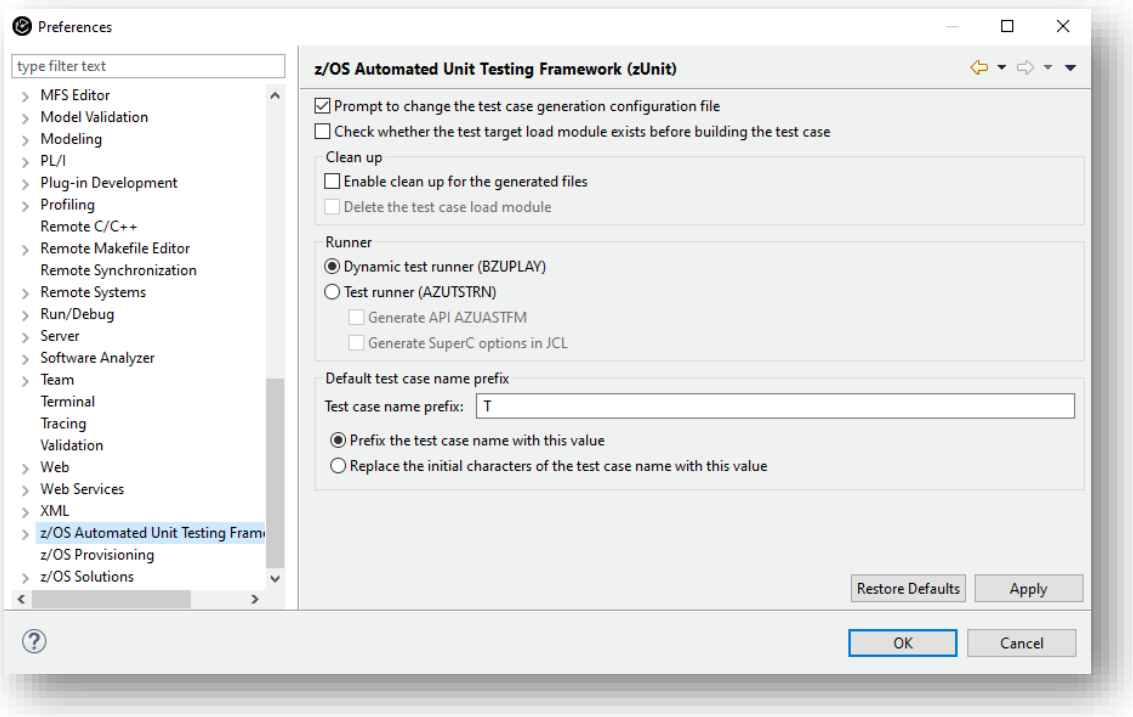
```
-zTest,--runzTests Specify if zUnit Tests should be run
```

Please also see: https://github.com/IBM/dbb-zappbuild/blob/development/BUILD.md

2.3 Required IDz client configuration

Make sure that the your IDz environment is configured to use the z/OS Dynamic Test Runner.

Please review your IDz settings under *Window > Preferences … z/OS Automated Unit Testing Framework*. This will make sure, that the generated test cases use the Dynamic Test Runner.

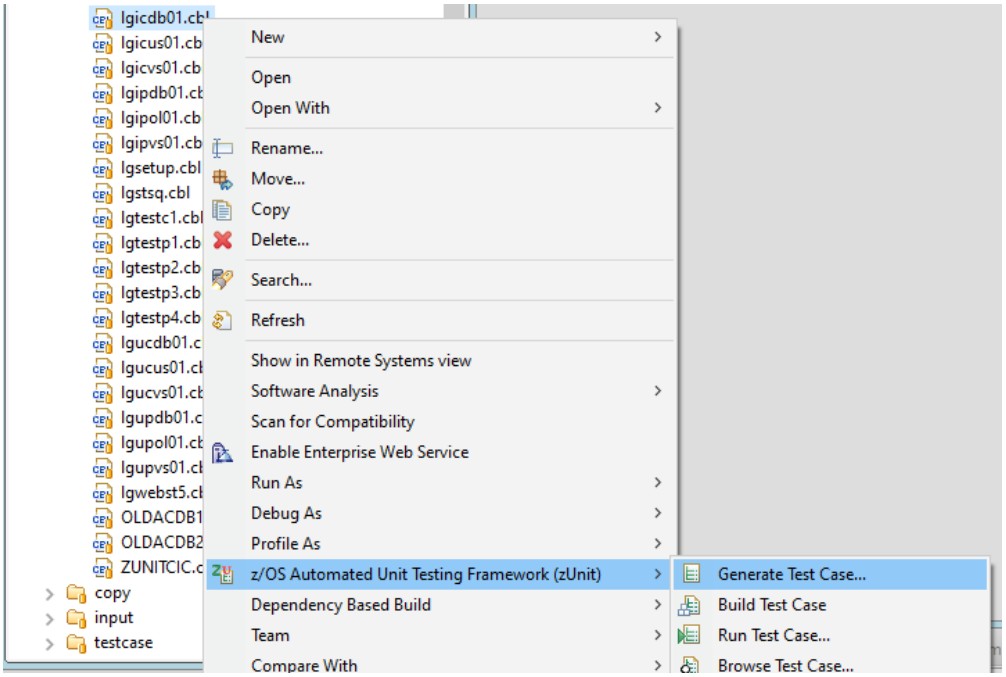


3 Workflow

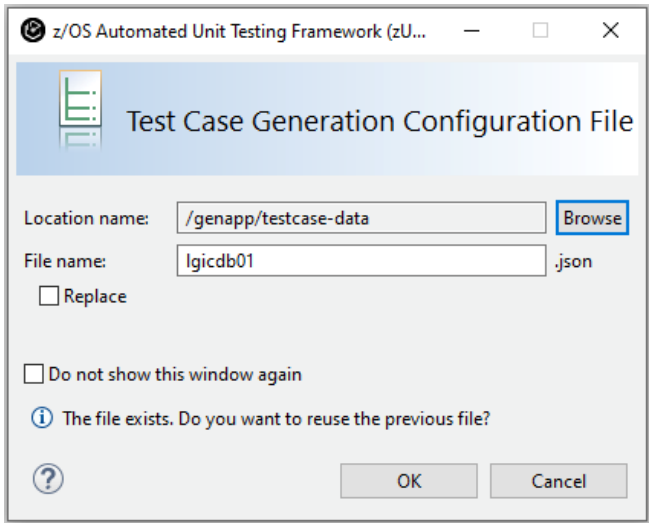
3.1 Create the Test Case, Record Data, and Generate Test Case

We assume, that you already have cloned your git repository and imported the application project into the IDz workspace.

To create/modify the Test Case, right-click on the program being tested, in our case *lgicdb01.cbl* and select *z/OS Automated Unit Testing Framework (zUnit)*:

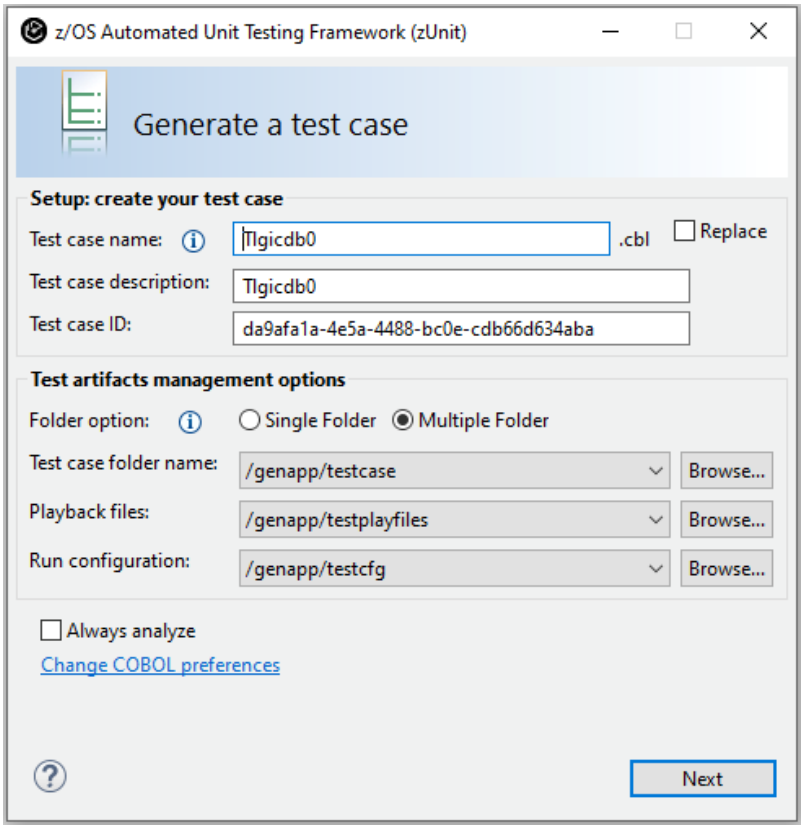


This will open a dialog box where we can specify the project, folder and name where the test case generation configuration file will be placed. In our case, this file is stored in the *testcase-data* folder.

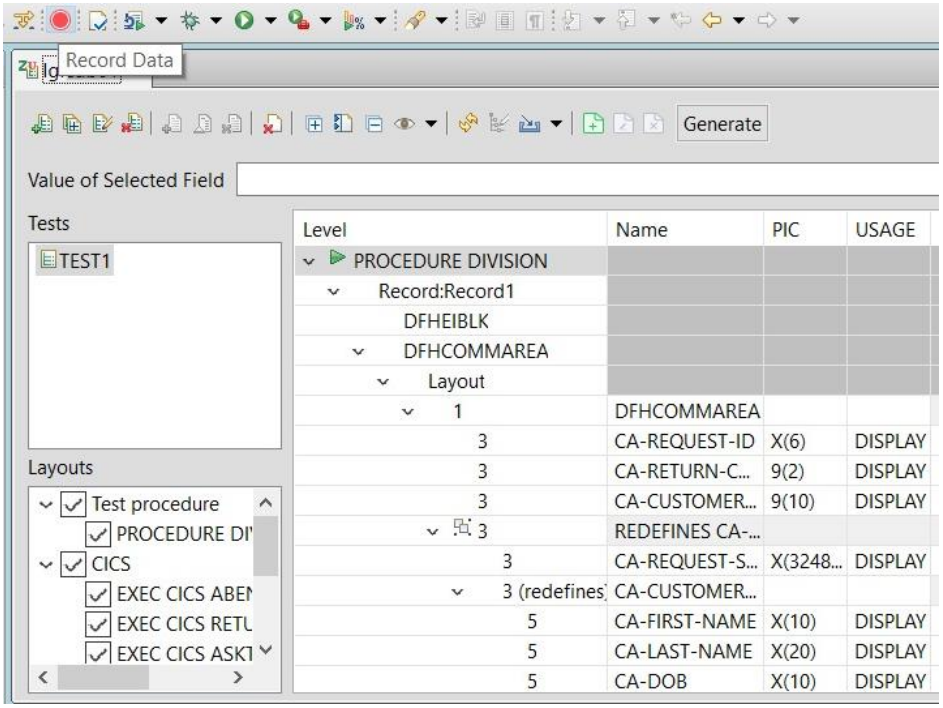


In the next dialog, select the destination folders for the configuration file, playback file and Test Case program. Select the destination folders in the project you created and click Next.

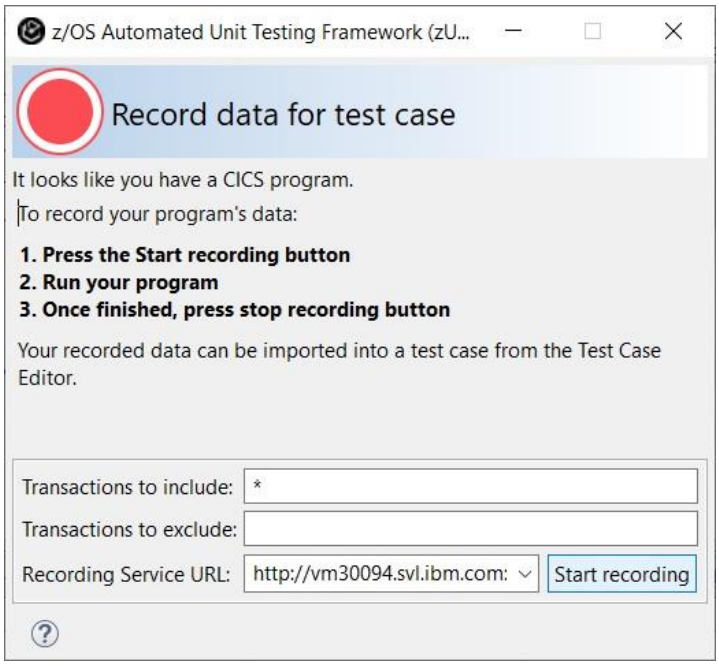
It is important to note that DBB requires the files to be located in different subfolders to enable the correct dependency analysis and resolution.



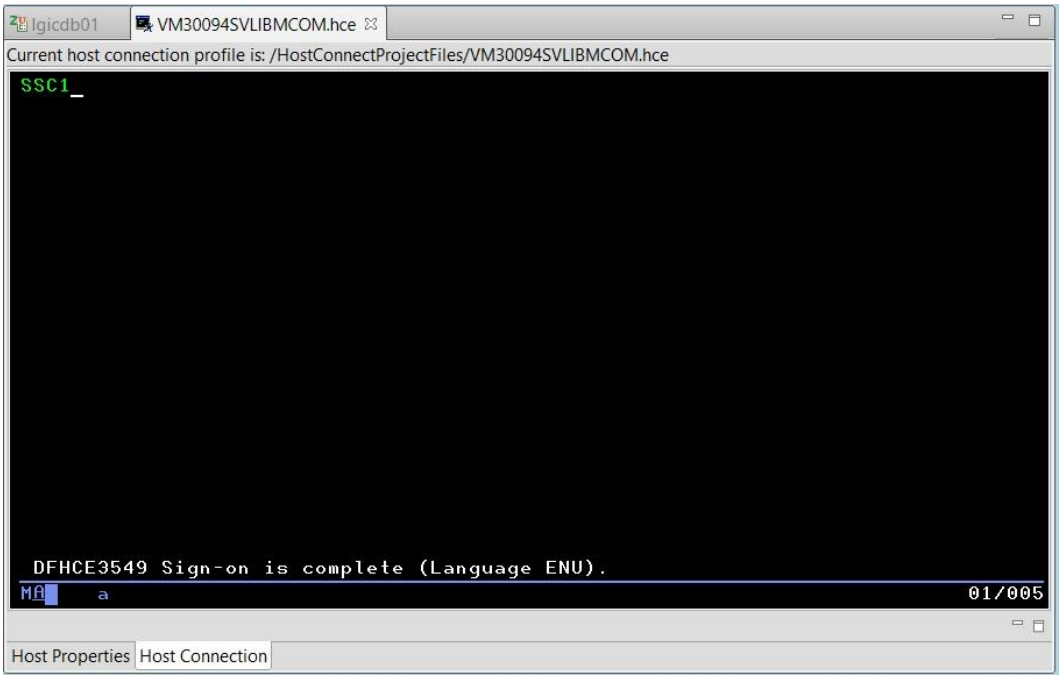
We are not going to change anything here, but we are going to start recording, so click on the Record data button on the action bar:



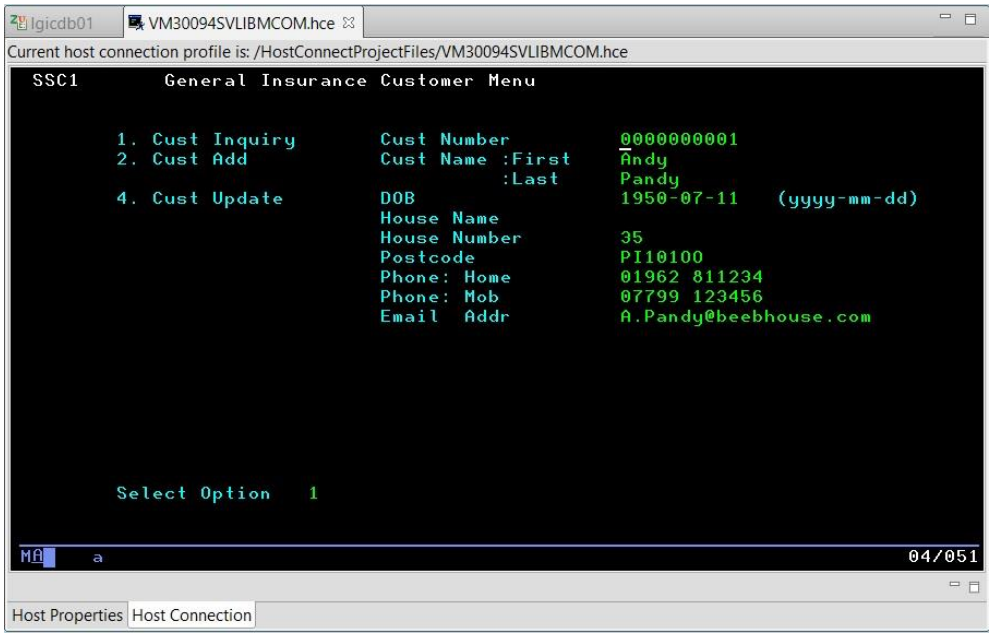
The recording will start when you press the Start Recording button:



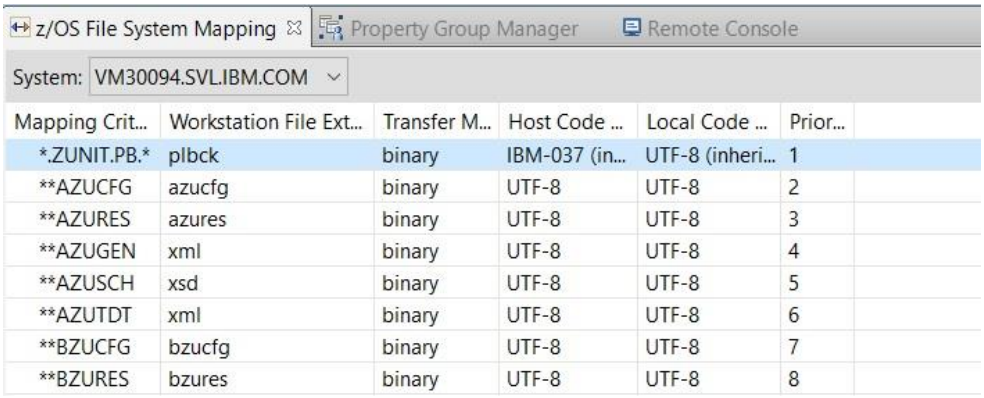
As our example is a COBOL/CICS/DB2 program we are going to login to our CICS Region and execute the transaction with the corresponding COBOL programs. Login to CICS and execute the CICS transaction, in our case SSC1:



Then perform some function in the transaction, in our example performing a Customer Inquiry:

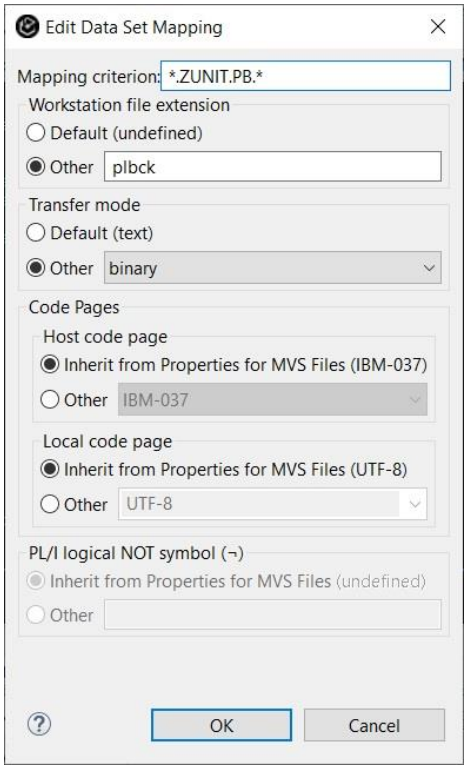


We have executed our CICS transaction, so we need to make sure that the data set mapping is correct, especially for the playback file. This file needs to be stored in EBCDIC and in binary such that non-roundtripable characters are not affected. Go to the *z/OS File System Mapping* view:

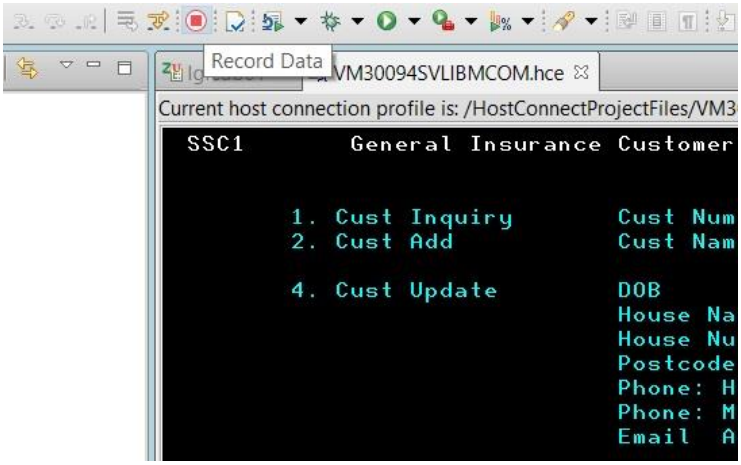


Mapping Crit...	Workstation File Ext...	Transfer M...	Host Code ...	Local Code ...	Prior...
.ZUNIT.PB.	plbck	binary	IBM-037 (in...	UTF-8 (inheri...	1
**AZUCFG	azucfg	binary	UTF-8	UTF-8	2
**AZURES	azures	binary	UTF-8	UTF-8	3
**AZUGEN	xml	binary	UTF-8	UTF-8	4
**AZUSCH	xsd	binary	UTF-8	UTF-8	5
**AZUTDT	xml	binary	UTF-8	UTF-8	6
**BZUCFG	bzucfg	binary	UTF-8	UTF-8	7
**BZURES	bzures	binary	UTF-8	UTF-8	8

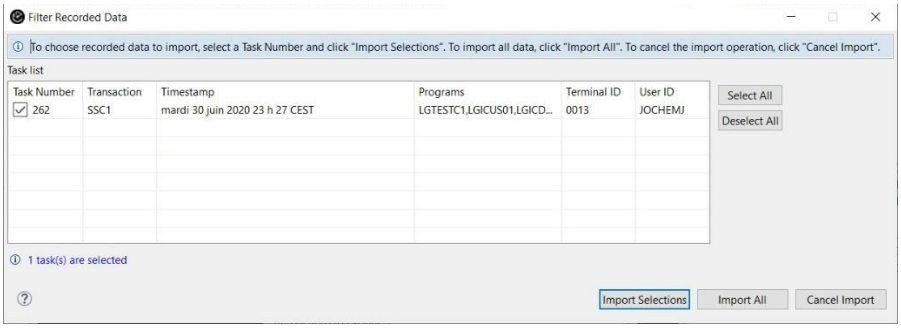
Select the mapping for the playback file if it exists, or right click in the view and select *Add Data Set Mapping* if it does not. Remember, right at the beginning, in our preferences, we select the name that would be used to create the playback file. The mapping we have created aligns with that:



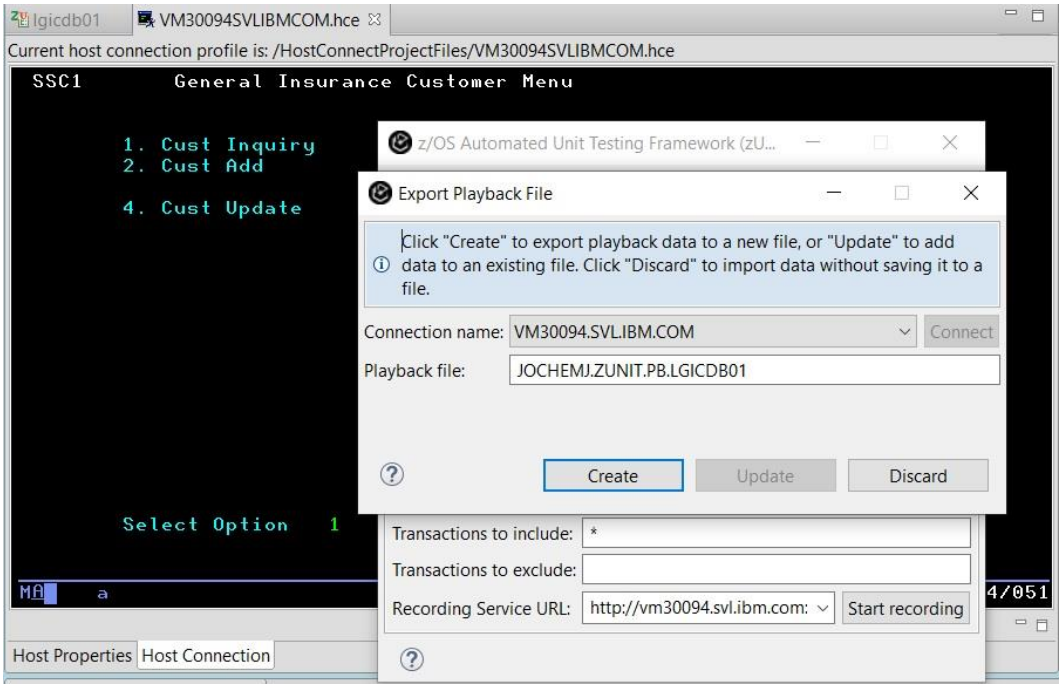
We need to make sure that the transfer mode is set to Binary, and the Host Code Page used is an EBCDIC one, either IBM-037 or IBM-1047. Once we have checked our settings, we can stop the recording by clicking the Record Data button again:



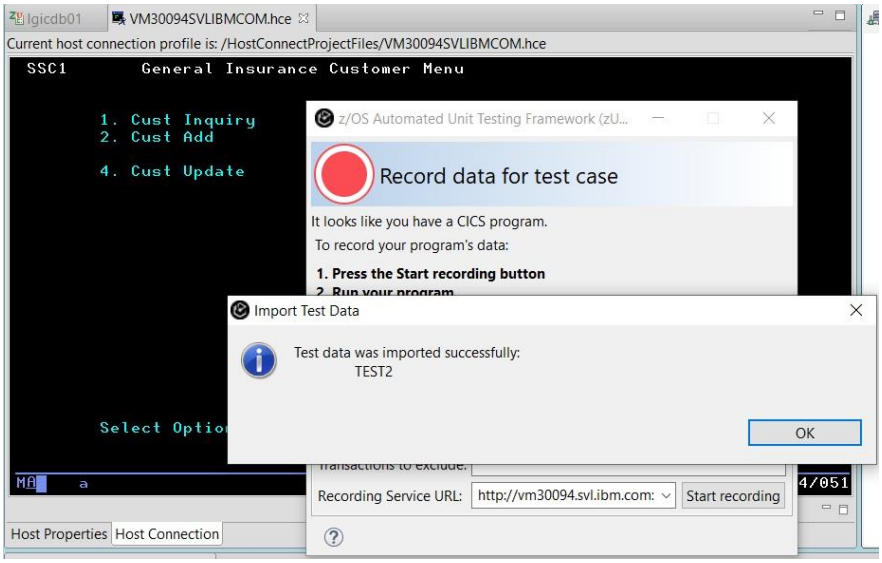
When the recording session is stopped the Filter Recorded Data dialog opens. So the next step is to select the transactions to import.



Click on the *Import Selections* button and select the RSE connection to the system where the playback file will be created and click Create, which will create the playback file in a z/OS data set and then import it to the local project:

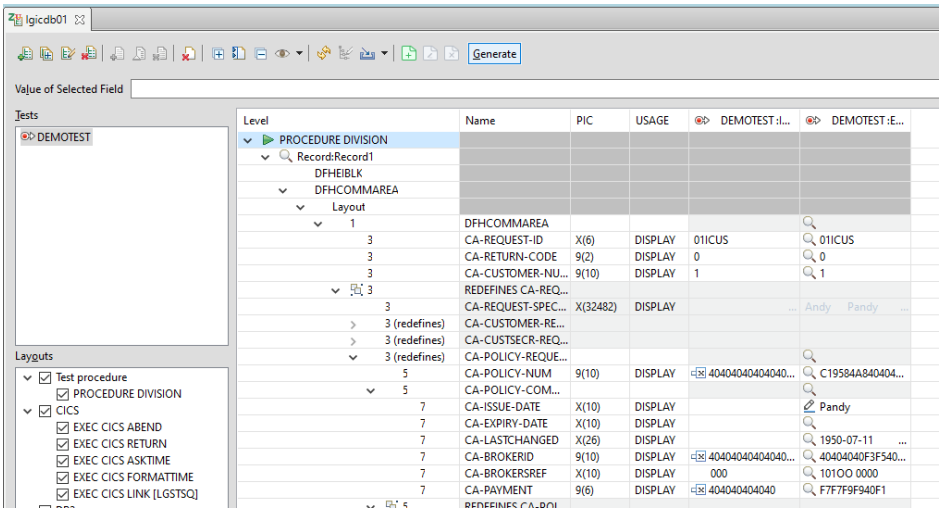


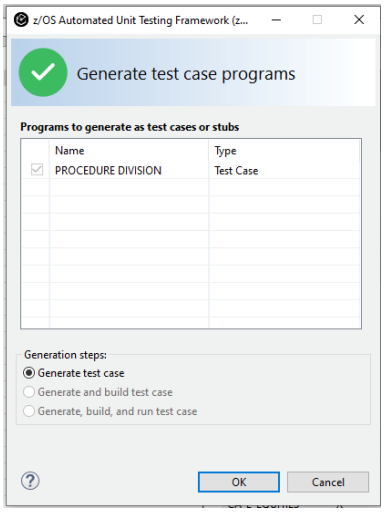
Once the connection is selected and the playback file name is confirmed click Create and the playback file will be created on the z/OS system specified.



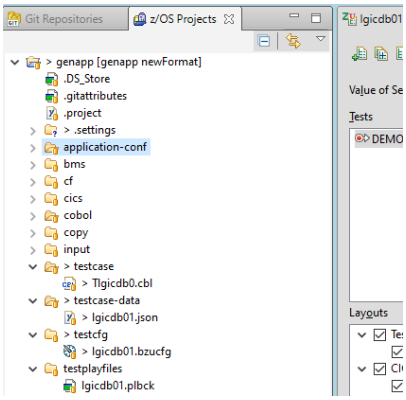
Notice in the local project that the playback file now exists, but the test case program and the test configuration file do not exist yet / or require to be regenerated.

So, go back to our Test case editor view and select the *Generate* button:





This will generate the relevant zUnit files – the generated test case, the *bzucfg* test configuration file and also update the test case generation file:



We are now at a stage, where we can push the changes to the central git provider.

3.2 zAppBuild's Impact Build after changing the test setup

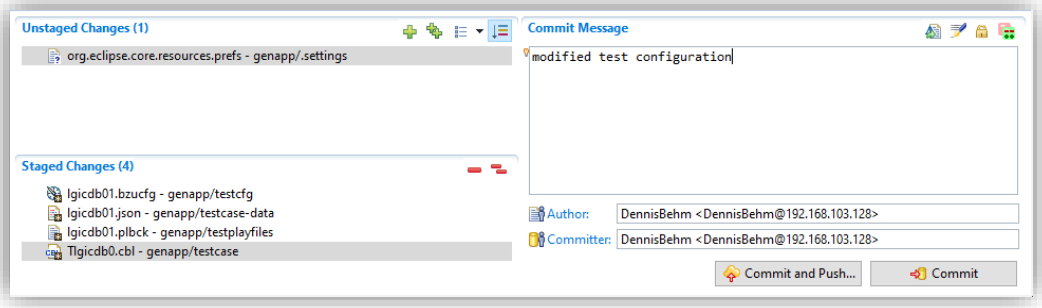
The zAppBuild framework heavily relies on the commits made in git for Impact Builds.

So, when making changes to zUnit tests (equivalent to the test setup), it is important to make sure the impacted files are stored into the git repository or are defined in the build scope.

The following files are important and which make up a zUnit test scenario:

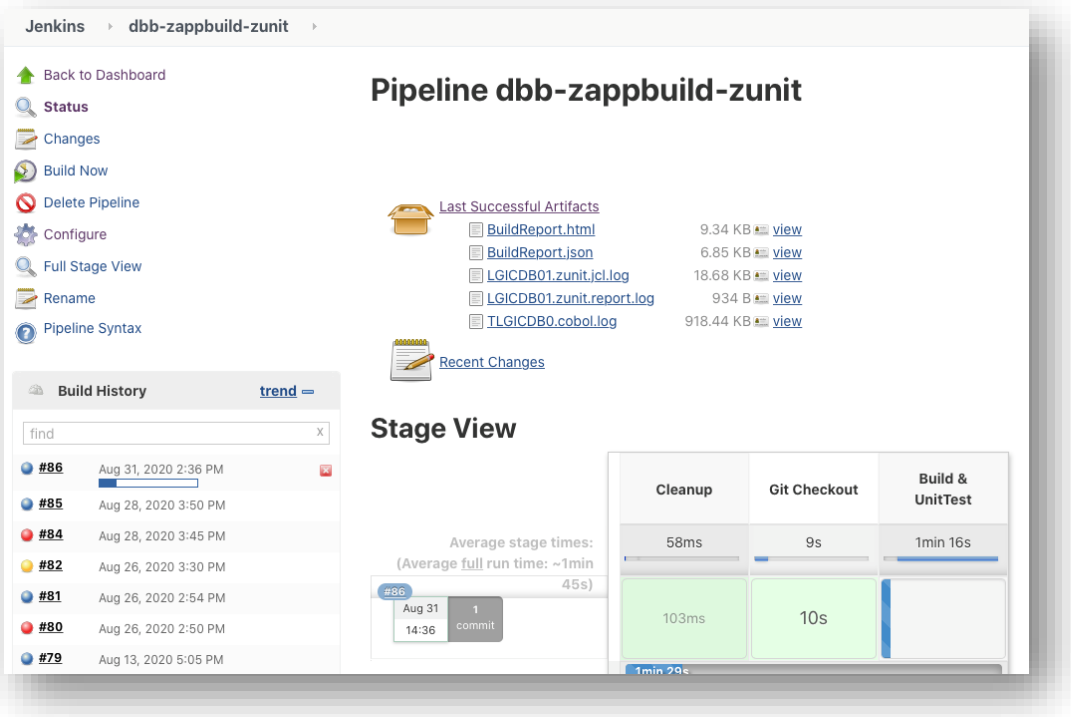
- The generated Test case file – a generated Cobol program,
- The *bzucfg* file – This specific file contains the test configuration. It is not updated all the time, so you may not push it to the server as often as the other files, because unless you are adding/changing the playback files, the only changes made to a *bzucfg* file is a uid. This is not a problem, as dependencies are established from this file, and when they change, they will trigger running the test through this file.
- The *bzuplay* file – it contains the captured data to run the test without an actual execution environment.
- The zUnit JSON file – this file contains all the needed information about the zUnit tests for the client tool in IDz. It is not necessary to push this file to git in order to run zUnit tests, that being said it is important to keep this file tracked in git for version control purposes and to be able to share the test data configuration with your development peers.

In the above workflow, we created a new test without modifying the application program. Next, we will commit the new test scenario to git, and run a build using Jenkins. The first step is to commit the changes and push them to the central git server.

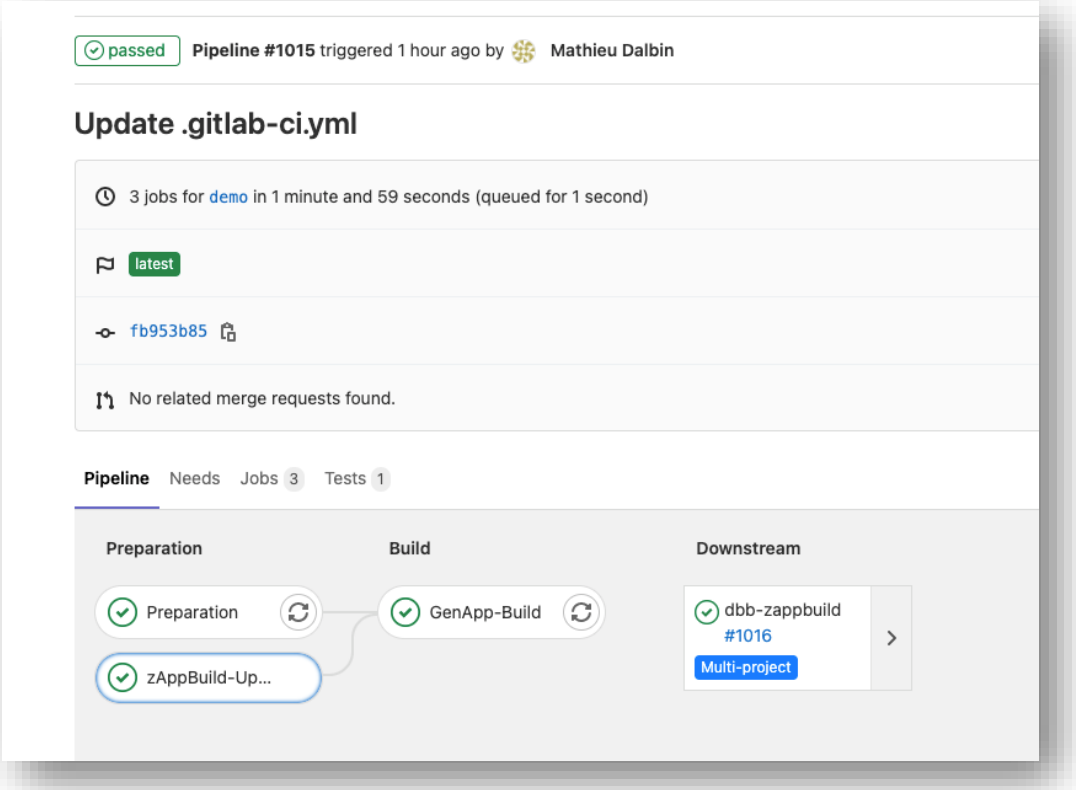


The next time we build using Jenkins or GitLab, the build will pick up these changes.

Jenkins:



GitLab:



With a closer look to the console log, we see that when zAppBuild is invoked, it will add the following files to the build list.

```
** Writing build list file to /var/dbb/buildhome/workspace/dbb-zappbuild-
zunit/work/build.20200828.015107.051/buildList.txt
genapp/testcase/Tlgicdb0.cbl
genapp/testcfg/lgicdb01.bzucfg
```

Let's have a look to the records in the DBB collection stored on the DBB WebApp. The collection for source level dependencies contain both the generated test case *tlgicdb0.cbl* and the test configuration file *lgicdb01.bzucfg*.

TLGICDB0	genapp/testcase/Tlgicdb0.cbl	COB	link
LGICDB01	genapp/testcfg/lgicdb01.bzucfg	ZUNITCFG	link

The generated COBOL test case references a system copybook. Which is included to the SYSLIB concatenation through the setting *BZUSAMP* in *build-conf/dataset.properties*.

DBB LogicalFile

Field	Value
id	35897
lname	TLGICDB0
file	genapp/testcase/Tlgicdb0.cbl
language	COB
cics	false
sql	false
dli	false

Logical Dependencies (count=1)

lname	category	library	link
BZUITERC	COPY	SYSLIB	link

The data which was captured for the test configuration file connects the test configuration file *bzucfg* with the application program, the generated test case and the playback file.

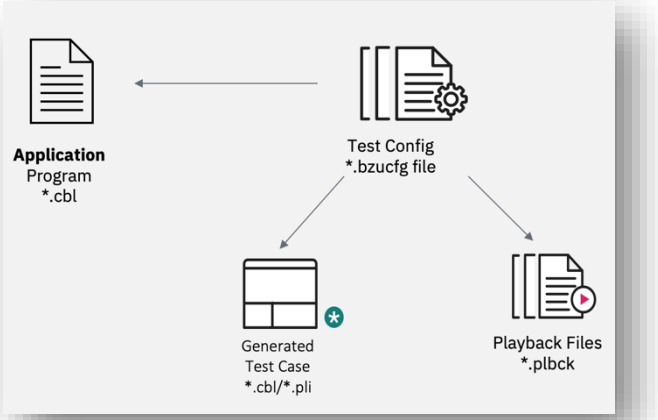
DBB LogicalFile

Field	Value
id	25187
lname	LGICDB01
file	genapp/testcfg/lgicdb01.bzucfg
language	ZUNITCFG
cics	false
sql	false
dli	false

Logical Dependencies (count=3)

lname	category	library	link
LGICDB01	ZUNITINC	SYSPROG	link
LGICDB01	ZUNITINC	SYSPLAY	link
TLGICDB0	ZUNITINC	SYSTEST	link

Based on this information the dependency resolver is capable to understand the dependencies.



The generated test case will be compiled and linked through the *COBOL.groovy* language script but will be stored in a different output dataset.

Let's have a closer look to the processing of the test case configuration file *bzucfg*, while the *--verbose* option was specified to zAppBuild.

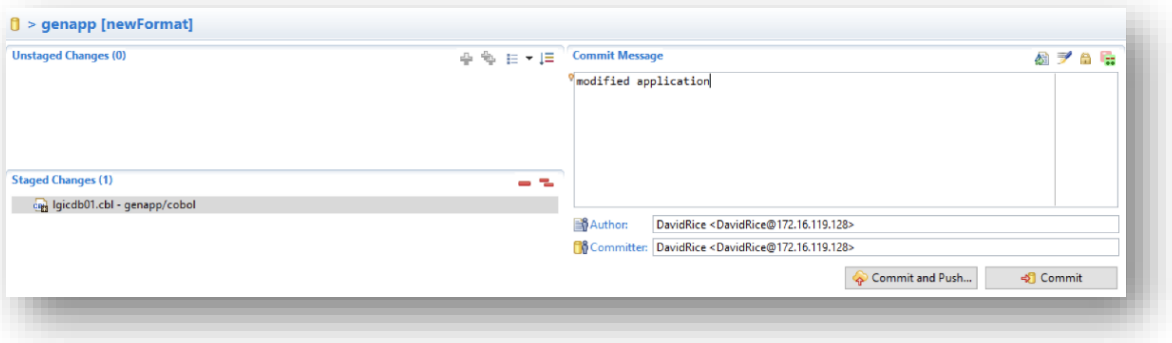
Action	Console log
zAppBuild resolve dependencies and load them to the build datasets including the bzuplay file.	** Building files mapped to ZunitConfig.groovy script *** Building file genapp/testcfg/lgicdb01.bzucfg *** Creating dependency resolver for genapp/testcfg/lgicdb01.bzucfg with [{"library": "SYSPLAY", "searchPath": [{"sourceDir": "/var/dbb/buildhome/workspace/dbb-zappbuild-zunit/genapp-zunit", "directory": "genapp/testplayfiles"}] }] rules *** Scanning file with the ZUnitConfigScanner *** Resolution rules for /var/dbb/buildhome/workspace/dbb-zappbuild-zunit/genapp-zunit/genapp/testcfg/lgicdb01.bzucfg: {"library":"SYSPLAY","searchPath":[{"sourceDir":"\/var\/dbb\/buildhome\/workspace\/dbb-zappbuild-zunit\/genapp-zunit","directory":"genapp\/testplayfiles"}]} ***** Physical dependencies for /var/dbb/buildhome/workspace/dbb-zappbuild-zunit/genapp-zunit/genapp/testcfg/lgicdb01.bzucfg:** **{"sourceDir":"\/var\/dbb\/buildhome\/workspace\/dbb-zappbuild-zunit\/genapp-zunit","lname":"LGICDB01","library":"SYSPLAY","file":"genapp\/testplayfiles\/lgicdb01.plbck","category":"ZUNITINC","resolved":true}**
Generated JCL will be displayed	//RUNZUNIT JOB ,MSGCLASS=H,CLASS=A,NOTIFY=&SYSUID,REGION=0M //* //BADRC EXEC PGM=IEFBR14 //DD DD DSN=&SYSUID..BADRC,DISP=(MOD,CATLG,DELETE), // DCB=(RECFM=FB,LRECL=80),UNIT=SYSALLDA, // SPACE=(TRK,(1,1),RLSE) //* //* Action: Run Test Case... //RUNNER EXEC PROC=BZUPPLAY, **// BZUCFG=DRICE.DBB.ZUNITPL.BZU.BZUCFG(LGICDB01),** // BZUCBK=DRICE.DBB.ZUNITPL.TEST.LOAD, // BZULOD=DRICE.DBB.ZUNITPL.LOAD, // PARM=('STOP=E,REPORT=XML') //BZUPLAY DD DISP=SHR, **// DSN=DRICE.DBB.ZUNITPL.BZU.BZUPLAY(LGICDB01)** //BZURPT DD DISP=SHR, // DSN=DRICE.DBB.ZUNITPL.BZU.BZURPT(LGICDB01) //* //IFGOOD IF RC<=4 THEN //GOODRC EXEC PGM=IEFBR14 //DD DD DSN=&SYSUID..BADRC,DISP=(MOD,DELETE,DELETE), // DCB=(RECFM=FB,LRECL=80),UNIT=SYSALLDA, // SPACE=(TRK,(1,1),RLSE) // ENDIF
Reporting the result and store information in the build result	*** zUnit Test Job RUNZUNIT(JOB04033) completed with 0 ***************** Module [TLGICDB0] ***************** Name: TLGICDB0 Status: pass Test cases: 1 (1 passed, 0 failed, 0 errors) Details: TEST2 pass ***************** Module [TLGICDB0] ***************** ** Writing build report data to /var/dbb/buildhome/workspace/dbb-zappbuild-zunit/work/build.20200828.015107.051/BuildReport.json ** Writing build report to /var/dbb/buildhome/workspace/dbb-zappbuild-zunit/work/build.20200828.015107.051/BuildReport.html ** Updating build result BuildGroup:genapp-newFormat BuildLabel:build.20200828.015107.051

3.3 zAppBuild's Impact Build after changing an application program

In the previous section, we walked through the process of modifying the test case, pushing to git, and running a pipeline with a CI orchestrator. In this step we will show the workflow of modifying the application code rather than the test case.

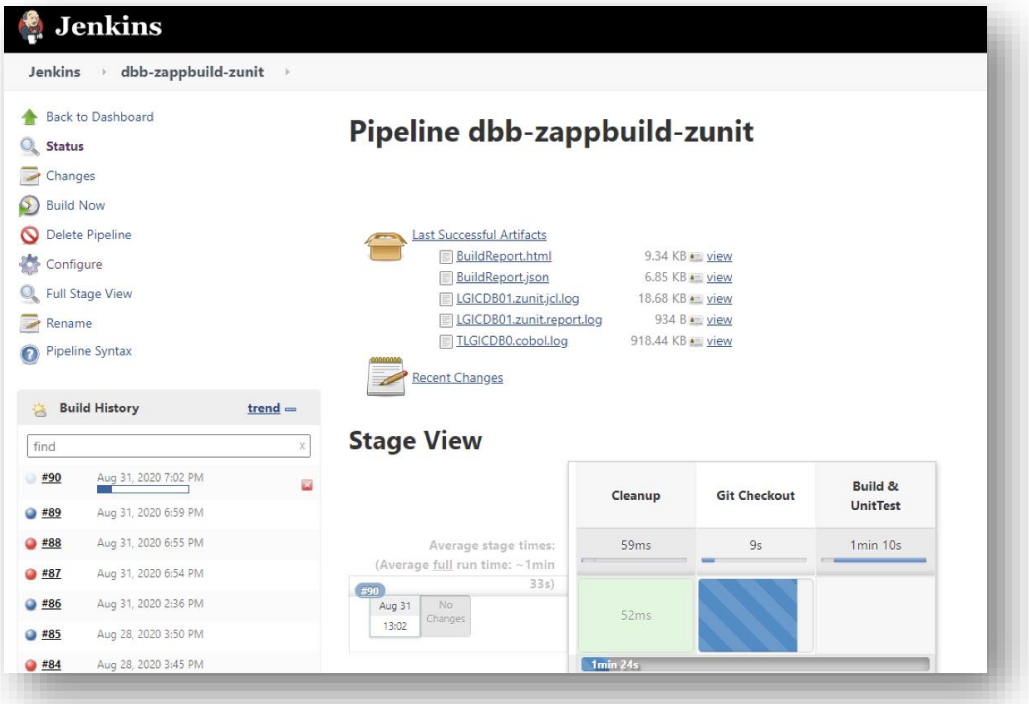
In this example we will assume a change was made to the application code.

The first step is to commit the changes and push of the application program *lgicdb01.cbl* to the central git server.

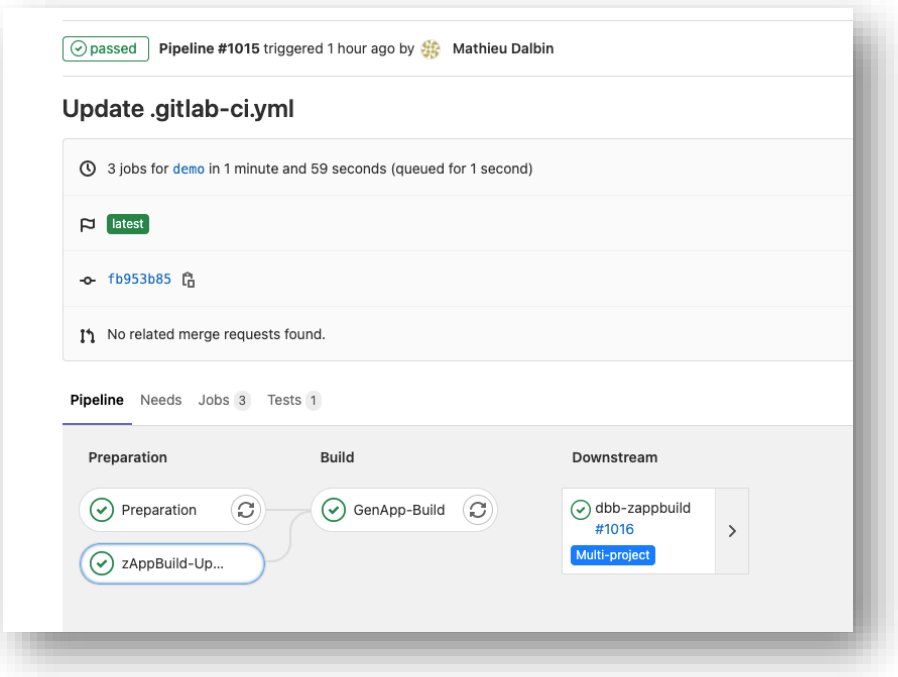


The next time we build, the build will pick up this change.

Jenkins:



GitLab:



With a closer look to the console log, we, see that when zAppBuild is invoked, it will add the modified file *lgicdb01.cbl* and its impacted files to the build list.

The test configuration file *lgicdb01.bzucfg* is added to the build list as an impacted file. This is a result of our configurations and DBBs understanding about the dependencies between the application source code and available test configurations – which are represented through the zUnit test configuration files. With the new scanner capability, the zUnit test configuration files are scanned, and the dependencies get captured and stored in the DBB webapp.

```
** Performing impact analysis on changed file genapp/cobol/lgicdb01.cbl
*** Creating impact resolver for genapp/cobol/lgicdb01.cbl with [{"library": "SYSLIB",
"searchPath": [ {"sourceDir": "/var/jenkins/workspace/genapp-zunit/genapp", "directory":
"genapp/copy"} ]               },{"library": "SYSLIB", "searchPath": [ {"sourceDir":
"/var/jenkins/workspace/genapp-zunit/genapp", "directory": "genapp/bms"} ]
},{"category": "LINK", "searchPath": [ {"sourceDir": "/var/jenkins/workspace/genapp-
zunit/genapp", "directory": "genapp/cobol"}, {"sourceDir": "/var/jenkins/workspace/genapp-
zunit/genapp", "directory": "genapp/link"} ]            },{"library": "SYSPLAY",
"searchPath": [ {"sourceDir": "/var/jenkins/workspace/genapp-zunit/genapp", "directory":
"genapp/testplayfiles"} ]            },{"category": "ZUNITINC", "searchPath": [ {"sourceDir":
"/var/jenkins/workspace/genapp-zunit/genapp", "directory": "genapp/testcfg"}, {"sourceDir":
"/var/jenkins/workspace/genapp-zunit/genapp", "directory": "genapp/testcase"} ]            }]
rules
** Found impacted file genapp/testcfg/lgicdb01.bzucfg
** genapp/testcfg/lgicdb01.bzucfg is impacted by changed file genapp/cobol/lgicdb01.cbl.
Adding to build list.
** Writing build list file to /var/dbb/buildhome/workspace/dbb-zappbuild-
zunit/work/build.20200831.050224.002/buildList.txt
genapp/cobol/lgicdb01.cbl
genapp/testcfg/lgicdb01.bzucfg
```

The application will be compiled and linked through the *COBOL.groovy* language script, while the test configuration file is processes by the *ZunitConfig.groovy* script. Please see the previous section for the console log.

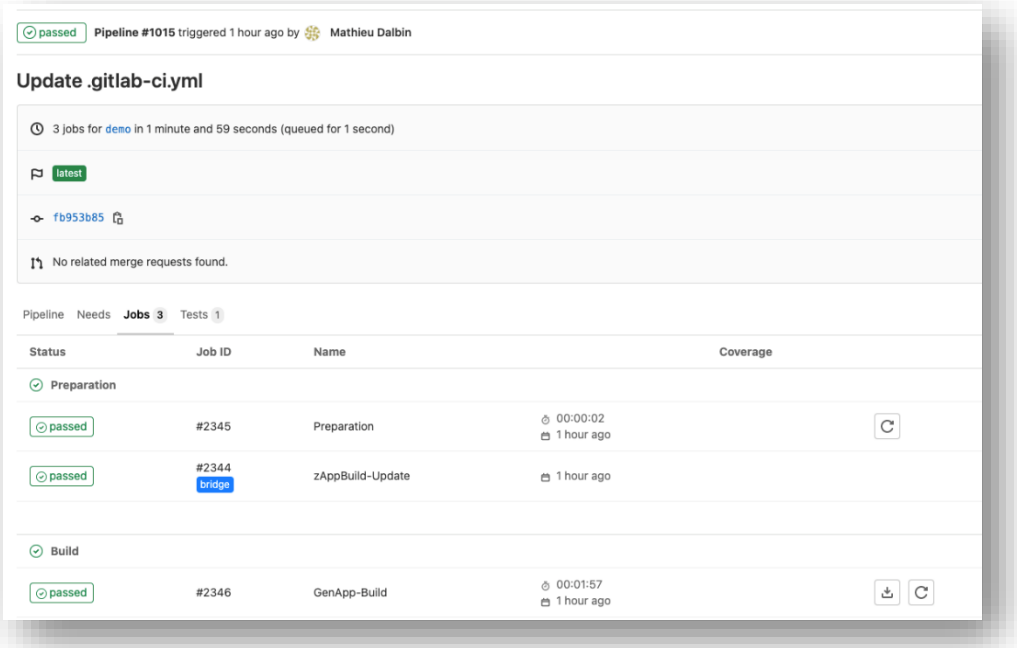
4 Publishing the Unit tests results into the pipeline

After the execution of Unit tests, the test results are generated into several files that can uploaded for archiving purposes in the pipeline.

- *zunit.jcl.log contains the spool of the job
- *.zunit.report.log contains the BZU report

With Jenkins, you can store these test results with the publisher plugin:

With GitLab, you can archive these files through the artifact keyword of the CI/CD pipeline. For the stages that produce such artifacts, a download button is available, or artifacts can be browsed through the GitLab interface:



zUnit test results can be transformed with the help of an XSL transformation process into standard industry reports for unit test results (like JUNIT or SonarQube).

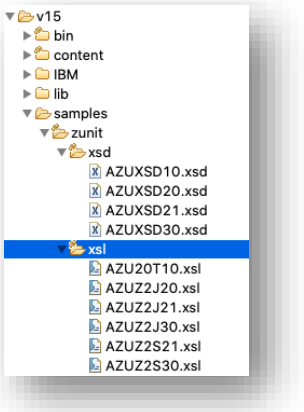
You can easily transform the zUnit report into the JUnit format, which can be added to the pipeline build result as well.

For Jenkins, a call to the IBM z/OS XML toolkit program *Xalan* can be integrated into the Jenkins pipeline like this:

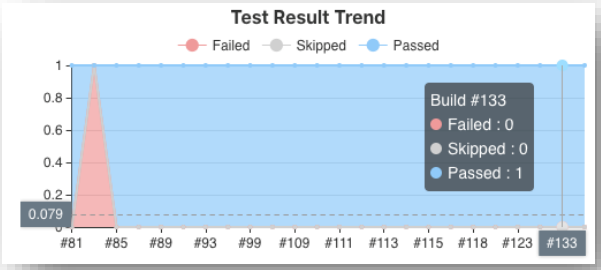
```
sh "Xalan -o ${WORKSPACE}/work/junit.xml ${WORKSPACE}/work/*/*.zunit.report.log
/var/dbb/extensions/zunit2junit/AZUZ2J30.xsl"
```

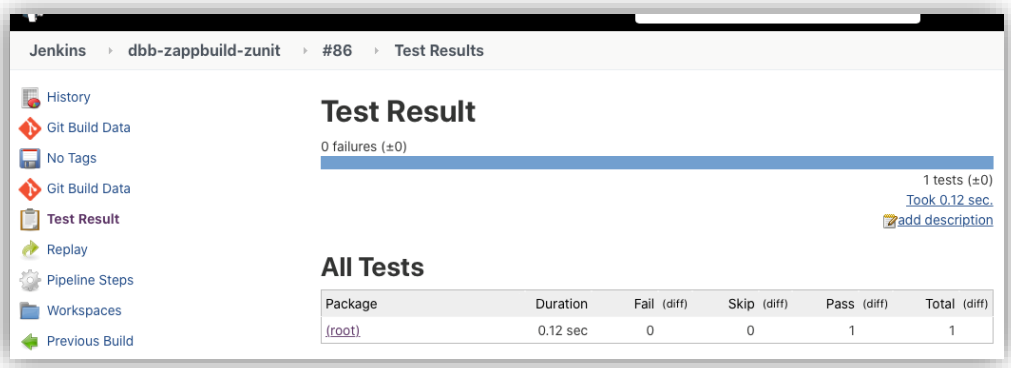
Note: The XSL schemas definitions are provided via the IDz server installation. You find them in the USS installation directory within the *samples/zunit/xsl* subfolder.

Use AZUZ2J30 to converts the zUnit report of the Dynamic Test Runner into a JUnit execution report file. AZUZ2S30 converts the zUnit report into a SonarQube execution report file.



The Jenkins JUnit plugin will visualized the report in the dashboard for the Jenkins pipeline:

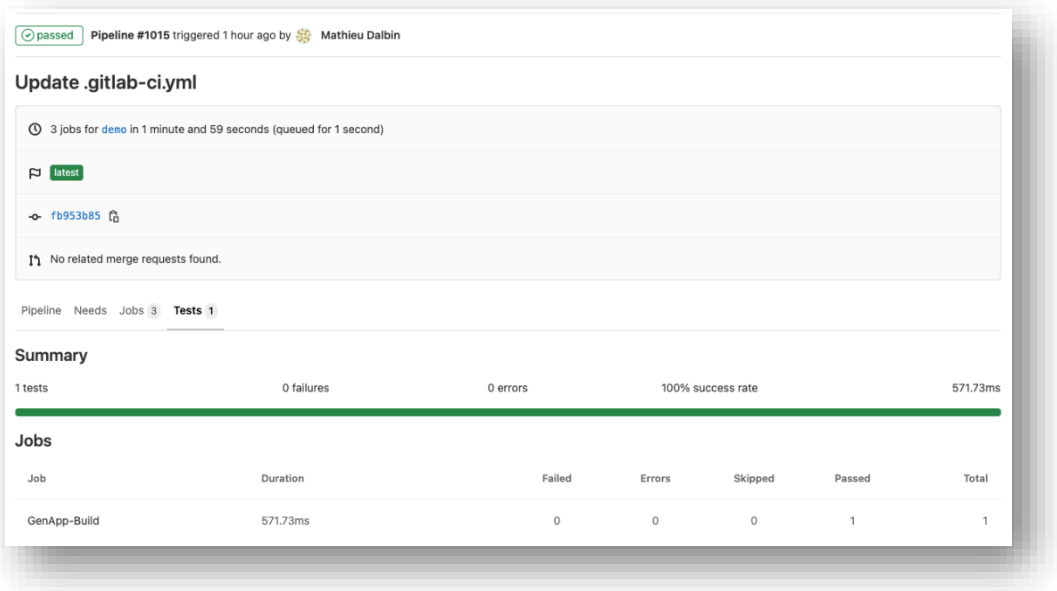




In the GitLab pipeline, the same utility can be invoked to generate JUnit compatible files:

```
for f in `ls $CI_PROJECT_DIR/BUILD-$CI_PIPELINE_ID/build*/*.zunit.report.log`; do Xalan -o
$f.xml $f /var/dbb/extensions/zunit2junit/AZUZ2J30.xsl; done;
```

The test results are automatically presented in under the *Tests* tab for each pipeline execution:



5 Integrating Code Coverage in zUnit testings

IBM Developer for z/OS and IBM z/OS Debugger provide a feature called Code Coverage, which enables the developer to understand which lines of the application code was executed. Code coverage information is very relevant when implementing a unit testing practice.

The Code Coverage feature is usually accessible from a graphical interface for the developer (either IBM Developer or z/OS or Visual Studio Code), but it is also a step of pipelines as well. In context with unit testing, it will document which application code was tested and provides information about the test coverage of an application.

With the Code Coverage Headless Collector daemon, the recoding of test coverage can be integrated into the automated CI/CD pipeline. The documentation on how to setup the headless daemon is available at:

https://www.ibm.com/support/knowledgecenter/en/SSQ2R2_15.0.0/com.ibm.debug.pdt.codecoverage.zpcl.doc/topics/tcchlsdm.html

The Code Coverage Headless collector is available on x86 as well as on z/OS. In this scenario, the Headless Collector on z/OS is configured - close to where the zUnit tests are executed. It is executed as a Started Task (STC) on z/OS, which launches a daemon task on z/OS Unix System Services (USS)[5].

The startup script passes a TCP/IP port to the daemon, where it will listen for incoming connections. Additional parameters can be passed to the daemon, especially the "*-P*" flag is useful to print the command key when a Code Coverage session is initialized.[6]

To collect code coverage information for unit tests, the zAppBuild framework has been enhanced through zAppBuild's the pull request #93[7]. Additional parameters are passed to the build scripts to specify:

- the *hostname* and the *port* on which the Headless Collector listens,
- additional options for the Code Coverage processing. For instance, you can specify the type of report that Code Coverage is able to produce (either PDF, SonarQube compatible format, or both) and the location where to store the reports (please see the -e and -o options in the documentation of the Code Coverage Collector daemon).

The different options for the Headless Collector are described in this section of the knowledge center:

https://www.ibm.com/support/knowledgecenter/en/SSQ2R2_15.0.0/com.ibm.debug.pdt.codecoverage.zpcl.doc/topics/tccstartup.html

[5] Please note that the setup of a started task is custom to this setup and not part of the standard product configuration.

[6] https://www.ibm.com/support/knowledgecenter/en/SSQ2R2_15.0.0/com.ibm.debug.pdt.codecoverage.zpcl.doc/topics/tcchdls.html

[7] https://github.com/IBM/dbb-zappbuild/pull/93

5.1 Setting up the Code Coverage Headless Collector on z/OS

As the daemon is running as a Started Task, it is defined as a procedure named GLCC in the PROCLIB concatenation. In our configuration, the following definition is used, triggering the startup script:

```
//GLCC      PROC
//*
//UMASK    EXEC PGM=BPXBATCH,DYNAMNBR=30,REGION=0M,TIME=1440,
//    PARM='SH umask 000'
//GLCC     EXEC PGM=BPXBATCH,DYNAMNBR=30,REGION=0M,TIME=1440,
//    PARM='SH /u/gitlab/ccstart.sh'
//STDOUT   DD SYSOUT=*
//STDERR   DD SYSOUT=*
//STDENV DD *
_BPXK_AUTOCVT=ON
//*
//        PEND
```

The previous STC definition is shipped as a sample. Please work with your mainframe administration teams to set this up, while it is not part of the standard product customization.

The sample startup script *ccstart.sh* contains the following statements and executes the Headless Collector coming from IBM Debugger for z/OS (which is also available in the IBM Developer for z/OS installation):

```
#!/bin/bash
source .bash_profile

export JAVA_HOME=/usr/lpp/java/J8.0_64/
export IBM_JAVA_ENABLE_ASCII_FILETAG=ON
export IBM_JAVA_OPTIONS=""

echo "### Environment Variables ###"
env

/usr/lpp/debug/v15/headless-code-coverage/bin/ccstart.sh -port=8009 -P
```

This configuration will enable the Code Coverage daemon to listen on port 8009 for incoming connections.

5.2 Enabling Code Coverage in the zUnit Tests

When collecting the Code Coverage information for a program, one important aspect is to include the required options to enable debugging. Depending on the language and the compiler version, these parameters can vary. This page lists the compatible options to be used with IBM z/OS Debugger v15:

https://www.ibm.com/support/knowledgecenter/en/SSQ2R2_15.0.0/com.ibm.debug.pdt.codecoverage.zpcl.doc/topics/compidtcc.html

In our case, we will use a Cobol program compiled with Cobol 6.1 compiler. The options *TEST(EJPD,SOURCE)* will be added to the list of compilation options for the program LGICDB01.cbl, for which we collect Code Coverage information.

In zAppBuild, you can leverage the use of the *--debug* flag which enables the addition of TEST compilation options, as defined by the *cobol_compileDebugParms* for Cobol (also available for other languages).

In our case, we will enable the TEST options only for one specific file. To do that, we will override the compilation options for this file, adding a specific entry in the *file.properties* configuration file:

```
cobol_compileParms=SOURCE,NOOFFSET,APOST,LIST,FLAG(W,E),NOSEQ,NOCOMPILE(E),TRUNC(BIN),OPTIMIZE
,XMLPARSE(COMPAT),XREF,MAP,SIZE(4000K),TEST(EJPD,SOURCE) :: **/cobol/lgicdb01.cbl
```

As the zUnit test is part of the zAppBuild build process, additional properties need to be provided to collect the Code Coverage information. These properties are defined in the *ZunitConfig.properties* file and help to define the Headless Collector's hostname and port, along with additional Code Coverage options.

Alternatively, Code Coverage parameters to the zAppBuild framework can be provided at execution time, taking precedence over the defined properties. Four parameters can be specified:

- -cc, --ccczUnit: Flag to indicate to collect code coverage reports during zUnit step
- -cch, --cccHost: Headless Code Coverage Collector host (if not specified IDz will be used for reporting)
- -ccp, --cccPort: Headless Code Coverage Collector port (if not specified IDz will be used for reporting)
- -cco, --cccOptions: Headless Code Coverage Collector Options

To use the Code Coverage Headless Collector on z/OS, the parameters *-cch* and *-ccp* will be used to point to the z/OS machine where the Collector daemon is running. Specific Code Coverage options can be passed through the *-cco*, for instance to specify the type or reports to generate and the location where the files will be stored.

5.2.1 Running the pipeline in Jenkins

The pipeline definition needs to be updated to reflect the use of these new parameters.

```
stage("Build & UnitTest") {
    sh "/usr/lpp/dbb/v1r0/bin/groovyz ${dbbZappbuildDir}/build.groovy --sourceDir
${WORKSPACE}/${genappDir} --workDir ${workOutputDir} --hlq JENKINS.GENAPP.DEMO --logEncoding
UTF-8 --application genapp --verbose --fullBuild -cc -cch localhost -ccp 8009 -cco
\"e='CCPDF,CCSONARQUBE',o=${workOutputDir}\" "
}
```

Please note the imbricated use of single quotes and double quotes and the escaped double quotes -with backslashes) when specifying the Code Coverage options with the *-cco* parameter: this is necessary to prevent quotes processing which would break the content of the string.

The variable *${workOutputDir}* will be filled in at the execution of the pipeline and is pointing to the work directory for this build.

The JCL generated by the *ZunitConfig.groovy* language script will contain the additional information in the *CEEOPTS* DD card, related to the Code Coverage collection. For instance, the following JCL was generated to enable the Code Coverage collection for the program *LGICDB01.cbl*:

```
//RUNNER EXEC PROC=BZUPPLAY,
// BZUCFG=JENKINS.GENAPP.DEMO.BZU.BZUCFG(LGICDB01),
// BZUCBK=JENKINS.GENAPP.DEMO.TEST.LOAD,
// BZULOD=JENKINS.GENAPP.DEMO.LOAD,
//  PARM=('STOP=E,REPORT=XML')
//REPLAY.BZUPLAY DD DISP=SHR,
// DSN=JENKINS.GENAPP.DEMO.BZU.BZUPLAY(LGICDB01)
//REPLAY.BZURPT DD DISP=SHR,
// DSN=JENKINS.GENAPP.DEMO.BZU.BZURPT(LGICDB01)
//CEEOPTS DD *
TEST(,,,TCPIP&localhost%8009:*)
ENVAR(
"EQA_STARTUP_KEY=CC,LGICDB01,t=LGICDB01,i=LGICDB01,e='CCPDF,CCSONARQUBE'
,o=/var/jenkins/workspace/genapp-demo/BUILD-61")
/*
```

The information provided through the *-cco* parameter (or through the *zunit_CodeCoverageOptions* property) is concatenated to a generated string which specifies the name of the load module to test.

Both PDF and SonarQube formats were specified in this request, so the Code Coverage Collector created the reports in the working directory:

The file *LGICDB01_2021_03_04_170104_0129E.pdf* contains Code Coverage information in a PDF format, while the file *LGICDB01_2021_03_04_170104_0129E.xml* contains SonarQube compatible information.

These files, along with the file *sonar-project.properties*, can be published to Jenkins or to a SonarQube server for further processing.

5.2.2 Running the pipeline in GitLab

The same configuration must be put in place when capturing Code Coverage in a GitLab CI/CD pipeline. As for Jenkins, the GitLab CI/CD *.gitlab-ci.yml* configuration file must be tailored to integrate the additional parameters for Code Coverage in the build phase.

```
GenApp-Build:
    stage: Build
    script:
        - $DBB_HOME/bin/groovyz -Djava.library.path=$DBB_HOME/lib:/usr/lib/java_runtime64
${ZAPPBUILD}/build.groovy --workspace $CI_PROJECT_DIR --workDir $CI_PROJECT_DIR/BUILD-
$CI_PIPELINE_ID --hlq GITLAB.GENAPP.DEMO --logEncoding UTF-8 --application genapp --verbose --
fullBuild  -cc -cch localhost -ccp 8009 -cco "e='CCPDF,CCSONARQUBE',o=${CI_PROJECT_DIR}/BUILD-
${CI_PIPELINE_ID}"
        - cd $CI_PROJECT_DIR/BUILD-$CI_PIPELINE_ID/build*
        - for f in `ls $CI_PROJECT_DIR/BUILD-$CI_PIPELINE_ID/build*/*.zunit.report.log`; do
Xalan -o $f.xml $f /var/dbb/extensions/zunit2junit/AZUZ2J30.xsl; done;
```

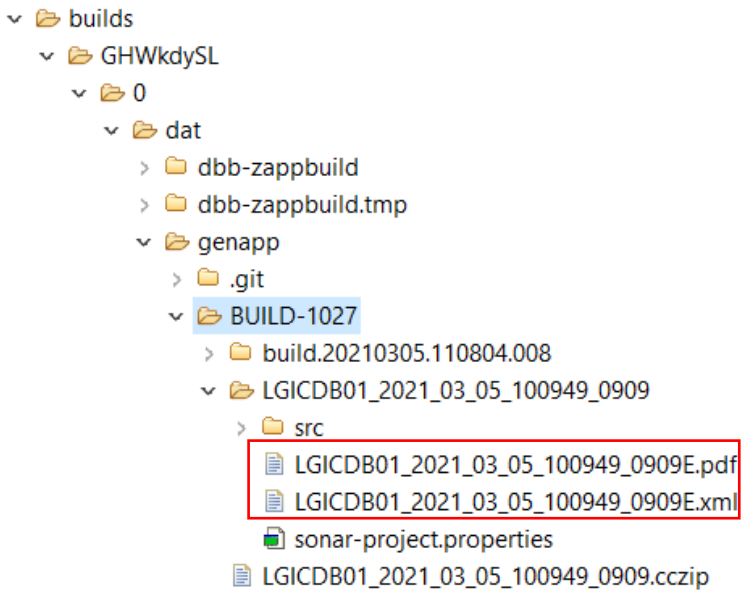
Please note the imbricated use of single quotes and double quotes when specifying the *-cco* parameter.

In this configuration, the output directory will be qualified at the execution of the pipeline and will point to the work directory used for this GitLab project.

The execution of this pipeline will generate a specific JCL, which contains the necessary information to perform the Code Coverage collection (for the program *LGICDB01*):

```
//RUNNER EXEC PROC=BZUPPLAY,
// BZUCFG=GITLAB.GENAPP.DEMO.BZU.BZUCFG(LGICDB01),
// BZUCBK=GITLAB.GENAPP.DEMO.TEST.LOAD,
// BZULOD=GITLAB.GENAPP.DEMO.LOAD,
// PARM=('STOP=E,REPORT=XML')
//REPLAY.BZUPLAY DD DISP=SHR,
// DSN=GITLAB.GENAPP.DEMO.BZU.BZUPLAY(LGICDB01)
//REPLAY.BZURPT DD DISP=SHR,
// DSN=GITLAB.GENAPP.DEMO.BZU.BZURPT(LGICDB01)
//CEEOPTS DD *
TEST(,,,TCPIP&localhost%8009:*)
ENVAR(
"EQA_STARTUP_KEY=CC,LGICDB01,t=LGICDB01,i=LGICDB01,e='CCPDF,CCSONARQUBE'
,o=/u/gitlab/gitlab-runner/zos/builds/GHWkdySL/0/dat/genapp/BUILD-1027")
/*
```

In this configuration, similarly to the Jenkins processing, both PDF file and SonarQubes files are generated in the working directory where the GitLab project was checked out by the pipeline:



Both PDF files and SonarQube files can be uploaded for further processing or for the need of the developer.

6 Summary

IBM Dependency Based Build and IBM Developer for z/OS contribute to the implementation of open and modern CI/CD pipelines and enable the implementation of a shift-left testing practice and culture.

IBM Developer for z/OS release version 14.2.3 and version 15.0 delivered the redesigned z/OS Dynamic Test Runner framework which simplifies the implementation of a unit testing practice for Mainframe development.

While DBB and zAppBuild provide the understanding of the dependencies between the application programs and available test cases, and which artifacts belong to the test configuration it enables to execute the unit tests as part of the pipeline.

The integration of the Code Coverage collection in the pipeline is a major value-added feature for developers who can, at a very early stage, better appreciate the quality of their development, while measuring their level of test coverage.