TGH

Making Integrations Simpler

Call Java methods with DataWeave

Author
Anshul Singh

Contents

Introduction

Given Mule's foundation on Java and Spring, robust integration capabilities exist for invoking Java methods within Mule. Various methods are available to achieve this, with one effective approach being the utilization of the Java Component. This blog explores this method to call custom Java classes and covers:

- Invoking a static method on a given instance.
- Instantiating a Java class.
- Invoking a non-static method.

Background

Mule seamlessly allows the invocation of Java methods through DataWeave statements, enabling calls to methods in Java classes present within your Mule project.

Approach

This blog focuses on leveraging the Java Component for invoking Java methods, showcasing practical examples for static method invocation, class instantiation, and non-static method invocation.

By understanding these techniques, developers can enhance their Mule integration capabilities with seamless Java method invocation.

Overview

Call a Java Method

Before you can call a method, you must first import the class it belongs to into your DataWeave code. You can import Java classes just as you import DataWeave modules by including `java!` into the statement.

For example, below is a simple Java class with a single method that appends a random number at the end of a string. Assume that you created this class as part of a Java package named `org.mycompany.utils` in your Mule project's `src/main/java` folder.

```
package org.mycompany.utils;
```

```
import java.util.Random;


public class MyUtils {


        public static String appendRandom(String base) {
                return base + new Random().nextInt();
        }}
```

You can call the method `appendRandom()` from DataWeave code, in any of the following ways.

- Import the class and then refer to the method:

```
%dw 2.0
import java!org::mycompany::utils::MyUtils
output application/json
---
{a: MyUtils::appendRandom("myString")}
```

- Import one or more methods instead of the whole class:

```
%dw 2.0
import java!org::mycompany::utils::MyUtils::appendRandom
output application/json
---
{
        a: appendRandom("myString")
}
```

```
```

- If the method is a static method, import and call it in a single line:

```
%dw 2.0

output application/json

---

{

        a: java!org::mycompany::utils::MyUtils::appendRandom(vars.myString)

}
```

All three methods return the variable value with a random string appended:

```
{

  "a":"myString969858409"

}
```

Instantiate a Java Class

Through DataWeave code, you can instantiate a new object of any class. Note that after creating an instance, you can't call its instance methods through DataWeave. However, you can reference its variables.

This simple Java class has a method and a variable.

```
package org.mycompany.utils;

public class MyClass {



        private String foo;

         public MyClass(String foo) {

                this.foo = foo;
```



```
        }


         public String getFoo() {return foo;}}
```

To create the `MyClass` example in a Studio project:

1. Create a project for the class in Studio by clicking **New → Mule Project** and providing a name, such as **my-app**, and clicking **Finish**.
2. From **my-app**, create the `org.mycompany.utils` package by right-clicking **src/main/java** from Studio's **Package Explorer**, selecting **New → Package**, and naming the package `org.mycompany.utils`.
3. Create the `MyClass` class in the your new `org.mycompany.utils` package by right-clicking **org.mycompany.utils**, selecting **New → Class**, naming that class `MyClass`, and clicking **Finish**.
4. In the **MyClass.java** tab that opens in Studio, copy, paste, and save the contents `MyClass`.

The following DataWeave example imports `MyClass`, creates a new instance of the class, and calls its instance variable `foo`. (Note that it is not possible for DataWeave to call the object's private `getFoo()` method.)

```
%dw 2.0

import java!org::mycompany::utils::MyClass

output application/json

---

{

        a: MyClass::new("myString").foo

}
```

To add this DataWeave example to your Studio project:

1. Returning to `my-app.xml` in **src/main/mule**, drag a Transform Message component into the project's canvas, and click to open it.

2. In the component's **Transform Message** tab, replace the entire script in the source code area (on the right) with the DataWeave script (above).
3. Save your changes and click **Preview** (located farthest right on the tab) to view the following output:

```
{

  "a":"myString"

}
```

Note that the XML for the Transform Message configuration in the Mule flow looks something like this:

```
<flow name="my-appFlow" >

        <ee:transform doc:name="Transform Message" >

                <ee:message >

                        <ee:set-payload ><![CDATA[%dw 2.0

import java!org::mycompany::utils::MyClass

output application/json

---

{

a: MyClass::new("myString").foo

}]]></ee:set-payload>

                </ee:message>

        </ee:transform>

</flow>
```

Use a Java Bridge

DataWeave enables you to call any Java static function or constructor by using the Java bridge. This feature is useful when you need to reuse business logic written in Java or to instantiate any Java object that does not have an empty public constructor.

To use a Java bridge, transform a fully qualified Java name to a DataWeave name:

1. Add the prefix `!java`.
2. Replace the dots in the fully qualified name with `::`.

When invoking the Java function, DataWeave transforms each argument to the expected type in the function parameter.

The following example shows how to create a new instance of a `java.lang.NullPointerException`:

```
%dw 2.0

output application/json

---

java!java::lang::NullPointerException::new("foo")
```

The following example invokes the function `java.lang.String.valueOf`:

```
%dw 2.0

output text/plain

import valueOf from java!java::lang::String

---

valueOf(true)
```

Java Bridge Example

The following example shows an API that returns a greeting message based on the input `uriParam`. The Java bridge calls a static method within the `my.class.org.Greeter` class that builds the response message.

```
%dw 2.0

output application/json

import java!my::class::org::Greeter

---

{

response: Greeter::greet(attributes.uriParams."name"

}
```

XML for Java Bridge Example

Paste this code into your Studio XML editor to quickly load the flow for this example into your Mule app:

```
<?xml version="1.0" encoding="UTF-8"?>




<mule xmlns:ee="http://www.mulesoft.org/schema/mule/ee/core"
xmlns:http="http://www.mulesoft.org/schema/mule/http"


        xmlns="http://www.mulesoft.org/schema/mule/core"


        xmlns:doc="http://www.mulesoft.org/schema/mule/documentation"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.mulesoft.org/schema/mule/core
http://www.mulesoft.org/schema/mule/core/current/mule.xsd
```

```
http://www.mulesoft.org/schema/mule/http
http://www.mulesoft.org/schema/mule/http/current/mule-http.xsd


http://www.mulesoft.org/schema/mule/ee/core
http://www.mulesoft.org/schema/mule/ee/core/current/mule-ee.xsd">


        <http:listener-config name="HTTP_Listener_config" doc:name="HTTP Listener
config" doc:id="addcdb50-cdfa-400d-992f-495ab6dd373d" >


                <http:listener-connection host="0.0.0.0" port="8081" />


        </http:listener-config>


        <flow name="data-weave-java-bridge-example-flow" >


                <http:listener doc:name="Listener" config-
ref="HTTP_Listener_config" path="/greet/{name}"/>


                <ee:transform doc:name="Transform Message" >


                        <ee:message >


                                <ee:set-payload ><![CDATA[%dw 2.0


output application/json


import java!my::class::org::Greeter


---


{
```

```
        response: Greeter::greet(attributes.uriParams."name")



}]]></ee:set-payload>

                        </ee:message>

                    </ee:transform>

        </flow>

</mule>
```

Test the App

To test your app, follow these steps:

1. Save and run your Mule app.
2. Run the following curl command: `http://localhost:8081/greet/Username`.

The Mule app returns the following response message:

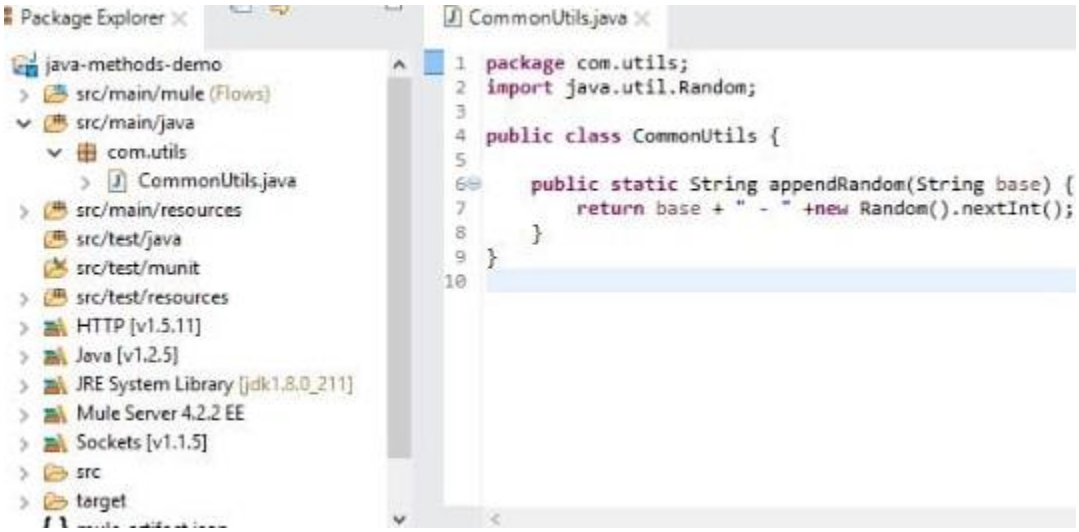
```
{
 "response": "Greetings, Username!"
}
```

Implementation

Invoke a Java Static Method from a Class

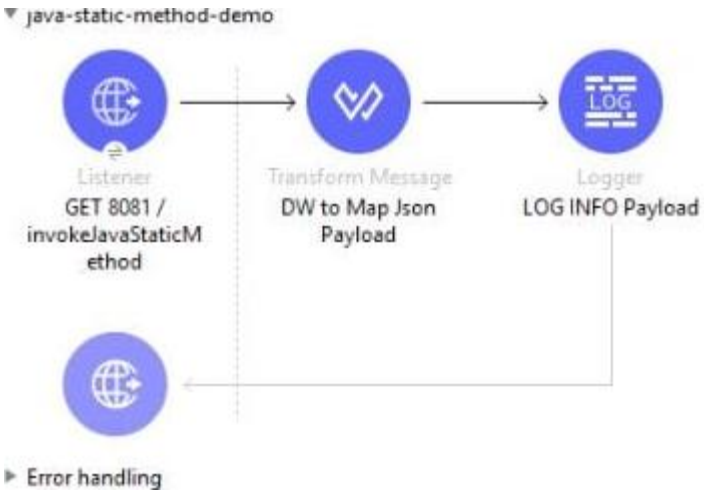
Step 1: Create a Java Class

- Create a class within the "utils" package in the `src/main/java` folder of your Mule project.



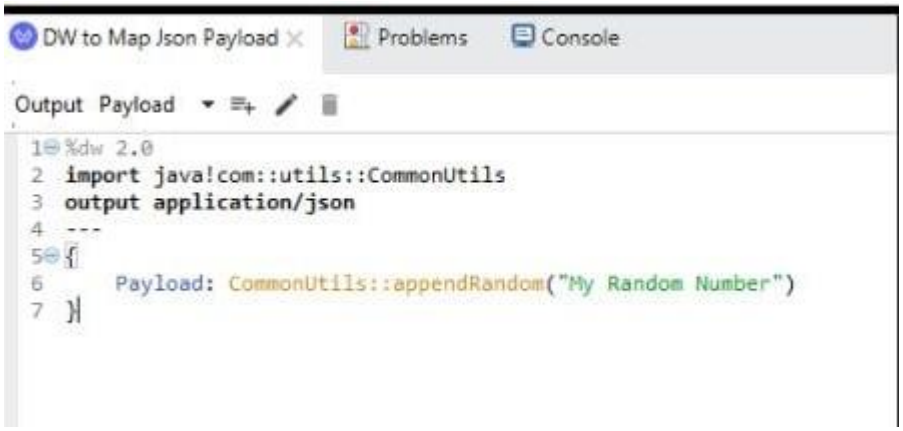
Step 2: Build Mule Flow

- Develop a Mule flow to invoke the static method `appendRandom()` from DataWeave.



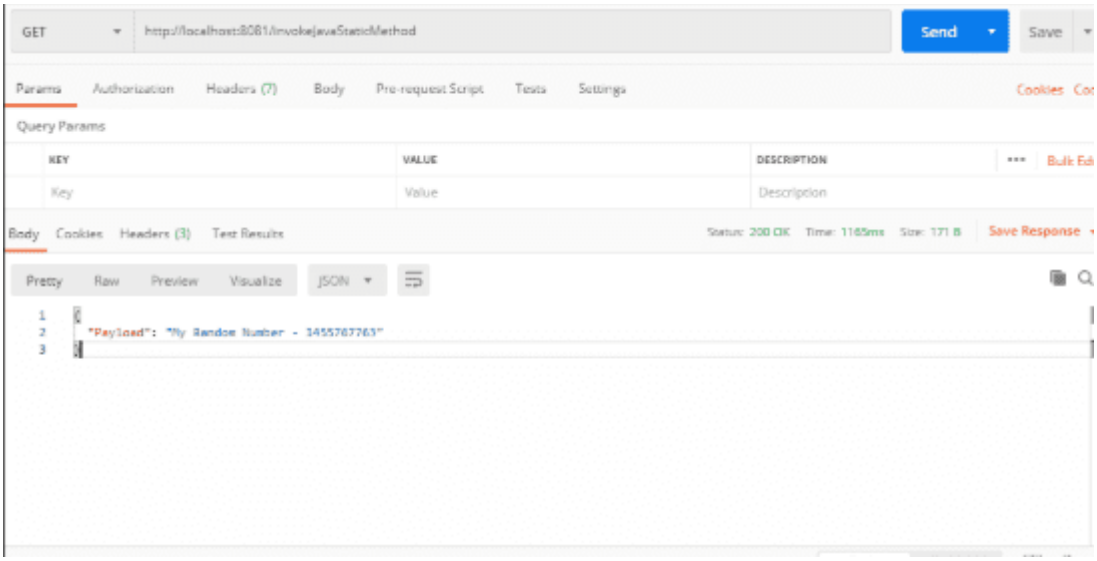
Step 3: DataWeave Script

- Craft a DataWeave script to call the static method `appendRandom()` within your Mule flow.



Step 4: Postman Console

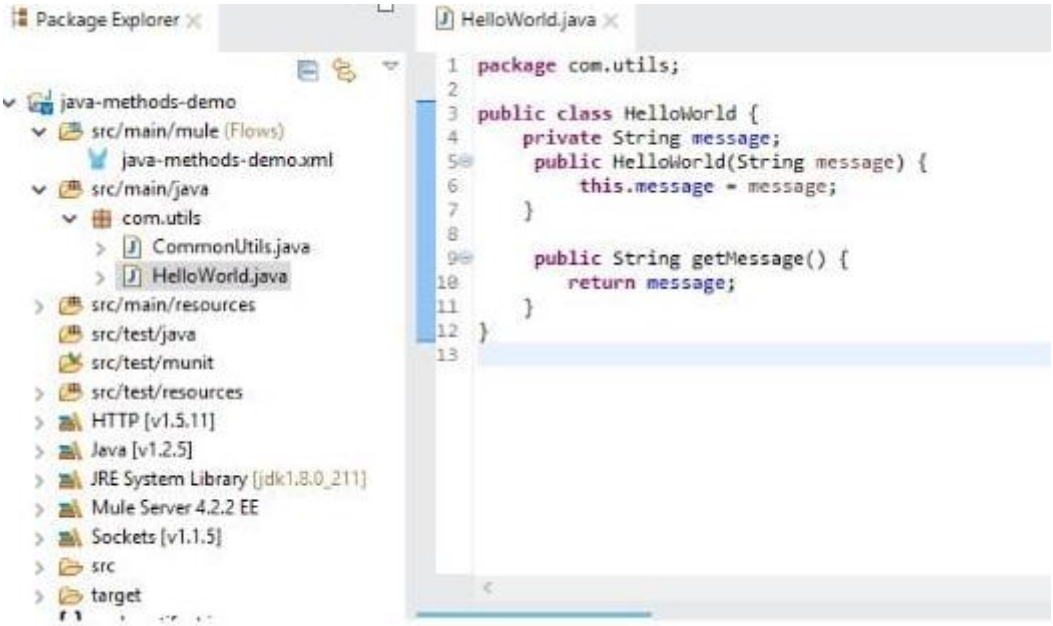
- Utilize Postman Console to test and validate your Mule flow.



Instantiate a Java Class

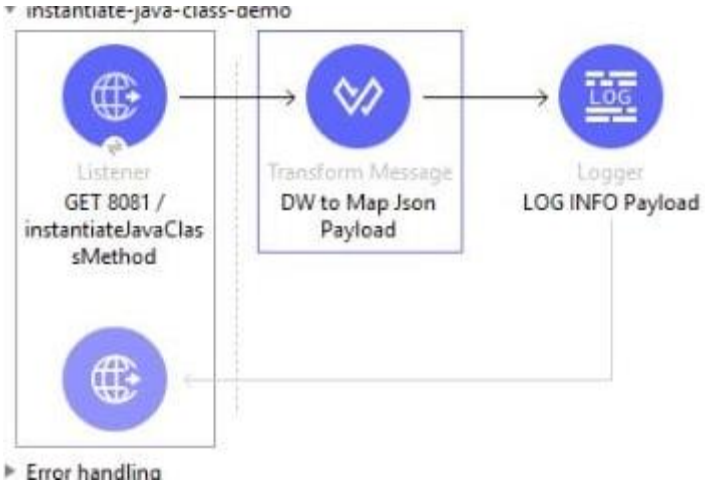
Step 1: Create a Java Class

- Develop a class within the "utils" package in the `src/main/java` directory of your Mule project.



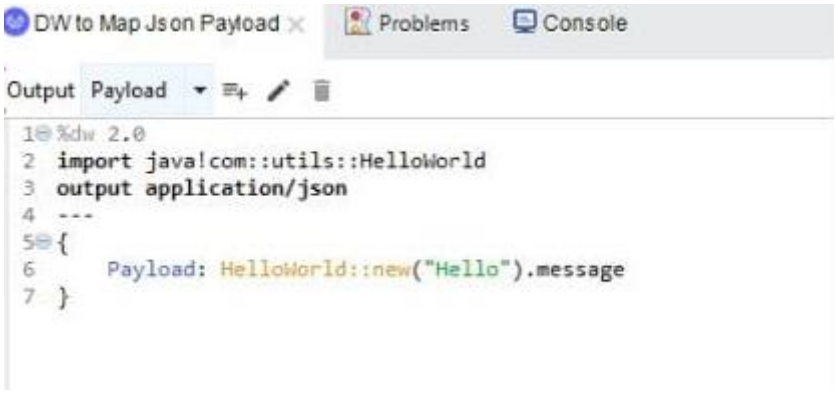
Step 2: Design Mule Flow

- Create a Mule flow to invoke the `appendRandom()` method from DataWeave.



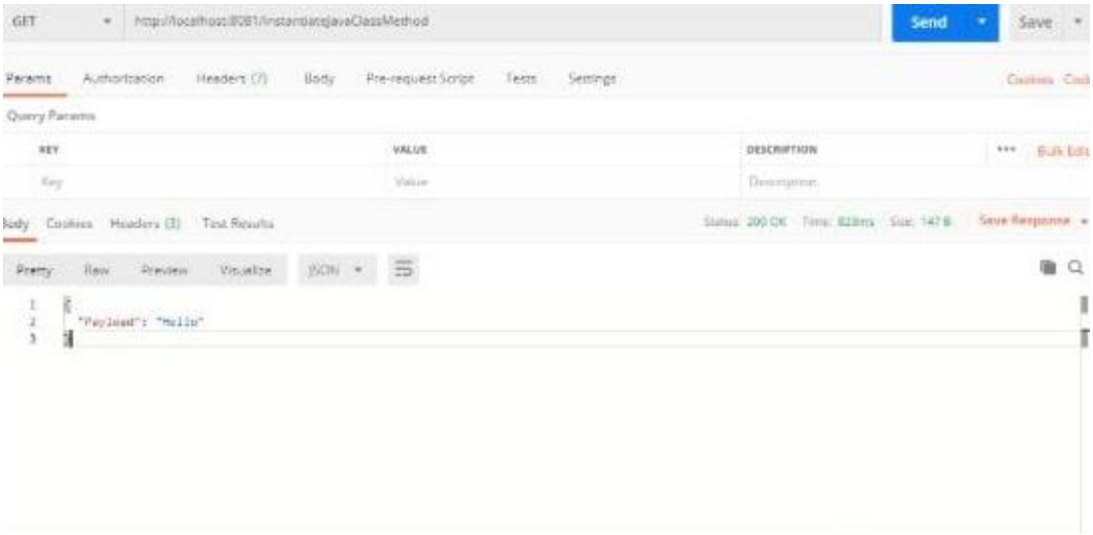
Step 3: DataWeave Script

- Construct a DataWeave script to invoke the `appendRandom()` method within your Mule flow.



Step 4: Postman Console

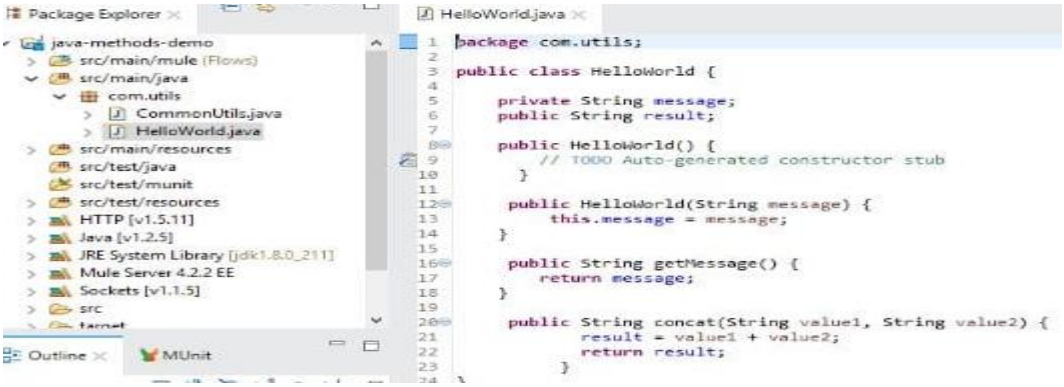
- Use Postman Console for testing and validation.



Invoke a Java Non-Static Method from a Class

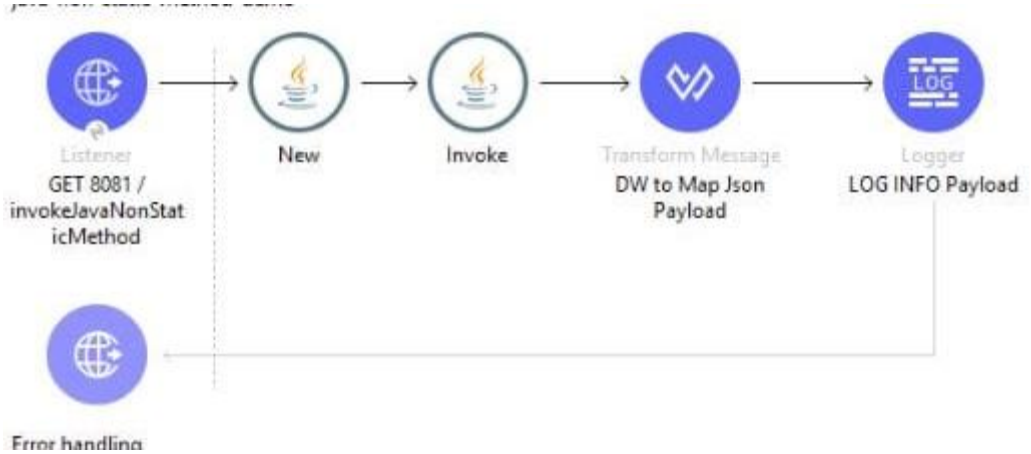
Step 1: Develop Java Class

- Create a class in the "utils" package in the `src/main/java` directory of your Mule project.



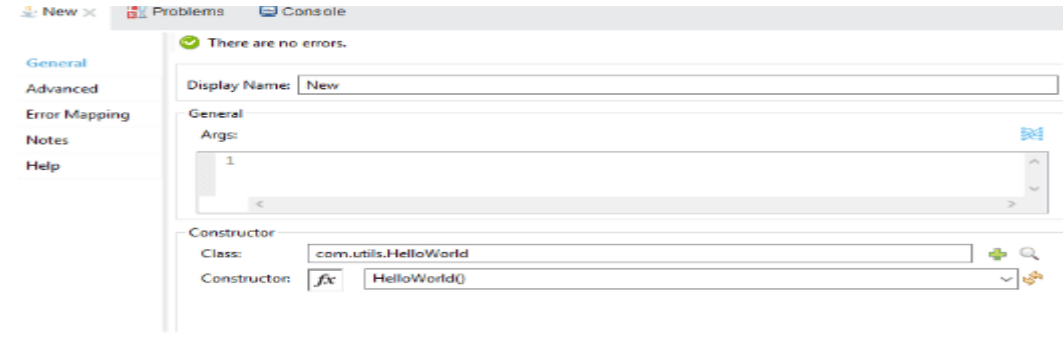
Step 2: Construct Mule Flow

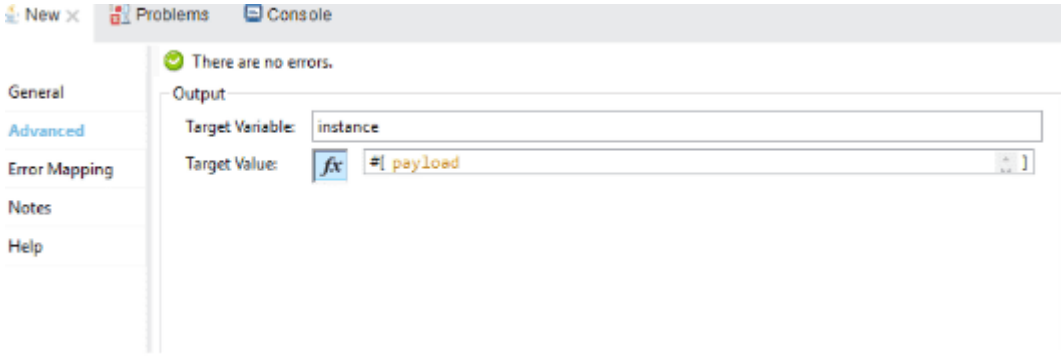
- Develop a Mule flow to call a non-static method. Instantiate the class using "New" and "Invoke" Java components from the Mule palette.

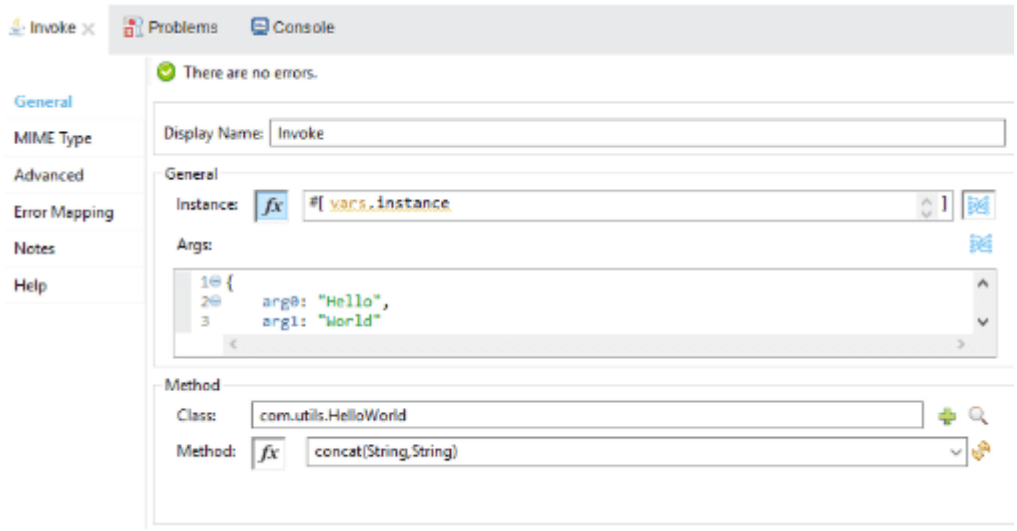


Step 3: Configure New and Invoke Java Components

- Configure the "New" and "Invoke" Java components within your Mule flow to create an instance and call the non-static function.

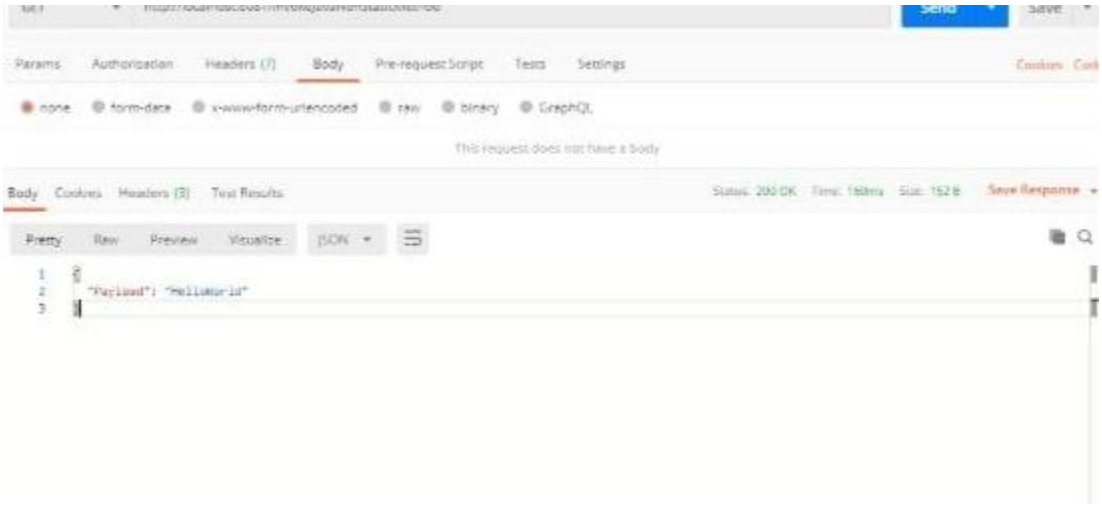






Step 4: Postman Console

- Employ Postman Console for testing and verification.



Reference

- https://docs.mulesoft.com/
- https://dataweave.mulesoft.com/
- https://www.youtube.com/

TGH

Making Integrations Simpler

TGH Software Solutions Pvt. Ltd.

www.techygeekhub.com

At TGH, we specialize in driving digital transformation through seamless Integration Technologies.

Operating as an INTEGRATION FACTORY, we serve as a one-stop shop for all your integration needs. Our expert team is well-versed in enterprise software and legacy system integration, along with leading iPaaS technologies like Boomi, MuleSoft, Workato, OIC, and more.

We're committed to enhancing business processes and solving problems through our integration expertise.

Email address

connect@techygeekhub.com

Phone number

+ 011-40071137

+ 91-8810610395

Our offices

Noida Office

iThum
Plot No -40, Tower A,
Office No: 712,
Sector-62, Noida,
Uttar Pradesh, 201301

Hyderabad Office

Plot no: 6/3, 5th Floor,
Techno Pearl Building,
HUDA Techno Enclave,
HITEC City, Hyderabad,
Telangana 500081