In MultiModule OM based project how to execute main class from specific module jar file

**ChatGPT said:**

In a multi-module project using an Object Model (OM)-based build system like Maven or Gradle, you can execute the main class from a specific module’s JAR file using the following approaches:

**1. Maven-Based Multi-Module Project**

If you are using Maven, follow these steps:

**Step 1: Ensure Your Module is a Runnable JAR**

Make sure the module that contains the main class has a manifest entry defining the Main-Class in its pom.xml:

xml

CopyEdit

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-jar-plugin</artifactId>

<version>3.2.0</version>

<configuration>

<archive>

<manifest>

<mainClass>com.example.MainClass</mainClass>

</manifest>

</archive>

</configuration>

</plugin>

</plugins>

</build>

This ensures that the JAR can be run directly using java -jar.

**Step 2: Build the Module**

Navigate to the root of the project and build the specific module:

sh

CopyEdit

mvn clean install -pl module-name -am

* -pl module-name → Specifies the module to build.
* -am → Ensures that dependencies of the module are also built.

**Step 3: Execute the JAR**

Once built, navigate to the target directory of the module and run:

sh

CopyEdit

java -jar target/module-name-1.0-SNAPSHOT.jar

If dependencies are not included, you may need to use the **shade plugin** to create a fat JAR:

xml

CopyEdit

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-shade-plugin</artifactId>

<version>3.2.4</version>

<executions>

<execution>

<phase>package</phase>

<goals>

<goal>shade</goal>

</goals>

</execution>

</executions>

</plugin>

Then rebuild and run again.

**2. Gradle-Based Multi-Module Project**

If using Gradle, follow these steps:

**Step 1: Define Application Plugin in build.gradle**

In the specific module’s build.gradle:

gradle

CopyEdit

plugins {

id 'application'

}

application {

mainClass = 'com.example.MainClass'

}

**Step 2: Build the Module**

From the root project directory:

sh

CopyEdit

./gradlew :module-name:build

**Step 3: Execute the JAR**

Navigate to the module’s build/libs/ directory and run:

sh

CopyEdit

java -jar build/libs/module-name.jar

Alternatively, if dependencies are needed inside a fat JAR, use:

gradle

CopyEdit

jar {

manifest {

attributes 'Main-Class': 'com.example.MainClass'

}

}

Or use the **Shadow plugin**:

gradle

CopyEdit

plugins {

id 'com.github.johnrengelman.shadow' version '8.1.1'

}

shadowJar {

archiveClassifier.set('')

}

Then rebuild and execute:

sh

CopyEdit

./gradlew :module-name:shadowJar

java -jar build/libs/module-name.jar

**3. Directly Running Main Class Without Creating a JAR**

If you don’t want to package a JAR, you can run the class directly from the module:

**Maven**

sh

CopyEdit

mvn exec:java -pl module-name -Dexec.mainClass="com.example.MainClass"

**Gradle**

sh

CopyEdit

./gradlew :module-name:run

**Conclusion**

* If using Maven, mvn clean install -pl module-name followed by java -jar target/module-name.jar is the best approach.
* If using Gradle, ./gradlew :module-name:shadowJar followed by java -jar build/libs/module-name.jar works best.
* Use exec:java or gradle run for quick execution without packaging.

Would you like help debugging your specific setup? 🚀

4o

**You said:**

Extend the Mavenbased example to take arguments

**ChatGPT said:**

To extend the Maven-based example to accept arguments when executing the main class from a specific module JAR file, follow these steps:

**Step 1: Modify Your Main Class to Accept Arguments**

Ensure your MainClass is designed to handle command-line arguments:

java

CopyEdit

package com.example;

public class MainClass {

public static void main(String[] args) {

System.out.println("Application Started!");

if (args.length > 0) {

System.out.println("Arguments passed:");

for (String arg : args) {

System.out.println(arg);

}

} else {

System.out.println("No arguments provided.");

}

}

}

**Step 2: Ensure the JAR is Executable**

Modify pom.xml in the specific module to include the maven-jar-plugin:

xml

CopyEdit

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-jar-plugin</artifactId>

<version>3.2.0</version>

<configuration>

<archive>

<manifest>

<mainClass>com.example.MainClass</mainClass>

</manifest>

</archive>

</configuration>

</plugin>

</plugins>

</build>

If you need to package dependencies into a single JAR (fat JAR), use the **maven-shade-plugin**:

xml

CopyEdit

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-shade-plugin</artifactId>

<version>3.2.4</version>

<executions>

<execution>

<phase>package</phase>

<goals>

<goal>shade</goal>

</goals>

</execution>

</executions>

</plugin>

**Step 3: Build the Module**

From the root project directory, build the module:

sh

CopyEdit

mvn clean package -pl module-name -am

This creates the JAR in target/module-name-1.0-SNAPSHOT.jar.

**Step 4: Run the JAR with Arguments**

Navigate to the target directory of the module and execute:

sh

CopyEdit

java -jar target/module-name-1.0-SNAPSHOT.jar arg1 arg2 arg3

Example Output:

yaml

CopyEdit

Application Started!

Arguments passed:

arg1

arg2

arg3

**Alternative: Run Without Packaging JAR**

Instead of creating a JAR, you can directly execute the main class with arguments:

sh

CopyEdit

mvn exec:java -pl module-name -Dexec.mainClass="com.example.MainClass" -Dexec.args="arg1 arg2 arg3"

**Conclusion**

* **For packaged execution**: Use java -jar target/module-name.jar arg1 arg2
* **For direct execution**: Use mvn exec:java -pl module-name -Dexec.mainClass="com.example.MainClass" -Dexec.args="arg1 arg2"

Would you like additional configurations like logging or environment variable handling? 🚀

4o

**You said:**

what is different between VM arguments and Appliation aruguments .. how to pass VM arguments for POM based module

**ChatGPT said:**

**Difference Between VM Arguments and Application Arguments**

**1. Application Arguments (args[] in main)**

* These are **arguments passed to the main method** of your Java application.
* They are accessed via String[] args in public static void main(String[] args).
* They are **specific to the application logic** and can be used for input parameters, file paths, user options, etc.
* Passed **after** the JAR file in the command line.

🔹 **Example** (Passing application arguments):

sh

CopyEdit

java -jar my-app.jar arg1 arg2 arg3

🔹 **How to Access in Code**:

java

CopyEdit

public class MainClass {

public static void main(String[] args) {

System.out.println("Application Arguments:");

for (String arg : args) {

System.out.println(arg);

}

}

}

**2. VM Arguments (JVM Options)**

* These **configure the JVM itself** (memory settings, garbage collection, system properties, etc.).
* They are not passed to main(String[] args).
* Used for setting **heap size**, **garbage collector**, **system properties**, etc.
* Passed **before** the -jar option.

🔹 **Example** (Passing VM arguments):

sh

CopyEdit

java -Xmx512m -Denv=prod -jar my-app.jar arg1 arg2

* -Xmx512m → Sets max heap memory to 512MB.
* -Denv=prod → Defines a system property (System.getProperty("env")).
* arg1 arg2 are application arguments.

🔹 **How to Access VM Arguments in Code**:

java

CopyEdit

public class MainClass {

public static void main(String[] args) {

// Accessing application arguments

System.out.println("Application Arguments:");

for (String arg : args) {

System.out.println(arg);

}

// Accessing VM arguments (System Properties)

String env = System.getProperty("env", "default");

System.out.println("VM Argument: env = " + env);

}

}

**How to Pass VM Arguments in a Maven-Based Module**

**1. Running with exec:java Plugin**

If you are using Maven's exec:java, pass VM arguments using -Dexec.jvmArgs:

sh

CopyEdit

mvn exec:java -pl module-name -Dexec.mainClass="com.example.MainClass" \

-Dexec.jvmArgs="-Xmx512m -Denv=prod" \

-Dexec.args="arg1 arg2"

* -Dexec.jvmArgs="-Xmx512m -Denv=prod" → Sets VM arguments.
* -Dexec.args="arg1 arg2" → Sets application arguments.

**2. Running a Packaged JAR**

After building the JAR, pass VM arguments using java command:

sh

CopyEdit

java -Xmx512m -Denv=prod -jar target/module-name.jar arg1 arg2

**Summary**

| **Feature** | **Application Arguments (args[])** | **VM Arguments (-D, -X, etc.)** |
| --- | --- | --- |
| **Purpose** | Passed to main(String[] args) | Configures JVM (memory, properties, GC, etc.) |
| **Example** | java -jar my-app.jar arg1 arg2 | java -Xmx512m -Denv=prod -jar my-app.jar arg1 arg2 |
| **Access in Code** | args[] in main | System.getProperty("env") |
| **Maven Execution** | -Dexec.args="arg1 arg2" | -Dexec.jvmArgs="-Xmx512m -Denv=prod" |

Would you like an example with logging or more advanced configurations? 🚀

4o

**You said:**

Yes please

**ChatGPT said:**

**Advanced Example: Using VM and Application Arguments with Logging**

This example demonstrates: ✅ How to use **VM arguments** (-Dkey=value)  
✅ How to use **Application arguments** (args[])  
✅ How to configure **logging** (Logback example)

**Step 1: Create a Java Class**

This class:

* Reads **VM arguments** using System.getProperty()
* Reads **Application arguments** using args[]
* Uses **Logback logging** for structured logging

java

CopyEdit

package com.example;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class MainClass {

private static final Logger logger = LoggerFactory.getLogger(MainClass.class);

public static void main(String[] args) {

logger.info("Application Started");

// Reading Application Arguments

logger.info("Application Arguments:");

for (String arg : args) {

logger.info(arg);

}

// Reading VM Arguments (System Properties)

String env = System.getProperty("env", "development");

String logLevel = System.getProperty("log.level", "INFO");

logger.info("VM Argument: env = {}", env);

logger.info("VM Argument: log.level = {}", logLevel);

// Simulating Different Log Levels

logger.debug("This is a DEBUG message (only visible if log level is DEBUG)");

logger.info("This is an INFO message");

logger.warn("This is a WARN message");

logger.error("This is an ERROR message");

logger.info("Application Finished.");

}

}

**Step 2: Add Logback for Logging**

If you are using **Maven**, add the following dependency in pom.xml:

xml

CopyEdit

<dependencies>

<!-- SLF4J API -->

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>2.0.9</version>

</dependency>

<!-- Logback (Implementation) -->

<dependency>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-classic</artifactId>

<version>1.4.11</version>

</dependency>

</dependencies>

**Step 3: Add Logback Configuration (Optional)**

Create a logback.xml file inside src/main/resources/:

xml

CopyEdit

<configuration>

<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">

<encoder>

<pattern>%d{yyyy-MM-dd HH:mm:ss} [%level] %msg%n</pattern>

</encoder>

</appender>

<logger name="com.example" level="${log.level:-INFO}"/>

<root level="INFO">

<appender-ref ref="STDOUT"/>

</root>

</configuration>

💡 **Dynamic Log Level**:

* Uses ${log.level:-INFO} to allow runtime changes via VM arguments (-Dlog.level=DEBUG).
* Default log level is **INFO** if log.level is not provided.

**Step 4: Build the JAR**

Run the following Maven command to build the module:

sh

CopyEdit

mvn clean package -pl module-name -am

**Step 5: Run with Application and VM Arguments**

**🔹 Example 1: Run with Default Logging**

sh

CopyEdit

java -jar target/module-name.jar arg1 arg2

📌 **Output:**

yaml

CopyEdit

2025-02-17 12:00:00 [INFO] Application Started

2025-02-17 12:00:00 [INFO] Application Arguments:

2025-02-17 12:00:00 [INFO] arg1

2025-02-17 12:00:00 [INFO] arg2

2025-02-17 12:00:00 [INFO] VM Argument: env = development

2025-02-17 12:00:00 [INFO] VM Argument: log.level = INFO

2025-02-17 12:00:00 [INFO] This is an INFO message

2025-02-17 12:00:00 [WARN] This is a WARN message

2025-02-17 12:00:00 [ERROR] This is an ERROR message

**🔹 Example 2: Run with Custom Environment and Logging Level**

sh

CopyEdit

java -Xmx512m -Denv=production -Dlog.level=DEBUG -jar target/module-name.jar user1 input.txt

📌 **Output:**

yaml

CopyEdit

2025-02-17 12:05:00 [INFO] Application Started

2025-02-17 12:05:00 [INFO] Application Arguments:

2025-02-17 12:05:00 [INFO] user1

2025-02-17 12:05:00 [INFO] input.txt

2025-02-17 12:05:00 [INFO] VM Argument: env = production

2025-02-17 12:05:00 [INFO] VM Argument: log.level = DEBUG

2025-02-17 12:05:00 [DEBUG] This is a DEBUG message (only visible if log level is DEBUG)

2025-02-17 12:05:00 [INFO] This is an INFO message

2025-02-17 12:05:00 [WARN] This is a WARN message

2025-02-17 12:05:00 [ERROR] This is an ERROR message

**Step 6: Running the Application Using Maven (exec:java)**

Instead of using java -jar, you can run it directly with Maven and pass both **VM and Application arguments**:

sh

CopyEdit

mvn exec:java -pl module-name \

-Dexec.mainClass="com.example.MainClass" \

-Dexec.jvmArgs="-Xmx512m -Denv=production -Dlog.level=DEBUG" \

-Dexec.args="user1 input.txt"

📌 **This achieves the same result as running the JAR directly.**

**Summary**

| **Feature** | **Application Arguments (args[])** | **VM Arguments (-D, -X, etc.)** |
| --- | --- | --- |
| **Purpose** | Passed to main(String[] args) | Configures JVM (memory, properties, GC, etc.) |
| **Example** | java -jar my-app.jar arg1 arg2 | java -Xmx512m -Denv=prod -jar my-app.jar arg1 arg2 |
| **Access in Code** | args[] in main | System.getProperty("env") |
| **Logging Example** | System.out.println(args[i]) | logger.info(System.getProperty("log.level")) |
| **Maven Execution** | -Dexec.args="arg1 arg2" | -Dexec.jvmArgs="-Xmx512m -Denv=prod" |

**Next Steps**

* Would you like to **log to a file** instead of the console?
* Need help with **Spring Boot integration**?
* Want to **automate argument parsing** with Apache Commons CLI?

Let me know! 🚀

4o

**You said:**

Yes please extend the application using all above 3 points

**ChatGPT said:**

**Extending the Application with Additional Features**

This extension includes:  
✅ **Logging to a file** using Logback  
✅ **Spring Boot integration** for easier configuration  
✅ **Automated argument parsing** with Apache Commons CLI

**Step 1: Update Dependencies (pom.xml)**

Modify your **Maven pom.xml** to include:

* **SLF4J + Logback** (for logging to file)
* **Apache Commons CLI** (for argument parsing)
* **Spring Boot Starter** (for auto-configuration)

xml

CopyEdit

<dependencies>

<!-- SLF4J API -->

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>2.0.9</version>

</dependency>

<!-- Logback (Logging) -->

<dependency>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-classic</artifactId>

<version>1.4.11</version>

</dependency>

<!-- Apache Commons CLI (Command-line parsing) -->

<dependency>

<groupId>commons-cli</groupId>

<artifactId>commons-cli</artifactId>

<version>1.5.0</version>

</dependency>

<!-- Spring Boot -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter</artifactId>

<version>3.2.0</version>

</dependency>

</dependencies>

**Step 2: Configure Logback to Log to Both Console & File**

Create a logback.xml inside src/main/resources/:

xml

CopyEdit

<configuration>

<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">

<encoder>

<pattern>%d{yyyy-MM-dd HH:mm:ss} [%level] %msg%n</pattern>

</encoder>

</appender>

<appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">

<file>logs/app.log</file>

<rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">

<fileNamePattern>logs/app-%d{yyyy-MM-dd}.log</fileNamePattern>

<maxHistory>7</maxHistory>

</rollingPolicy>

<encoder>

<pattern>%d{yyyy-MM-dd HH:mm:ss} [%level] %msg%n</pattern>

</encoder>

</appender>

<logger name="com.example" level="${log.level:-INFO}"/>

<root level="INFO">

<appender-ref ref="STDOUT"/>

<appender-ref ref="FILE"/>

</root>

</configuration>

✔ Logs messages to **console**  
✔ Stores logs in a **file (logs/app.log)**  
✔ Uses a **rolling file policy** (keeps logs for 7 days)  
✔ Allows **dynamic log level changes** with -Dlog.level=DEBUG

**Step 3: Create Main Application Class with Spring Boot & CLI Parsing**

**Features of MainApplication**

* Uses **Spring Boot** for configuration
* Parses **command-line arguments** using Apache Commons CLI
* Logs **to console & file**
* Reads **VM arguments (env, log.level)**
* **Handles errors gracefully**

**MainApplication.java**

java

CopyEdit

package com.example;

import org.apache.commons.cli.\*;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class MainApplication implements CommandLineRunner {

private static final Logger logger = LoggerFactory.getLogger(MainApplication.class);

public static void main(String[] args) {

SpringApplication.run(MainApplication.class, args);

}

@Override

public void run(String... args) {

logger.info("Application Started");

// Parse Command-Line Arguments

CommandLine commandLine = parseArguments(args);

if (commandLine != null) {

// Process CLI Arguments

if (commandLine.hasOption("u")) {

String user = commandLine.getOptionValue("u");

logger.info("User: {}", user);

}

if (commandLine.hasOption("f")) {

String filePath = commandLine.getOptionValue("f");

logger.info("Processing file: {}", filePath);

}

}

// Read VM Arguments

String env = System.getProperty("env", "development");

String logLevel = System.getProperty("log.level", "INFO");

logger.info("VM Argument: env = {}", env);

logger.info("VM Argument: log.level = {}", logLevel);

// Simulate Different Log Levels

logger.debug("This is a DEBUG message (only visible if log level is DEBUG)");

logger.info("This is an INFO message");

logger.warn("This is a WARN message");

logger.error("This is an ERROR message");

logger.info("Application Finished.");

}

private CommandLine parseArguments(String[] args) {

Options options = new Options();

options.addOption("u", "user", true, "User name");

options.addOption("f", "file", true, "File path");

CommandLineParser parser = new DefaultParser();

HelpFormatter formatter = new HelpFormatter();

try {

return parser.parse(options, args);

} catch (ParseException e) {

logger.error("Failed to parse command-line arguments: {}", e.getMessage());

formatter.printHelp("java -jar app.jar", options);

return null;

}

}

}

**Step 4: Build the Application**

Run the following Maven command:

sh

CopyEdit

mvn clean package

**Step 5: Running the Application with Arguments**

**🔹 Example 1: Run with Default Logging**

sh

CopyEdit

java -jar target/module-name.jar -u Alice -f data.txt

📌 **Output (Console + logs/app.log)**:

yaml

CopyEdit

2025-02-17 12:00:00 [INFO] Application Started

2025-02-17 12:00:00 [INFO] User: Alice

2025-02-17 12:00:00 [INFO] Processing file: data.txt

2025-02-17 12:00:00 [INFO] VM Argument: env = development

2025-02-17 12:00:00 [INFO] VM Argument: log.level = INFO

2025-02-17 12:00:00 [INFO] This is an INFO message

2025-02-17 12:00:00 [WARN] This is a WARN message

2025-02-17 12:00:00 [ERROR] This is an ERROR message

2025-02-17 12:00:00 [INFO] Application Finished.

**🔹 Example 2: Run with Custom Environment & Logging**

sh

CopyEdit

java -Xmx512m -Denv=production -Dlog.level=DEBUG -jar target/module-name.jar -u Bob -f config.json

📌 **Output (Console + logs/app.log)**:

pgsql

CopyEdit

2025-02-17 12:05:00 [INFO] Application Started

2025-02-17 12:05:00 [INFO] User: Bob

2025-02-17 12:05:00 [INFO] Processing file: config.json

2025-02-17 12:05:00 [INFO] VM Argument: env = production

2025-02-17 12:05:00 [INFO] VM Argument: log.level = DEBUG

2025-02-17 12:05:00 [DEBUG] This is a DEBUG message

2025-02-17 12:05:00 [INFO] This is an INFO message

2025-02-17 12:05:00 [WARN] This is a WARN message

2025-02-17 12:05:00 [ERROR] This is an ERROR message

2025-02-17 12:05:00 [INFO] Application Finished.

**Conclusion**

✔ **Logging to file & console**  
✔ **Spring Boot auto-configuration**  
✔ **Command-line argument parsing**

Would you like to:

1. Add **Spring Boot properties** (application.properties)?
2. Improve **error handling** for invalid arguments?  
   Let me know! 🚀