Technical Insights into Running Java Applications in CICS

Chandra Mouli Yalamanchili

chandu85@gmail.com

Abstract

Running Java applications in IBM's CICS (Customer Information Control System) transaction server enables enterprises to leverage modern application frameworks, system interoperability, and developer productivity.

This paper covers the integration of Java applications in CICS and describes the strategic value that this integration provides to enterprises wishing to modernize legacy systems. It outlines the benefits, historical evolution, technical implementation, and deployment possibilities of running Java applications in CICS environments.

This paper highlights how Java can modernize legacy mainframe applications using its portability, integration capabilities, and developer support ecosystem. Different options, such as OSGi JVM and Liberty JVM, are outlined in the article, along with their strengths, limitations, installation processes, and typical use cases.

This paper also discusses major technical features of Java-CICS integration, including the JCICS API, IBM SDK features, and CICS resource management, and their roles in enabling Java-CICS interoperability.

This paper also discusses deployment approaches, performance factors, and common problems encountered, guiding organizations towards effectively exploiting Java in mainframe-based OLTP systems.

Finally, the paper outlines future research directions to optimize Java workloads and enhance legacy interoperability.

Keywords: Java; CICS; OLTP; z/OS; JCICS API; OSGi JVM; Liberty JVM; Mainframe modernization; IBM SDK; zIIP Engines

Introduction

Online transaction processing (OLTP) is a computer system that facilitates transaction-processing applications, normally for data entry and retrieval transaction processing. OLTP systems maintain real-time transactions, which are applied within banking, communications, and sales industries where real-time processing and availability are paramount. Benefits include better data integrity, improved response time, and optimum resource utilization.

IBM Customer Information Control System (CICS) is an extremely powerful transactional processing system that is extensively utilized in the mainframe system. It supports high scalability and reliability, maintains transaction integrity, and can support quick processing of concurrent transactions.

Unlike legacy mainframe languages COBOL and HLASM, Java offers portability, ease of integration with modern distributed systems, productivity for developers, and a huge community of open-source libraries. Adopting Java on mainframes fills the gap between legacy infrastructure and cloud-native patterns as businesses incrementally transition to distributed technologies and cloud-native patterns.

This paper explains how IBM has evolved to include Java in CICS, employing zIIP processors to optimize resource usage and performance, thus providing modern solutions to traditional OLTP challenges.

1. Why use Java in CICS

Java integration with CICS enables enterprises to modernize the platform and adopt agile development to improve market speed. It bridges the gap between new service-based designs and legacy mainframe applications and provides a scalable and sustainable approach to digital transformation on z/OS. [1]

- **Modernization:** Enables legacy applications to use new architectures, frameworks, and agile development practices. Java's modularity, support for RESTful services, and microservices architecture compatibility position it advantageously to support legacy system extension. [1]
- **Developer Productivity:** Java's familiar syntax and rich tooling ecosystem (e.g., Eclipse, IntelliJ, Maven) accelerate development and testing cycles while reducing onboarding times for new developers transitioning from distributed environments. [2]
- **Integration Capabilities:** Java on CICS seamlessly connects with distributed technologies like web services, REST APIs, and messaging middleware like IBM MQ, facilitating hybrid cloud integrations and cross-platform transactions. [2]
- **Performance:** Optimizes resource usage by offloading eligible Java workloads to zIIP engines, reducing mainframe processing costs while ensuring high throughput for OLTP applications. [1]
- **Example Use Case**:

  Consider an online banking scenario where Java is utilized to develop responsive web applications interfacing with CICS-based legacy transaction logic. Java helps modernize the front end without altering stable backend systems, enhancing user experience and maintaining transactional integrity.

2. History of Java & JVM support in CICS

CICS support for Java has gone through a significant evolution over the last couple of decades, driven by growing demand from enterprises for modernization and improved interoperability:

- **Initial Support (CICS TS 1.3 and earlier):** Java support began as a basic embedded environment using Java 2 Standard Edition (J2SE), offering minimal functionality and performance. This support allowed basic servlet-like Java programs to run under CICS but lacked integration with core CICS resources. [1]
- **Introduction of JVM Servers (CICS TS 3.1+):** IBM introduced the concept of dedicated JVM servers. JVM servers allowed Java programs to run in a managed and isolated environment within the CICS region, improving performance, scalability, and resource allocation. Java integration with COBOL and JCICS APIs also became more robust. [2]

- **OSGi JVM Support (CICS TS 4.2+):** OSGi modular support enabled deploying Java components as dynamically loadable bundles, significantly improving maintainability and modularity. OSGi support allowed Java applications to adopt enterprise-level lifecycle and version control mechanisms. [2]

- **Liberty JVM Integration (CICS TS 5.1+):** CICS began supporting WebSphere Liberty Profile, offering full Java EE capabilities, RESTful services, and integration with modern frameworks. Liberty JVMs provided a powerful, cloud-compatible environment for enterprise-grade application deployment on z/OS. [2]

Today, CICS supports both OSGi and Liberty JVMs, allowing organizations to run modular Java applications or full Java EE workloads based on their architectural needs and modernization goals.

3. How Java in CICS works

Java applications in CICS utilize various integrated layers and components essential for seamless operations and interoperability:

- **IBM 64-bit SDK for z/OS, Java Technology Edition**: A specialized Java runtime tailored for optimized performance on mainframes, offering necessary libraries and tools for efficient Java application execution within CICS environments. [2]

- **JVM Profiles**: This is the critical setup file for CICS JVMs that defines memory management, threads, and performance optimization parameters for the JVM so that it will run efficiently in CICS.

- **Unix System Services (USS)**: Provides a UNIX-like environment in z/OS to support Java application artifacts and libraries, with a similar familiar file system and runtime environment.

- **CICS Resources**: Required CICS resources such as JVM server definitions, bundle definitions, web services pipeline configurations, and transaction setups are necessary to deploy and manage Java applications in CICS successfully.

- **JCICS API**: An interface allowing Java applications to directly interact with CICS resources like VSAM files and Temporary Storage, ensuring smooth integration with existing CICS data and application logic. [3]

4. Options for Running Java in CICS

Java in CICS can be deployed via two primary options: OSGi JVM and Liberty JVM.

4.1. OSGi JVM

OSGi JVM leverages the OSGi (Open Services Gateway initiative) modular architecture, a dynamic component system for Java. It enables applications to be composed of small, reusable, and loosely coupled modules called bundles that can be installed, started, stopped, updated, and uninstalled independently without restarting the JVM. [1] This modularity offers numerous benefits such as version control at the component level, the capability of dynamic deployment, and more maintainable application structures—especially beneficial in CICS environments where stability and uptime are critical.[2]

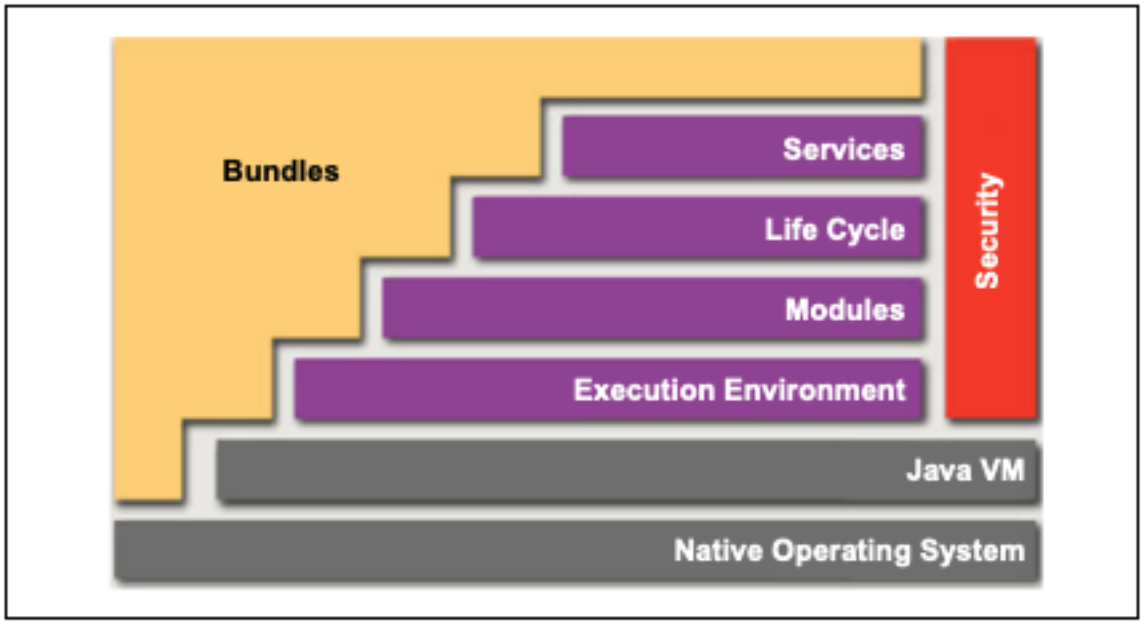

Figure 1: Illustrating different layers of OSGi framework. [1]

Benefits:

- **Modularity:** Applications are divided into bundles, and every component is independently manageable. Modularity enables companies to construct, deploy, and manage parts of their application without the danger of impacting the overall system. [1]
- **Dynamic Component Updates:** OSGi enables hot-swapping components, crucial for production environments like CICS that demand high availability and minimal downtime. [2]
- **Reduced Downtime:** Due to the modularity offered by the OSGi framework, each module can be individually started, stopped, or updated, and updates can be deployed without taking down the entire application, significantly improving operational efficiency and availability of services.

Use Case: Ideal for applications requiring frequent updates or modular extensions. A practical, real-world example includes a telecom billing platform that processes usage records. These platforms often require updates to calculation logic, promotions, or bundling rules based on dynamic regulatory or business needs. Using OSGi allows developers to deploy new logic as a module without stopping the entire system, minimizing downtime while keeping pace with service innovation. IBM has documented similar modular deployment patterns within customer environments using OSGi to manage isolated microservices within JVM servers. [2]

Components Required:

- **JVM Server Definitions:** These definitions tell CICS how to initialize and manage JVMs. You can define a JVM server using CICS system definitions (RDO) or via CEDA as follows:

```
CEDA DEFINE JVMServer(JVMSRV1) GROUP(JVMSDEMO) \
DESCRIPTION('Java OSGi server') \
    PROGRAM(JVMMAIN) \
    JVMPROFILE(JVMProfile) \
    WORKDIR('/u/user1/workdir') \
```

```
CLASSLOADER(CICS)
```

This definition creates a named JVM server with a profile, main entry point, and a work directory. [2]

- **JVM Profile**:The JVM Profile file is required to start the JVM server in CICS. It defines initial and maximum heap size, encoding, and diagnostic parameters. Importantly, the JVM profile file must be in EBCDIC encoding so the CICS region in z/OS can read it appropriately. [2]
- **OSGi Bundle Projects:** Java applications must be packaged as OSGi bundles. These bundles must include a manifest file (MANIFEST.MF) that declares the bundle's symbolic name, version, and imported/exported packages. These can be built using tools like Eclipse PDE or Maven Tycho. For example:

```
Bundle-SymbolicName: com.example.billing.logic
Bundle-Version: 1.0.0
Import-Package: com.ibm.cics.server,org.osgi.framework
```

- **JCICS Libraries:** These are required for interacting with CICS services. The JCICS JAR (com.ibm.cics.server.jar) must be added to your project's class path, and the corresponding classes allow for operations like file access, containers, and program calls.
- **Bundle Deployment:** Use CICS Explorer or FTP to deploy the bundle JAR into the JVM server's deployed bundles directory. Then, the bundle can be started dynamically using the CICS Explorer GUI or by defining it within a CICS bundle definition.

  Example with CICS Explorer: Right-click the JVM server > "Install Bundle" > select JAR file. [2]

Limitations:

- **Dependency Management Complexity:** OSGi modularity can introduce complexity in managing inter-bundle dependencies. Each bundle must explicitly import and export; version incompatibilities can occur if dependent bundles are not properly aligned. Coordinating all packages and their versions to be compatible across bundles can require a lot of coordination, especially in large applications. [1]
- **Initial Learning Curve:** Developers unfamiliar with the OSGi framework may face a steep learning curve, especially when transitioning from monolithic Java applications. Understanding how OSGi resolves bundles and manages lifecycle stages requires a mindset different from that of traditional Java application development. [2]
- **Tooling and Debugging:** Although tools like Eclipse PDE support development, debugging, and troubleshooting in dynamic modules, they are occasionally harder to handle than usual programs—mistakenly set up MANIFEST.MF files or incorrect package visibility may lead to runtime exceptions that may be difficult to trace. Additionally, developers may have problems dealing with OSGi runtime-related issues, i.e., classloading conflicts or bundle lifecycle mismatches. IBM's documentation acknowledges that modular deployment enables flexibility but requires diligent configuration and testing to ensure components interact reliably. [2]

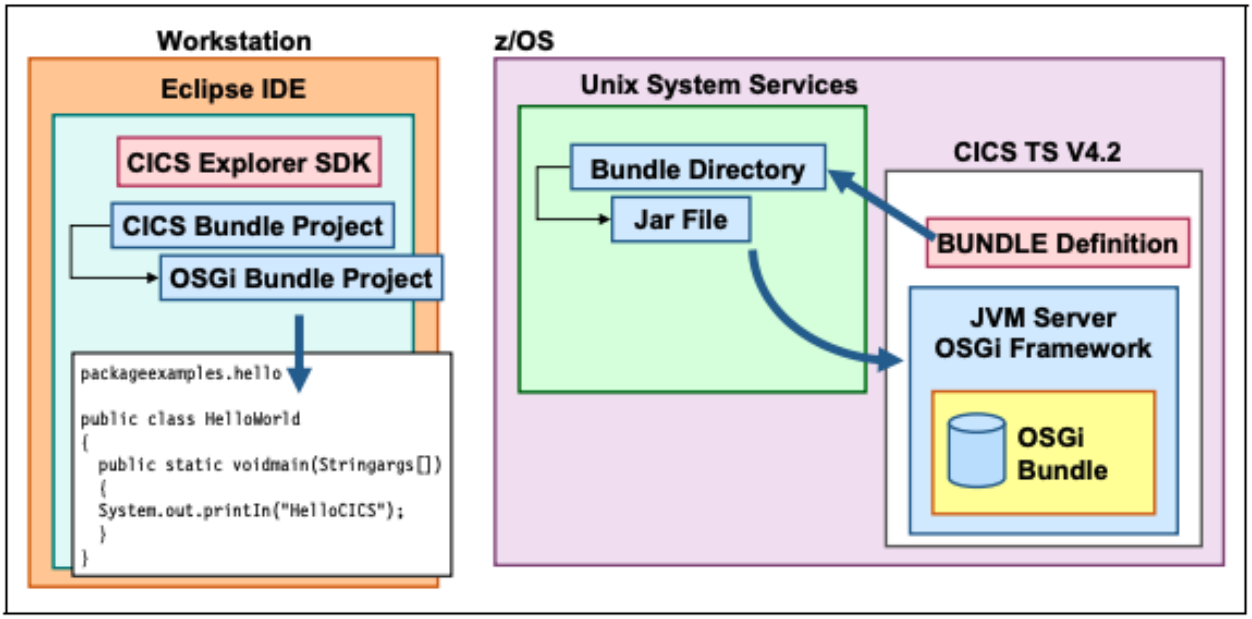

Figure 2: illustrating the deployment of OSGi bundle into a specific CICS region. [1]

Example Setup:

1. Define the JVM server in CICS using RDO or CEDA commands. For example:

```
CEDA DEFINE JVMServer(JVMSRV1) GROUP(JVMSDEMO) \
DESCRIPTION('Java OSGi server') \
    PROGRAM(JVMMAIN) \
    JVMPROFILE(JVMProfile) \
    WORKDIR('/u/user1/workdir') \
    CLASSLOADER(CICS)
```

   This server will act as the hosting environment for the Java program that will write to a VSAM file.

2. Develop and package the Java application to accept input from a channel container and write it to a VSAM file using the JCICS API.
   Example code:

```
Channel channel = Task.getTask().getChannel();
Container container = channel.getContainer("INPUTDATA");
byte[] data = container.get();
File file = new File();
file.setName("VSAMFILE");
file.write(data);
```

3. Create an OSGi-compliant JAR bundle, including MANIFEST.MF:

```
Bundle-SymbolicName: com.example.cics.vsamwriter
Bundle-Version: 1.0.0
Import-Package: com.ibm.cics.server,org.osgi.framework
```

   Need to ensure com.ibm.cics.server.jar is on your classpath during development.

4. Deploy the OSGi bundle using CICS Explorer by following below steps:
   - Open CICS Explorer

- Right-click the JVM server → "Install Bundle"
- Browse to your bundle JAR and upload
- The bundle will be loaded and managed dynamically by the JVM server

5. Define a CICS TRANSACTION and PROGRAM to invoke the Java application, and optionally link it to a COBOL program:

```
* COBOL program invoking Java transaction
EXEC CICS LINK PROGRAM('JAVAPGM')
   CHANNEL('MYCHANNEL')
   RESP(RESP-CODE)
END-EXEC.
```

6. Summary of bundle and deploy steps:
   - Use ‘mvn clean install’ or PDE export for bundle creation
   - FTP or CICS Explorer to place JAR in the appropriate USS path or bundle directory
   - Optionally configure bundle in a CICS bundle project XML for automation. [2]

```
// Example JCICS API usage
import com.ibm.cics.server.*;

Channel channel = Task.getTask().getChannel();
Container container = channel.getContainer("INPUTDATA");
byte[] data = container.get();

// Write to VSAM file using JCICS File API
File file = new File();
file.setName("VSAMFILE");
file.open(File.WRITE, File.NOTIFY);
Record record = file.createRecord();
record.setBytes(data);
file.write(record);
file.close();

// Write same data to Temporary Storage Queue
TemporaryStoragetsq = new TemporaryStorage();
tsq.setName("TEMPQ001");
tsq.writeItem(data);
```

4.2. Liberty JVM

Liberty JVM supports Java EE capabilities like web services, RESTful APIs, and modern application frameworks. Liberty JVM is based on IBM's WebSphere Liberty Profile, a light, modular, and highly composable Java runtime for cloud-native and enterprise applications. In a CICS environment, Liberty JVM executes in a dedicated CICS JVM server configured for hosting Liberty profile workloads. This setup allows Java applications to leverage enterprise-class features inside a managed, scalable, and secure mainframe environment. [2]

Integrating Liberty with CICS enables developers to build Java EE applications that can act as RESTful endpoints or SOAP services while directly interacting with CICS-managed resources via the JCICS API. Liberty servers are defined in CICS using standard resource definitions, and application artifacts such as WAR or EAR files are deployed to the designated file paths under z/OS Unix System Services (USS). These applications are then mapped in the server.xml configuration file, which defines the Liberty runtime behavior. [2]

Liberty in CICS supports several enterprise grade features like connection pooling, security constraints, session persistence, and asynchronous servlet processing. Modularly configurable components lead to the loading of only required runtime components, which further optimizes resource utilization. This support is particularly valuable in mainframe environments where system resource usage and performance are well-controlled. Additionally, developers can use the Liberty Developer Tools for Eclipse and CICS Explorer to ease the development, testing, and deployment of Liberty-based applications on z/OS. [2]

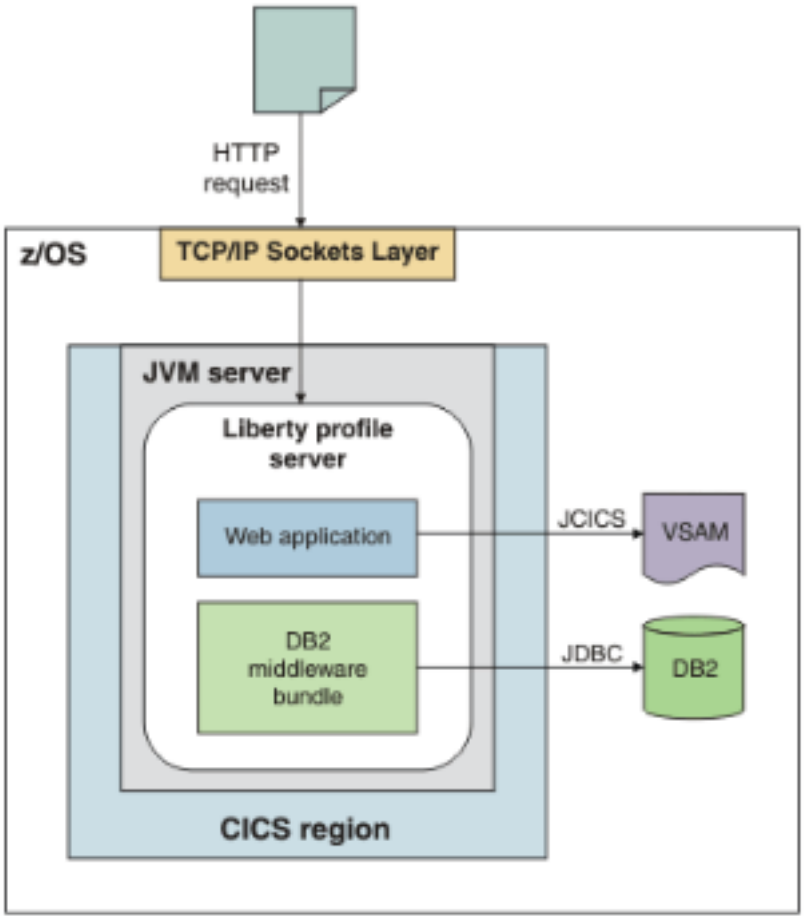

Figure 3: Depicting the high levelLiberty JVM architecture in CICS deployment. [2]

Benefits:

- **Ease of Deployment**: Liberty supports simplified deployment models with loose application logic coupling and quick configuration via server.xml.
- **Standards Compliance**: Supports a broad set of Java EE and Jakarta EE specifications, making it ideal for hosting portable and compliant enterprise applications.

- **Rapid Startup and Lightweight Footprint**: Unlike traditional application servers, Liberty starts fast and uses fewer resources when configured minimally, enabling faster development and testing cycles. [2]
- **Integration with DB2**: Liberty JVM servers can access DB2 databases using JDBC configuration within server.xml. This capability enables modern Java applications to participate in CICS-managed transactional contexts while accessing z/OS-resident relational data. [2]
- **Messaging with IBM MQ**: Liberty JVM applications can integrate with IBM MQ using JMS or message-driven beans (MDBs), allowing seamless message-based communication between distributed environments and mainframe transaction systems. [4]
- **z/OS Security Integration**: Applications running on Liberty JVM can utilize RACF and SAF-based authentication and authorization, leveraging centralized enterprise-wide security policies. Java applications can enforce security roles mapped to RACF groups for robust and compliant access control. [2]
- **Transaction Coordination and Workload Management**: Liberty workloads can be integrated into CICS regions' workload management infrastructure, enabling optimized dispatching, region-level failover, and participation in CICS-managed transactions, which improves reliability and resource efficiency. [2]

Use Case: Ideal for web-facing applications or microservices that require REST endpoints or SOAP-based service interfaces. For example, a retail inventory system deployed on Liberty JVM can expose a RESTful API to allow distributed mobile apps to query stock levels or initiate purchase orders in real time, integrating directly with CICS transaction backends.

Components Required:

- Liberty JVM Server Definition in CICS: Liberty JVM servers are defined using standard CEDA or resource definitions in CICS. For example:

```
CEDA DEFINE JVMServer(JVMLIB01) GROUP(JVMLIBGRP) \
DESCRIPTION('Liberty JVM Server') \
    PROGRAM(JVMMAIN) \
    JVMPROFILE(LIBJVMPR) \
    WORKDIR('/u/user1/liberty/wlp/usr/servers/myserver') \
    CLASSLOADER(SERVER)
```

- Each CICS Liberty JVM server requires a JVM profile file—typically named after the JVMPROFILE parameter in the CEDA definition—to configure runtime behavior for the JVM instance. This profile file must be encoded in EBCDIC format to ensure it is correctly interpreted by the CICS region under z/OS.
  Below is a sample JVM profile file (e.g., LIBJVMPR):

```
# JVM Profile for Liberty
-Dfile.encoding=UTF-8
-Xms256m
-Xmx512m
-Dcom.ibm.cics.server.traceLevel=INFO
```

- The Liberty profile JVM's behavior is configured via server.xml located in the server's root directory. This XML file defines runtime features, application deployment paths, and endpoints. This file is placed in the workdir directory defined for the Liberty JVM server and helps fine-tune memory and diagnostic options. [2]
  Below is a sample server.xml file:

```
<server description="Liberty Server">
<featureManager>
<feature>servlet-4.0</feature>
<feature>jaxrs-2.1</feature>
<feature>jdbc-4.2</feature>
<feature>mpMetrics-3.0</feature>
</featureManager>

<httpEndpoint id="defaultHttpEndpoint" host="*" httpPort="9080" httpsPort="9443"/>

<application context-root="/cicsapp" location="cicsapp.war" type="war"/>
</server>
```

This configuration defines the features available in the runtime and the web application to be loaded. The WAR file (cicsapp.war) must be placed in the Liberty server's dropins directory. [2]

- **WAR/EAR files:** Java web applications are packaged using Maven, Gradle, or Eclipse as standard WAR or EAR files for deployment into the Liberty server.
- **JCICS API:** When Java applications in Liberty need to access CICS resources, the com.ibm.cics.server package (JCICS API) can be used to interact with containers, VSAM files, temporary storage, etc. This allows seamless interaction between Java EE components and the CICS runtime.
- **Web Descriptors:** The deployment uses web.xml to define servlet configurations, and server.xml to configure Liberty server behavior such as ports, context paths, features, and security roles. [2]

Limitations:

- **Resource Usage:** Liberty JVM requires more levels of memory and CPU compared to OSGi JVM when running full Java EE stacks. The need for more resources is most commonly seen when many Java EE features are enabled because they contribute to the base footprint required to set up and operate the Liberty runtime effectively. [2]
- **Startup Time with Full EE Features:** Although Liberty is designed for quick startup, enabling a large number of Java EE or MicroProfile features (e.g., JMS, JPA, JAX-RS) can increase initialization time and impact responsiveness during server restarts or region restarts, especially in shared or constrained environments.
- **Debugging CICS-Java Integration:** When Java code invokes or is invoked by CICS programs, tracing issues across boundaries may require advanced diagnostic tooling. Developers often rely on the combined use of CICS trace, Liberty server logs, and specialized dump analysis tools. Diagnostic

complexity increases when Java interacts with multiple CICS resources (e.g., DB2, MQ, or TSQs) in a single transaction flow. [2]

- **Concurrency and Thread Management Constraints:** Liberty servers in CICS are subject to CICS tasking models and region constraints. Developers must be aware of limitations in thread creation, thread pooling, and asynchronous processing to avoid violating CICS system integrity and performance recommendations. [2]

Example Setup:

1. Define the Liberty JVM server in CICS**:**

   Use CEDA to define the Liberty JVM server that will host your Java EE application:

```
CEDA DEFINE JVMServer(JVMLIB01) GROUP(JVMLIBGRP) \
   DESCRIPTION('Liberty JVM Server for API Processing') \
   PROGRAM(JVMMAIN) \
   JVMPROFILE(LIBJVMPR) \
   WORKDIR('/u/user1/liberty/wlp/usr/servers/myserver') \
   CLASSLOADER(SERVER)
```

2. Build the WAR or EAR file:
   Develop your Java EE application using your preferred IDE or build tool such as Maven or Gradle. A sample Maven command to build a WAR:

```
mvn clean package
```

   This will generate a cicsapp.war file in the target/ directory.

3. Deploy to USS file system:
   Upload the WAR file (e.g., cicsapp.war) to the Liberty server's dropins directory within the work directory defined in the JVMServer (e.g.,/u/user1/liberty/wlp/usr/servers/myserver/dropins)

4. JVM Profile example (LIBJVMPR):

```
# JVM Profile for Liberty
-Dfile.encoding=UTF-8
-Xms256m
-Xmx512m
-Dcom.ibm.cics.server.traceLevel=INFO
```

   This file is placed under the JVM server’s workdir and helps configure memory and diagnostic behavior.

5. Liberty server.xml example:

```
<server description="Liberty Server">
<featureManager>
<feature>servlet-4.0</feature>
<feature>jaxrs-2.1</feature>
<feature>jdbc-4.2</feature>
</featureManager>
```

```
<httpEndpoint id="defaultHttpEndpoint" host="*" httpPort="9080" httpsPort="9443"/>

<application context-root="/cicsapp" location="cicsapp.war" type="war"/>
</server>
```

6. **Use Case:**
   This example supports an API endpoint that receives data over HTTP and writes the payload into both a CICS-managed VSAM file and a Temporary Storage Queue. This pattern is common in financial services where transactional input (like payment or account update requests) must be persisted in multiple formats for audit, recovery, or downstream processing.

7. Java Code example:

```
import com.ibm.cics.server.*;
import javax.servlet.http.*;
import javax.servlet.annotation.*;
import java.io.*;

@WebServlet("/writeToFile")
public class FileWriterServlet extends HttpServlet {
  protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws IOException {
byte[] inputData = req.getInputStream().readAllBytes();

   // Write to VSAM file
   File file = new File();
file.setName("VSAMFILE");
file.open(File.WRITE, File.NOTIFY);
   Record record = file.createRecord();
record.setBytes(inputData);
file.write(record);
file.close();

   // Write to Temporary Storage
TemporaryStoragetsq = new TemporaryStorage();
tsq.setName("TEMPQ001");
tsq.writeItem(inputData);
  }
}
```

8. **Test through RESTful API**:
   To invoke this servlet from a distributed environment or test client, you can use a curl command. This assumes the Liberty server is listening on port 9080 and that the cicsapp.war has been deployed correctly into the dropins directory. The following example sends binary data from a file (input.dat) to the servlet and checks for an HTTP 200 OK response:

```
curl -X POST http://<host>:9080/cicsapp/writeToFile \
    -H "Content-Type: application/octet-stream" \
```

```
--data-binary @input.dat \
-v
```

A successful response should include a line such as:

```
< HTTP/1.1 200 OK
```

This indicates that the payload was successfully received and processed by the Liberty-hosted Java application running under CICS. [2]

5. JCICS APIs

The JCICS API facilitates Java applications interacting with CICS resources seamlessly. Below are a few core APIs commonly used and how they operate within a transactional CICS Java environment:

- **VSAM File Operations (FileAPI):** The File class in JCICS provides access to CICS-managed files (typically VSAM). Applications can open a file, create a record, write or read data, and close the file.

```
File file = new File();
file.setName("CUSTOMERVSAM");
file.open(File.WRITE, File.NOTIFY);
Record record = file.createRecord();
record.setBytes(data);
file.write(record);
file.close();
```

- **Temporary Storage Access (TemporaryStorage API):** This API allows you to create and manage temporary storage queues. Useful for queuing data across tasks or intermediate buffering.

```
TemporaryStoragetsq = new TemporaryStorage();
tsq.setName("TEMPQ001");
tsq.writeItem(data);
byte[] readData = tsq.readItem(1);
```

- **Container Management (Channel and Container APIs):** Java programs in CICS often receive input via containers inside channels. This mechanism supports structured and multi-element data passing.

```
Channel channel = Task.getTask().getChannel();
Container container = channel.getContainer("INPUTDATA");
byte[] input = container.get();

// Optionally send response through same channel
Container outContainer = channel.createContainer("OUTPUTDATA");
outContainer.put("Processed".getBytes());
```

- **Program-to-Program Calls (Program API):** Java applications can link to COBOL or HLASM programs using the Program class. This enables reuse of existing logic.

```
Program legacyProgram = new Program();
legacyProgram.setName("LEGACYPGM");
legacyProgram.link();
```

These APIs allow Java code to behave as first-class CICS programs, ensuring data consistency, transactional integrity, and interoperability across the z/OS stack. [2]

Section 6 - Deployment Strategies

Deployment strategies vary from manual approaches using CICS Explorer to automated CICSPlex-based APIs:

- **Eclipse-based tooling** facilitates interactive development and testing. Developers can use IBM CICS Explorer with the CICS Deployment Assistant or the Liberty Developer Tools to deploy OSGi or Liberty-based applications directly from the IDE into the CICS development region. [5]
- **Automated deployment pipelines** can leverage tools like Jenkins, GitHub Actions, or IBM UrbanCode Deploy to automate building and transferring artifacts to z/OS USS. Artifacts such as WAR files (for Liberty) or OSGi bundles (for Java programs) are deployed into designated directories configured in the CICS JVM server.
- **CICSPlex System Manager APIs** allow for dynamic resource management during deployment, such as turning on or off JVM servers or updating installed bundles and Liberty applications across multiple CICS regions. These APIs help with operational consistency in large enterprises. [2]
- **Shell or scripting automation** can also be used with FTP, SSH, or z/OSMF workflows to copy files and invoke CICS commands in batch jobs for non-interactive environments.

These deployment options allow organizations to scale from test environments to production with greater control and agility.

Section 7 - Performance Considerations

Java in CICS performance considerations cover some aspects vital to high throughput and efficient resource utilization:

- **Thread Pool Limits and Region-Level Constraints:** Each CICS JVM server supports a maximum of 256 running threads. New requests beyond that will be queued or rejected depending on server configuration. At the CICS region level, CICS imposes a practical upper limit of approximately 2000 threads across all JVM servers and internal system threads. Managing concurrency under these thresholds is essential to prevent degradation or transaction delays in high-throughput environments. [2]
- **JVM Heap Size Optimization:** Tuning the -Xms and -Xmx values in the JVM profile file ensures that the JVM has enough memory to avoid excessive garbage collection while staying within CICS region constraints. Under-provisioning can lead to performance bottlenecks, while over-provisioning can affect region stability. [2]
- **zIIP Offloading and Workload Balancing:** IBM allows eligible Java workloads to be executed on z Integrated Information Processors (zIIPs), which are typically less expensive than general-purpose

processors. Ensuring that the Java workload is zIIP-eligible and properly dispatched by CICS allows organizations to reduce software licensing costs and improve performance under high load. [1]

- **Efficient JCICS API Usage:** Accessing resources such as VSAM files and temporary storage queues or invoking COBOL programs through JCICS must be done with extra care to avoid unnecessary open/close cycles, reducing the number of transactions per task, and batching operations where possible, contribute to better performance. For instance, reusing a File object across multiple operations reduces overhead compared to repeated instantiation.
- **Thread and Region Configuration:** Liberty JVM servers within CICS must comply with CICS's threading model. Thread pool settings in server.xml, the number of concurrent servlet requests, and transaction thresholds should align with the MVS dispatching model to prevent contention and resource starvation.
- **JVM Monitoring and Diagnostics:** Monitoring the performance and health of JVM servers is essential for proactive issue resolution and optimization. IBM provides tools like CICS Performance Analyzer, IBM Monitoring and Diagnostic Tools for Java (Health Center), and Liberty's built-in monitoring features like mpMetrics, verbose GC, and server logs. These can be integrated with enterprise observability platforms to track memory usage, garbage collection, response times, and thread utilization. [2]

Section 8 - Challenges

Java applications in CICS possess a unique set of run-time and design concerns. These limitations must be resolved when creating and implementing Java solutions within mainframe environments:

- **Memory Management Constraints:** Java applications must manage memory consumption carefully within the allocated JVM heap to avoid excessive garbage collection or region instability.
- **CICS Thread Concurrency Limit:** Each JVM server is restricted to a maximum of 256 concurrent threads because of a constraint tied to the design of CICS's internal control structures. This thread limit can limit scalability in high-throughput workloads. [2]
- **Threading Model Restrictions:** Java applications must adhere to the CICS or container based thread pool management and task dispatching model, discouraging unbounded thread creation. Constructs such as Thread, ExecutorService, or ForkJoinPool must be used cautiously or avoided to preserve region stability and transactional consistency. [2]
- **Integration Complexity with Legacy Code:** Integrating Java with existing COBOL or HLASM programs may require bridging different programming models, adding complexity to transaction design and error handling.
- **Monitoring and Diagnostic Challenges:** JVM resource usage and performance tuning are more complex in CICS environments. Effective monitoring may entail multiple tools and familiarity with Java and CICS run-time behavior.
- **Skill Set Requirements:** Successful deployment and operations often require cross-domain expertise spanning Java development, CICS administration, and mainframe DevOps practices. [2]

Conclusion

Blending Java with CICS offers businesses a compelling path to modernization by enabling the reuse of legacy systems and embracing modern development methods. With the ability to develop modular applications using OSGi and enterprise-class services using Liberty JVM, hybrid application design on the mainframe becomes increasingly feasible.Java programs can now exploit CICS resources such as VSAM files, temporary storage areas, and COBOL programs in a transactionally consistent and transparent fashion through JCICS API.

However, realizing these benefits requires a solid understanding of CICS limitations, such as threading models, heap management, and the 256-thread per JVM server limit. Efficient deployment practices and diagnostic tools are also necessary to deploy at scale. Interoperability with z/OS functions such as DB2, MQ, RACF, and zIIP processing delivers maximum performance and security.

Future research and development should focus on increasing Java performance on z/OS, simplifying deployment and monitoring pipelines, and simplifying interoperability between legacy CICS applications and Java. Standardizing reusable Java components for common transaction patterns facilitates easier adoption in enterprise environments.

References

[1] IBM, "IBM CICS and the JVM server: Developing and Deploying Java Applications", IBM Redbooks, July 2013. [Online]. Available: https://www.redbooks.ibm.com/redbooks/pdfs/sg248038.pdf. [Accessed: October 2023].

[2] IBM, "CICS Transaction Server for z/OS 5.2", IBM Documentation, March 2021. [Online]. Available: https://www.ibm.com/docs/en/cics-ts/5.2. [Accessed: October 2023].

[3] SHARE.org, "Jumpstart your Java on CICS TS," September 2021. [Online]. Available: https://blog.share.org/Article/jumpstart-your-java-on-cics-ts. [Accessed: October 2023].

[4] IBM, "WebSphere Application Server Liberty", IBM Documentation. [Online]. Available: https://www.ibm.com/docs/en/was-liberty/base. [Accessed: October 2023].

[5] IBM Documentation, "CICS Explorer Overview", IBM Documentation, March 2021. [Online]. Available: https://www.ibm.com/docs/en/cics-explorer/5.4.0. [Accessed: October 2023].