

Using IBM Dependency Based Build (DBB) with Scripted Deployments

Author:

Nelson Lopez Nelson.lopez1@ibm.com

Reviewers:

Dennis Behm
Jean-Yves Baudy
Mathieu Dalbin
Tim Donnelly
Shabbir Moledina

Abstract

This primer outlines a basic z/OS configuration with DBB/Git and a customer-built Continuous Deployment process using IBM's sample starter scripts.



1	Introduction	3
1.1	Prerequisites	3
2	Overview of a DBB-Based CI/CD Flow	4
3	Package and Publish – Setup and a Sample Run	5
4	Summary	11
	Appendix – Technical references	12
4.1	DBB BuildReport Groovy Implementation	12

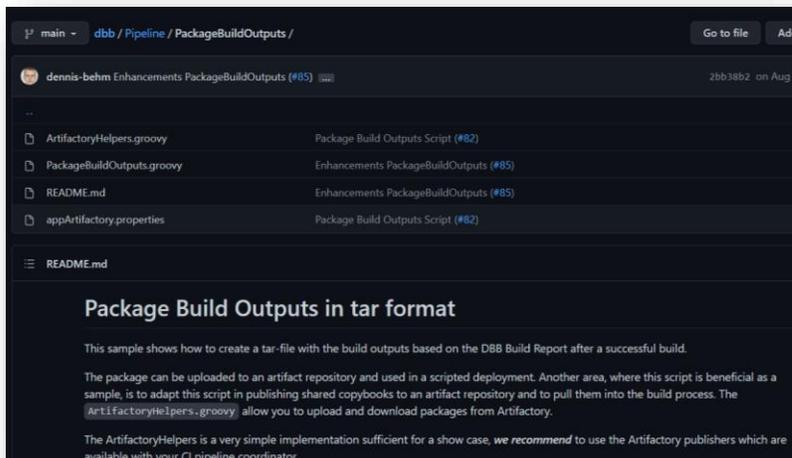
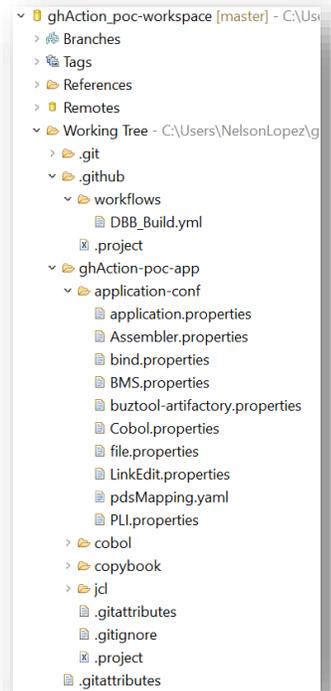
1 Introduction

The purpose of this document is to outline the basic steps to install, configure and test a simple CI/CD pipeline using any Git server and any Orchestrator that runs Dependency Based Build (DBB) builds for z/OS-based applications. The focus will be on how to design and implement a customer-built "scripted" deployment process (CD) using sample starter scripts provided by IBM.

In the example provided below, GitLab will be used as the Repo and CI-Pipeline orchestrator.

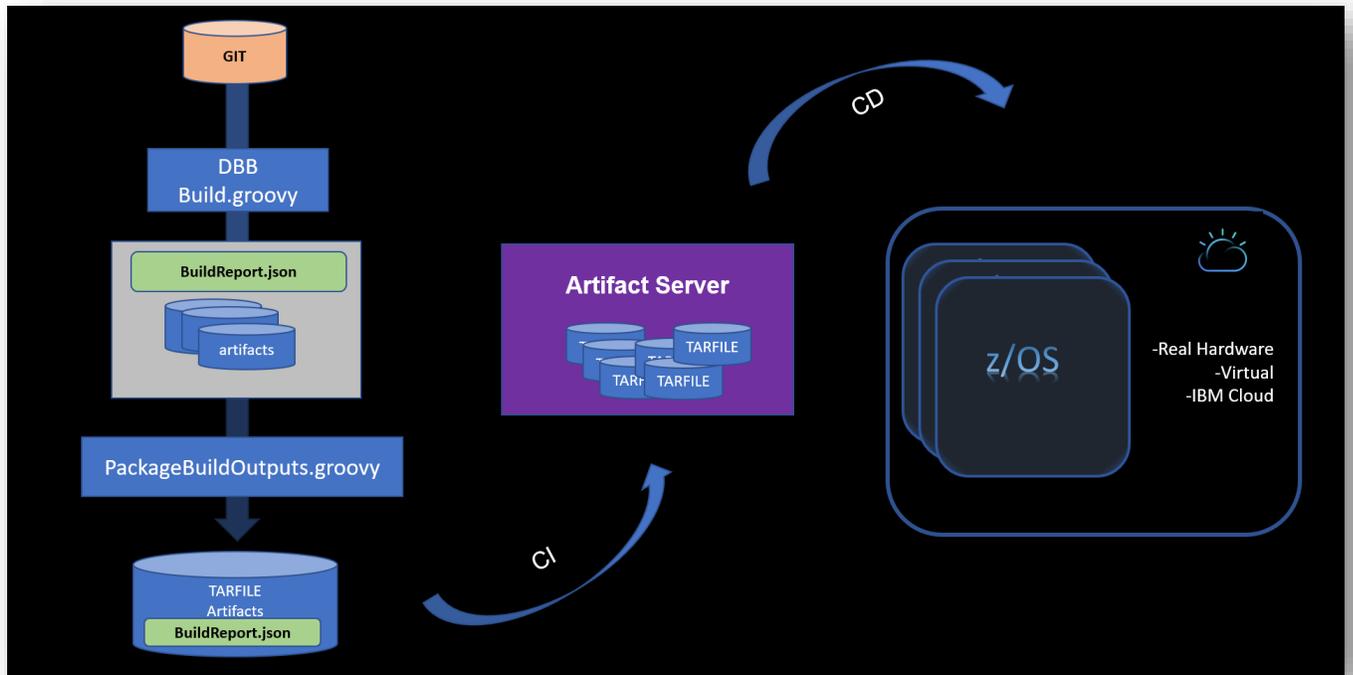
1.1 Prerequisites

- A pipeline Orchestrator like GitLab CI-Pipelines
- A source Repo Server like GitLab Repos structured for DBB-based builds (see diagram on the right). It must have a z/OS ".gitattributes" file, a subfolder for your source code and DBB's "application-conf" files. And any other objects you may need.
- An Artifact Repo Server like Artifactory
- DBB Toolkit, DBB Server and dbb-zappbuild
- Rockets port of Git for z/OS
- An IDE like IBM Developer for z/OS (IDz)
- Access to z/OS with an account that has a standard Unix System Services profile (OMVS). Contact your z/OS Admin for assistance.
- Download/clone, review readme, install and configure the sample 'scripted' Package and Publish scripts from <https://github.com/IBM/dbb/tree/main/Pipeline/PackageBuildOutputs>



2 Overview of a DBB-Based CI/CD Flow

A standard DBB build produces output artifacts like load modules and DBRM's. It also creates a BuildReport.json that lists all the artifacts. This report provides attributes that are used to package the artifacts into a tar file and publish it to an Artifact repo. This document provides details on the continuous Integration (CI) phase. A topic on Continuously Deployment (CD) will be added soon.



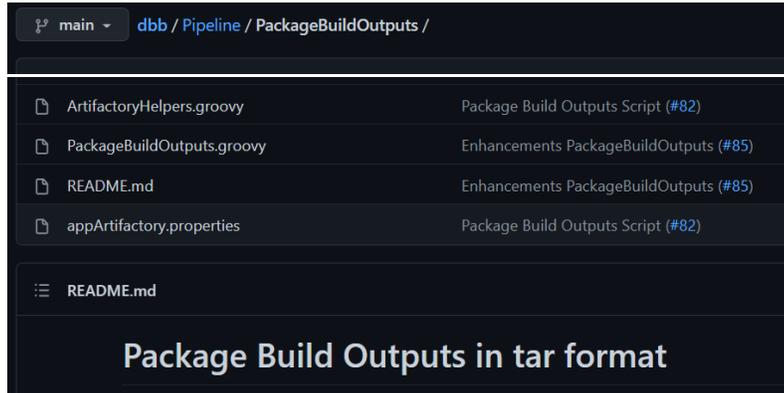
3 Package and Publish – Setup and a Sample Run

Summary of setup steps:

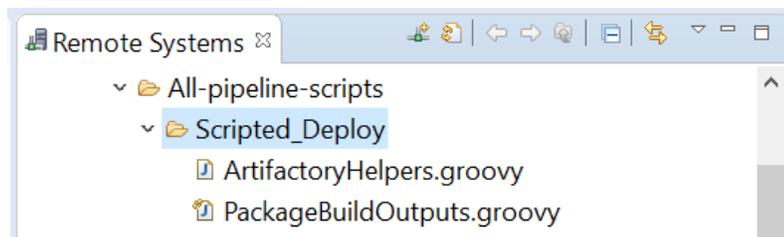
1. Add the scripts to USS (see the Prerequisites section).
2. Add artifactory properties to your configuration
3. Add a property in the application's dbb configuration file ("createBuildOutputSubfolder=false")
4. Add the script step to the CI-pipeline
5. Test to validate your pipeline

1 – Download and Install the scripts

Clone the samples repo¹. The repo has many folders. The PackageBuildOuptuts folder has the scripts needed for this configuration.



Add the 2 groovy scripts to a z/OS Unix home folder as shown.

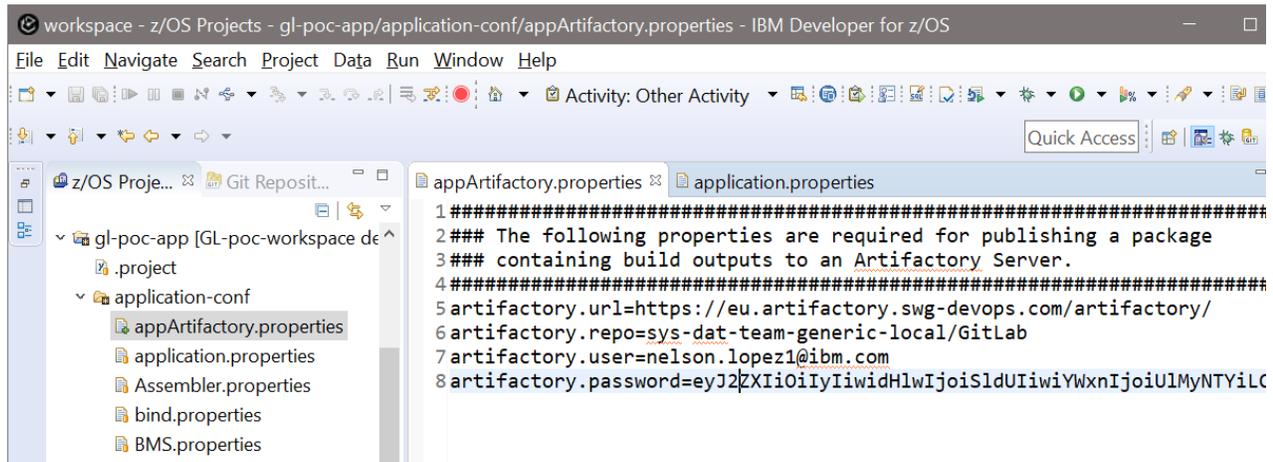


¹ <https://github.com/IBM/dbb>

2 – Add Artifactory Properties file to application-conf

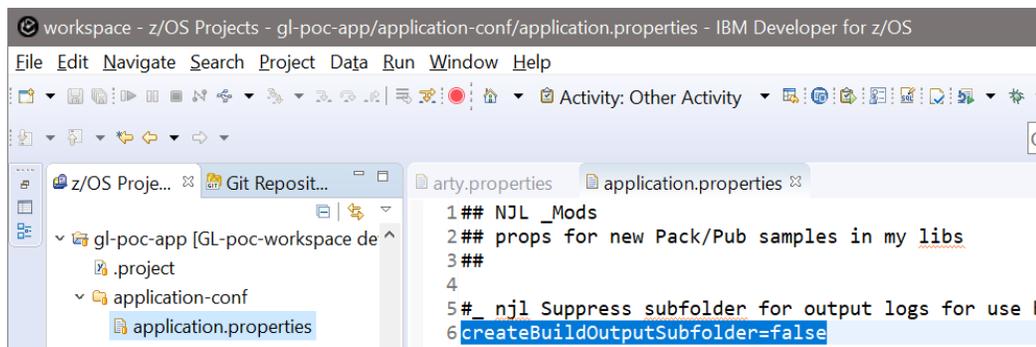
The sample property file is added to the application's application-conf folder as shown.

Fill in the details to reflect your configuration. Note, for POCs and trials, adding the password in a repo may be acceptable. However, in a production environment, consider using your Orchestrator's vault or other secret store to protect the password.



3 - Add a new property in the application's dbb configuration

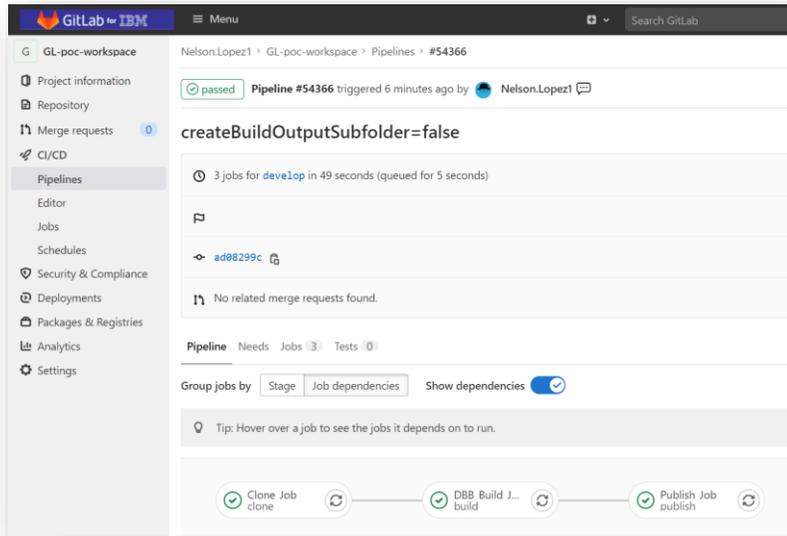
By default, DBB creates a subfolder to store artifacts. Add the property below (createBuildOutputSubfolder=false) to your 'application.properties' file. This causes DBB to place the BuildReport.json file at the workDir root level and not a subfolder.



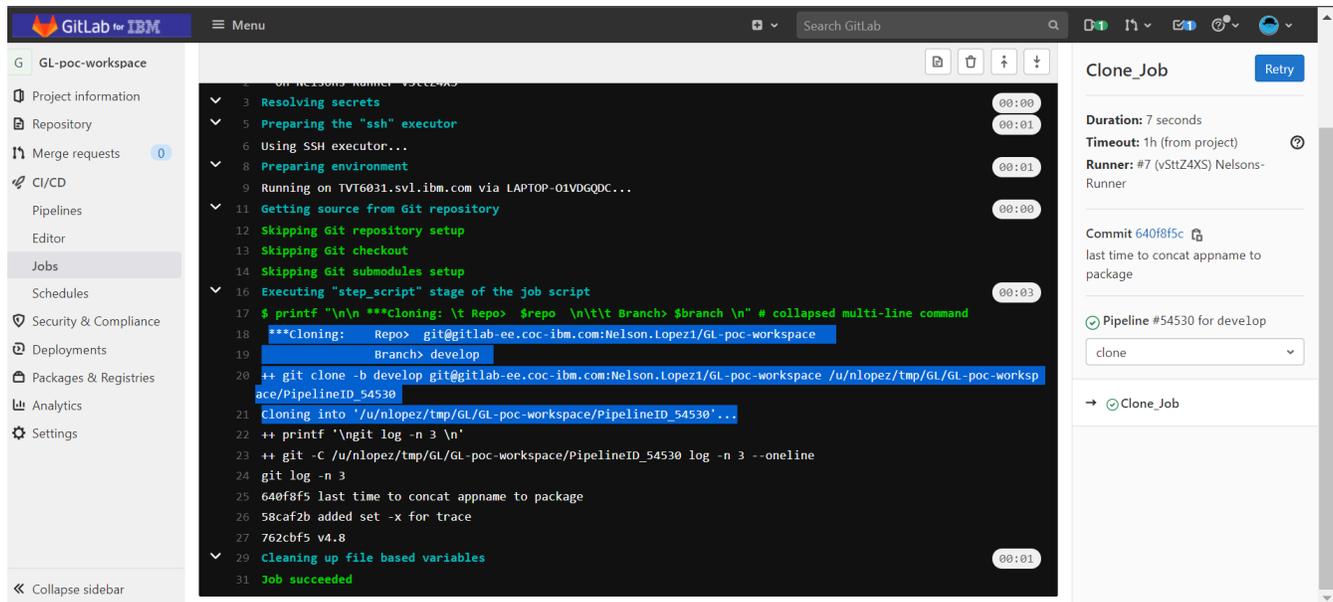
4 - Add the script to the CI-pipeline

Below is a sample Gitlab pipeline. There are 3 basic steps:

1. Clone the application repo
2. DBB build
3. Package and publish



1 - The **clone** creates a workDir folder that is used by all steps. In this case the workDir ends with "PipelineID_5430". 5430 is the jobID provided by the system to help create a unique working folder for each run. Note that the clone script in this step can be replaced by Gitlab's internal Git Clone plugin.



2 – A clean DBB build places artifacts in the workDir root.

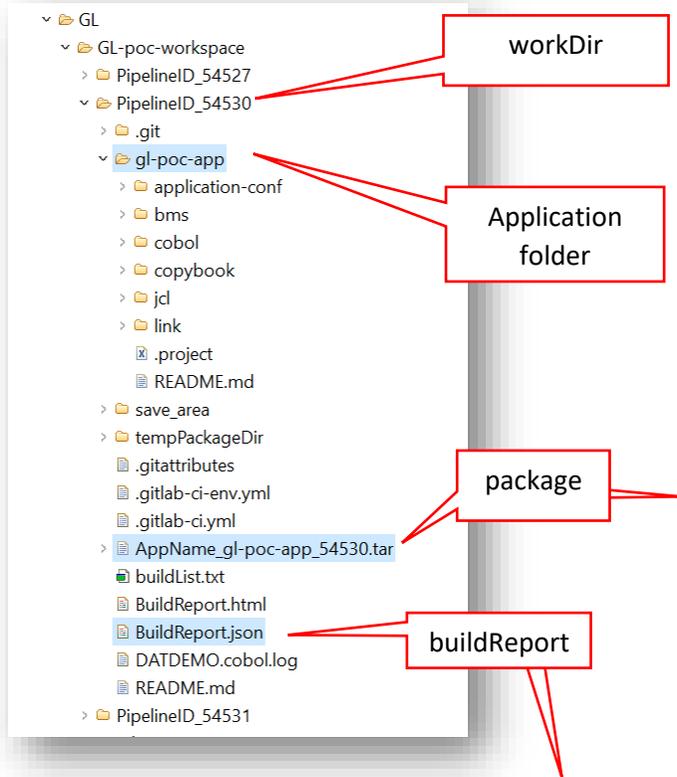
```
16 Executing "step_script" stage of the job script
17 ++ groovy /u/nlopez/dbb-zappbuild/build.groovy -w /u/nlopez/tmp/GL/GL-poc-workspace/PipelineID_54530 -a gl-poc
-app -o /u/nlopez/tmp/GL/GL-poc-workspace/PipelineID_54530 -h DAT.GITLAB.B54530 --impactBuild
18 $ printf "\n\n ***DBB Build on Branch> $branch \n\n" # collapsed multi-line command
19 ***DBB Build on Branch> develop
20 ** Build start at 20211111.080523.005
21 ** Repository client created for https://9.160.16.167:9443/dbb/
22 ** Build output located at /u/nlopez/tmp/GL/GL-poc-workspace/PipelineID_54530
23 ** Build result created for BuildGroup:gl-poc-app-develop BuildLabel:build.20211111.080523.005 at https://9.16
0.16.167:9443/dbb/rest/buildResult/17791
24 ** --impactBuild option selected. Building impacted programs for application gl-poc-app
25 ** Writing build list file to /u/nlopez/tmp/GL/GL-poc-workspace/PipelineID_54530/buildList.txt
26 ** Invoking build scripts according to build order: BMS.groovy,Assembler.groovy,Cobol.groovy,LinkEdit.groovy
27 ** Building files mapped to Cobol.groovy script
28 *** Building file gl-poc-app/cobol/datdemo.cbl
29 *** Building file gl-poc-app/cobol/datdemo.cbl
30 *** Building file gl-poc-app/cobol/datdemo.cbl
31 ** Writing build report data to /u/nlopez/tmp/GL/GL-poc-workspace/PipelineID_54530/BuildReport.json
32 ** Writing build report to /u/nlopez/tmp/GL/GL-poc-workspace/PipelineID_54530/BuildReport.html
33 ** Updating build result BuildGroup:gl-poc-app-develop BuildLabel:build.20211111.080523.005 at https://9.160.1
6.167:9443/dbb/rest/buildResult/17791
34 ** Build ended at Thu Nov 11 08:05:55 EST 2021
35 ** Build State : CLEAN
36 ** Total files processed : 1
37 ** Total build time : 31.167 seconds
38 ** Build finished
```

3 - The PackageBuildOutput script reads the DBB json file, tars the DBB artifacts that have a supported deployType and publishes them to Artifactory using the properties in application-conf. In addition, the DBB BuildReport.json file is included to support the CD phase.

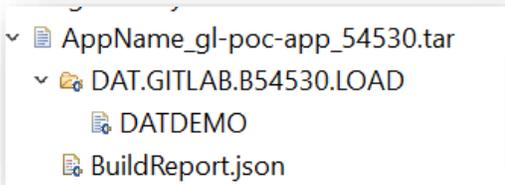
Supported deployTypes are any default or user supplied attributes assigned to an output artifact by a dbb-zappbuild groovy language script.

```
18 ***Scripted Package_Publish. Package> AppName_gl-poc-app_54530.tar Version>
19 Property File> /u/nlopez/tmp/GL/GL-poc-workspace/PipelineID_54530/gl-poc-app/application-conf/appArtifa
ctory.properties
20 ++ groovy /u/nlopez/All-pipeline-scripts/Scripted_Deploy/PackageBuildOutputs.groovy --publish --verbose -w /u/
nlopez/tmp/GL/GL-poc-workspace/PipelineID_54530 -t AppName_gl-poc-app_54530.tar -prop /u/nlopez/tmp/GL/GL-poc-wo
rkspace/PipelineID_54530/gl-poc-app/application-conf/appArtifactory.properties -v GL-poc-workspace
21 ** PackageBuildOutputs start at 20211111.080601.006
22 ** Properties at startup:
23 workDir -> /u/nlopez/tmp/GL/GL-poc-workspace/PipelineID_54530
24 startTime -> 20211111.080601.006
25 publish -> true
26 versionName -> GL-poc-workspace
27 verbose -> true
```

At the end of the job, the workDir will have the cloned repo, the DBB BuildReport.json and the tar file created by the package script.

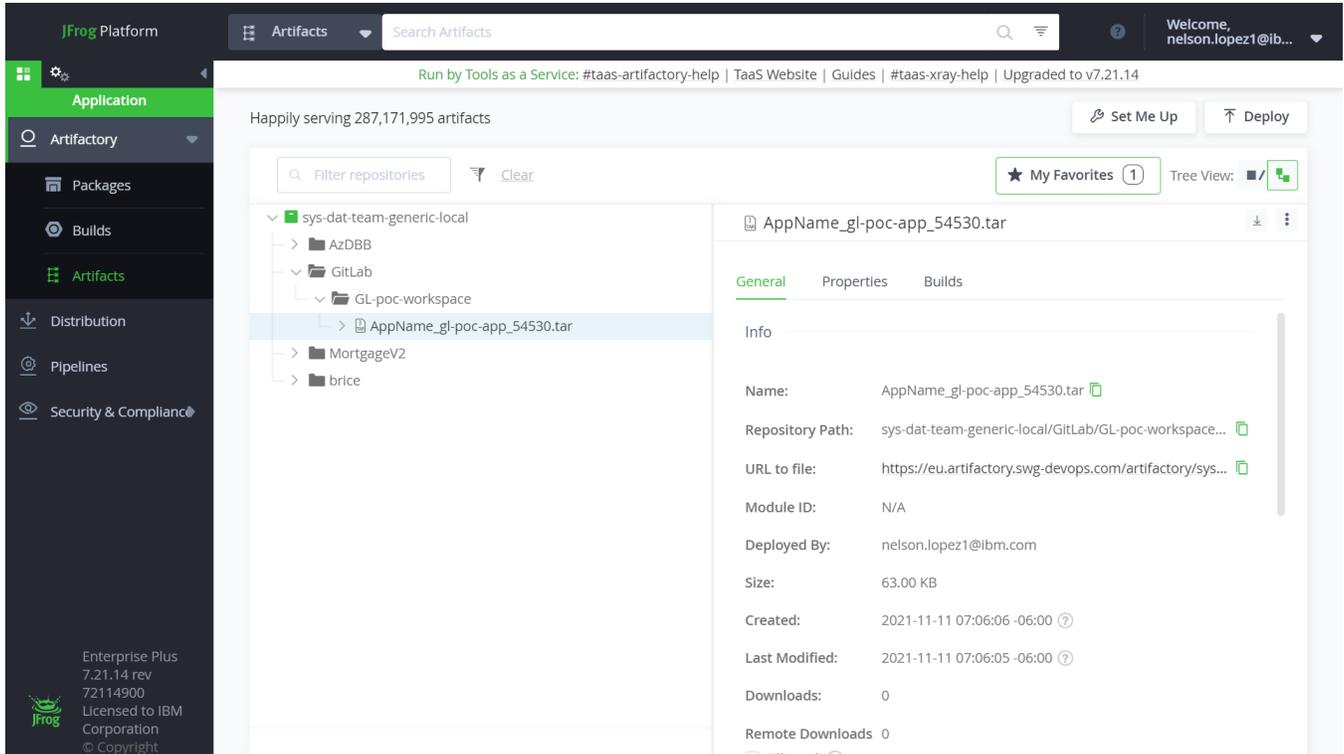


The sample package below includes all DBB output artifacts that have a valid deployType. In this example, it includes one Cobol load module (DATDEMO). It also includes the DBB BuildReport.json file for use in a subsequent deployment phase.



```
88=  {
89     "mode": "TEXT",
90     "destination": "\\u\\nlopez\\tmp\\GL\\GL-poc-workspace\\PipelineID_54530\\DATDEMO.cobol.log",
91     "id": "COPY_TO_HFS_1",
92     "source": "SYSPRINT",
93     "type": "COPY_TO_HFS"
94 },
95=  {
96     "outputs": [{
97         "dataset": "DAT.GITLAB.B54530.LOAD(DATDEMO)",
98         "deployType": "LOAD"
99     }],
100     "rc": 0,
101     "file": "gl-poc-app\\cobol\\datdemo.cbl",
102     "options": "MAP,RENT,COMPAT(PM5)",
103     "id": "EXECUTE_2",
104     "type": "EXECUTE",
105     "logs": ["\\u\\nlopez\\tmp\\GL\\GL-poc-workspace\\PipelineID_54530\\DATDEMO.cobol.log"],
106     "command": "IEWBLINK"
107 },
```

The package is published and ready for deployment to any z/OS environment. In this sample, the repo's project name "GL-poc-workspace" is used as the upper folder. The package tar file has the application name 'gl-poc-app' and the pipeline's buildID for traceability.



4 Summary

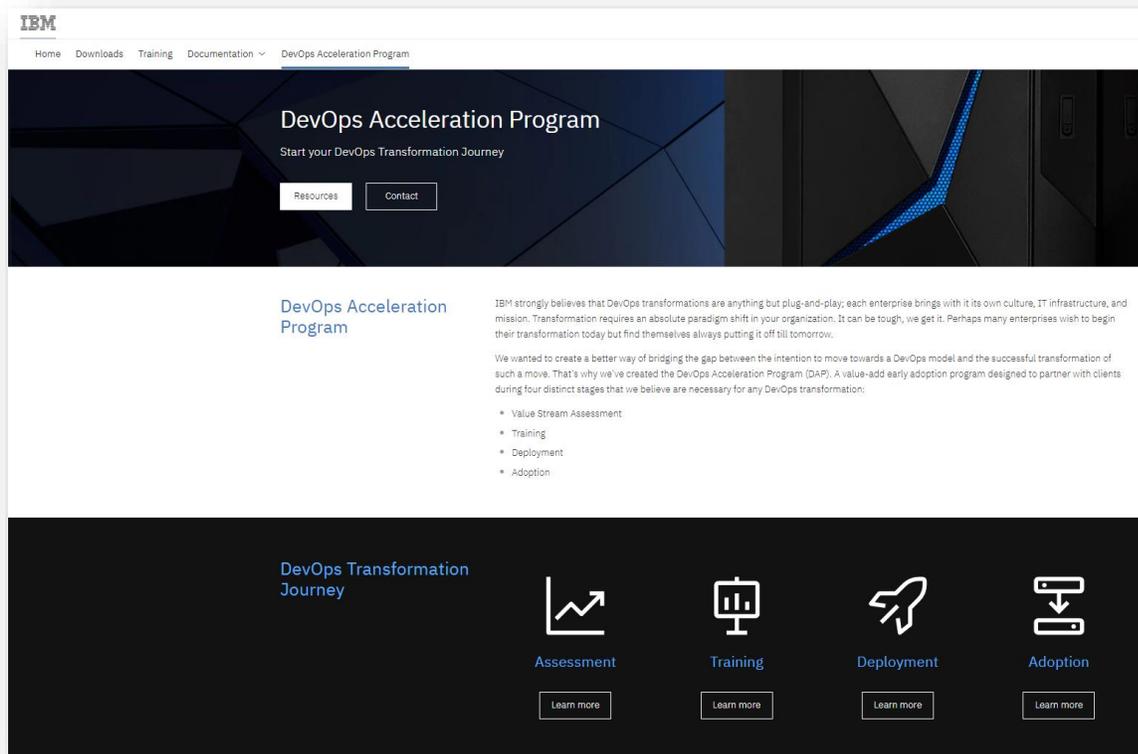
As shown in the sample run above, adding a scripted package and publish to your ci-pipeline is a one-step job. Additional steps should be considered like adding a CD step, automated tests and cleanup of logs. CD will be added in a future update of this document.

The examples provided are meant to get you started. Over time it can be extended to support production-like workloads, automated quality gates and more robust logging and audit trails.

For more reference material please visit IBM's "DevOps Acceleration Program" website

https://ibm.github.io/mainframe-downloads/DevOps_Acceleration_Program/devops-acceleration-program.html

Reach out to your account representative for support and services.



Appendix – Technical references

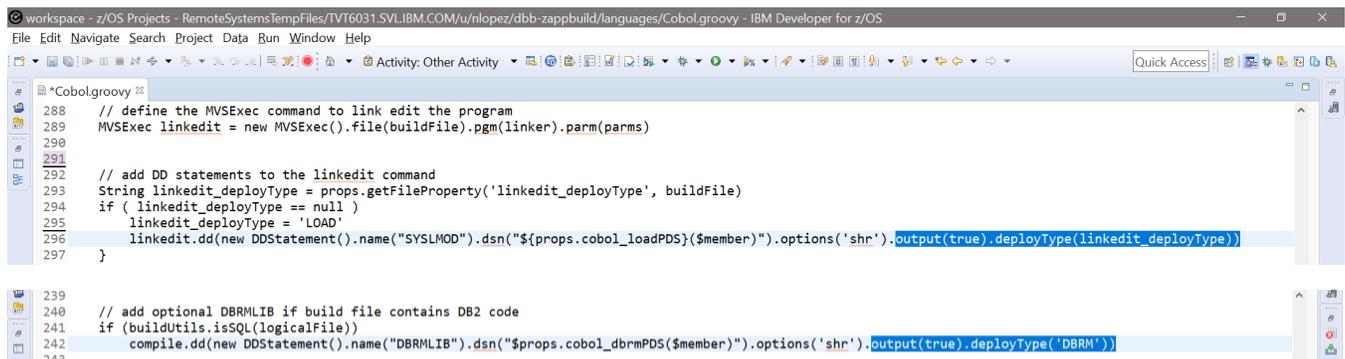
4.1 DBB BuildReport Groovy Implementation

The code snippets below are from IBM's sample dbb-zappbuild² framework. It shows how a Cobol program's output load module (artifact) is tagged with a **deployType** of "LOAD". Based on the nature of the source code, zAppBuild is capable of setting different deployTypes to allow different processing during the deployment of the outputs artifacts.³ It also shows how the BMS.groovy script assigns the "MAPLOAD" deployType to the map load module. In addition to the deployType, output file DD allocations must set the "output" flag to "true" to include the artifact in the BuildReport.

DeployTypes and file names are used to drive the actual deployment logic during the CD phase.

You can add or change the deployType to fit your design.

Cobol.groovy Snippet – LOAD & DBRM DeployType



```
workspace - z/OS Projects - RemoteSystemsTempFiles\TVT6031.SVLIB.COM\u/nlopez/dbb-zappbuild\languages\Cobol.groovy - IBM Developer for z/OS
File Edit Navigate Search Project Data Run Window Help
Activity: Other Activity
Quick Access

* Cobol.groovy
288 // define the MVSEExec command to link edit the program
289 MVSEExec linkedit = new MVSEExec().file(buildFile).pgm(linker).parm(params)
290
291
292 // add DD statements to the linkedit command
293 String linkedit_deployType = props.getFileProperty('linkedit_deployType', buildFile)
294 if ( linkedit_deployType == null )
295     linkedit_deployType = 'LOAD'
296 linkedit.dd(new DDStatement().name("SYSLMOD").dsn("${props.cobol_loadPDS}${member}").options('shr').output(true).deployType(linkedit_deployType))
297 }

239
240 // add optional DBRMLIB if build file contains DB2 code
241 if (buildUtils.isSQL(logicalFile))
242     compile.dd(new DDStatement().name("DBRMLIB").dsn("${props.cobol_dbrmPDS}${member}").options('shr').output(true).deployType('DBRM'))
243 }
```

² See <https://github.com/IBM/dbb-zappbuild>

³ See <https://github.com/IBM/dbb-zappbuild/blob/main/samples/application-conf/Cobol.properties#L62-L72>

BMS.groovy Snippet – MAPLOAD DeployType

```
uild_wazi06.md  Cobol.groovy  BMS.groovy  ❏  
  
/*  
 * createLinkEditCommand - creates a MVSEExec xommand for link editing the bms object module produced by the compile  
 */  
def createLinkEditCommand(String buildFile, String member, File logFile) {  
    String parameters = props.getFileProperty('bms_linkEditParms', buildFile)  
  
    // define the MVSEExec command to link edit the program  
    MVSEExec linkedit = new MVSEExec().file(buildFile).pgm(props.bms_linkEditor).parm(parameters)  
  
    // add DD statements to the linkedit command  
    linkedit.dd(new DDStatement().name("SYSLIN").dsn("&&TEMPOBJ").options("shr"))  
    linkedit.dd(new DDStatement().name("SYSLMOD").dsn("${props.bms_loadPDS}${member}").options('shr').output(true).deployType('MAPLOAD'))  
    linkedit.dd(new DDStatement().name("SYSPRINT").options(props.bms_tempOptions))  
    linkedit.dd(new DDStatement().name("SYSUT1").options(props.bms_tempOptions))  
}
```

For IMS applications, the "IMSLOAD" deployType is assigned.

Below is a sample Build Report. Notice the Output element with the deployType attribute. In this case, it's a Cobol load module.

```
    {  
      "mode": "TEXT",  
      "destination": "\\u\nlopez\GITLAB -Workspace\AzDBB_1786\.\build.20211023.074424.044\DATDEMO.c",  
      "id": "COPY_TO_HFS_3",  
      "source": "SYSPRINT",  
      "type": "COPY_TO_HFS"  
    },  
    {  
      "outputs": [{  
        "dataset": "NLOPEZ.GITLAB.LOAD(DATDEMO)",  
        "deployType": "LOAD"  
      }],  
      "rc": 0,  
      "file": "GITLAB -main-app\cobol\datdemo.cbl",  
      "options": "MAP,RENT,COMPAT(PM5)",  
      "id": "EXECUTE_4",  
      "type": "EXECUTE",  
      "logs": ["\\u\nlopez\GITLAB -Workspace\AzDBB_1786\.\build.20211023.074424.044\DATDEMO.cobol.1",  
        "command": "IEWBLINK"  
    },  
    {  
      "mode": "TEXT",  
      "destination": "\\u\nlopez\GITLAB -Workspace\AzDBB_1786\.\build.20211023.074424.044\DATDEMO.c",  
      "id": "COPY_TO_HFS_4",  
      "source": "SYSPRINT",  
      "type": "COPY_TO_HFS"  
    },  
    {  
      "id": "DBB_BuildResultProperties",  
      "type": "PROPERTIES",  
      "properties": {  
        "impactBuild": "true",  
        "filesProcessed": "2",  
        ":giturl:GITLAB -Mortgage-SA\GITLAB -main-app": "git@ssh.dev.GITLAB .com:v3\GITLAB -Repo-DBB\  
    }  
  }  
}
```

For more information on the BuildReport, see:

<https://www.ibm.com/docs/en/adfz/dbb/1.1.0?topic=scripts-how-create-access-dbb-build-report>

The screenshot shows the IBM Documentation page for 'Outputs attribute'. The page title is 'Outputs attribute'. The main text explains that the outputs attribute contains the target data set when the CopyToPDS command has been flagged as an output for the build report. To set it, add an output(true) attribute to the CopyToPDS command. A note states that the output(true) option is required to generate build report attributes to the CopyToPDSRecord. If output(true) is missing from the command, the other build report attributes are not generated. Below the text is a code block with the following content:

```
// Flag the target data set for inclusion in the build report
new CopyToPDS().file(new File("${properties.sourceDir}/${file}").dataset(cobolPDS).member(member).output(true).key(file).execute()
```

The page also includes a section for 'DeployType attribute' with a similar explanation and a code block:

```
// Provide a deploy type for the target dataset that can be used in post build tools like UrbanCode Deploy
new CopyToPDS().file(new File("${properties.sourceDir}/${file}").dataset(cobolPDS).member(member).output(true).key(file).deployType("JCI
```

For more detailed information, refer to the DBB JavaDoc's BuildReport Packages and classes.

Ref: https://www.ibm.com/docs/api/v1/content/SS6T76_1.1.0/javadoc/index.html

The screenshot shows the IBM JavaDoc page for the class `com.ibm.dbb.build.report.records.DefaultRecordFactory`. The page is titled 'Overview' and includes navigation links for 'PACKAGE', 'CLASS', 'USE TREE', 'DEPRECATED', 'INDEX', and 'HELP'. The class is part of the package `com.ibm.dbb.build.report.records`. The class description states that it extends `AbstractRecordFactory` and implements `IRecordFactory`. The class is used to create default records. The page also includes a 'Field Summary' section with the following table:

Modifier and Type	Field and Description
static java.lang.String	TYPE_BUILD_MODE
static java.lang.String	TYPE_BUILD_RESULT
static java.lang.String	TYPE_COPY_TO_HFS
static java.lang.String	TYPE_COPY_TO_PDS
static java.lang.String	TYPE_CREATE_PDS
static java.lang.String	TYPE_DEPENDENCY_SET
static java.lang.String	TYPE_EXECUTE
static java.lang.String	TYPE_PROPERTIES
static java.lang.String	TYPE_REFERENCE
static java.lang.String	TYPE_VERSION