White Paper

Zowe CLI

DevOps with Mainframe-Driven Applications: Implementing Cross-Platform CI/CD

Description

The purpose of this whitepaper is to introduce Zowe CLI and to describe how it empowers you to take a new approach to building automated continuous delivery and continuous integration (CI/CD) pipelines for mainframe applications.

Features

Use Zowe CLI with mainframe-driven applications to eliminate manual deployment and to automate your CI/CD pipeline:

- Continuous Integration
- Continuous Delivery
- Continuous Deployment

Applications

- Zowe CLI
- CA Endevor®

Introduction

The purpose of this whitepaper is to introduce Zowe CLI and to describe how it empowers you to take a new approach to building automated continuous delivery and continuous integration (CI/CD) pipelines for mainframe applications.

DevOps automation in a purely distributed environment is a topic that has been thoroughly explored and implemented in the software industry. When your application also involves mainframe tools and code on the back-end, building an automated pipeline becomes increasingly complex. You might ask yourself the following questions:

- How do I orchestrate the events and processes that run on IBM z/OS from my off-platform automation tools?
  - Respondents to a survey by CA Technologies, a Broadcom company, said that “Automating tests for the mainframe,” and “Lack of mainframe experience,” are top challenges in developing automation.
- Do I have to be an expert in mainframe languages and distributed tools to build and orchestrate my team’s pipeline?
- How can I increase the efficiency of so many disparate tasks?

One development team at Broadcom created a sample web application named Marbles that consists of front-end code stored in GitHub and mainframe code stored in CA Endevor® Software Change Manager (CA Endevor® SCM). The Marbles application models an enterprise software product. We then created an automated integration and delivery pipeline for Marbles to help us to better identify solutions to the challenges that are common to implementing CI/CD for mainframe applications.

While developing this proof-of-concept, we used an in-house solution, Zowe CLI, to direct actions on the mainframe through a remote command line or automation server. Zowe CLI lets the team use familiar tools and scripting languages, which increase productivity and reduce the learning curve for new developers working on cross-platform applications.

In this white paper, we examine the Marbles pipeline and how we used a combination of mainframe and distributed tools to automate all stages of development, testing, and delivery.

We provide guidance on implementing a cross-platform pipeline with your own applications. We also will provide you with the sample application source code and example scripts that you can use as a starting point. We try to take a non-prescriptive approach on tools so that you can choose what works best for your environment and teams.

What Does Cross-Platform CI/CD Look Like?

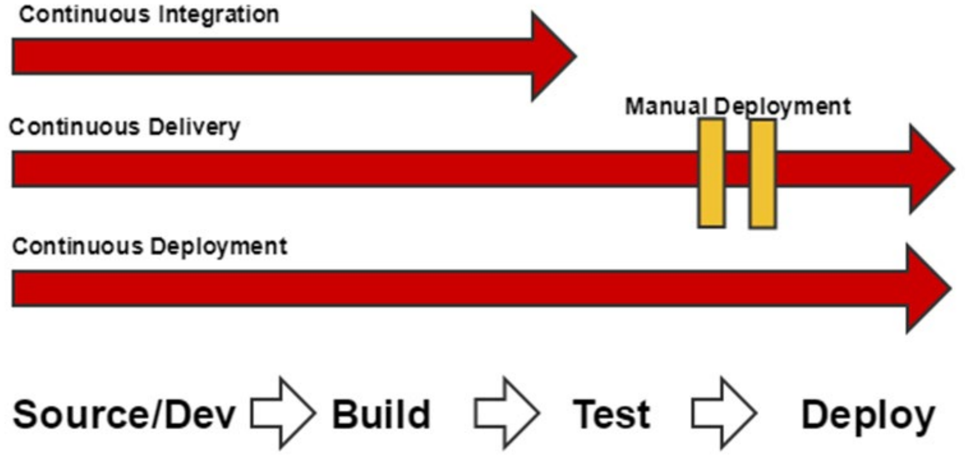
Before we describe cross-platform CI/CD, we might first define what we mean by continuous integration, delivery, and deployment.

Continuous Integration (CI)
In continuous integration, every time a developer checks in code to a source code repository, the code is automatically compiled and tested.

Continuous Delivery (CD)
Continuous delivery goes one step further than continuous integration. In continuous integration, the tested and compiled artifacts are automatically moved to an environment where end users can access them. The deployment to production is triggered manually to meet business needs.

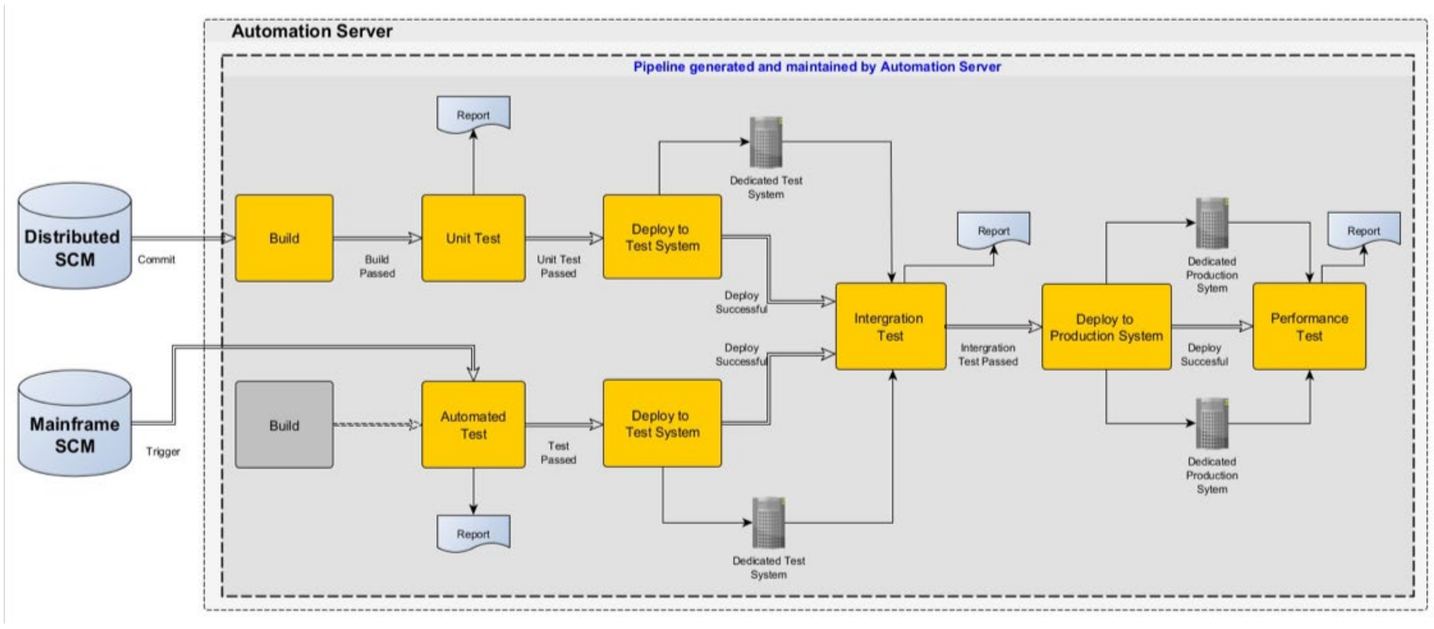
Continuous Deployment
Continuous deployment is like continuous delivery, except the artifacts are automatically deployed to production with no human intervention.

Figure 1 The following figure illustrates how, in continuous deployment, artifacts are automatically deployed to production with no human intervention.



Often, web UI servers for an application run on Windows or Linux, while the transaction processing back-end runs on the mainframe. Tying together these separate components into a single CI/CD pipeline can be a challenge. Cross-platform CI/CD lets you automate the build, test, and deployment steps across the mainframe and distributed ends of your product.

Figure 2 The following diagram illustrates one possible implementation of a cross-platform CI/CD pipeline. The example shows an automation server driving the pipeline, which interacts with both distributed and mainframe source control managers (SCMs).



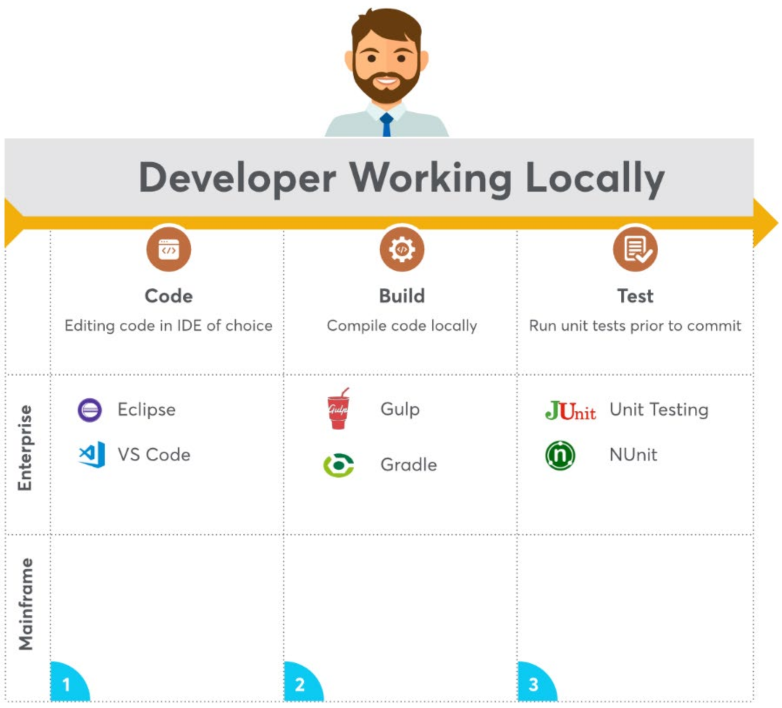
Understanding the Stages of a Cross-Platform Pipeline

You might be familiar with some pieces of the following CI/CD workflow, but we want to highlight the stages of an end-to-end pipeline that integrates both mainframe and distributed code bases.

Code-Build-Test Stage

The code-build-test stage represents the development work that occurs before an automated pipeline starts. An application developer writes code (local or mainframe), compiles the code, and runs tests against the code.

Figure 3 The following figure shows a developer working locally to manually code, build, and test an application.

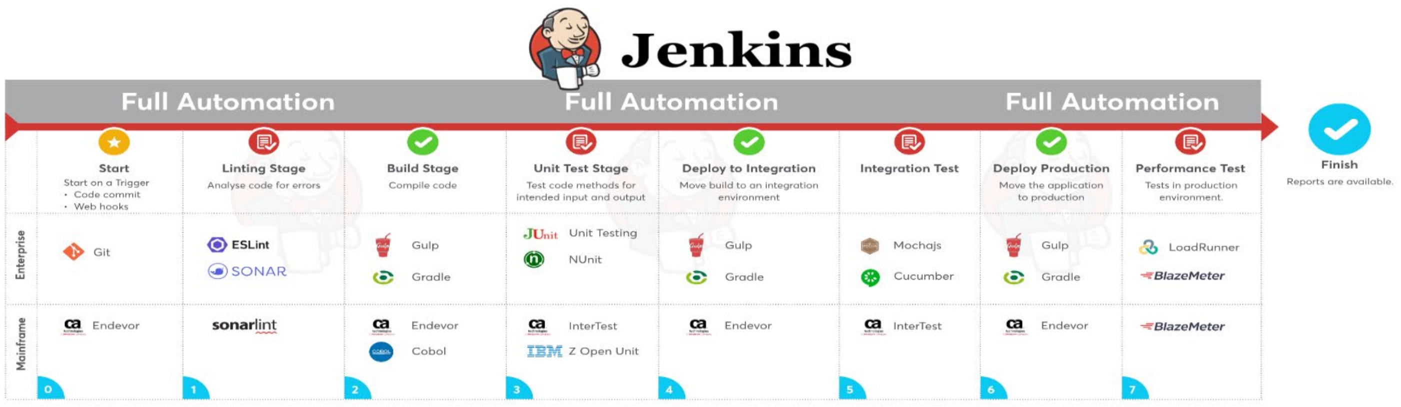


In this code-build-test stage, you can use long-standing mainframe and distributed environments to test your application manually. When you are ready to introduce your code changes into the pipeline, you interact with the Source Control Management (SCM) tool that you use at your site.

Source Code Check-In Triggers the Pipeline

When you check in source code to your SCM, an automation server initiates the CI/CD pipeline. For example, a Jenkins automation server might initiate the CI/CD pipeline. Or, if you use GitHub, a push to a remote branch or merge of a pull request into a high-level branch can initiate the CI/CD pipeline. For a mainframe SCM product, such as CA Endevor® SCM, a promotion of source code from the development stage to the QA stage can start the CI/CD pipeline. After the automated pipeline starts, a failure during any stage of the pipeline prevents later stages of the pipeline from executing.

Figure 4 The following figure shows what the stages of the full automation process might look like from start to finish for both Enterprise and Mainframe platforms.



Build Stage

After a source code promotion starts the pipeline, the pipeline builds the source code and moves it to an environment where tests can run. The build stage of the pipeline marks the first point in the CI/CD process where no further input is required from the developer.

The mechanisms for building distributed and mainframe source code differ. However, both mechanisms follow the same general process:

1. The build-scripts access the promoted source code.
2. The build-tools build the source in a location from which it can be unit tested (for example, a Linux testing environment). CA Endevor SCM uses generators to build the mainframe source.
3. If the build completes, then the pipeline continues. If the build fails, then the pipeline stops; and the automation server reports a failure.

Automated Test and Unit Test Stage

In this stage, the pipeline runs a set of automated tests and a set of unit tests against the compiled code. You might have run some tests manually during the code-build-test phase; however, now the pipeline automates the process on the front-end and back-end of the code base. The automated test stage and unit test stage ensures that all code changes are properly tested early, before the changes continue through the pipeline. Any failures in these tests result in a failure of the pipeline. The test results are saved to a report for later review.

Deploy to Test System Stage

After a pipeline successfully completes the automated test and unit test stage, the pipeline creates an environment that contains all systems and resources that are required for the application to run. The objective in this stage is to produce a new, clean environment for integration tests. Automatically creating a new, clean environment helps to avoid the delays that are associated with manually setting up testing environments. If the deployment stage fails, the pipeline does not move on to the integration test stage.

Deploy Artifacts

The pipeline deploys artifacts from the build onto a test system. Artifacts can include items such as binary files, load libraries, JavaScript, HTML files, and WAR files. The pipeline could deploy artifacts to a separate Windows or Linux test machine, to a dynamically created virtual machine, or to a mainframe system.

Provision Resources

In addition to moving artifacts, the deployment stage provisions resources and subsystems for the application. The provisioning might be partial; however, the ideal result is to recreate the entire environment for each deployment.

Integration Test Stage

After the distributed and mainframe artifacts are successfully deployed to test systems, the pipeline runs integration tests on the entire application in the new environment. The integration test stage performs the following operations:

1. Confirms that artifacts from the deploy-to-test-system stage were created. If the artifacts were not created, the pipeline fails.
2. Runs automated integration tests on the application.
3. If the automated tests succeed, the pipeline deprovisions the dynamically created test systems. If the tests fail, the pipeline leaves all subsystems intact to allow for debugging.

   Alternatively, you can configure the deployment and integration stage so that the systems stay intact after a successful integration test. On every subsequent run of the pipeline, the test systems are deprovisioned and immediately provisioned again to provide a clean testing environment.

Deploy to Production Stage

The deploy-to-production stage involves moving the fully tested application into a production environment where the end user can access it. This stage can be triggered by the pipeline; however, it is likely that you want to perform manual QA tests before you overwrite production software.

When you are ready to move your application into a production environment, you can trigger a deploy-to-production task manually. The task moves the artifacts from the deploy-to-test-system stage into production systems.

Performance Test Stage

In the performance test stage, tests are run while the application runs in the live production environment. The pipeline automatically produces reports that provide test results to the developers.

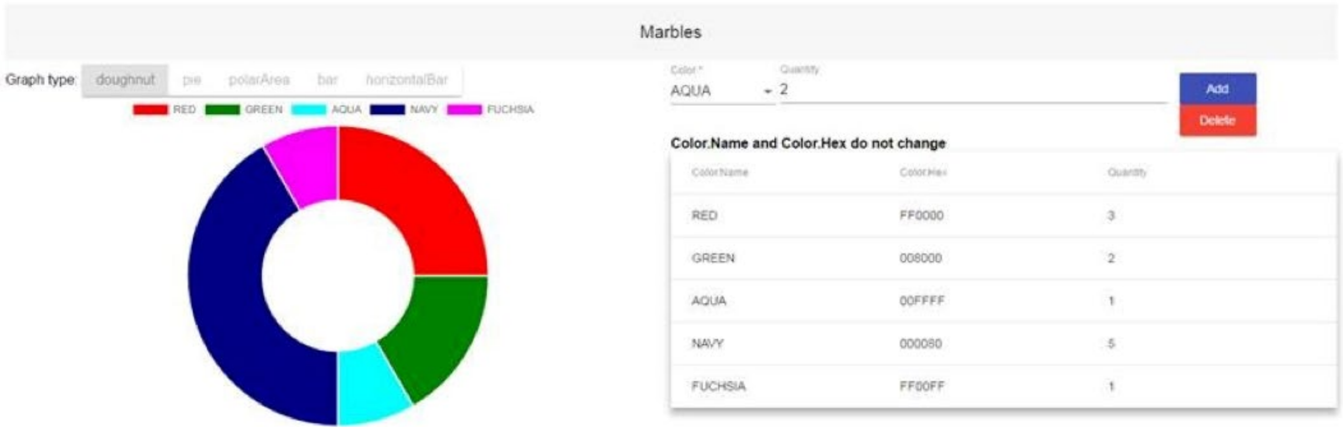
Reporting Stage

After the pipeline is complete, you can review reports. The ideal CI/CD pipeline creates reports for unit and integration tests, deployment, and more. Reporting is highly configurable depending on the types of reports that you require. As shown in the cross-platform diagram (**Figure 2**), reports can be generated at various pipeline stages.

Marbles: A Proof of Concept Application

We created an application called Marbles to simulate a full-featured software application. The application consists of both mainframe and distributed code and it interacts with various systems and resources to function. Simulating the development process with Marbles let us focus on solving problems and developing the CI/CD pipeline itself.

Figure 5 Marbles is a simple application that tracks the quantity and color of marbles in a bucket. The application displays as a GUI in a web browser where users can see and interact with a visual representation of how many marbles there are of each color: red, green, aqua, navy, and fuchsia.


Marbles
Graph type: doughnut pie polarArea bar horizontalBar
🟥 RED 🟩 GREEN 🟦 AQUA 🟦 NAVY 🟪 FUCHSIA
Color * AQUA
Quantity 2
Add
Delete

Color.Name and Color.Hex do not change

Color.Name	Color.Hex	Quantity
RED	FF0000	3
GREEN	008000	2
AQUA	00FFFF	1
NAVY	000080	5
FUCHSIA	FF00FF	1

The Marbles application uses the following systems to function:

- **IBM Db2 Database**
  Stores the data to track the color and quantity of marbles.
- **IBM CICS Transaction Server for z/OS**
  Handles input from the user to change the quantity or color of marbles in the database.
- **IBM WebSphere Application Server (WAS) Liberty**
  Hosts the code that is contained in a deployed Web application ARchive (WAR) file so that users can access the application at a given URL.
- **IBM WebSphere MQ**
  Handles the communication between WAS Liberty and the CICS region that handles transactions.

Choosing Tools

The choice of tools for a CI/CD pipeline is highly open-ended. We use the following types of tools in our cross-platform CI/CD pipeline. But you should feel free to start with the tools that you already have or are interested in using and then build up from there.

The Marbles CI/CD lifecycle model uses a Jenkins automation server to control the source control tools and the build tools in the process. Jenkins orchestrates the flow of the pipeline. Similar tools include TeamCity, GoCD, Bamboo, and more.

Remote Mainframe Interaction Tools

Actions on the mainframe typically do not activate operations on distributed systems. Also, many tools exist for distributed code that do not effectively control build, test, and deliver activities on the mainframe. To bridge that gap, our pipeline relies heavily on the Zowe CLI.

Zowe CLI is a command-line tool that is a primary enabler for the automation of our pipeline. In fact, Zowe CLI was primarily developed to assist with this use case. Using Zowe CLI, you can perform a variety of mainframe operations remotely. For instance, you can write scripts to automate manual mainframe tasks.

Because of this capability, the CLI lets us easily interact with the mainframe from within our CI/CD pipeline in Jenkins. Alternatively, you could write scripts that make REST calls to mainframe APIs; but this approach is more labor intensive and requires a thorough knowledge of the back-end APIs.

Source Control Tools

The distributed code base for Marbles is managed in GitHub Enterprise. In our mainframe environment, we use CA Endevor SCM.

The specific operations in your pipeline will be different if you use different version control software but the overall sequence steps should not change.

Build Tools

Build tools are used to trigger source code compilation and to trigger the automated test tools in the pipeline. We used a combination of Gradle and Gulp to create our CI/CD pipeline. In our mainframe environment, we rely on the generators that are included with CA Endevor SCM to control the build process. Similar tools include Grunt, Ant, and more.

Automated Testing Tools

We used Jasmine, Karma, and Cucumber to accomplish the automated testing steps in the pipeline. In some cases, we used handwritten scripts to act as test drivers. The CI/CD workflow does not rely on any particular tool; so, you can use the tools that work for your team and organization.

Reporting Tools

We used a plug-in for Jenkins, Blue Ocean, to simplify the view of reports within Jenkins. The unit tests, integration tests, and performance tests also produce reports that we can review after each run of the pipeline. You can also deploy reports from your build to a web server.

Source Code Editors

The choice of Integrated Development Environment (IDE) does not affect the CI/CD lifecycle. Use the IDE that you prefer. Zowe CLI lets you edit mainframe code directly from your preferred IDE while keeping the code in sync with CA Endevor SCM.

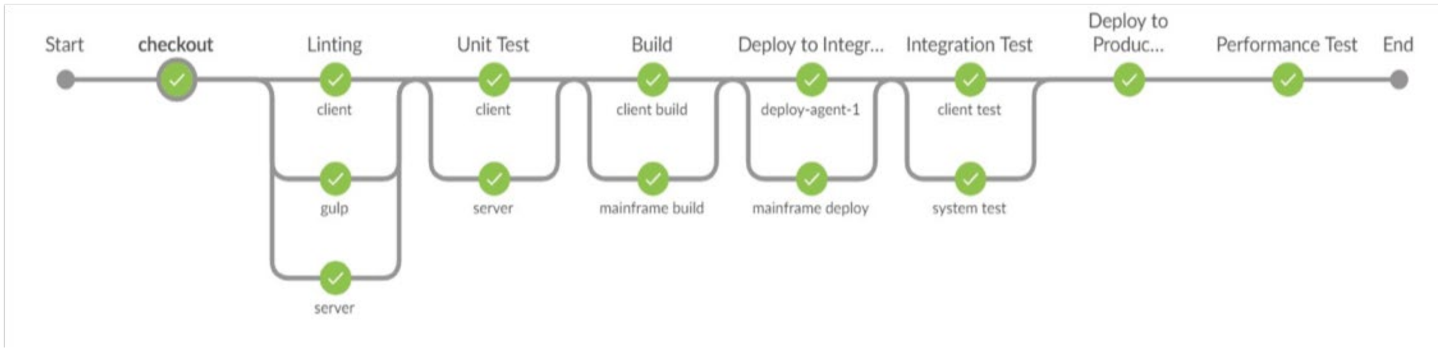
Language Compilers

The choice of compilers does not affect the CI/CD lifecycle. Scripts within the build tools control the operations to compile, link, copy, or any other operations for the programming languages that are involved in the application.

Examining the Pipeline that was Implemented for the Marbles Sample Application

In this section, we describe our implementation of each stage of the CI/CD pipeline for the Marbles sample application.

Figure 6 The following diagram illustrates the major steps in our pipeline as they are defined in the Jenkins automation server.



Code-Build-Test Phase

During the code-build-test phase, developers manually edit, build, and test the Marbles application before the pipeline is initiated.

Distributed Operations
Front-end components of the Marbles application are stored in GitHub. A developer uses Git to perform push, pull, and commit operations to retrieve and manipulate the source code.

Front-end components are built using a combination of the Gulp and Gradle build tools. A Gulp script launches Gradle tasks as required. Starting a single build task can build the entire front-end.

Mainframe Operations
We used Zowe CLI commands to interact with the mainframe; although, the same operations can be performed on the mainframe through ISPF interfaces. We edit Marbles code locally in an IDE and move the source code between a PC and the mainframe system.

Back-end components of the Marbles application are managed by CA Endevor SCM on z/OS. The CI/CD lifecycle assumes that a CA Endevor SCM environment already exists, and that elements and processors are already defined to build the source code. We assume that WebSphere MQ and Db2 are set up manually for the environment. Our pipeline automatically provisions and deprovisions WAS Liberty and CICS.

Developers perform the following operations during the code-build-test stage:

1. Create a CA Endevor SCM sandbox.
2. To automate the creation of the sandbox, we use Zowe CLI commands to submit Software Control Language (SCL) directly to CA Endevor SCM. Alternatively, you can use Zowe CLI to submit JCL that contains SCL to perform the same CA Endevor SCM operations.
3. Place elements into the sandbox.
4. Copy elements from the sandbox to the local workstation.
5. Change some source code using the text editor or IDE of your choice.
6. Build the back-end components. Use the Generate operations in CA Endevor SCM to build from the sandbox. The generators place the resulting binary files into a data set or load library from which the back-end component can be executed.
7. Perform spot-testing. If unit tests are defined, the developer can run the unit tests manually during this stage.

Source Code Check-In Triggers the CI/CD Pipeline

The CI/CD pipeline for the Marbles application is implemented using a Jenkins server. When code is committed and pushed into GitHub, or back-end code is promoted in CA Endevor SCM, the CI/CD pipeline is initiated.

We wrote Bash scripts in the Jenkins server because the Bash scripts can run without installing additional software tools. Alternatively, you could use a scripting language such Gulp or Gradle.

Distributed Operations
We defined GitHub hooks in Jenkins that kick off the distributed pipeline when a developer runs a git push to a specified remote branch.

Mainframe Operations
We start the pipeline on the mainframe side when we promote source code in CA Endevor SCM to the next stage in production. For example, our pipeline starts when source code is promoted from stage one to stage two in the Dev environment.

The CA Endevor SCM promotion action is configured to run a batch job. The batch job runs a REXX (Restructured Extended Executor) program that uses IBM z/OS Client Web Enablement Toolkit to issue the HTTP request to a URL on the Jenkins server. We configured Jenkins so that the HTTP request against the specified URL starts the mainframe pipeline.

Another option for implementing the communication between your automation server and the mainframe is CA Endevor SCM Integrations for Enterprise DevOps, which facilitates the use of platforms outside the mainframe to edit, change, and push code, and monitor events.

Linting Stage

Linting is the process of checking the static code for programmatic and stylistic errors before you introduce the changes to your code base. During the linting and unit test stages, linting of code occurs on the client, server, and Gulp code bases.

Client Code
We use Angular CLI to run TSLint, a linter for TypeScript code, against the client side.

Mainframe Code
You might use tools such as SonarCOBOL to perform linting on the COBOL code in your application. You can use Zowe CLI to easily move the source from CA Endevor SCM to SonarCOBOL.

Server Code
For linting on the server side, we use the Gradle checkstyle plug-in to check for errors in the Java code.

Gulp Scripts
We use the TSLint plug-in to perform linting on the JavaScript Gulp scripts in our pipeline.

Build Stage

In the build stage, the source is built from the mainframe and the distributed code base and saved as artifacts to prepare for integration testing.

Distributed Operations
During the build stage, the front-end code and server code is built and stored in a Web Application Resource (WAR) file for later use in the pipeline. We use a combination of Gulp and Gradle scripts to build the distributed source code and package it into a WAR file.

Mainframe Operations
CA Endevor SCM processors build the mainframe source after the code is promoted. In our model, we manually trigger a source code promotion in CA Endevor SCM (or issue a Zowe CLI command). We use the CA Endevor SCM AUTOGEN feature to regenerate all dependent components.

Before the pipeline deploys the artifacts to a test system, the pipeline uses Zowe CLI commands to confirm that the CA Endevor SCM build completed successfully. In this stage, we also deprovision an existing CICS region and provision a new CICS region. Provisioning a new region provides a clean environment for the integration testing that occurs in the following stages. The existing CICS region might have been long-standing to run manual tests prior to this run of the pipeline. The CICS region and transactions are also configured in this stage.

Unit Test Stage

We set up the pipeline to run Karma and Jasmine, a JavaScript test framework that supports behavior driven development (BDD), unit tests automatically on the separate code bases. The unit tests generate reports that you can review later. When the unit tests pass, the pipeline can continue to the next stage.

Server Code
We use the JUnit testing framework to write tests for Java server code.

Client Code
For unit tests on the client side, we use Angular CLI to run tests that are written in the Jasmine test framework. We use Karma to run the tests, which spawns multiple browsers to automatically run the Jasmine tests.

Deploy to Integration Test System Stage

The deployment to an integration test system stage is automatically triggered after the successful completion of the build stage. In this stage, the artifacts that are copied into a test environment are prepared for integration tests. We collect as much data as possible about the operations that are performed in this stage to assist with debugging.

Distributed Operations
When a change is made to the master branch of Marbles, we serve the front-end application from a WAS Liberty environment. The WAS Liberty environment is provisioned by the pipeline. For any other branch, we deploy Marbles to a long-standing WAS Liberty region. We prefer to deploy to a longstanding region in some cases to avoid time delays and resource consumption during development. We use Docker to provision Linux test environments in the pipeline.

Mainframe Operations
In the previous build stage of the pipeline, the deployment task ran a Zowe CLI command to confirm that CA Endevor SCM completed the build process. The generators in CA Endevor SCM moved the binary files to the next stage in the CA Endevor inventory map at the time of a manual promotion.

In this stage, the pipeline moves the WAR file to WAS Liberty on the test system by using Zowe CLI commands. Our pipeline deprovisions an existing WAS Liberty instance and provisions a new WAS Liberty instance for every run of the pipeline that begins with a change to the master branch.

The pipeline uses Zowe CLI to provision a CICS region in the previous build stage. WebSphere MQ and Db2 must be set up manually in advance for the integration stage of our pipeline to work; although, you can integrate automatic Websphere MQ and Db2 provisioning into your pipeline.

Integration Test Stage

For the integration testing stage of the Marbles application, our pipeline uses a test framework that considers a successful test of the front-end to be a successful test of the entire application. For example, the Cucumber UI test framework in our pipeline drives the front-end and confirms that the correct back-end data is successfully displayed. In addition to the front-end testing, we set up our pipeline to run tests on the underlying subsystems.

Scripts in Jenkins perform the following operations:

Distributed Operations
The Jenkins pipeline uses a combination of Gradle and Gulp tasks to run automated tests on the GUI with the Cucumber test framework. We convert the Cucumber report into a JUnit report so that Jenkins can track and manage our reports.

Mainframe Operations
During the integration test stage, Jenkins uses Zowe CLI commands to run system tests on the Db2 and CICS subsystems. Our pipeline uses Zowe CLI commands to issue Modify statements to a CICS region. The Modify statements execute transactions between the CICS region and the Db2 instance.

The integration tests can produce reports (as .txt or .xml files) that describe the health of the communication between the CICS region and the Db2 instance on our test system.

After the integration tests complete, the test systems stay in place until the next run of the pipeline to allow for manual testing. The Jenkins pipeline uses Zowe CLI commands to delete the data sets that were copied to the test system. During the next build stage, the test systems are automatically deprovisioned and then provisioned again to provide a new environment.

Deploy to Production Stage

Movement of the application into a production environment is not automatically triggered by the CI/CD pipeline. The entire pipeline up to this point can complete quickly; therefore, you might want to perform manual QA tests before you move the application into production.

Jenkins uses Zowe CLI commands to copy the same application files that were copied in the deploy-to-test-system stage, but now moves them to a production system. The required subsystems for Marbles (CICS, WebSphere MQ, IBM Db2, and WAS Liberty) already exist in a production system, so provisioning is not required in this stage.

Publishing Artifacts
For our Marbles sample application, we do not publish artifacts to any external repositories such as a Node Package Manager (npm) library. However, you can include scripts in your pipeline to move your artifacts to a location where end users can access them. Gulp and Gradle are examples of popular scripting tools

Building a Pipeline at Your Site

So far, we covered the basics of a cross-platform pipeline, the Marbles PoC application, and our approach on tools and implementing each pipeline stage. Now we want to provide some guidance on how you can begin to implement a similar pipeline. You might decide to experiment with our sample application and pipeline to try this yourself.

Are You a Candidate?

How do you know if you should implement a similar approach with your products? Use this simple checklist:

- Are you interested in using distributed tools like Jenkins, Gulp, Bamboo, Gradle, and others to accomplish DevOps on Mainframe?
- Do you like the idea of using a remote command line tool, such as Zowe CLI, to orchestrate mainframe tasks?
- Do you currently lack automation in your development lifecycle? Is mainframe-side automation falling behind your distributed automation?

Getting Started

If you answered yes to any of the previous questions, then this might be a great approach for you. Here are some next steps that might take to get started implementing a pipeline similar to the one we describe in this white paper:

- Learn more about and download the Zowe solution for free.
- Check out our CLI script samples. You can use the scripts for inspiration to develop your own automation.
- Start small! Build a simple pipeline that builds on the manual tasks and tools that you already have in place.

Conclusion

Implementing CI/CD for all aspects of your application can increase developer productivity, ensure quality through consistent testing, and reduce time-to-market for new applications and features. We hope that this white paper helps you to understand the concepts of CI/CD and gets you thinking about how you can implement it with your products.

Revision History

Zowe-CLI-WP100; July 15, 2019

Initial document version.