Smart Modernization on z/OS

Using WCA4z AI & DevOps to Extend COBOL CICS DB2 with Java Only Where It Makes Sense

Hands-on lab guide

Code and templates are shareable

Joseph Taffe-Atkins, IBM
Steve Foley, IBM

IBM

Hands on Lab includes:

1. IBM watsonx Code Assistant for Z Understand
2. IBM watsonx Code Assistant for Z Explanation
3. IBM watsonx Code Assistant for Z Refactor
4. IBM watsonx Code Assistant for Z Transform (using live WCA4z Service)
5. IBM DevOps solutions CI/CD Pipeline for Java running CICS
   (Git on z/OS, Gradle on z/OS and x86, Git Lab CI and IBM DevOps Deploy aka Urban Code Deploy)
6. IBM DevOps solutions CI/CD Pipeline for COBOL CICS DB2 VSAM Application CICS-Java
   (Git on z/OS, IBM Dependency Based Build, Git Lab CI and IBM DevOps Deploy aka Urban Code Deploy)

Collated and prepared by:

- Joseph Taffe-Atkins
- Steve Foley

Reviewed and Amended By:

- Roberto Calderon
- Max Weiss
- Sharad Ramsamooj

Table of Contents

Notices and disclaimers

U.S. Government Users Restricted Rights — use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM.

This document is current as of the initial date of publication and may be changed by IBM at any time. Not all offerings are available in every country in which IBM operates.

Information in these presentations (including information relating to products that have not yet been announced by IBM) has been reviewed for accuracy as of the date of initial publication and could include unintentional technical or typographical errors. IBM shall have no responsibility to update this information.

This document is distributed "as is" without any warranty, either express or implied. In no event, shall IBM be liable for any damage arising from the use of this information, including but not limited to, loss of data, business interruption, loss of profit or loss of opportunity.
IBM products and services are warranted per the terms and conditions of the agreements under which they are provided. The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

IBM products are manufactured from new parts or new and used parts.
In some cases, a product may not be new and may have been previously installed. Regardless, our warranty terms apply."

Any statements regarding IBM's future direction, intent or product plans are subject to change or withdrawal without notice.

Performance data contained herein was generally obtained in a controlled, isolated environments. Customer examples are presented as illustrations of how those customers have used IBM products and the results they may have achieved. Actual performance, cost, savings or other results in other operating environments may vary.

References in this document to IBM products, programs, or services does not imply that IBM intends to make such products, programs or services available in all countries in which IBM operates or does business.

Workshops, sessions and associated materials may have been prepared by independent session speakers, and do not necessarily reflect the views of IBM. All materials and discussions are provided for informational purposes only, and are neither intended to, nor shall constitute legal or other guidance or advice to any individual participant or their specific situation.

Notices and disclaimers (Continued)

It is the customer's responsibility to ensure its own compliance with legal requirements and to obtain advice of competent legal counsel as to the identification and interpretation of any relevant laws and regulatory requirements that may affect the customer's business and any actions the customer may need to take to comply with such laws. IBM does not provide legal advice or represent or warrant that its services or products will ensure that the customer follows any law.

Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products. IBM does not warrant the quality of any third-party products, or the ability of any such third-party products to interoperate with IBM's products. **IBM expressly disclaims all warranties, expressed or implied, including but not limited to, the implied warranties of merchantability and fitness for a purpose.**

The provision of the information contained herein is not intended to, and does not, grant any right or license under any IBM patents, copyrights, trademarks or other intellectual property right.

Learn more →

Purpose of the document

This document is designed to guide you through the capabilities of IBM watsonx Code Assistant for Z, specifically focusing on its ability to execute the Understand, Refactor, and Transform phases. Using a real-world COBOL CICS-to-Java scenario, you will experience how the Watson AI Code Transformation Model can help modernize applications where transitioning from COBOL to Java makes sense within a "best fit language" approach.

By following the step-by-step instructions in this guide, you will gain hands-on experience with IBM watsonx Code Assistant for Z, helping you build confidence in using it for modernization initiatives—especially during an adoption project. Depending on your specific environment, adjustments to the execution sequence may be necessary.

Who is this guide for?
IBM watsonx Code Assistant for Z is designed to support various roles, including:

- Architects – to plan and oversee modernization strategies.
- Business analysts – to understand and define refactoring opportunities.
- Developers – to implement and optimize code transformation.

What to Expect
This scenario demonstrates how Java and COBOL developers can collaborate to refactor and implement a feature originally written in COBOL but better suited for Java. The IBM watsonx Code Assistant for Z Refactoring Assistant helps developers identify the right components to modularize and convert into reusable services, enabling a more efficient, scalable, and modernized application architecture.

Whether you are new to IBM watsonx Code Assistant for Z or looking to integrate it into your projects, this guide will provide you with the foundational knowledge needed to leverage AI-powered code transformation with confidence.

The hands-on flow for this lab and the client storyline

wca4z / GenappExtended-Java / Issues / #1

Add a new rule to the POSTCODE IF / ELSE or EVALUATE validation

Edit

Open Issue created 4 months ago by wca4z

Business users have stated that the POSTCODE logic needs an additional validation rule for 'EU' in the function that is within the COBOL program, LGACUS01.

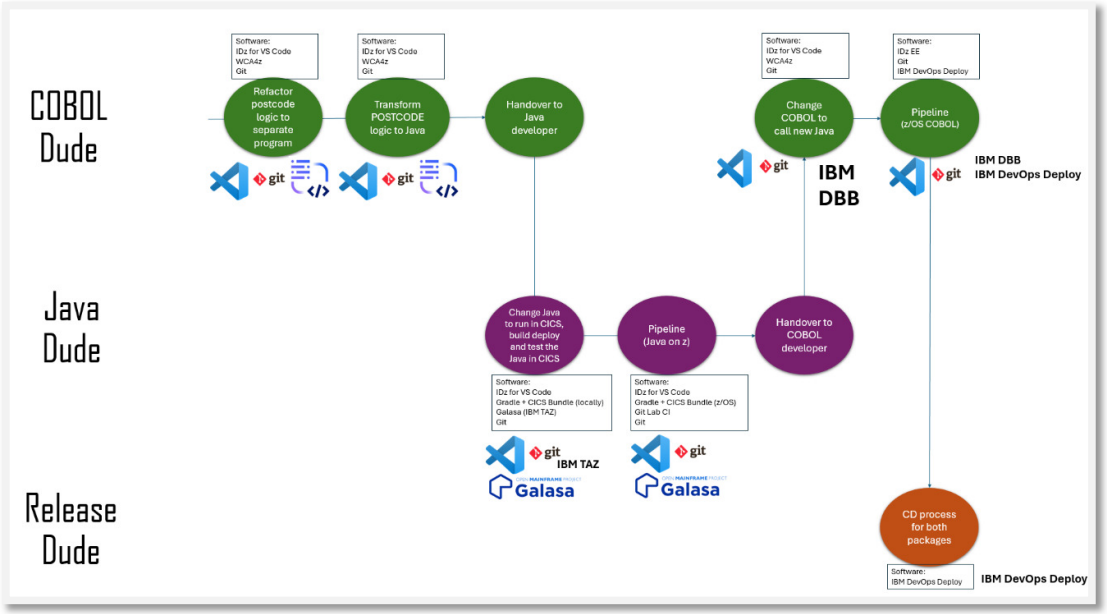
The technical architect stated the POSTCODE logic is <u>better suited in Java</u> therefore we will separate this function from LGACUS01 into its own Java module and call it from COBOL CICS using CICS APIs.

0 0

Create merge request

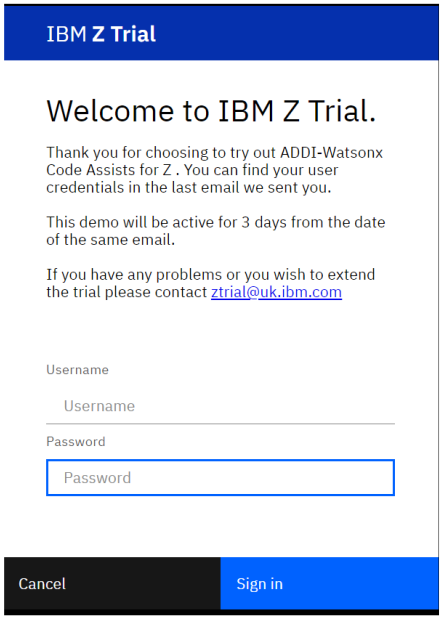
Customer Fact 1	**Customer Fact 2**	**Customer Fact 3**	**Customer Fact 4**
They want to make changes to application functions in the most <u>suitable</u> language on Z and not just COBOL.	They determined that the POSTCODE logic, originally written in COBOL, is better <u>suited</u> to Java and they have available zIIP capacity.	Some business logic, better <u>suited</u> for Java, is already implemented in COBOL. They plan to migrate this logic to Java as changes to functions are required.	COBOL and Java developers are paired up to ensure successful migration to Java before adding any new functionality in Java.

<u>Demo flow</u>

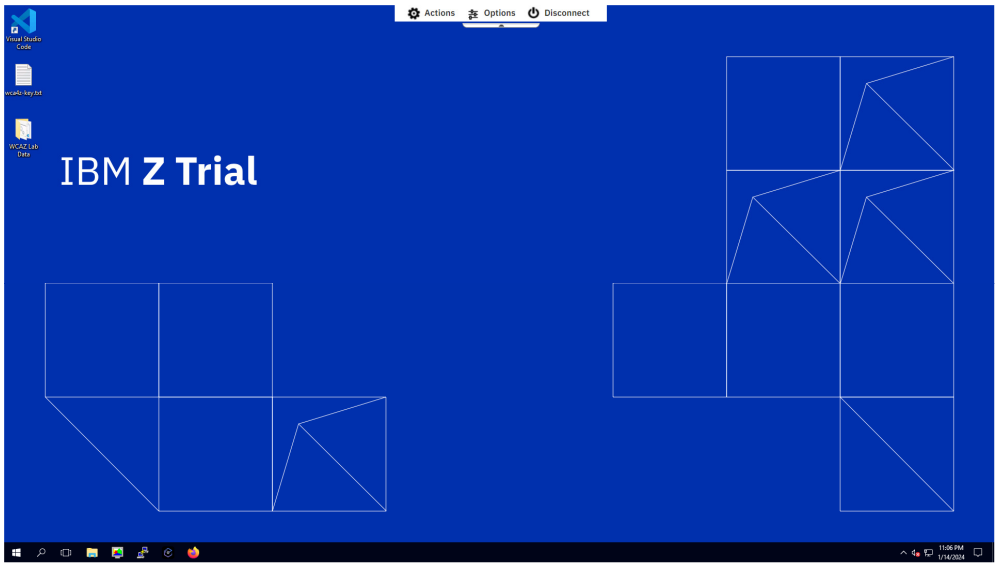


How to login and initialize the lab

1. Get your access for workshop link provided prior to the workshop.

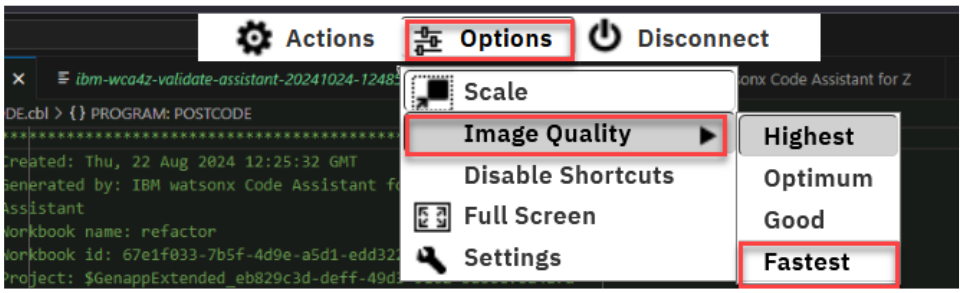


2. Login with your credentials to get into IBM Z TRIAL.

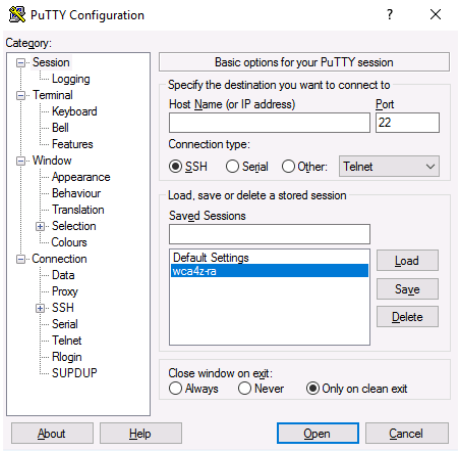


3. On RDP browser, Open Putty from taskbar.

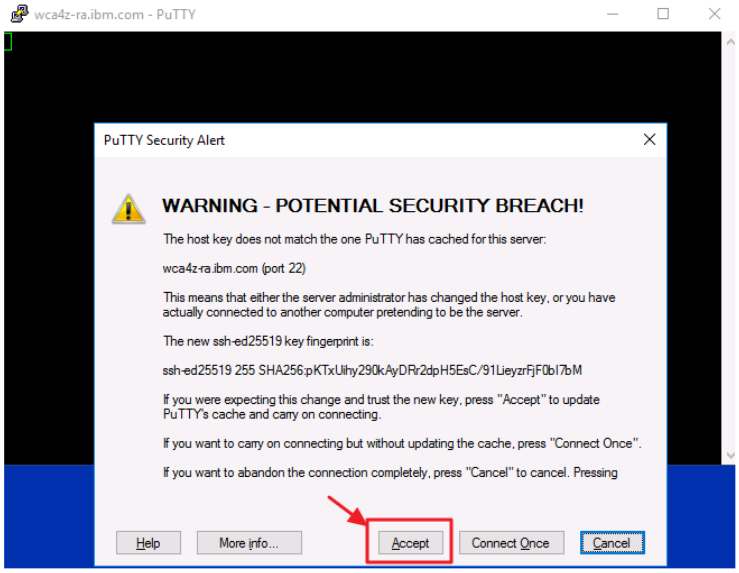
4. Adjust your session to the optimal performance by selecting this at the top middle of your screen. ***Please note, if you screen freezes up during your session, please refresh the browser screen! Apologies for the inconvenience caused.***



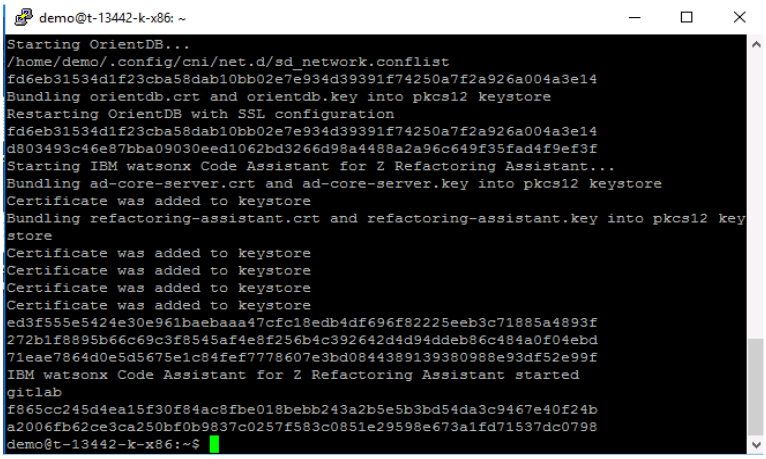
5. select wca4z-ra and click on open.



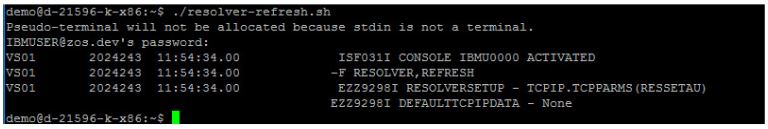
6. After clicking on Open in previous step following warning will pop-up. Select “Accept”. This will be displayed only first time.



7. Auto scripts will run on Putty to start the Refactoring Assistant.
   Wait until message “IBM Watson code assistant for Z Refactoring Assistant started” is displayed on putty. **Leave putty running. If prompted for ID and password, its IBMUSER and sys1**

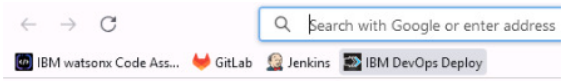


Now enter ./resolver-refresh.sh after the command above has completed.

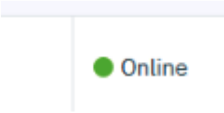


Leave putty running.

8. Open Firefox from the desktop and navigate to IBM DevOps Deploy and login with **admin/admin**



9. Navigate to “Resources” and then “Agents” and confirm that you see both agents online

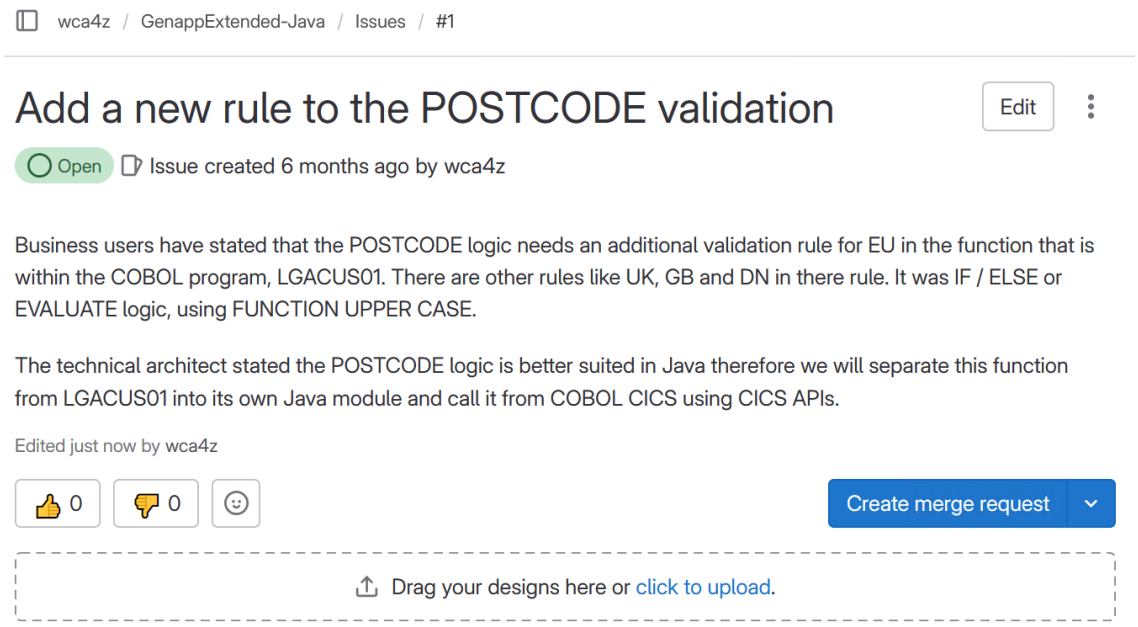


Let’s view the Git Lab ticket where you need to make the code change to a mainframe existing transaction

1. Open Firefox and type in this URL to access the Git Lab ticket: http://wca4z-gitlab.ibm.com:10880/wca4z/genappextended-java/-/issues/1 which is showing the work you will undertake but you being the developer has decided to do this in Java in instead of COBOL.

   You will be prompted for ID and password which is

   ID: wca4z-git
   Password : P@assw0rd

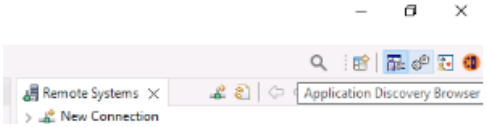


How to execute the Understand phase using IBM Developer for z Systems

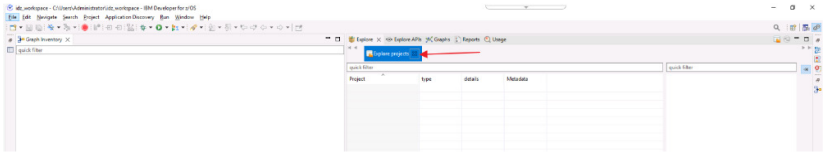
1. Open IBM IDz on RDP browser from taskbar. Please allow some time for this to load (3 minutes).

When prompted for workspace name in IDz, select the default one.

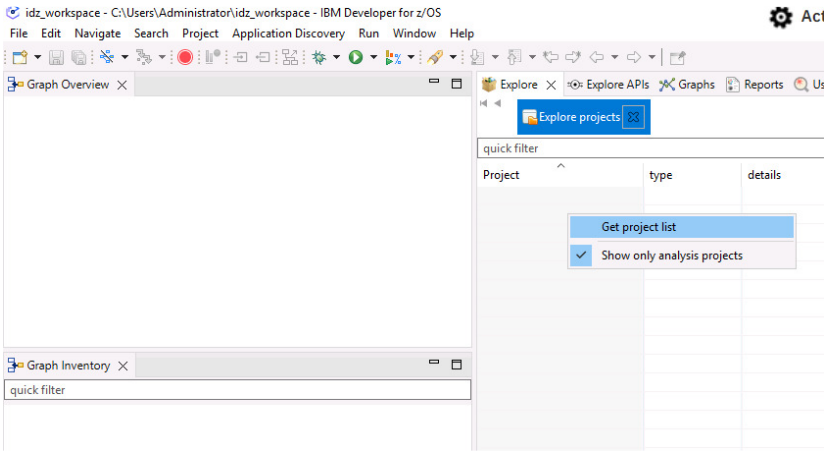
2. If not already open, can you open the “Application Discovery Browser” eclipse perspective from the top right of the eclipse view



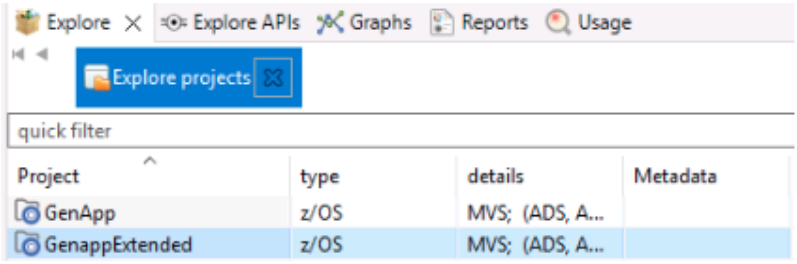
3. In Aqua Eclipse, “**Explore projects**” tab will be opened by defaut.



4. Right click in the blank space below “Explore projects” and Select option “Get project list”.



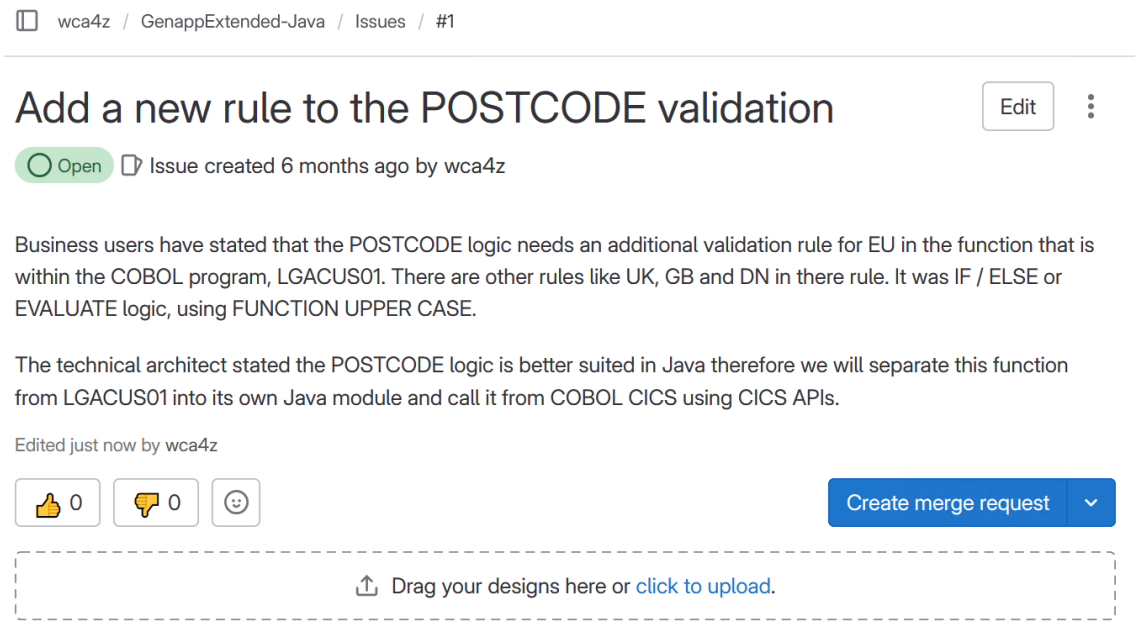
5. All projects will be listed. Single click “GenApp-Extended1”.



6. (If NOT done in STEP 1 above) Open Firefox and type in this URL to access the Git Lab ticket: http://wca4z-gitlab.ibm.com:10880/wca4z/genappextended-java/-/issues/1 which is showing the work you will undertake but you being the developer has decided to do this in Java in instead of COBOL.

   You will be prompted for ID and password which is

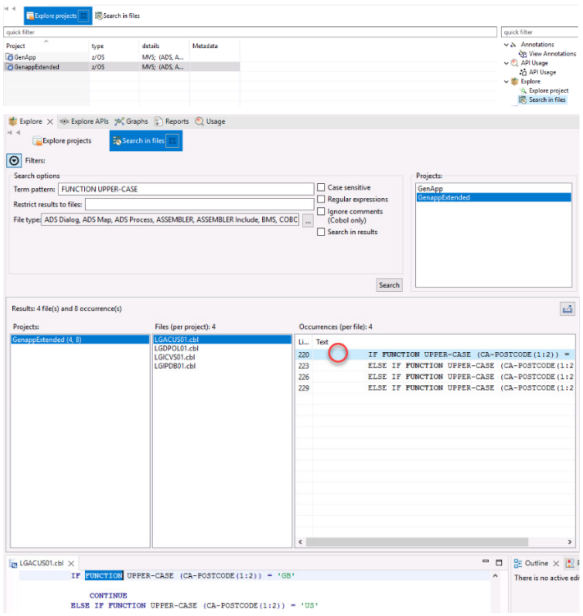
   ID: wca4z-git
   Password : P@assw0rd



7. The clues given about where the code change needs to occur are the following:
    a. The **“POSTCODE”** validation routine needs changing
    b. Assuming there is some **IF / ELSE** or **EVALUATE** logic in the code
    c. Can you think of other things in the ticket to help you locate the exact program where the change needs to happen?
    d. **FUNCTION UPPER-CASE** is potentially code used in the POSTCODE rules

    Using the WCA4z Understand capability, you will try to locate the program that needs to be changed in the GenappExtended application.

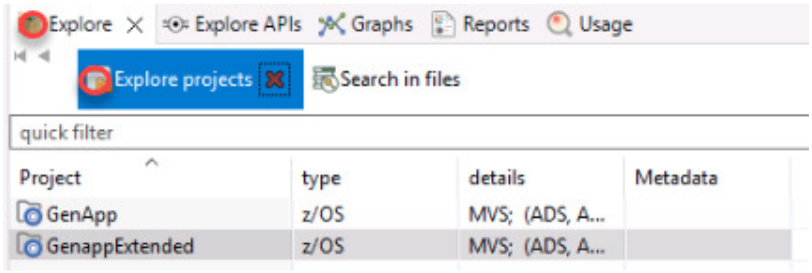
8. Return back to ADDI in IBM Developer for z/OS and double click in “Search in files” under “Explore projects” for the GenappExtended project.



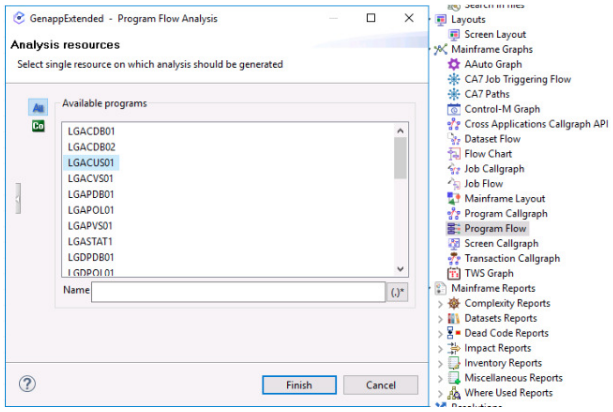
9. Optional – click on the 3 dots next to ‘File Types’ and uncheck everything except for COBOL.
10. From the COBOL syntax above in **bold** from **(a)-(d)**, you will try to locate the program that has the POSTCODE validation rule. You will be putting the COBOL syntax to help you locate the program in the “Term pattern” input field and select “Search” each time until you locate the program.
11. When you get to searching the COBOL syntax in **(d)** you should see that **LGACUS01** is clearly where the POSTCODE Validation is. Double click on the text next to line 220 (see illustration above) to be shown the text below.

There are other ways to explore and get to the program using the Understand capability. This was one of many ways. Feel free to explore after the lab.

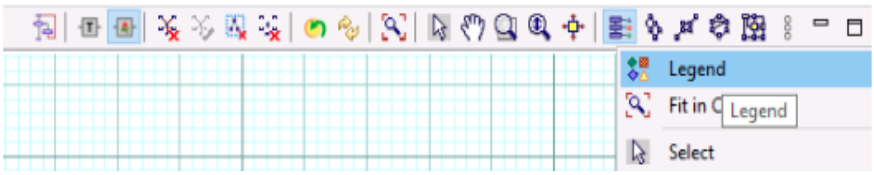
12. Now, let's look at this program in more detail. Single click the "Explore" tab then single click "Explore projects", single click project 'GenApp-Extended1' and new tab with all options will appear at the right side.



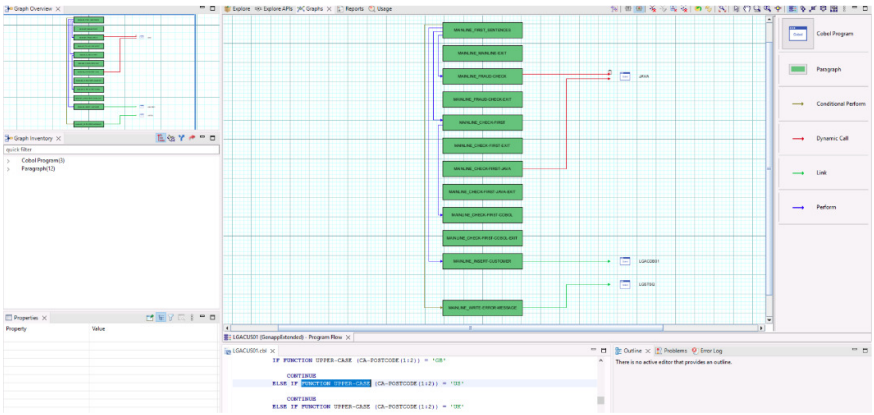
13. On the right side, **double click** to open option "Program Flow" and select **LGACUS01** and "finish".



14. Click on the 3 dots, then open the Legend to understand the diagram



15. Analyse the Program Flow chart



Information: This application and lab use a simple framework for invoking Java programs from COBOL. Appendix A provides an overview of COBOL and Java inter-operation in CICS, and Appendix B describes the simple framework used in this lab.

16. Right click on the CHECK-FIRST-COBOL paragraph and select “view analysis source” to double confirm the logic is correct.

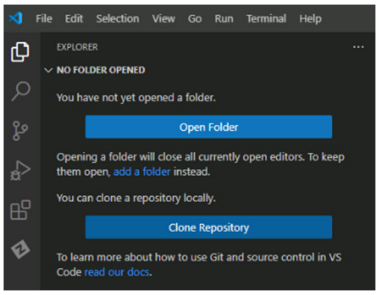
In this **Understand** phase,

1. Here is the logic that needs to be changed in program LGACUS01.
2. The Java developer has identified that they would rather want to implement this feature in Java instead of COBOL.
3. They will look to Refactor this code to its own COBOL service to then transform to the logic to Java.
4. Especially as the program is already calling Java CICS from the COBOL CICS program to do some fraud checking.

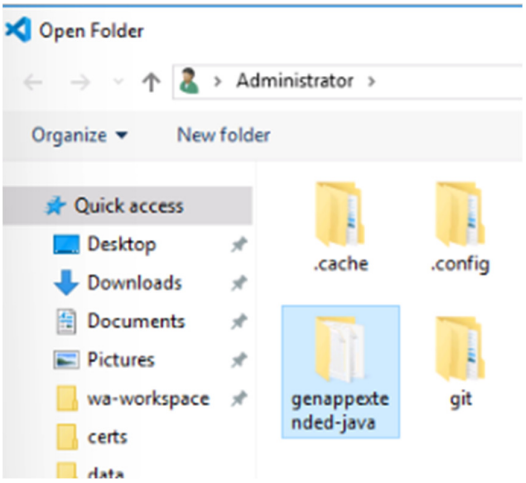
We will use this knowledge in next steps to Refactor and Transform.

How to execute the Refactor phase using VS Code

1. Open VS Code, then click on “Open Folder”

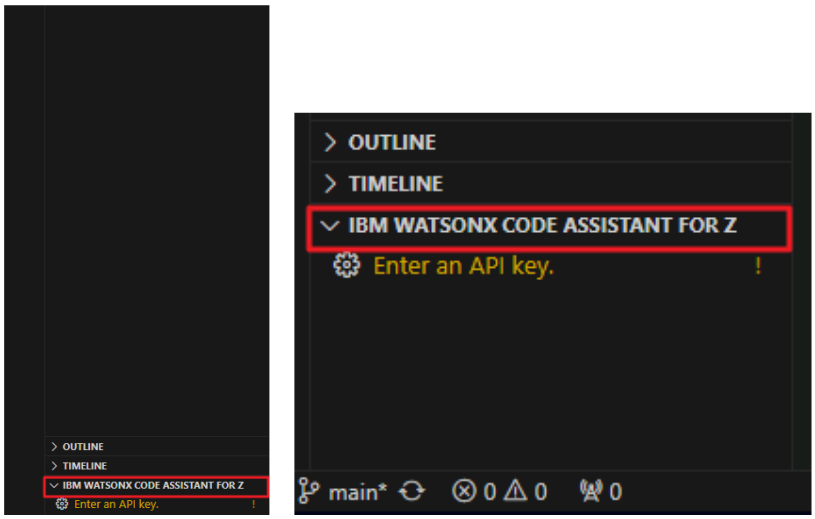


2. Then select the folder “genappextended-java” from C:\Users\Administrator. This folder is an already cloned a Git repository where you will be working from.

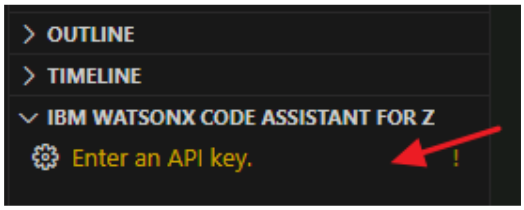


3. Then click select “Select Folder” (give it time to load).

1. On the EXPLORER menu, expand the “IBM WATSONX CODE ASSISTANT FOR Z” at the left corner of the visual studio.

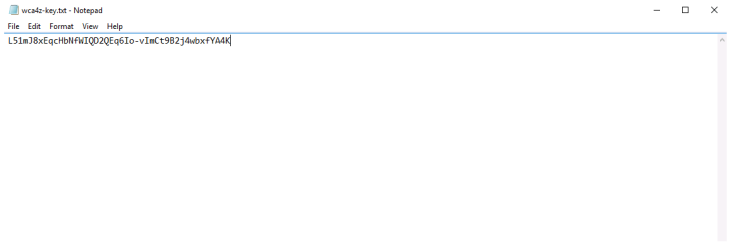


2. You can see warning “Enter an API key” in yellow colour.

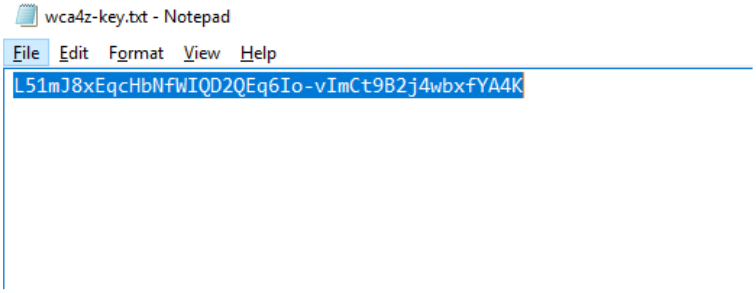


3. To get the API key, minimize all windows and go to desktop.

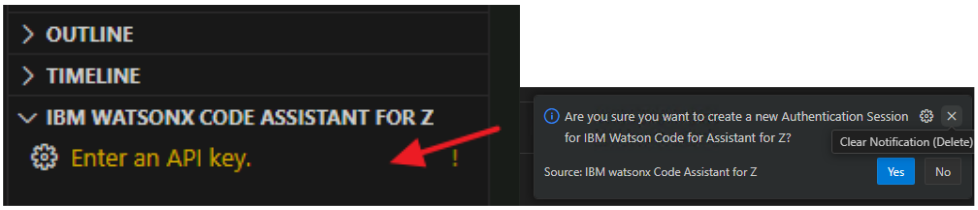
4. Click on notepad named “wca4z-key.txt” to open it.



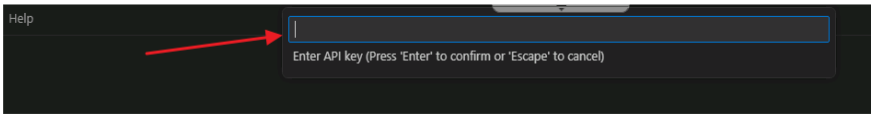
5. Copy API key from the notepad and go back to VS code.

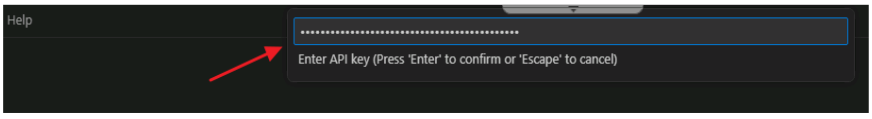


6. Click on warning to get an option “Enter an API key” besides the warning. Click on this option. You may see this warning, if you do, select yes.

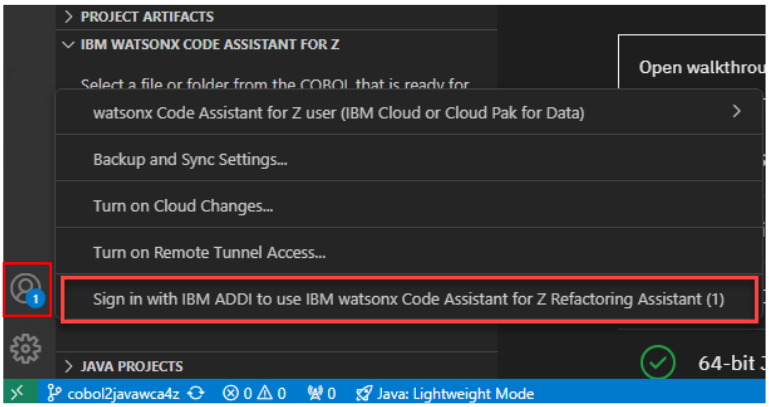


7. At the top middle an input bar will open. Give API key just copied from notepad and press enter.

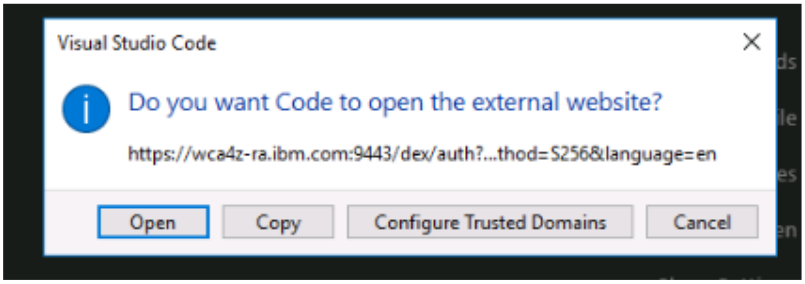




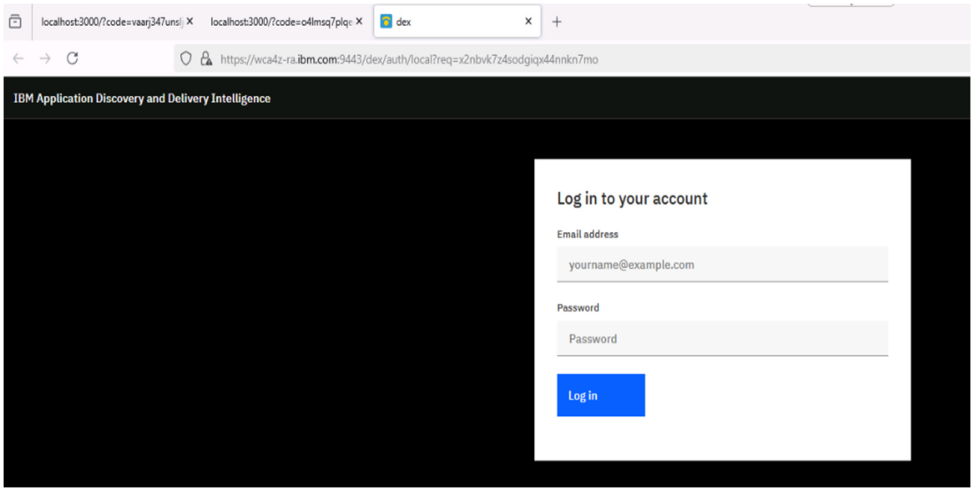
4. Now, go to accounts, and select sign in “IBM Watsonx Code Assistant for Z Refactor Assistant”



5. You will see this prompt. Select ‘Open’ and then a browser will pop to enter your credentials to log into Refactor Assistant.

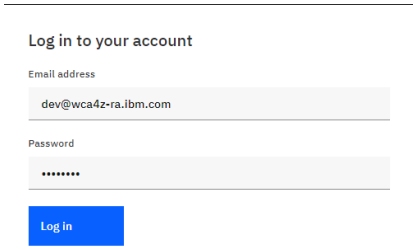


6. Login using the following credentials.

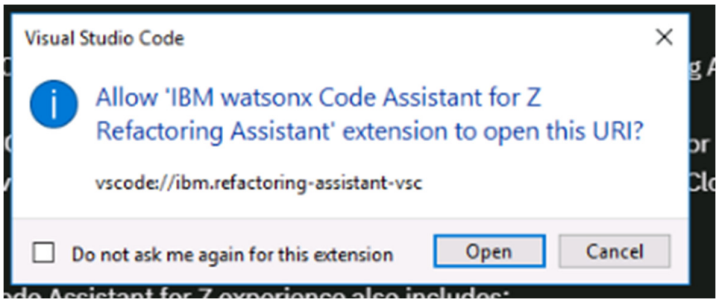


ID: dev@wca4z-ra.ibm.com

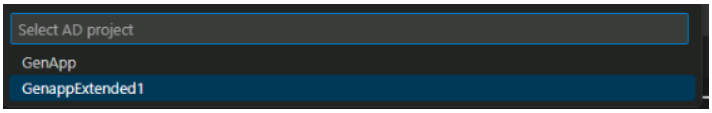
Password: password



7. As soon as you select enter, navigate back to VS Code (the VS Code logo will be flashing on the taskbar). Then select “Open”.

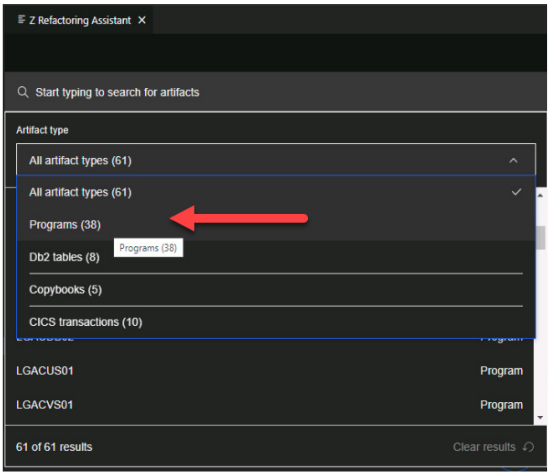


8. If prompted, select “GenappExtended1” project.

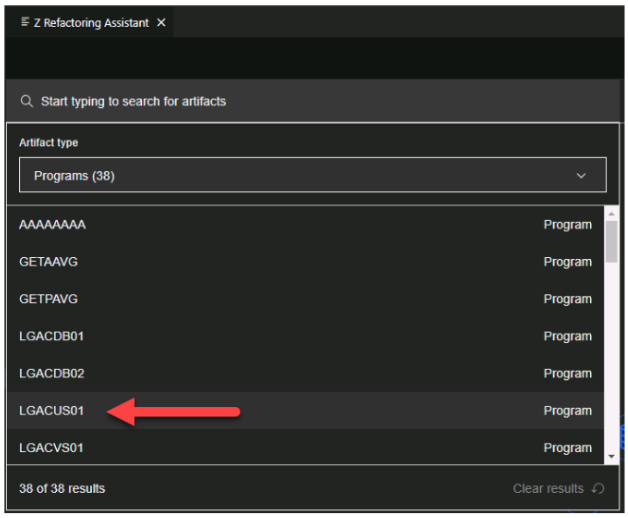


9. Single click on “**Start typing to search for artifacts**”

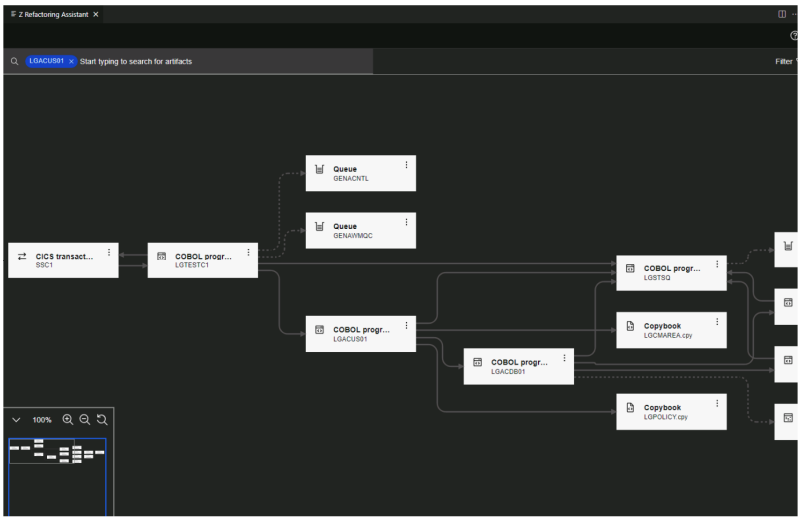
10. Then Select “**All artifact types**” and then select "**Programs**” programs.



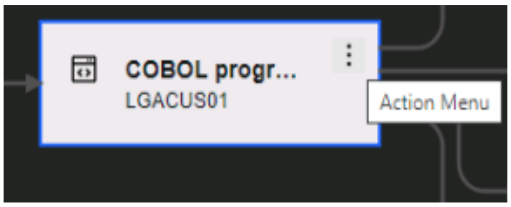
11. From the dropdown list of Programs, click on **LGACUS01** and press enter.



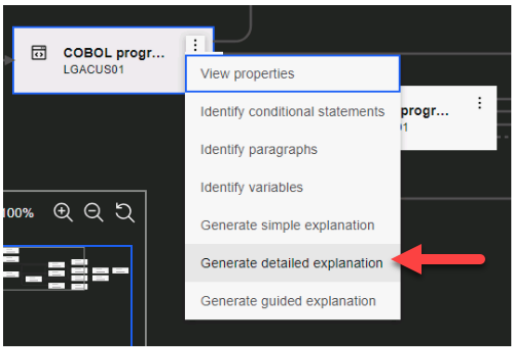
12. The graph for **LGACUS01** Program will open. Take time to analyse it.



13. In the Understand phase, we checked the program flow for **LGACUS01** programs and we checked the IF/ELSE conditions for the **POSTCODE** in the code.
    Click on the 3 dots/ellipses besides the **LGACUS01** to get below options.



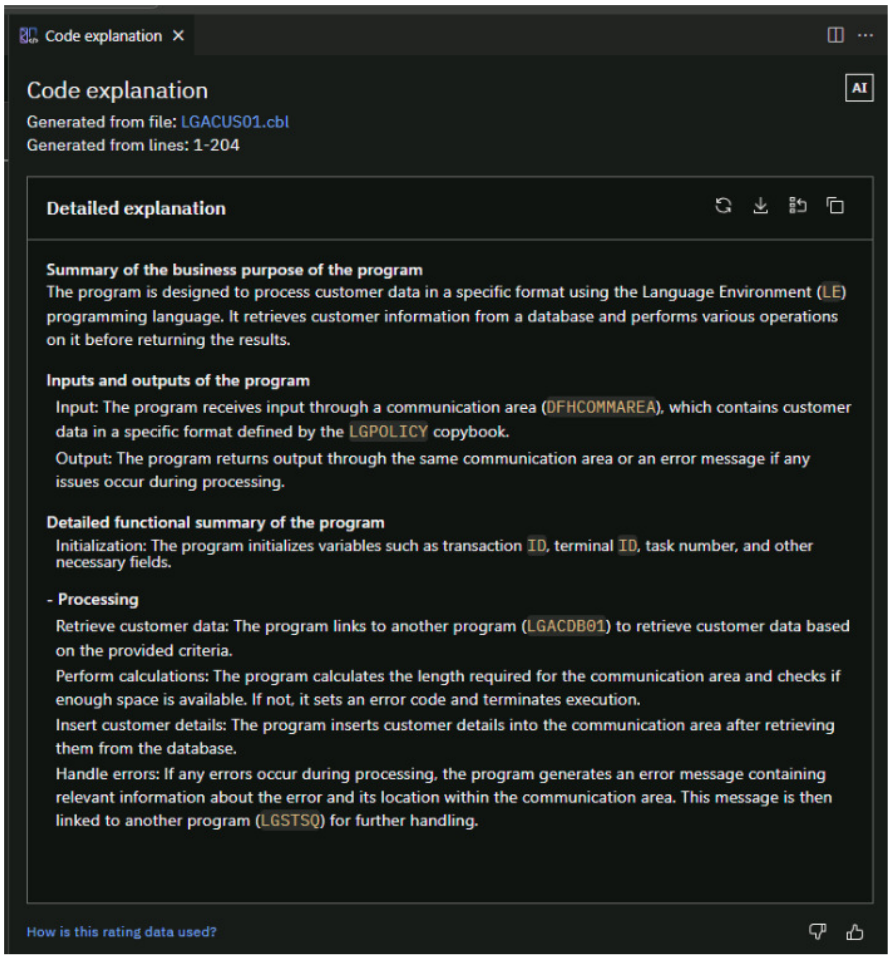
14. After clicking on the 3 dots/ellipses besides the **LGACUS01**, you will get the following options.



a. View properties
b. Identify conditional statements.
c. Identify paragraphs.
d. Identify tables/file access statements.
e. Identify variables
f. Generate simple explanation (This is the WCA4z Code Explanation feature)
g. Generate detailed explanation (This is the WCA4z Code Explanation feature)
h. Generate guided explanation (This is the WCA4z Code Explanation feature)

15. Review the Code Explanation feature which uses AI to explain to the COBOL developer what the function does.

    Please note, to execute Code Explanation, you need to have entered in the API Key in the previous step.



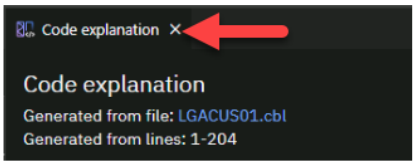
 Regenerate the explanation

 Download the code explanation

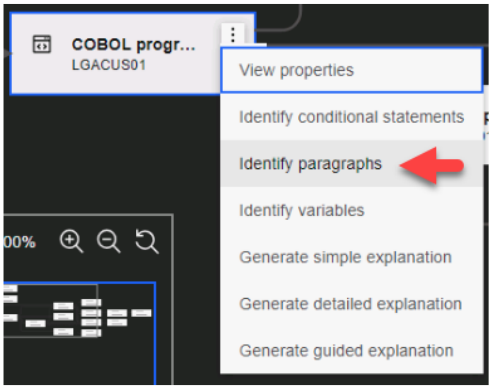
 Insert as a comment in the COBOL program

 Copy the explanation

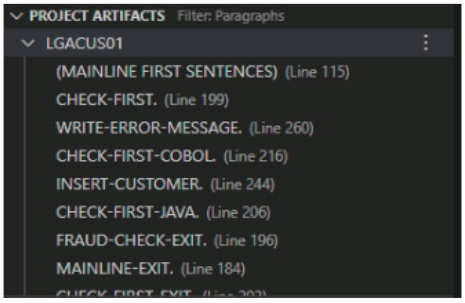
16. Close the explanation by clicking the ‘X’, once you have finished reviewing the code explanation.



17. Go back to the “Z Refactoring Assistant” window, click on the 3 dots/ellipses besides the **LGACUS01** and select **Identify paragraphs** to display all paragraphs in the code in order.

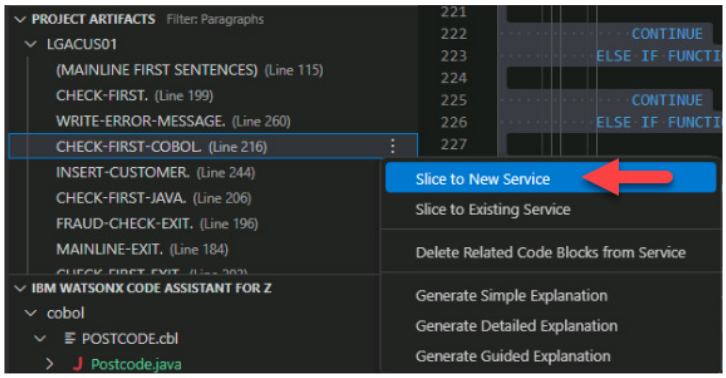


18. It will open the “Project Artifacts” section and show the COBOL paragraphs

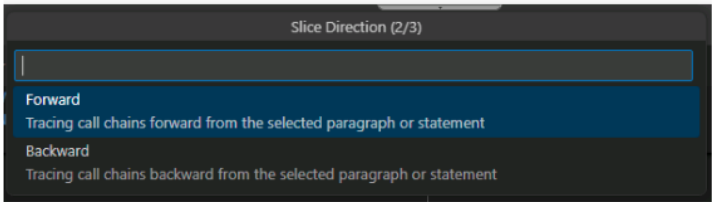


19. Single click on paragraph **CHECK-FIRST-COBOL** in “Project Artifacts” section and then single left click on the CHECK-FIRST-COBOL paragraph in that view.

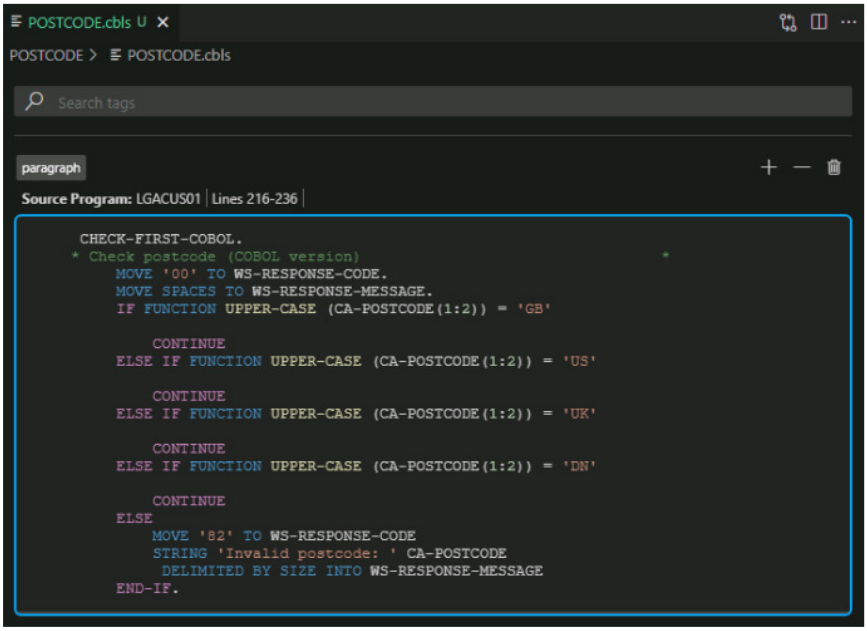
20. Click on the 3 dots/ellipses besides the CHECK-FIRST-COBOL and select “**Slice to New Service”** and then give it a name of **POSTCODE**.



21. Select **forward** and then press **enter** when you see ‘paragraph,’. This will give your sliced code a tag name of ‘paragraph’.

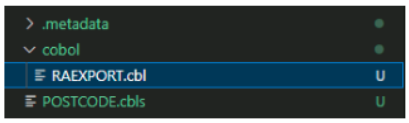


22. Confirm that the extracted logic is correct in the view on the right. This view shows you’re your code before you extract it into a COBOL program, hence the **.cbls** extension.

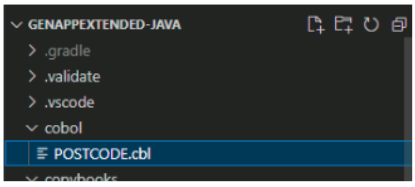


23. Once finished, you typically right click on the POSTCODE.cbls file and select **watsonx Code Assistant for Z > Generate Service Code** and give it a name called **RAEXPORT.**

    This is what you would view. **However, this lab doesn't require you to do this because the COBOL program has already been extracted for you**.

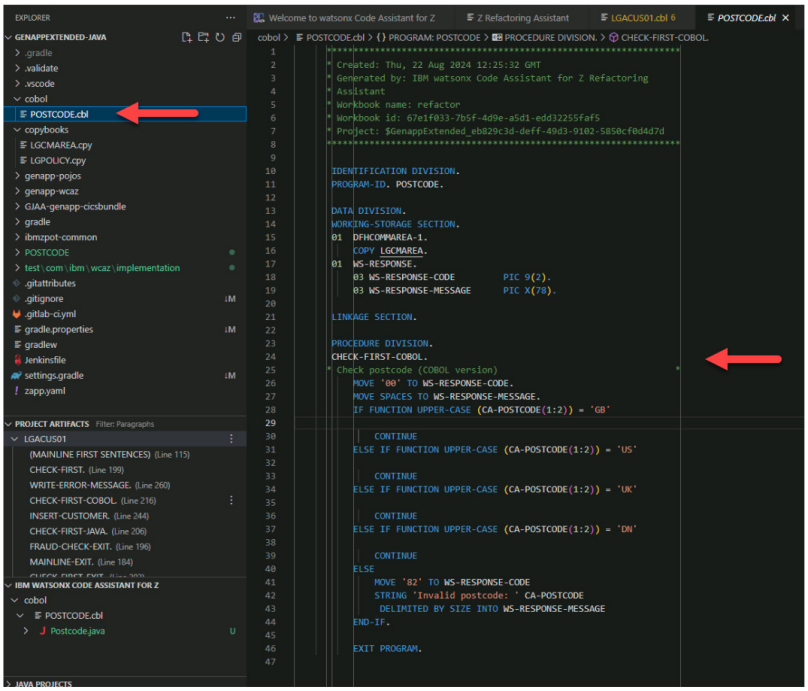


24. This lab has extracted the POSTCODE refactored program for you already and stored it here.



25. For the rest of this lab, you will use this COBOL program called POSTCODE.

26. Open the program and you will see the code.



Please note: Because we are transforming this COBOL functionality to Java, we don't need to think about how we call this new COBOL module from the main COBOL module.

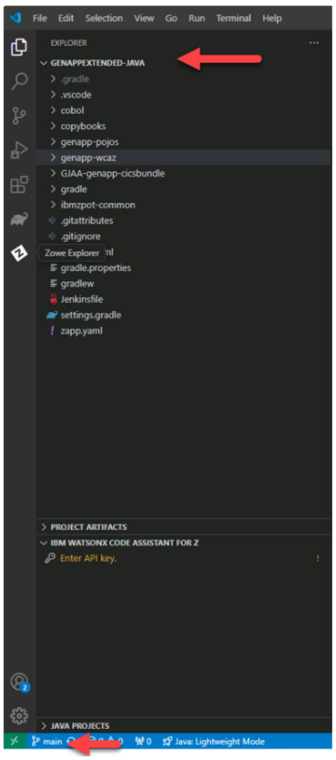
In this **Refactor** phase,

1. The Java developer (accompanied with the COBOL developer) has collaborated using their individual logins to RA to refactor service to POSTCODE validation.
2. They both (Java and COBOL developer) used the web user interface to enable better collaboration.
3. Now the Java developer is ready to Transform the COBOL service to Java using AI.

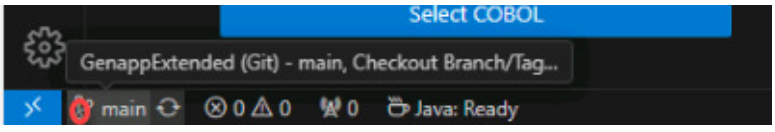
We will use this refactored program for the Transform phase next.

How to execute the WCA4z Transform Phase using VS Code

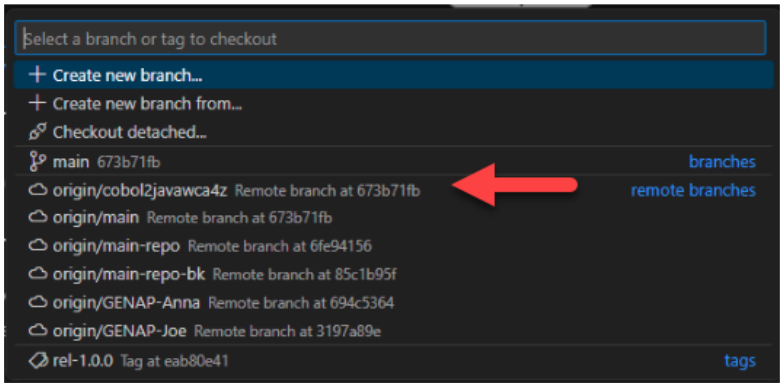
8. In the VS Code, you will see a Git repository that has already been cloned in your repo called **'GENAPPEXTENDED-JAVA'** on branch **'main'**.



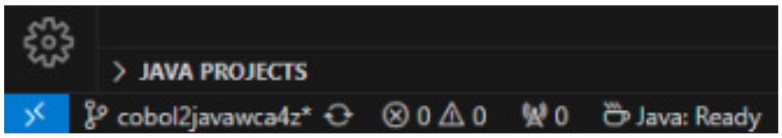
9. Switch from the main branch (production) to your own personal branch (**cobol2javawca4z**) by clicking **'main'**.



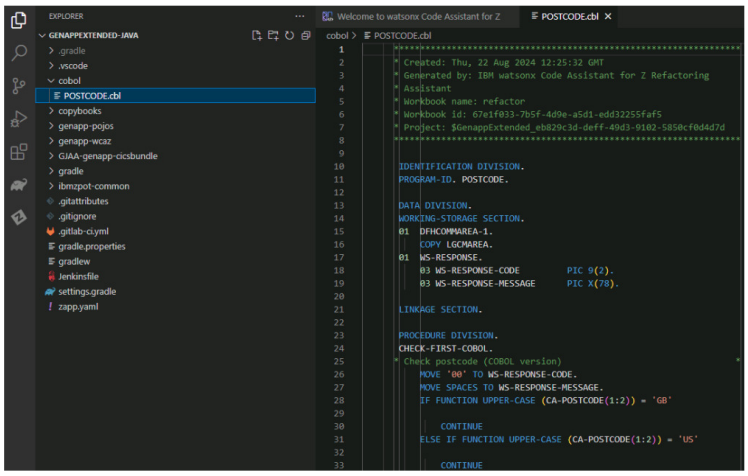
10. Then select **cobol2javawca4z** branch



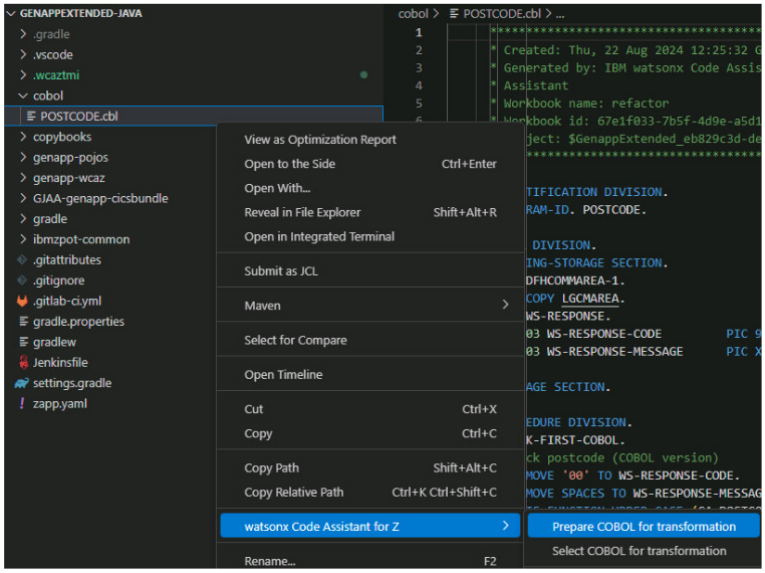
11. To confirm your branch has been switched, you should see this:



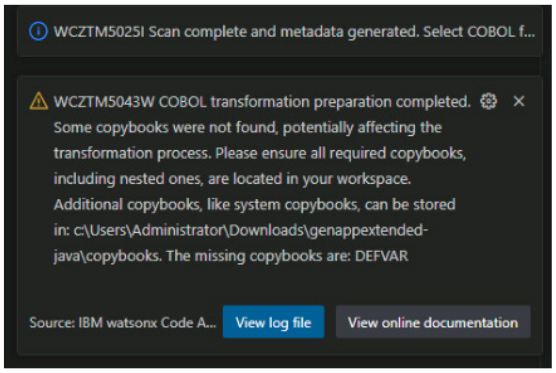
12. Locate the program POSTCODE.cbl in the Project Explorer which is the refactored program from the previous task.



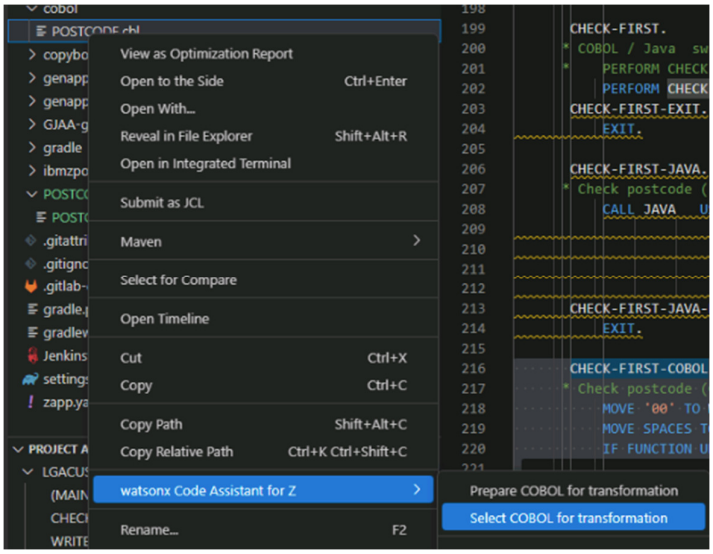
13. Right click **POSTCODE** and go to "**watsonx Code Assistant for Z**" then click on option "**Prepare COBOL for Transformation**".



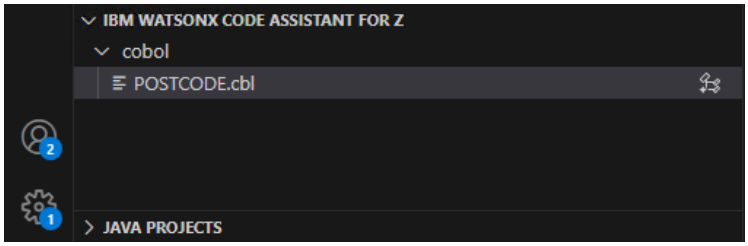
When you see this, this means the Prepare COBOL for transformation step is complete.



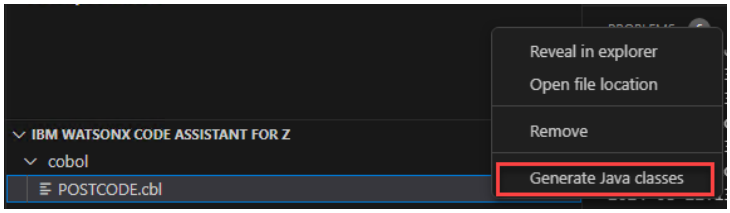
14. Then right click **POSTCODE** and go to “**watsonx Code Assistant for Z**” then click on option “**Select COBOL for Transformation**”.



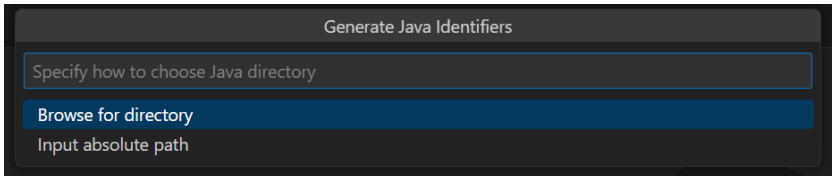
15. This will import the COBOL program into “IBM WATSONX CODE ASSISTANT FOR Z”.



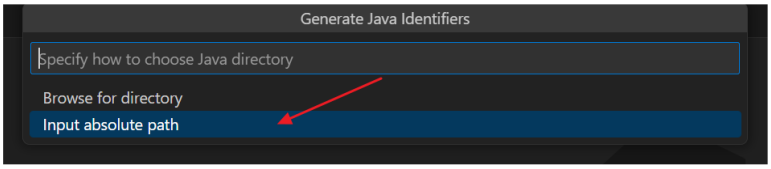
16. Right Click on **‘cobol\POSTCODE’** and select ‘**Transform to Java**’.



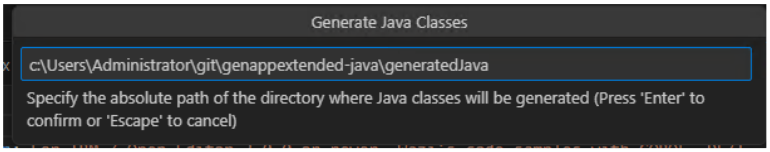
17. This will open new input bar in the top middle.



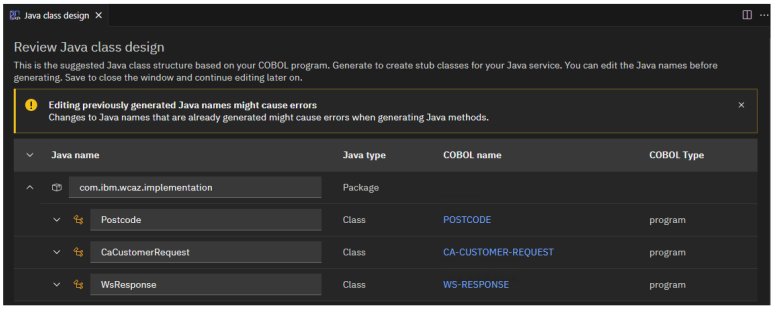
18. Click on “**Input absolute path”** option.



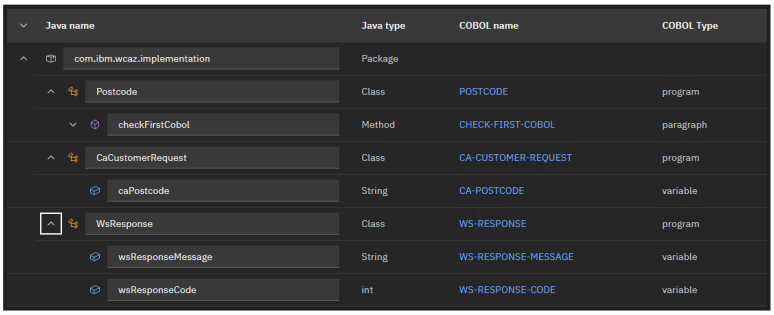
19. Press **end** to reach end of the default path and type addition to path **\generatedJava**.



20. New tab, **Code transformation**, will open.



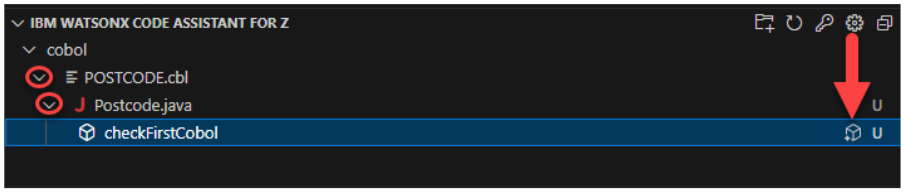
21. Click to expand different artifacts using following buttons.



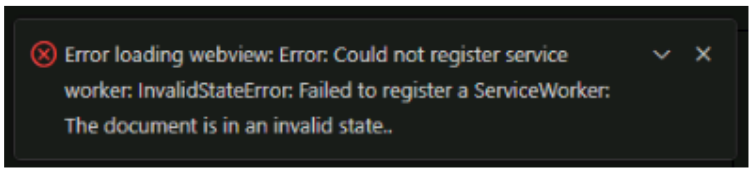
22. At the right bottom of '**Code transformation**' tab, click on "**Generate Java classes**".

23. Click on the icon to the left after expanding the **POSTCODE.cbl** and **Postcode.java.**

IF THE TWISTY TAKES TOO LONG TO LOAD THEN PLEASE RESTART VSCODE AND TRY AGAIN. IT CAN TAKE UP TO 30 SECONDS FOR THE TWISTY TO LOAD.

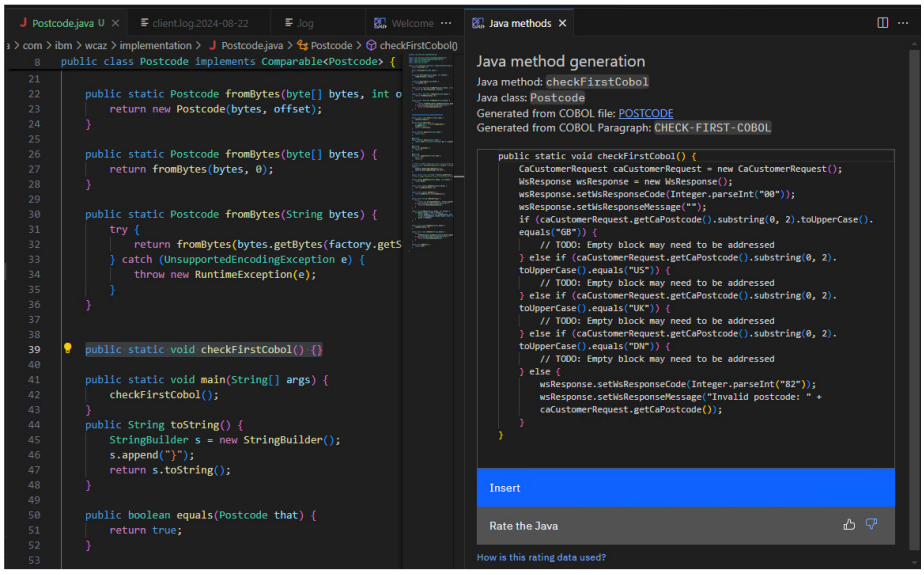


IF YOU SEE THIS ERROR AFTER SELECTING THE OPTION ABOVE.



RESTART VSCODE AND WAIT FOR ABOUT A MINUTE, THEN RETRY THE ACTION ABOVE.

24. After clicking on icon, Java method gets generated on the right , click on **insert**



Generated java method for '**postcode**' will get inserted in the java class.

25. Once done, all the methods are now tick marked that means it is not empty.

26. The WCA4z AI is continuously evolving, which means the transformed Java code it generates may vary over time.

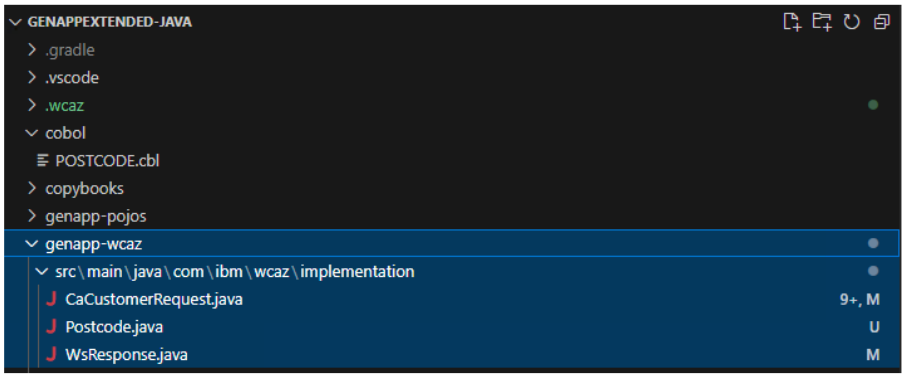
    To ensure consistency during your lab experience, we've already included the version of the transformed Java that was generated at the time the lab was created. This version has been copied into the Gradle project folder within the repository (you'll find the exact path listed below).

    We copied this code ahead of time because if you were to re-run the transformation today, the Java might look slightly different due to ongoing updates in the AI model. By providing a fixed version, we avoid inconsistencies and make sure everyone in the lab is working from the same baseline.

    Additionally, we've already applied the necessary changes to make the Java code CICS-compatible—that is, runnable in a CICS environment. These changes are specific modifications required for the Java application to interact properly with CICS, IBM's transaction processing system for mainframes, so a client can gain the best of both worlds in running COBOL (traditional) and Java (newer) languages on one platform.

    You'll be able to review exactly what those CICS-specific changes are in the following section of the lab.

    genapp-wcaz\src\main\java\com\ibm\wcaz\implementation



    Note: "Transformed Java" refers to the Java code that has been automatically generated from COBOL source code by the WCA4z AI. Since the AI can improve and update how it transforms the code, the output might change slightly depending on when the transformation is done e.g. you may see more or less classes, with different names.

In this **Transform** phase,

1. We imported this workbook using IBM watsonx code assistant for z.
2. We generated Java classes and Java methods.

We executed steps for **Understand, Refactor and Transform** phases to create java classes and methods for the **postcode validation** functionality in **POSTCODE** program.

Now, let's review the changes that have already taken place in your transformed Java (read only section)

This section explains the key steps that were taken to make the transformed Java code callable from COBOL via CICS. These updates ensure that Java logic can be triggered by COBOL programs and that input/output data is handled properly within the hybrid application structure.

1. Annotate the checkFirst method of the Postchck class with @CICSProgram
    - to enable the method to be invoked via EXEC CICS LINK
2. Update the CaCustomerRequest constructor method
    - to initialise the Java object with the input data provided by the COBOL caller
3. Add a method to the WsResponse class, and invoke it at the end of checkFirst
    - to create and return the output data expected by the COBOL caller

Background Information

In order to minimise the following code changes, most of the code (including the CICS API commands and JCICS method calls) has been encapsulated within a common framework provided with the lab environment. It is not necessary to understand the framework code in order to complete this lab, but reference information is provided in the Appendices for those who are interested. You can choose which Appendices to review depending on your level of interest:

Appendix A provides a short overview of COBOL and Java interoperability in CICS

Appendix B provides an overview of the 3 functions provided by the framework

Appendix C provides a technical overview of the framework

Appendix D lists the COBOL source code used in the framework

Appendix E lists the Java source code used in the framework

Feel free to continue with the lab instructions - you can reference an Appendix if required as you proceed.

1) Annotate the method to be invoked from COBOL with a @CICSProgram

Java Method Annotated with @CICSProgram

To enable COBOL to invoke Java logic, the checkFirstCobol method in the Postcode class was annotated with @CICSProgram.

- This annotation marks the method as callable from COBOL using the EXEC CICS LINK command.
- As part of this, CICS automatically generated a corresponding PROGRAM definition named LGACJV02.

Code Elements:

- **Note 1:** Required CICS package was imported.
- **Note 2:** The method was marked with @CICSProgram("LGACJV02"), making it callable via COBOL.

This setup allows COBOL programs to seamlessly invoke this Java method as part of hybrid application logic.

```
package com.ibm.wcaz.implementation;
.
.
public static void checkFirstCobol() {
  .
  .
```

Change made:

```
package com.ibm.wcaz.implementation;
import com.ibm.cics.server.invocation.CICSProgram;          // Note 1
.
.
@CICSProgram("LGACJV02")                                    // Note 2
public static void checkFirstCobol() {
  .
  .
```

2) Retrieve the input data provided by the COBOL caller

Input Data Retrieved from COBOL Caller

When COBOL invokes a Java program, it can pass input data that needs to be captured and processed on the Java side. This was achieved in the CaCustomerRequest class by leveraging the constructor.

- The no-argument constructor initializes request data using a CobolData object.
- The caPostcode field is populated from COBOL-provided input.

Code Elements:

- **Note 1:** Framework package for CobolData was imported.
- **Note 2:** A new CobolData object (input) was instantiated.
- **Note 3:** Input data from COBOL was assigned to the caPostcode variable.

This ensures that when the COBOL caller invokes the Java method, the input is properly captured and available for business logic execution.

```
package com.ibm.wcaz.implementation;
.
.
public class CaCustomerRequest implements Comparable<CaCustomerRequest> {
.
   public CaCustomerRequest() {
   }
```

Change made:

```
package com.ibm.wcaz.implementation;
import com.ibmzpot.common.CobolData;                    // Note 1
.
.
public class CaCustomerRequest implements Comparable<CaCustomerRequest> {    .
   public CaCustomerRequest() {
      CobolData input = new CobolData();                    // Note 2
      this.caPostcode = input.getCobolData();          // Note 3
   }
```

3a) Create a method to return the response data to the COBOL caller

Response Method Created in WsResponse Class

To send data back to COBOL after Java logic executes, a method named returnWsResponse was added to the WsResponse class.

- This method constructs a CobolData object named output, which is populated with the response.
- Formatting conventions (2-digit code with leading zero, 78-character padded message) were applied to meet COBOL expectations.

Code Elements:

- **Note 1:** Framework package for CobolData was imported.
- **Note 2:** Method returnWsResponse was defined.
- **Note 3:** output object instantiated.
- **Note 4:** Response code padded to two digits.
- **Note 5:** Response message padded to 78 characters.
- **Note 6:** output was populated with both fields.
- **Note 7:** The method was closed properly within the class scope.

This design guarantees that COBOL receives well-formed and expected response structures from the Java logic.

```
package com.ibm.wcaz.implementation;

.

.

public class WsResponse implements Comparable<WsResponse> {

  .

  .

}
```

Change made:

```
package com.ibm.wcaz.implementation;
import com.ibmzpot.common.CobolData;                                    // Note 1
.
.
public class WsResponse implements Comparable<WsResponse> {
  .
  .
   public void returnWsResponse() {                                     // Note 2
          CobolData output = new CobolData();                           // Note 3
    // pad the response code with leading zeros       Note 4
          String fixedResponseCode = String.format("%02d", this.wsResponseCode);
    // pad the response msg with trailing spaces      Note 5
          String fixedResponseMessage = String.format("%-78s", this.wsResponseMessage);
          output.putCobolData(fixedResponseCode + fixedResponseMessage);   // Note 6
   }                                                 // Note 7

}
```

3b) Invoke the method to return the response data to the COBOL caller

Response Method Invoked in Postcode.checkFirstCobol

After completing the core business logic inside checkFirstCobol, the new returnWsResponse method was invoked as the final step.

- This ensures response data is packaged and returned to the COBOL caller before the method exits.

Code Elements:

- **Note 1:** returnWsResponse() is invoked as the **last statement** inside checkFirstCobol.

This final invocation completes the end-to-end flow: from COBOL to Java and back—firmly establishing the round-trip communication between COBOL and transformed Java.

Change made:

```
@CICSProgram("LGACJV02")
public static void checkFirstCobol() {
  .
  .
  wsResponse.returnWsResponse();                    // Note 1
}
```

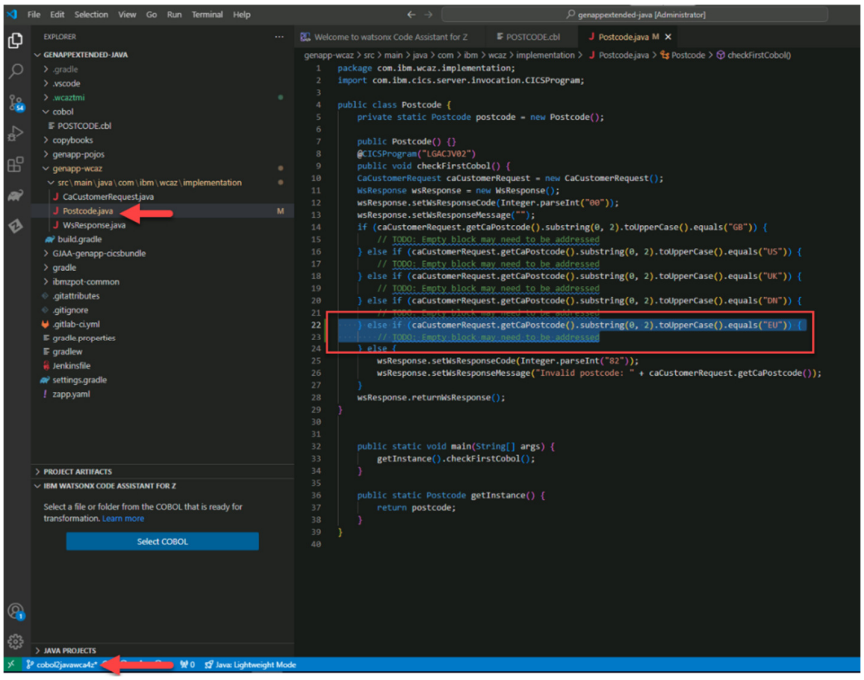
Note 1: This statement invokes the **returnWsResponse** method you created in Step 3a. It needs to be the very last statement in **checkFirstCobol**, i.e. after the business logic, and before the implicit return to the COBOL caller.

Implement the new feature in Java

Remember what it says in your Git Lab ticket: http://wca4z-gitlab.ibm.com:10880/wca4z/genappextended-java/-/issues/1

Now, locate the Java program, postcode.java and open it in your editor and add this change (highlighted) in your postcode.java.

Note: Make sure there are no syntax errors. You will see red lines if there are any syntax errors.




```
} else if (caCustomerRequest.getCaPostcode().substring(0, 2).toUpperCase().equals("EU")) {
    // TODO: Empty block may need to be addressed
```

The Java source changes are now complete:

- the **Postcode** class has been updated to enable it to be invoked from COBOL CICS with the new business logic
- the **CaCustomerRequest** class has been updated to receive the input data from COBOL CICS
- the **WsResponse** class has been updated to return the output data to COBOL CICS

You are now ready to build and deploy the Java programs to CICS. Proceed to the next section.

Now, let's build the Java locally, deploy to CICS and test using Galasa part of IBM Test Accelerator for z

Return to Visual Studio Code

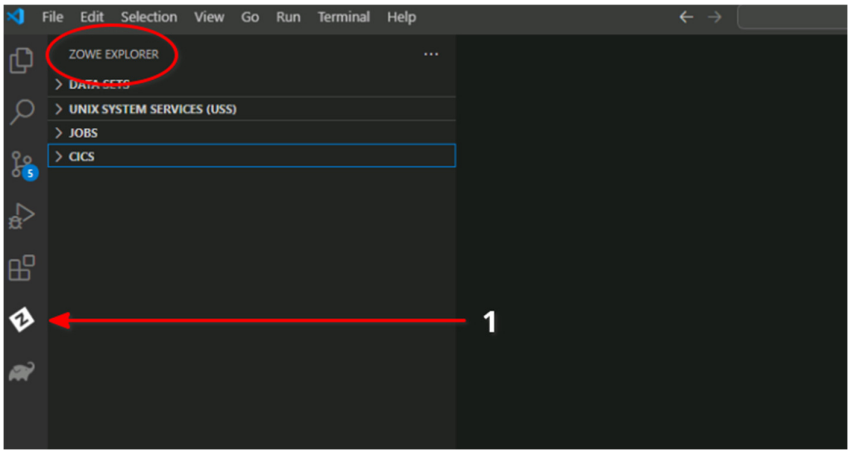
a. **Action:** In the Primary Sidebar (left hand side), confirm your project remains open in the Explorer view

b. **Optional Action:** In the main Editor area, close any open files

Confirm LGACJV02 (Postcode Java) is not already deployed to CICS

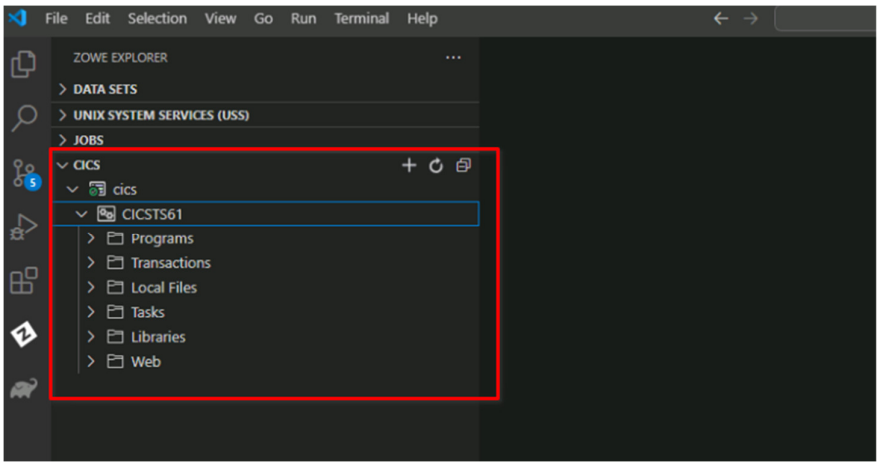
a) **Action:** In the Activity Bar (left hand side), single click the Zowe Explorer extension icon (shaped like the letter "Z") as illustrated below.

- **Result:** The VS Code primary sidebar now contains the Zowe Explorer panels:

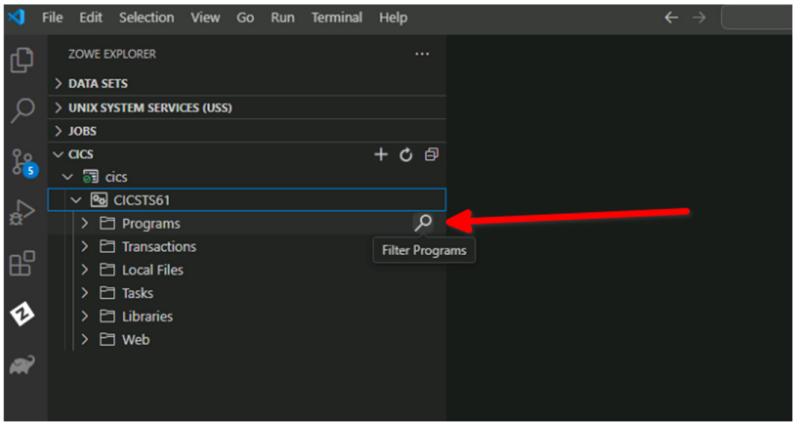


b) **Action:** At the bottom of the VS Code primary sidebar, in the Zowe Explorer CICS panel, expand the CICS tree recursively until the submenu under CICS region **CICSTS61** is displayed.

- **Result:**

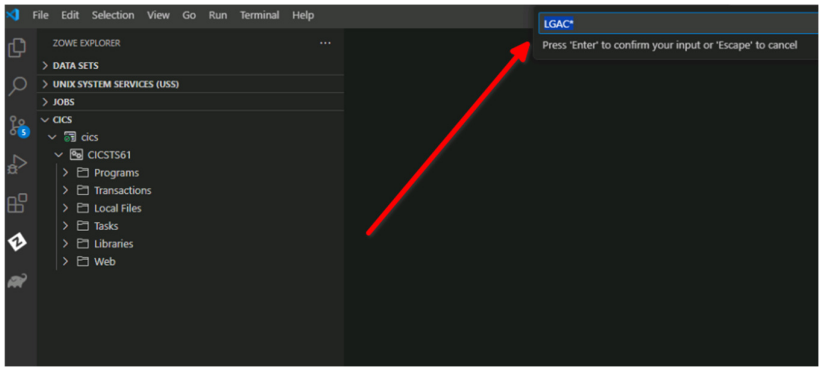


c) **Action:** To the right of the "Programs" entry, click the "Filter Programs" icon (shaped like a magnifying glass) as illustrated below:

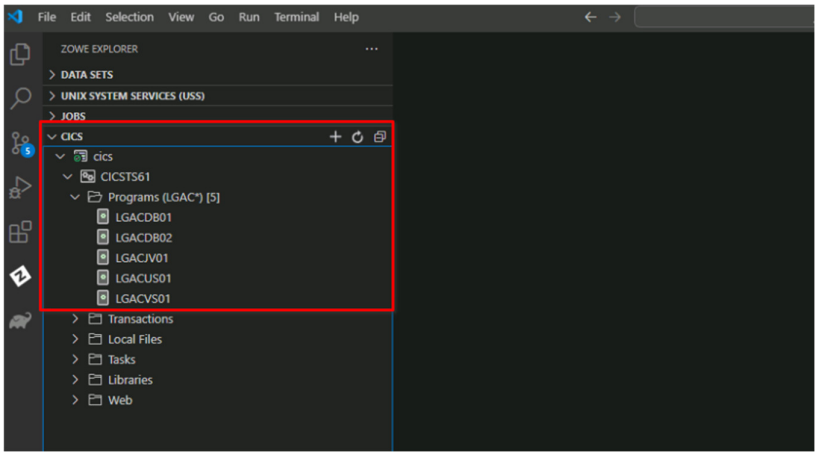


- **Result:** A filter prompt will open in the VS Code command palette at the top of the display.

d) **Action:** In the VS Code command palette (top of display), type a filter value of **LGAC*** and press <ENTER>, then press <ENTER> again to confirm.



- **Result:** All programs matching the filter are displayed in the Zowe Explorer panel:



e) **Action:** Review the list of programs displayed and note **LGACJV02** is **NOT** listed.

- **Info: LGACJV02** is the **@CICSProgram** annotation you assigned to **Postcode** method **CheckFirstCobol**; it's not listed because the Java program has not been deployed to CICS yet.

f) **Optional Action:** The CICS Explorer can also be used to display a list of installed programs. If you're familiar with the CICS Explorer:

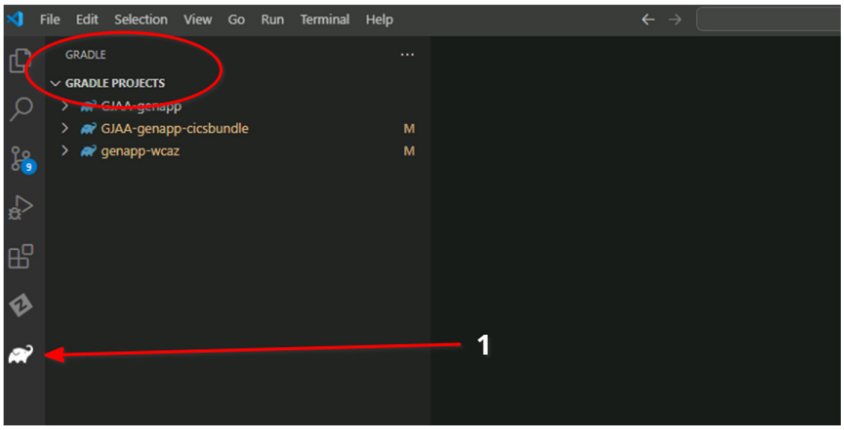
- return to IBM Developer for Z and switch to the CICS SM perspective
- In the Host Connections view, connect to CMCI CICSTS61
- In the CICSplex Explorer view, single click CICSTS61
- In the Programs view, review the list of installed programs and note LGAJV02 is **NOT** listed

3. **Perform a local build of the generated Java**

a) **Info:** You will perform a local build using the VS Code **Gradle for Java** extension.

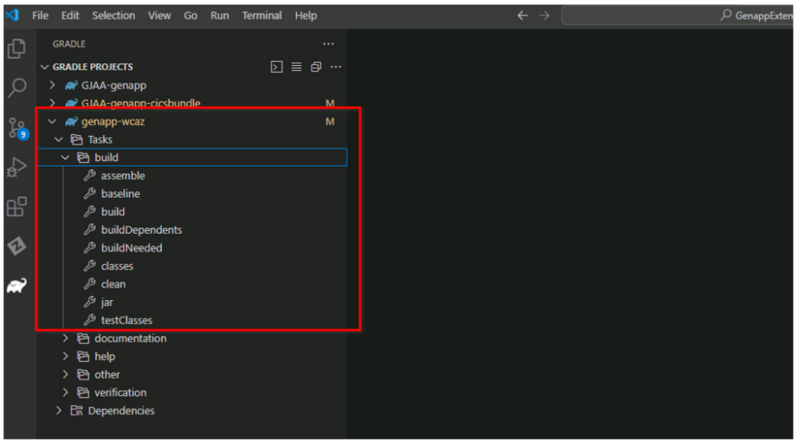
b) **Action:** In the VS Code activity bar (left hand side), single click the Gradle extension icon (shaped like an elephant) as illustrated below.

- **Result:** The VS Code primary sidebar now displays your Gradle project:

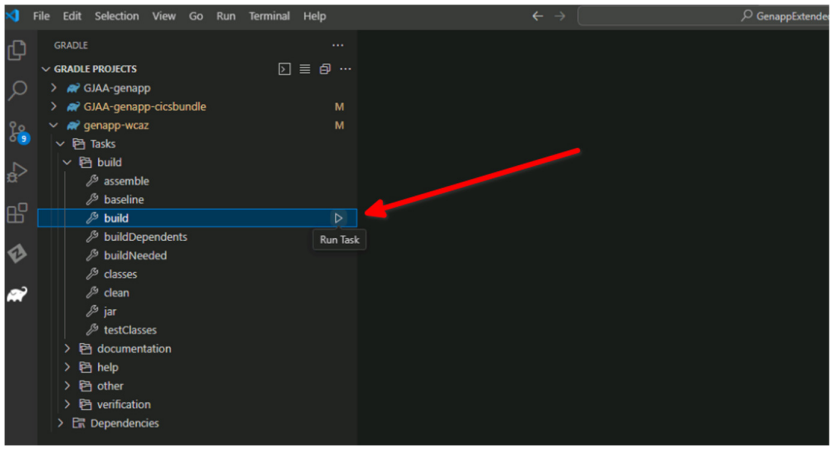


c) **Action:** In the VS Code Gradle project list, recursively expand the **genapp-wcaz** project until the **build** menu is displayed as illustrated below.

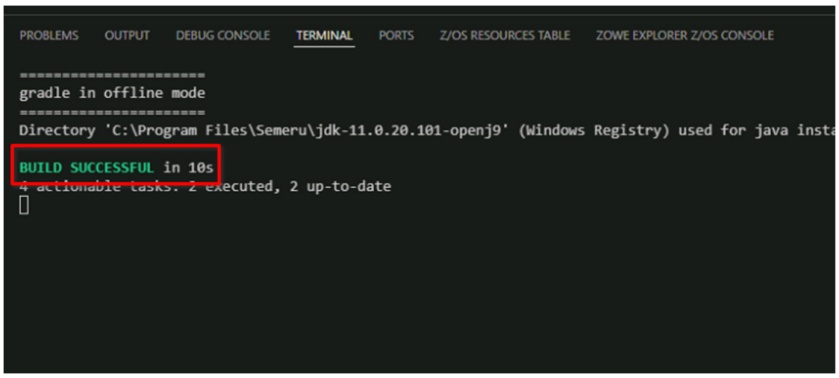
- **Result:** The **build** menu for **genapp-wcaz** is displayed:



d) **Action:** To the right of the **build** task, single click the **"Run Task"** icon, as illustrated below:



- **Info:** The build task compiles the Java source into Java bytecode.
- **Result:** A gradle build of **genapp-wcaz** is performed, and the result of the build is displayed in the VS Code Terminal view:



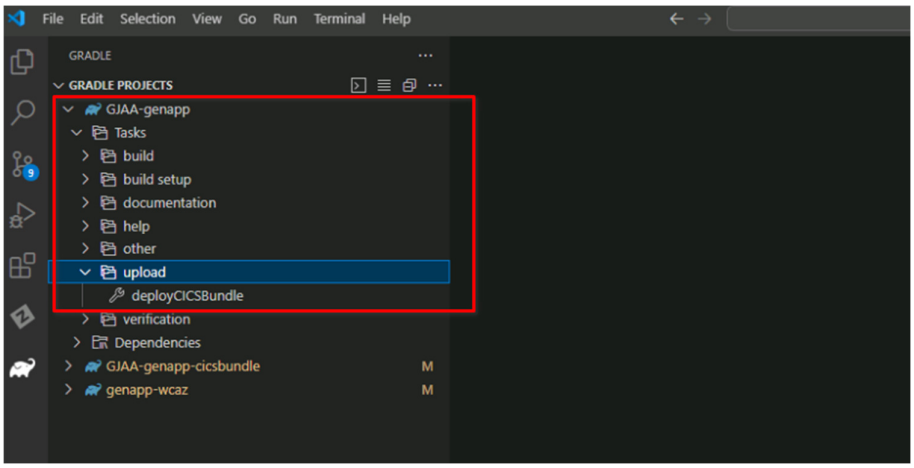
e) **Action:** Review the result of the build task. **"BUILD SUCCESSFUL"** in green letters denotes a successful build. If the build was unsuccessful, review and correct any errors; then repeat the build task. Notify the lab instructor you need assistance.

4. **Deploy the generated Java to CICS**

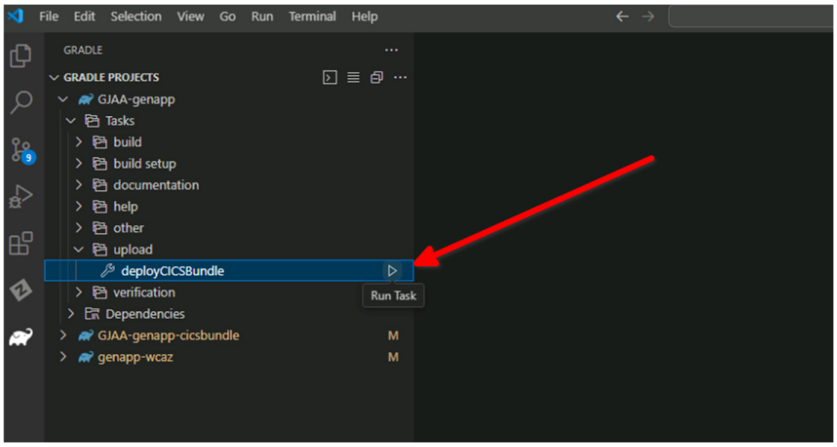
   a) **Info:** You will deploy the Java program to CICS using the VS Code Gradle for Java extension and the **CICS bundle Gradle plugin**.

   b) **Action:** In the VS Code Gradle project list, recursively expand the **GJAA-genapp** project until the **upload** menu is displayed as illustrated below.

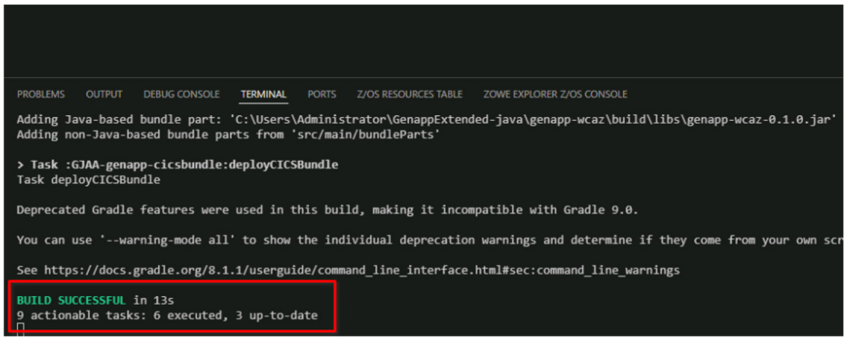
   - **Result:** The Upload menu for GJAA-genapp is displayed:



c) **Action:** To the right of the **deployCICSBundle** task, single click the **"Run Task"** icon, as illustrated below:



- **Info:** Running the **deployCICSBundle** task results the following actions:
  - the Java is packaged as an OSGi bundle inside a CICS bundle project
  - the CICS bundle project is uploaded to z/OS
  - the associated CICS bundle resource is installed into the target CICS region
  - the OSGi bundle is deployed to the target CICS JVM server
  - CICS automatically installs a PROGRAM resource for each @CICSprogram annotation it detects
- **Result:** The result of the **deployCICSBundle** task is displayed in the VS Code Terminal view as illustrated below:

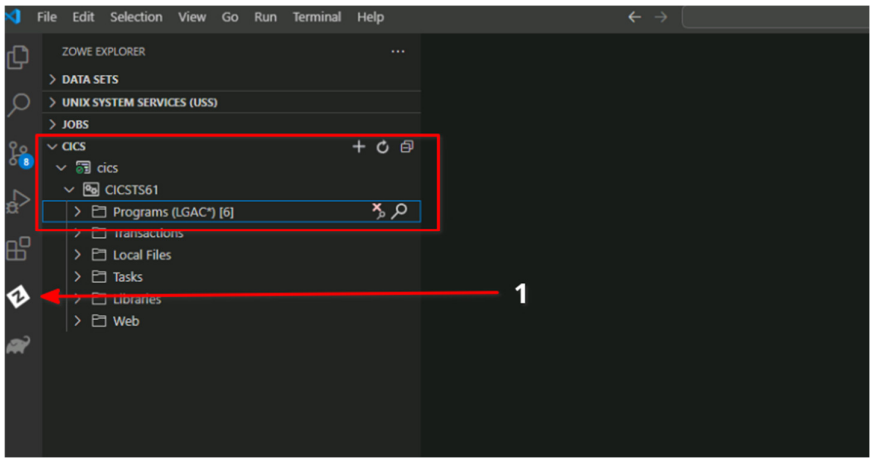


d) **Action:** Review the result of **deployCICSBundle** task. **"BUILD SUCCESSFUL"** in green letters denotes a successful deployment. If the deployment was unsuccessful, review the error messages to determine what went wrong. Notify the lab instructor if you need help.

5. **Confirm LGACJV02 (Postcode Java) has been deployed to CICS**

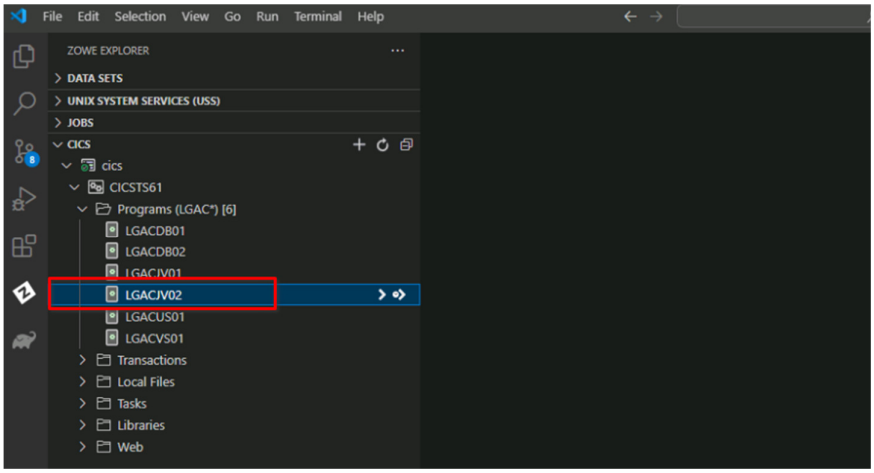
a) **Action:** In the VS Code activity bar (left hand side), single click the Zowe Explorer extension icon (shaped like the letter "Z").

- **Result:** The VS Code primary sidebar reverts to show the Zowe Explorer panels. At the bottom of the primary sidebar, the Zowe Explorer CICS panel should still show the list of programs in region CICSTS61, filtered with LGAC* (as illustrated below):



b) **Action:** Underneath the **CICSTS61** entry in the expanded CICS profile, double click the **Programs (LGAC*)** item to refresh the list of programs.

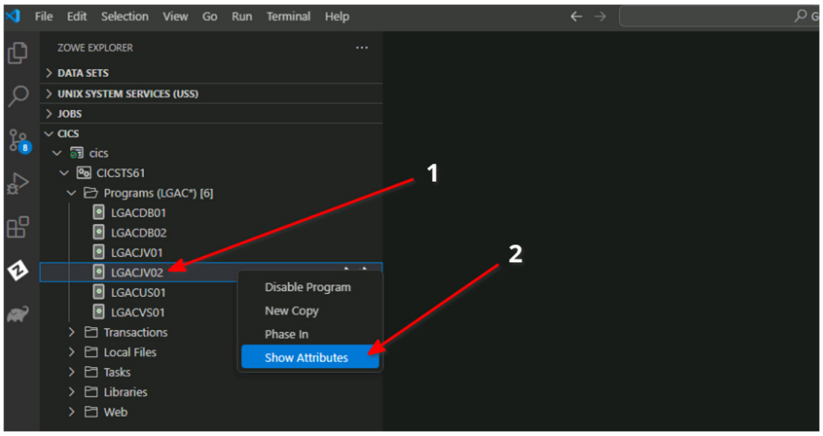
- **Result:** The list of installed programs is refreshed as illustrated below:



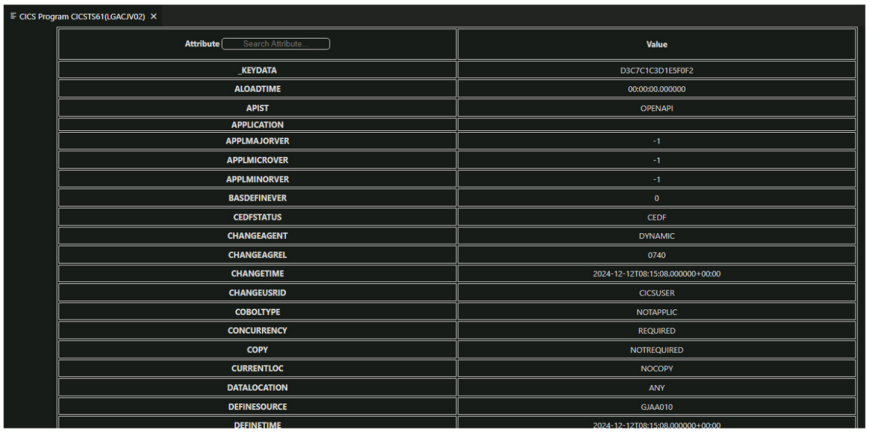
c) **Action:** Review the list of programs displayed and confirm **LGACJV02 IS NOW** listed.

- **Info:** CICS automatically installs a **PROGRAM** resource for each **@CICSprogram** annotation it detects during deployment

d) **Action:** Right click the **LGACJV02** program entry; from the resultant pop-up menu, click **Show Attributes**

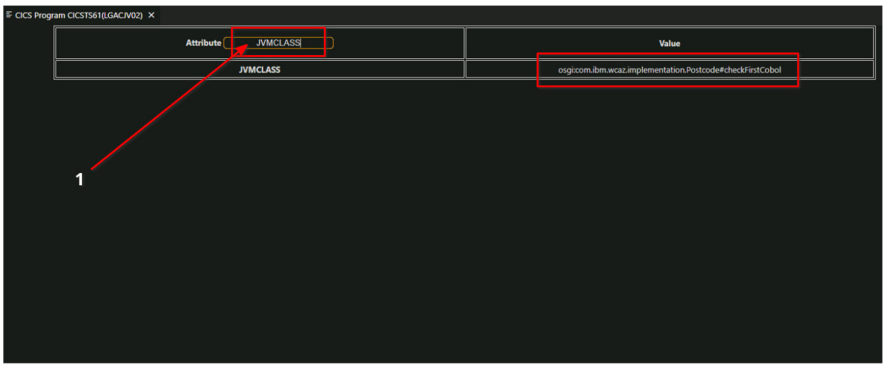


- **Result:** A new VS Code view opens displaying the attributes for program LGACJV02



e) **Action:** At the top of the Program Attributes view, in the **Search Attribute** box, type **JVMCLASS** as illustrated below.

- **Result:** A filter is applied to the Program Attributes view, and now shows the value associated with the attribute **JVMCLASS**



f) **Action:** Confirm the **JVMCLASS** attribute for program **LGACJV02** has the value **osgi:com.ibm.wcaz.implementation.Postcode#checkFirstCobol**

- **Info:** This **J**ute value confirms **LGACJV02** is targeting the **@CICSProgram** annotated method **checkFirstCobol** in class **com.ibm.wcaz.implementation.Postcode** in an OSGi bundle.

g) **Action:** In the **Search Attribute** box, search instead for **usecoun**t

- **Quiz question:** What is the value of **usecount**? What does this tell you?

h) **Action:** Close the Program Attributes view by single clicking the **X** in the tab at the top of the view.

- **Result:** The Program Attributes view is closed.

i) **Optional Action:** The CICS Explorer can also be used to validate the new program has been deployed. If you're familiar with the CICS Explorer:

- return to IBM Developer for Z and switch to the CICS SM perspective
- In the Host Connections view, connect to CMCI CICSTS61 (if not connected already)
- In the CICSplex Explorer view, single click CICSTS61
- In the Programs view, refresh the list of installed programs and confirm **LGACJV02** has now been installed

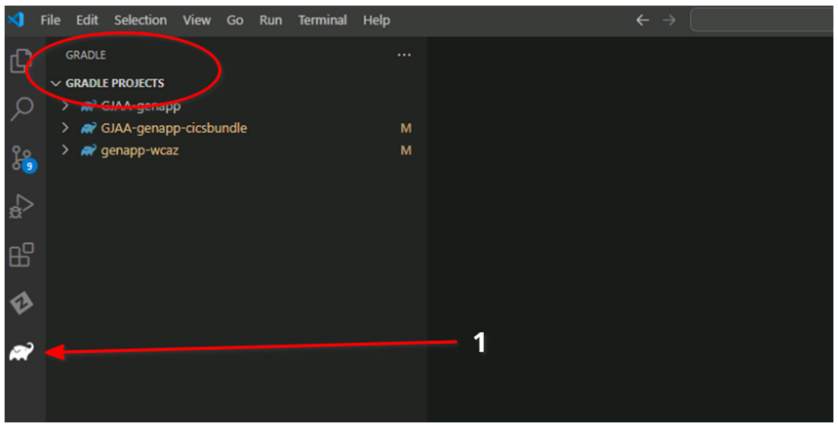
6. Test the deployed Java program

a) **Info:** You will test the deployed Java program by triggering **Galasa** tests from within the VS Code Gradle for Java extension. The Galasa tests will in turn invoke the deployed Java program via the CICS Command Level Interpreter transaction **CECI** - by executing the command **EXEC CICS LINK PROGRAM(LGACJV02)**.

b) **Info:** Galasa is an open source integration test framework. Galasa tests can be easily created using **IBM Test Accelerator for Z -** the Galasa tests used to invoke the deployed Java are already provided in the lab environment. Ask a lab instructor if you would like more information about Galasa and/or IBM Test Accelerator for Z.

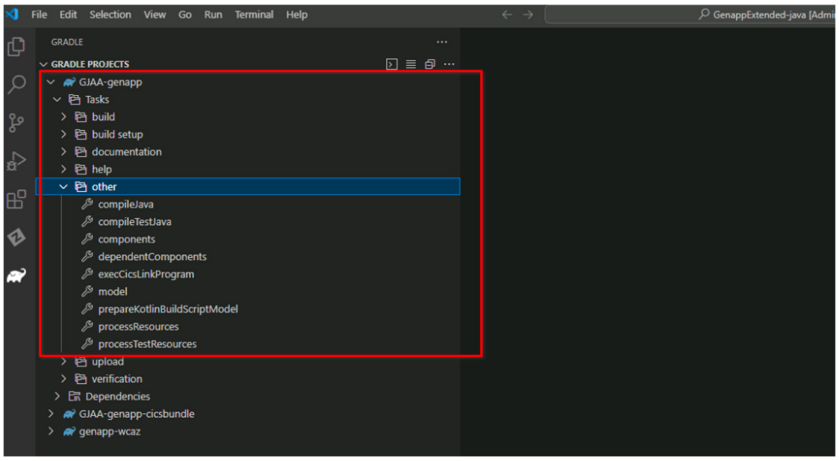
c) **Action:** In the VS Code activity bar (left hand side), single click the Gradle extension icon (shaped like an elephant).

- **Result:** The VS Code primary sidebar reverts to show your Gradle project illustrated below):

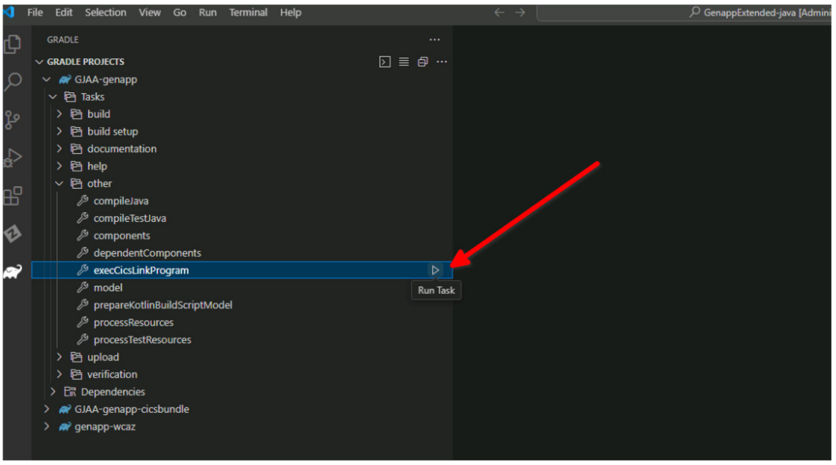


d) **Action:** In the VS Code Gradle project list, recursively expand the **GJAA-genapp** project until the **other** menu is displayed.

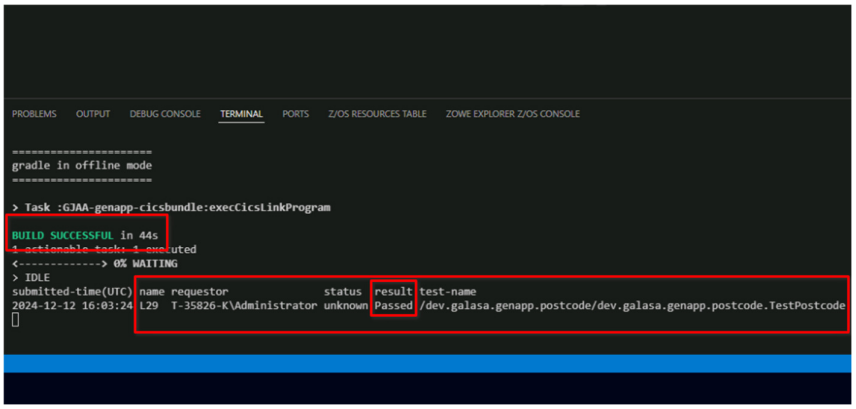
- **Result:** The **other** menu for **GJAA-genapp** is displayed as illustrated below:



e) **Action:** To the right of the **execCicsLinkProgram** task, single click the **"Run Task"** icon as illustrated below:



- **Info:** The **execCICSLinkProgram** task invokes the deployed Java program via EXEC CICS LINK as described previously.
- **Result:** The **execCICSLinkProgram** task might take up to 2 minutes to complete - please be patient :-). The result of the task is displayed in the VS Code Terminal view as illustrated below:



f) **Action:** Review the result of test(s) performed by the **execCicsLinkProgram** task. **"BUILD SUCCESSFUL"** in green letters denotes the task itself has completed, but the result of each test (**Passed** or **Failed**) is displayed individually as illustrated above.

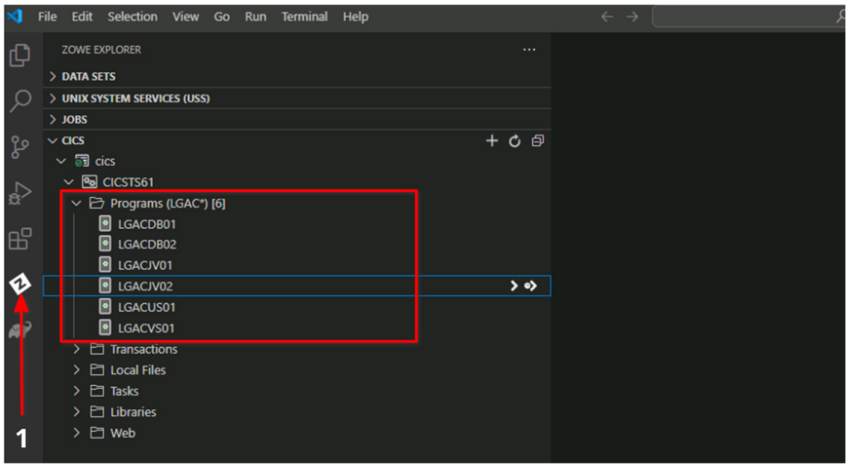
- **Info:** The Galasa task consists of one or more EXEC CICS LINK tests: for example one with a valid postcode as input, and perhaps one with an invalid postcode. The response from each test is checked to ensure it contains the appropriate response data.

g) **Optional Action:** If you want to review the full output from the Galasa task, open the desktop file manager and navigate to **C:\User\Administrator\.galasa\ras\<name>**, where **<name>** denotes the name of the test - the name of each test is displayed with the test result status (e.g., L29 in the above example).

7. Confirm your deployed Java has been executed

a) **Action:** In the VS Code activity bar (left hand side), single click the Zowe Explorer extension icon (shaped like the letter "Z")

- **Result:** The VS Code primary sidebar reverts to show the Zowe Explorer panels. At the bottom of the primary sidebar, the Zowe Explorer CICS panel should still show the list of programs in region CICSTS61, filtered with LGAC* (as illustrated below):



b) **Action:** Underneath the CICSTS61 entry in the expanded CICS profile, double click the **Programs (LGAC*)** item to refresh the list of programs.

- **Result:** The list of installed programs is refreshed.

c) **Action:** Right click the **LGACJV02** program entry; from the resultant pop-up menu, click **Show Attributes**

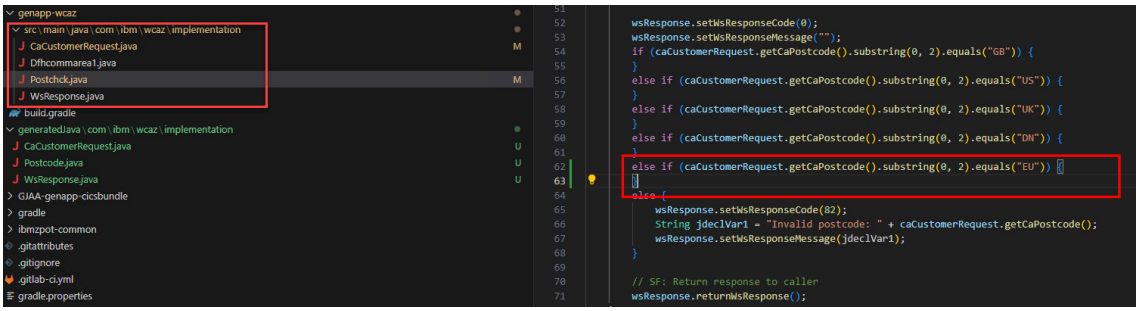
- **Result:** A new VS Code view opens displaying the attributes for program LGACJV02

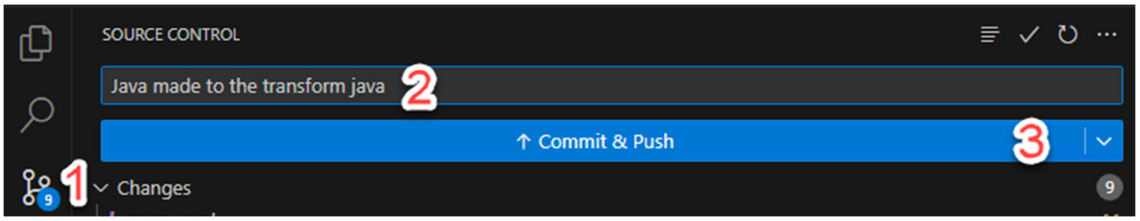
d) **Action:** At the top of the Program Attributes view, in the **Search Attribute** box, type **usecount**

- **Quiz question:** What is the value of **usecount**? What does this tell you?

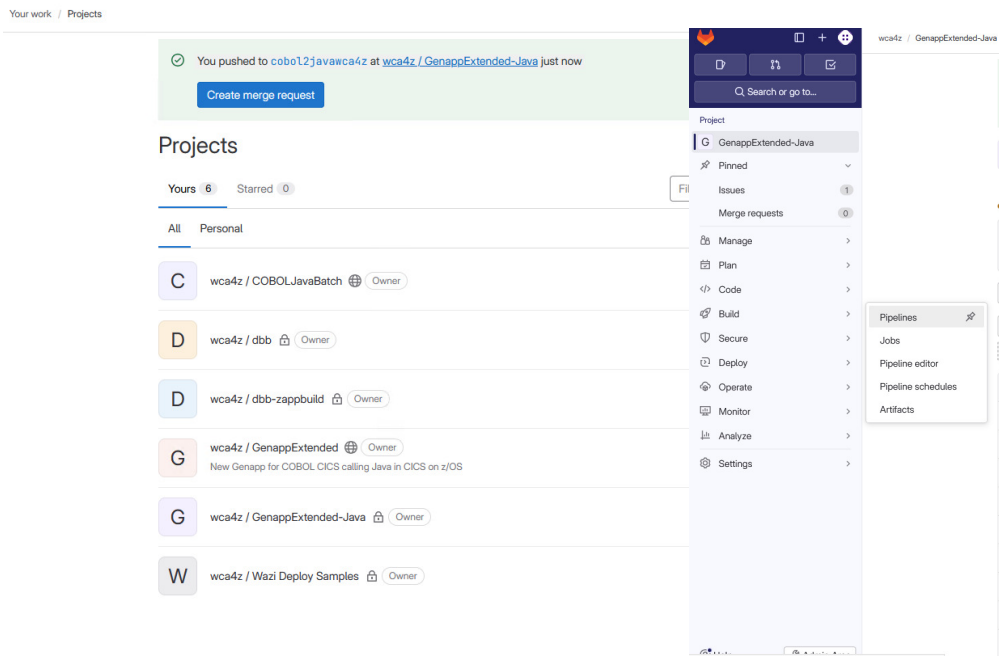
Execute the Java on z DevOps Pipeline

Now the Java changes have been completed, let's deploy your Java via the DevOps pipeline to CICS on z/OS.

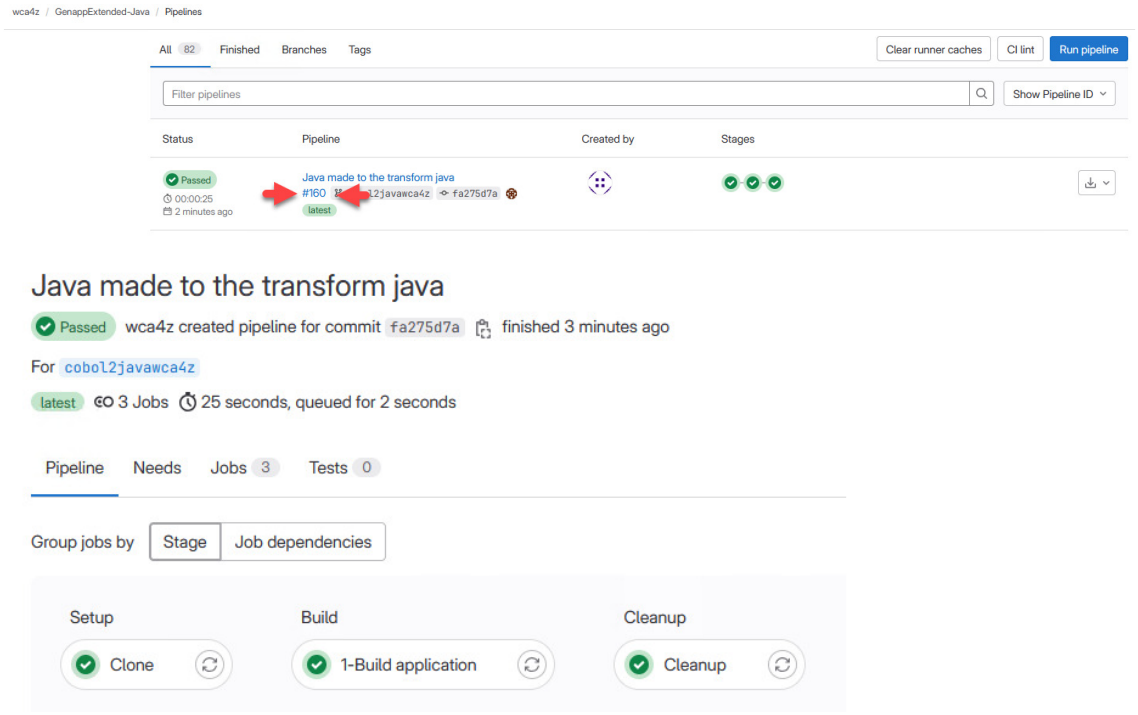




Now login to Git Lab by going to Firefox and clicking the bookmark. It should already be open from before. Then select **GenappExtended-Java** and then **Build** > **Pipelines**



Then examine each step in the latest pipeline that was just automatically executed based on your Java change that you just pushed in to Git Lab.

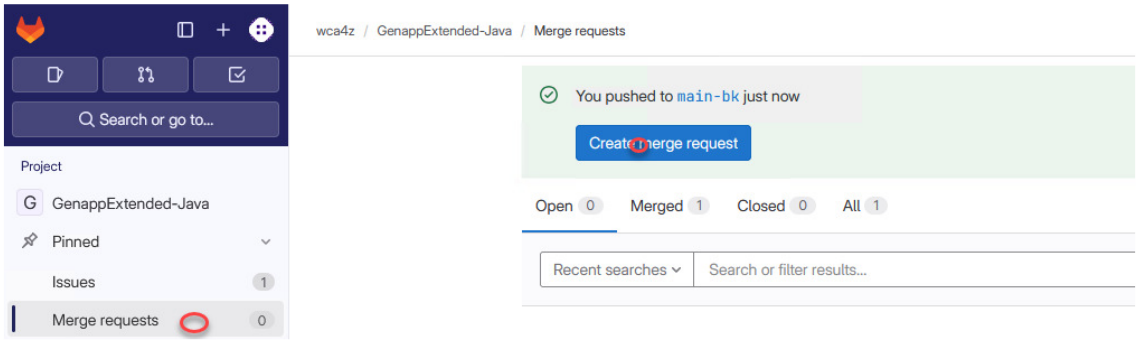


IF THE BUILD FAILED, THEN YOU MADE A MISTAKE IN YOUR CODE CHANGE (IF YOU MADE THE CHAGE YOURSELF).

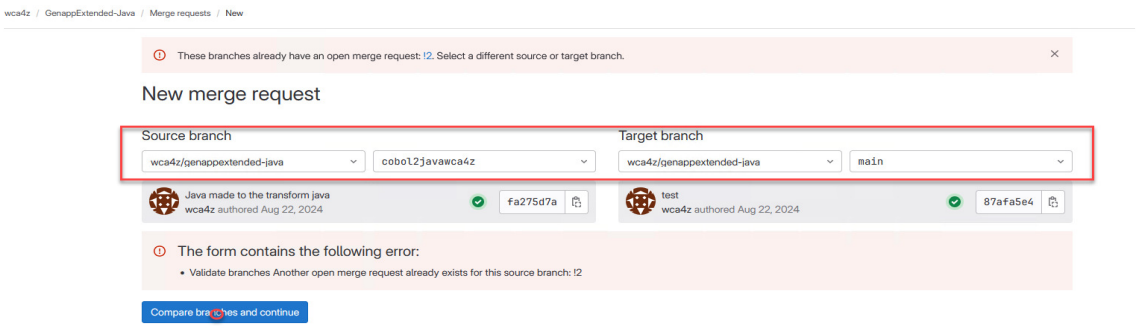
GO BACK TO THE CODE CHANGE SECTION AND DOUBLE CHECK AND ENSURE YOU HAVE MADE THE APPROPRIATE CHANGES. ASK AN INSTRUCTOR FOR HELP, IF NEEDS BE.

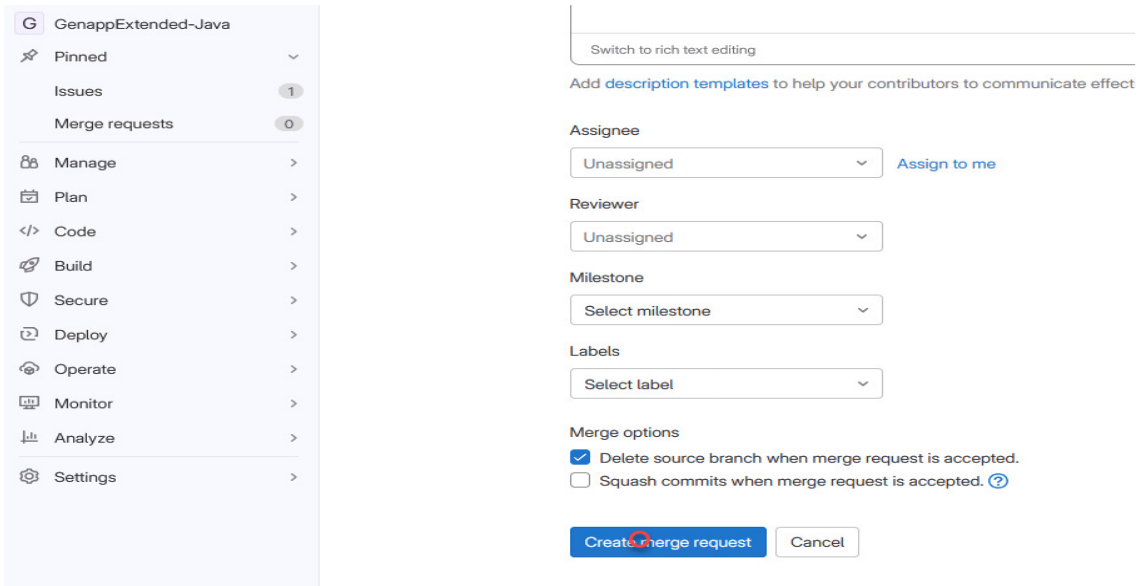
ELSE, IF YOU MADE DIDN'T MAKE THE CODE CHANGE AND USED THE PRECOOKED JAVA AND YOU HAVE A FAILURE, PLEASE CONTACT YOUR INSTRUCTOR.

Now we will merge your branch with the main/integration branch. Navigate to '**Merge Request'** and click "**Create Merge Request**".



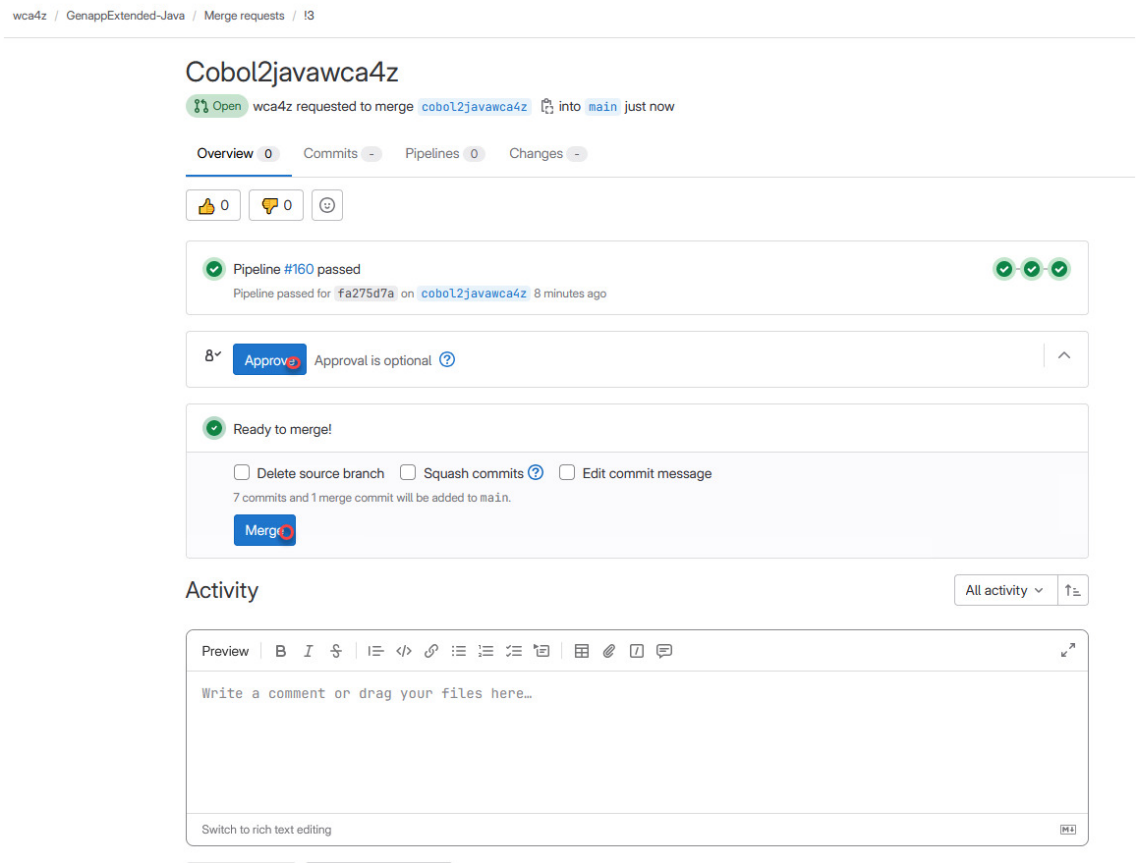
Set these branches as the **Source** and **Target** branches before clicking "Compare branches and continue".



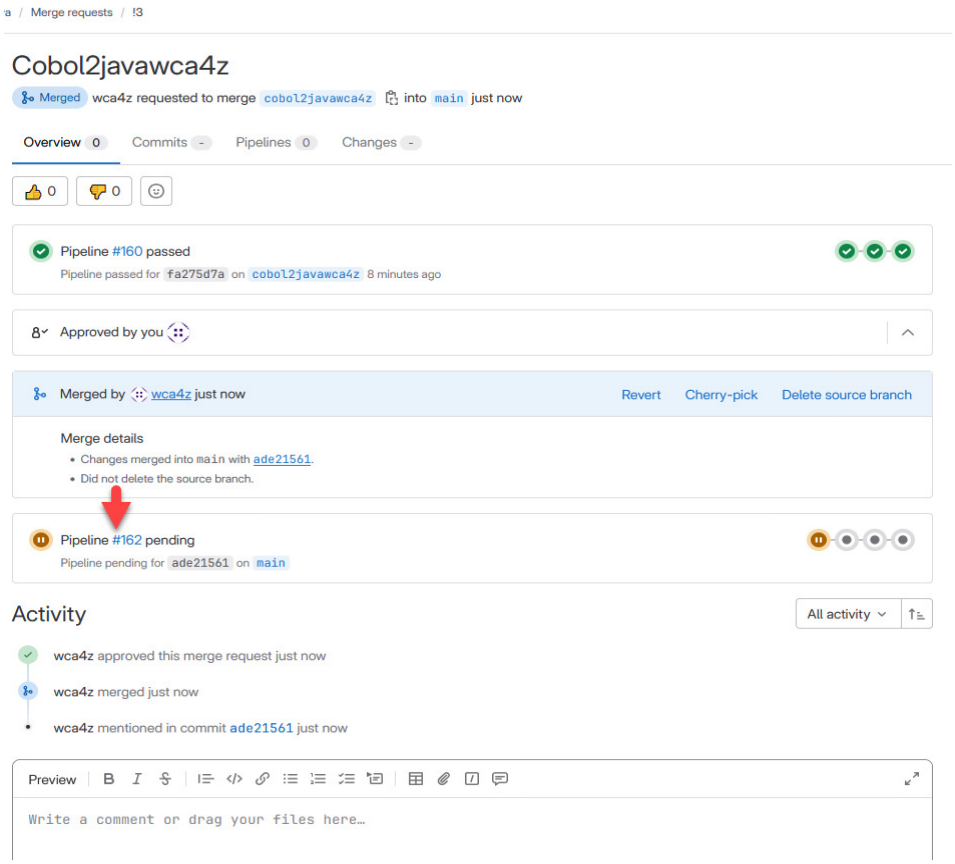


Then select '**Approve**' and '**Merge**' (**see red circle below**) once completed.

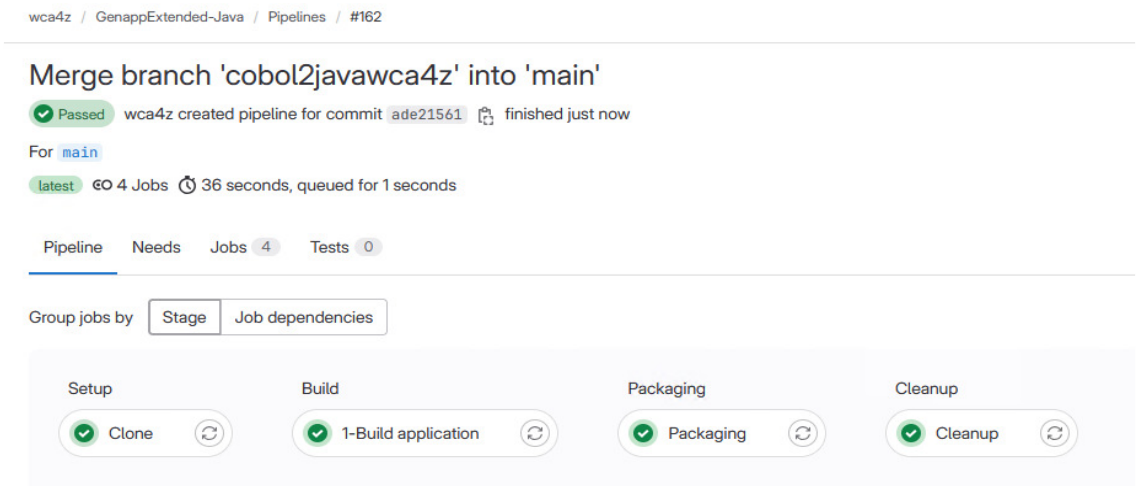
Uncheck the "Delete source branch" option.



Then **double click** the **pipeline execution number** (your number may be different) to view the release pipeline that will now build and package your Java code, ready to be deployed via the route-to-live.



Follow the steps until its completed. Feel free to explore each step and notice that compared to the developer pipeline, we are packaging to IBM DevOps Deploy.



The Java is now packaged and ready for the QA process.

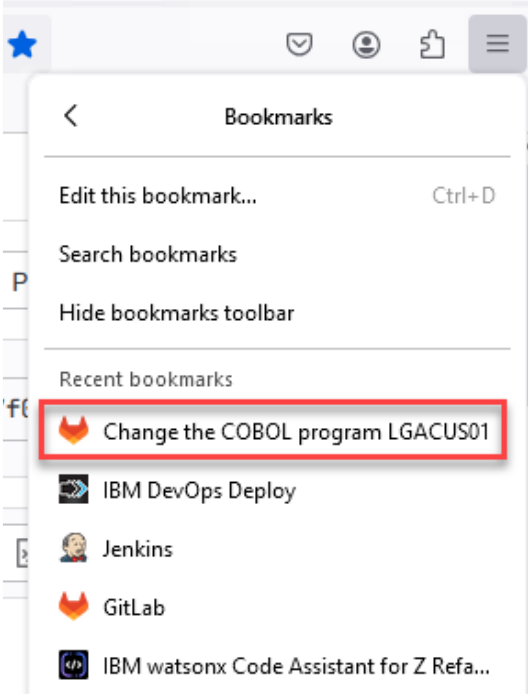
Now, let’s change the COBOL CICS to call the Java and execute the COBOL CICS pipeline

Now let’s make a change to the COBOL using the Git Lab embedded IDE

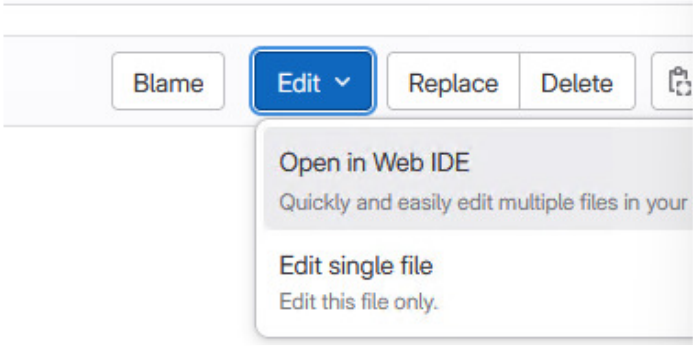
Navigate to this link to access the COBOL program LGACUS01 on the production ‘main’ branch. We will do something unorthodox and change the program directly on the main branch.

http://wca4z-gitlab.ibm.com:10880/wca4z/GenappExtended/-/blob/main/cics-genapp/base/src/COBOL/LGACUS01.cbl?ref_type=heads

This link is also in the Bookmark menu in Firefox

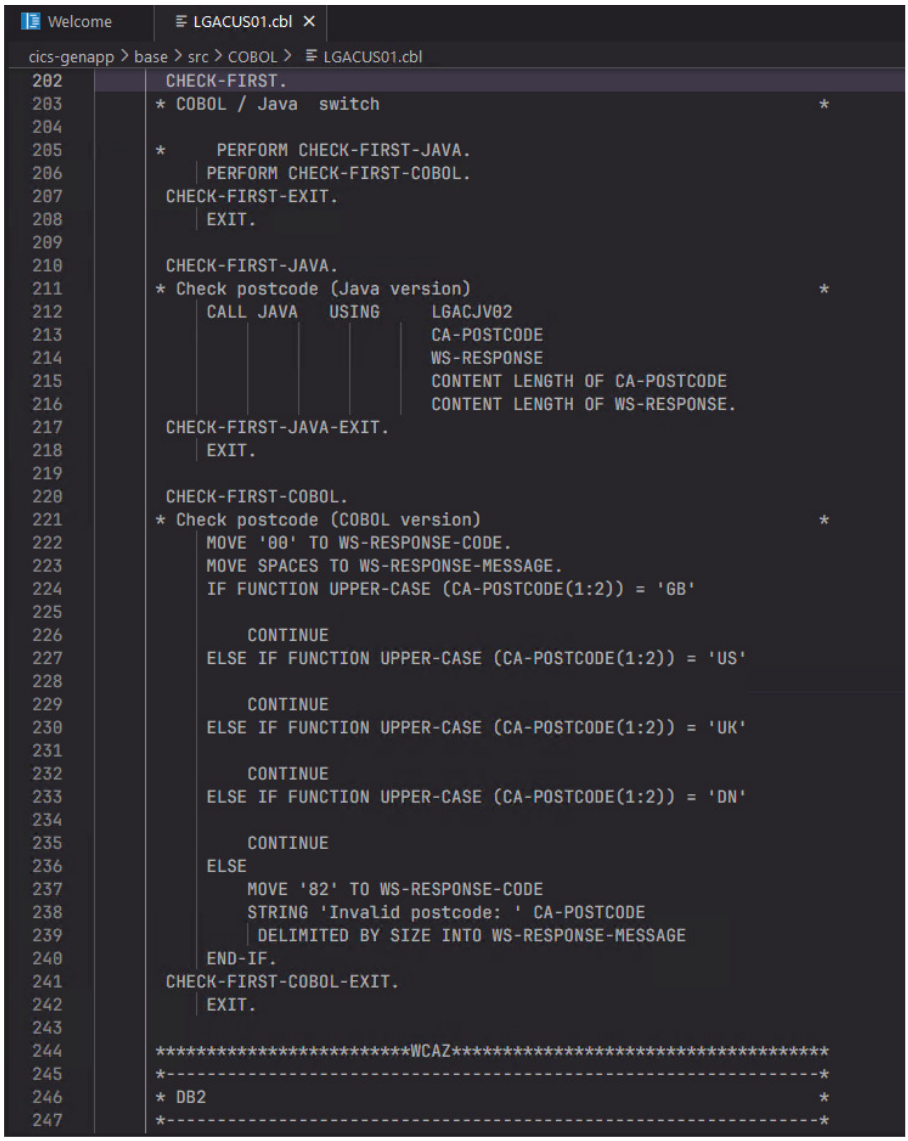


Then navigate to ‘Edit’ and the ‘Open in web IDE’, where you will change the code in the inbuilt web IDE.

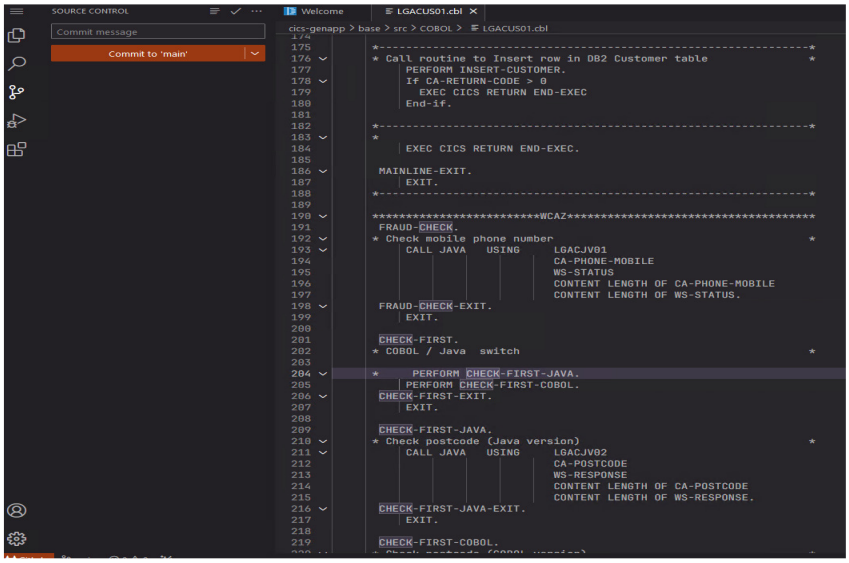


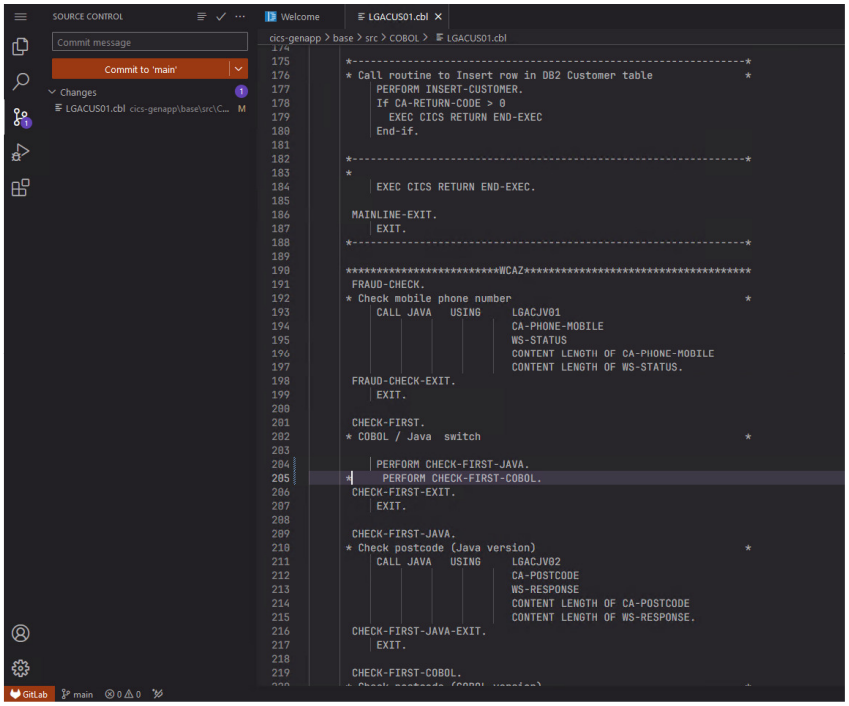
Then **remove the comment (*)** from **line 204** and place a comment in **line 205** in the same column. You should change it to this:

```
      PERFORM CHECK-FIRST-JAVA.
*     PERFORM CHECK-FIRST-COBOL.
```

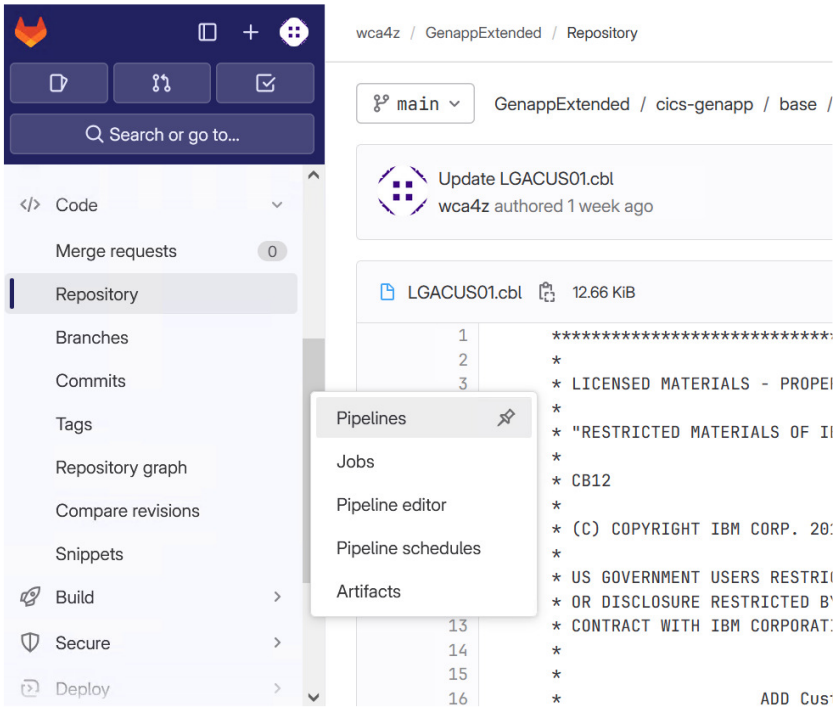


Select the Git tab on the left and then “Commit to main” in the orange to save change to production ‘main’ branch. When prompted select “Continue”.

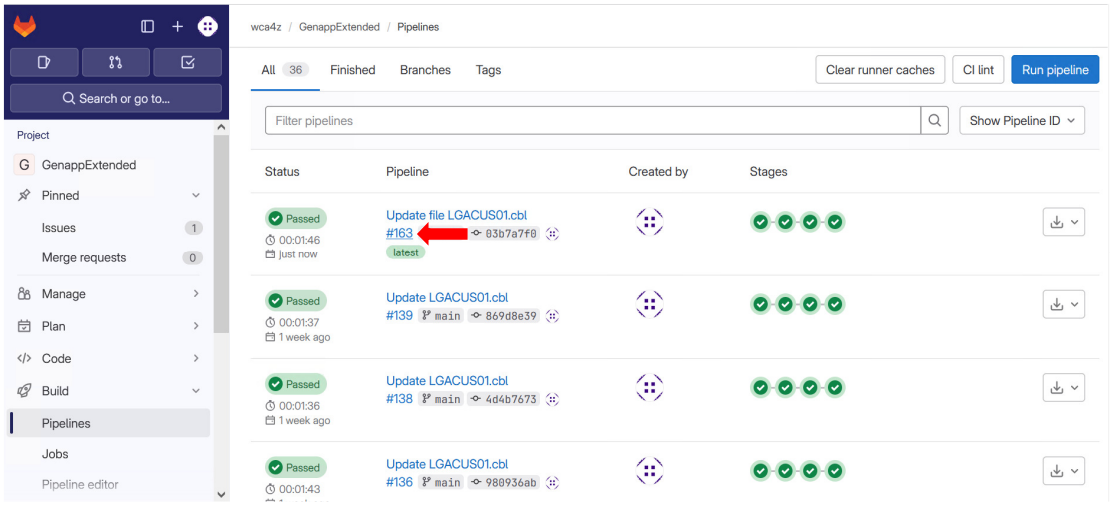




Go back to Git Lab and navigate to this link http://wca4z-gitlab.ibm.com:10880/wca4z/GenappExtended and then go to "**Build**" and then "**Pipelines**".

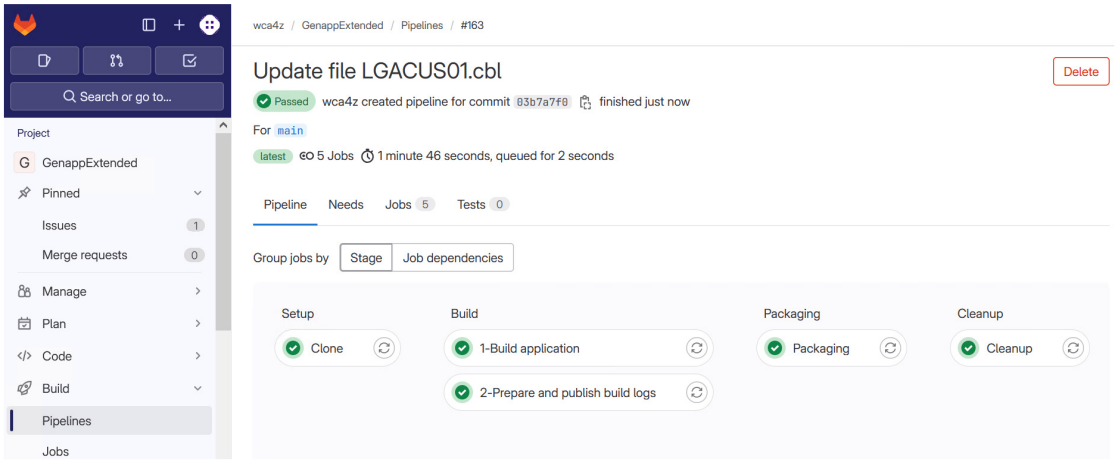


Then click on the latest pipeline number link that was executed from your COBOL change. In the picture below, this is illustrated as **163 (your pipeline number may be different).**

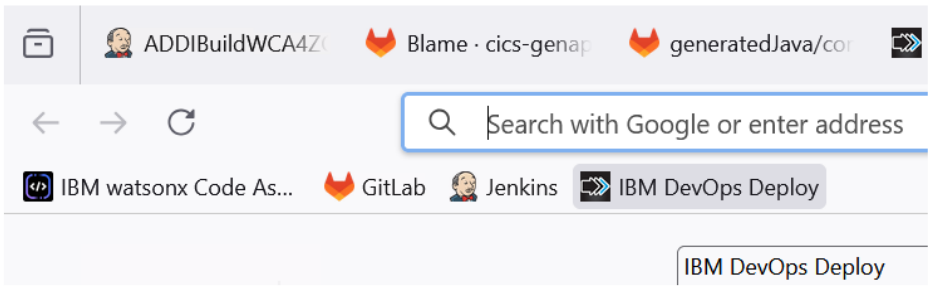


Now, let’s deploy the COBOL and Java using the Continuous Deployment process with test automation

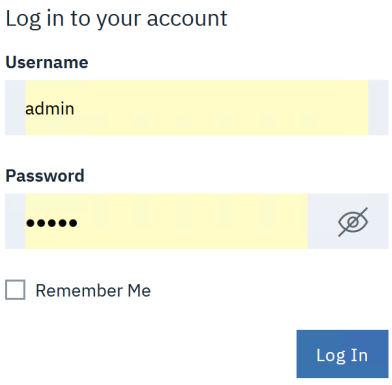
Your COBOL program has been automatically cloned to z/OS using z/OS Git, built via IBM Dependency Based Build, Packaged by IBM DevOps Deploy and you will soon deploy both the Java and COBOL together to your DevTest LPAR system using IBM DevOps Deploy.



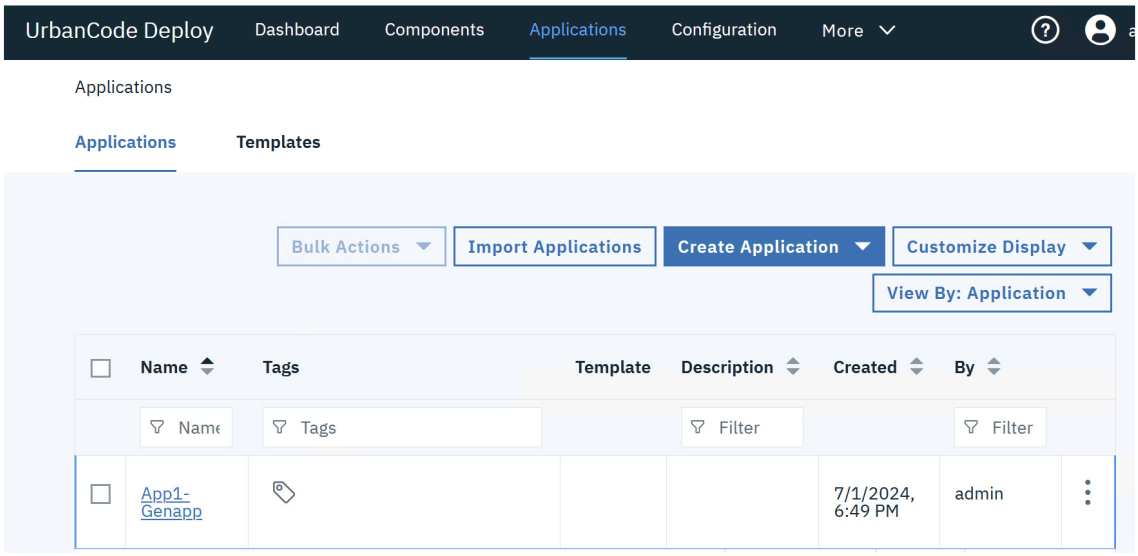
Click on ‘**IBM DevOps Deploy**’ bookmark in Firefox



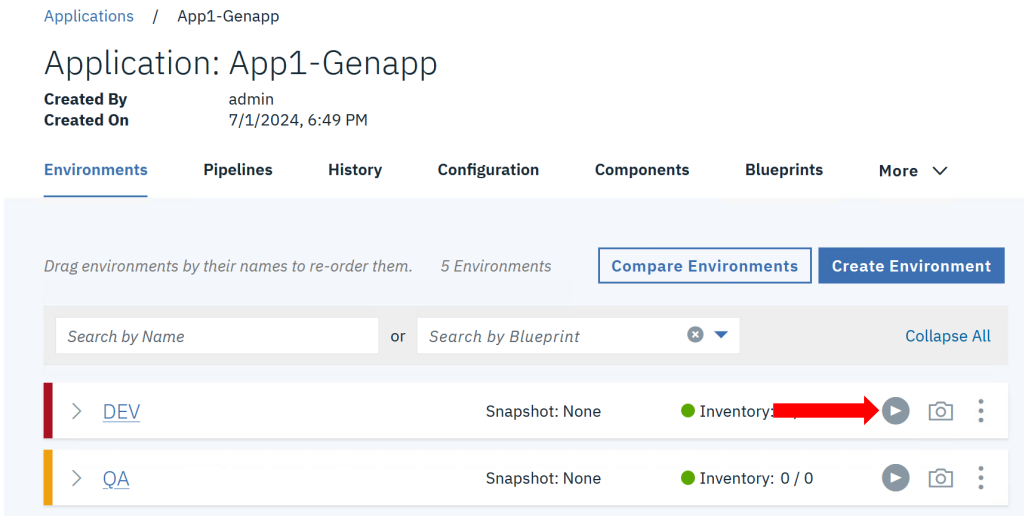
Login with **admin** / **admin** to IBM DevOps Deploy. The ID and password should already be saved.



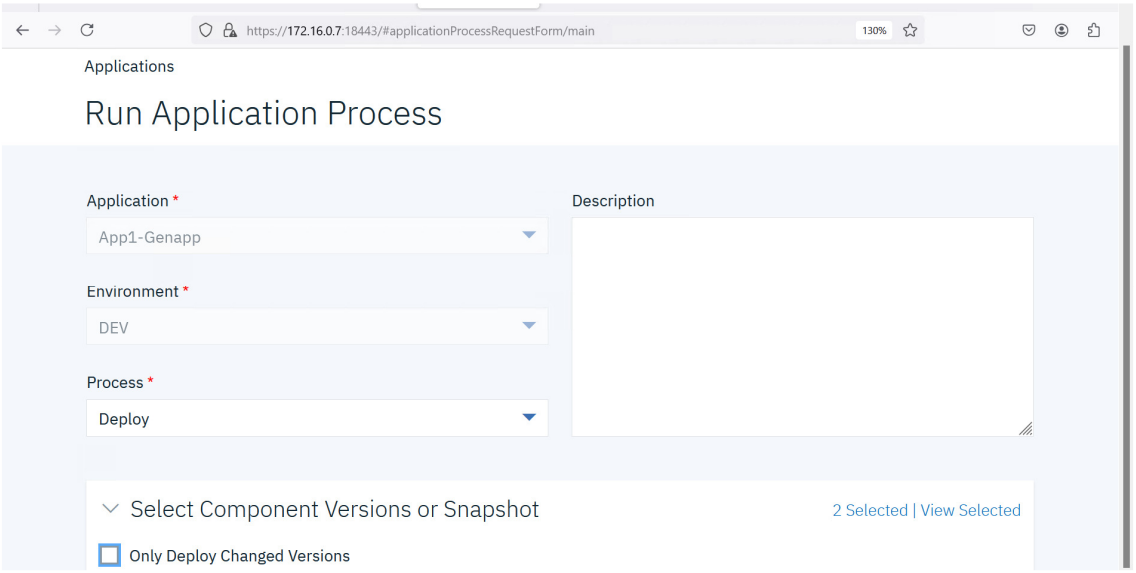
Navigate to “**Applications**” and select “**App1-Genapp**”.



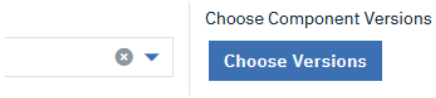
Then select the “play” button next to the DEV environment, see in illustration below.



Select the following settings and **uncheck** “Only Deploy Changed Versions”.

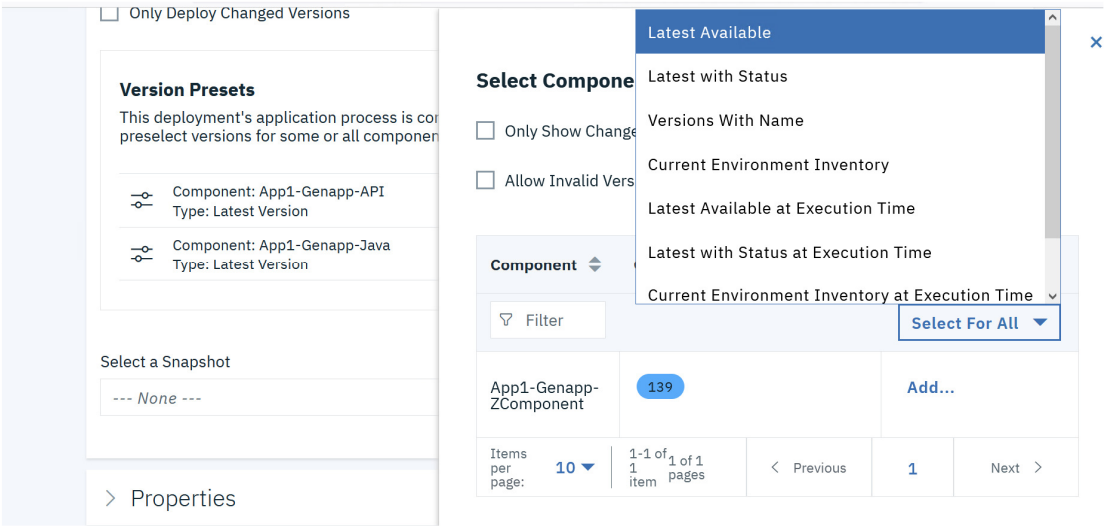


Then select ‘**Choose Versions**”.

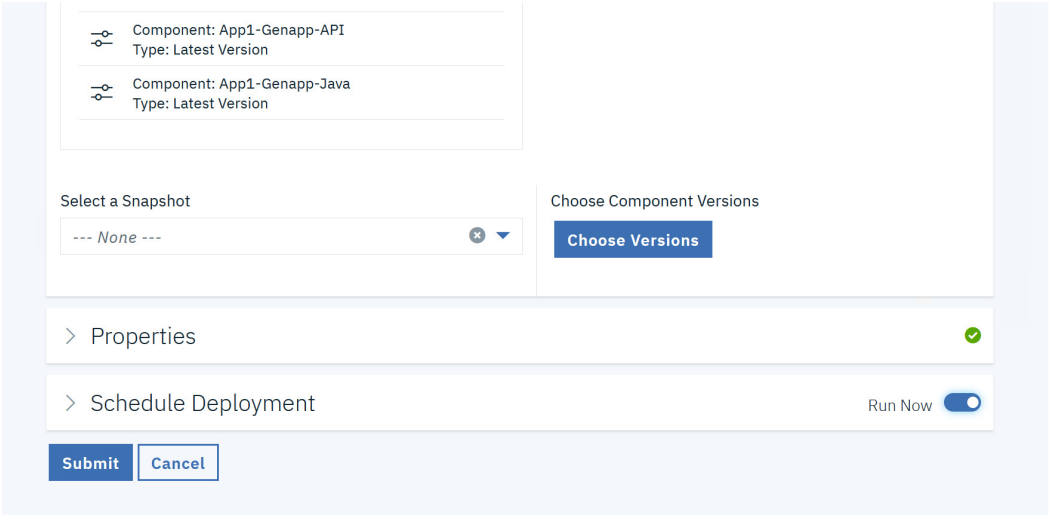


And select the **Latest Available**.

Note: As you are the only one using the system, the latest will be your last COBOL and Java package will be selected for deployment. Alternatively, you could select the package number, which will be the same number as your pipeline number. E.g. if your Java pipeline was 129 and COBOL pipeline was 130, you would select 130 for the COBOL version/package and 129 for the Java version/package.

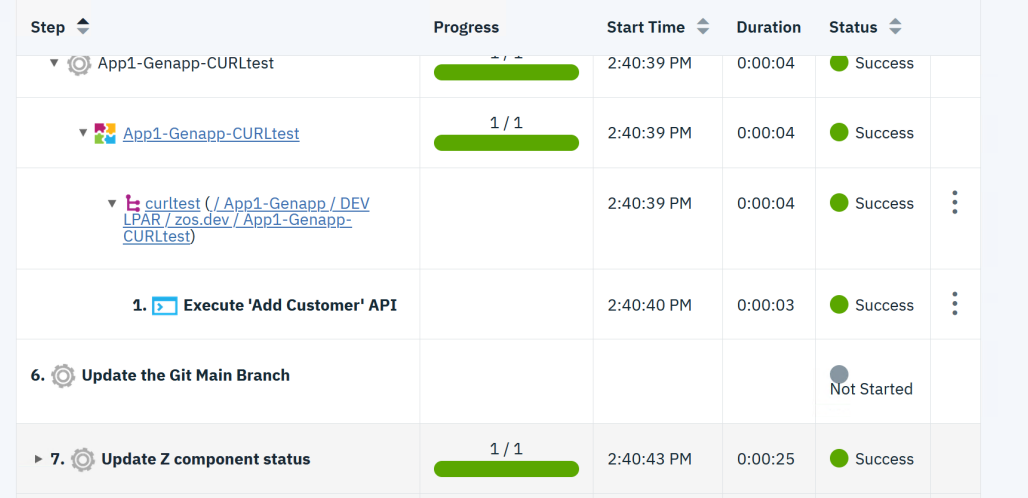


Now select “Submit” to trigger the deployment of your COBOL and Java to CICS on z/OS to the DevTest environment to be tested.



Follow the steps in the pipeline, you may need to select ‘expand all’ at the top. Then select the ‘3 dots/ellipses for each step to explore what is going on.

There is an automated test in the pipeline which will call the AddCustomer z/OS Connect API to add a new customer with a postcode UK to validate the Java is being called from COBOL CICS! If there is a HTTP code 200, then the WCA4Z Java will have been successfully transformed and integrated into CICS, with the new functionality, which will result in a ‘Success’.



In this **Changed Transformed Java and COBOL to call Java and trigger DevOps Pipeline to deploy to COBOL CICS** phase,

1. We changed the Java to run in CICS for this specific use case
2. We changed the COBOL to call Java in CICS
3. We triggered CI process for the Java
4. We triggered CD process for the COBOL and Java.

Lab completed

Appendix A – Java and COBOL interoperability in CICS

CICS Transaction Server for z/OS is a multi-language application server, and provides a simple API to allow programs written in different languages to easily interoperate with each other. The API commands relevant to this lab are described in the sections below, and the full API is documented in the IBM CICS Documentation here -> https://www.ibm.com/docs/en/cics-ts/6.x?topic=application-development-reference

CICS API commands used in this Lab

For the purposes of this lab, we will limit the discussion to COBOL and Java, but all programming languages supported by CICS can make use of these API commands.

CICS LINK Command

EXEC CICS LINK PROGRAM(<program-name>)

This command allows a COBOL program to invoke another program written in any language. Java methods can be annotated with CICS program names, making it easy to invoke specific Java methods from COBOL. Similarly, Java programs can invoke COBOL programs using the **link** method provided with the **Program** class; for example, to LINK to a program with the name **PROG01**

```
Program prog = new Program();
prog.setName("PROG01");
prog.link()
```

CICS Container API Commands

EXEC CICS PUT CONTAINER(<container-name>) FROM(<data-area>)

EXEC CICS GET CONTAINER(<container-name>) INTO(<data-area>)

These commands allow programs to exchange data using CICS containers.

For example, before invoking a target program, a calling program might place data in an input container using

EXEC CICS PUT CONTAINER(“INPUT”)

The target program can retrieve the input data using

EXEC CICS GET CONTAINER(“INPUT”)

and place the response in a different container, for example

EXEC CICS PUT CONTAINER(“OUTPUT”)

When control returns to the calling program, it can retrieve the response using

EXEC CICS GET CONTAINER (“OUTPUT”)

Similarly, Java programs can exchange container data with COBOL programs using the **getContainer** and **createContainer** methods provided with the **Channel** class; for example:

```
Container input;
Container output;
input = currentChannel.getContainer("INPUT");
output = currentChannel.createContainer("OUTPUT");
```

The CICS Container API commands can also handle code page conversion automatically. For example, EBCDIC character data stored in a container by a COBOL program can be retrieved by a Java program in UTF-16 (and vice-versa).

Appendix B – Functions provided by the lab interoperability framework

To ensure this lab remains focused on the use of **IBM watsonx Code Assistant for Z**, the programming changes required to invoke and exchange data between the newly generated Java code and the existing refactored COBOL programs have been have encapsulated into a common framework. The objective is to minimise any code changes required by the lab.

This lab-provided framework provides 3 functions:

1) COBOL subroutine to invoke a Java method

A subroutine is provided (named **"JAVA")**, which simplifies invoking a Java method from COBOL to a single CALL statement. The **CALL** takes 3 parameters:

- the name of the target Java program (the value of the **CICSProgram** annotation associated with the target method)
- the name (and length) of the data structure representing the input data
- the name (and length) of the data structure which will contain the response.

Example:

```
CALL JAVA  USING     LGACJV02                          <- Target program name
                     CA-POSTCODE                       <- Input data structure
                     WS-RESPONSE                       <- Output data structure
                     CONTENT LENGTH OF CA-POSTCODE     <- Input length
                     CONTENT LENGTH OF WS-RESPONSE.    <- Output length
```

2) Java method to retrieve input data from COBOL

A Java class (**CobolData**) and method (**getCobolData**) is provided to simplify the retrieval of input data provided by the COBOL caller.

Example:

```
import com.ibmzpot.common.CobolData;          <- the framework package
```

```
CobolData input = new CobolData();          <- instantiate a CobolData object
this.caPostcode = input.getCobolData();     <- retrieve COBOL-provided data
```

3) Java method to return output data to COBOL

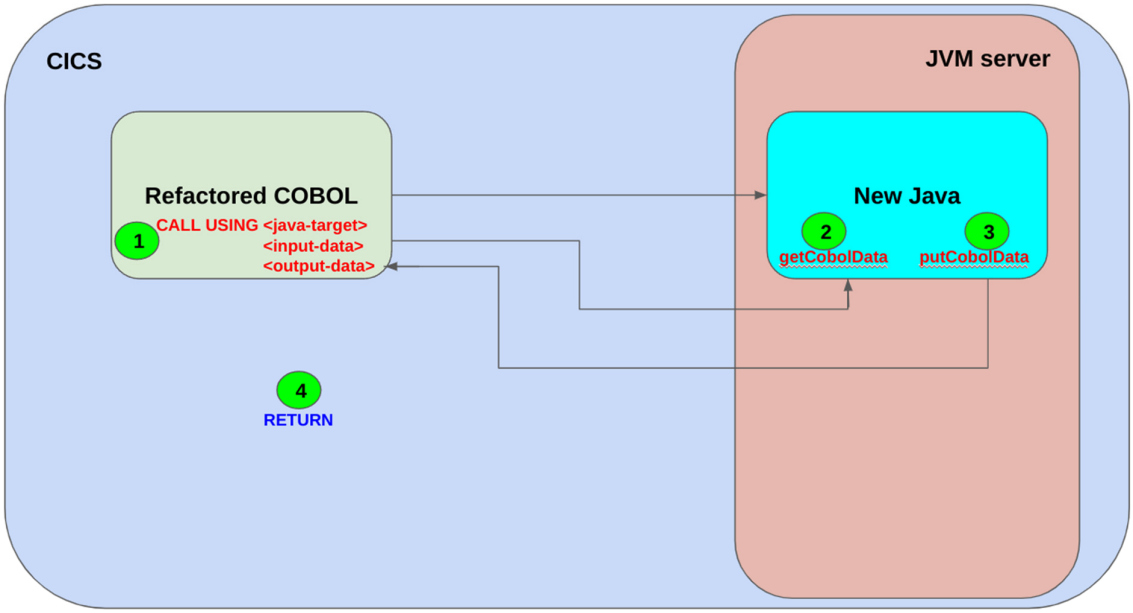
A Java class (**CobolData**) and method (**putCobolData**) is provided to simplify the returning of output data to the COBOL caller.

Example:

```
import com.ibmzpot.common.CobolData;        <- the framework package
CobolData output = new CobolData();         <- instantiate a CobolData object
output.putCobolData(responseMsg);           <- return output data to COBOL
```

An illustration and summary of how the three functions can be used together is provided below.

Summary - functions provided by interoperability framework



1. The COBOL application program indirectly invokes the Java application program by directly invoking the framework, passing the name of the target Java program, the name (and length) of the data structure containing the input data, and the name (and length) of the data structure for the response (COBOL CALL)

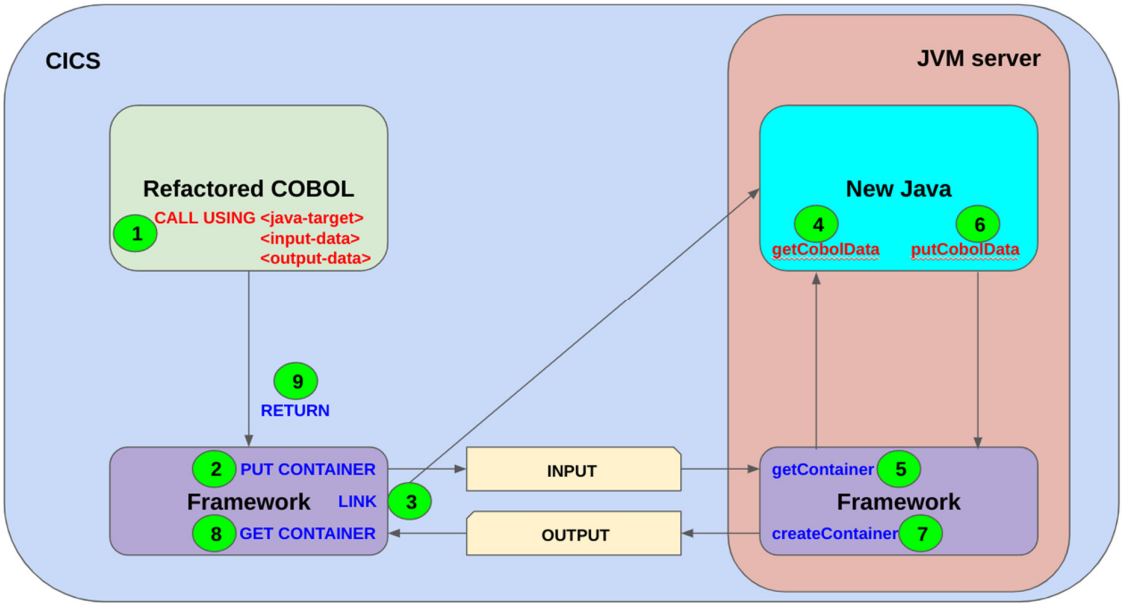
2. The Java application program retrieves the input data by invoking a method provided by the framework (framework-provided method getCobolData)

3. After the Java application program has finished executing its business logic, it invokes a method provided the framework to return the response data to the COBOL application program (framework-provide method putCobolData)

4. The framework code has finished executing, resulting in an implicit RETURN back to the COBOL application program

It is not necessary to understand the framework code in order to complete this lab, however for reference, more information is provided in Appendix C (Technical Overview), Appendix D (COBOL source) and Appendix E (Java source).

Appendix C – Framework Technical Overview



1. The application program invokes the framework code, passing the name of the target Java program, the name (and length) of the data structure containing the input data, and the name (and length) of the data structure for the response (COBOL CALL)

2. The framework creates and populates a container containing the input data (EXEC CICS PUT CONTAINER)

3. The framework invokes the target Java program (EXEC CICS LINK)

4. The Java application invokes the framework to retrieve the input data (framework-provided method getCobolData)

5. The framework retrieves the input data from the input container (CICS-provided method getContainer)

6. After the Java application program has finished executing its business logic, it invokes the framework to return the response data (framework-provide method putCobolData)

7. The framework creates and populates a container containing the output data (CICS-provided method createContainer

8. The Java code having finished executing, control returns to the framework program that issued the EXEC CICS LINK. It retrieves the data from the output container and stores it in the data structure for the response (EXEC CICS GET CONTAINER)

9. The framework finishes executing resulting in an implicit RETURN back to the COBOL application program

Appendix D – Framework Source Code (COBOL)

```
*****************************************************************
* Proxy for invoking a Java program from COBOL in CICS.
* NB. Only intended for demo purposes and not for general use.
*****************************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. JAVA.
 ENVIRONMENT DIVISION.
 DATA DIVISION.

*****************************************************************
 WORKING-STORAGE SECTION.
 01  WORK-FIELDS.
   05  JAVA-CHANNEL                PIC X(16).
   05  INPUT-CONTAINER    PIC X(16).
   05  OUTPUT-CONTAINER            PIC X(16).
   05  MAX-LENGTH                  PIC S9(9) COMP.

*****************************************************************
 LINKAGE SECTION.
 77  JAVA-PROGRAM           PIC X(8).
 01  JAVA-INPUT.
   05  FILLER          PIC X OCCURS 1 TO 999999999
                                  DEPENDING ON JAVA-INPUT-L.
 01  JAVA-OUTPUT.
   05  FILLER          PIC X OCCURS 1 TO 999999999
                                  DEPENDING ON JAVA-OUTPUT-L.
 77  JAVA-INPUT-L              PIC S9(9) COMP.
 77  JAVA-OUTPUT-L                PIC S9(9) COMP.
*****************************************************************
```

```
*****************************************************************
 PROCEDURE DIVISION USING
                   JAVA-PROGRAM
                   JAVA-INPUT
                   JAVA-OUTPUT
                   JAVA-INPUT-L
                   JAVA-OUTPUT-L.

*****************************************************************
 MAIN-PROCESSING SECTION.
   MOVE 'COBOL2JAVA'    TO JAVA-CHANNEL.

   IF JAVA-INPUT-L > ZERO
     PERFORM PUT-INPUT-CONTAINER
   END-IF.

   EXEC CICS LINK PROGRAM(JAVA-PROGRAM)
      CHANNEL(JAVA-CHANNEL)
   END-EXEC.

   IF JAVA-OUTPUT-L > ZERO
     PERFORM GET-OUTPUT-CONTAINER
   END-IF.

 MAIN-PROCESSING-EXIT.
   EXIT PROGRAM.
*****************************************************************
```

```
*****************************************************************
 PUT-INPUT-CONTAINER SECTION.
   MOVE SPACES        TO INPUT-CONTAINER.
   STRING   JAVA-PROGRAM         DELIMITED BY SPACE
            '-INPUT'             DELIMITED BY SIZE
            INTO INPUT-CONTAINER.

   EXEC CICS PUT CONTAINER(INPUT-CONTAINER)
      CHANNEL(JAVA-CHANNEL)
      FROM(JAVA-INPUT)
      CHAR
   END-EXEC.

 PUT-INPUT-CONTAINER-EXIT.
   EXIT.
*****************************************************************


*****************************************************************
 GET-OUTPUT-CONTAINER SECTION.
   MOVE SPACES        TO OUTPUT-CONTAINER.
   STRING   JAVA-PROGRAM         DELIMITED BY SPACE
            '-OUTPUT'            DELIMITED BY SIZE
            INTO OUTPUT-CONTAINER.
   COMPUTE MAX-LENGTH    = JAVA-OUTPUT-L.

   EXEC CICS GET CONTAINER(OUTPUT-CONTAINER)
      CHANNEL(JAVA-CHANNEL)
      INTO(JAVA-OUTPUT)
      FLENGTH(MAX-LENGTH)
   END-EXEC.

 GET-OUTPUT-CONTAINER-EXIT.
   EXIT.
*****************************************************************
```

Appendix E – Framework Source Code (Java)

```
package com.ibmzpot.common;

import com.ibm.cics.server.Task;
import com.ibm.cics.server.Channel;
import com.ibm.cics.server.Container;
import com.ibm.cics.server.CicsConditionException;


public class CobolData {

   Task currentTask = Task.getTask();
   Channel currentChannel;
   String currentProgram = currentTask.getProgramName();

   public String getCobolData() {

      Container inputContainer;
      String inputData = null;

      try {
            currentChannel = currentTask.getCurrentChannel();
            if (currentChannel != null) {
               inputContainer = currentChannel.getContainer(currentProgram + "-INPUT");
               if (inputContainer != null) {
                  inputData = inputContainer.getString();
               }
            }
      } catch (CicsConditionException cce) {
         throw new RuntimeException(cce);
      }

      return inputData;
   }
```

```
    public void putCobolData(String outputData) {

      Container outputContainer;

      try {
        currentChannel = currentTask.getCurrentChannel();
            if (currentChannel != null) {
              outputContainer = currentChannel.getContainer(currentProgram + "-OUTPUT");
              if (outputContainer == null) {
                outputContainer = currentChannel.createContainer(currentProgram + "-OUTPUT");
              }
              outputContainer.putString(outputData);
            }
        } catch (CicsConditionException cce) {
          throw new RuntimeException(cce);
        }
    }

}
```

Notices and disclaimers

U.S. Government Users Restricted Rights — use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM.

This document is current as of the initial date of publication and may be changed by IBM at any time. Not all offerings are available in every country in which IBM operates.

Information in these presentations (including information relating to products that have not yet been announced by IBM) has been reviewed for accuracy as of the date of initial publication and could include unintentional technical or typographical errors. IBM shall have no responsibility to update this information.

This document is distributed "as is" without any warranty, either express or implied. In no event, shall IBM be liable for any damage arising from the use of this information, including but not limited to, loss of data, business interruption, loss of profit or loss of opportunity.
IBM products and services are warranted per the terms and conditions of the agreements under which they are provided. The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

IBM products are manufactured from new parts or new and used parts.
In some cases, a product may not be new and may have been previously installed. Regardless, our warranty terms apply."

Any statements regarding IBM's future direction, intent or product plans are subject to change or withdrawal without notice.

Performance data contained herein was generally obtained in a controlled, isolated environments. Customer examples are presented as illustrations of how those customers have used IBM products and the results they may have achieved. Actual performance, cost, savings or other results in other operating environments may vary.

References in this document to IBM products, programs, or services does not imply that IBM intends to make such products, programs or services available in all countries in which IBM operates or does business.

Workshops, sessions and associated materials may have been prepared by independent session speakers, and do not necessarily reflect the views of IBM. All materials and discussions are provided for informational purposes only, and are neither intended to, nor shall constitute legal or other guidance or advice to any individual participant or their specific situation.

Notices and disclaimers (Continued)

It is the customer's responsibility to ensure its own compliance with legal requirements and to obtain advice of competent legal counsel as to the identification and interpretation of any relevant laws and regulatory requirements that may affect the customer's business and any actions the customer may need to take to comply with such laws. IBM does not provide legal advice or represent or warrant that its services or products will ensure that the customer follows any law.

Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products. IBM does not warrant the quality of any third-party products, or the ability of any such third-party products to interoperate with IBM's products. **IBM expressly disclaims all warranties, expressed or implied, including but not limited to, the implied warranties of merchantability and fitness for a purpose.**

The provision of the information contained herein is not intended to, and does not, grant any right or license under any IBM patents, copyrights, trademarks or other intellectual property right.

Learn more →