blueplanet®
a division of Ciena

Blue Planet Inventory Developer Advanced

BPI133ILT, Revision 1.0

Blue Planet Inventory

Hands-on Training Lab Guide

LEGAL NOTICES

THIS DOCUMENT CONTAINS CONFIDENTIAL AND TRADE SECRET INFORMATION OF CIENA CORPORATION AND ITS RECEIPT OR POSSESSION DOES NOT CONVEY ANY RIGHTS TO REPRODUCE OR DISCLOSE ITS CONTENTS, OR TO MANUFACTURE, USE, OR SELL ANYTHING THAT IT MAY DESCRIBE. REPRODUCTION, DISCLOSURE, OR USE IN WHOLE OR IN PART WITHOUT THE SPECIFIC WRITTEN AUTHORIZATION OF CIENA CORPORATION IS STRICTLY FORBIDDEN.

EVERY EFFORT HAS BEEN MADE TO ENSURE THAT THE INFORMATION IN THIS DOCUMENT IS COMPLETE AND ACCURATE AT THE TIME OF PUBLISHING; HOWEVER, THE INFORMATION CONTAINED IN THIS DOCUMENT IS SUBJECT TO CHANGE.

While the information in this document is believed to be accurate and reliable, except as otherwise expressly agreed to in writing CIENA PROVIDES THIS DOCUMENT "AS IS" WITHOUT WARRANTY OR CONDITION OF ANY KIND, EITHER EXPRESS OR IMPLIED. The information and/or products described in this document are subject to change without notice. For the most up-to-date technical publications, visit www.ciena.com.

Security

Ciena cannot be responsible for unauthorized use of equipment and will not make allowance or credit for unauthorized use or access.

Contacting Ciena

Corporate headquarters	410-694-5700 or 800-921-1144	www.ciena.com
Customer technical support/warranty		
In North America	1-800-CIENA24 (243-6224) 410-865-4961	Email: CIENA24@ciena.com
In Europe, Middle East, and Africa	800-CIENA-24-7 (800-2436-2247) +44-207-012-5508	Email: CIENA24@ciena.com
In Asia-Pacific	800-CIENA-24-7 (800-2436-2247) +81-3-6367-3989 +91-124-4340-600	Email: CIENA24@ciena.com
In Caribbean and Latin America	800-CIENA-24-7 (800-2436-2247) 410-865-4944 (USA)	Email: CIENA24@ciena.com
Sales and General Information	410-694-5700	E-mail: sales@ciena.com
In North America	410-694-5700 or 800-207-3714	E-mail: sales@ciena.com
In Europe	+44-207-012-5500 (UK)	E-mail: sales@ciena.com
In Asia	+81-3-3248-4680 (Japan)	E-mail: sales@ciena.com
In India	+91-124-434-0500	E-mail: sales@ciena.com
In Latin America	011-5255-1719-0220 (Mexico City)	E-mail: sales@ciena.com
Training		E-mail: learning@ciena.com

For additional office locations and phone numbers, please visit the Ciena website at www.ciena.com

Change History

Blue Planet Release	Revision	Publication Date	Reason for Change
22.08	1.0		Initial release.

Contents

Lab 1: BPI Customization Points

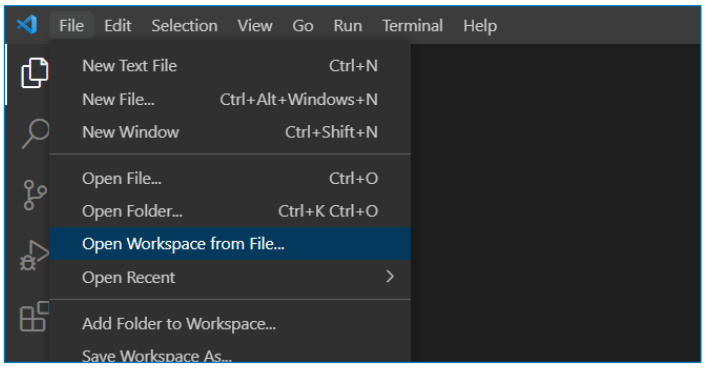
Objectives

- Learn how to set up VSC IDE for use with BPI area customizations
- Write and deploy Groovy scripts using IDE to implement business logic customizations
- Extend your BPI project by creating new Java classes, by extending the existing ones, to add new capabilities and customizations to the existing product
- Use Custartifacts loader together with Gitlab-based CI system to build and deploy business logic customizations
- Verify the intended operation of the customizations and describe the log file locations for basic troubleshooting

Task 1: Setup the IDE

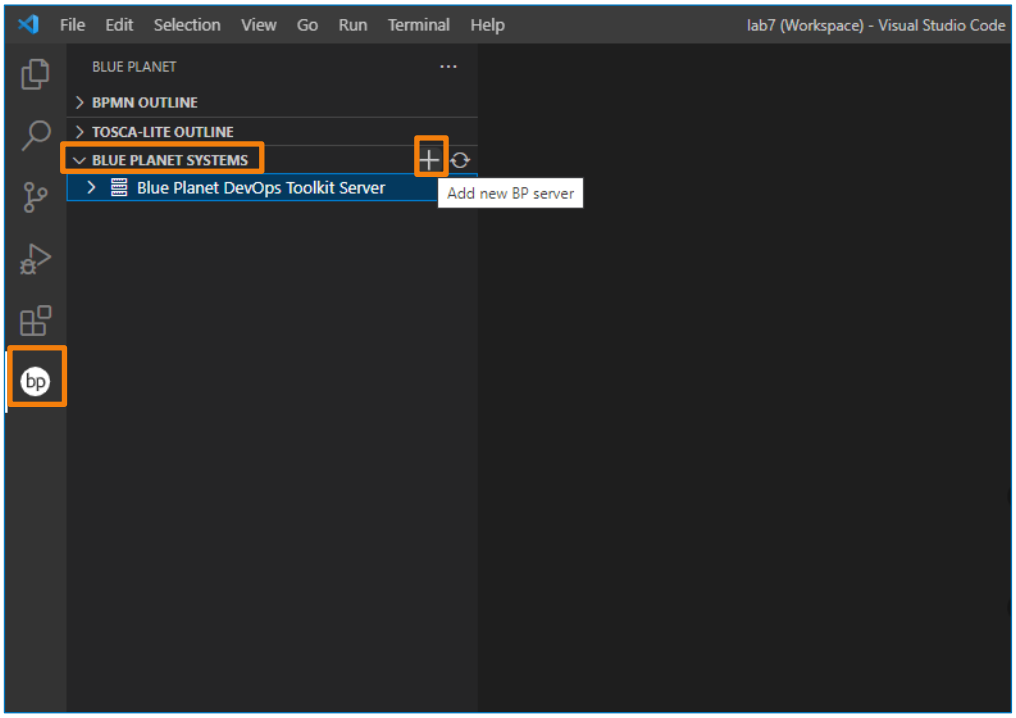
In this task, you set up the Visual Studio Code (VSC) integrated development environment (IDE) on your local environment that is going to be used to complete other exercises in your lab.

1. Open VSC from your desktop by clicking the **VSC** icon.
2. From the main menu, select **File > Open Workspace from File…** to open the workspace for this lab.

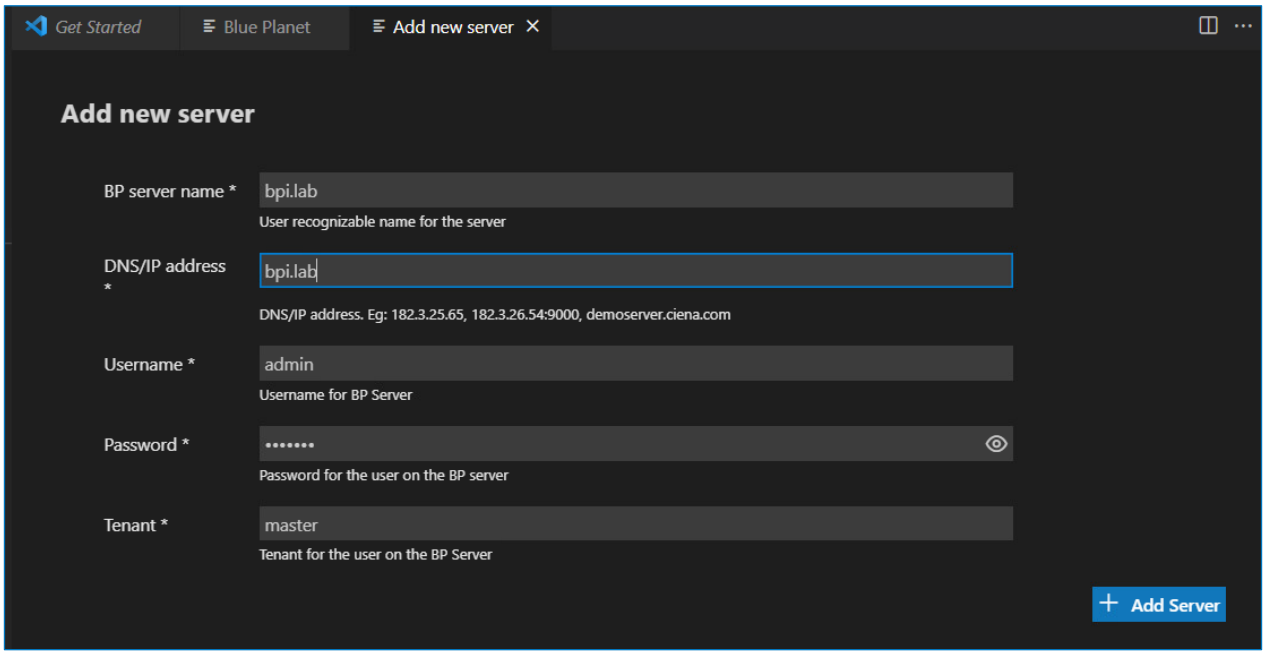


3. In the file browser, find the workspace file **C:\Users\student\Workspaces\Lab1\lab1.code-workspace** and click **Open**.

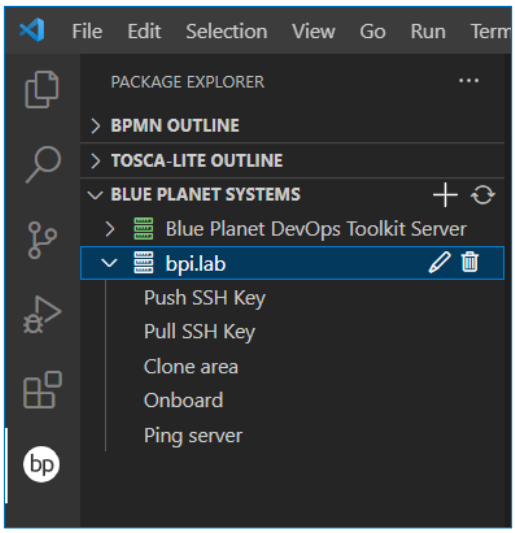
4. Add your BPI server instance in the VSC BP extension. First, click the **Blue Planet** menu option on the left side of VSC. Next, expand **Blue Planet Systems**, and finally, click the **+** button to add a new BP server.



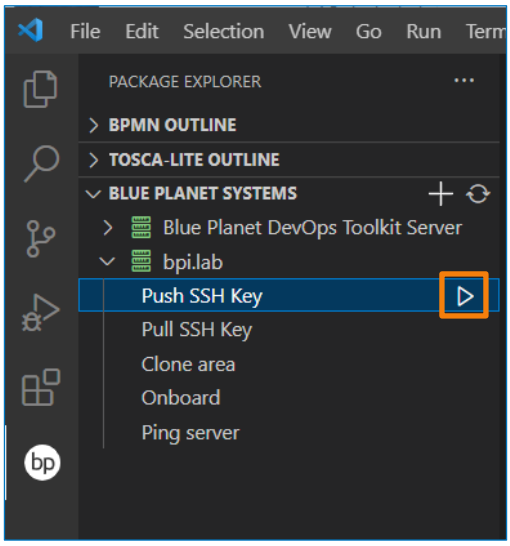
5. For the BPI server name, enter **bpi.lab**. For DNS/IP address enter **bpi.lab**. Leave other values as they are and press the **+ Add Server** button.



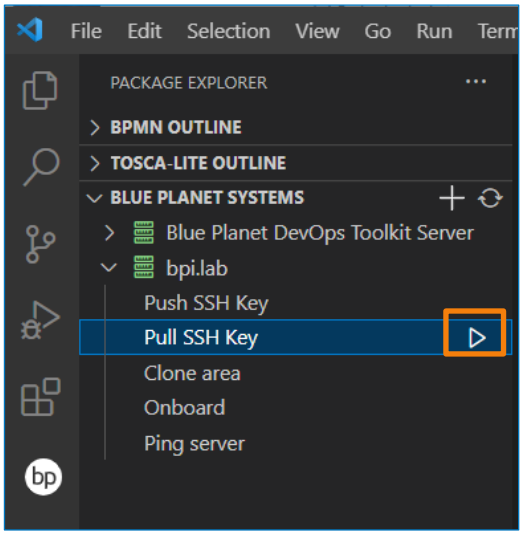
6. Your server is now defined and appears in the list under BLUE PLANET SYSTEMS.



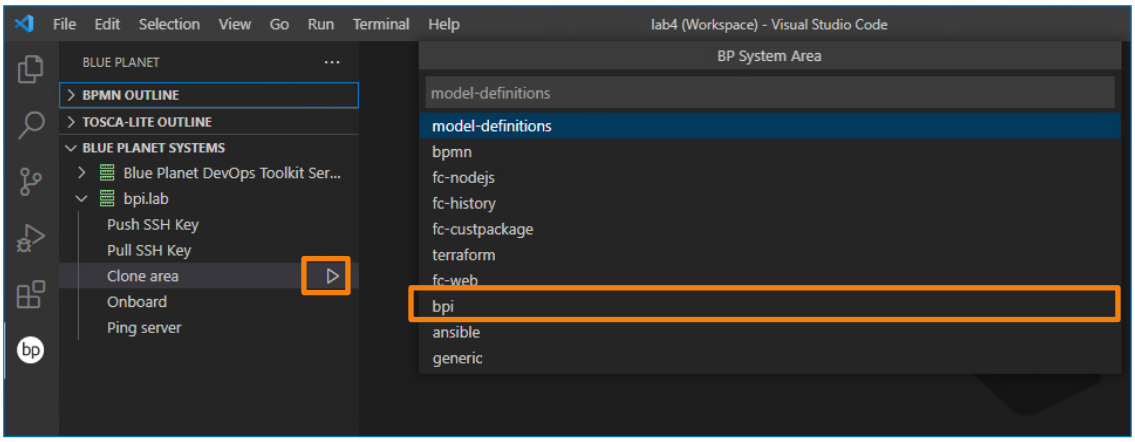
7. To set up secure communication with the server through certificates, click the **play icon** next to the Push SSH Key option, then press **ENTER** when asked for the key to use the default key location (**C:\Users\student\.ssh\id_rsa.pub**).



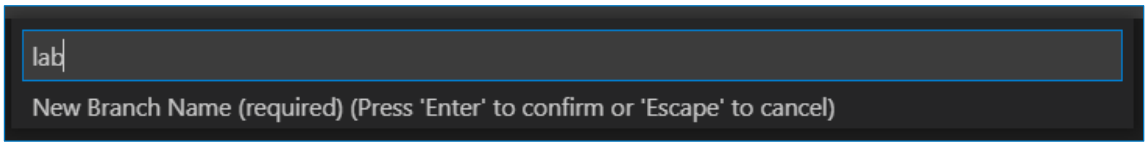
8. Next, pull the key from the server to your Student PC. Click the **play icon** next to the Pull SSH Key option, then press **ENTER** when asked to use the default location (**C:\Users\student\.ssh\known_hosts**).



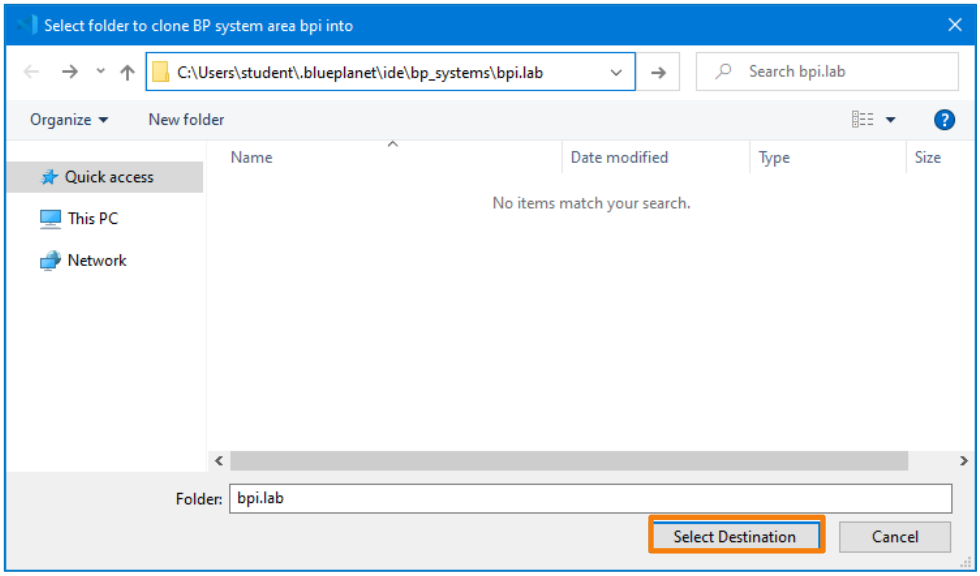
9. Next, clone the directory structure from the server to your VSC workspace. Click on the **play icon** next to **Clone area** and then select **bpi** when asked for BP System Area.



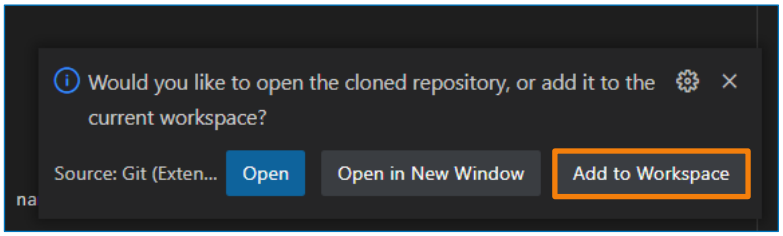
10. For the branch name, enter **lab** and press the **ENTER** key.



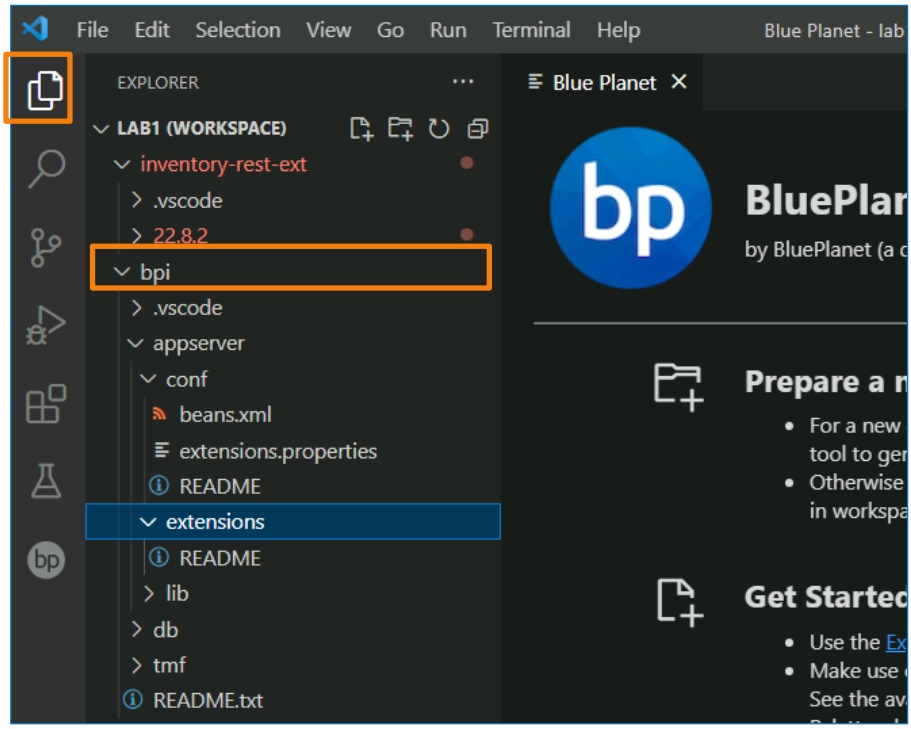
11. Select the offered default location by clicking on **Select Destination**. No other selection is required.



12. When asked, click the **Add to Workspace** button to add the directory structure to the VSC workspace.



13. The cloning process is now complete. Using Git, VSC cloned the bpi area files from your BPI server to your local Student PC. Your IDE is now ready. Click on **Explorer** from the left menu and expand **LAB1 (WORKSPACE)** to see the file structure. Expand **bpi > appserver > conf** and **bpi > appserver > extensions** to explore the directory structure. The Groovy script with your new business logic, that you create, will be placed in the extensions directory.



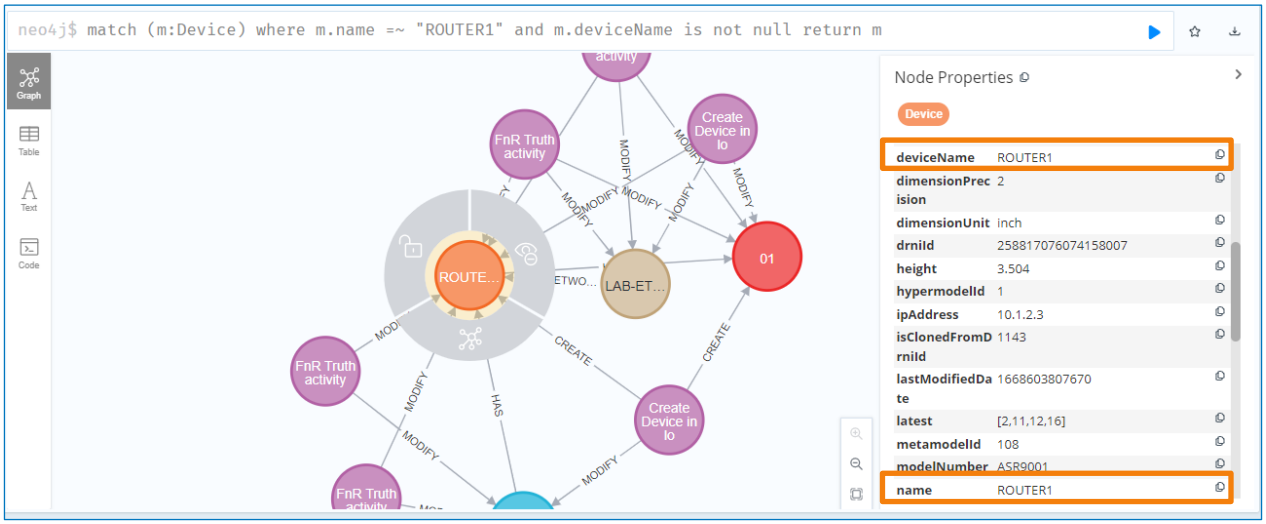
NOTE: In your workspace, there is also the source code of the **inventory-rest-ext** library present. You can use this tree to browse through the source code of various customization points that you can extend to modify the business logic. For example, as part of this exercise, you will modify the logic that is responsible for creating device names when creating new or editing existing devices.

Task 2: Device Naming Customization

In this task, you customize the business logic at the device naming extension point. When creating or modifying devices from BPI UI, after the Apply button is clicked, the deviceNaming bean is executed. Extending the existing DeviceNaming class, and overriding methods of interest, allows you to programmatically influence the behavior of BPI when it comes to naming devices that you are currently working on. In this exercise you implement the following business logic for device naming after the information has been input in the UI:

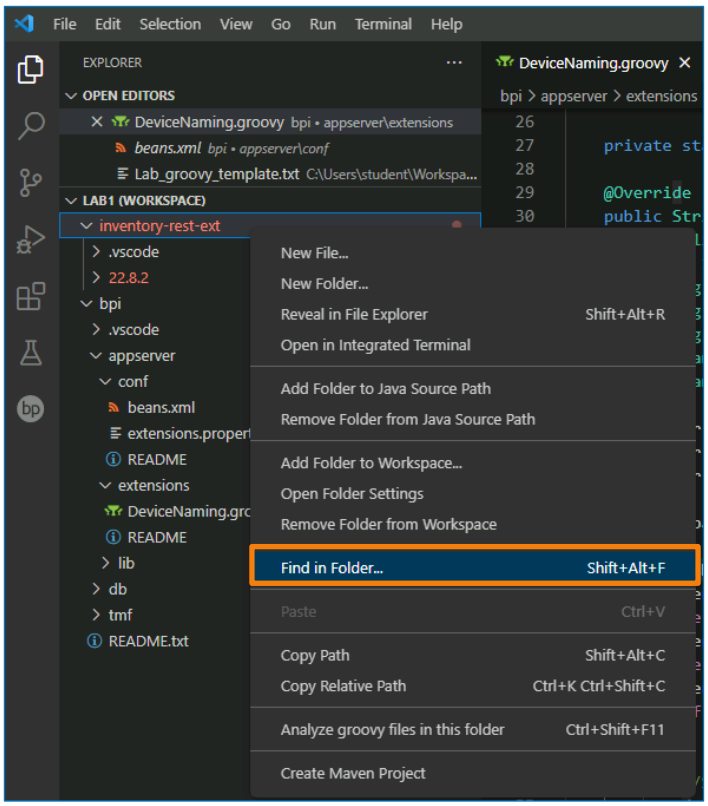
- If the device name is absent, generate a node name that consists of the current location (building name) followed by the device type, for example, “Building1_ASR9001”.
- If the device name is present, generate a node name that is equal to the device name from the UI field, for example, “Router1”.
- If the device name is present, but the IP address is also present, generate a node name that consists of the name and the IP address, for example, “Router1_10.0.0.45”.

Here is an example from the Neo4j database of a newly created device.

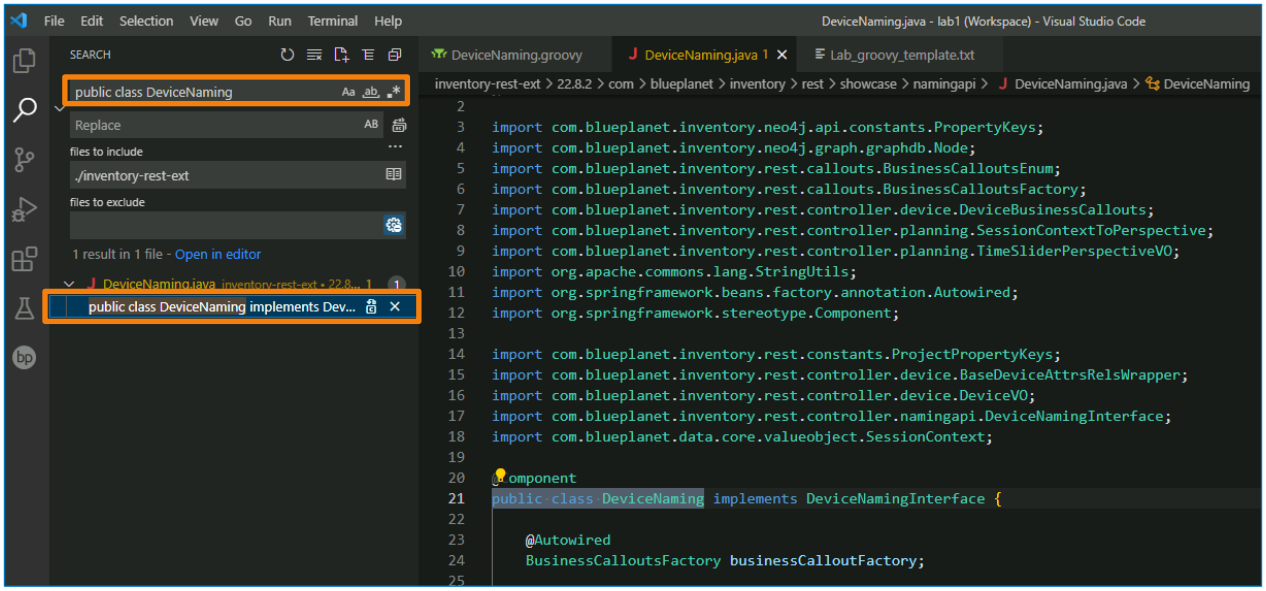


Note how there are two distinct fields, **name**, and **deviceName**, where name refers to the node name in the database, and deviceName is a separate field, typically representing the hostname of the device. Those two parameters are made the same by default, but they can be set independently of each other as you will learn in this lab exercise. The node name value is taken into consideration when using the global search in the BPI UI.

1. From VSC, find the current implementation of the device naming interface, the DeviceNaming Java class.



2. In the search field enter **public class DeviceNaming**. The search result will show the DeviceNaming.java file. Click on the search result and the file will open in the right pane. The class **DeviceNaming** implements the **DeviceNamingInterface** interface, and you will modify the logic by extending from this class in this lab exercise.

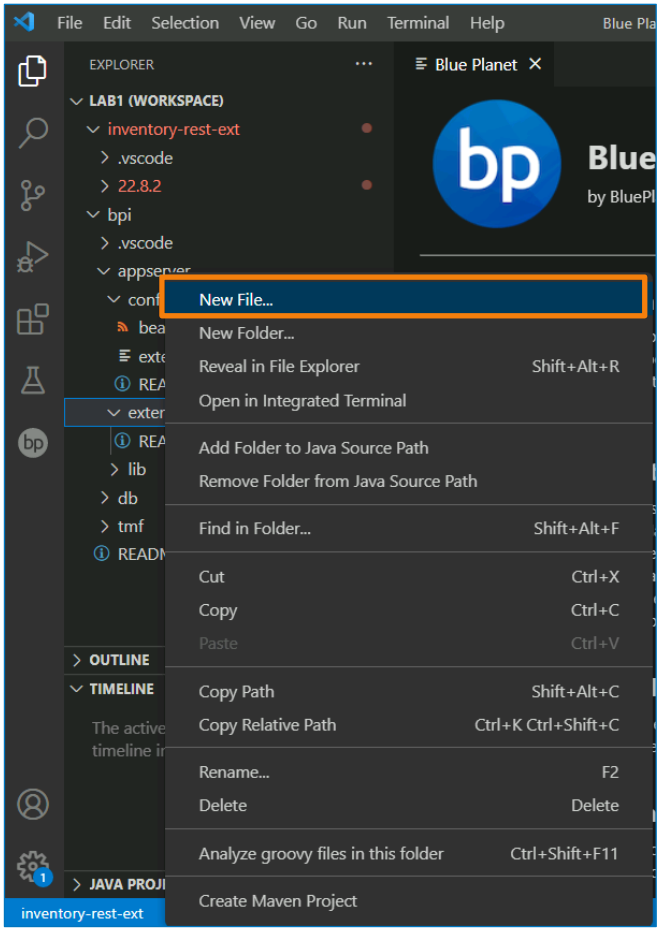


The com.blueplanet.inventory.rest.controller.namingapi.DeviceNamingInterface from the above class extends the interface com.blueplanet.inventory.model.publicapi.naming. DeviceNamingInterface. The implementation requires only one method, **generateName**, with two parameters, DeviceVO and SessionContext. The interface code is as follows:

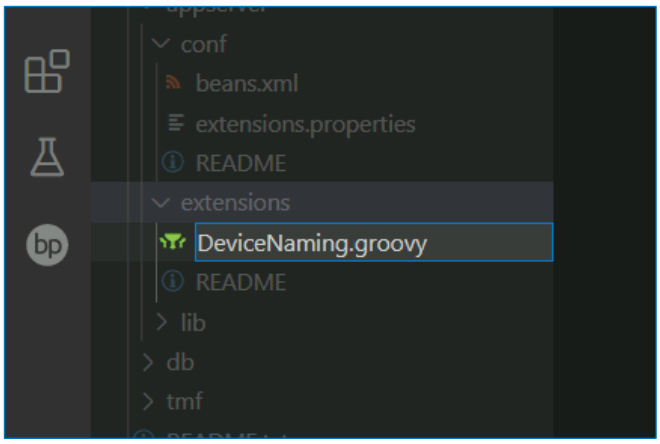
```
package com.blueplanet.inventory.model.publicapi.naming;

public abstract interface DeviceNamingInterface {


  public abstract java.lang.String
    generateName(com.blueplanet.inventory.rest.controller.device.DeviceV
    O arg0, com.blueplanet.data.core.valueobject.SessionContext arg1);

}
```

NOTE: Examine the current logic of the class by inspecting the code. Later you will modify the naming logic in a Groovy script and then you can compare the files. The Groovy script will look very similar to the Java file with a few basic modifications.

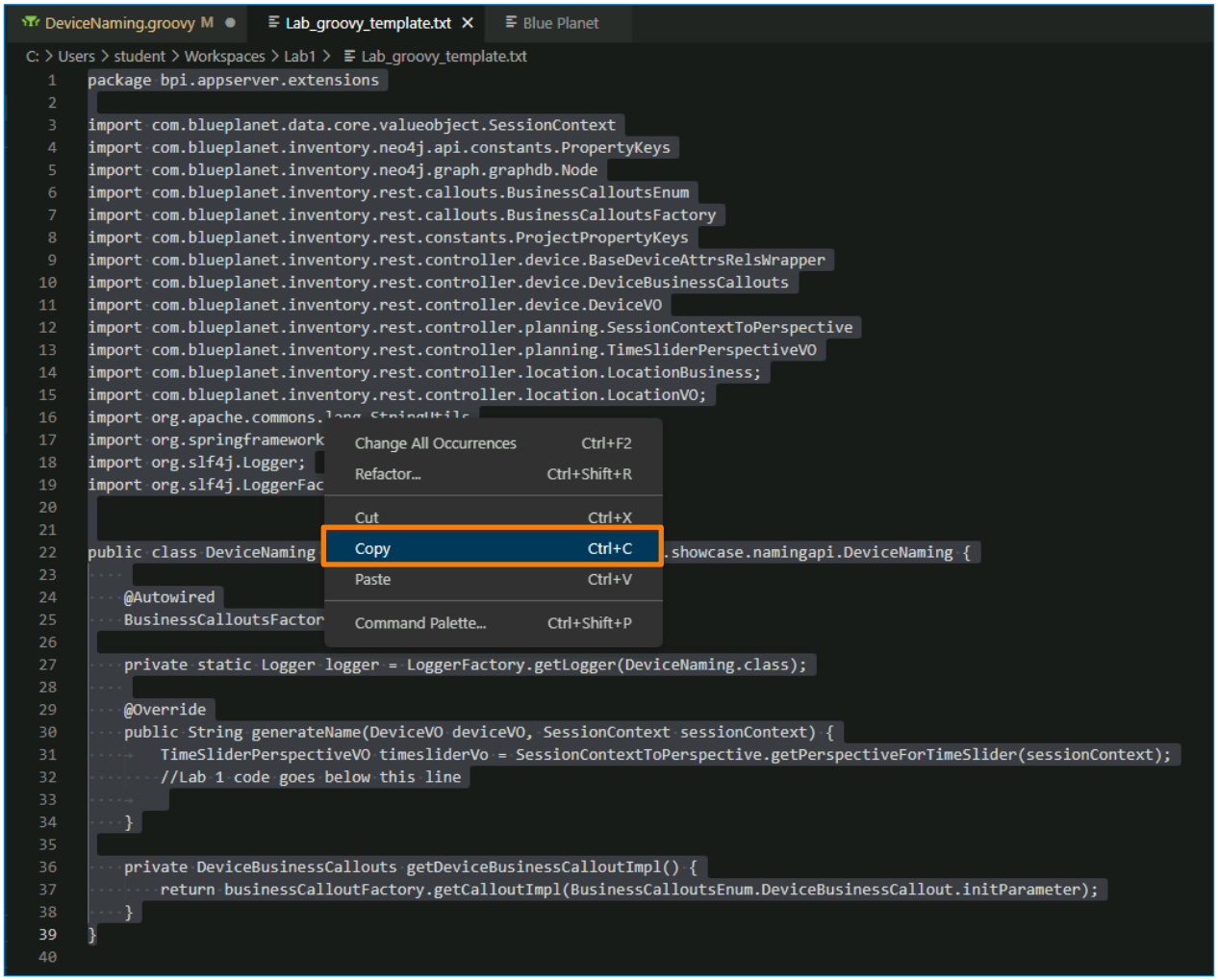
3. Now proceed with the customization. Start by creating a new file in your workspace. Right-click on **bpi/appserver/extensions** and select the **New File…** option.



4. For the file name, enter **DeviceNaming.groovy**. Be careful to correctly enter the name in the proper case.



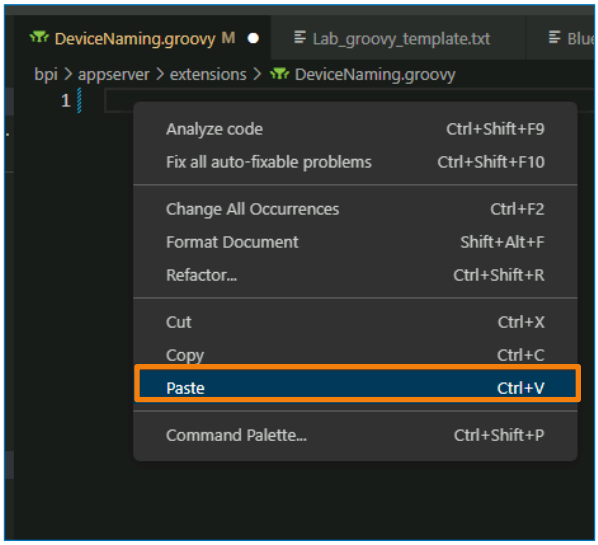
5. On your student PC, open the file explorer, navigate to **C:\Users\student\Workspaces\Lab1**, and open the file **Lab_groovy_template.txt** by double-clicking it. This file represents the skeleton of your new Groovy script, you will add code to it in subsequent steps. Now select all text from the file and copy it to the clipboard.




```
package bpi.appserver.extensions

import com.blueplanet.data.core.valueobject.SessionContext
import com.blueplanet.inventory.neo4j.api.constants.PropertyKeys
import com.blueplanet.inventory.neo4j.graph.graphdb.Node
import com.blueplanet.inventory.rest.callouts.BusinessCalloutsEnum
import com.blueplanet.inventory.rest.callouts.BusinessCalloutsFactory
import com.blueplanet.inventory.rest.constants.ProjectPropertyKeys
import com.blueplanet.inventory.rest.controller.device.BaseDeviceAttrsRelsWrapper
import com.blueplanet.inventory.rest.controller.device.DeviceBusinessCallouts
import com.blueplanet.inventory.rest.controller.device.DeviceVO
import com.blueplanet.inventory.rest.controller.planning.SessionContextToPerspective
import com.blueplanet.inventory.rest.controller.planning.TimeSliderPerspectiveVO
import com.blueplanet.inventory.rest.controller.location.LocationBusiness;
import com.blueplanet.inventory.rest.controller.location.LocationVO;
import org.apache.commons.lang.StringUtils
import org.springframework
import org.slf4j.Logger;
import org.slf4j.LoggerFac

public class DeviceNaming                                    .showcase.namingapi.DeviceNaming {

    @Autowired
    BusinessCalloutsFactor

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

    @Override
    public String generateName(DeviceVO deviceVO, SessionContext sessionContext) {
        TimeSliderPerspectiveVO timesliderVo = SessionContextToPerspective.getPerspectiveForTimeSlider(sessionContext);
        //Lab 1 code goes below this line

    }

    private DeviceBusinessCallouts getDeviceBusinessCalloutImpl() {
        return businessCalloutFactory.getCalloutImpl(BusinessCalloutsEnum.DeviceBusinessCallout.initParameter);
    }
}

```

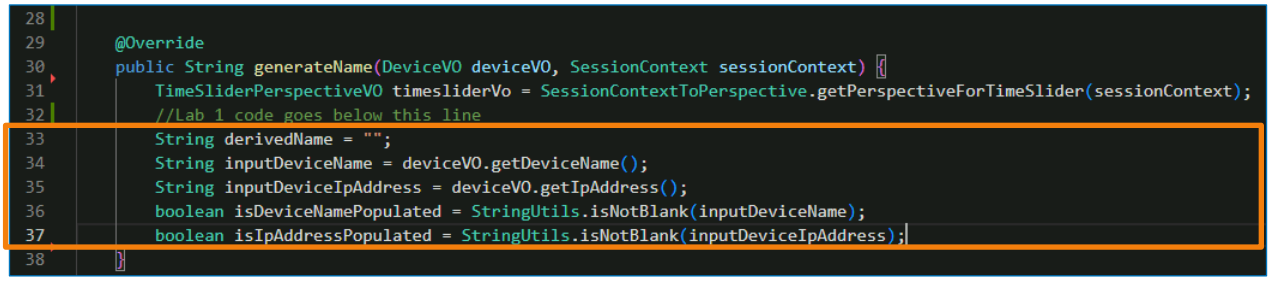
NOTE: At this point, you can spot several differences between the original implementation of DeviceNaming.java and this Groovy script file. Namely, the package is now defined as bpi.appserver.extensions, and this is the most important difference. Also, note the omission of @Component annotation in the Groovy script version. Some of the import lines may also differ but this is purely because of differences in implementation code, and not necessarily Groovy specific.

6. In VSC, click the **DeviceNaming.groovy** tab and paste the code.



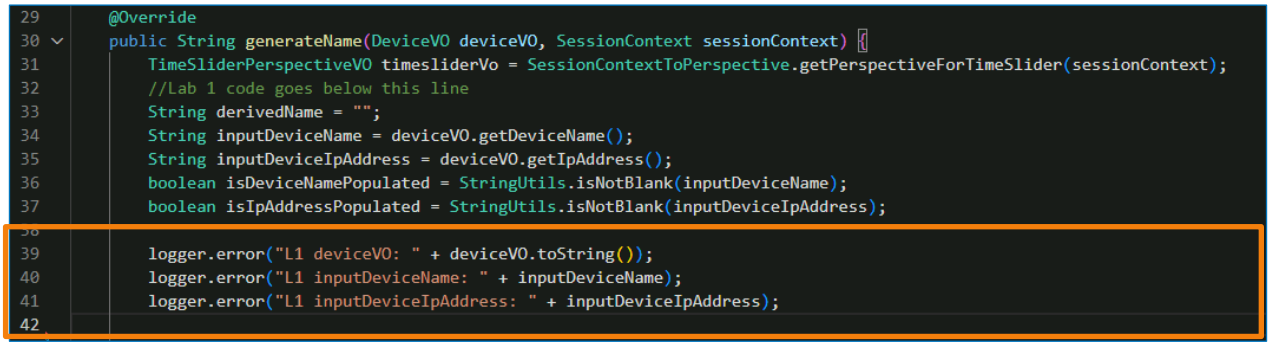
7. To implement the interface, you must override the generateName method. The method definition is already in the file, start writing the following code below line 32. First, declare the variables that will be used in the script.
   - **derivedName:** the device object name which is returned by the method.
   - **inputDeviceName:** the text that you entered for the “Device Name” in the UI.
   - **inputDeviceIpAddress:** the text that you entered for the “IP Address” in the UI.
   - **isDeviceNamePopulated**, **isIpAddressPopulated:** Booleans to signify if the previous values are empty or not.

```
String derivedName = "";
String inputDeviceName = deviceVO.getDeviceName();
String inputDeviceIpAddress = deviceVO.getIpAddress();
boolean isDeviceNamePopulated = StringUtils.isNotBlank(inputDeviceName);
boolean isIpAddressPopulated = StringUtils.isNotBlank(inputDeviceIpAddress);
```



8. Next, set up logging statements in lines 39-41 to help you troubleshoot if something goes wrong. We are using the error severity for visibility; in production, you can use any severity according to your requirements. The messages will be logged in the web-0 container in the /bp2/log/catalina.out file. deviceVO is the object that stores many values about the current device, including the device name and the IP address.

```
logger.error("L1 deviceVO: " + deviceVO.toString());
logger.error("L1 inputDeviceName: " + inputDeviceName);
logger.error("L1 inputDeviceIpAddress: " + inputDeviceIpAddress);
```



```
29      @Override
30      public String generateName(DeviceVO deviceVO, SessionContext sessionContext) {
31          TimeSliderPerspectiveVO timesliderVo = SessionContextToPerspective.getPerspectiveForTimeSlider(sessionContext);
32          //Lab 1 code goes below this line
33          String derivedName = "";
34          String inputDeviceName = deviceVO.getDeviceName();
35          String inputDeviceIpAddress = deviceVO.getIpAddress();
36          boolean isDeviceNamePopulated = StringUtils.isNotBlank(inputDeviceName);
37          boolean isIpAddressPopulated = StringUtils.isNotBlank(inputDeviceIpAddress);
38
39          logger.error("L1 deviceVO: " + deviceVO.toString());
40          logger.error("L1 inputDeviceName: " + inputDeviceName);
41          logger.error("L1 inputDeviceIpAddress: " + inputDeviceIpAddress);
42
```

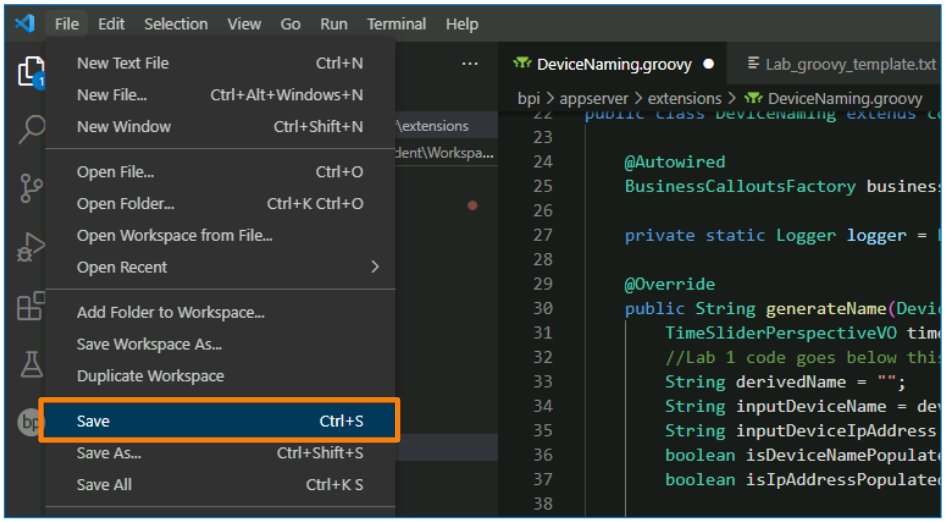
9. Next, enter the following lines. The first part is getting the **parentLocation**, which in this case is the name of the building. This value is used for the conditional construction of the derived name. The if-else block contains the logic of how the name of the device object is derived. The logic is explained in the introduction of this task. In the end, after the if-else block, the **derivedName** is returned.

```
    Node parentLocation =
getDeviceBusinessCalloutImpl().getDeviceLocationBasedOnInputs(deviceVO,
timesliderVo, sessionContext);


    if (isDeviceNamePopulated && isIpAddressPopulated) {
        derivedName = inputDeviceName + "_" + inputDeviceIpAddress;
    } else  if (isDeviceNamePopulated) {
        derivedName = inputDeviceName;
    } else { //Make derivedName based on deviceType and parent Location Name
        derivedName = deviceVO.getDeviceType().getLabel();
        if(parentLocation != null) {
            derivedName =
parentLocation.getPropertyAsString(PropertyKeys.name) + "_" + derivedName;
        }
        //Set the deviceName with derived deviceName if it wasn't provided as
input
        deviceVO.setDeviceName(derivedName);
    }
    return derivedName;
```

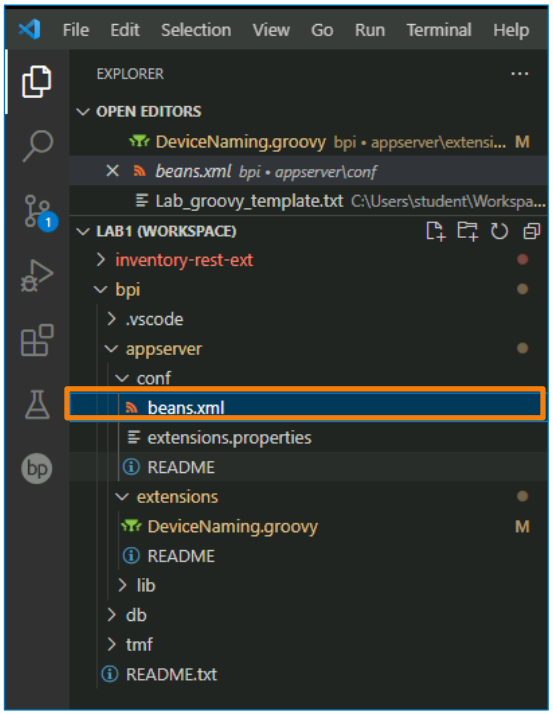
```
39        logger.error("L1 deviceVO: " + deviceVO.toString());
40        logger.error("L1 inputDeviceName: " + inputDeviceName);
41        logger.error("L1 inputDeviceIpAddress: " + inputDeviceIpAddress);
42
43        Node parentLocation = getDeviceBusinessCalloutImpl().getDeviceLocationBasedOnInputs(deviceVO, timesliderVo, sessionContext);
44
45        if (isDeviceNamePopulated && isIpAddressPopulated) {
46            derivedName = inputDeviceName + "_" + inputDeviceIpAddress;
47        } else  if (isDeviceNamePopulated) {
48            derivedName = inputDeviceName;
49        } else { //Make derivedName based on deviceType and parent Location Name
50            derivedName = deviceVO.getDeviceType().getLabel();
51            if(parentLocation != null) {
52                derivedName = parentLocation.getPropertyAsString(PropertyKeys.name) + "_" + derivedName;
53            }
54            //Set the deviceName with derived deviceName if it wasn't provided as input
55            deviceVO.setDeviceName(derivedName);
56        }
57        return derivedName;
58    }
```

NOTE: Make sure there are no obvious syntactic errors that the IDE would highlight for you. Also, be careful not to introduce typos.

10. Your Groovy script is now done. Save your work by selecting **File > Save** from the main menu, or alternatively by pressing **CTRL+S**.

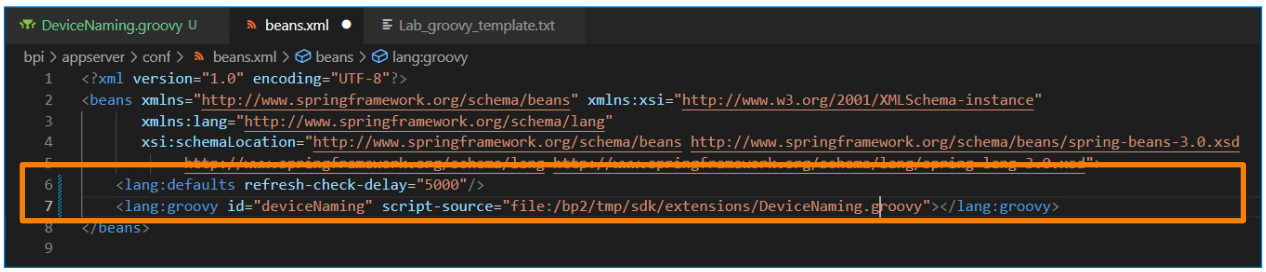


11. Before the changes are onboarded, you must modify the beans.xml file where you will register your new bean with the script's location on the server. From your workspace, click **bpi > appserver > conf > beans.xml** to open the file.



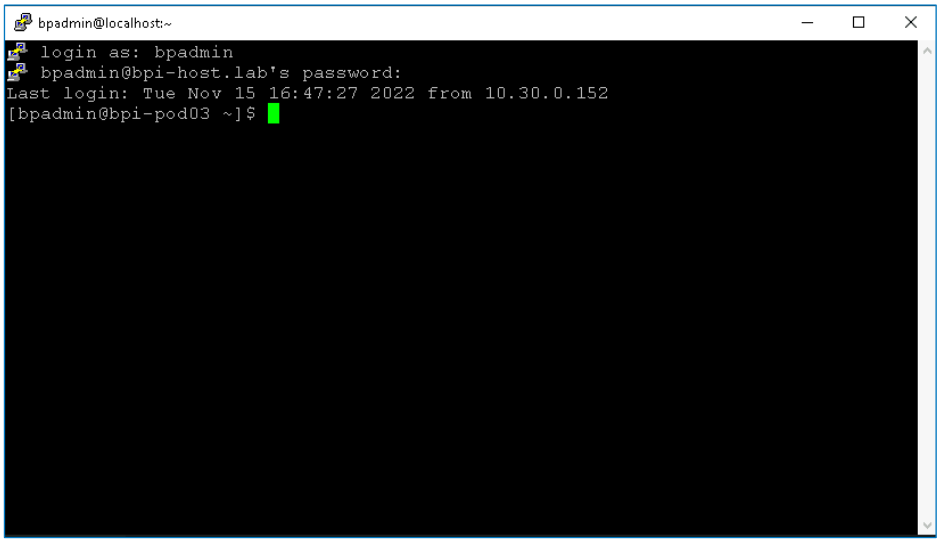
12. Enter the following statements in lines 6-7:

```
<lang:defaults refresh-check-delay="5000"/>

<lang:groovy id="deviceNaming" script-
   source="file:/bp2/tmp/sdk/extensions/DeviceNaming.groovy"></lang:gro
   ovy>
```



13. Save the changes with **CTRL+S**. Local files are now ready.

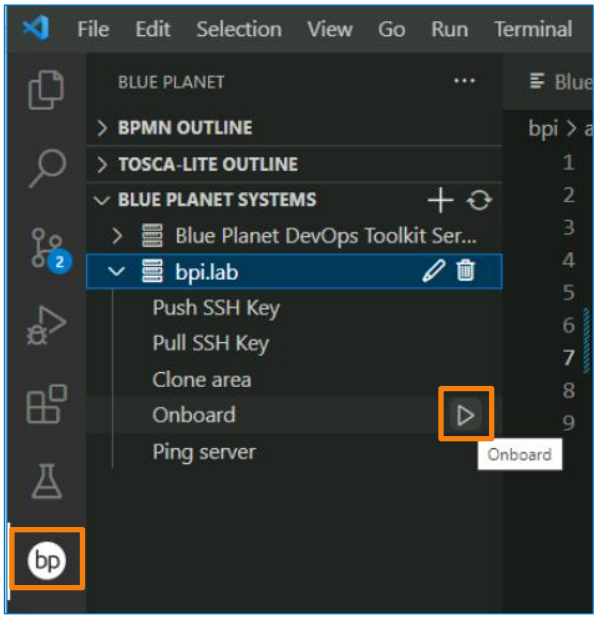
14. Before onboarding, use the Putty software from your desktop to open a terminal session to your BPI server. In Putty, select **bpi-host.lab** and click **Open**. Use **bpadmin/bpadminpw** for credentials.



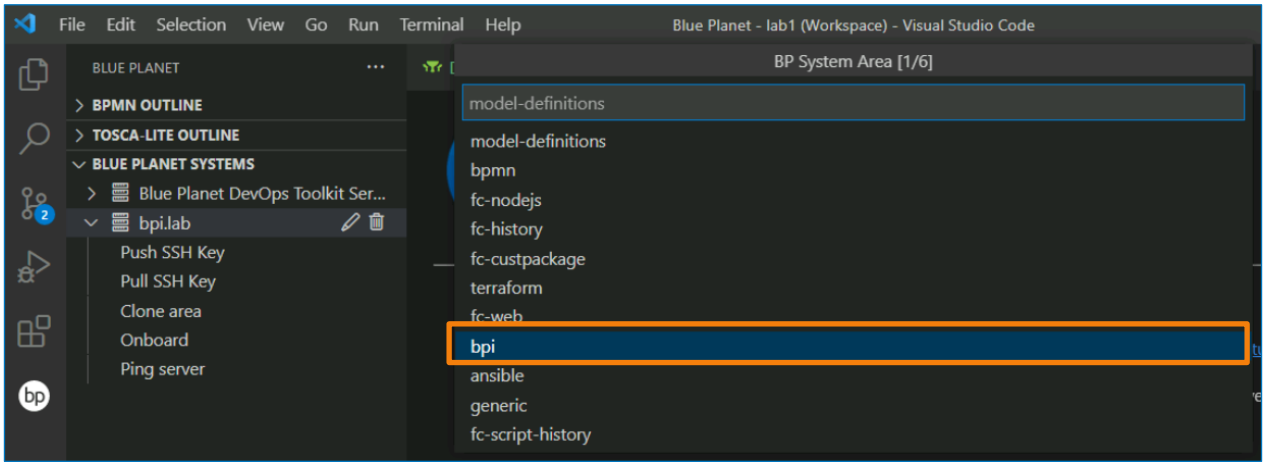
15. Connect to the web-0 container and set up log monitoring by entering the following commands in the terminal:

```
[bpadmin@bpi-pod03 log]$ kubectl exec -it web-0 -- bash
root@web-0:/dev/shm# cd /bp2/log
root@web-0:/bp2/log# tail -f catalina.out | grep onboard
```

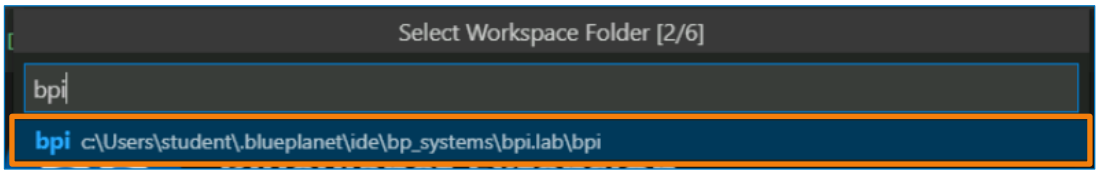
16. Return to VSC. From the menu on the left click the **Blue Planet** icon. Under BLUE PLANET SYSTEMS, bpi.lab, click the **triangle** icon next to the Onboard option to onboard the files to the server.



17. Click **bpi** to select the appropriate BP System Area.

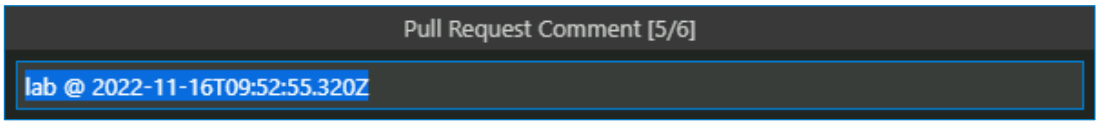


18. Select **bpi** for Select Workspace Folder. Be careful not to select other options if available.

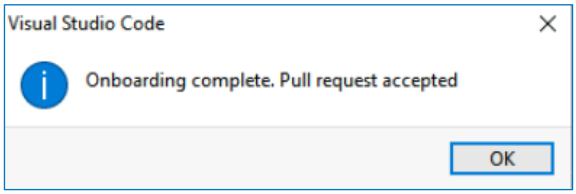


NOTE: If you want to cancel the process, press the **ESC** key at any time.

19. When asked for Pull Request Comment, just press the **ENTER** key.



20. When onboarding is complete you will be notified. Click **OK**.



21. Return to the server terminal window that you opened earlier. If onboarding was successful, log lines should appear similar to the following:

```
INFO : 2022-11-14 14:18:16,847 : kafkaConsumerTaskExecutor-1 :
   com.blueplanet.inventory.asset.sdk.onboard.service.OnboardService :
   Asset onboarding started

<…output omitted…>

INFO : 2022-11-14 14:18:16,968 : kafkaConsumerTaskExecutor-1 :
   com.blueplanet.inventory.asset.sdk.onboard.util.OnboardUtil :
   Checking given directory path is valid

INFO : 2022-11-14 14:18:16,971 : kafkaConsumerTaskExecutor-1 :
   com.blueplanet.inventory.asset.sdk.onboard.util.OnboardUtil :
   Checkout remote git repo to : /bp2/tmp/sdk-temp//bpi
```

```
INFO : 2022-11-14 14:18:16,980 : kafkaConsumerTaskExecutor-1 :
   com.blueplanet.inventory.asset.sdk.onboard.util.OnboardUtil :
   Pulling remote git repo.

INFO : 2022-11-14 14:18:17,431 : kafkaConsumerTaskExecutor-1 :
   com.blueplanet.inventory.asset.sdk.onboard.util.OnboardUtil :
   Closing repository.
```

22. In the terminal, press **CTRL+C** to exit the tail. Exit from the container with **exit**.
23. Restart the web-0 container for your customizations to take effect. Use the following command to restart web-0:

```
[bpadmin@bpi-pod03 ~]$ kubectl delete pod web-0

pod "web-0" deleted
```

24. Wait for 3-5 minutes for web-0 to fully initialize. You can check Nagios for the status of web-0. Leave your server terminal open for the subsequent tasks.

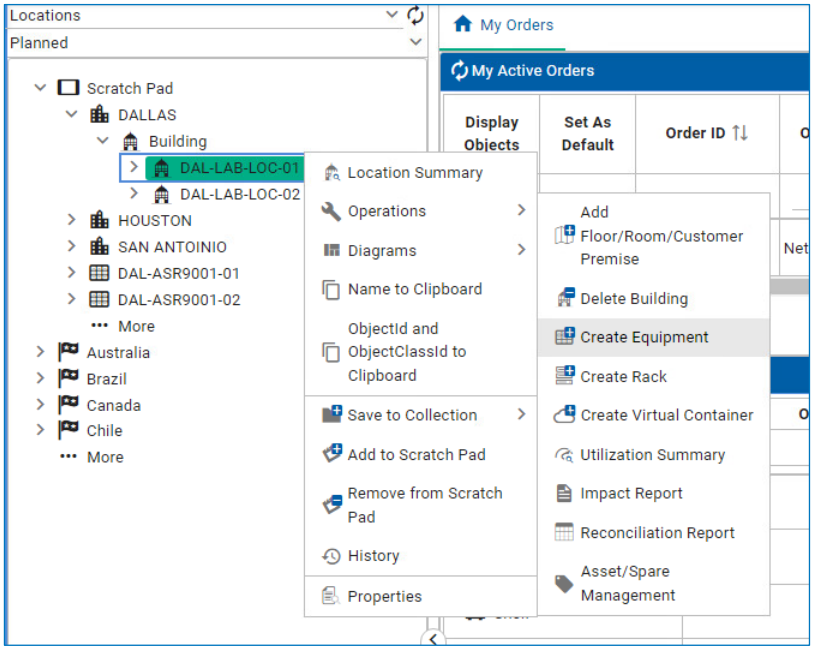
Task 3: Verify the New Business Logic

In this task, you will verify the device naming logic that you implemented in the previous task.

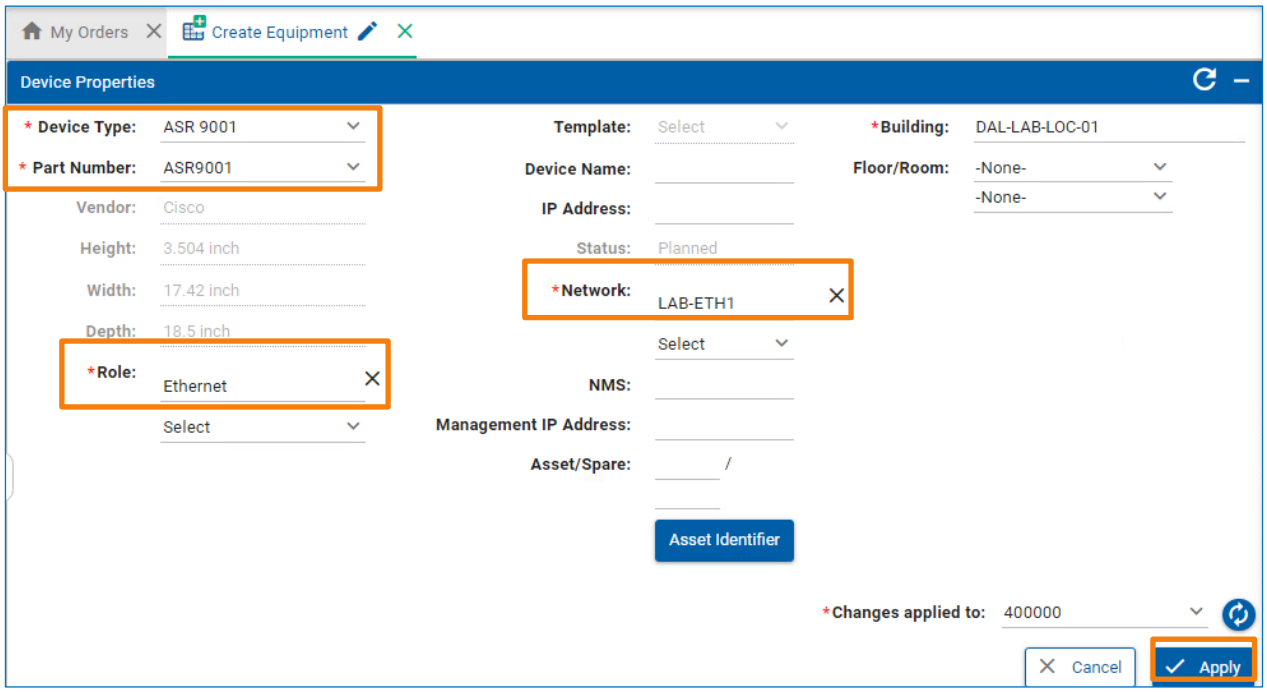
1. In the server terminal, enter the web-0 container again and set up log monitoring:

```
[bpadmin@bpi-pod03 log]$ kubectl exec -it web-0 -- bash
root@web-0:/dev/shm# tail -f /bp2/log/catalina.out | grep "ERROR.*L1 "
```

2. Using a web browser, log in to your BPI UI with the credentials **normal/12345678**. In your scratch pad, locate building **DAL-LAB-LOC-01** in Dallas and create a new device by right-clicking **DAL-LAB-LOC-01** and selecting **Operations > Create Equipment**.



3. Populate the UI form with the following information and click **Apply**:
   Device Type: **ASR 9001**
   Part Number: **ASR 9001**
   Role: **Ethernet**
   Network: **LAB-ETH1**

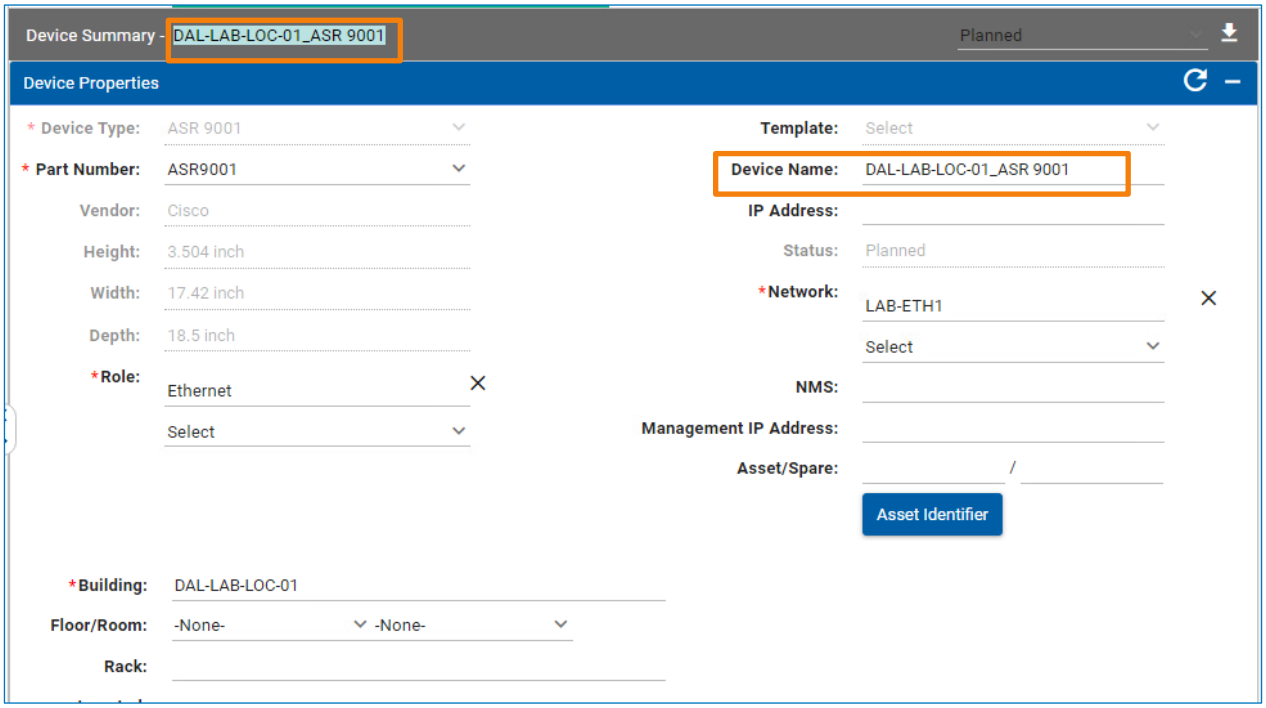


4. In the terminal window, observe the log output. The log lines that you coded in the DeviceNaming class are now producing the output. The first line shows the contents of the deviceVO object, and the following lines show your input for the Device Name and IP Address fields in the UI. They are now empty as you did not enter any values for them.

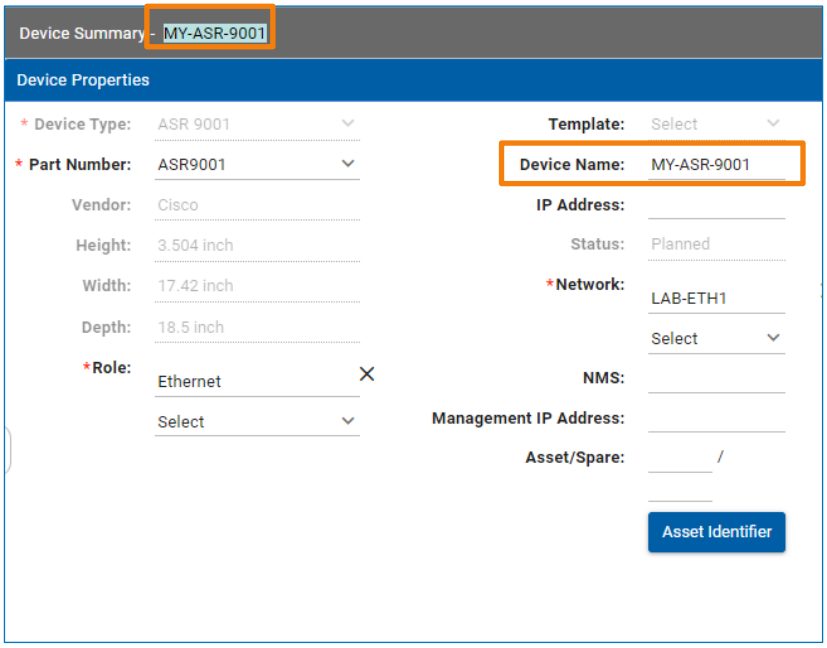
```
ERROR: 2022-11-14 14:35:00,460 : http-nio-8080-exec-4 :
   bpi.appserver.extensions.DeviceNaming : L1 deviceVO:
   DeviceVO{device=null, building=UIInventoryObject [label=DAL-LAB-LOC-
   01,
   main=com.blueplanet.inventory.rest.model.common.BaseIdentifiableObje
   ctWithTimestampImpl@e0a2524[objectTimestamp=1668436149563,objectId=2
   58817076074156609,objectClassId=101], today=null, status=null,
   toolTip=null, icon=null], floor=null, room=null,
   customerPremise=null, rackManagementVO=null, deviceName='null',
   deviceType=UIEnumeration [isDefault=null, label=ASR 9001,
   value=1143], ipAddress='', status='Planned', partNumber='ASR9001',
   baseDeviceAttrs=DeviceDefaultAttrsVo{deviceFqdn='null',
   deviceClli='null', buildingClli='null', macAddress='null',
   serialNumber='null', hardwareVersion='null', managementIPAddress='',
   clonedFrom=null, nms=UIInventoryAttribute [value=null, mdif=null],
   deviceDefaultMandatoryFlags=null}, roles=[UIEnumeration
   [isDefault=null, label=Ethernet, value=0]],
   networks=[UIInventoryObject [label=LAB-ETH1,
   main=com.blueplanet.inventory.rest.model.common.BaseIdentifiableObje
```

```
   ctWithTimestampImpl@59432e75[objectTimestamp=<null>,objectId=2557652
   24536044979,objectClassId=115], today=null, status=null,
   toolTip=null, icon=null]], order=UIInventoryObject [label=400000,
   main=com.blueplanet.inventory.rest.model.common.BaseIdentifiableObje
   ctWithTimestampImpl@55ab9d35[objectTimestamp=<null>,objectId=2557651
   55816568224,objectClassId=118], today=null, status=null,
   toolTip=null, icon=null], cascadeNewCLLIToConnections=true,
   deviceToCopy=null, copyAdditionalShelves=false, copyCards=false,
   copyInternalConnections=false, height='3.504 inch', width='17.42
   inch', depth='18.5 inch', vendor='Cisco', assetIdentifier=null,
   mandatoryFlags=DeviceMandatoryFlags [ nameMandatory=true,
   ipAddressMandatory=false, partNumberMandatory=true,
   roleMandatory=true, networkMandatory=true, rackMandatory=false,
   rackPositionMandatory=false, locationMandatory=true,
   orderMandatory=true]}

ERROR: 2022-11-14 14:35:00,461 : http-nio-8080-exec-4 :
   bpi.appserver.extensions.DeviceNaming : L1 inputDeviceName: null

ERROR: 2022-11-14 14:35:00,461 : http-nio-8080-exec-4 :
   bpi.appserver.extensions.DeviceNaming : L1 inputDeviceIpAddress:
```

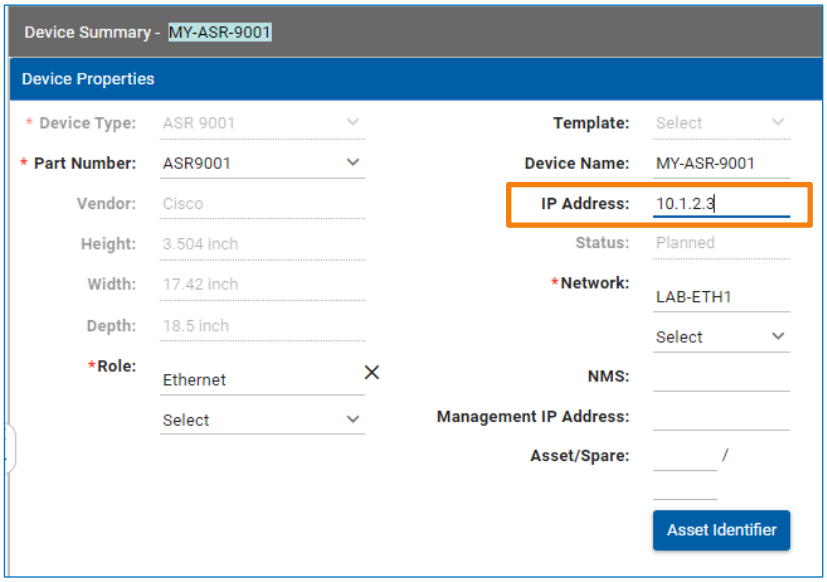
5. Following the coded logic, the device name and the derived name for the device object are set to location name + device type (DAL-LAB-LOC-01_ASR 9001).



6. In the same UI location, now change the device name to **MY-ASR-9001** and click **Apply**. Note how the device name on top of the window changes to MY-ASR-9001 according to the coded logic.



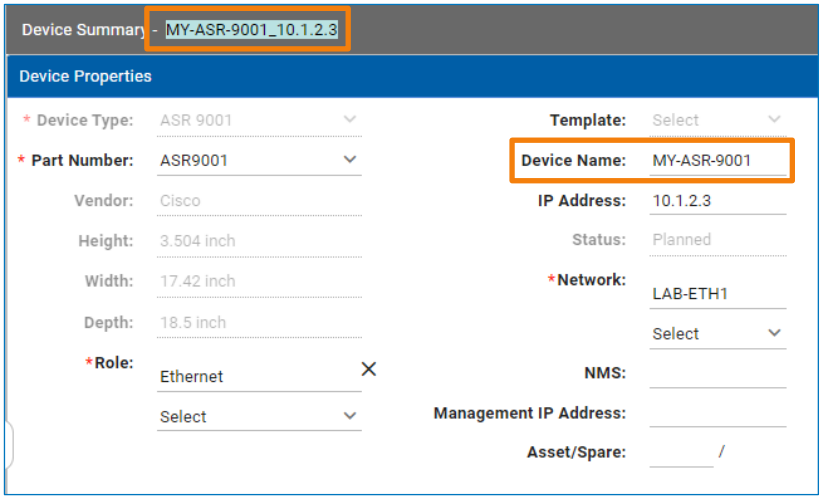
7. As in the last scenario, in the same UI, now add the **IP Address** 10.1.2.3 and click **Apply**.



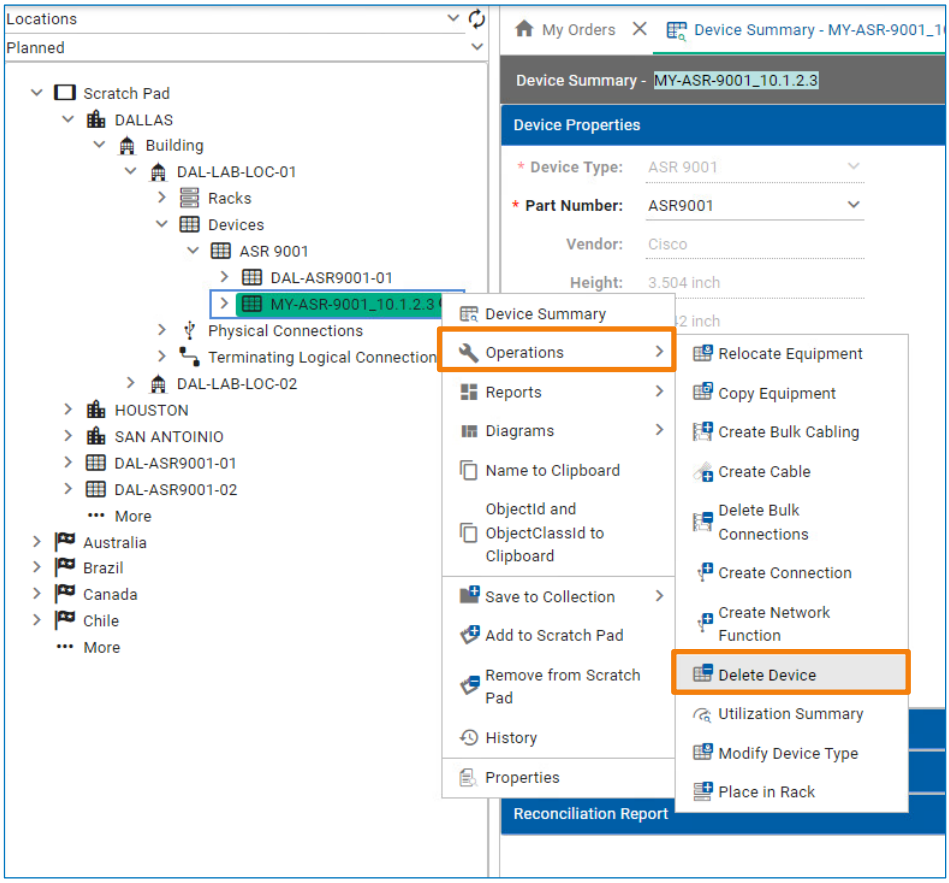
8. In the terminal observe the log printout corresponding to your UI inputs.

```
ERROR: 2022-11-14 14:44:01,251 : http-nio-8080-exec-1 :
   bpi.appserver.extensions.DeviceNaming : L1 inputDeviceName: MY-ASR-
   9001

ERROR: 2022-11-14 14:44:01,251 : http-nio-8080-exec-1 :
   bpi.appserver.extensions.DeviceNaming : L1 inputDeviceIpAddress:
   10.1.2.3
```

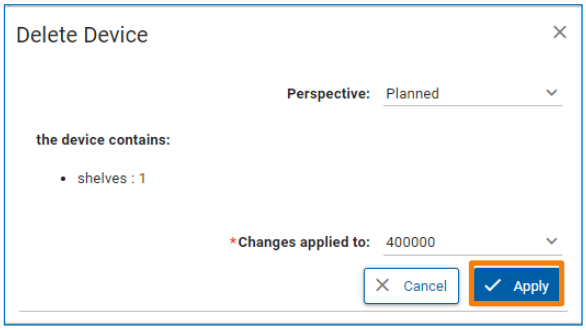
9. In the UI, verify that since you entered both the Device Name and IP Address, the device object is named **MY-ASR-9001_10.1.2.3**, while the Device Name field remains **MY-ASR-9001**.



10. Find the device **MY-ASR-9001_10.1.2.3** in your scratch pad and delete it. Right-click **MY-ASR-9001_10.1.2.3** and choose **Operations > Delete Device**.



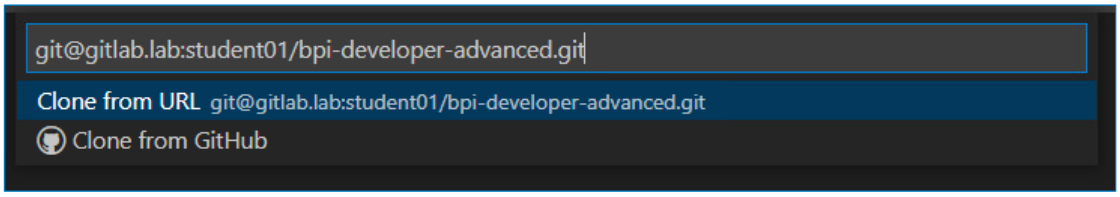
11. Confirm the deletion by clicking **Apply**.



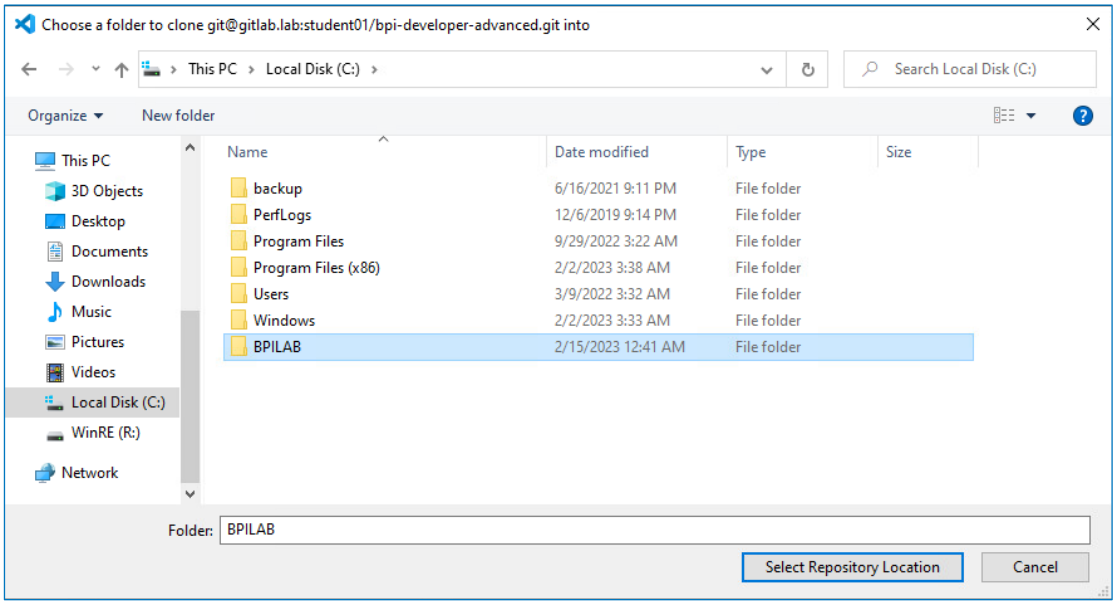
Task 4: Custartifacts Customization – Scenario 1 Device Naming

In this task, you will code a functional customization of naming devices in BPI, when creating new or modifying existing devices. This scenario will utilize the BPI custartifacts loader, which in cooperation with a continuous integration (CI) Gitlab setup allows engineers to build their own custom solutions, and deploy them to live servers. In this exercise you will create a new Java class that extends the existing class for the device naming behavior, then push the code to the remote repository on Gitlab, from where the CI process will prepare, build, test, and deploy the newly built web apps to the server in your lab pod.

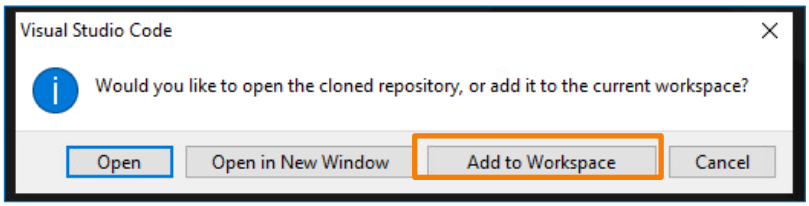
1. Open VSC from your student PC desktop. From the main menu, choose the **File > Open Workspace from File…** option and then select and open the workspace by the name **lab1-part2.code-workspace**.
2. The empty workspace opens. From VSC Explorer, click the **Clone Repository** button. Enter the following repository URL: **git@gitlab.lab:student**XX**/bpi133-bpi-customization-points.git** where XX is the number of your pod, and press Enter.



3. When asked for the destination folder, select c:\BPILAB and press **Select Repository Location**.

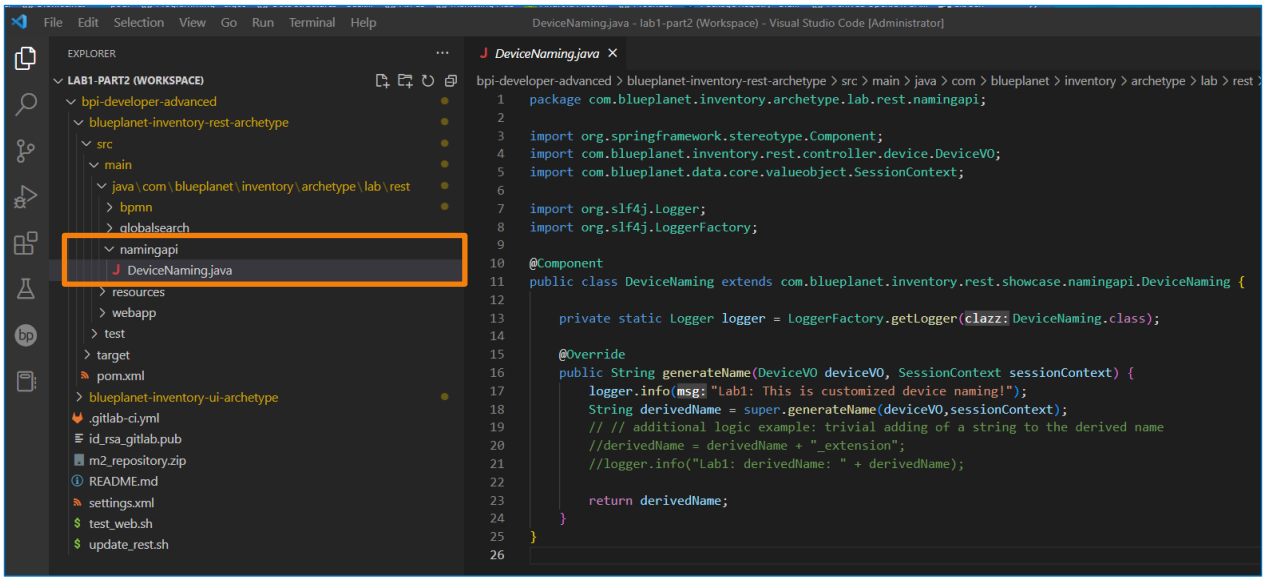


4. When asked whether to add the repository to the workspace, select **Add to Workspace**.



NOTE: If asked whether you trust the author of the files, click **Yes**.

5. Navigate to the source directory **bpi133-bpi-customization-points\blueplanet-inventory-rest-archetype\src\main\java\com\blueplanet\inventory\lab\rest.** In this directory you will find the directory named **namingapi** and inside a Java class file **DeviceNaming.java**, both of which have been configured and populated for this exercise. Click **DeviceNaming.java** to open the file in the right pane.



6. Take a minute to examine the code of the DeviceNaming class. The class extends the existing class **com.blueplanet.inventory.rest.showcase.namingapi.DeviceNaming**, which is the default implementation that comes with the product. The interface implementation requires that the generateName method be present in the class. In this exercise, you will first execute the parent class generateName method, as denoted by **super.generateName()** in line 18, and then you will add additional logic to the method below line 19. You will use the generated **derivedName** from the parent class method to expand on it. In line 20, add the string "_extension" to the generated name and return it at line 23. Now uncomment the lines 20-21.

DeviceNaming.java M

src > main > java > com > blueplanet > inventory > archetype > lab > rest > namingapi > DeviceNaming.java > DeviceNaming > generateN

```java
package com.blueplanet.inventory.archetype.lab.rest.namingapi;

import org.springframework.stereotype.Component;
import com.blueplanet.inventory.rest.controller.device.DeviceVO;
import com.blueplanet.data.core.valueobject.SessionContext;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

@Component
public class DeviceNaming extends com.blueplanet.inventory.rest.showcase.namingapi.DeviceNaming {

    private static Logger logger = LoggerFactory.getLogger(clazz: DeviceNaming.class);

    @Override
    public String generateName(DeviceVO deviceVO, SessionContext sessionContext) {
        logger.info(msg: "Lab1: This is customized device naming!");
        String derivedName = super.generateName(deviceVO, sessionContext);
        // additional logic example: trivial adding of a string to the derived name
        derivedName = derivedName + "_extension";
        logger.info("Lab1: derivedName: " + derivedName);

        return derivedName;
    }
}

```

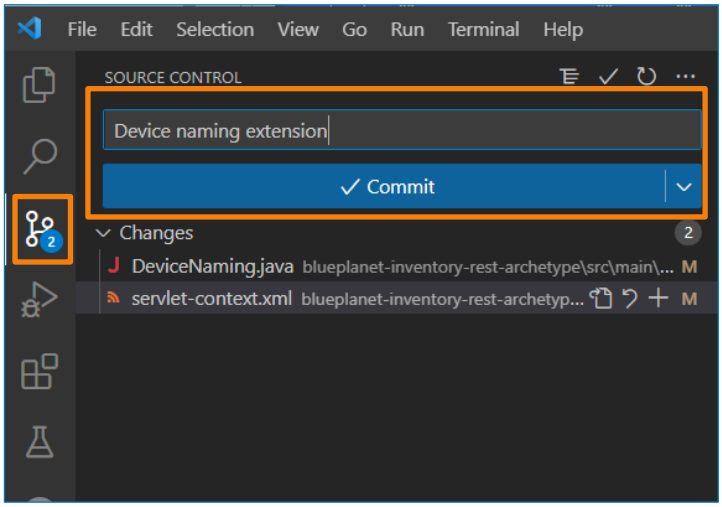
7. Save the file by pressing **CTRL+S**.
8. You extended the DeviceNaming class with the same, conflicting component name deviceNaming. As this causes a conflict, you have to exclude the original class from the Spring component scan. Open the file **servlet-context.xml** located in the **bpi-developer-advanced\blueplanet-inventory-rest-archetype\src\main\webapp\WEB-INF\spring\appServlet** directory.

```xml
<!--  This is the core donriver rest api -->

    <context:component-scan base-package="com.blueplanet.rest" />
    <context:component-scan base-package="com.blueplanet.inventory.sdwanias" />
    <context:component-scan base-package="com.blueplanet.bpmn" />
    <context:component-scan base-package="com.blueplanet.inventory.bpmn" />
    <context:component-scan base-package="com.blueplanet.inventory.metrics" />
    <context:component-scan base-package="com.blueplanet.inventory.archetype" />
    <!--
    <beans:bean id="deviceNaming" class="com.blueplanet.inventory.archetype.lab.rest.namingapi.DeviceNaming" />
    -->
    <!--
    <beans:bean id="connectionNaming" class="com.blueplanet.inventory.archetype.lab.rest.namingapi.ConnectionNaming" />
    -->
    <!-- BPI specific bits
The reason for this seperation is that we want to be able to include JUST the bpi bits in a labelText context.
We cannot include This file because it loads up the core.
Therefore all BPI specific bits have been put in a different file. -->
  <beans:import resource="classpath:/META-INF/spring/servlet-context-bpi.xml"/>

</beans:beans>

```

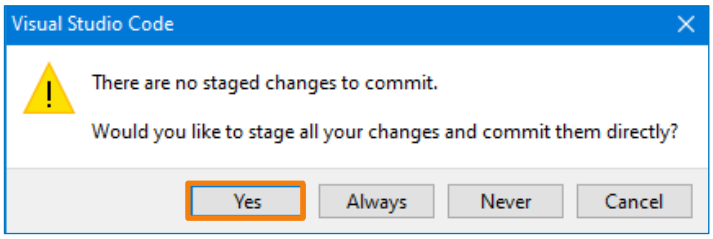
9. Uncomment the **<beans:bean** in line 23 by deleting comment tags in lines 22 and 24:

```
<!--  This is the core donriver rest api -->

    <context:component-scan base-package="com.blueplanet.rest" />
    <context:component-scan base-package="com.blueplanet.inventory.sdwanias" />
    <context:component-scan base-package="com.blueplanet.bpmn" />
    <context:component-scan base-package="com.blueplanet.inventory.bpmn" />
    <context:component-scan base-package="com.blueplanet.inventory.metrics" />
    <context:component-scan base-package="com.blueplanet.inventory.archetype" />

    <beans:bean id="deviceNaming" class="com.blueplanet.inventory.archetype.lab.rest.namingapi.DeviceNaming" />

    <!--
    <beans:bean id="connectionNaming" class="com.blueplanet.inventory.archetype.lab.rest.namingapi.ConnectionNaming" />
    -->
    <!-- BPI specific bits
The reason for this seperation is that we want to be able to include JUST the bpi bits in a labelText context.
We cannot include This file because it loads up the core.
Therefore all BPI specific bits have been put in a different file. -->
  <beans:import resource="classpath:/META-INF/spring/servlet-context-bpi.xml"/>

</beans:beans>
```

10. Press **CTRL+S** to save the file.
11. In VSC, click the **Source Control** icon in the left menu. Under Message, enter the commit message **Device naming extension** and then click **Commit**.

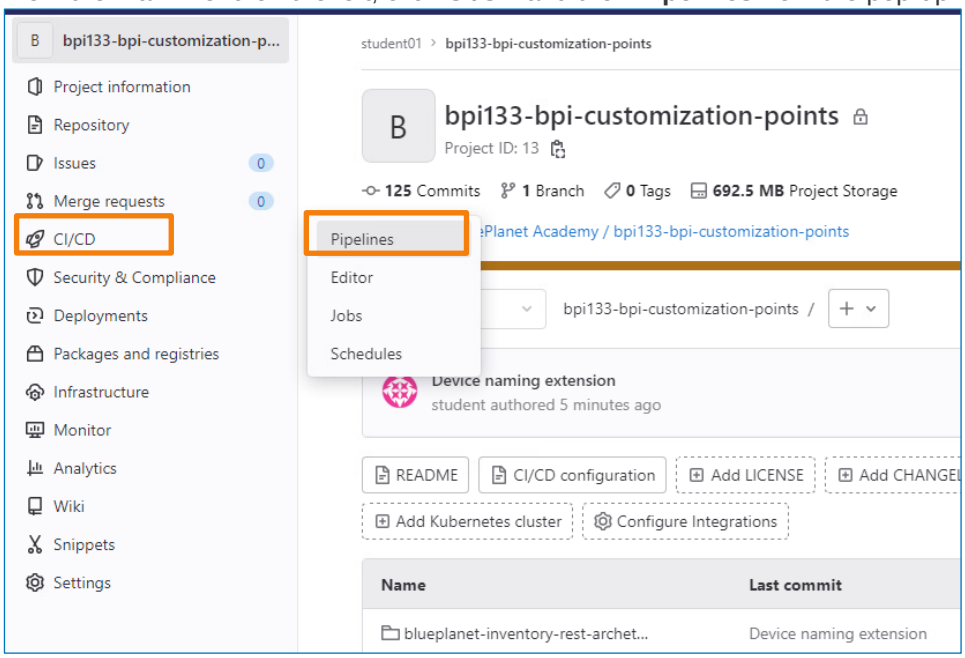


12. Click **Yes** to stage changes.

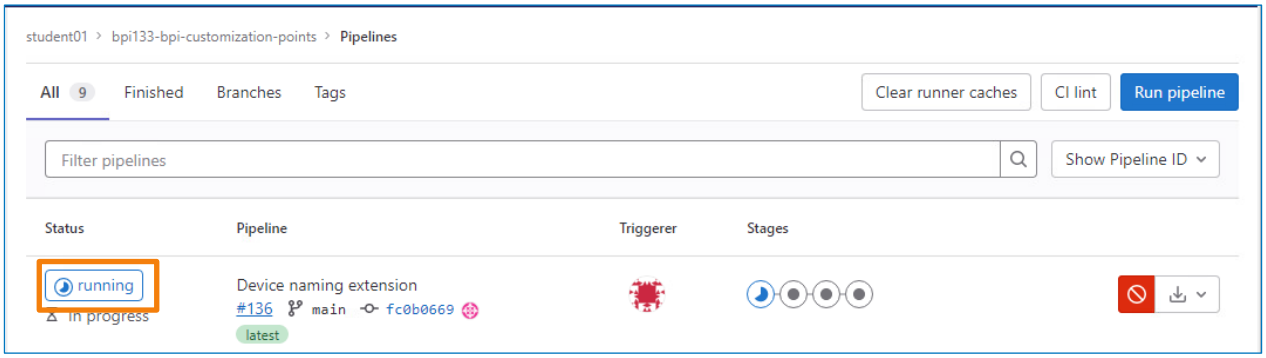


13. Click **Sync Changes** to push the changes to the git repository. This action will trigger your preconfigured CI pipeline which will take your code modifications, build the new REST application, and deploy it to the server.
14. Open a new web browser and log in to your Gitlab instance. Enter the following URL in your browser: **http://gitlab.lab**. Use the credentials **studentXX/12345678**, where XX is your pod number.

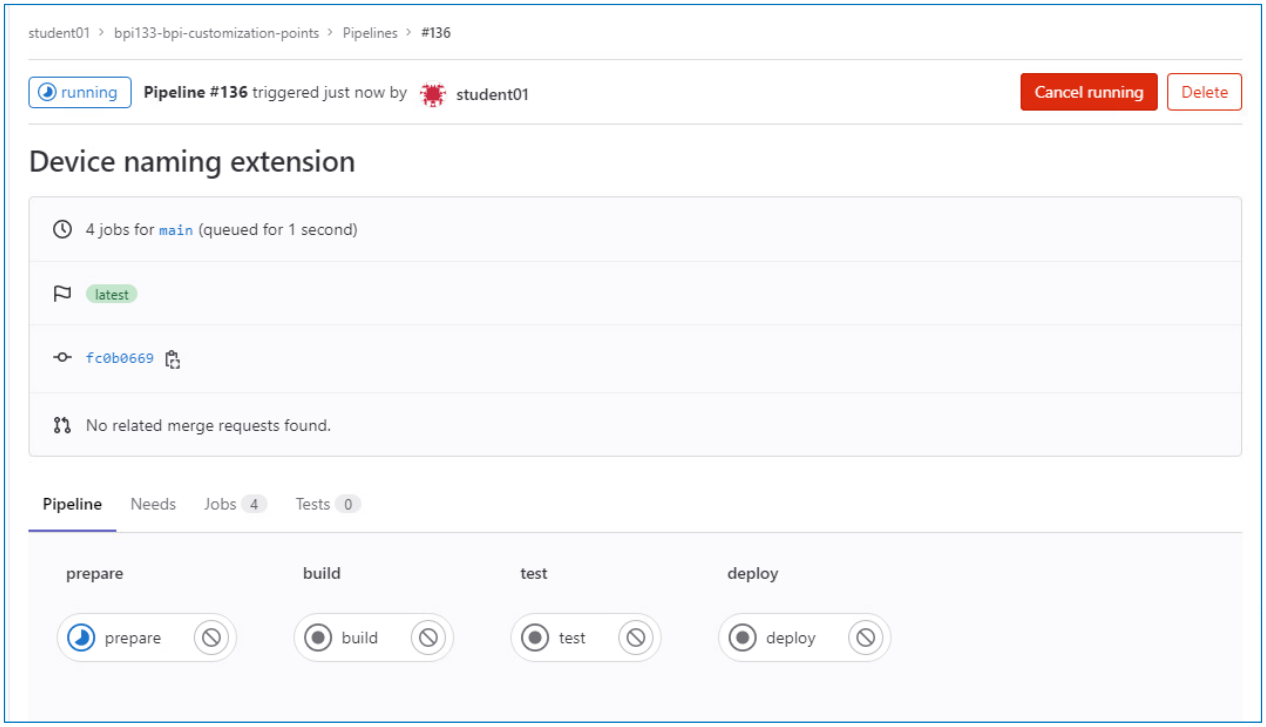
15. From the main menu on the left, click **CI/CD** and then **Pipelines** from the pop-up menu.



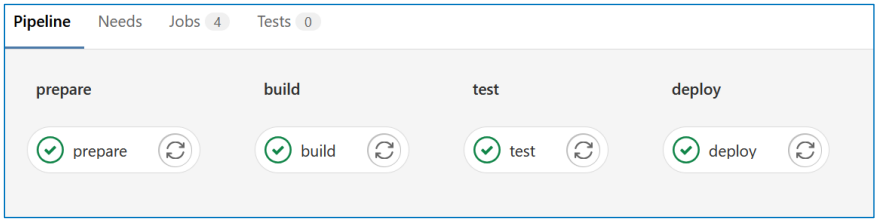
16. Your pipeline process appears at the top of the list. Click **running** to see the details. The Status will change from **running** to **passed** or **failed**. If the job fails, check your project for errors.



17. The details of your pipeline job will open.



18. The pipeline process is completed when the four green checks are displayed, one for each task.



NOTE: Optionally you can connect to the custartifacts-0 container, **using kubectl exec -it custartifacts-0 -- bash**, and examine the **/bp2/log/process_log/custartifacts/cust_process_log.log** file to get a detailed output of the application deployment processes.

NOTE: It might take up to 10 minutes for the containers to fully initialize after the CI/CD deployment has finished.

19. Open a new terminal window using Putty and connect to your BPI server. From within web-0, run a tail command to monitor the catalina.out log.

```
[bpadmin@bpi-pod03 ~]$ kubectl exec -it web-0 -- bash
root@web-0:/dev/shm# tail -f /bp2/log/catalina.out | grep Lab1
```

20. Before you continue, since there is now a custom application applied, you will have to clear the cookies (Settings > Privacy and security > Clear browsing data, or Ctrl+Shift+Delete) in your Chrome browser.
21. From BPI UI in your browser, create a new device. In your scratch pad, in DALLAS, right-click on the building **DAL-LAB-LOC-01** and select **Operations > Create Equipment**. Use the following parameters and click **Apply**:

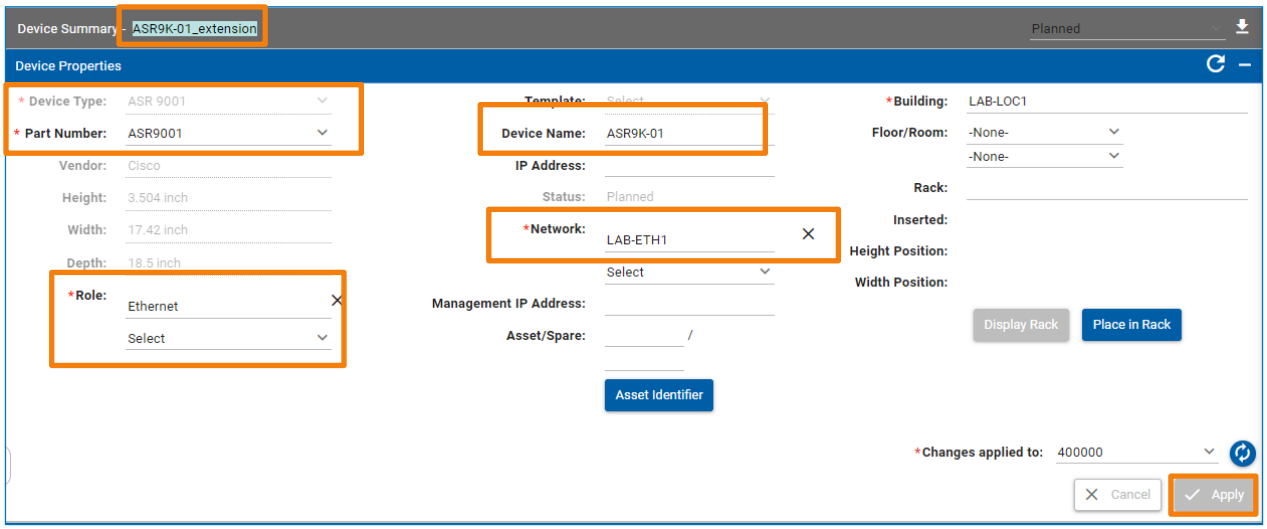
    Device Type: **ASR 9001**

Part Number: **ASR 9001**

Device Name: **ASR9K-01**

Role: **Ethernet**

Network: **LAB-ETH1**



NOTE: As soon as you press Apply, observe how the name in the Device Summary changed to ASR9K-01_extension as defined in your extension code.

22. Create another device, this time in the other building, DAL-LAB-LOC-02, with the following parameters. When done, click **Apply**.

    Device Type: **ASR 9001**

    Part Number: **ASR 9001**

    Device Name: **ASR9K-02**

    Role: **Ethernet**

    Network: **LAB-ETH1**

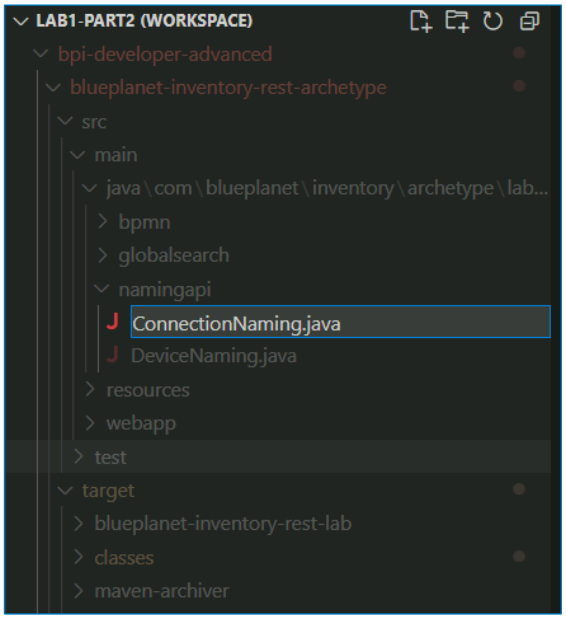
23. Return to the terminal window and observe the output. Your customization was triggered and the log messages for the two created devices are displayed as coded in the extension.

```
INFO : 2022-11-17 21:20:07,390 : http-nio-8080-exec-3 :
   com.blueplanet.inventory.archetype.lab.rest.namingapi.DeviceNaming :
   Lab1: This is customized device naming!

INFO : 2022-11-17 21:20:07,391 : http-nio-8080-exec-3 :
   com.blueplanet.inventory.archetype.lab.rest.namingapi.DeviceNaming :
   Lab1: derivedName: ASR9K-01_extension

INFO : 2022-11-17 21:27:59,670 : http-nio-8080-exec-1 :
   com.blueplanet.inventory.archetype.lab.rest.namingapi.DeviceNaming :
   Lab1: This is customized device naming!

INFO : 2022-11-17 21:27:59,671 : http-nio-8080-exec-1 :
   com.blueplanet.inventory.archetype.lab.rest.namingapi.DeviceNaming :
   Lab1: derivedName: ASR9K-02_extension
```

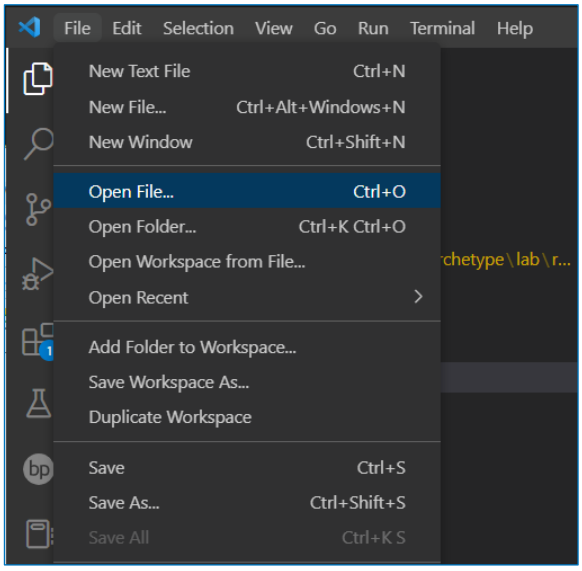
Task 5: Custartifacts Customization – Scenario 2 Connection Naming

In this task, you implement a trivial customization example at the point of connection naming. The point of this exercise is to demonstrate which class to modify to customize the behavior of generating connection names when creating new or modifying existing connections in BPI.

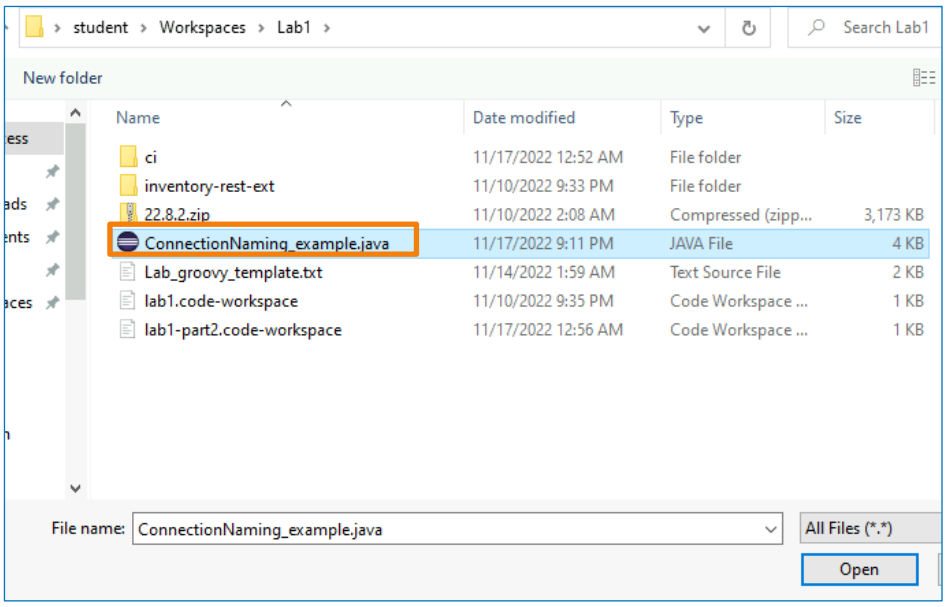
1. Create a new java file, named **ConnectionNaming.java** in the **namingapi** directory, at the same level as the existing DeviceNaming.java file. To complete this step, right-click on **namingapi** and choose **New File…** from the menu, then enter the file name.



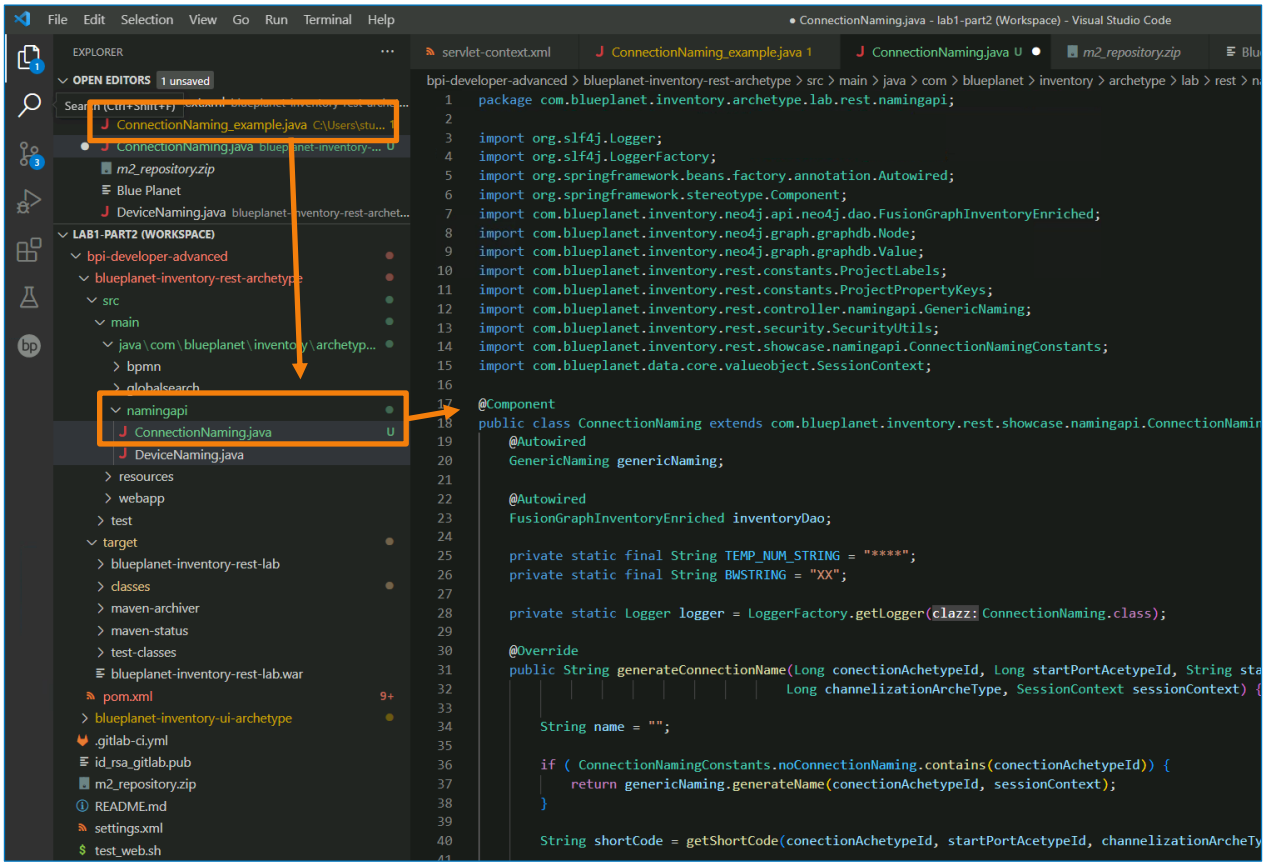
2. Now from the main VSC menu, open a new file, by choosing **File > Open File...** to open the prepared example java file that you will use in this exercise.



3. Choose the file **ConnectionNaming_example.java** from your **Lab1** folder in Workspaces and click **Open**.



4. Next, select all text from **ConnectionNaming_example.java**, right-click on the text, choose **Copy,** and then paste it into the file **namingapi\ConnectionNaming.java**.



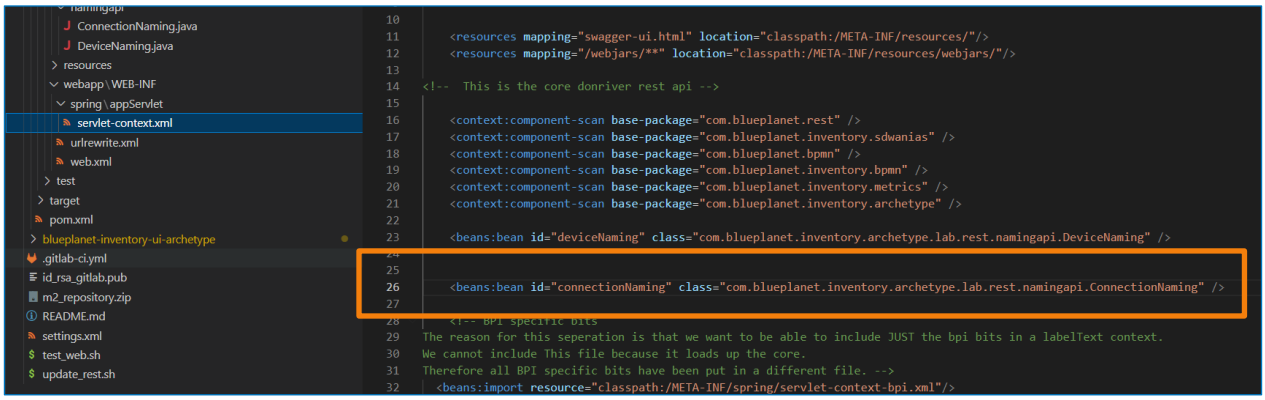
NOTE: If there was text in namingapi\ConnectionNaming.java before copying, delete that extra text before proceeding. Be sure to verify that the files now have identical lines of code.

5. Take a minute to examine the code of **ConnectionNaming.java**. In this scenario, you are extending the **com.blueplanet.inventory.rest.showcase.namingapi.ConnectionNaming** class which implements the **ConnectionNamingInterface.** This interface requires the implementation of two methods: **generateName** and **generateNameForCopiedOrMovedConnection** as seen from the source code:

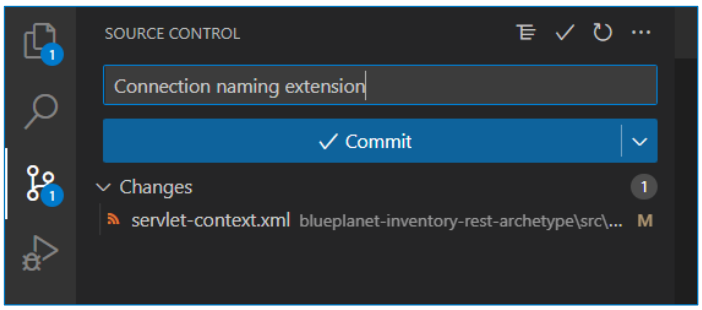
```
public interface ConnectionNamingInterface {

   ConnectionVO generateName(Node var1, ConnectionVO var2,
    SessionContext var3);


   String generateNameForCopiedOrMovedConnection(String var1, String
    var2, String var3);

}
```

   In the example of this exercise, you leave those two methods as they are implemented and instead just override another public method from the parent class called **generateConnectionName**. The overridden method contains log message statements in code at a specific place, to output the generated name and another comment. This is just an example; the modification possibilities are endless. In your production scenario, you will likely dive deeper into the customization and change other methods, and/or create new ones. You can find the product default implementation of ConnectionNaming.java in the rest-ext library for the code comparison. Rest-ext source code is available in this exercise in your workspace folder.

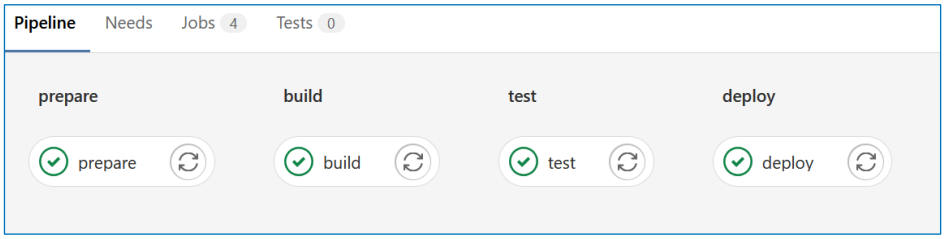
6. Open the file **servlet-context.xml** located in the **bpi-developer-advanced\blueplanet-inventory-rest-archetype\src\main\webapp\WEB-INF\spring\appServlet** directory and delete the comment tags in lines 25 and 27. **Save** the file.



7. Now you are ready to deploy your changes to the server. Enter **Connection naming extension** for the commit message and click **Commit**.



8. Click **Sync Changes** to push the changes to the git repository. This action will trigger your preconfigured CI pipeline one more time.
9. Monitor the pipeline in your Gitlab. The pipeline is finished when you observe 4 green checks.



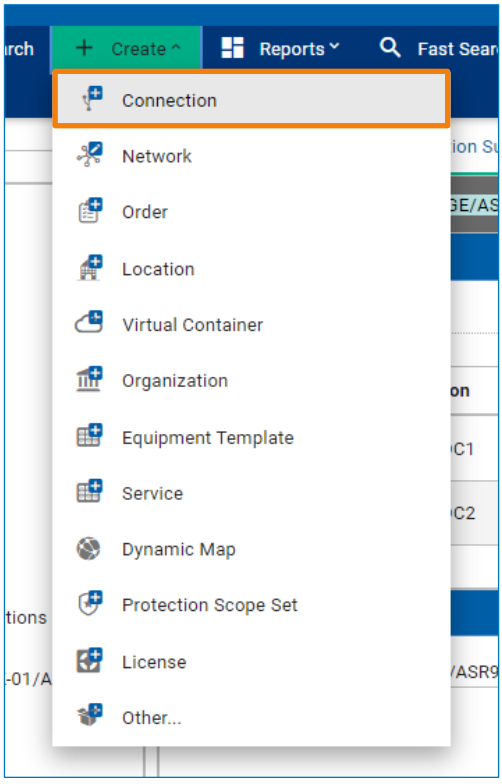
NOTE: In case of failure, click on the offending task. Typically, if you have syntax or other errors in your code, the build task fails. If you click it, you are redirected to a detailed page where you can see the location of the error.

After the pipeline successfully completes, it takes a short time, around 5 minutes, for your UI to become responsive. Optionally, you can use Nagios to verify when the server is ready.

10. Open the terminal window and set up catalina.out log monitoring as follows:

```
[bpadmin@bpi-pod03 ~]$ kubectl exec -it web-0 -- bash
root@web-0:/dev/shm# tail -f /bp2/log/catalina.out | grep Lab1
```

11. From BPI UI, choose **Create > Connection** from the main menu.



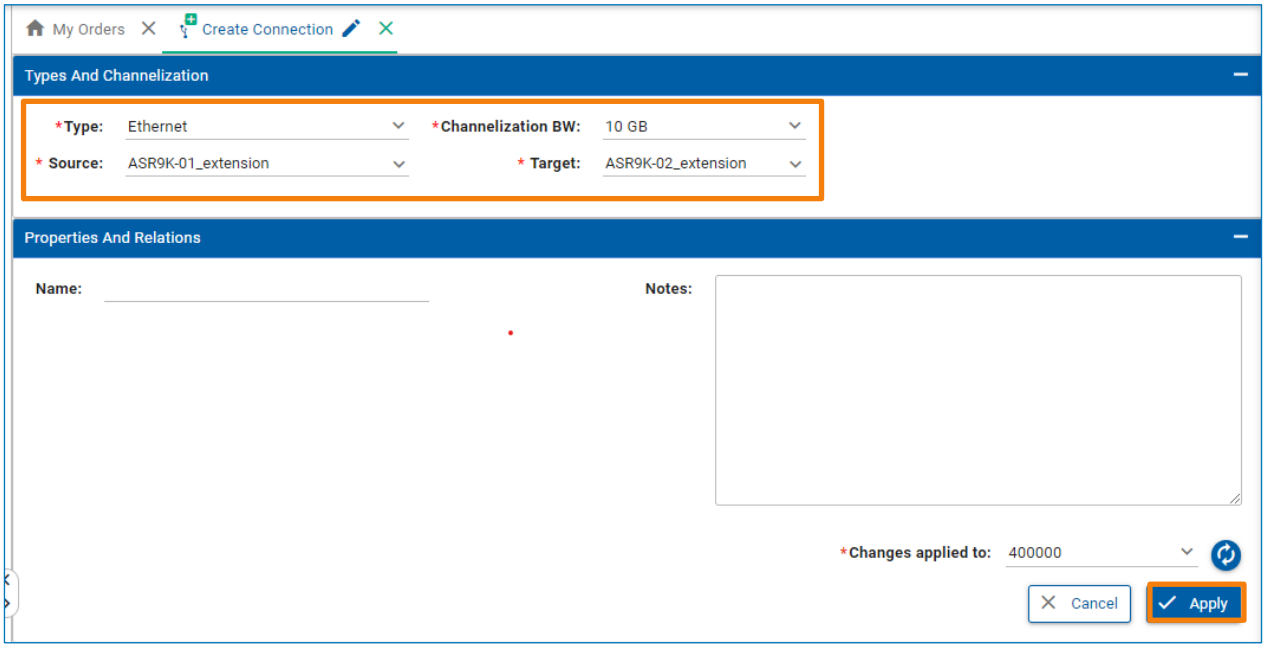
12. Enter the following parameters for the connection. Leave the **Name** field empty and click **Apply** when ready:

    Type: **Ethernet**

    Channelization BW: **10 GB**

    Source: **ASR9K-01_extension**

    Device Name: **ASR9K-02_extension**



13. In the terminal window verify that the messages were logged, proving that your custom class was triggered, and not the default implementation.

```
INFO : 2022-11-18 10:28:38,848 : http-nio-8080-exec-8 :
   com.blueplanet.inventory.archetype.lab.rest.namingapi.ConnectionNami
   ng : Lab1: Current connection name: ****/10GE/ASR9K-01/ASR9K-02
INFO : 2022-11-18 10:28:38,848 : http-nio-8080-exec-8 :
   com.blueplanet.inventory.archetype.lab.rest.namingapi.ConnectionNami
   ng : Lab1: Modify connection naming in this class
```

14. From your server terminal session, break the logging session, exit from the web-0 container.

```
CTRL+C
root@web-0:/dev/shm# exit
```

15. Restore the default web application on your server instance.

```
[bpadmin@bpi-pod20 ~]$ reset_bpi_webapps.sh
pod "custartifacts-0" deleted
pod "asset-manager-0" deleted
pod "web-0" deleted
[bpadmin@bpi-pod20 ~]$
```

NOTE: Make sure you reset the server to the default web application before you proceed to the next lab.

End of Lab

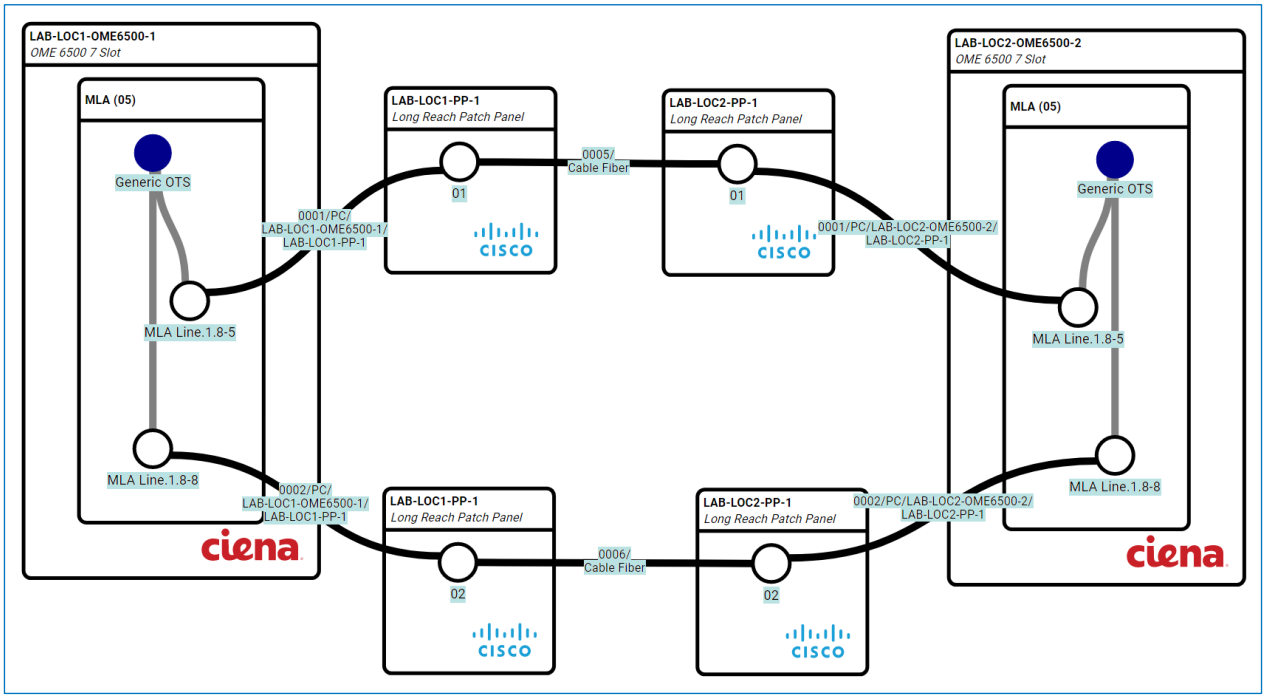
Lab 2: Advanced Graph DB Topics

Objectives

- Describe the Graph DB model for connectivity and logical inventory
- Implement physical and logical connections
- Implement physical ports and logical interfaces
- Implement new compatibilities in the Graph DB
- Construct various Cypher queries, including queries for loading data from CSV files

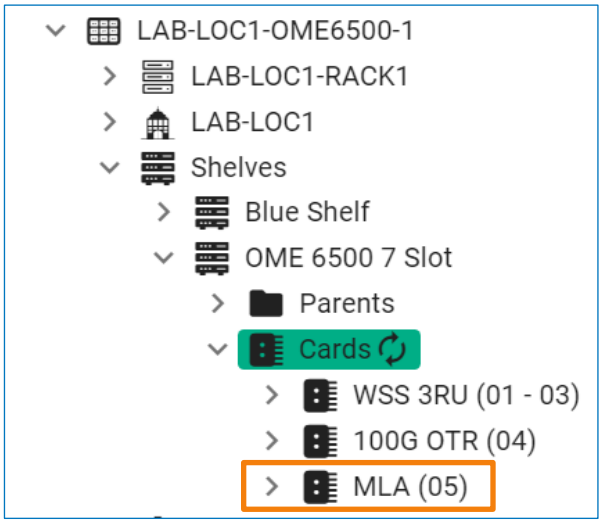
Task 1: Create a Physical Connection

In the first part of this lab, you will create connections in the BPI UI, and explore how that reflects in the Graph database. BPI allows and restricts which connections can terminate on which ports or logical interfaces by providing a compatibility relationship between the Archetype nodes. In this lab, you will explore those relationships, and create new instances of connections and logical interfaces. You will implement an end-to-end optical transmission section (OTS) connection between devices LAB-LOC1-OME6500-1 and LAB-LOC2-OME6500-2. In LAB-LOC2, everything has already been preconfigured, so you will only configure LAB-LOC1. The connections between locations are unidirectional, with one optical fiber for transmitting (Tx) data and one for receiving (Rx) data. The image below shows the final state of the OTS logical connection.



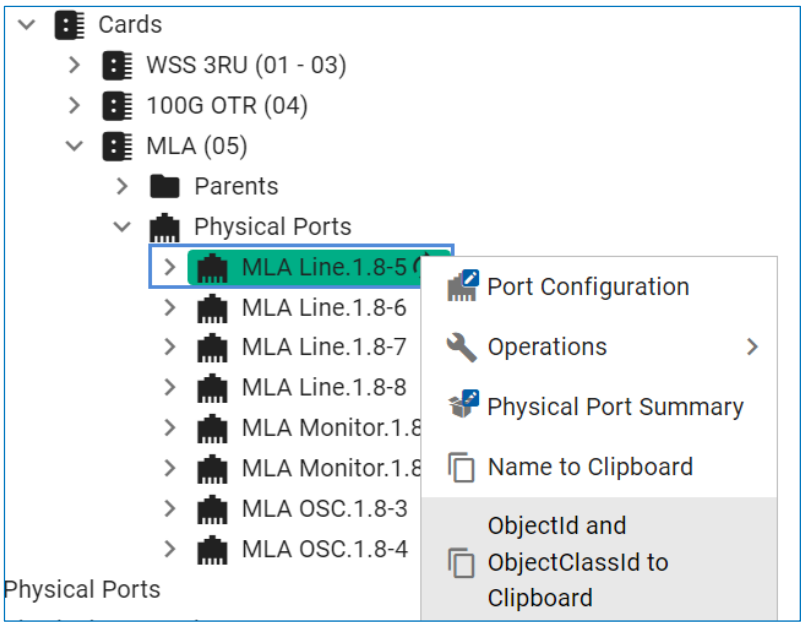
In this task, you will explore the Graph DB to see how compatibilities between physical ports and physical connections are defined. You will create new Patch Cable connections between your optical device and a patch panel, and inspect how that reflects in the Graph DB.

1. First, check the current physical configuration of your device. From your Scratch Pad, navigate to the device **LAB-LOC1-OME6500-1 > Shelves > OME 6500 7 Slot > Cards**, to view the configured cards.



The device is configured with the MLA (Midstage Line Amplifier) card in slot 5. The ports on this card will terminate the physical connections (fiber patch cables) to the patch panel. In the inventory tree, you can also see that there are currently no Physical Connections present under the LAB-LOC1-OME6500-1 device.

2. Identify which physical connections are compatible with the ports on the MLA card. To achieve that, you first need to identify which Archetype is associated with those ports. Collect the drnId of the first Line port on the MLA card. Expand **MLA (05) > Physical Ports**, right-click the **MLA Line.1.8-5** port and choose **ObjectId and ObjectClassId to Clipboard**.



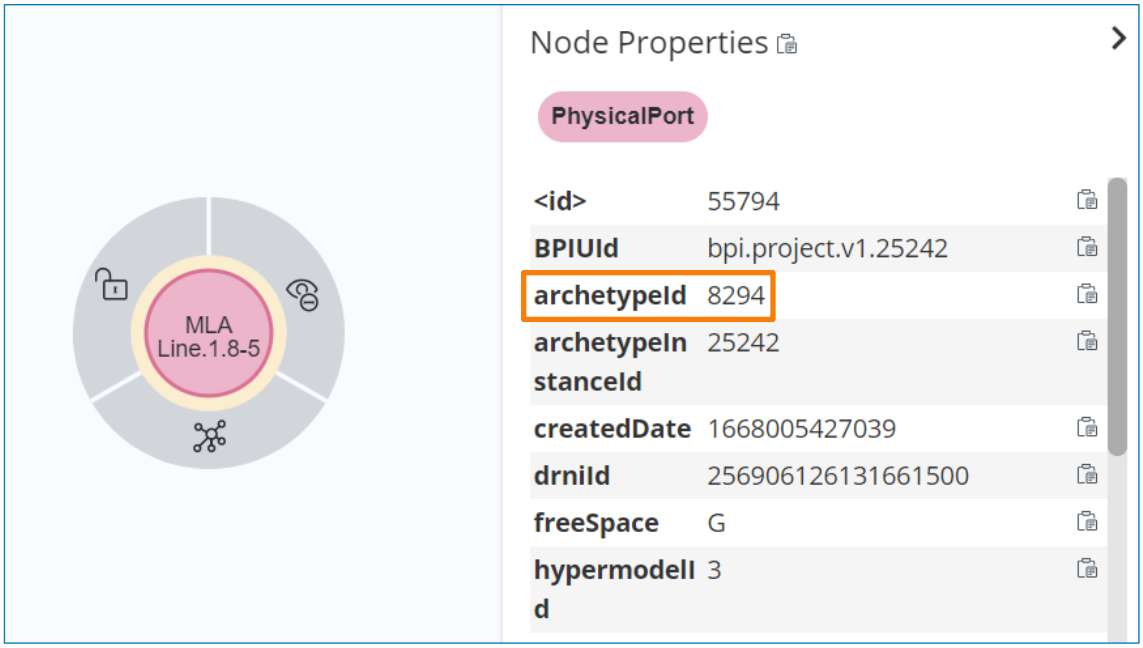
This copies the drnId of that port object and the object class, separated by a colon, to the clipboard.

3. From your Student PC Desktop, open the Neo4j Browser desktop app. Run a query to retrieve the details of the **MLA Line.1.8-5** physical port. Replace the drnId value in the query below with the

drniId you copied to the clipboard in the previous step. Make sure to use just the ObjectId, not the ObjectClassId, for example, 256906126131661500, not 256906126131661500:112.

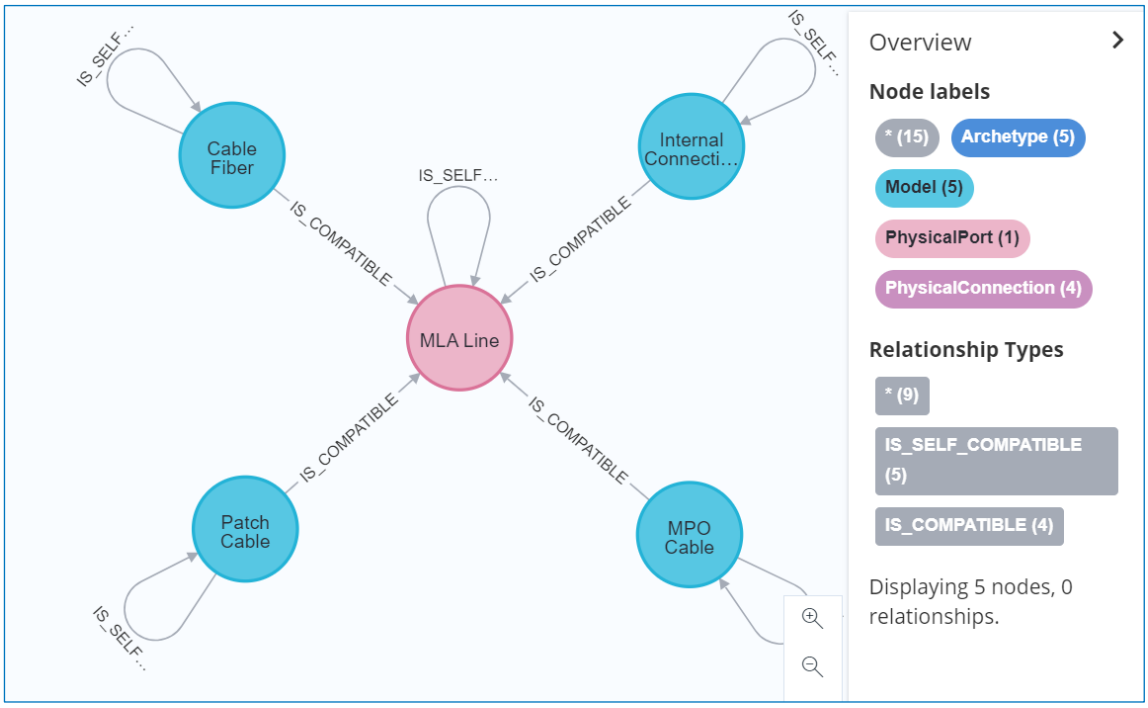
```
match (m{drniId:256906126131661500}) return m
```

4. Click the displayed node to open the Node Properties panel. Notice that this port has the **PhysicalPort** label assigned, and it is modeled from the Archetype with drniId 8294.



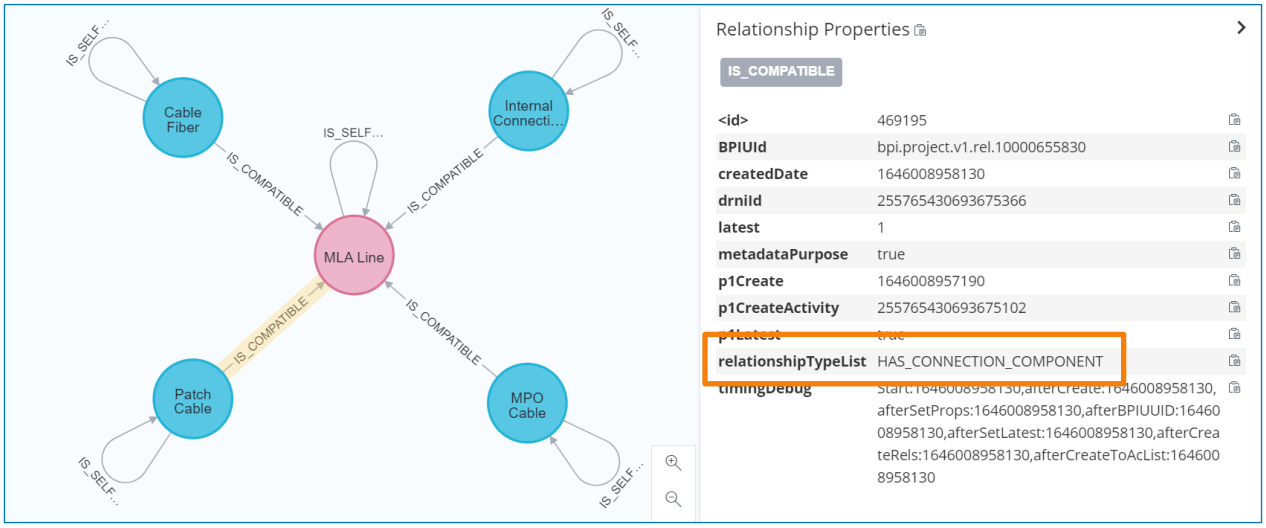
5. Verify the available compatibilities between this port and physical connections by listing the **PhysicalConnection** Archetypes, which have the **IS_COMPATIBLE** relationship to Archetype 8294.

```
match (m:Archetype {drniId:8294})<-[:IS_COMPATIBLE]-(n:PhysicalConnection)
return m,n
```



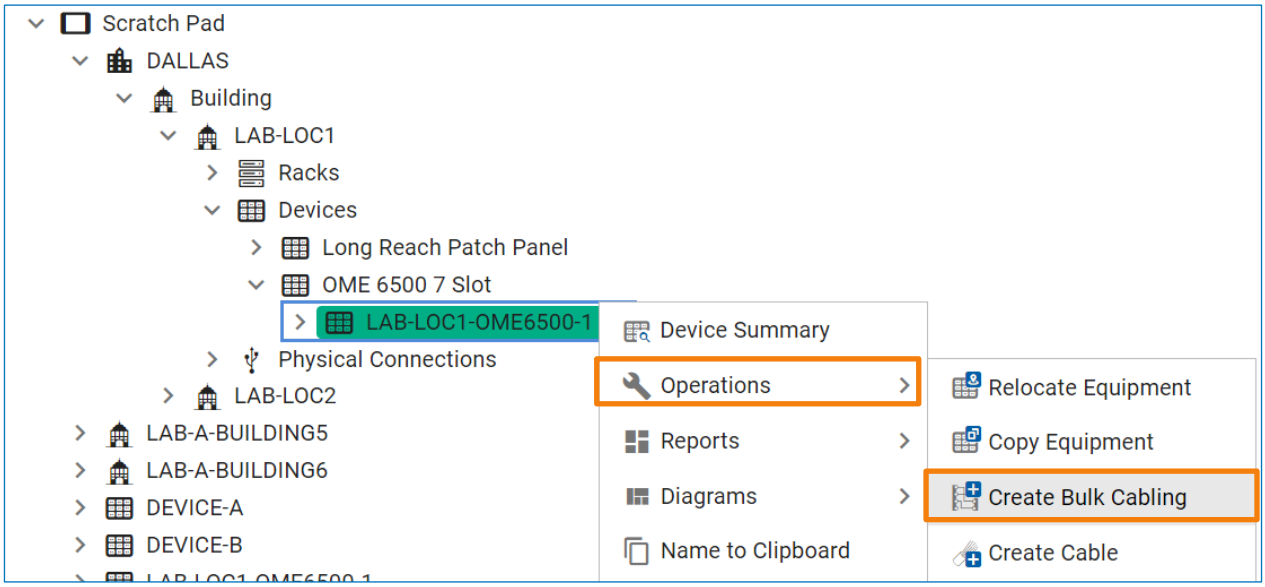
The graph displays the relationships between the **MLA Line** Archetype (drniId 8294) and some physical connection Archetypes. You now identified that the MLA Line ports are compatible with Patch Cable, Cable Fiber, Internal Connection, and MPO Cable physical connection types. This means that you should be able to add a new Patch Cable between the MLA ports on the OME 6500 device and a patch panel in LAB-LOC1.

6. Click the **IS_COMPATIBLE** relationship between the **Patch Cable** and **MLA Line** nodes.

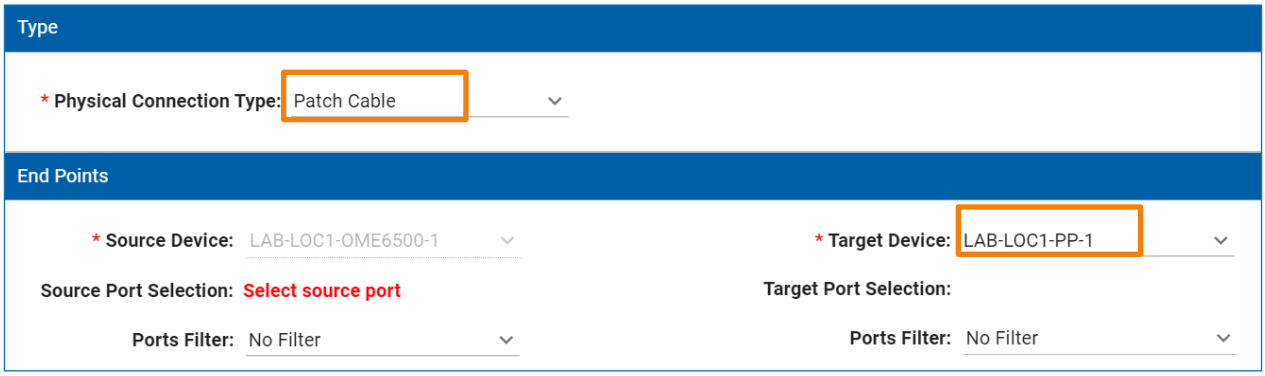


Notice that the **relationshipTypeList** property has a single value, **HAS_CONNECTION_COMPONENT**, which means that this is the only relationship type that can be created between instances derived from the Patch Cable and MLA Line Archetypes.

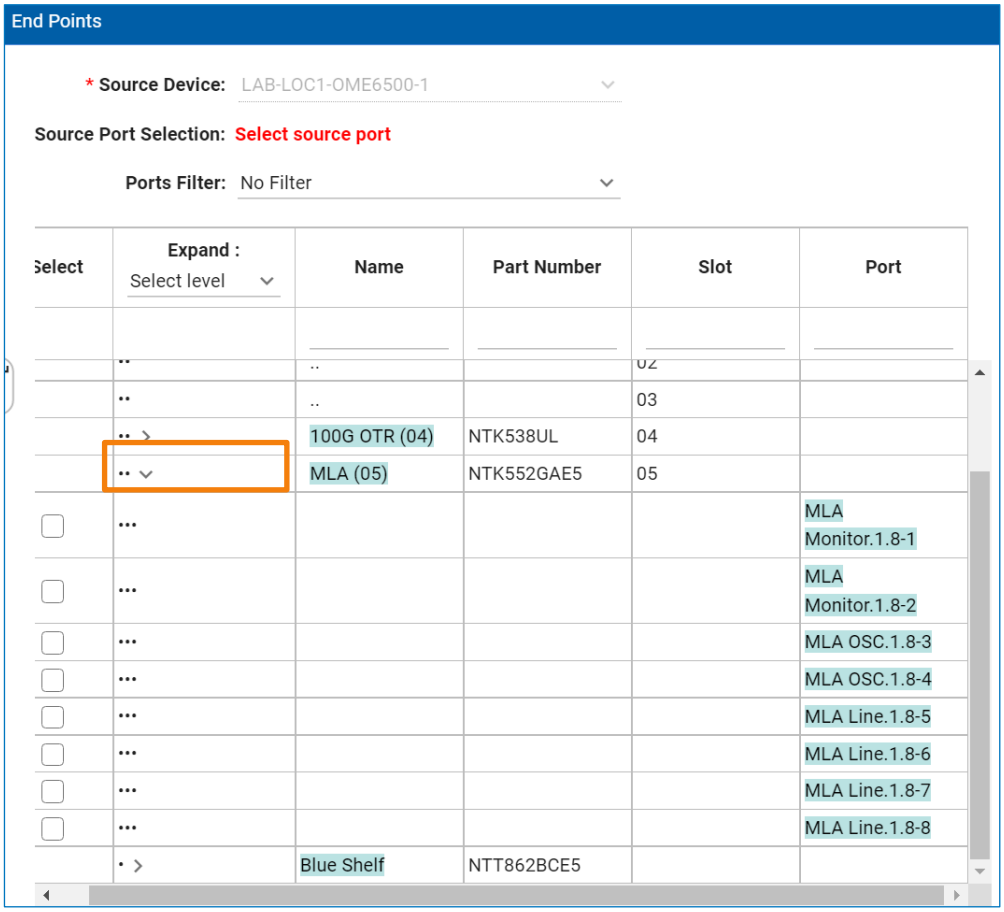
7. You will now create a new connection in the BPI UI. From the Scratch Pad, right-click the **LAB-LOC1-OME6500-1** device, expand **Operations**, and choose **Create Bulk Cabling**.



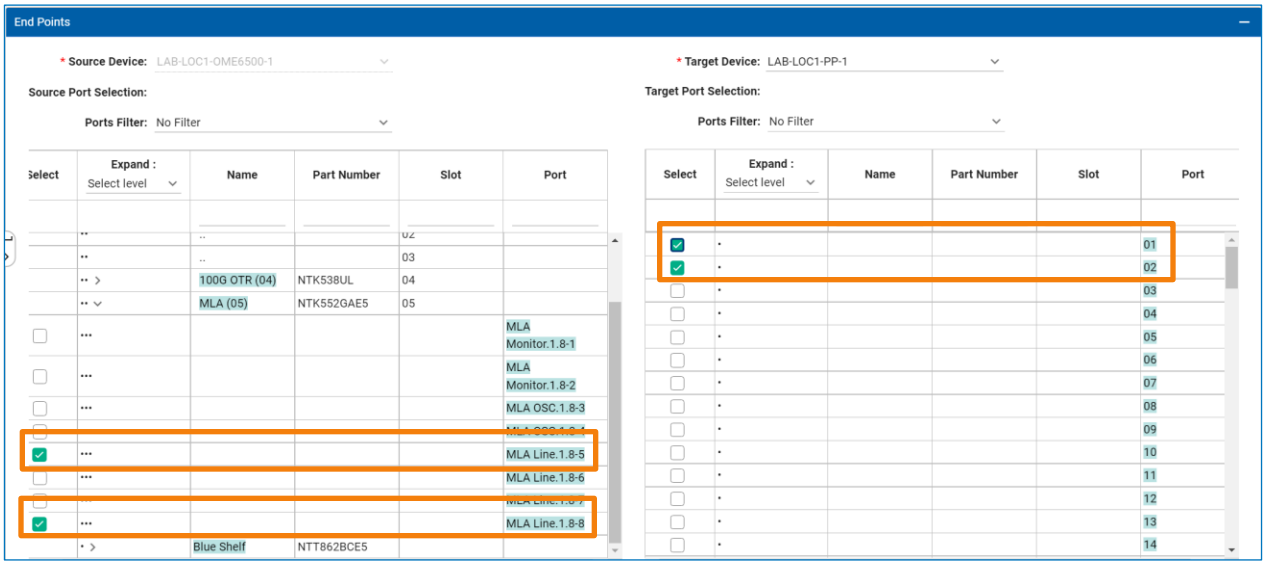
8. The **Source Device** field is already populated. Choose **Patch Cable** as the **Physical Connection Type**, and the patch panel **LAB-LOC1-PP-1** as the **Target Device**. Remember that you can either search for the device name from the dropdown or drag-and-drop the device from the inventory tree.



9. On the source device, expand the **OME 6500 7** slot shelf and the **MLA (05)** card in Slot 05.



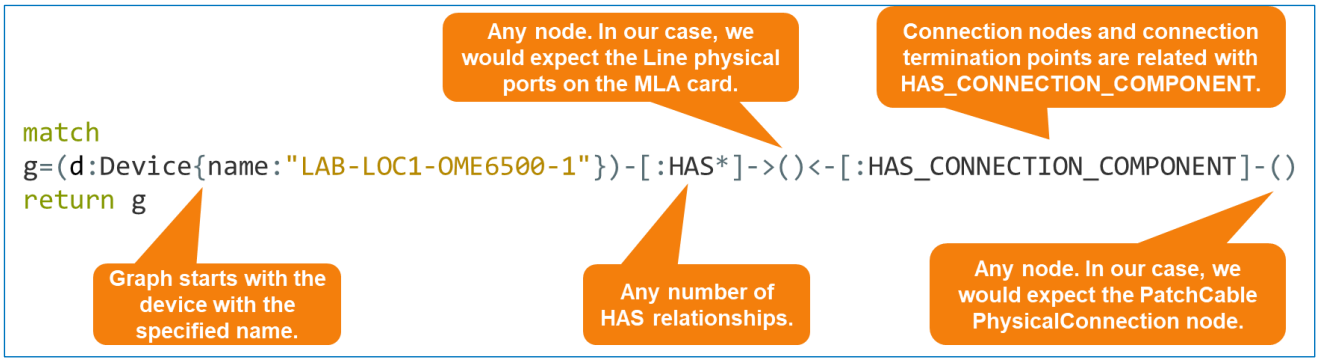
10. On the source device, choose ports **MLA Line.1.8-5** and **MLA Line.1.8-8**. On the target device, choose Port **01** and **02**. Note that depending on your screen resolution, you would need to scroll left or right to view the checkboxes and Port names.



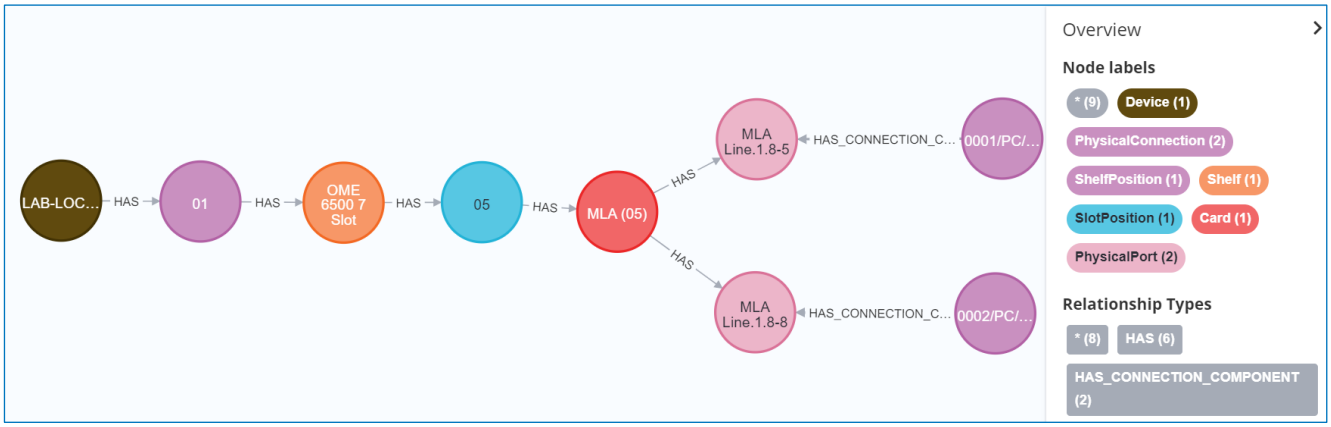
11. Choose your order number in the **Changes applied to** field and click **Apply** to apply the changes.

12. You have now created two patch cable connections between the physical ports on the MLA card on the device LAB-LOC1-OME6500-1 and the patch panel in LAB-LOC1. You will now construct a query in the Neo4j Browser to display the relationships from your device to the patch panel. Remember that physical components are related with the HAS relationship type, and the connection will be related to the port with the HAS_CONNECTION_COMPONENT. Use this information to build the following query:

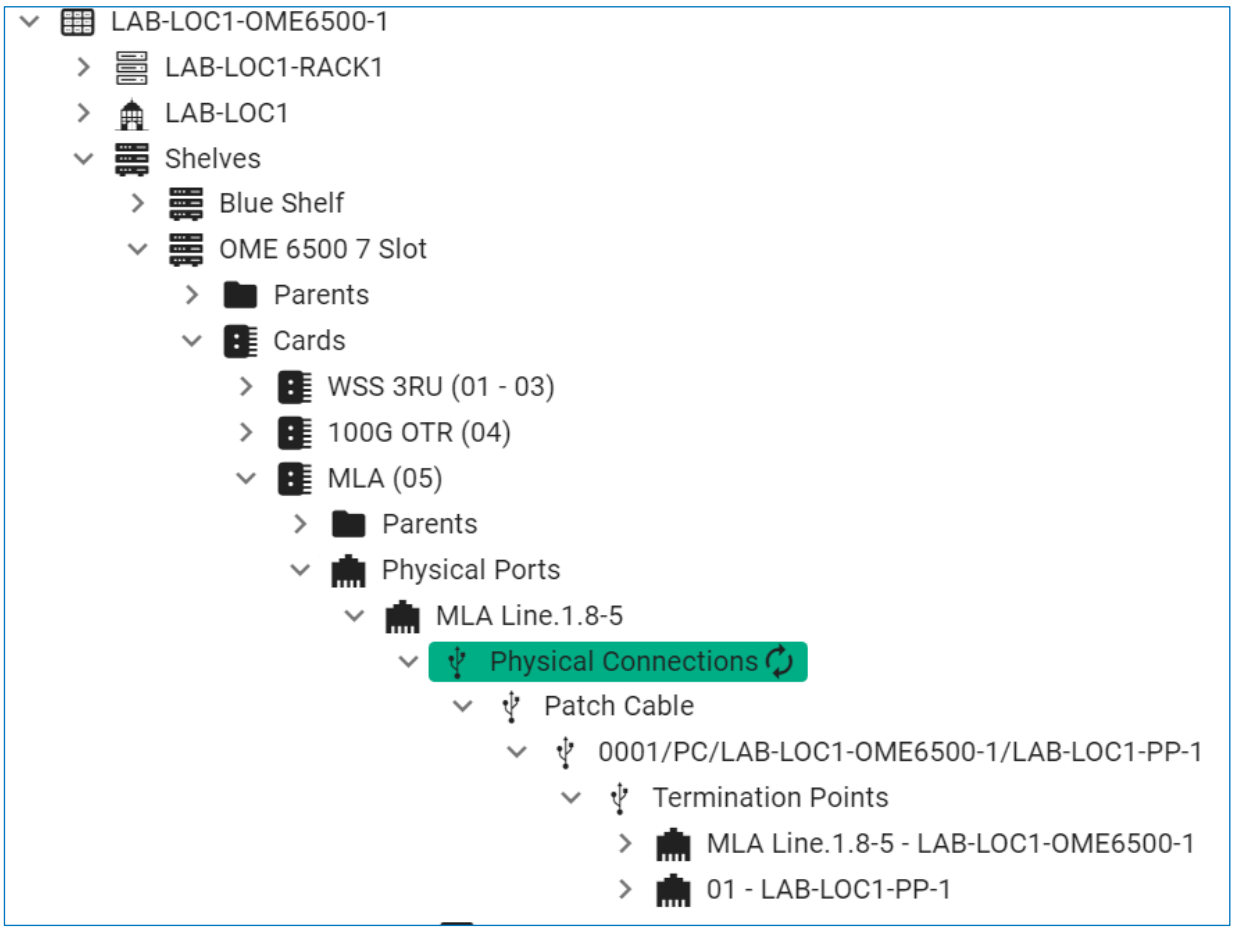
```
match g=(d :Device {name:"LAB-LOC1-OME6500-1"})-[:HAS*]->()<-
   [:HAS_CONNECTION_COMPONENT]-()

return g
```



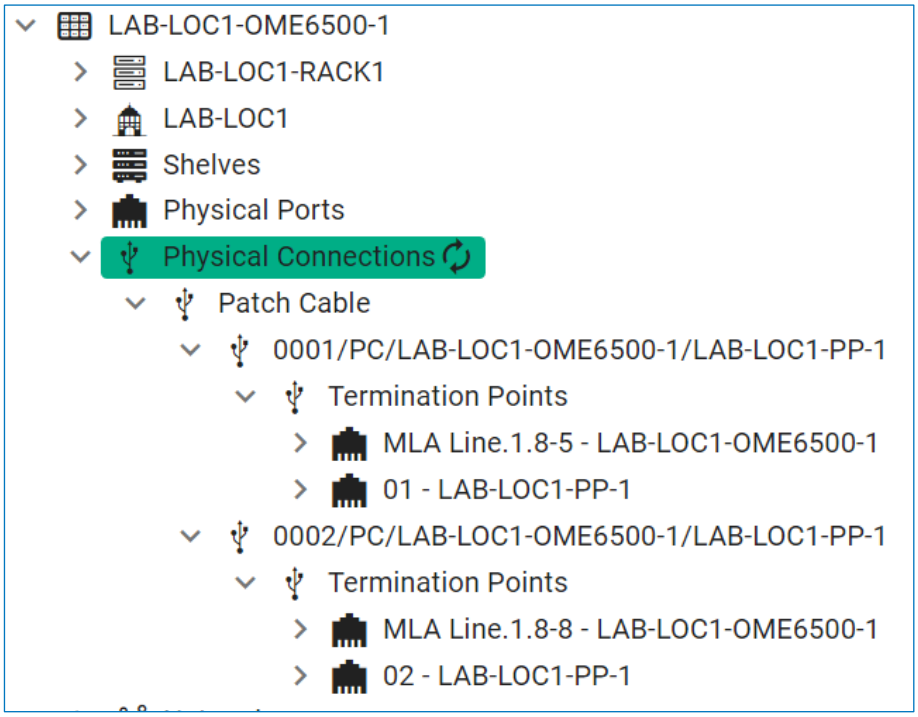
13. Run the query. You should see the relationships from your device to the Patch Cable connections you created. Observe the displayed node labels and the relationship types.



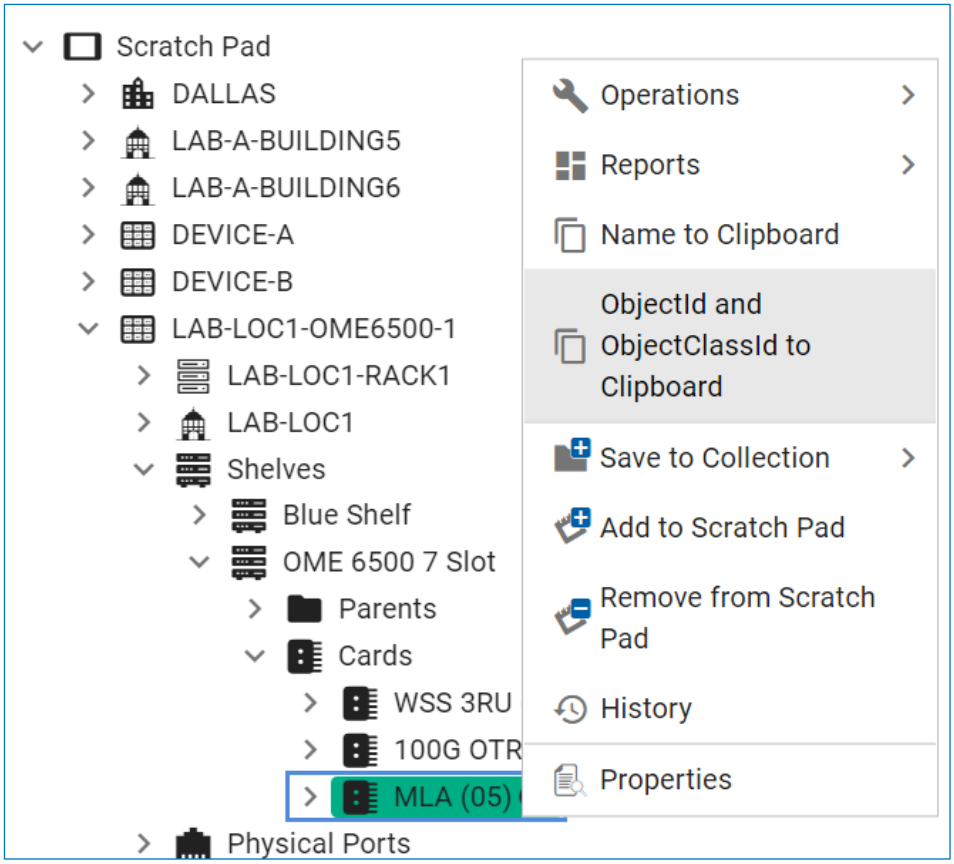
14. Go back to the BPI UI and refresh the inventory tree for the LAB-LOC1-OME6500-1 device. Expand the **MLA Line.1.8-5** port. You can now see a new Physical Connections element. You can expand it to view the Patch Cable (PC) connection and its Termination Points. You should see a similar output if you expend the MLA Line.1.8-8 port.



15. You can also see a new Physical Connections element directly under the LAB-LOC1-OME6500-1 tree. This is another way you can view the connections, which terminate on this device. In your case, you can see both Patch Cable connections, both Line ports, and both Patch Panel ports.

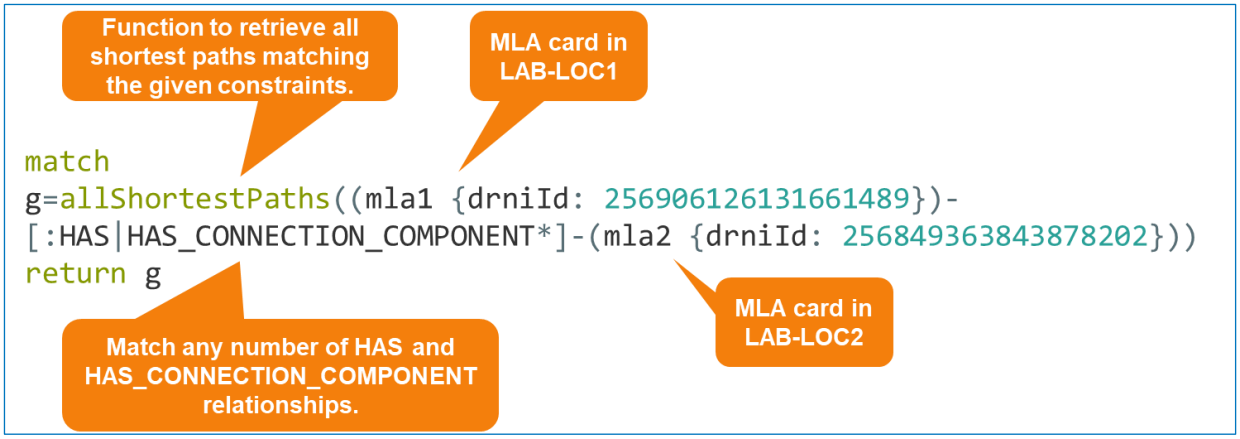


16. You will construct a query to show that you now have end-to-end physical connectivity between OME 6500 devices in LAB-LOC1 and LAB-LOC2. Remember that all connections in LAB-LOC2 and a Cable Fiber between the Patch Panels in both locations are already created. For simplicity, you will display the graph of connections between the MLA cards in both locations. You will use the Neo4j built-in **allShortestPaths** function to display the shortest paths. First, you need to retrieve the drniIds of the MLA cards in both devices. Navigate to **LAB-LOC1-OME6500-1 > Shelves > OME 6500 7 Slot > Cards**, right-click the MLA card and choose **ObjectId and ObjectClassId to Clipboard**.

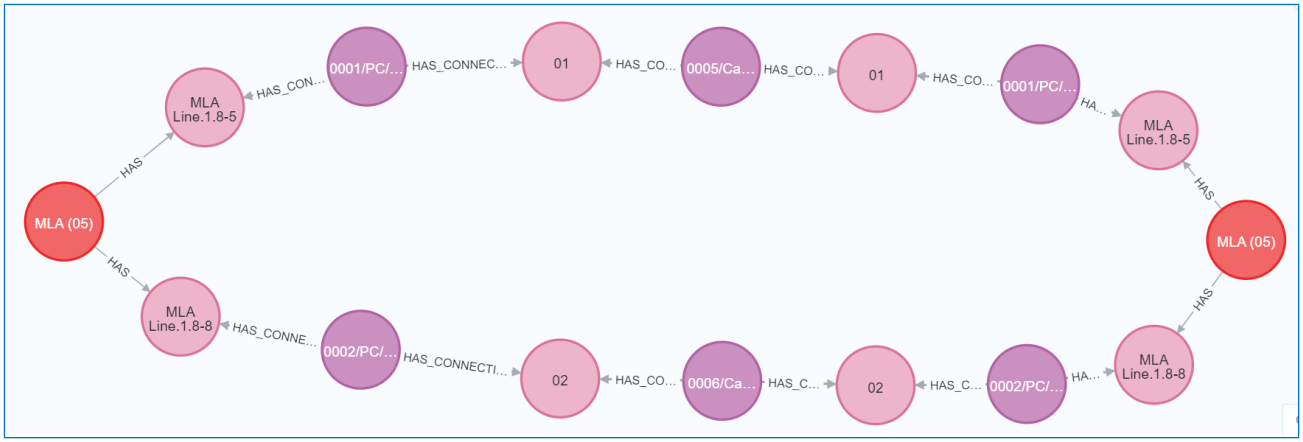


17. Paste the ObjectId (this is the drnId of the MLA card instance) to some text file, and repeat the same step to retrieve the drnId of the MLA card in the LAB-LOC2-OME6500-2 device.
18. In the Neo4j Browser, construct and run the following query. Replace the drniIds with the values you collected in the previous steps.

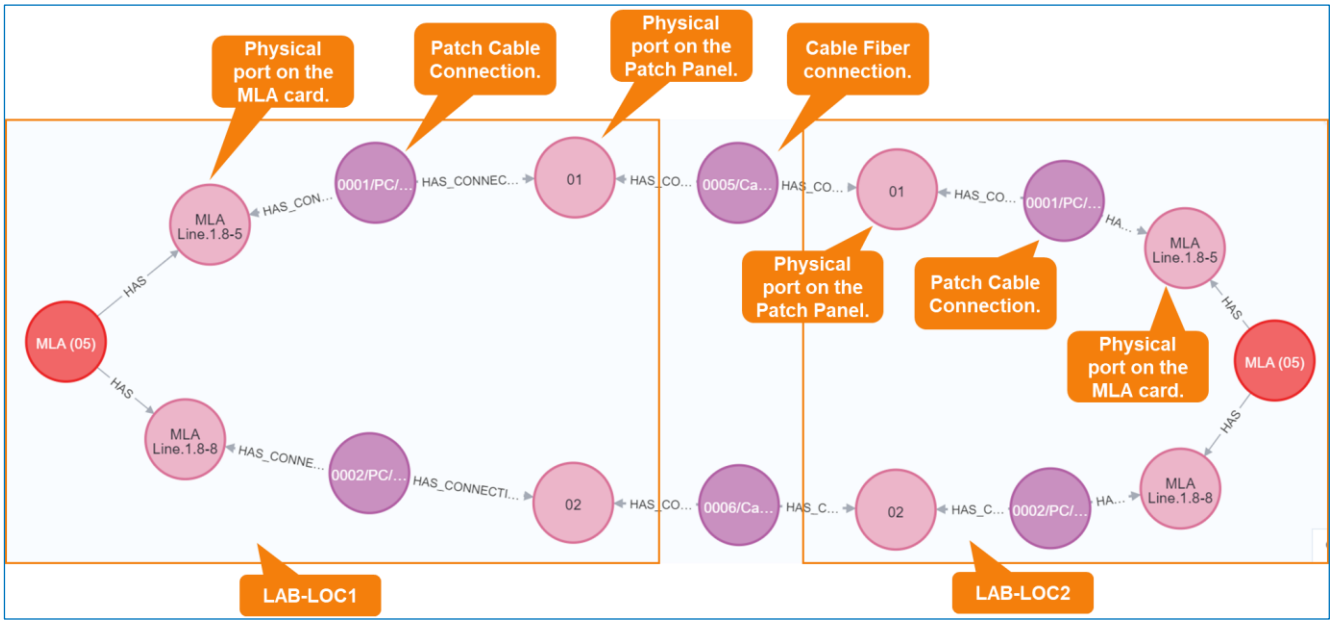
```
match
g=allShortestPaths((mla1 {drniId: 256906126131661489})-
   [:HAS|HAS_CONNECTION_COMPONENT*]-
   (mla2 {drniId: 256849363843878202}))
return g
```



19. If you move the nodes around a bit, you should get a graph like this:



20. The graph displays two separate paths over physical interfaces and physical connections, one for the Rx fiber and one for the Tx fiber.



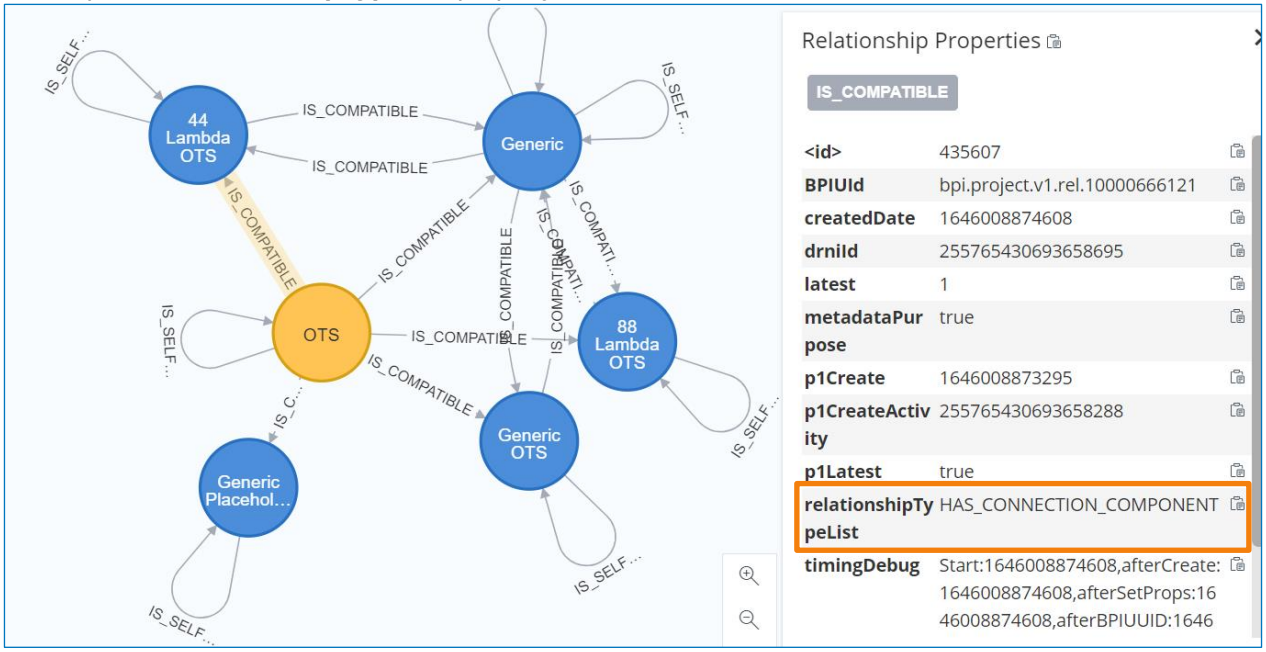
Task 2: Create a Logical Interface

Now that end-to-end physical connectivity between devices in LAB-LOC1 and LAB-LOC2 is established, you can create a logical connection that will be carried by that physical connection. You will create a bidirectional OTS connection between your devices, which will join the two unidirectional physical paths. First, you will inspect the Graph DB to identify the model and compatibilities of the OTS connection. Then you will create a compatible Logical Interface, which will join the two unidirectional ports to a single bidirectional interface and will be able to terminate the OTS logical connection.

1. Identify which logical interfaces are compatible with the OTS connection type, which means that they would be able to be configured as termination points for the OTS connection. In the Neo4j Browser, run the following query

```
match
(m :Archetype :LogicalConnection {name:"OTS"})-
   [:IS_COMPATIBLE]->(n :Archetype :LogicalInterface)
return m,n;
```

   Click any of the relationships from the OTS LogicalConnection node to a LogicalInterface node and inspect the **relationshipTypeList** property.



   The property has only one value, **HAS_CONNECTION_COMPONENT**. This means that this is the only relationship type that can be created from an instance of an OTS logical connection to an instance of the logical interface.

2. Instead of returning the whole graph, you could always retrieve just the relevant property values by running a query such as:

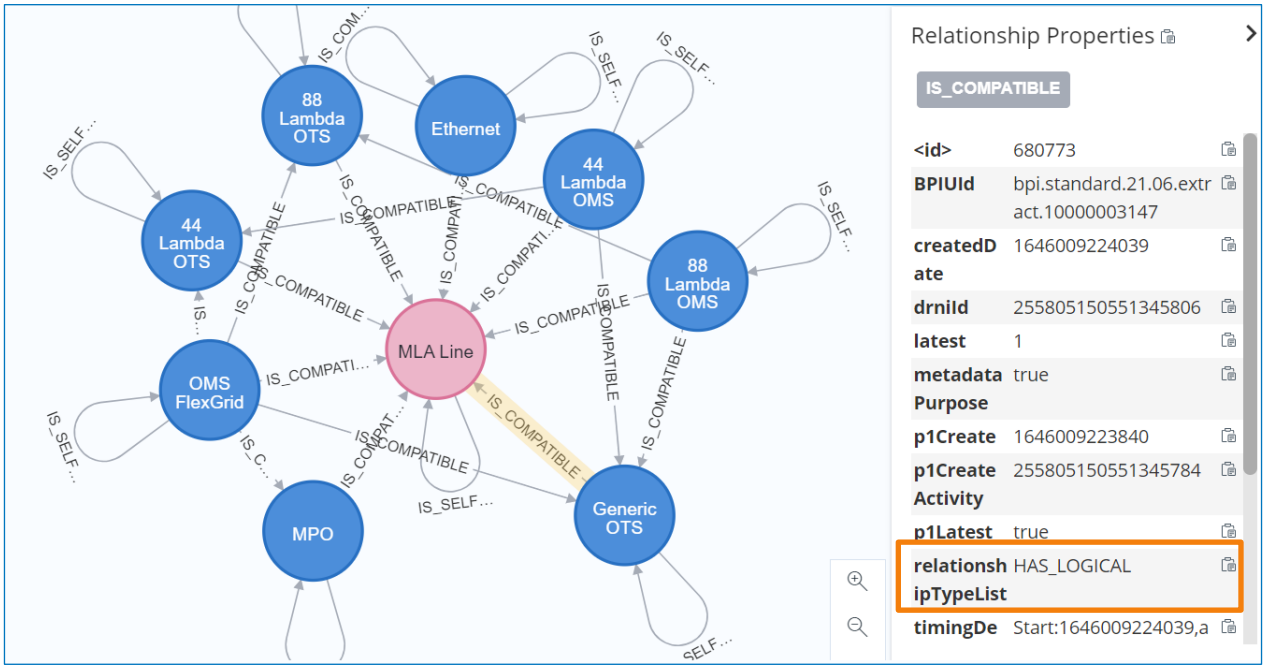
```
match
(m :Archetype :LogicalConnection {name:"OTS"})-
   [r:IS_COMPATIBLE]->(n :Archetype :LogicalInterface)
return n.name,r.relationshipTypeList;
```

	n.name	r.relationshipTypeList
1	"Generic OTS"	["HAS_CONNECTION_COMPONENT"]
2	"88 Lambda OTS"	["HAS_CONNECTION_COMPONENT"]
3	"Generic"	["HAS_CONNECTION_COMPONENT"]
4	"44 Lambda OTS"	["HAS_CONNECTION_COMPONENT"]
5	"Generic Placeholder"	["HAS_CONNECTION_COMPONENT"]

3. What you are looking for is a logical interface, that is compatible with the OTS logical connection and the MLA Line physical ports. Now identify which logical interfaces are compatible with the MLA Line ports. In the Neo4j Browser, run the following query:

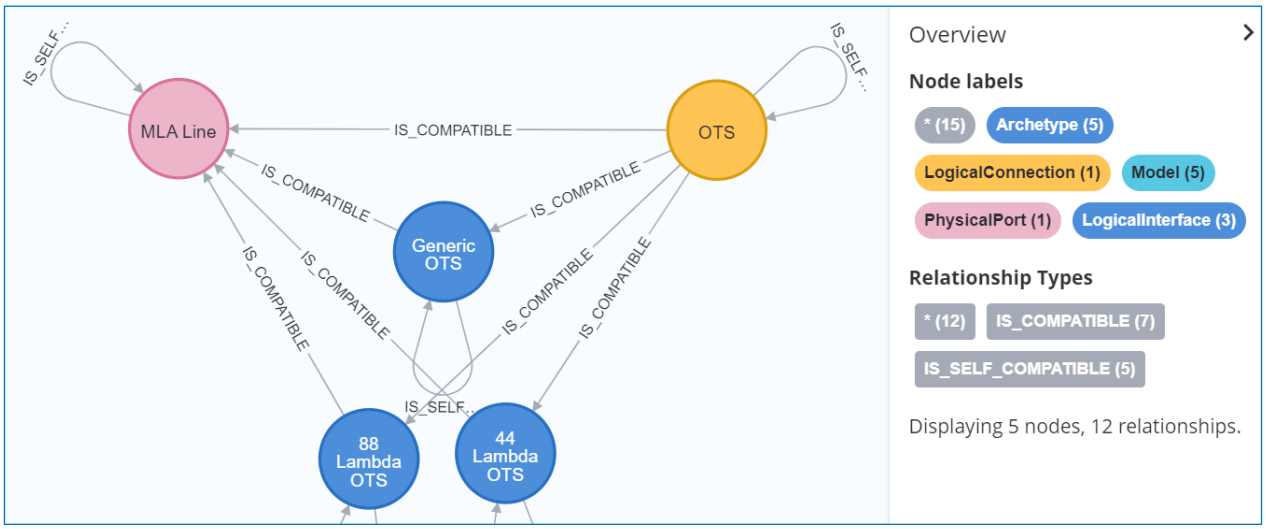
```
match
(m :Archetype :PhysicalPort {name:"MLA Line"})<-[:IS_COMPATIBLE]-
   (n :Archetype :LogicalInterface)
return m,n
```

4. Click any of the relationships from a LogicalInterface node to the MLA Line PhysicalPort and inspect the **relationshipTypeList** property. It has only one value, **HAS_LOGICAL**.



5. You can see that there are some logical interfaces, which are compatible with the OTS logical connection and the MLA Line ports. You can also combine both queries into one, to retrieve the names of those LogicalInterface nodes. Run the following query:

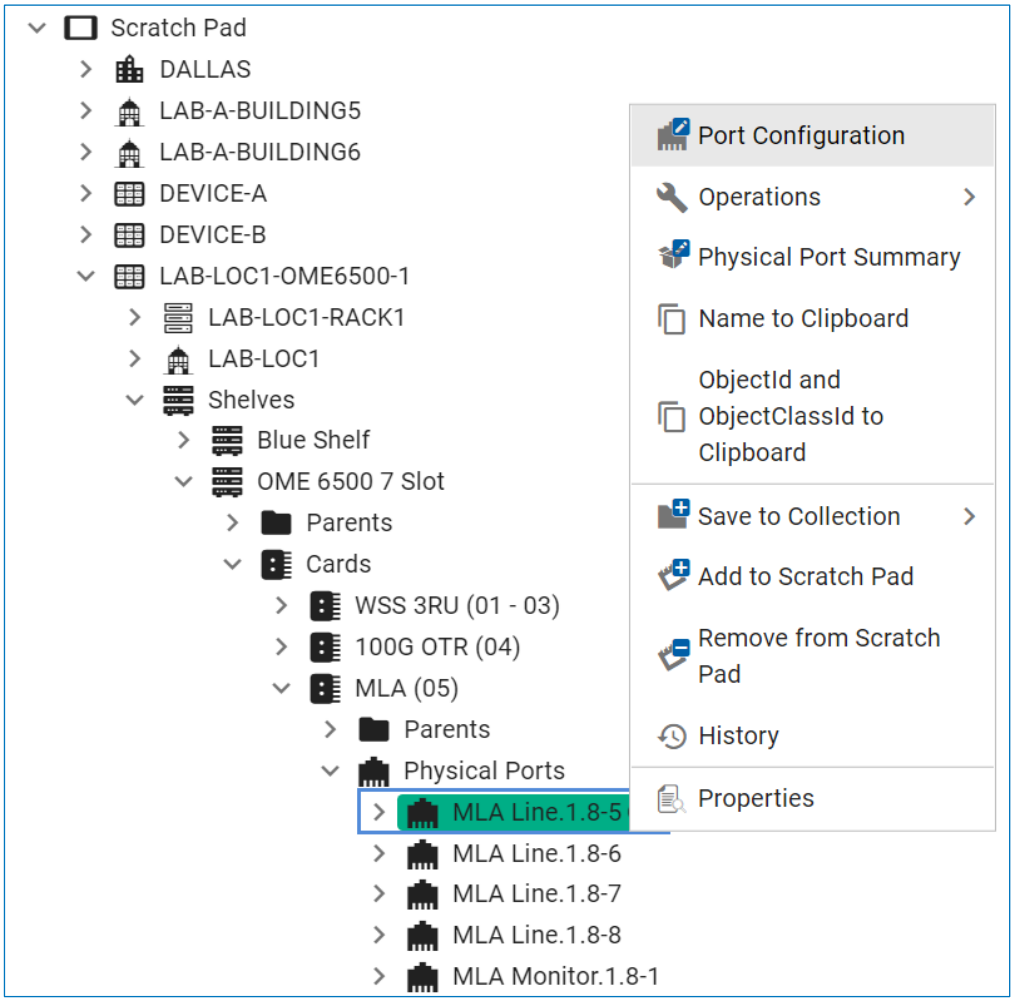
```
match
g=(m :LogicalConnection {name:"OTS"})-
   [:IS_COMPATIBLE]->(n :LogicalInterface)-
   [:IS_COMPATIBLE]->(:PhysicalPort {name:"MLA Line"})
return g
```



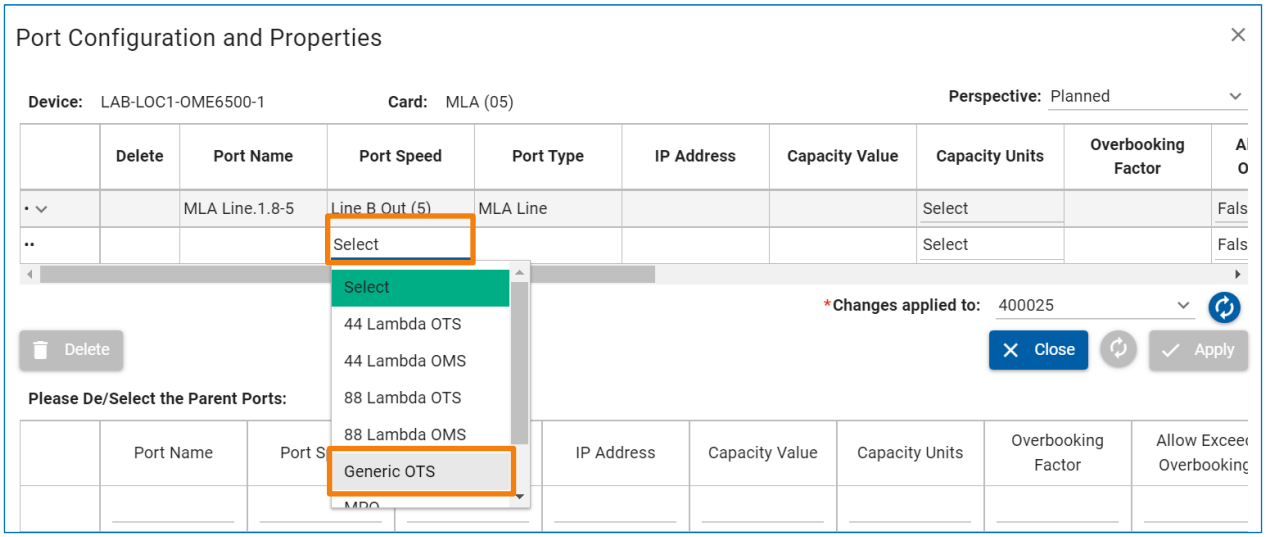
You will choose the **Generic OTS** logical interface to terminate the new OTS logical connection.

NOTE: The Generic OTS interface is routed over two physical paths, one for receive and one for transmit. Generic OTS channelization allows you to create OTS connections without having to worry about how to channelize subsequent overlying OMS (Optical Multiplexer Section) connections, for example using 44 or 88 channels, or as C-Band and L-Band.

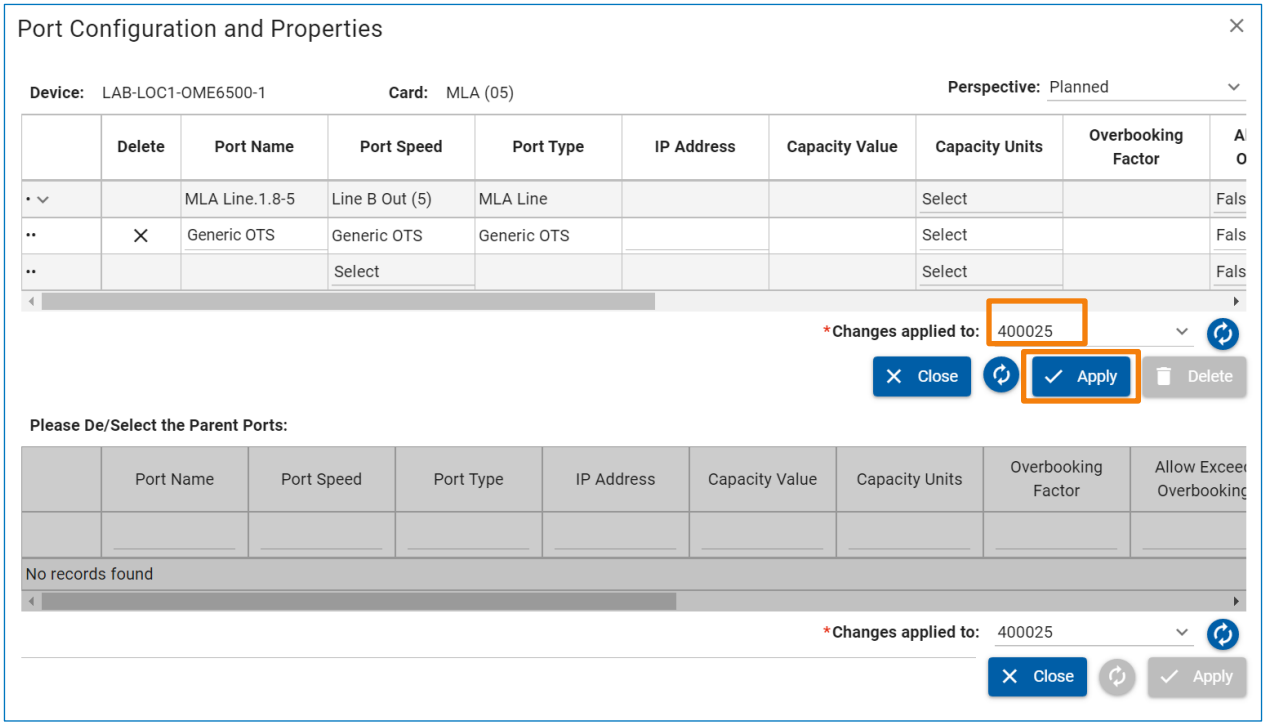
6. In the BPI UI, you will configure a new Generic OTS logical interface. In the inventory tree navigate to **LAB-LOC1-OME6500-1 > Shelves > Cards > MLA (05) > Physical Ports**, right-click the **MLA Line.1.8-5** port, and choose **Port Configuration**.



7. Click the **Select** field in the Port Speed column at the top of the **Port Configuration and Properties** panel. Notice that this list reflects what you determined in a previous step when you ran a query on the Graph DB that returned the compatible logical interfaces. Chose **Generic OTS** from the dropdown menu.

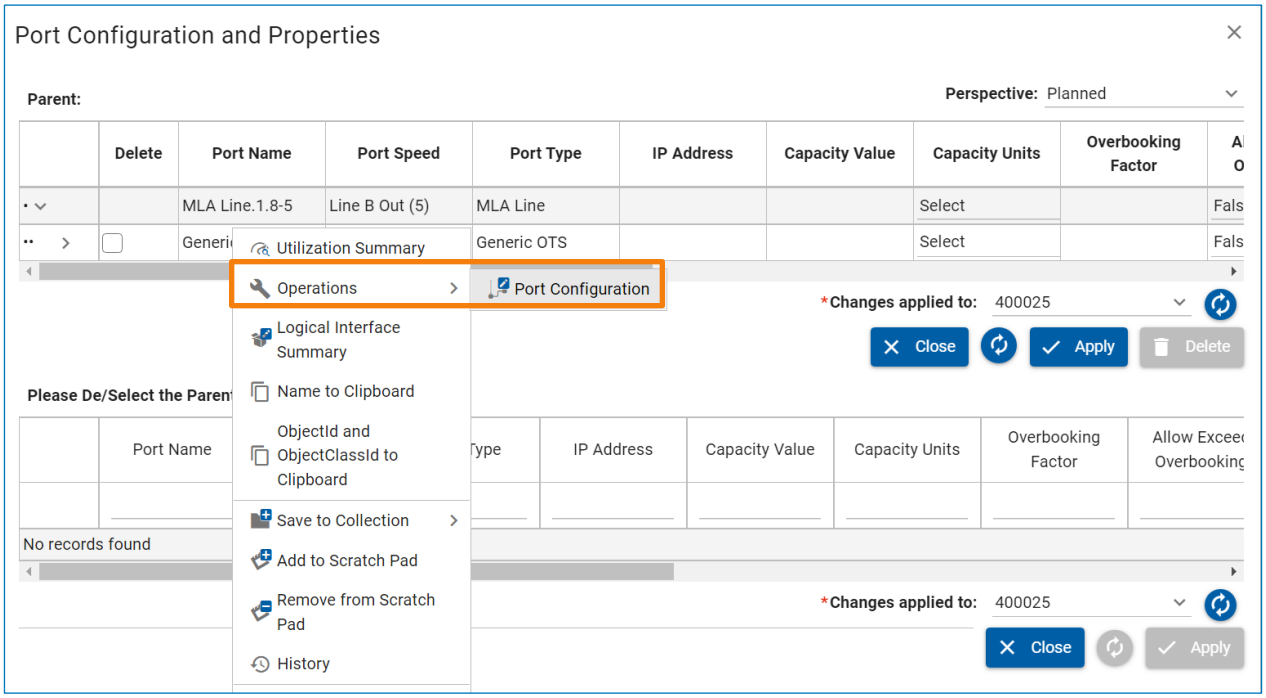


8. Choose your order in the **Changes applied to** field and click **Apply**. Do not close the Port Configuration and Properties panel.

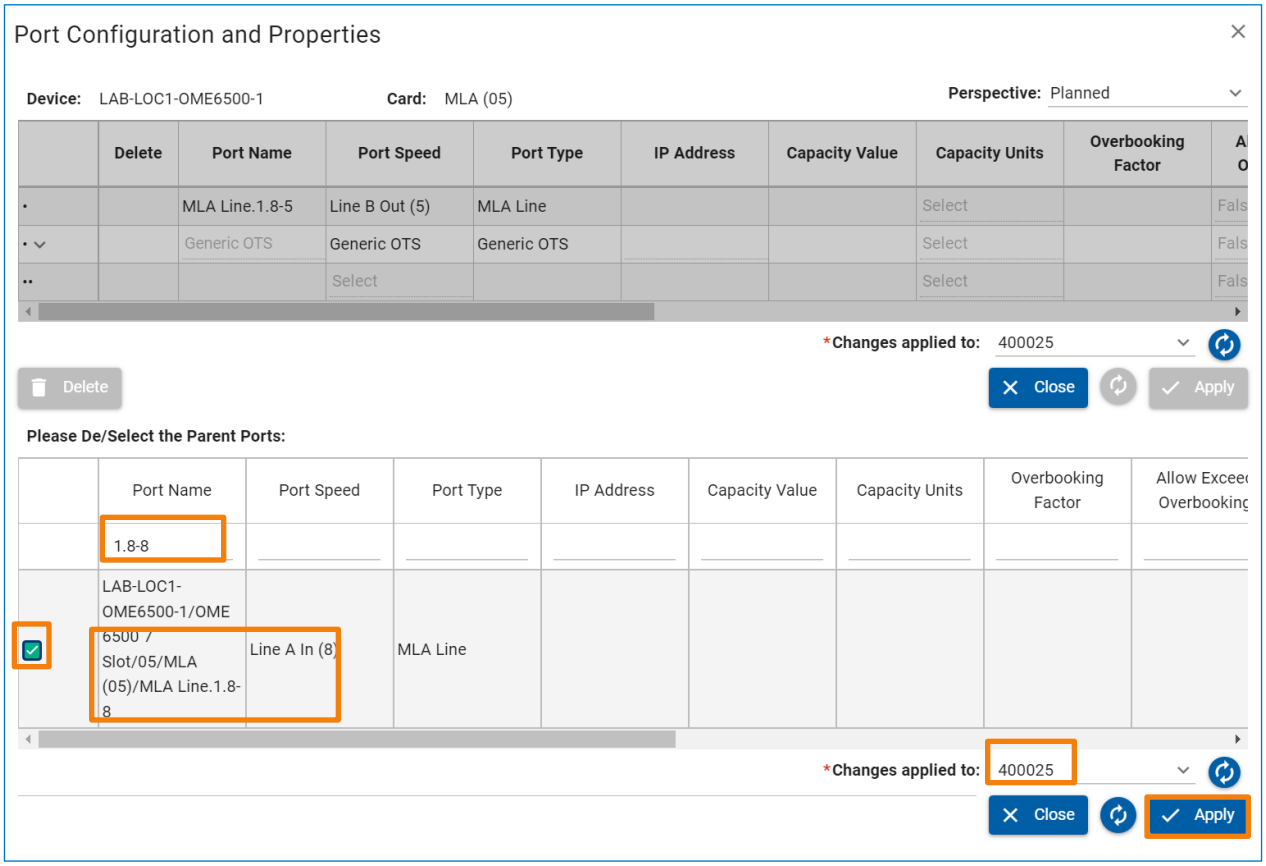


You have now added a Generic OTS logical interface to the Line Out port, which terminates the Tx fiber. This logical interface combines the Rx and Tx ports into a single object, which terminates the bidirectional OTS connection.

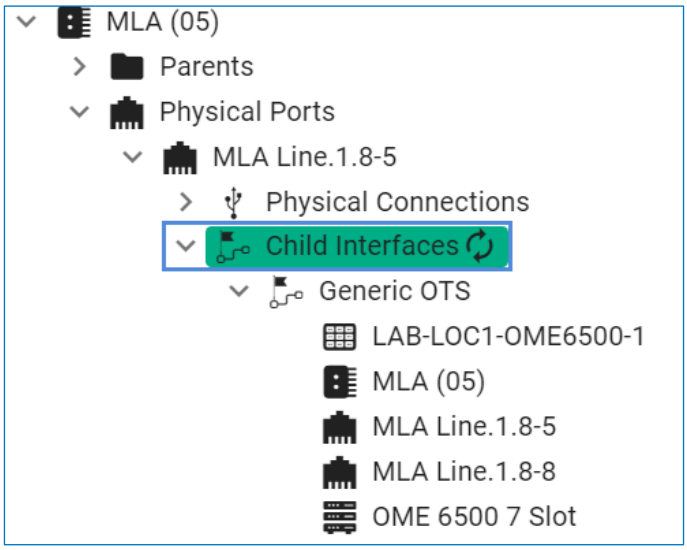
9. To add the Rx (Line In) port to the new logical interface, while still in the Port Configuration and Properties panel, right-click the **Generic OTS** field in the **Port Name** column, expand **Operations** and choose **Port Configuration**.



10. In the Please De/Select the Parent Ports section, you can see that the Line.1.8-5 (Line B Out) port is already selected. In the Port Name filter field, enter **1.8-8**. Click the checkbox to select the **Line.1.8-8** (Line A In) port. Select your order and click **Apply**. Once the configuration is saved, you can close the Port Configuration and Properties panel.

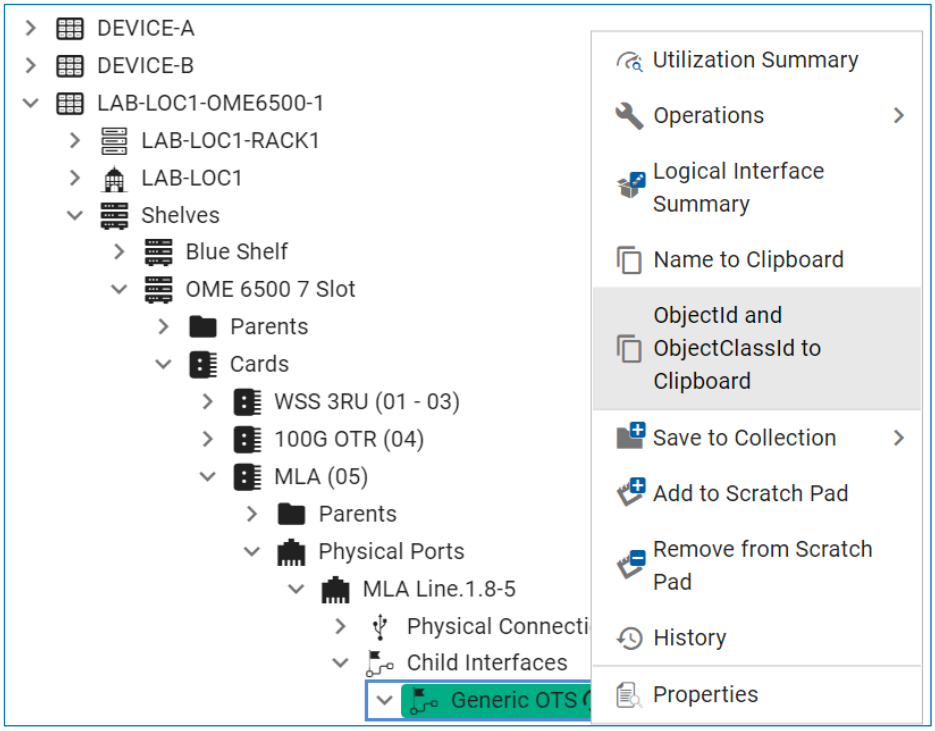


11. In the inventory tree, refresh the **MLA Line.1.8-5** port. You can see a new **Child Interfaces** element, which contains the **Generic OTS** interface and its two parent Line ports. You could see the same configuration by expanding the MLA Line.1.8-8 port.



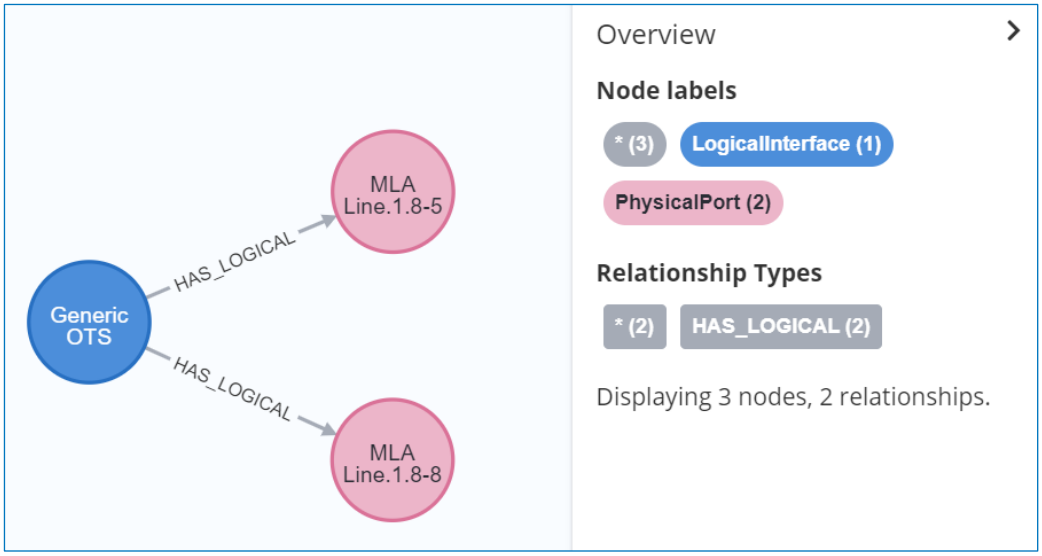
The bidirectional logical interface is now created on LAB-LOC1-OME6500-1.

12. You will now inspect how the newly created logical interface instance is related to the parent physical ports in the Graph DB. First, copy the drnId of the **Generic OTS** interface to the clipboard.



13. In the Neo4j Browser, run the following query. Replace the drnId value with the value you copied in the previous step. Note that for brevity, the Activity nodes will be excluded from the returned graph.

```
match
(m{drniId:256849363843878829})-[]-(n)
where not n:Activity
return m,n
```

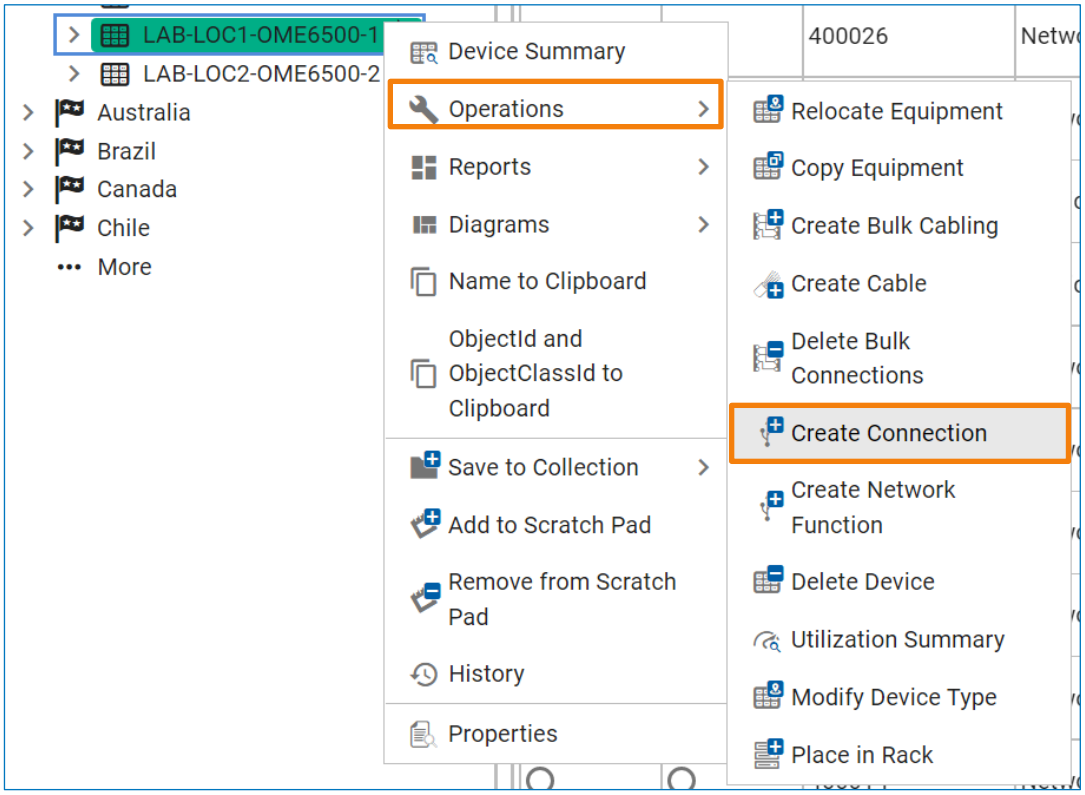


You can see the expected results, the **Generic OTS** logical interface instance has the **HAS_LOGICAL** relationship to the two MLA Line physical ports. This confirms what you identified in a previous step in this task, where you were inspecting the relationshipTypeList property of the IS_COMPATIBLE relationship between the Generic OTS and MLA Line Archetypes.

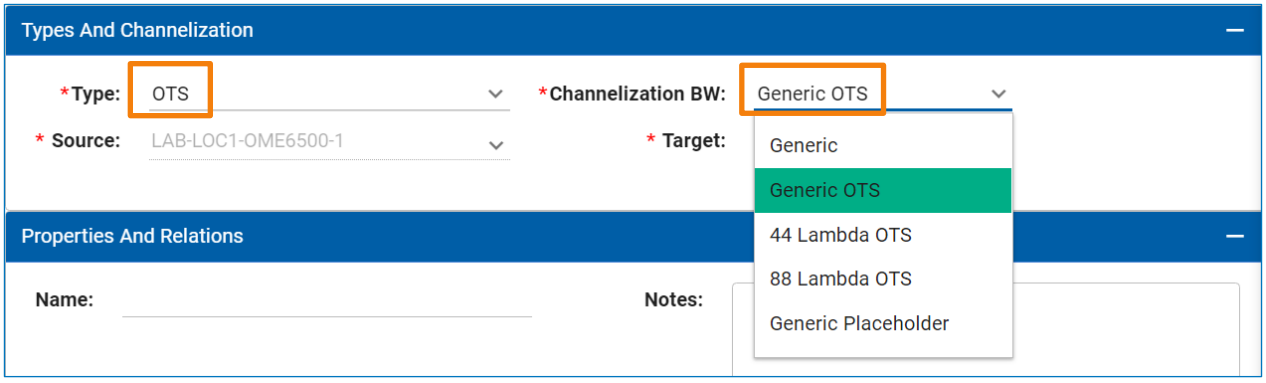
Task 3: Create an OTS Connection

In the previous task, you created the bidirectional Generic OTS logical interface on the LAB-LOC1-OME6500-1 device. The same has already been prepared for you on the LAB-LOC2-OME6500-2 device, so now you can connect the two logical interfaces with an OTS logical connection. In this task, you create an OTS connection, which terminates on the two Generic OTS logical interfaces and carries by the existing underlying physical connections.

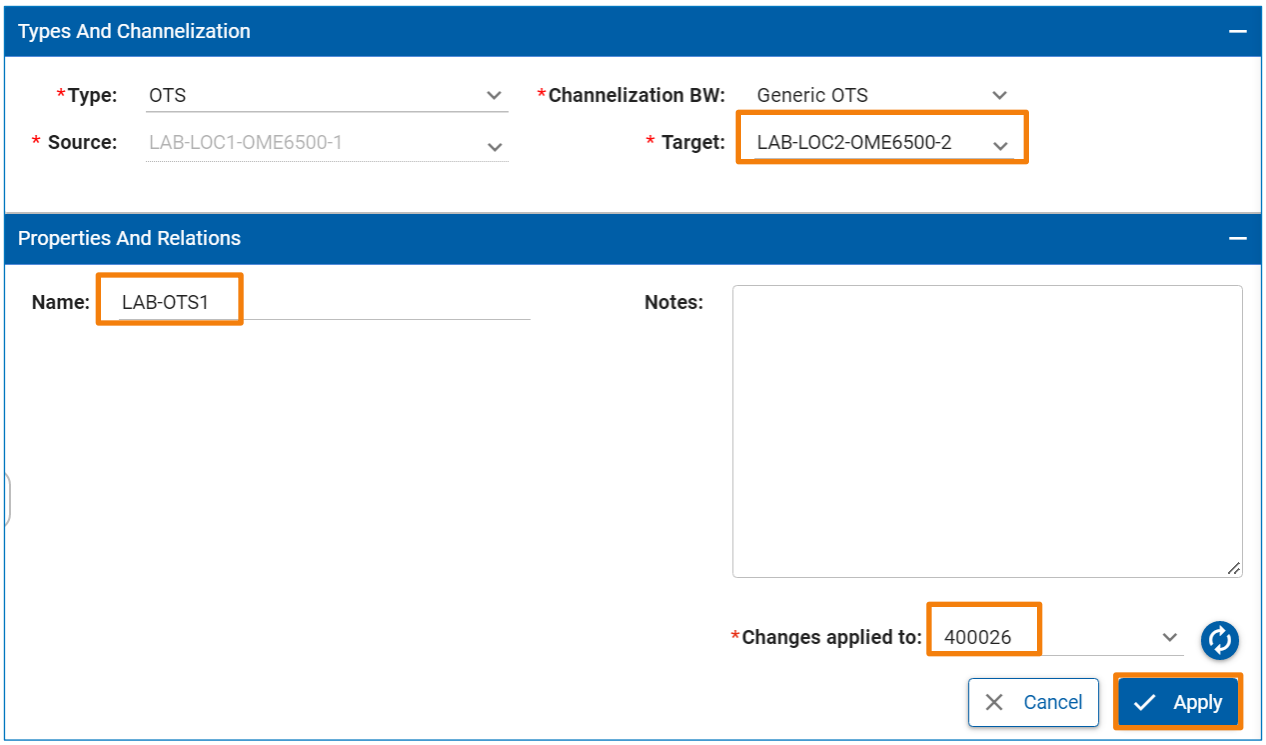
1. From the BPI UI, right-click the **LAB-LOC1-OME6500-1** device in the inventory tree, expand **Operations**, and choose **Create Connection**.



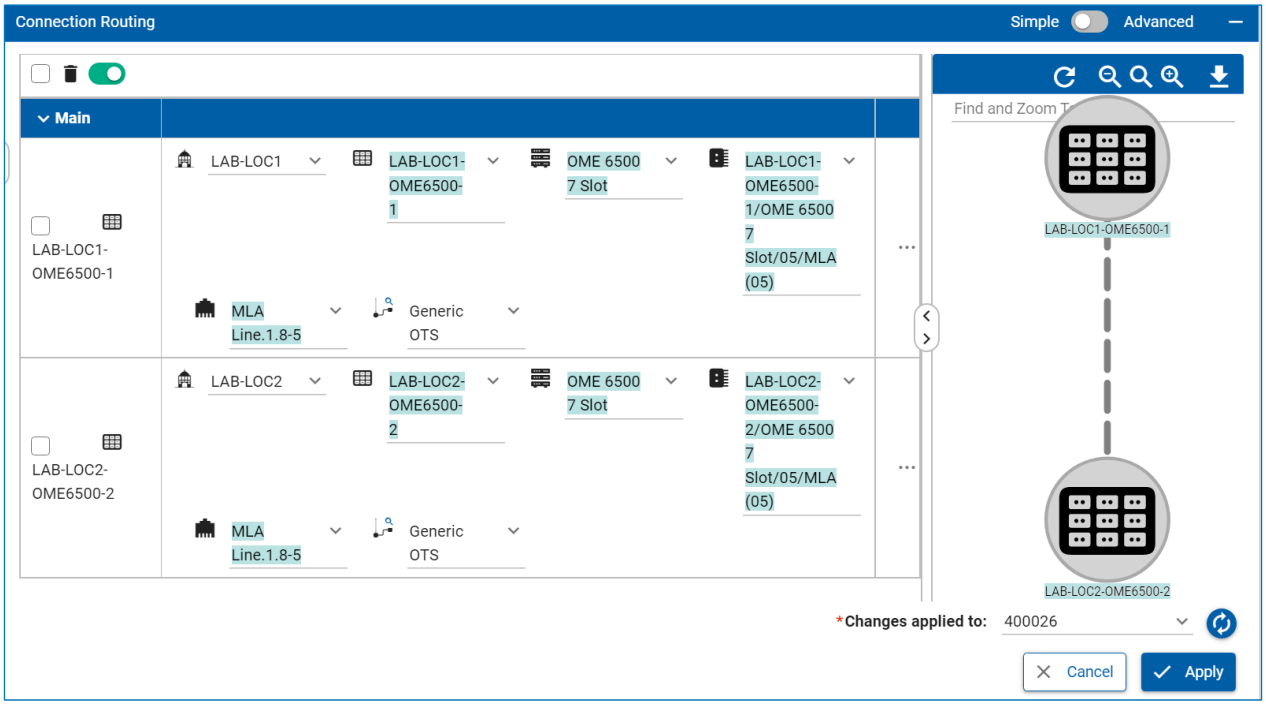
2. Choose **OTS** as the connection **Type** and choose **Generic OTS** from the **Channelization BW** dropdown menu.



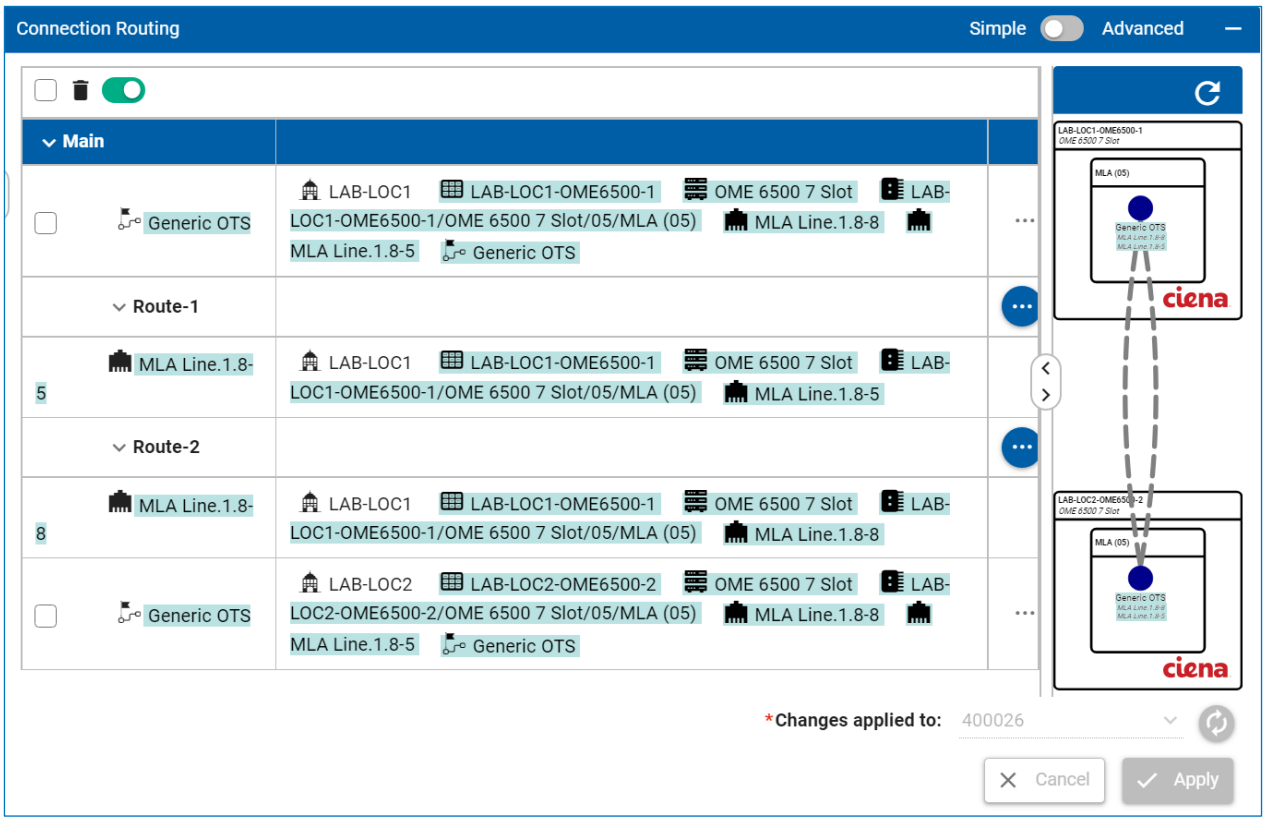
3. Drag-and-drop or type in **LAB-LOC2-OME6500-2** to the **Target** device field. Enter **LAB-OTS1** for the connection Name, choose your order, and click **Apply**.



4. The Connection Routing panel is displayed and populated with the Source and Target devices. On both the source and target rows, choose the following options by clicking the dropdown menus:
   a. Shelf: **OME 6500 7 Slot**
   b. Card: **MLA (05)**
   c. Port: **MLA Line.1.8-5**
   d. Channel: **Generic OTS**

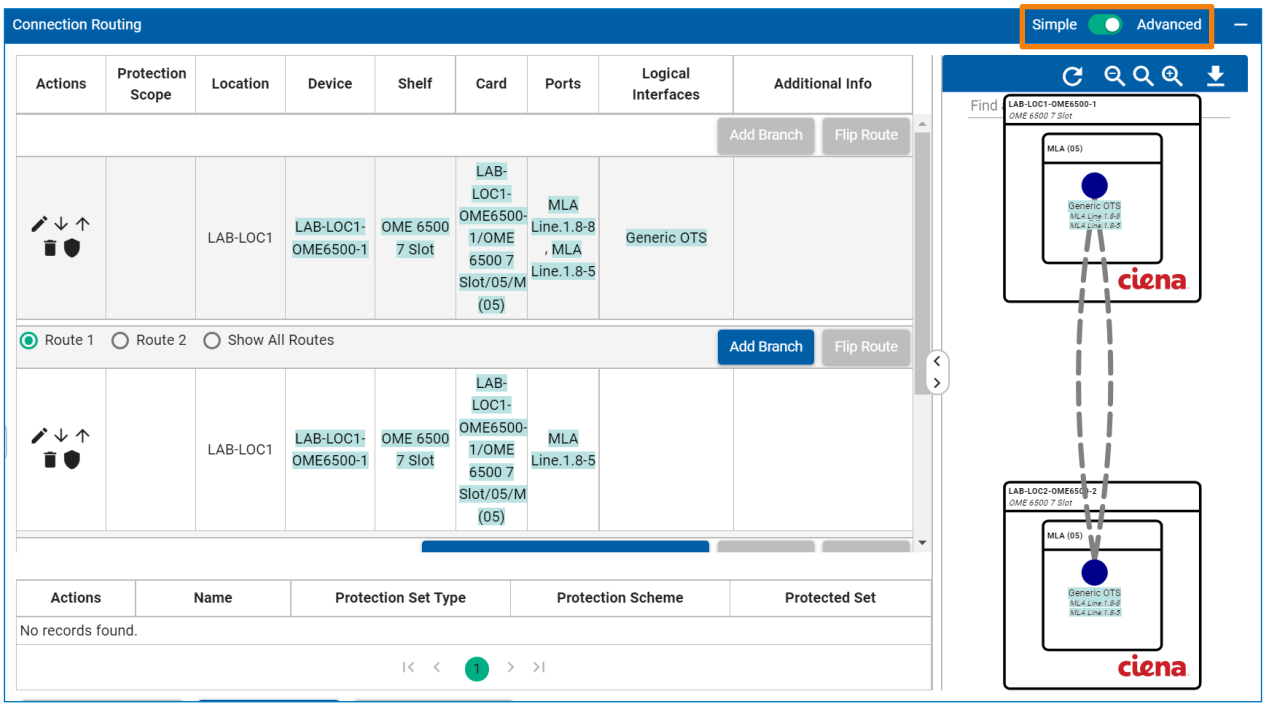


5. Choose your order and click **Apply**. The Connection Routing panel and the routing diagram will be updated.

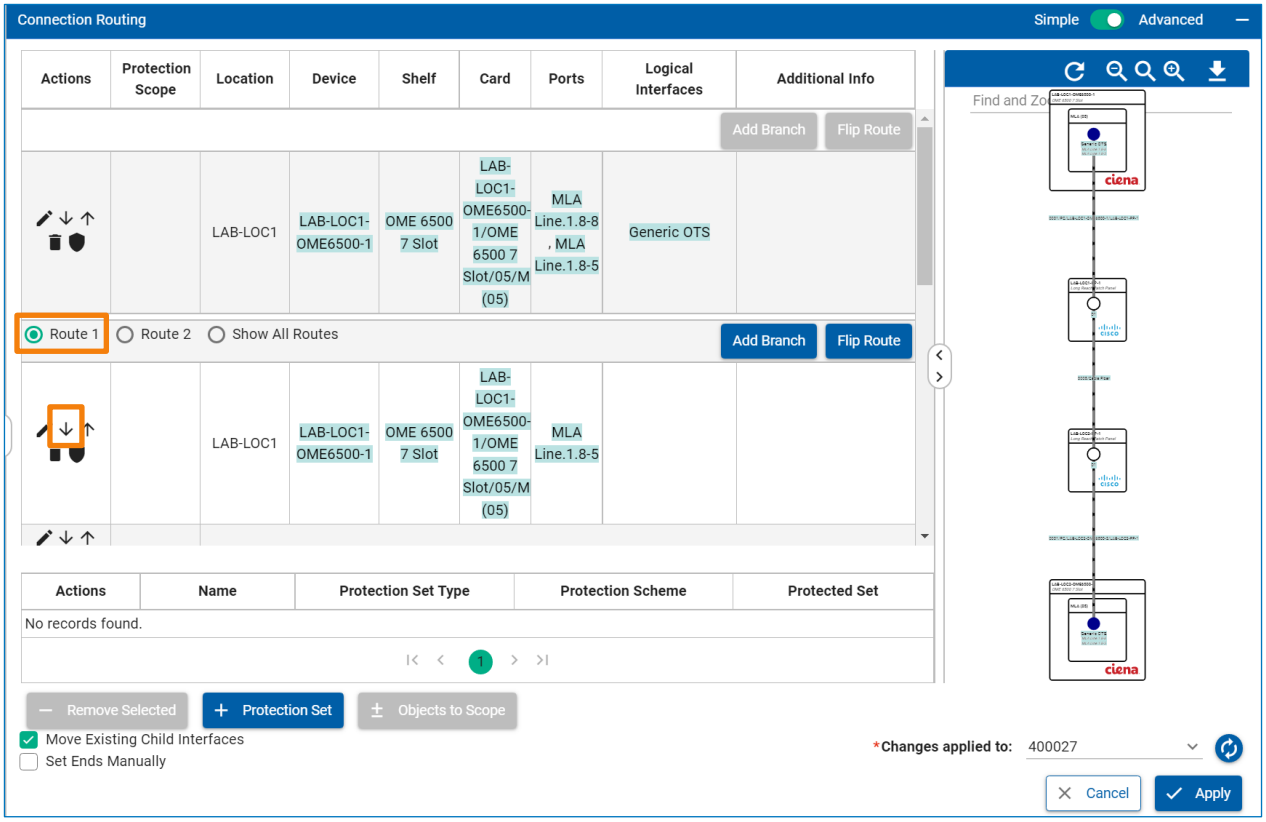


You can see that two Routes (paths) were automatically added, one for the Tx physical fiber, terminating on the MLA Line.1.8-5 (Line B Out) port, and one for the Rx, terminating on MLA Line.1.8-8 (Line A In) port. Both unidirectional routes terminate on the bidirectional Generic OTS logical interface.

6. Switch to the Advanced routing panel to add the physical sections to the connection.

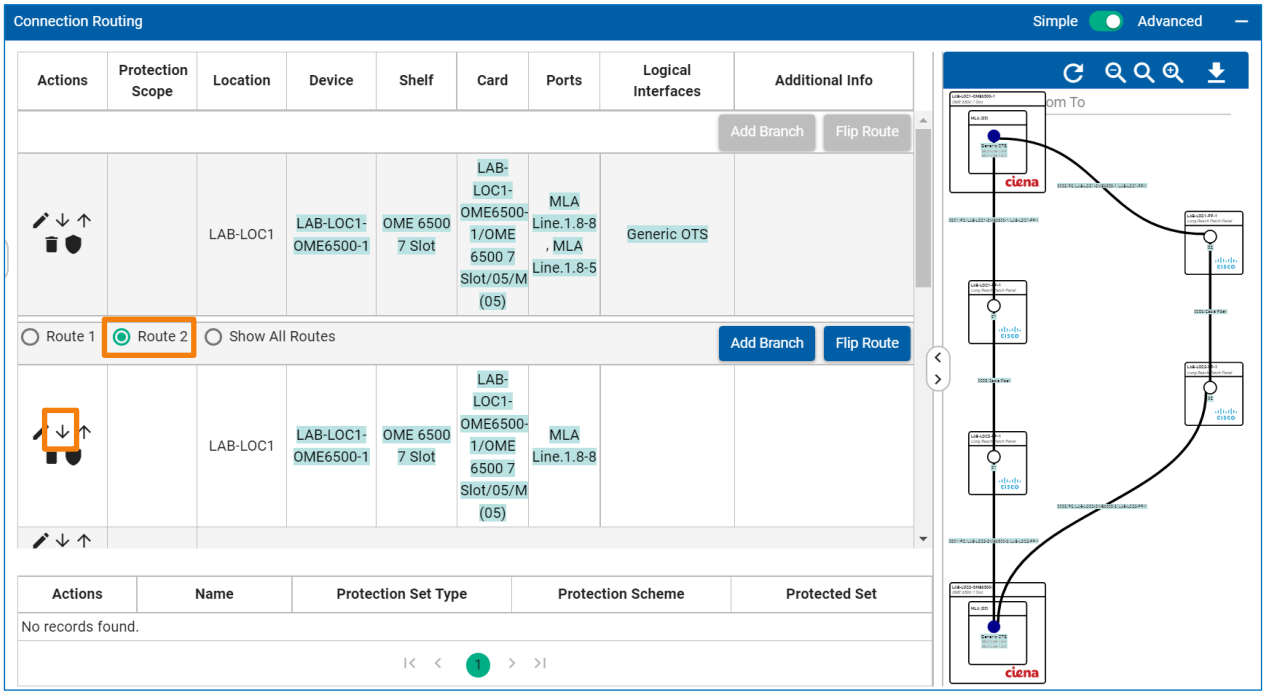


7. Click the down arrow in the first row of Route 1, which contains the MLA Line port, to add a physical section.



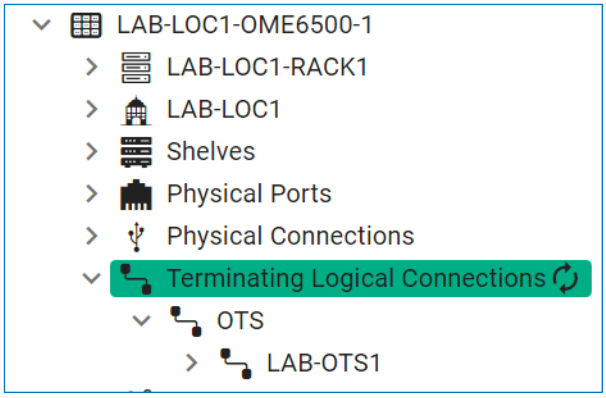
If you scroll down, you will see all the components of the route in separate rows: MLA Line physical port - Patch Cable - Patch Panel physical port in LAB-LOC1 - Cable Fiber - Patch Panel physical port in LAB-LOC2 - Patch Cable - MLA Line physical port. All the components are also displayed in the routing panel on the right.

8. Add the physical components for Route 2. Click the Route 2 radio button and then click the down arrow in the row with the MLA Line port.



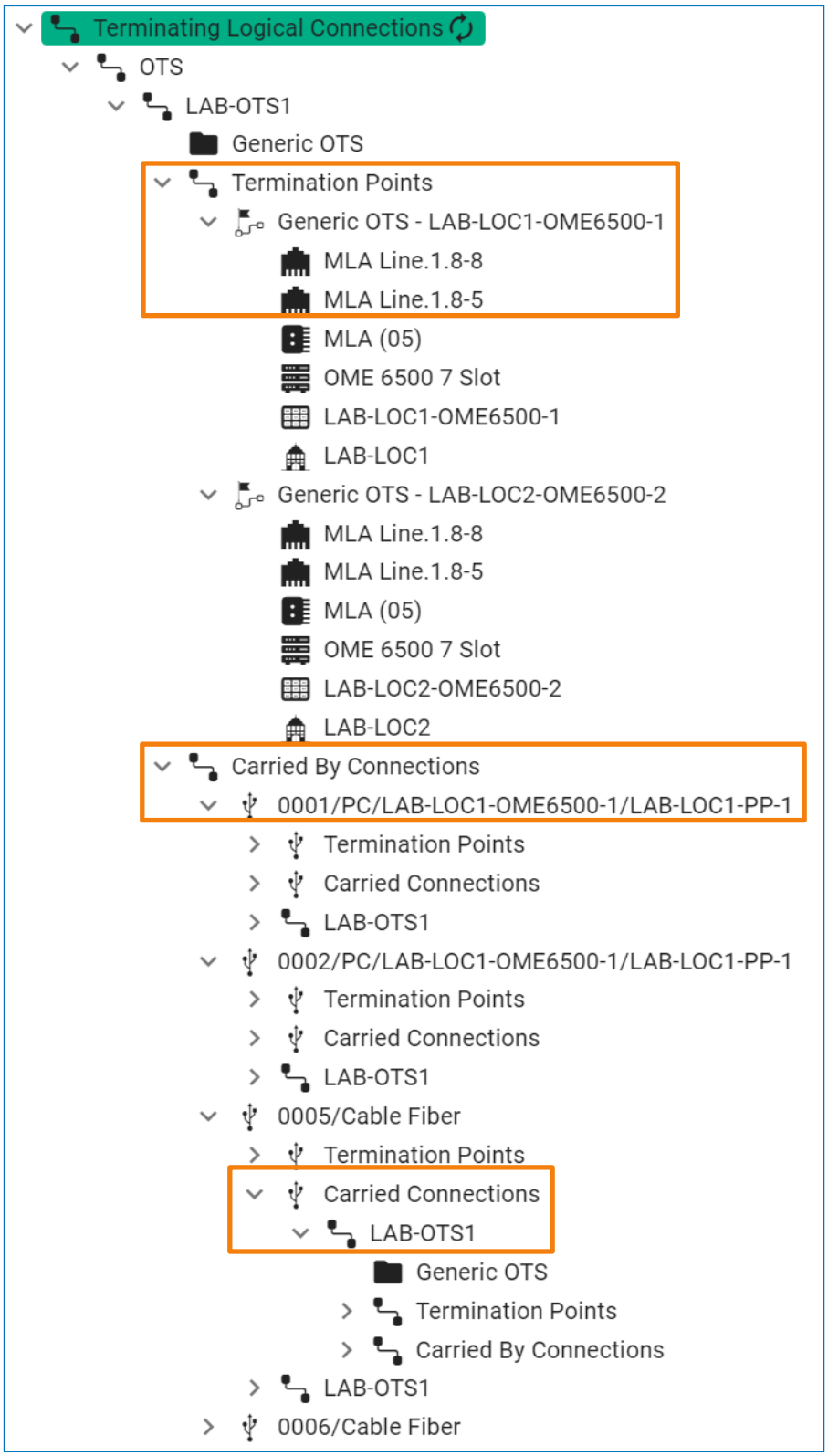
Notice the updated routing diagram, which now shows two branches, one for the receive and one for the transmit fiber.

9. Choose your order number and click **Apply** to apply the configuration.
10. Observe the new elements in the inventory tree. Refresh and expand the **LAB-LOC1-OME6500-1** device.



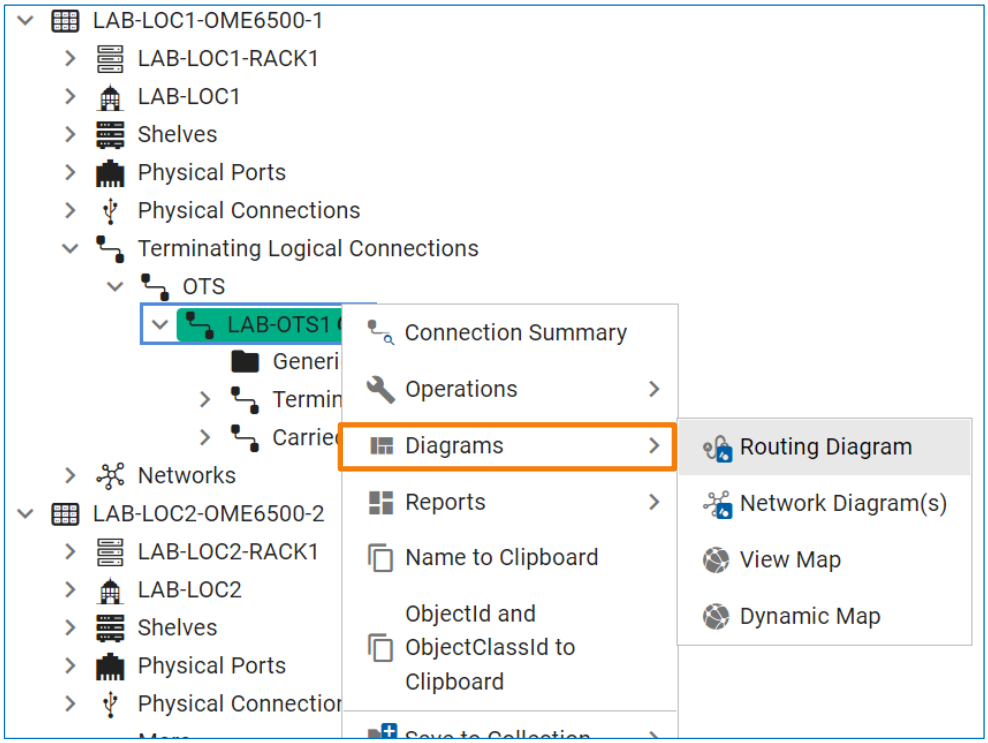
You can see that a new **Terminating Logical Connections** element is present. It provides the information about the OTS connection you created, so the Termination Points and their configuration, and the underlying connections that carry that OTS connection.

11. Try expanding the different elements to get more information about the ports, interfaces, connections, and their relationships.

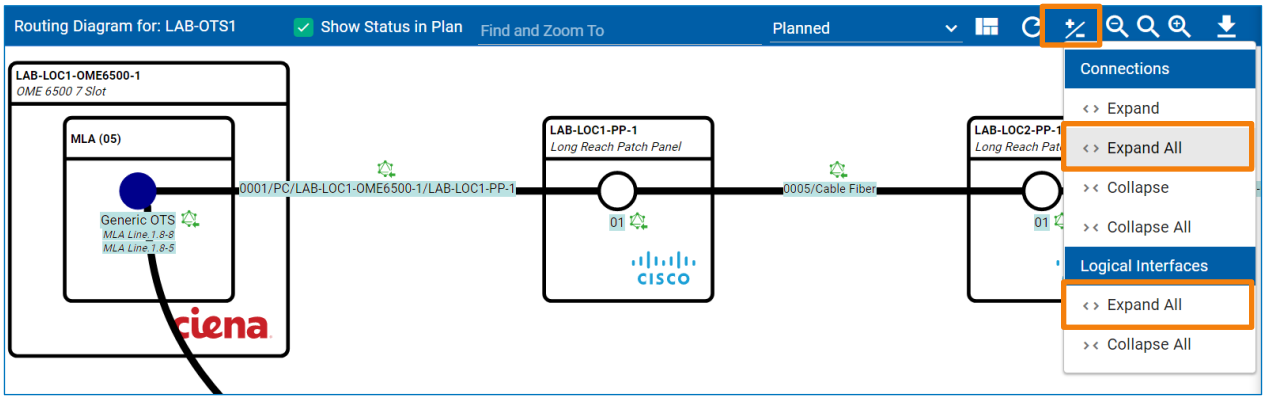


NOTE: Your connection name numbering may vary.

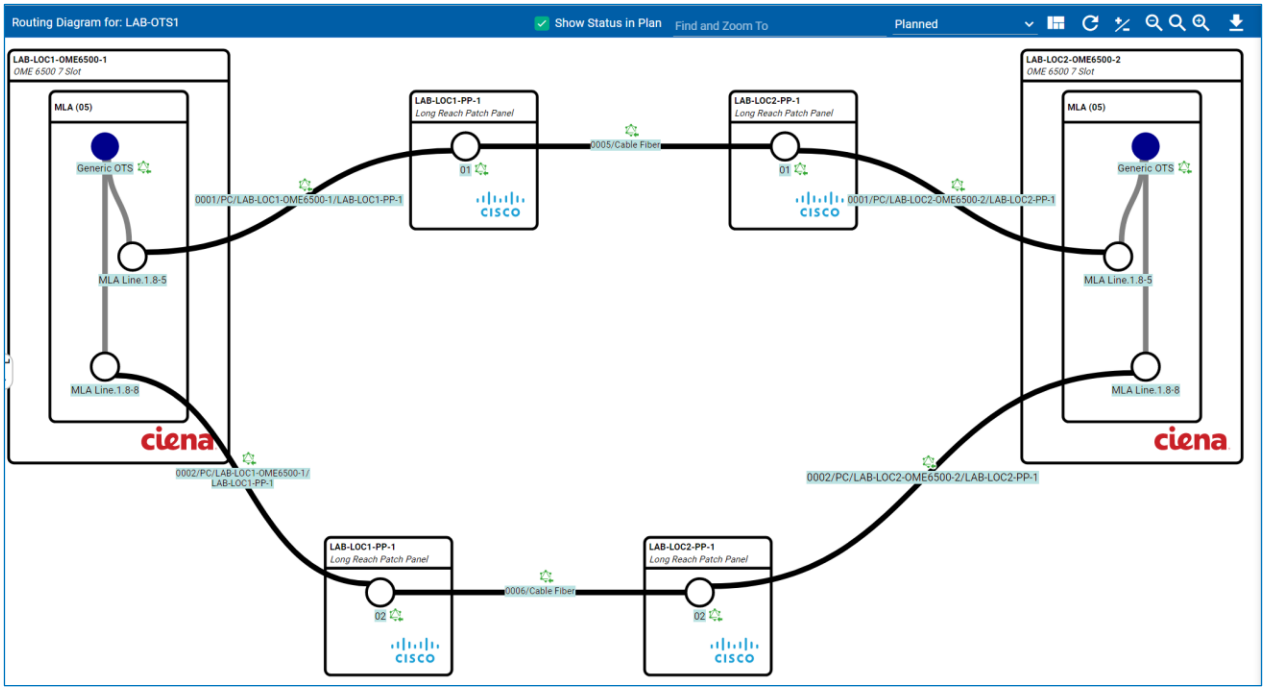
12. View the routing diagram for the OTS connection. Right-click the **LAB-OTS1** connection in the inventory tree, expand **Diagrams** and choose **Routing Diagram**.



13. In the Routing Diagram, click the **+/-** icon, and choose **Expand All** under Connections and Logical Interfaces.

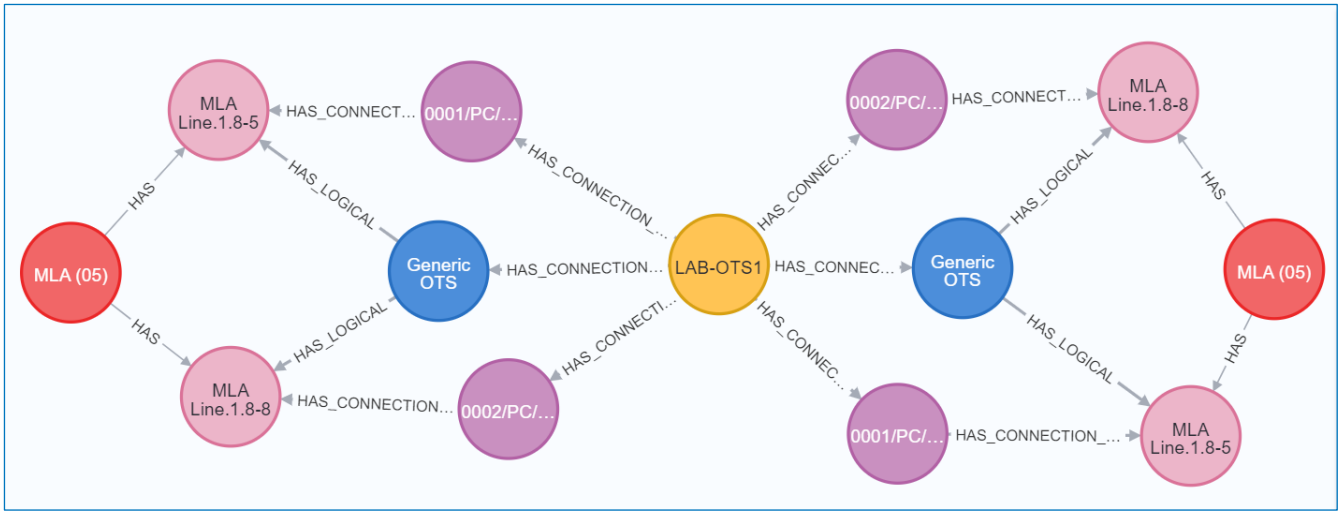


14. Observe the LAB-OTS1 connection routing diagram and inspect all the objects and relationships you created in this lab.



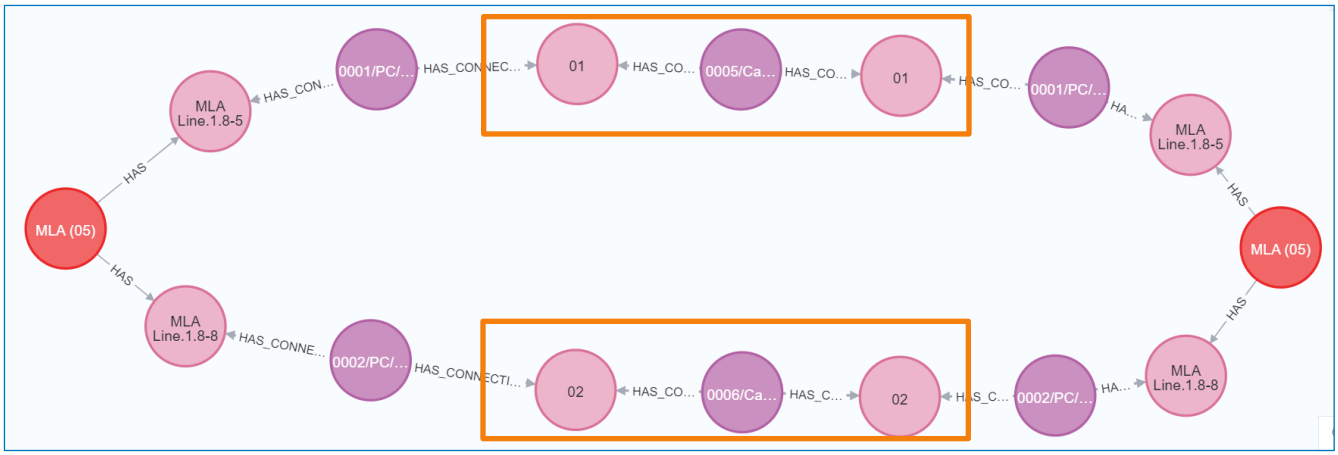
15. You will now verify this new logical connection, its components, and relationships in the Graph DB. In the Neo4j Browser, construct a similar query to the one you used in a previous step when you searched for the shortest paths between the MLA cards in OME 6500 devices in LAB-LOC1 and LAB-LOC2. This time, add the additional **HAS_LOGICAL** relationship type to the match query. Make sure to replace the drnIds in the example with the actual drnId values of both your MLA cards.

```
match

g=allShortestPaths((mla1 {drniId: 2569061261316614 89}) -
   [:HAS|HAS_CONNECTION_COMPONENT|HAS_LOGICAL*]-(mla2 {drniId:
   2568493638438782 02}))

return g
```



You can now see a new end-to-end path from the MLA cards, going through the Generic OTS logical interfaces and the LAB-OTS1 logical connection. Notice the HAS_LOGICAL relationship between the logical interface and the physical ports.

16. Compare this graph to the results that were returned when you ran the query with the allShortestPaths function before you added the logical interface (Task 1, Step 19). Use the screenshot below for reference.



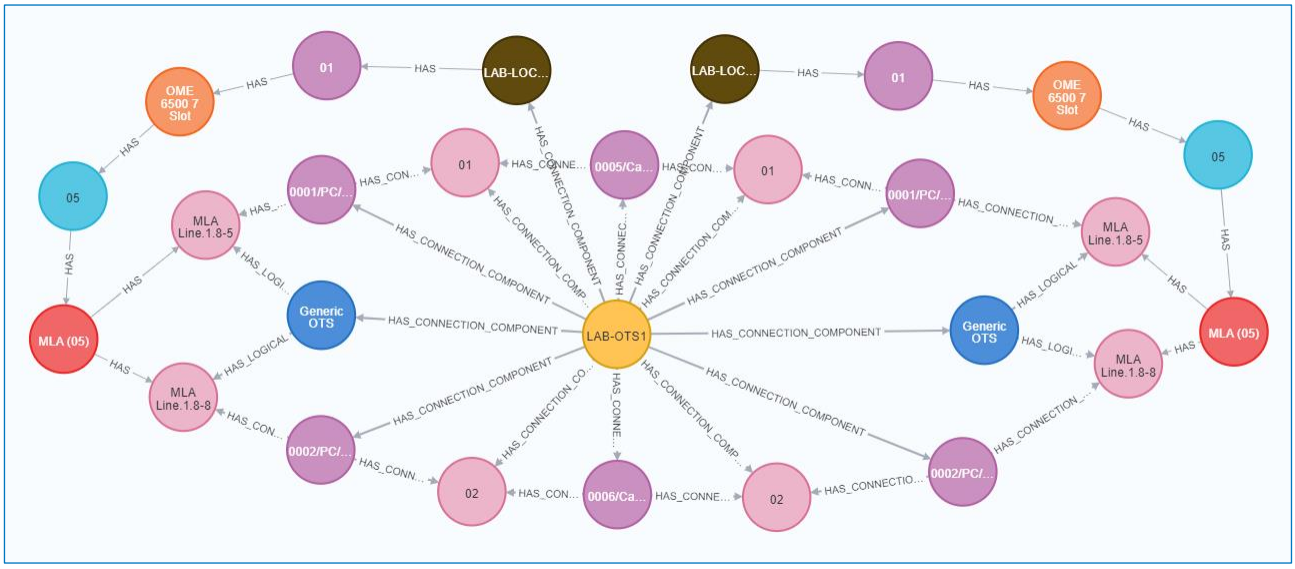
Notice that now some hops on the physical paths are not displayed. The graph is missing the Patch Panel physical ports and the Cable Fiber connection between the patch panels in both locations. The reason for this is that you ran the query with the allShortestPaths function, which only returns the paths with the minimum number of hops. With just the physical connections, the hop count was 8. With the LAB-OTS1 connection created, the minimum hop count is now 6, so only the paths through the OTS connection are returned. If you want to display the other paths between the MLA cards, you can construct a query, which will specify the maximum number of hops.

17. Construct and run the query below. Set the range of the number of occurrences of the relationships (hop count) to be between 1 and 8. Make sure to use the correct drnIds of your MLA cards.

```
match
```

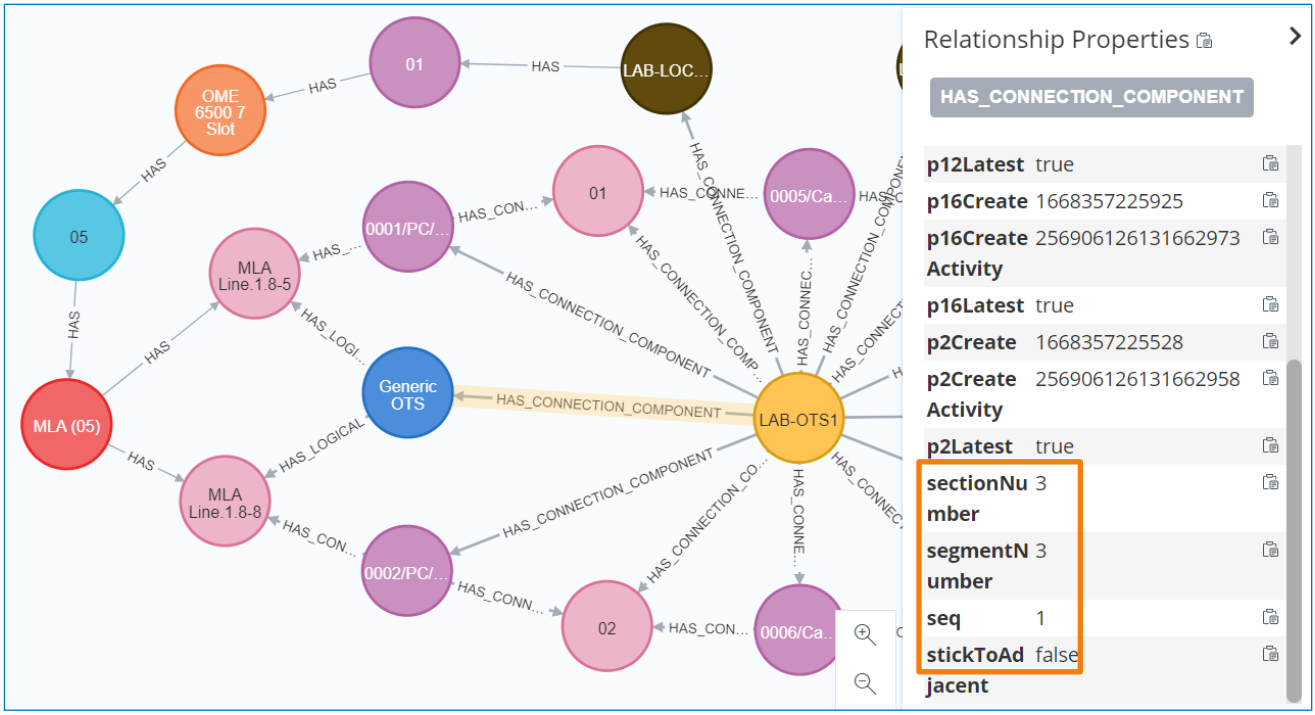
```
g=((mla1 {drniId: 256906126131661489})-
   [:HAS|HAS_CONNECTION_COMPONENT|HAS_LOGICAL*1..8]-(mla2 {drniId:
   256849363843878202}))

return g
```

18. If you rearrange the nodes, you should be able to see a graph similar to the following one.



Notice that the LAB-OTS1 logical connection also has direct relationships with the Device nodes.

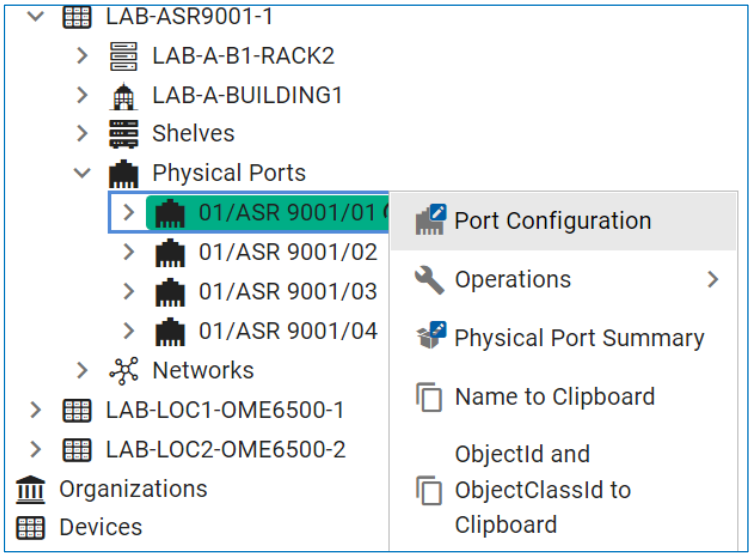
19. Inspect the properties of the displayed **HAS_CONNECTION_COMPONENT** relationships in the graph. Notice the different **sectionNumber**, **segmentNumber**, and **seq** property values, which identify how the routing components are structured. Note that in your lab the property values could vary.



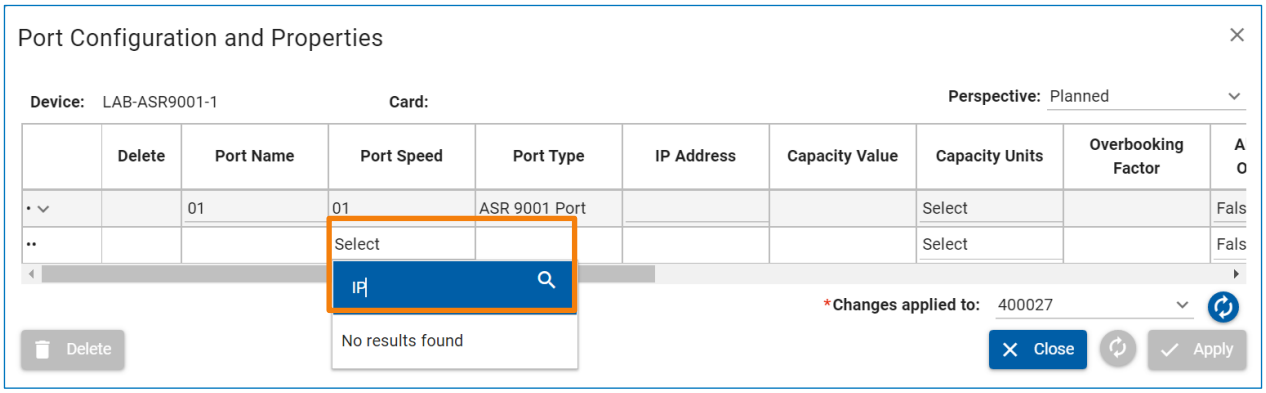
Task 4: Extend the Logical Model

In this task, you will extend the inventory model with new compatibilities. You will add the capability for the physical ports on the ASR 9001 devices to support the "IP Interface" logical interface and verify the functionality in the BPI UI.

1. First, verify the existing compatibilities in the BPI UI. From your Scratch Pad, navigate to the device **LAB-ASR9001-1 > Physical Ports**. You can see the four on-board ports, that are modeled on the ASR 9001 devices. Right-click the **01** port and choose **Port Configuration**.

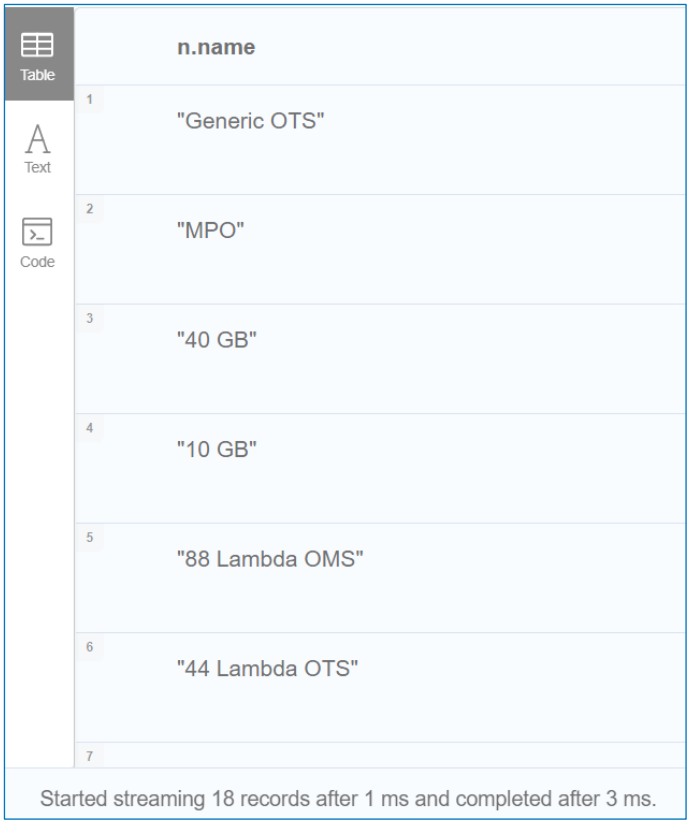


2. In the Port Configuration and Properties panel, click the second row of the Port Speed column. A list of all the supported channelizations is displayed. Type **IP** in the search field. Currently, no results are found.



3. Verify the existing compatibilities between the on-board ports of the ASR 9001 type devices and logical interfaces. In the Neo4j Browser, run this query:

```
match
(m:Archetype:PhysicalPort{name:"ASR 9001 Port"})<-[:IS_COMPATIBLE]-
   (n:Archetype:LogicalInterface)
return n.name
```

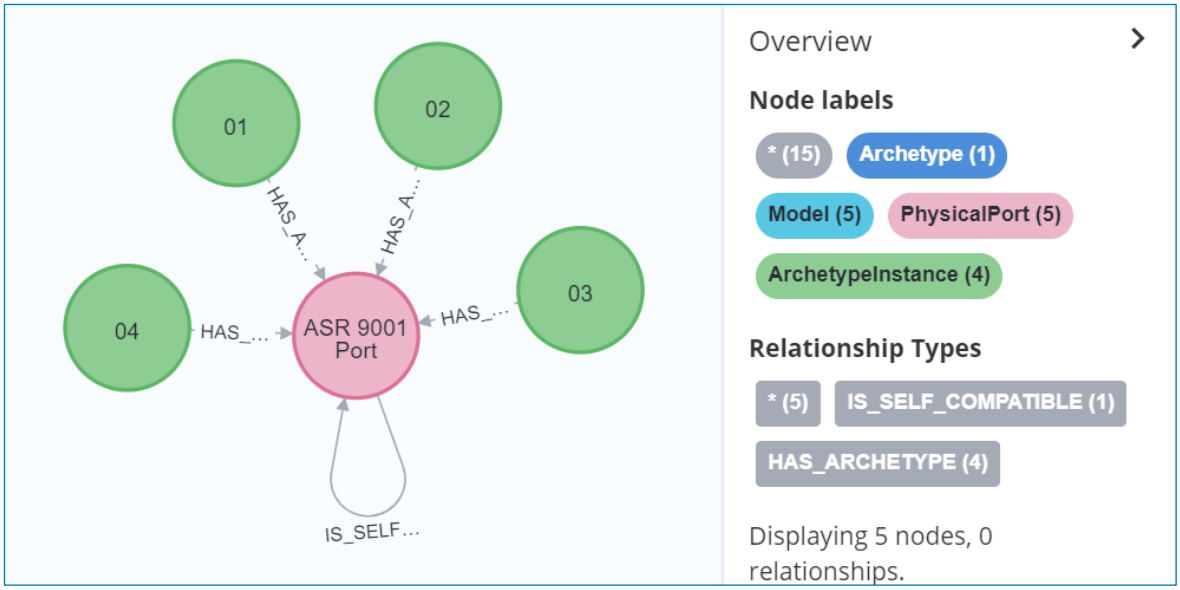


18 logical interfaces are returned, and you can notice that the IP Interface is not among them.

NOTE: The "ASR 9001 Port" Archetype name is provided for your convenience. By now you should be comfortable with retrieving the relevant objects and their drnIds from the BPI UI context menu (right-click on an object - **ObjectId and ObjectClassId to Clipboard**).

4. Remember that, in addition to Archetype relationships to define compatibility, BPI also uses relationships between ArchetypeInstances to additionally define the supported channelization and compatibilities between logical interfaces and other objects. Run the following query to retrieve the ArchetypeInstances, related to the **ASR 9001 Port** Archetype:

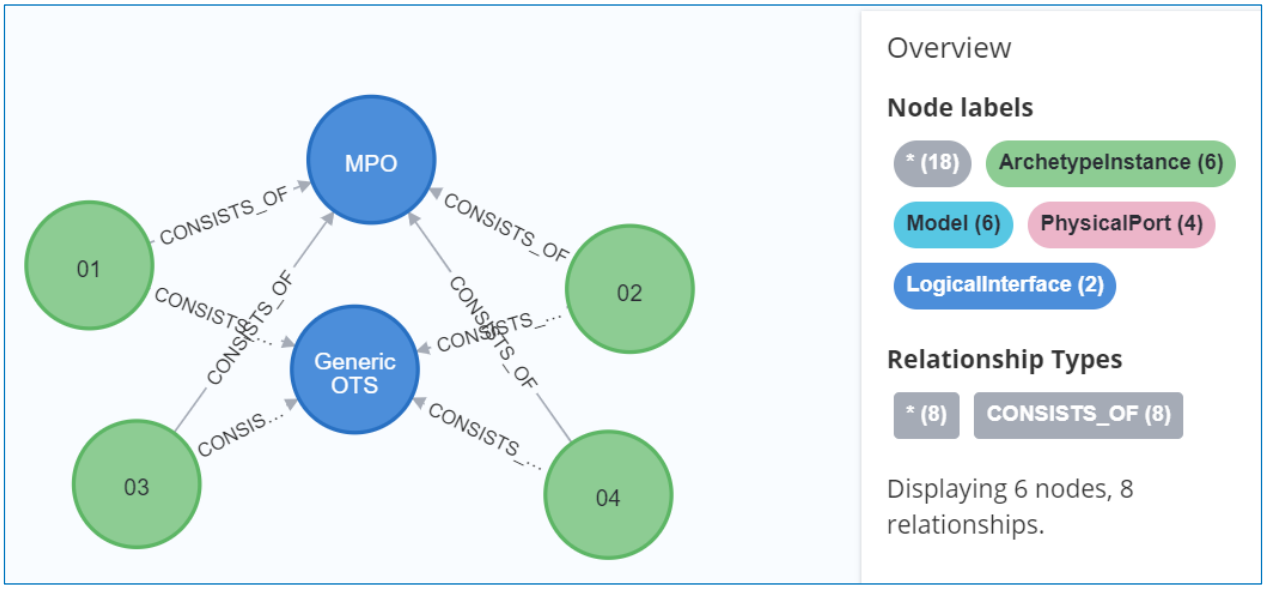
```
match (a:Archetype:PhysicalPort{name:"ASR 9001 Port"})<-
   [:HAS_ARCHETYPE]-(ai)

return a,ai
```



The four ArchetypeInstance nodes represent the four on-board ports that are modeled on the ASR 9001 devices.

5. Verify if there is any compatibility between these port ArchetypeInstances and the IP Interface logical interface. Run the following query:

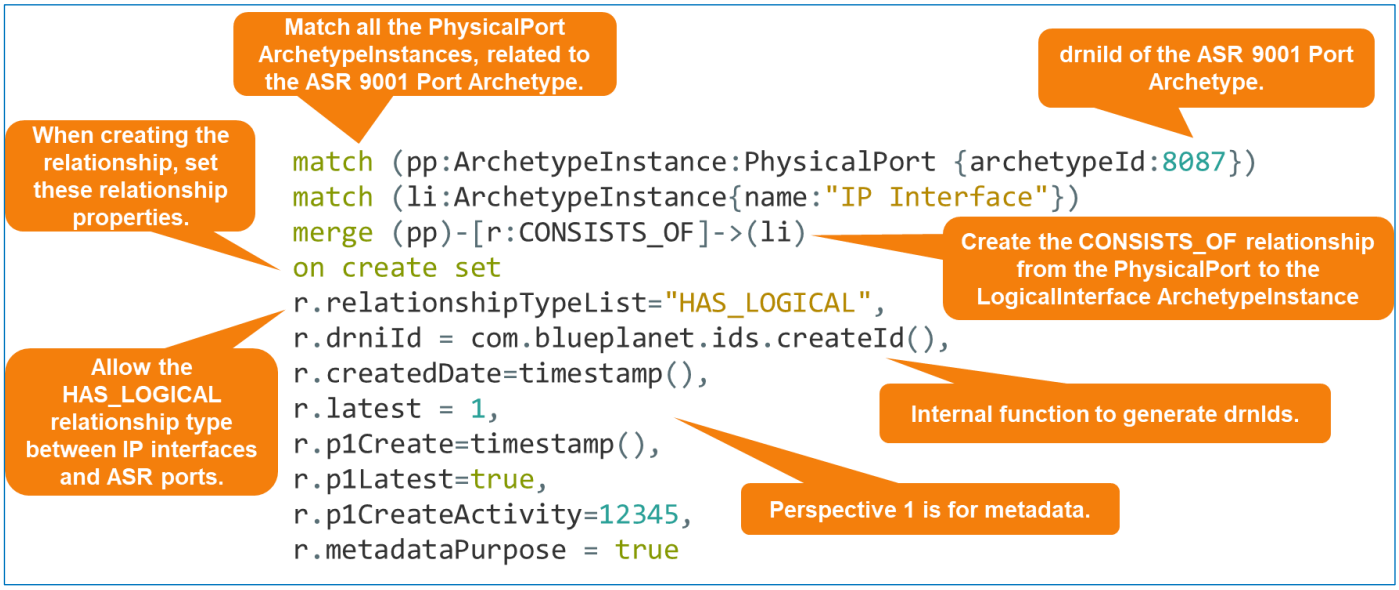
```
match (a:Archetype:PhysicalPort{name:"ASR 9001 Port"})<-
   [:HAS_ARCHETYPE]-(ai)
match (ai)-[]->(li:ArchetypeInstance:LogicalInterface)
return ai,li
```



The IP Interface node is not returned in the graph.

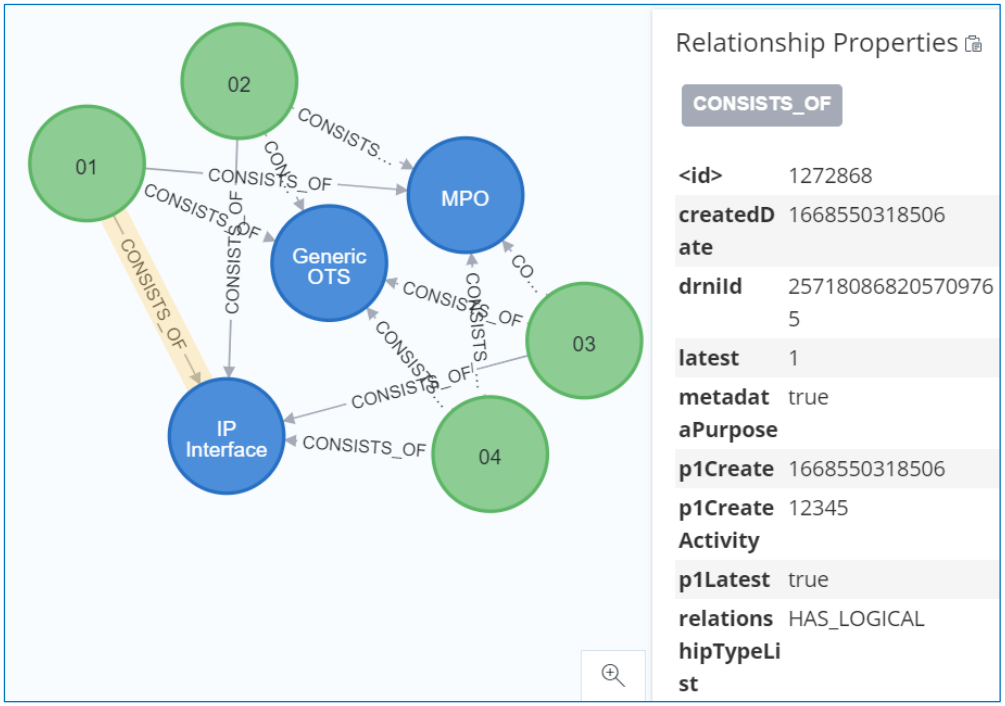
6. You will now add the relevant relationships between the IP Interface logical interface ArchetypeInstance, and all the ASR 9001 port ArchetypeInstances. The relationships will define that those instances are compatible. Construct the following query:

```
match (pp:ArchetypeInstance:PhysicalPort {archetypeId:8087})
match (li:ArchetypeInstance{name:"IP Interface"})
merge (pp)-[r:CONSISTS_OF]->(li)
on create set
r.relationshipTypeList="HAS_LOGICAL",
r.drniId = com.blueplanet.ids.createId(),
r.createdDate=timestamp(),
r.latest = 1,
r.p1Create=timestamp(),
r.p1Latest=true,
r.p1CreateActivity=12345,
r.metadataPurpose = true
```



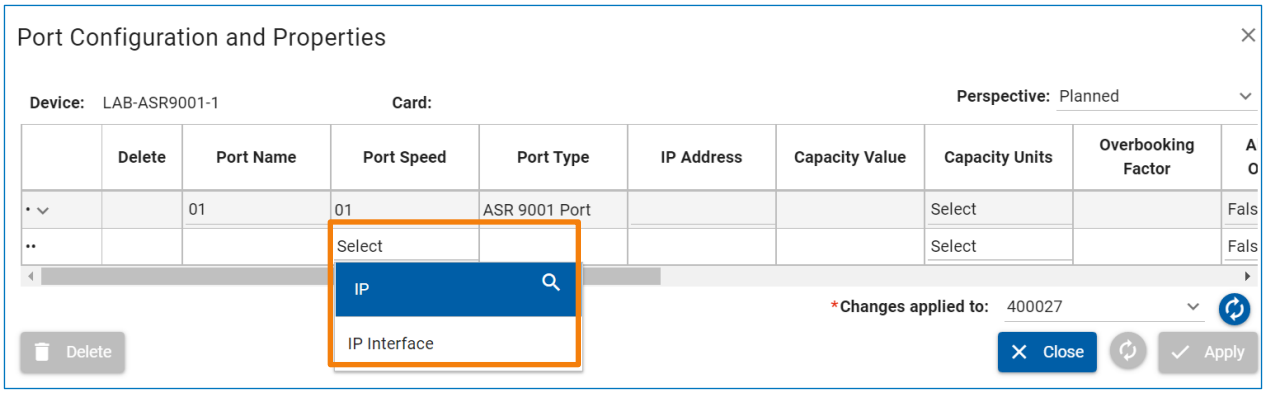
7. Run the query. You should see the message that the relationships and properties were created.
8. Run the query from Step 5 again to retrieve the relationship between archetype instances. Click the **CONSISTS_OF** relationship from one of the ports to the IP Interface node.

```
match (a:Archetype:PhysicalPort{name:"ASR 9001 Port"})<-
   [:HAS_ARCHETYPE]-(ai)
match (ai)-[]->(li:ArchetypeInstance:LogicalInterface)
return ai,li
```

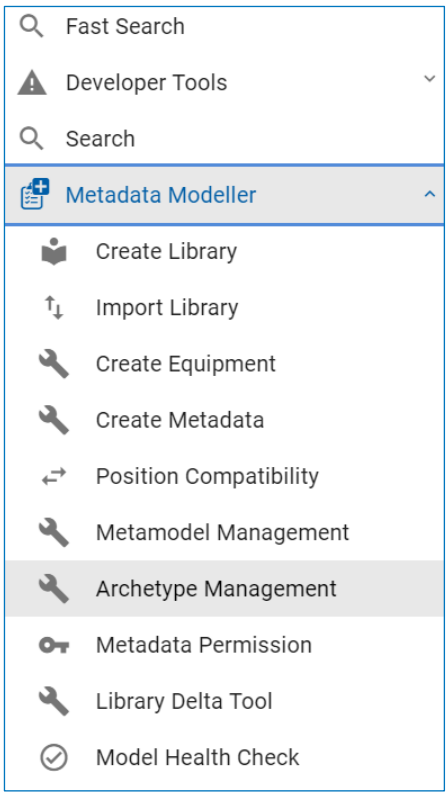


You can see that the properties you provided were generated and set.

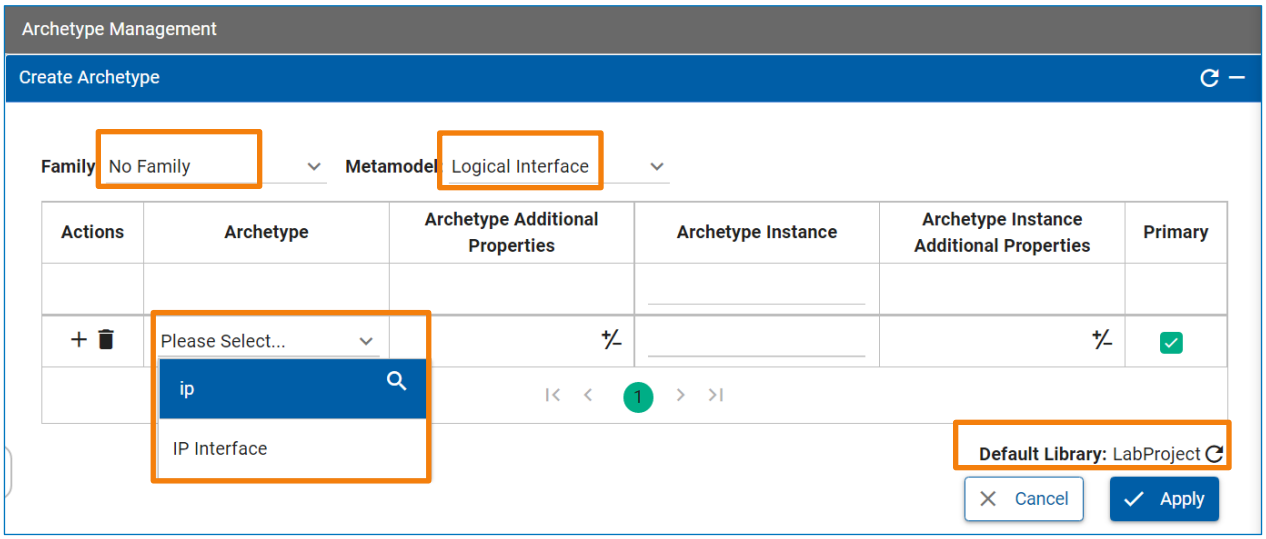
9. Go back to the BPI UI and open the **Port Configuration** page for ASR 9001 port 01 again. Type **IP** in the search field under Port Speed. You are still not able to add the IP Interface logical interface to this port, because there are still compatibility definitions missing between the ASR 9001 Port and the IP Interface Archetypes. You will use the Metadata Modeller to add that relationship.



10. From the main menu, expand **Metadata Modeller** and choose **Archetype Management**.

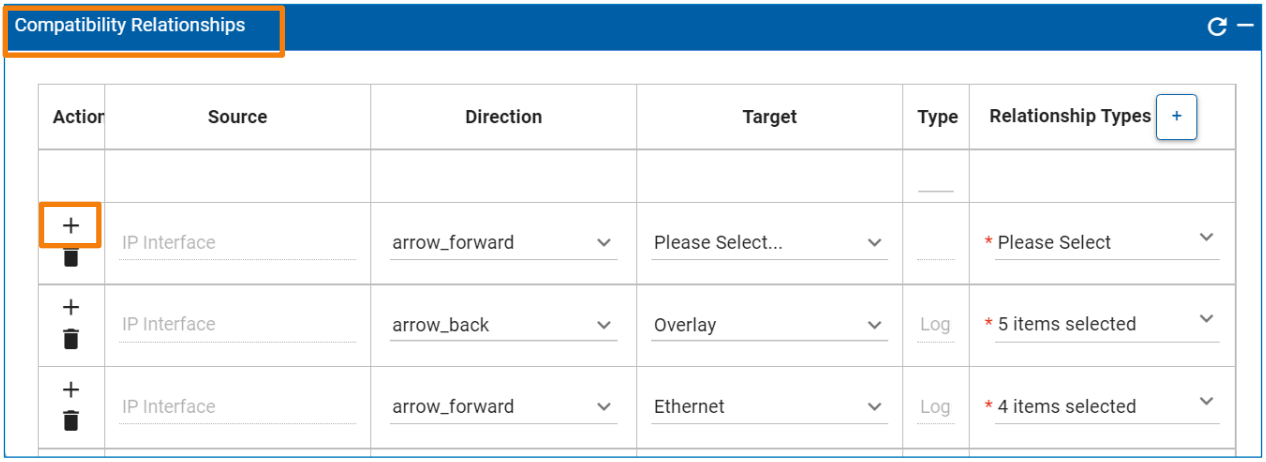


11. On the Archetype Management page, for the Family select **No Family**, and for the Metamodel select **Logical Interface**. Type **IP** in the search field of the Archetype column and choose **IP Interface** from the list. Also verify that the Default Library is set to **LabProject**.



Once you select the **IP Interface** Archetype, the page will get populated with the Archetype properties and relationships. You can create a new relationship in the **Compatibility Relationships** panel just below.

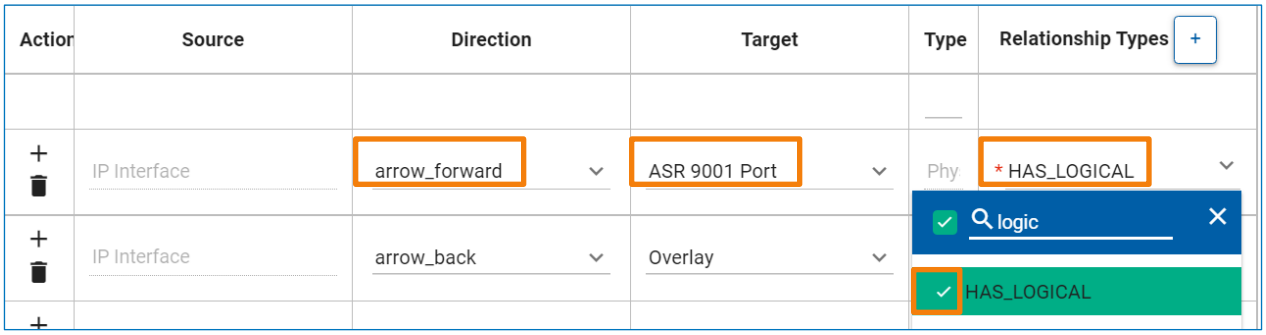
12. In the Compatibility Relationships panel, click the **+** icon in the first row. A new row will show up on the top. This is where you can configure a new relationship.



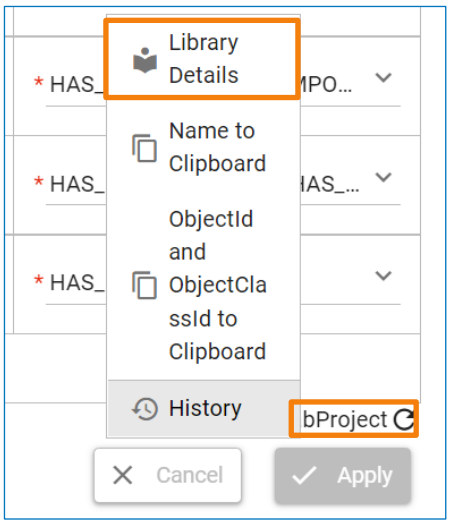
The configuration follows the concept of relationships in the graph database. You can set the direction of the relationship and the node (target) that you want to create the relationship to. You can also set the supported relationship types, which will be set as the values of the familiar **relationshipTypeList** property of the new relationship.

NOTE: You are not setting the actual relationship type of this new relationship. You are now creating compatibilities between Archetypes, so the relationship type will be set to IS_COMPATIBLE.

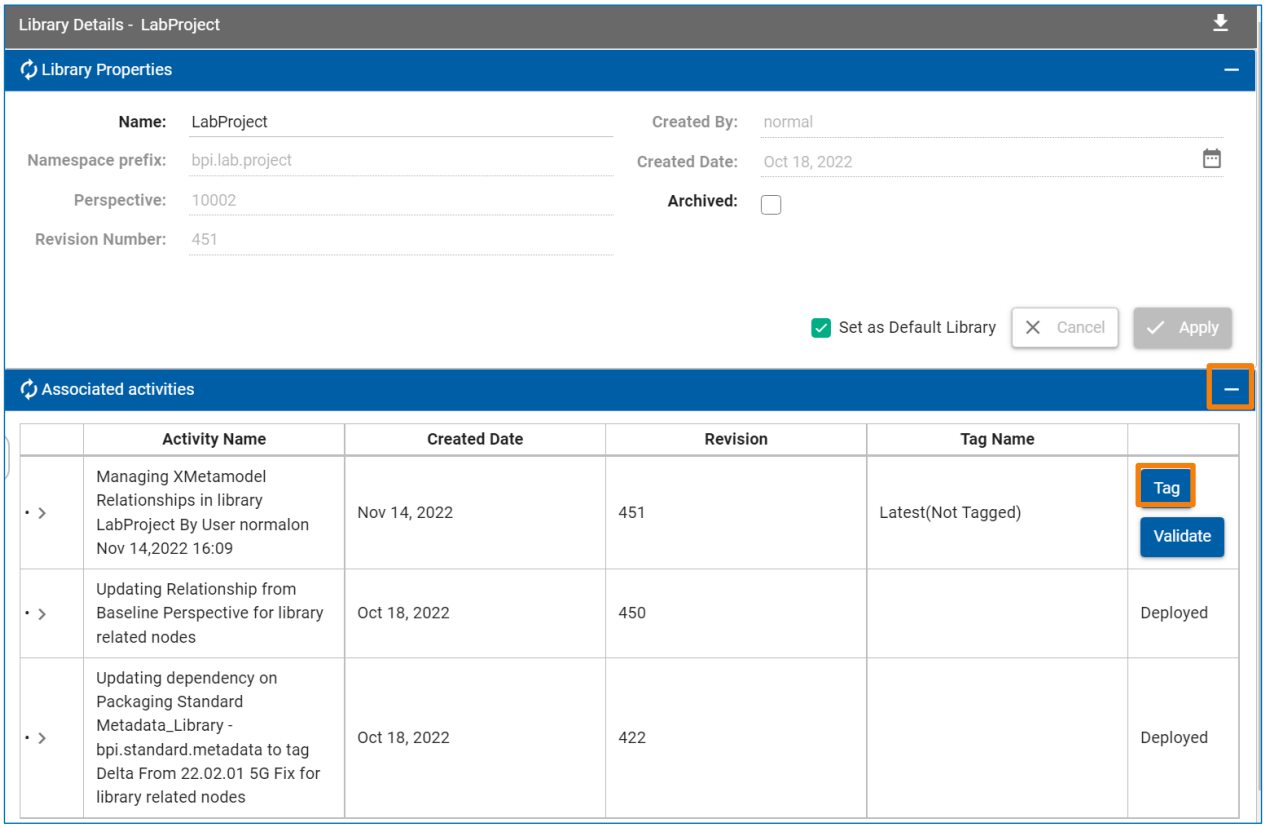
13. Set the following parameter values:
    a. Direction: **arrow_forward**
    b. Target: **ASR 9001 Port**
    c. Relationship Types: **HAS_LOGICAL**



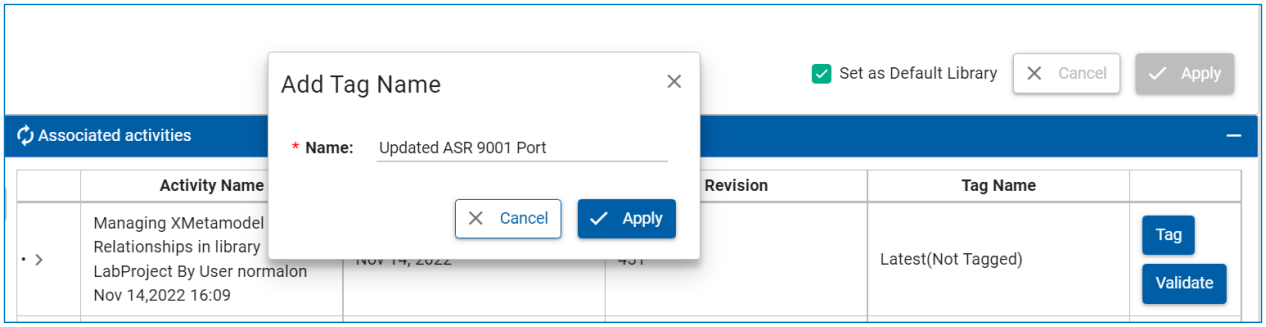
14. Click the **Apply** button at the bottom right of the page to save the changes to the LabProject library.
15. Now you need to deploy the updated library. Right-click the **LabProject** library above the Apply button at the bottom-right of the page, and choose **Library Details**.



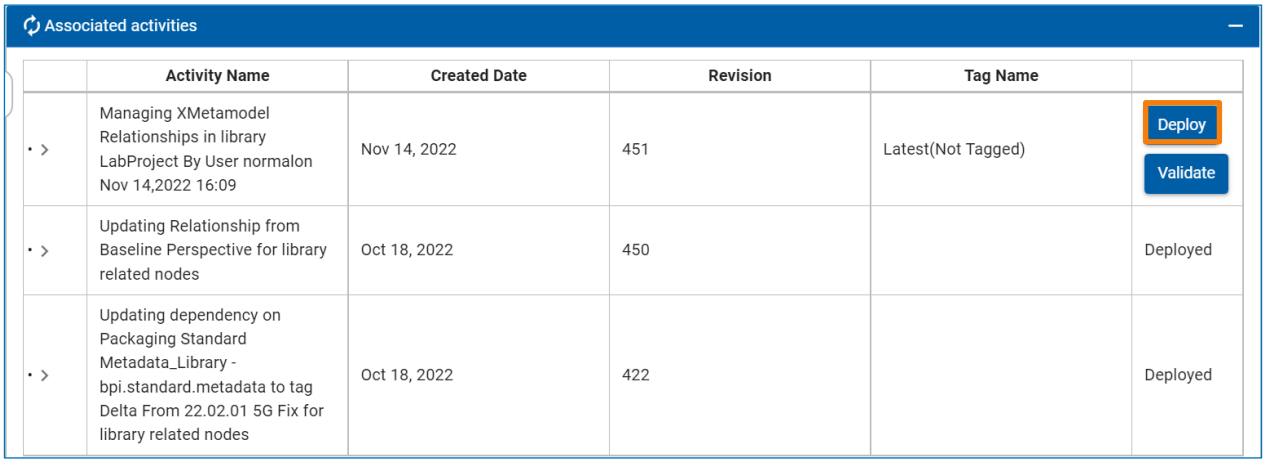
16. Before deploying, you first need to provide a tag for the updated library. On the Library Details page, expand the **Associated activities** panel, and click the **Tag** button.



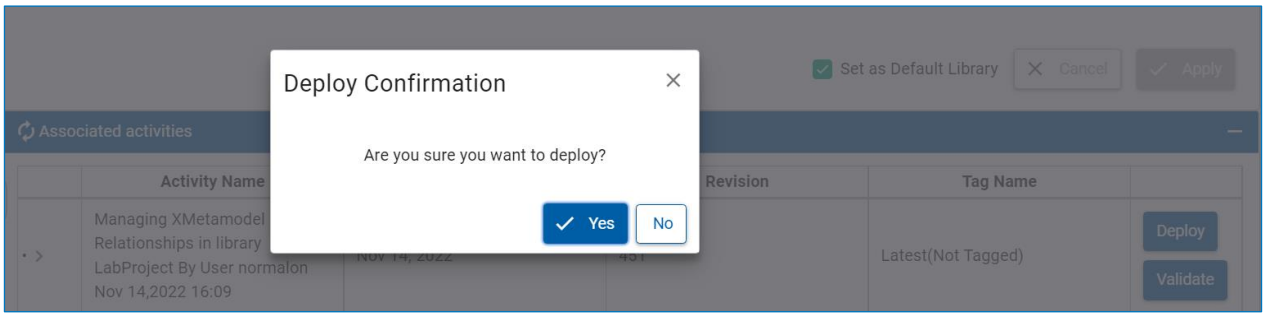
17. Provide the Tag Name **Updated ASR 9001 Port** and press **Apply**.



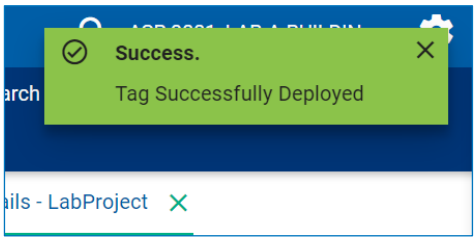
18. Ignore if the Tag Name column still shows Latest(Not Tagged). Click the **Deploy** button which appeared on the left.



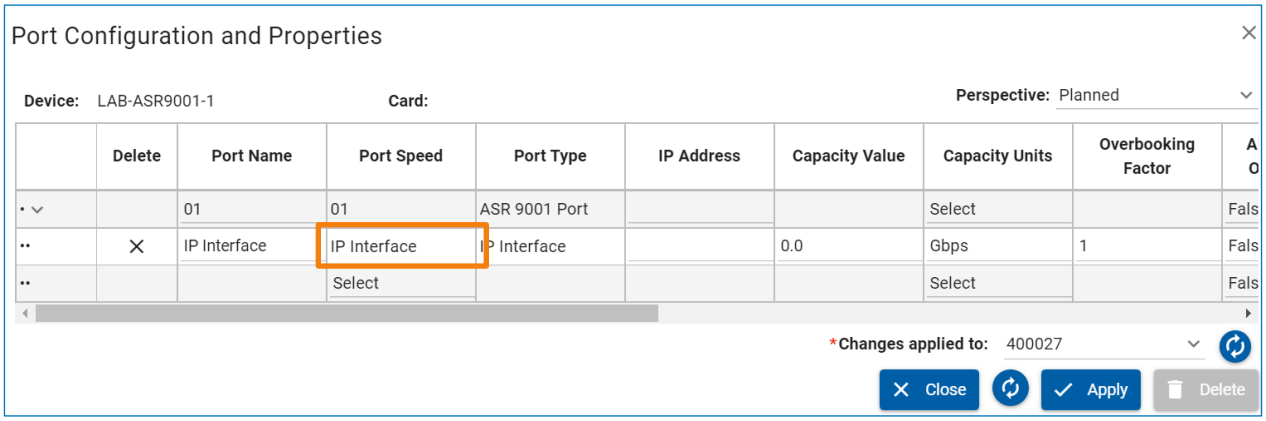
19. Confirm the deployment.



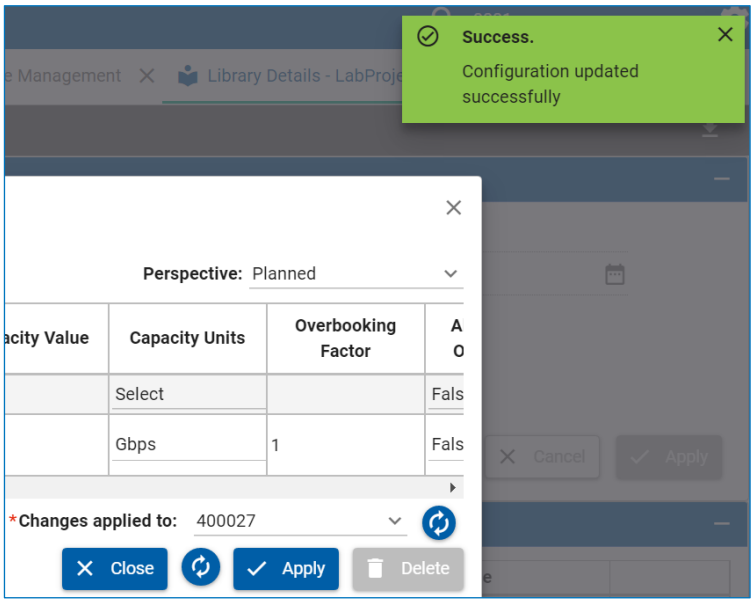
20. After a few minutes, you should see a message in the top-right of the page, notifying you that the updated library was deployed.



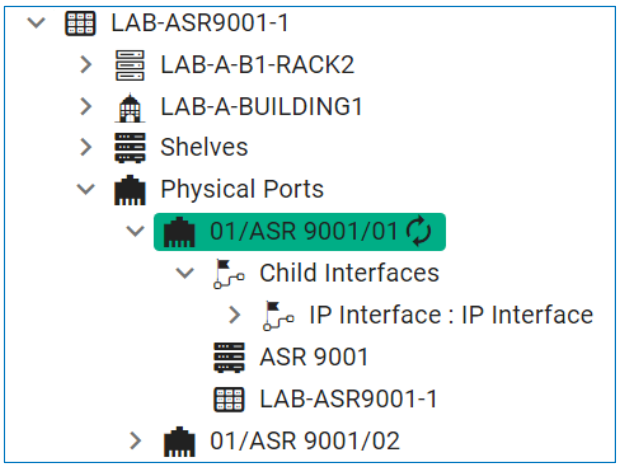
21. Try to configure the ASR 9001 port again. From the network inventory tree, right-click port **01** of your **LAB-ASR9001-1** device and open the **Port Configuration** page. Choose **IP Interface** in the Port Speed column.



22. Choose your order number and click **Apply**. You should see that the configuration was successfully updated because the proper compatibilities are now created.



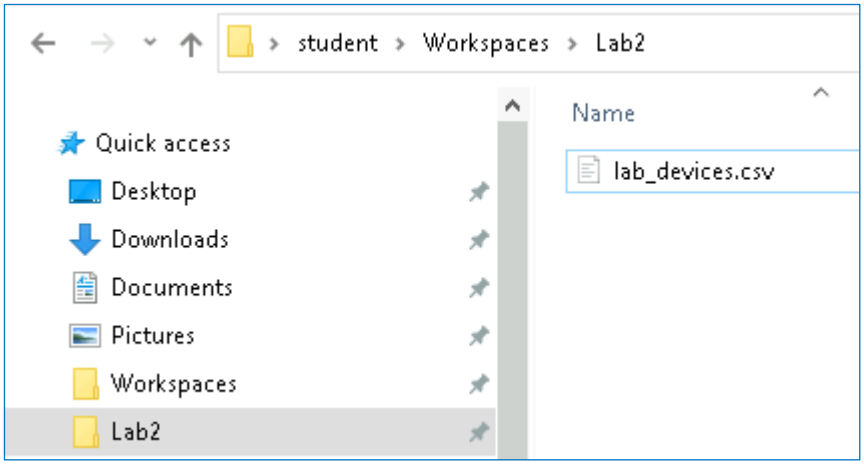
23. You can now see the **IP Interface** logical interface configured on the 01 physical port of your ASR9001 device.



Task 5: Neo4j Data Load

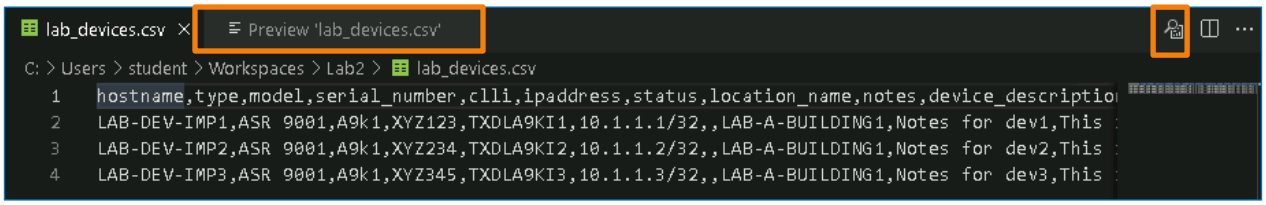
Neo4j supports several methods for importing data from CSV files. In this task, you will use the LOAD CSV Cypher command in the Neo4j Browser to import device data. You can refer to the **Importing CSV** section of the Neo4j Developer online documentation for more details.

1. From your Student PC desktop, open File Explorer and navigate to the **Workspaces > Lab2** folder.



   You can see the lab_devices.csv file, which contains the device data.

2. Double-click this file to open it in Visual Studio Code. The Excel Viewer VSC extension is installed, which can display the CSV file in a table. Click the **Open Preview** icon in the top right to display the **Preview** tab.



   This is the device data that you will import into the Neo4j database. The file contains three devices, with some properties provided, such as hostname, type, and CLLI code.

3. For security reasons, by default Neo4j can only read files from the Neo4j import directory. This is also how your lab environment is set up. You need to copy the CSV file to the Neo4j import directory on the BPI host. From the command prompt, navigate to **C:\Users\student\Workspaces\Lab2** and use **scp** with the **bpadmin** user to copy the lab_devices.csv file to the BPI host.

```
C:\Users\student> cd Workspaces\Lab2

C:\Users\student\Workspaces\Lab2> scp lab_devices.csv bpadmin@bpi-
   host.lab:.

lab_devices.csv
   100%  442    98.1KB/s   00:00
```

4. Connect to the BPI host and copy the file from the home directory to the neo4j pod. From the command prompt, **ssh** to **bpi.host** with the **bpadmin** user.

```
C:\Users\student\Workspaces\Lab2> ssh bpadmin@bpi-host.lab
```

```
Last login: Tue Oct  4 19:39:43 2022 from 10.30.0.94
[bpadmin@bpi-pod01 ~]$
```

5. On the BPI host, issue the **kubectl** command below to copy the CSV file to the Neo4j import directory **/var/lib/neo4j/import** inside the neo4j-0 pod.

```
[bpadmin@bpi-pod01 ~]$ kubectl cp ./lab_devices.csv neo4j-
   0:/var/lib/neo4j/import/
```

6. Verify that the file was copied. Issue the **kubectl** command below to verify the contents of the import dir.

```
[bpadmin@bpi-pod01 ~]$ kubectl exec -it neo4j-0 -- ls -l
   /var/lib/neo4j/import/ | grep lab
-rw-r--r-- 1 root root 442 Nov 14 21:39 lab_devices.csv
```

7. You can now construct a Cypher query which will load the data from this CSV file data and map the column names from the CSV file to the property names of the new Device nodes. In the Neo4j Browser, construct the following query (the query continues on the next page):

```
LOAD CSV WITH HEADERS FROM "file:///lab_devices.csv" AS payload
WITH payload WHERE NOT payload.hostname IS NULL


MATCH (h:Hypermodel) WHERE h.name = "Equipment"
MATCH (m:Metamodel) WHERE m.name = "Device"
MATCH(a:Archetype:Device) WHERE a.name = payload.type
MATCH (abi:ArchetypeInstance:Device) WHERE abi.archetypeId=a.archetypeId
MERGE (d:Device {name :payload.hostname})
ON CREATE SET
    d.drniId = com.blueplanet.ids.createId(),
    d.createdDate = timestamp(),
    d.lastModifiedDate = timestamp(),
    d.deviceName = payload.hostname,
    d.clli = payload.clli,
    d.modelNumber = payload.model,
    d.serialNumber = payload.serial_number,
    d.ipAddress = payload.ipaddress,
    d.status = payload.status,
    d.notes = COALESCE(payload.notes,payload.device_description),
    d.archetypeInstanceId = abi.archetypeInstanceId,
    d.precomputedtypename = abi.name,
    d.isClonedFromDrniId = abi.archetypeInstanceId,
    d.vendor = abi.vendor,
```

```
    d.hypermodelId = h.drniId,
    d.metamodelId = m.drniId,
    d.archetypeId = a.drniId,
    d.p2Latest = TRUE,
    d.p3Latest = TRUE,
    d.latest= [2,3]
ON MATCH SET
    d.lastModifiedDate= timestamp();
```

```
LOAD CSV WITH HEADERS FROM "file:///lab_devices.csv" AS payload
WITH payload WHERE NOT payload.hostname IS NULL

MATCH (h:Hypermodel) WHERE h.name = "Equipment"
MATCH (m:Metamodel) WHERE m.name = "Device"
MATCH(a:Archetype:Device) WHERE a.name = payload.type
MATCH (ai:ArchetypeInstance:Device) WHERE ai.archetypeId=a.archetypeId
MERGE (d:Device {name :payload.hostname})
ON CREATE SET
    d.drniId = com.blueplanet.ids.createId(),
    d.createdDate = timestamp(),
    d.lastModifiedDate = timestamp(),
    d.deviceName = payload.hostname,
    d.clli = payload.clli,
    d.modelNumber = payload.model,
    d.serialNumber = payload.serial_number,
    d.ipAddress = payload.ipaddress,
    d.status = payload.status,
    d.notes = COALESCE(payload.notes,payload.device_description),
    d.archetypeInstanceId = ai.archetypeInstanceId,
    d.precomputedtypename = ai.name,
    d.isClonedFromDrniId = ai.archetypeInstanceId,
    d.vendor = ai.vendor,
    d.hypermodelId = h.drniId,
    d.metamodelId = m.drniId,
    d.archetypeId = a.drniId,
    d.p2Latest = TRUE,
    d.p3Latest = TRUE,
    d.latest= [2,3]
ON MATCH SET
    d.lastModifiedDate= timestamp();
```

Load the file from the import folder and populate the payload variable.

The CSV file contains the Device Archetype name in the type column.

Generate the drnild.

Set various device properties, e.g. deviceName property is set to the value in hostname column.

COALESCE() returns the first non- null value in the given list of expressions.

Set the parent model drnild properties.

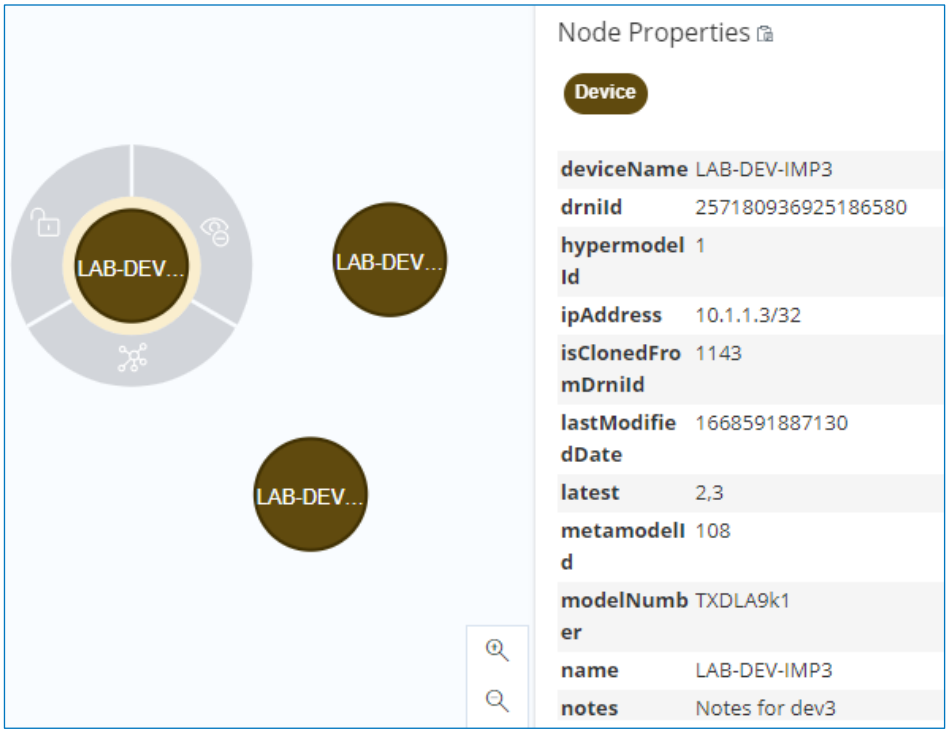
Set the Planned and Live perspectives.

If the device already exists, just update the lastModifiedDate.

8. Run the query. You should see a message that 3 nodes, 3 labels, and multiple relationships were created.

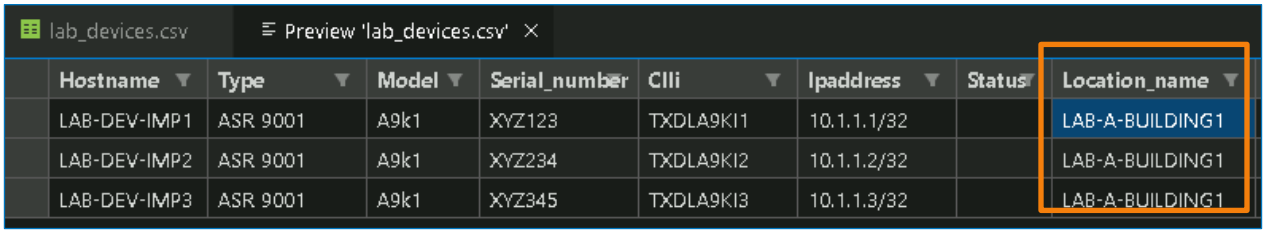
9. Because all the imported devices share a common name prefix, you can run the simple query below to return the newly created nodes. In other environments, you could also set a specific perspective value in the LOAD CSV query for the imported nodes, to distinguish them from other inventory objects.

```
match (d:Device) where d.name starts with "LAB-DEV-IMP" return d
```



You can see the created devices and the properties that you set with the LOAD CSV query. Notice that the new device nodes do not have any relationships yet.

10. The lab_device.csv file also contains information about the name of the building, where the device is located. In VS Code, verify the **location_name** column.



Hostname	Type	Model	Serial_number	Clli	Ipaddress	Status	Location_name
LAB-DEV-IMP1	ASR 9001	A9k1	XYZ123	TXDLA9KI1	10.1.1.1/32		LAB-A-BUILDING1
LAB-DEV-IMP2	ASR 9001	A9k1	XYZ234	TXDLA9KI2	10.1.1.2/32		LAB-A-BUILDING1
LAB-DEV-IMP3	ASR 9001	A9k1	XYZ345	TXDLA9KI3	10.1.1.3/32		LAB-A-BUILDING1

All three devices have the location_name set to LAB-A-BUILDING-1, which is the name of an existing building in your lab.

11. In the Neo4j Browser, construct a query that will read the device location names and create the proper relationships, so that the devices will be associated to their location.

```
LOAD CSV WITH HEADERS FROM "file:///lab_devices.csv" AS payload


MATCH (ai:ArchetypeInstance:Location) WHERE ai.name = 'Building'
```

```
MATCH (loc:Location) where loc.archetypeInstanceId =
   ai.archetypeInstanceId and not loc:Model AND loc.name =
   payload.location_name
MATCH (d:Device ) where NOT d:Model and d.name = payload.hostname
with d,loc
MERGE (d)<-[hasLoc:HAS]-(loc)
ON CREATE SET
hasLoc.latest = [2,3],
hasLoc.drniId = com.blueplanet.ids.createId(),
hasLoc.p2Latest = true,
hasLoc.p2Create = timestamp(),
hasLoc.p2CreateActivity = 12345,
hasLoc.p3Latest = true,
hasLoc.p3Create = timestamp(),
hasLoc.p3CreateActivity = 12345;
```

```
LOAD CSV WITH HEADERS FROM "file:///lab_devices.csv" AS payload

MATCH (ai:ArchetypeInstance:Location) WHERE ai.name = "Building"
MATCH (loc:Location) WHERE loc.archetypeInstanceId = ai.archetypeInstanceId
 AND loc.name = payload.location_name
MATCH (d:Device ) WHERE d.name = payload.hostname
WITH d,loc
MERGE (d)<-[hasLoc:HAS]-(loc)
ON CREATE SET
hasLoc.latest = [2,3],
hasLoc.drniId = com.blueplanet.ids.createId(),
hasLoc.p2Latest = true,
hasLoc.p2Create = timestamp(),
hasLoc.p2CreateActivity = 12345,
hasLoc.p3Latest = true,
hasLoc.p3Create = timestamp(),
hasLoc.p3CreateActivity = 12345;
```

Use the Building location type.

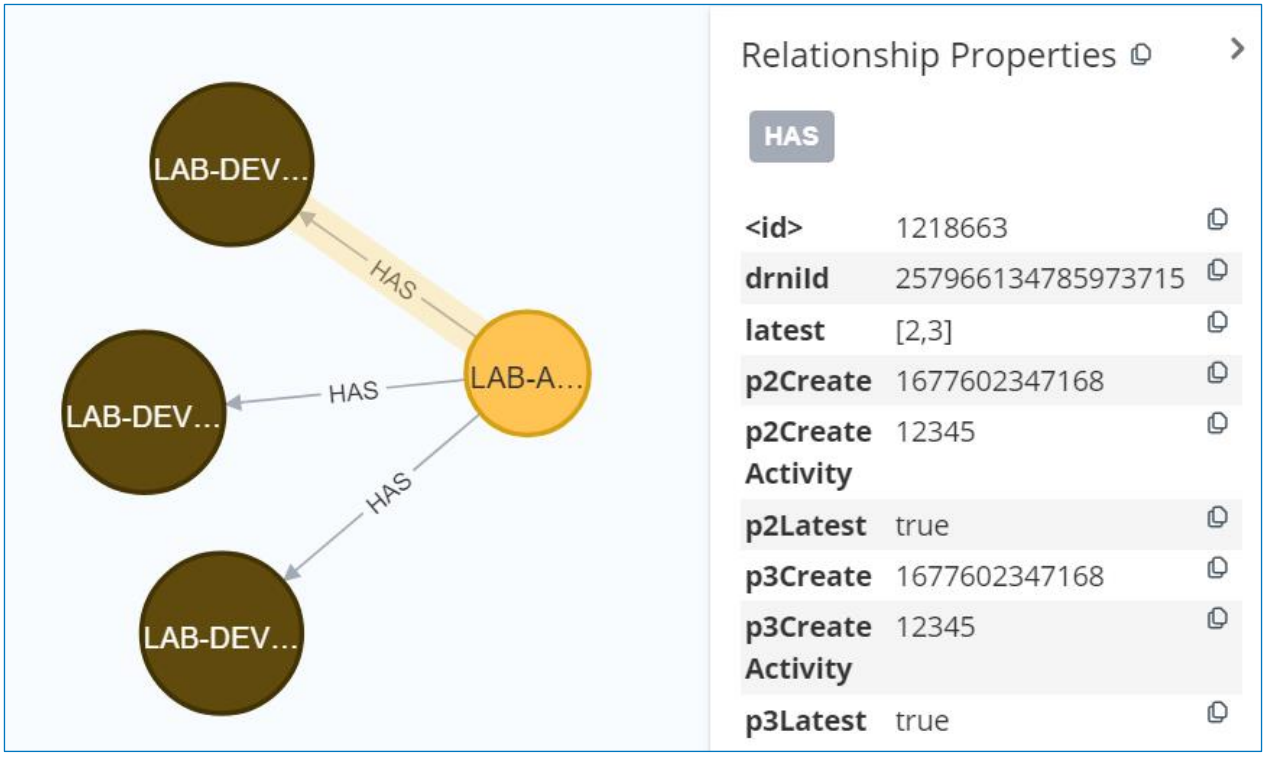
Set the loc variable to a Building location, which name equals the value of the location_name column.

Create the HAS relationship from the location to the device.

Set the relationship properties.

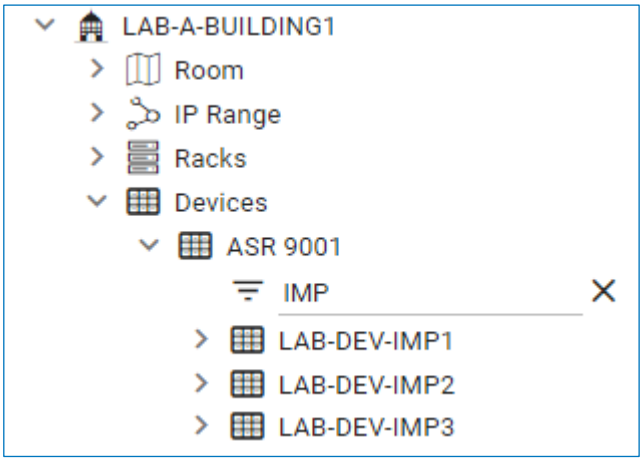
12. Run the query. You should see a message that 3 relationships were created, and the properties were set.
13. Verify the new relationships with the following query:

```
match (d:Device)--(l) where d.name starts with "LAB-DEV-IMP" return d,l
```



The HAS relationship has been created between the LAB-A-BUILDING1 and the imported devices.

14. You can also verify the new devices in the BPI UI. From the Scratch Pad, navigate to **LAB-A-BUILDING-1 > Devices > ASR 9001** and type **IMP** in the filter field.



You can see the three imported ASR 9001 devices.

End of Lab

Lab 4: Advanced Guided Operations

Objectives

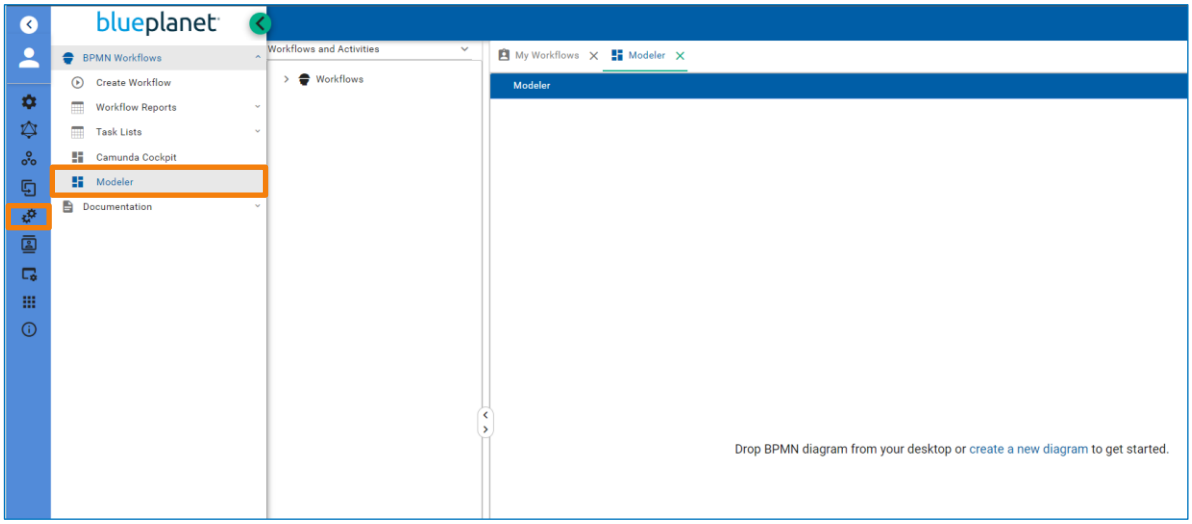
- Learn how to create workflow definitions, containing user and service tasks and their requisite variables and attributes
- Demonstrate how to configure the necessary Angular UI components
- Create the required groovy script code for service tasks
- Create workflow instances and verify their operation
- Use BPMN APIs to create, modify, and query workflows and tasks

Task 1: Create a New Workflow Definition

In this task, you will create a new workflow definition. The goal of the workflow in this exercise is to create a new building in a predefined city, contained in your Blue Planet Inventory database. The workflow will contain two tasks, one user task, and one service (automated) task. The workflow will accept user input in the form of a city name and the name of the building that is to be created. The service task will, in turn, use this information to create the building in the database with the provided parameters.

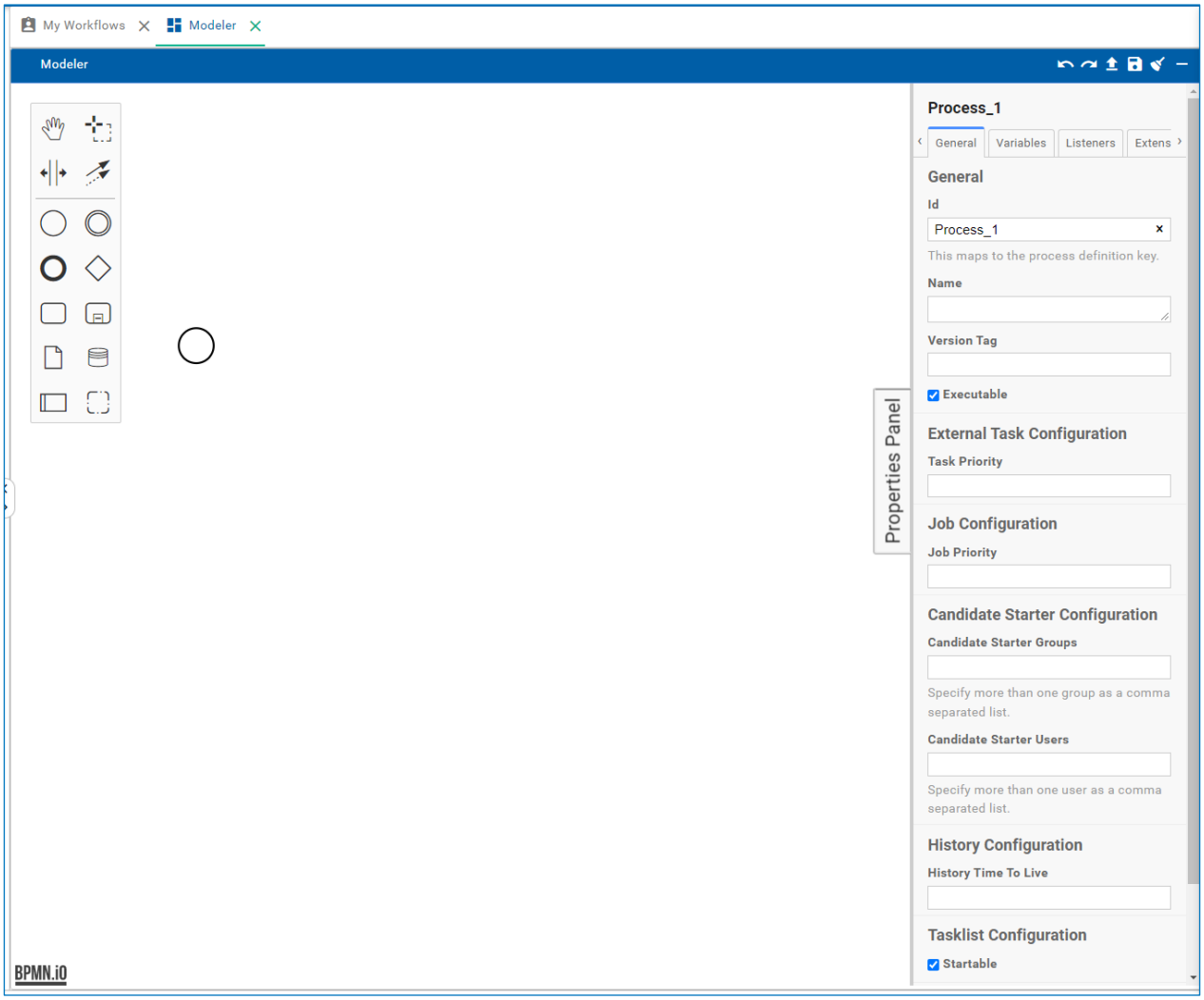
Note that this exercise will demonstrate the ability to design workflows from the Blue Planet UI, however, there are also other valid ways to accomplish this, such as Camunda Modeler and VS Code with the BP BPMN extension.

1. To log in to the Blue Planet Inventory UI, from your Student PC, open a Chrome browser session and go to https://bpi.lab or click on the BPI bookmark. Use your instructor-provided credentials to log in.
2. From the application bar on the left, select **Workflow**, then expand the **BPMN Workflows** menu and select **Modeler**.

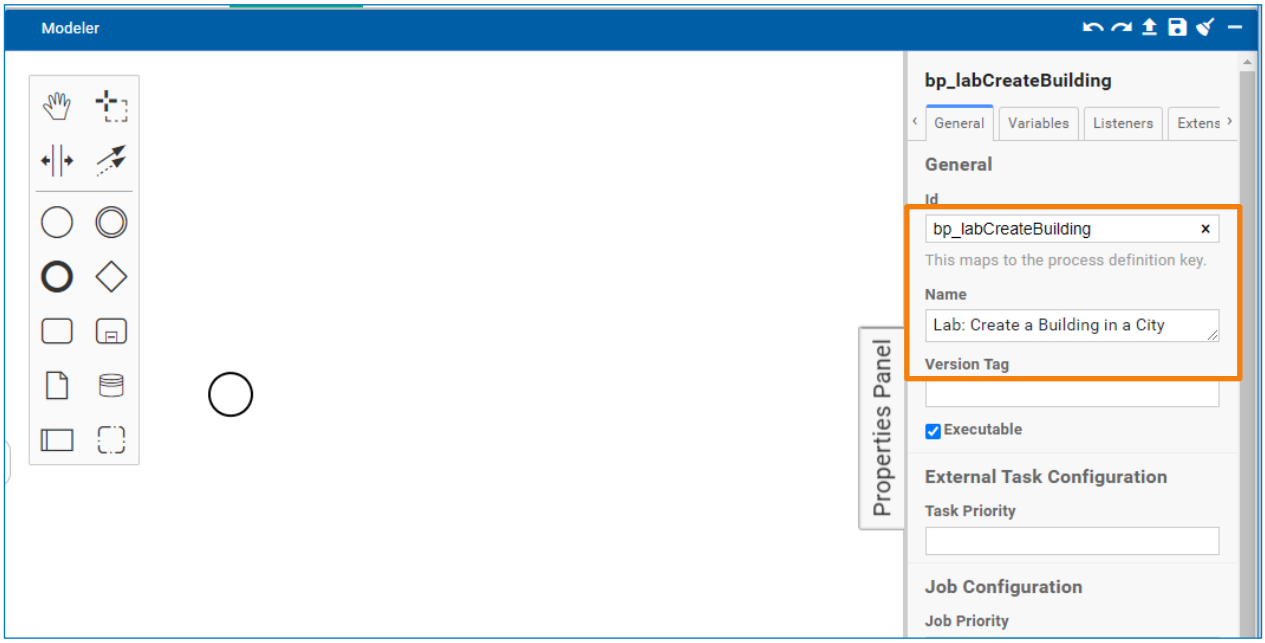


NOTE: A new diagram can be created from scratch by clicking the blue "**create new diagram**" link. Drag and drop functionality is also supported, you can drop your .bpmn files in the empty area to edit and deploy them.

3. Click the **create new diagram** in the Modeler pane on the right to start the modeler UI. The Modeler UI opens. The left side of the window shows the graphical representation of your workflow with the tools panel, and to the right is the **Properties Panel**.

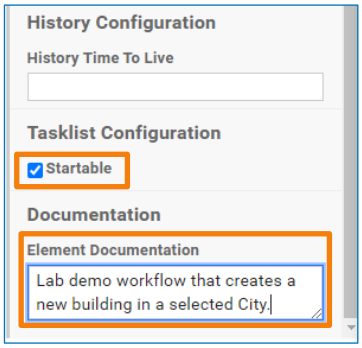


4. Name the workflow. Make sure that no event is selected (click on an empty area in the diagram) and that the Properties Panel now shows text *Process_1* on top. Under Id, replace the existing text with ***bp_labCreateBuilding***. Under Name, enter the following text: **Lab: Create a Building in a City**.

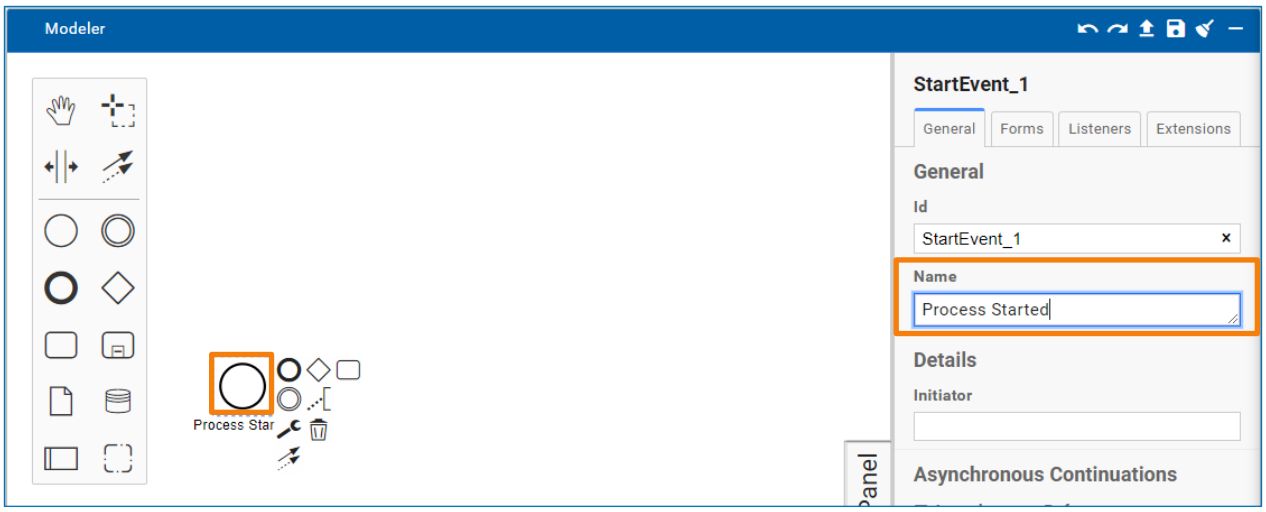


NOTE: The Id of the process now shows as **bp_labCreateBuilding** on top of the panel.

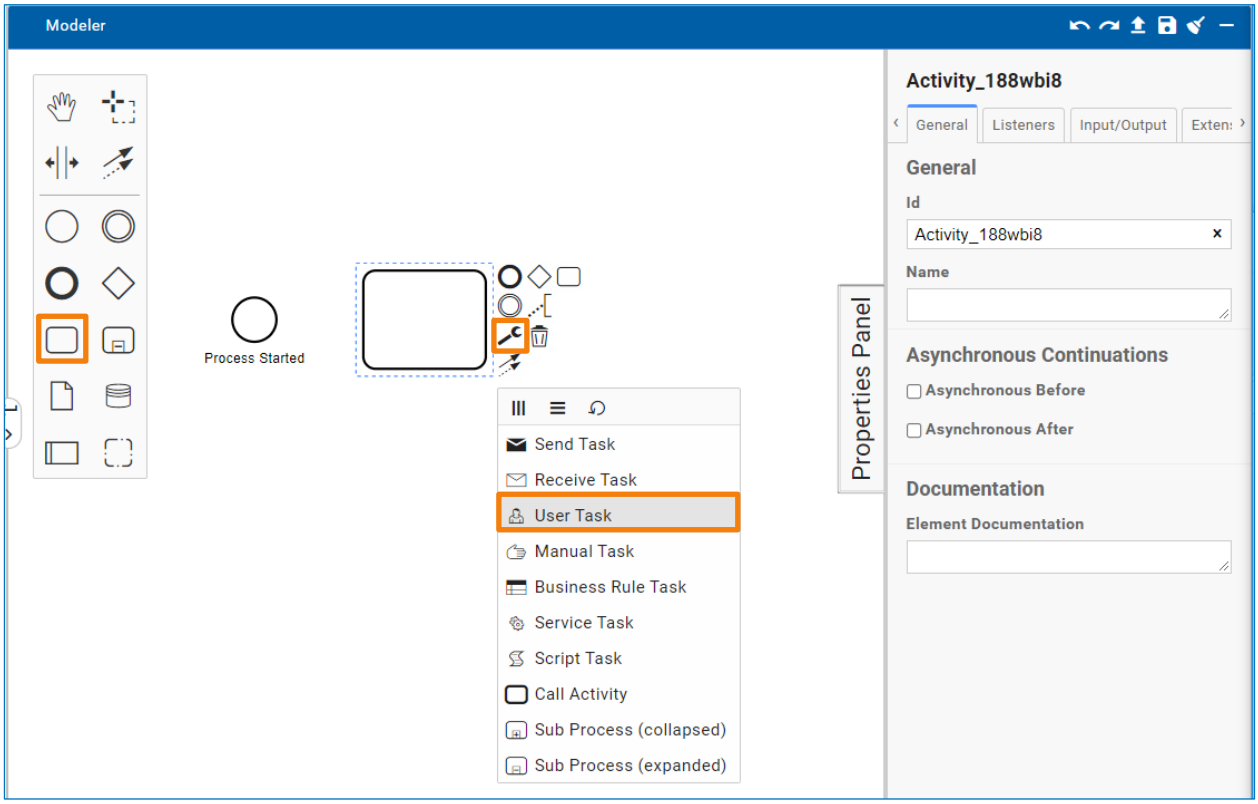
5. At the bottom of the Properties Panel, make sure that the **Startable** checkbox is selected and enter the following text under Element Documentation: **Lab demo workflow that creates a new building in a selected City**. This text will appear in the BPI UI when creating workflow instances.



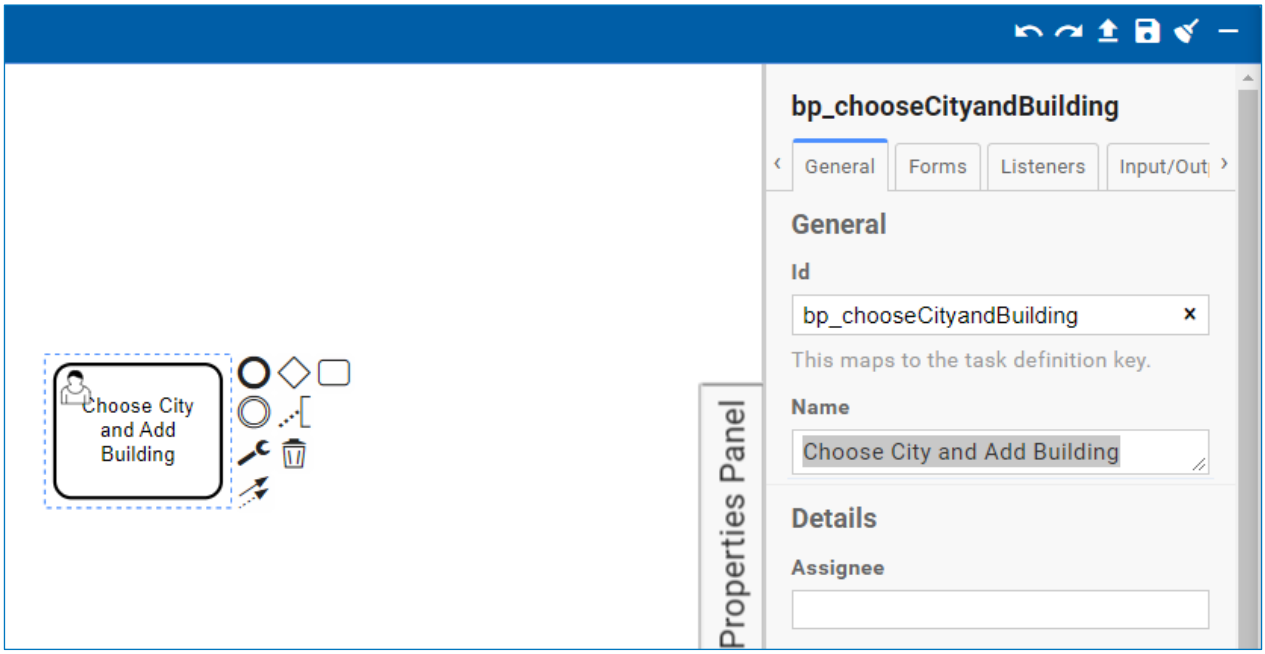
6. Click the **start event** (the circle entity), then in the Properties Panel, enter the text **Process Started** under Name.



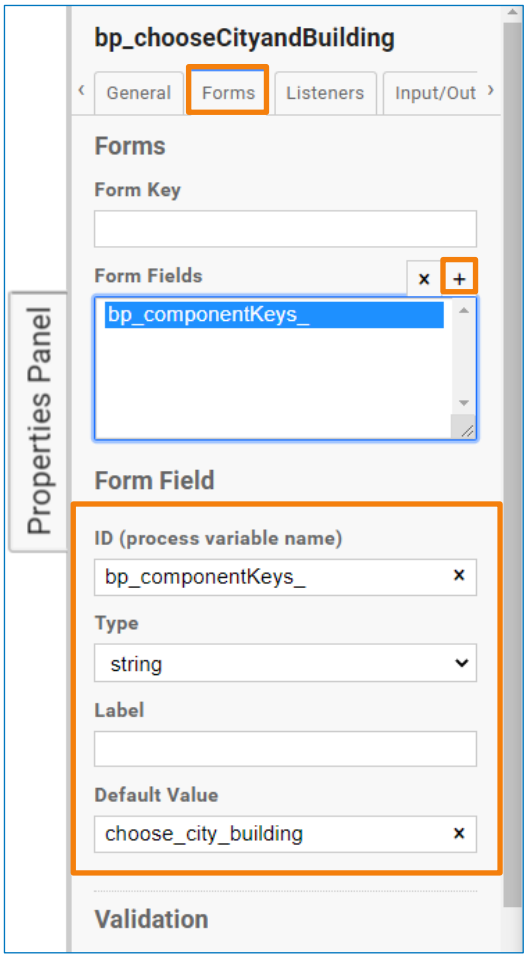
7. From the tools, select **Create Task** and drag a new task to the design area. Next, click the **Change type** (wrench icon next to the task component) and select **User Task** from the context menu.



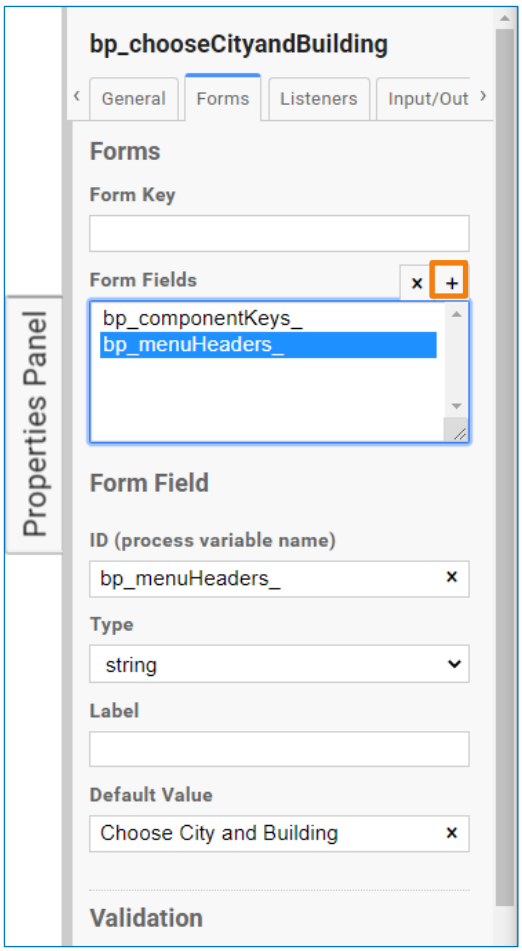
8. In the General tab, fill in the fields:
   a. Id: **bp_chooseCityandBuilding**
   b. Name: **Choose City and Add Building**



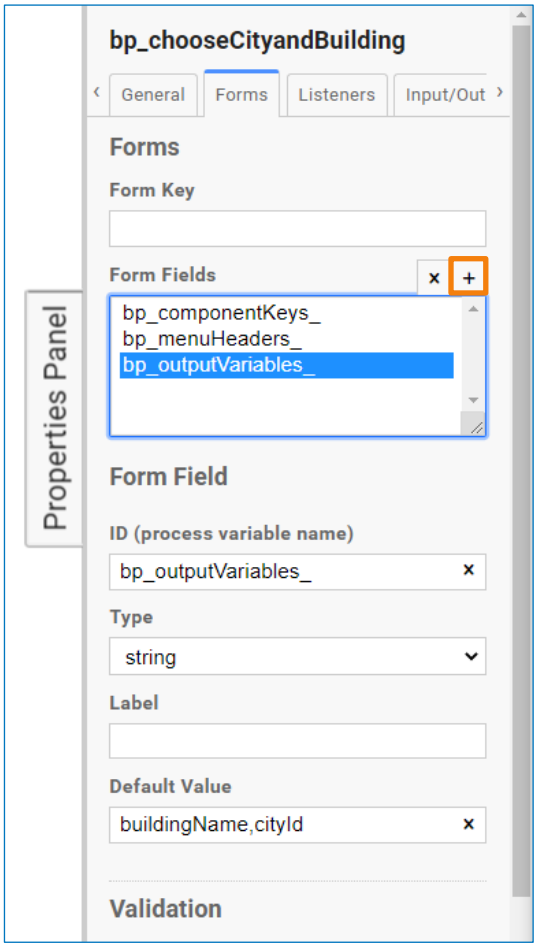
9. Click on the **Forms** tab inside your new user task properties. Add the form fields that are important for the functioning of the workflow. First, click the **+** button in Form Fields and add the **bp_componentKeys_** as the ID. Select **string** in the Type field and the default value of **choose_city_building**. Be careful to enter this value correctly because it must match a part of the UI component code in your project.



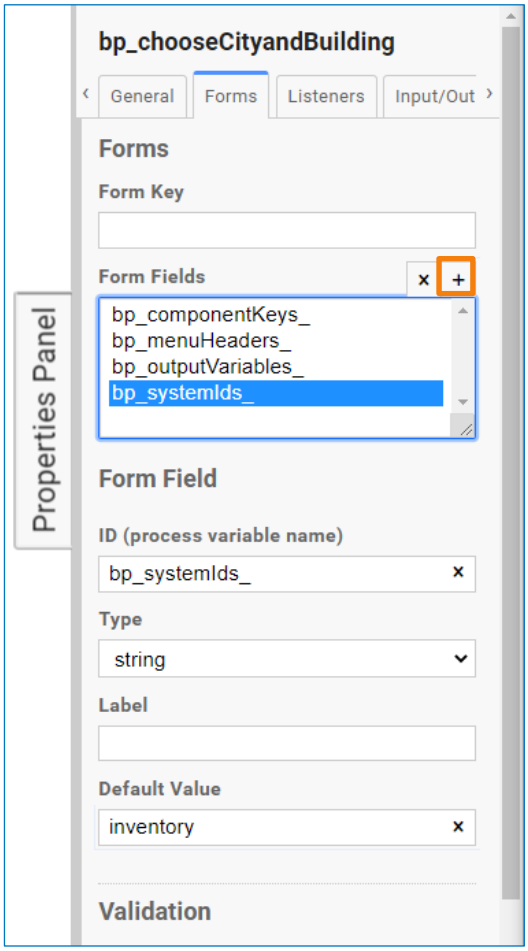
10. Add the menu headers Form Field by clicking the **+** button again. This defines the task menu entry text when you run the task. Enter the following parameters:
    a. ID: **bp_menuHeaders_**
    b. Type: **string**
    c. Default Value: **Choose City and Building**



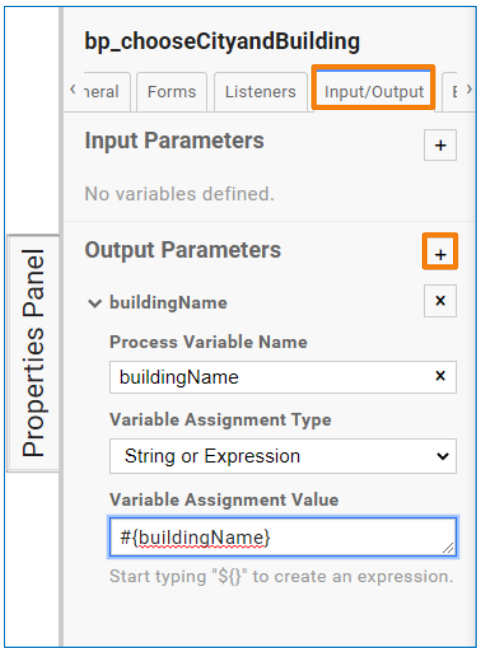
11. Next, add the output variables Form Field by clicking the **+** button again. This defines the names of the output variables that will be used in your workflow to create a new building. Enter the following parameters:
    a. ID: **bp_outputVariables_**
    b. Type: **string**
    c. Default Value: **buildingName,cityId**



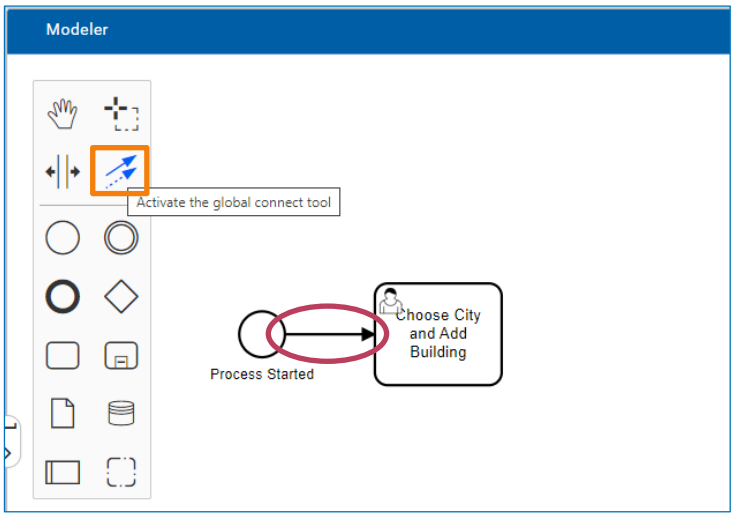
12. Finally, add the systemID Form Field. Click the **+** button and enter the following parameters:
    a. ID: **bp_systemIds_**
    b. Type: **string**
    c. Default Value: **inventory**



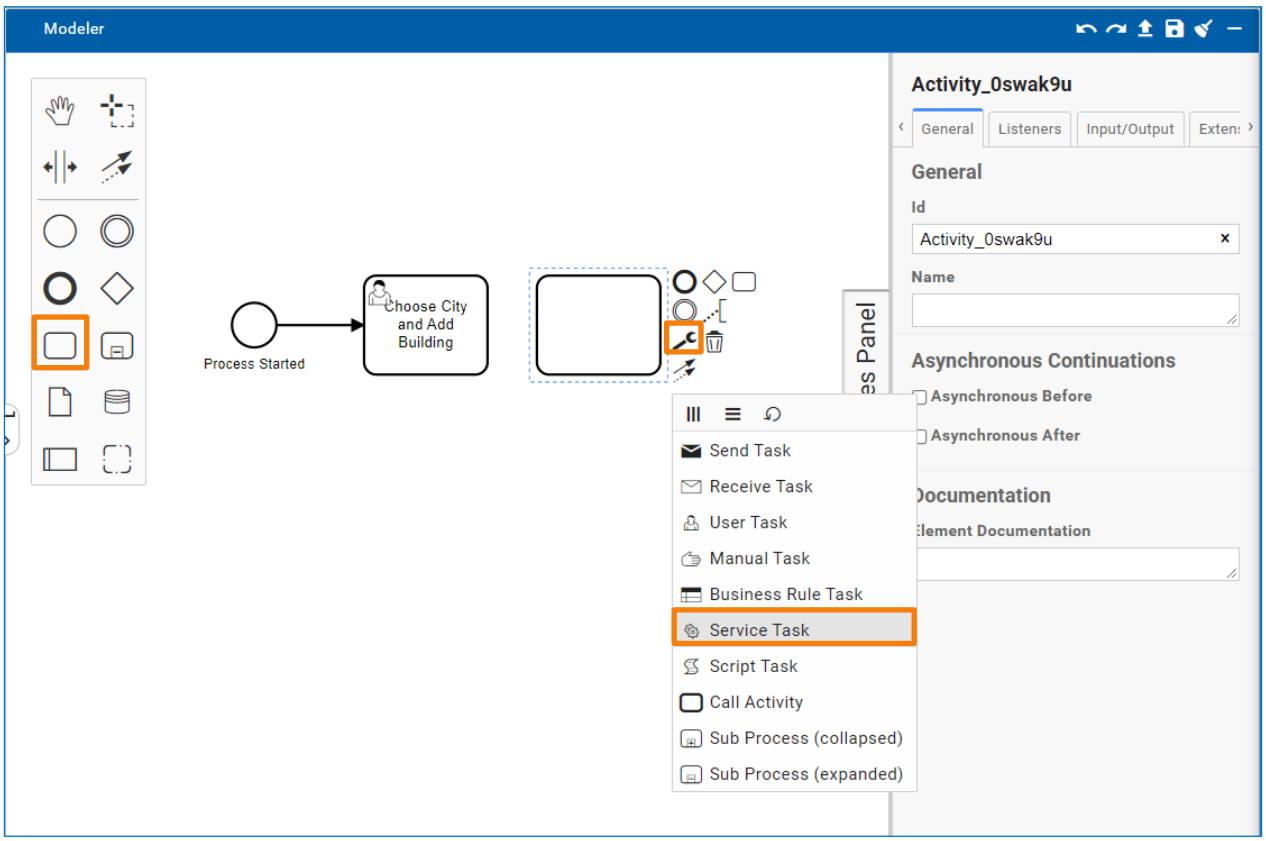
13. In the Properties Panel, click the **Input/Output** tab. Click the **+** button next to the Output Parameters and add a new parameter with the following settings:
    a. Process Variable Name: **buildingName**
    b. Variable Assignment Type: **String or Expression**
    c. Variable Assignment Value: **#{buildingName}**



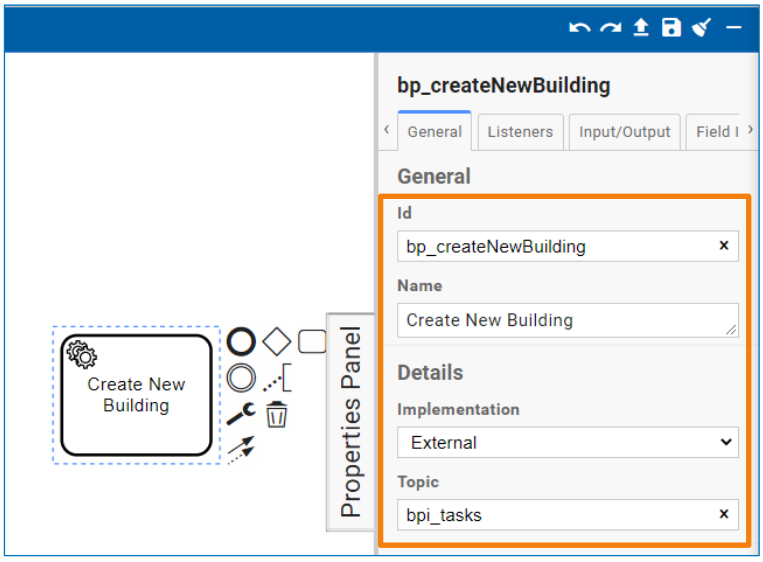
14. Use the **global connection tool** (arrows icon) to connect the two components in your diagram. The resulting connection will show as an arrow between the process start and your user task.



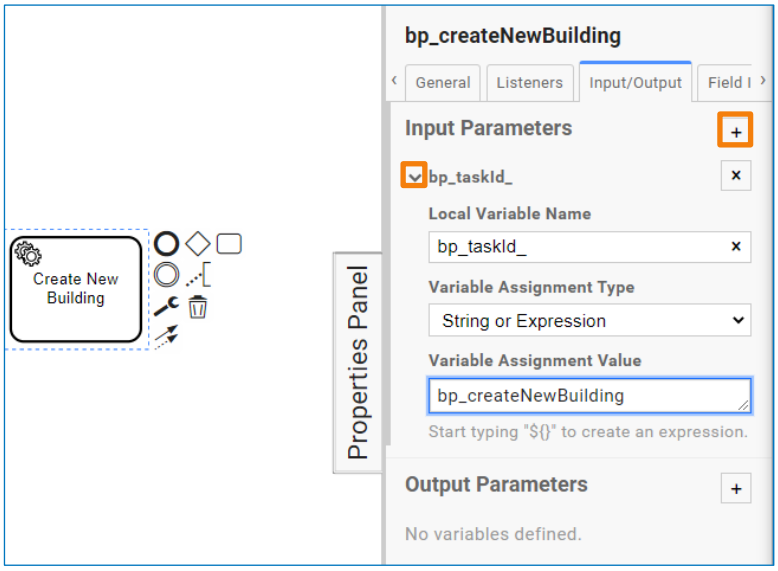
15. Now, configure your service task. This task will trigger the appropriate code in your BPI deployment which, in turn, will create the building with the previously supplied user input. First, add another task to your diagram. Click the **Change type** icon (wrench icon) and change the type to **Service Task**.



16. While the new service task is selected, change the Id to **bp_createNewBuilding**, and Name to **Create New Building**. Under Details, for Implementation select **External**, and under Topic specify **bpi_tasks**.

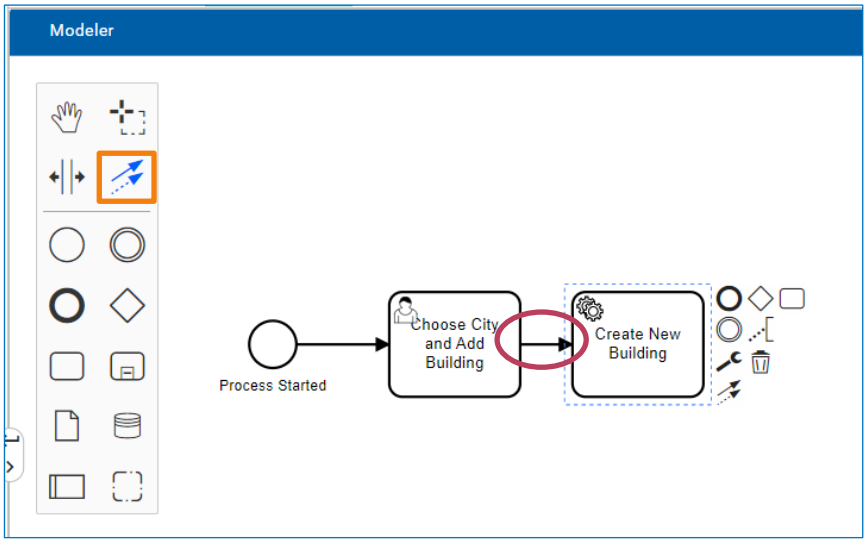


17. While the new service task is selected, change the tab to **Input/Output**. Click the **+** button next to Input Parameters to add a new parameter. Expand the new parameter and modify it to reflect the following:
    a. Local Variable Name: **bp_taskId_**
    b. Variable Assignment Type: **String or Expression**
    c. Variable Assignment Value: **bp_createNewBuilding**

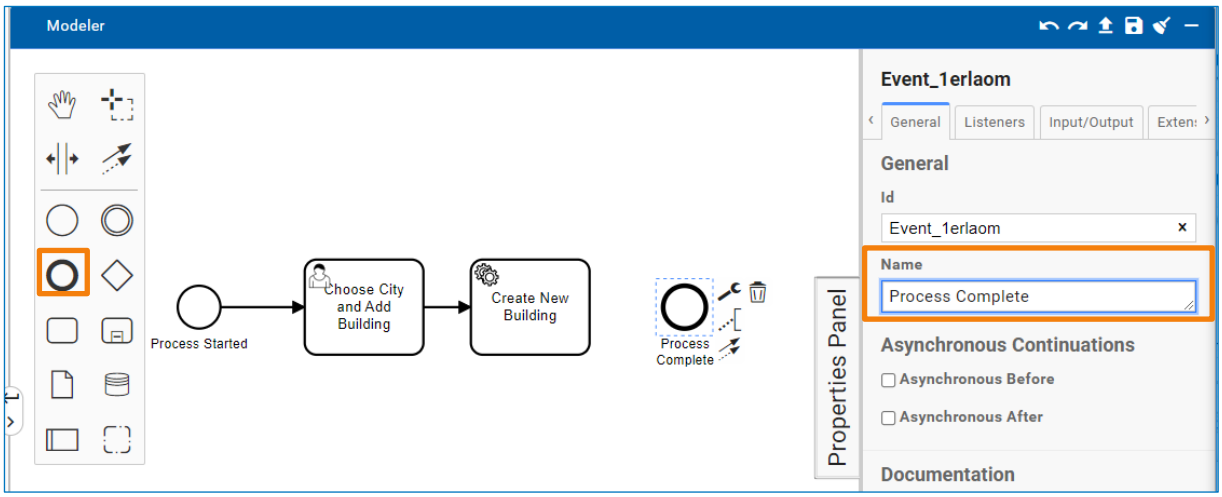


NOTE: Be careful to enter the information precisely, with regards to the case sensitivity.

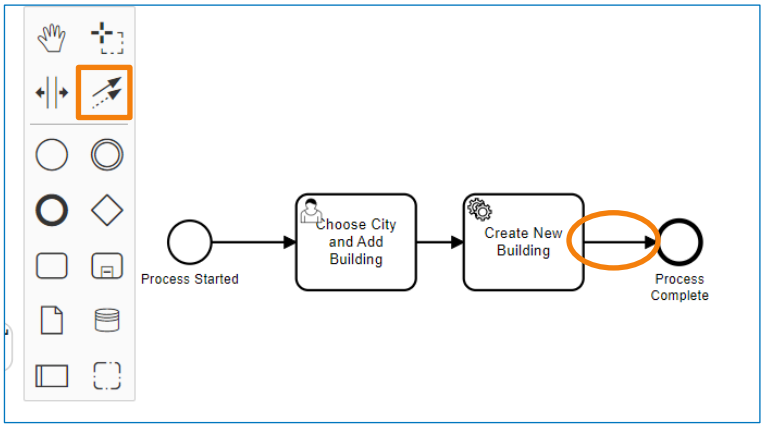
18. Use the **global connection tool** (arrows icon) to connect the user task and the service task.



19. In the tools widget, click the **Create EndEvent** button (bold circle) to create the end event for your workflow definition. Name the event **Process Complete**.



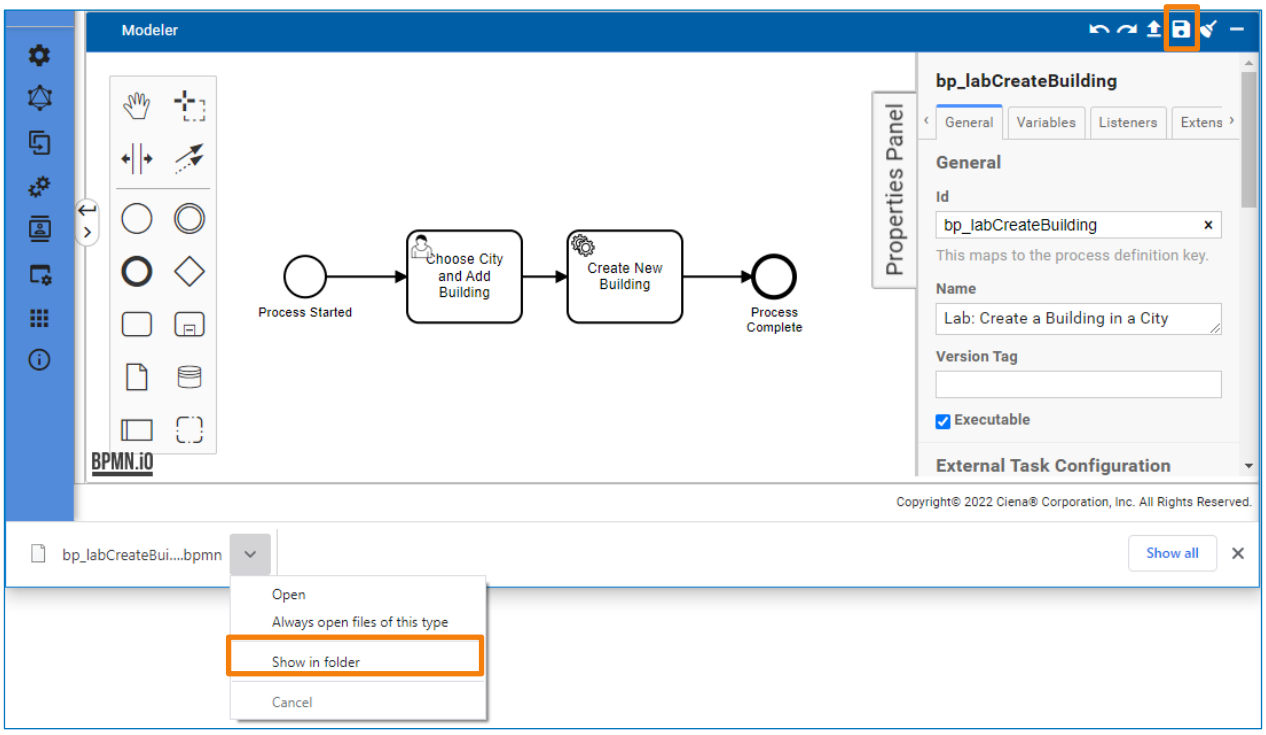
20. Finally, connect the service task to the end event with an arrow. Your new workflow definition is now complete.



Task 2: Save and Deploy the Workflow Definition

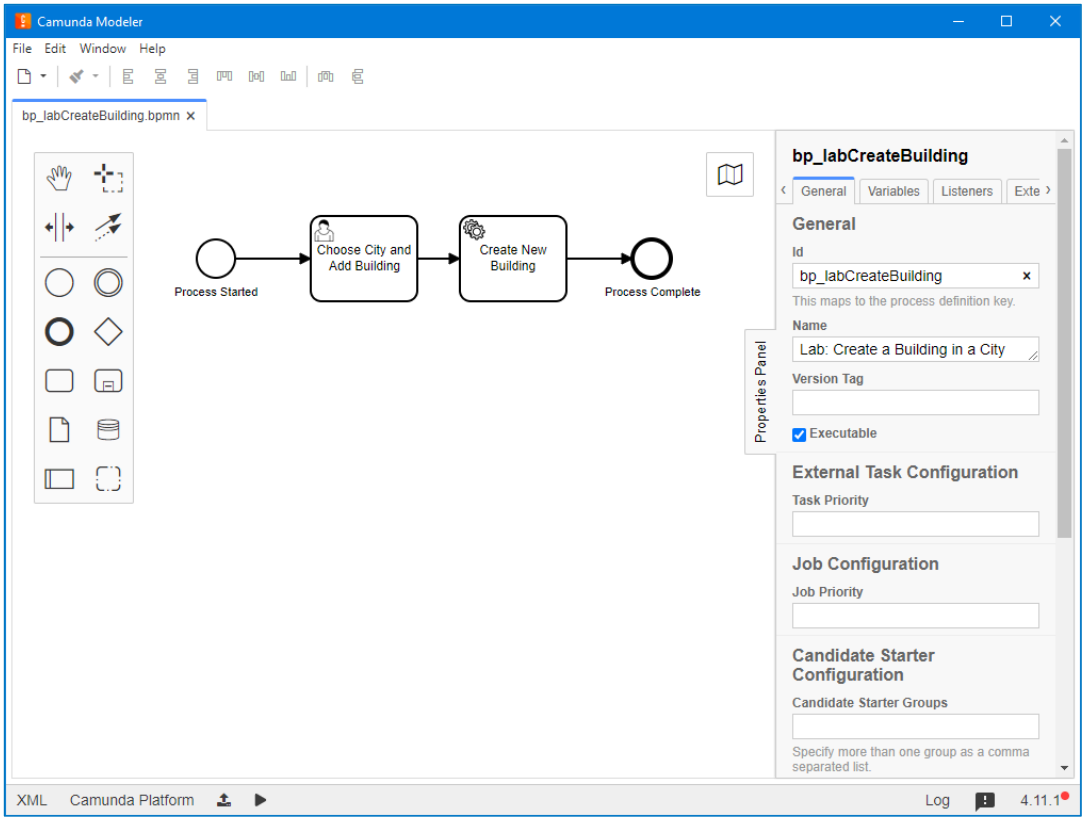
In this task, you will save your workflow definition locally and later proceed to deploy it to the server. Saving locally allows you to store the workflow for additional modifications later. You will also learn how to use the Camunda Modeler to open and modify the workflow definition. Designing workflows in Camunda Modeler is equivalent to designing them in the BPI UI, however, Camunda Modeler offers you more flexibility in terms of which servers to deploy to and managing multiple files at the same time. It also allows you to modify the XML code of the bpmn file directly.

1. To save the workflow to your local PC, click the **Save** icon in the top right corner of your Modeler page. The file will be saved under the process name that you configured in the previous task, **bp_labCreateBuilding** and it will have the **.bpmn** extension.

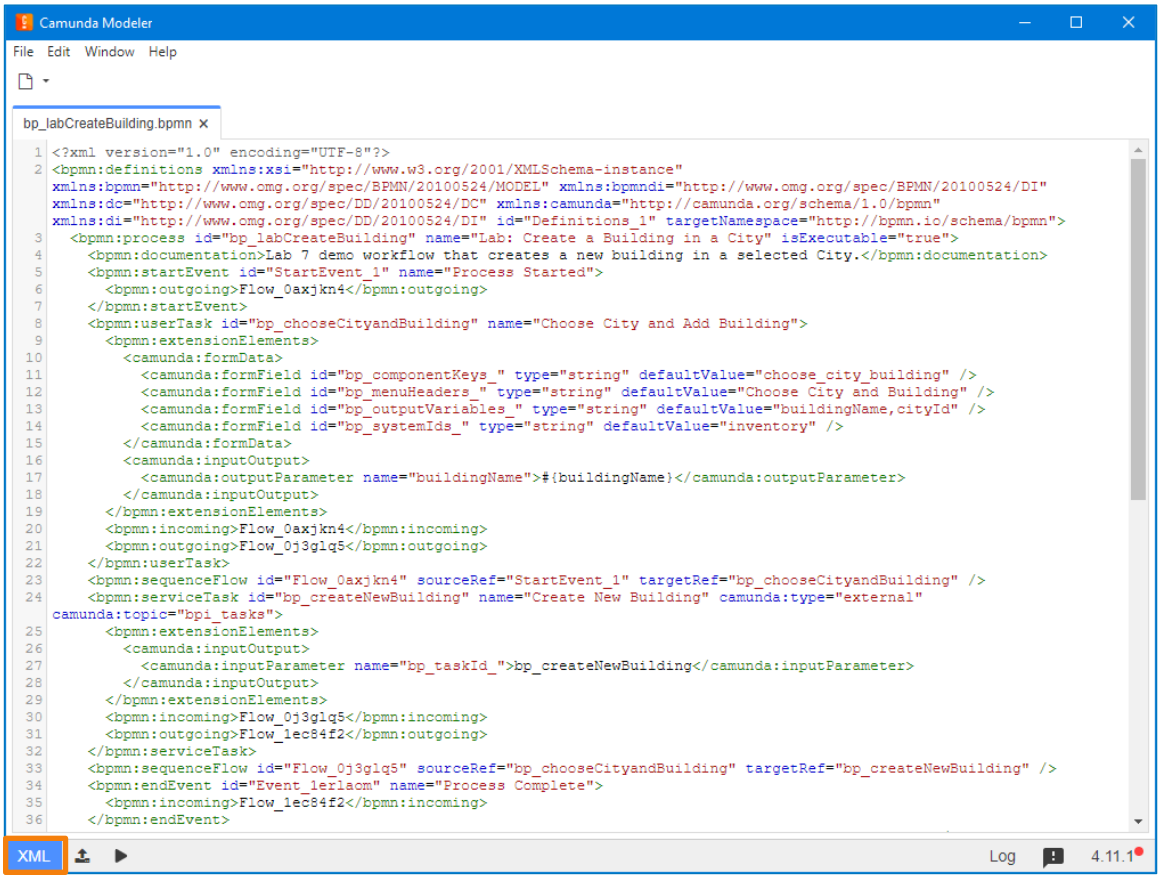


NOTE: Find out where your PC saved the file. You can find out the location by right clicking the file in your browser and selecting the **Show in folder** option.

2. From your desktop, open the **Camunda Modeler** standalone application and open the file **bp_labCreateBuilding.bpmn** by selecting the **File > Open File...** menu option.

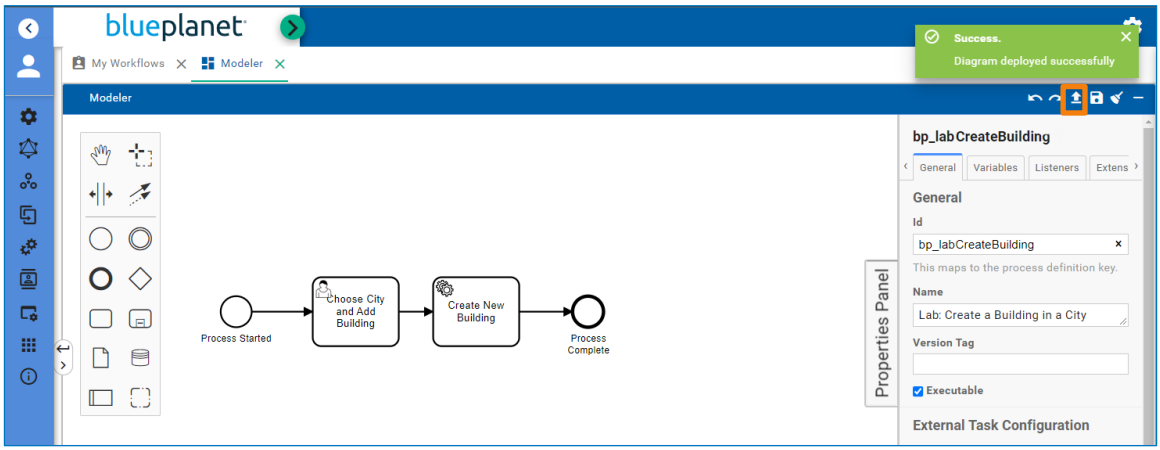


3. Click the **XML** button in the bottom left corner to display the XML representation of your workflow definition. The Camunda Modeler can be used to create BPMN workflows by directly editing the XML code.



NOTE: You can open the .bpmn files in regular text editors, like VS Code, and do your XML edits from there. VS Code gives you an additional advantage of being able to graphically design your workflows, just like you would do in the BPI UI and Camunda Modeler. For this, you must install the Blue Planet BPMN extension for VS Code.

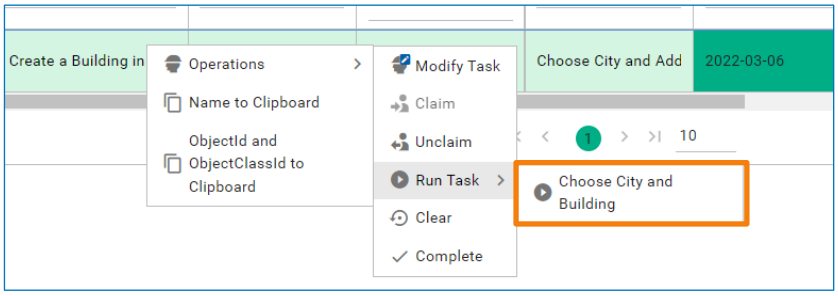
4. Now return to the BPI UI in your browser and click the **Deploy** button (top right corner) to deploy your new workflow to the server. The **Diagram deployed successfully** message will appear.



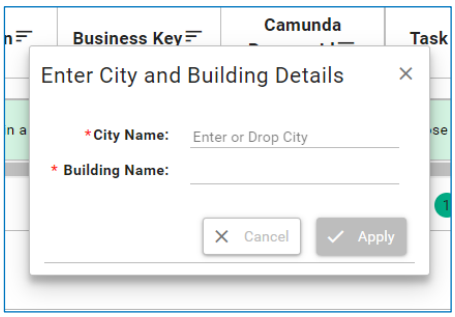
Task 3: Examine the Code for UI Angular Components

In this task, you will examine the UI code changes required to implement the user task in this exercise. Note that this task is demonstrational only and no action is required on your part. The appropriate code is already built into your lab pod server.

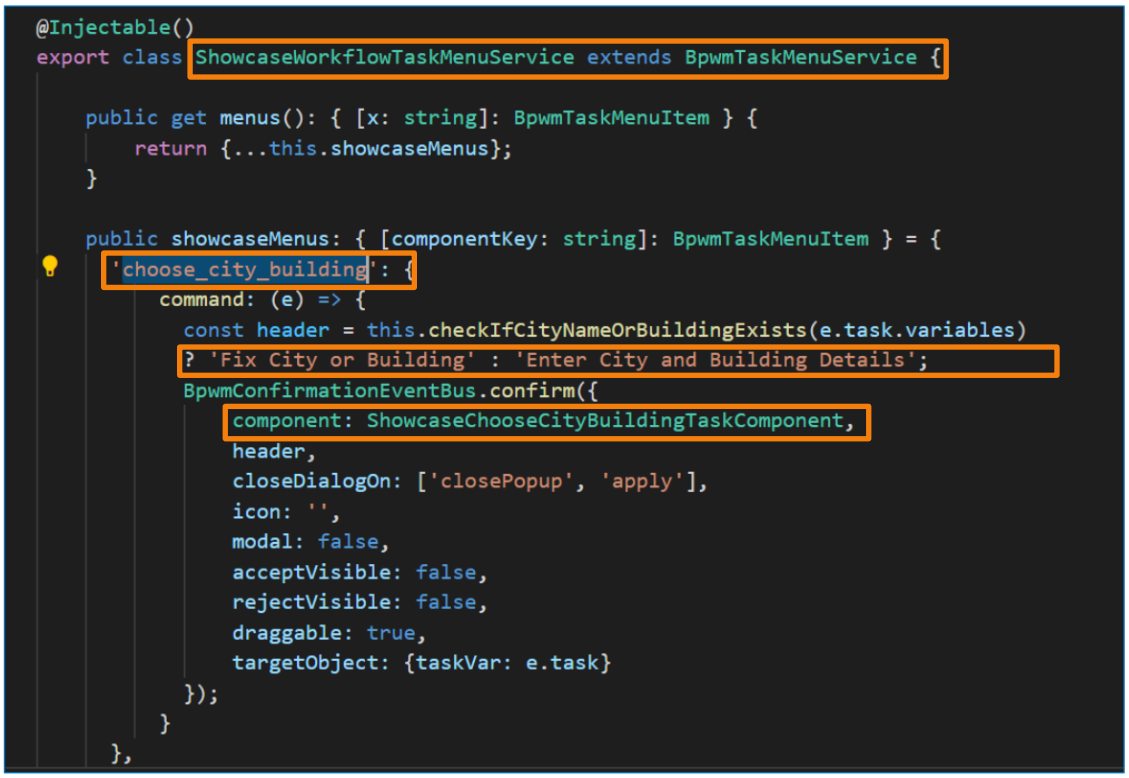
1. When running the user task from this exercise, the menu text that is displayed is referenced in the workflow definition form field **bp_menuHeaders_** of the user task with the value **Choose City and Building**. This can be seen in the resulting UI rendering as follows:



2. When a user clicks the menu item, a UI form appears on the screen. The Angular component for your user task must be created in the project for the user to be able to input the data. The component code will typically be in the common *bpmn* task components part of your project, depending on your deployment, the file naming and location can vary. In this exercise, the file is located at ***blueplanet-inventoryui\src\main\frontend\projects\blueplanet\inventory-ext\src\components\common-bpmn\task\add-deviceto-building\showcase-choose-city-building-task.component.ts.*** The following figure demonstrates the look and feel of the component.



3. A service named **ShowcaseWorkflowTaskMenuService** is used to store the metadata mapping for loading a user task component. This metadata will be stored against the **bp_componentKeys**_ key in the user task form field (in this case '**choose_city_building**') which you defined in the previous task as part of the workflow definition. The file for this code is usually named *<prefix>-task-menu-service-.ts*. For example: ***src\components\common-bpmn\service\showcase-task-menu.service.ts***



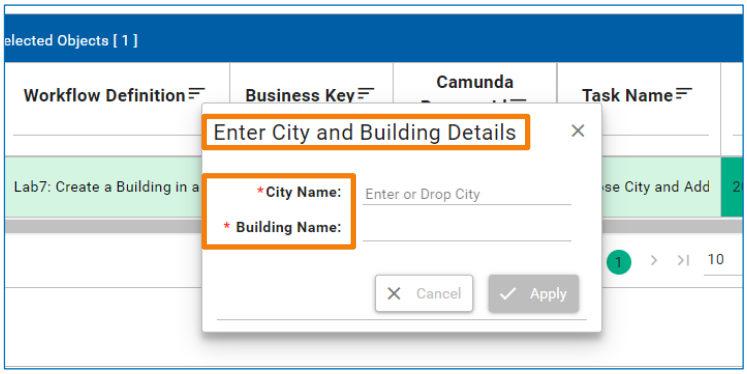
```
@Injectable()
export class ShowcaseWorkflowTaskMenuService extends BpwmTaskMenuService {

    public get menus(): { [x: string]: BpwmTaskMenuItem } {
        return {...this.showcaseMenus};
    }

    public showcaseMenus: { [componentKey: string]: BpwmTaskMenuItem } = {
      'choose_city_building': {
          command: (e) => {
            const header = this.checkIfCityNameOrBuildingExists(e.task.variables)
            ? 'Fix City or Building' : 'Enter City and Building Details';
            BpwmConfirmationEventBus.confirm({
                component: ShowcaseChooseCityBuildingTaskComponent,
                header,
                closeDialogOn: ['closePopup', 'apply'],
                icon: '',
                modal: false,
                acceptVisible: false,
                rejectVisible: false,
                draggable: true,
                targetObject: {taskVar: e.task}
            });
          }
      },
```

Note that your workflow task menu service (in this example, **ShowcaseWorkflowTaskMenuService** must extend the **BpwmTaskMenuService** class.

The service ShowcaseWorkflowTaskMenuService must be registered in the provider's array of the module (ShowcaseCommonBpmnModule) against the token**: BpwmTaskMenuService**

```
1  providers: [
2    { provide: BpwmTaskMenuService, useClass: ShowcaseWorkflowTaskMenuService
3  } ]
```

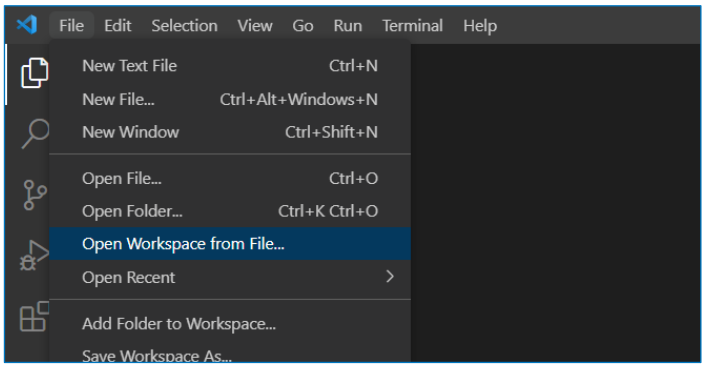
4. The constant **header** from the previous code excerpt determines the caption of the UI popup window as seen in the following figure (**Enter City and Building Details**). You can change this text in the code as per your requirements. Also, note the variables labels City Name and Building Name. Those variable labels are defined in another file:
   a. **ShowcaseWorkflowInputFormComponent** that extends
   b. **BpwmAbstractWorkflowInputComponent** class



Task 4: Create and Deploy the Service Task Groovy Script

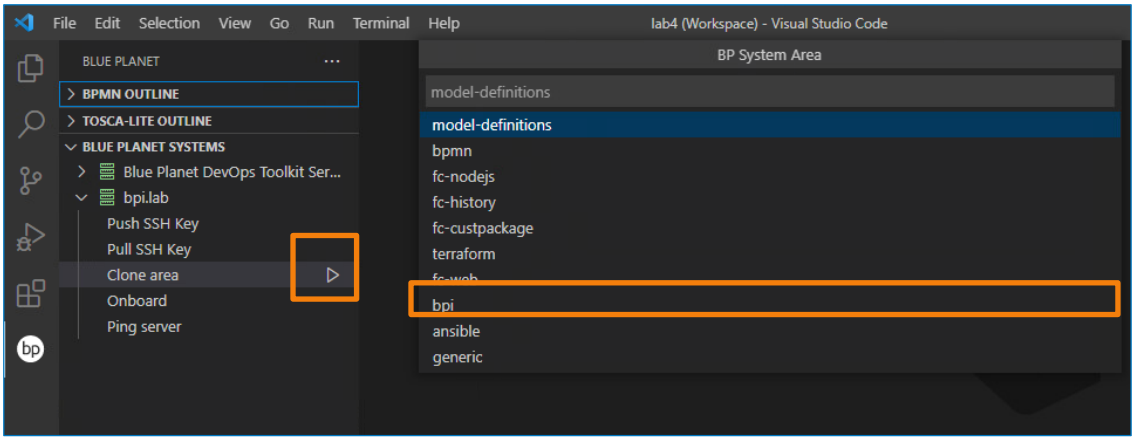
In this task, you will create a service task script that will create a building in a defined city, and then you will onboard the code to your BPI server instance. You will use the BPI extensions in Visual Studio Code (VSC) to achieve the objective of this lab exercise. The BPI extensions must be installed via the VSC extensions feature, however, in your lab pod they come pre-installed and ready for use.

1. Open VSC from your desktop by clicking the VSC icon.
2. From the main menu, select **File > Open Workspace from File…** to open the workspace for this lab.

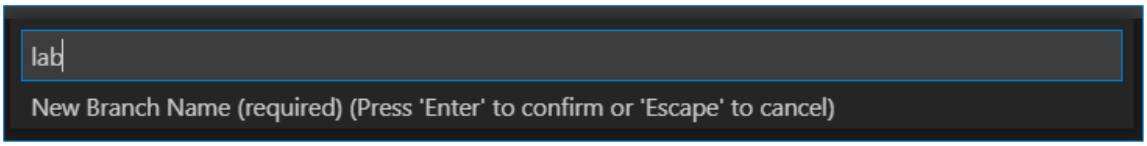


NOTE: Close all other VSC instances if open.

3. In the file browser, find the workspace file **C:\Users\student\Workspaces\Lab4\lab4.code-workspace** and click on **Open** to select it.
4. From BP extension, clone the directory structure from the server to your VSC workspace. Click on the **play icon** next to **Clone area** and then select **bpi** when asked for BP System Area.

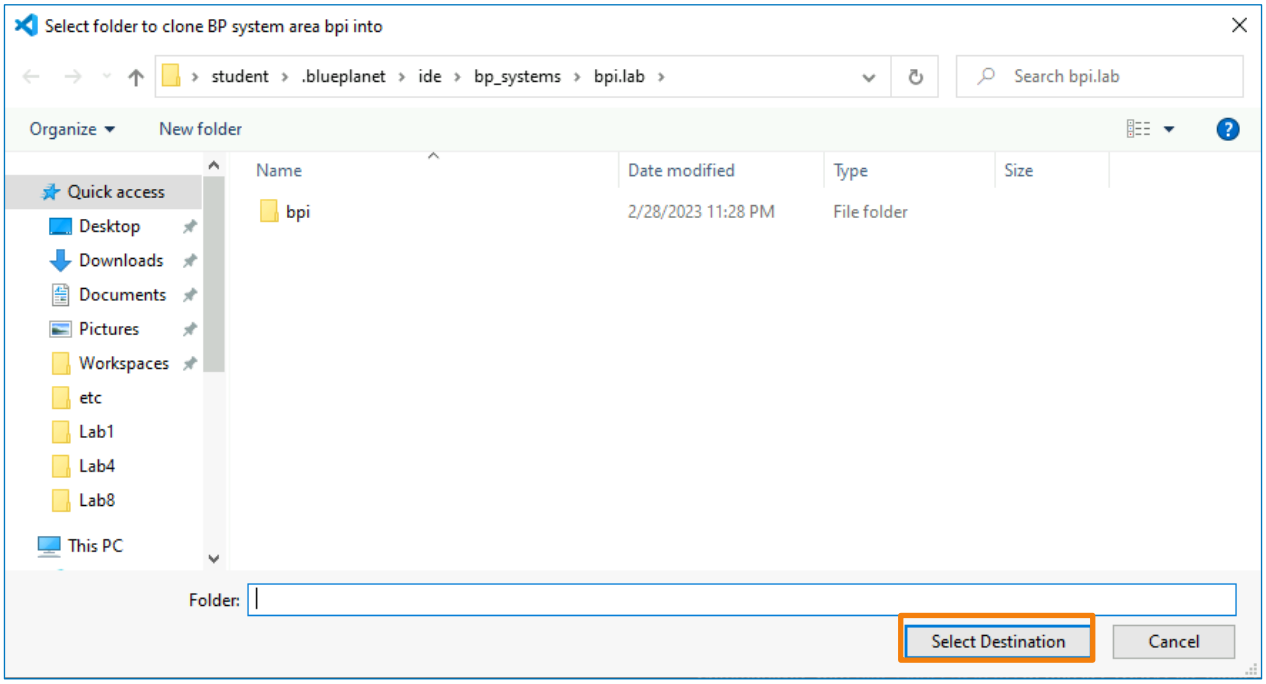


5. For the branch name, enter **lab** and press the **ENTER** key.

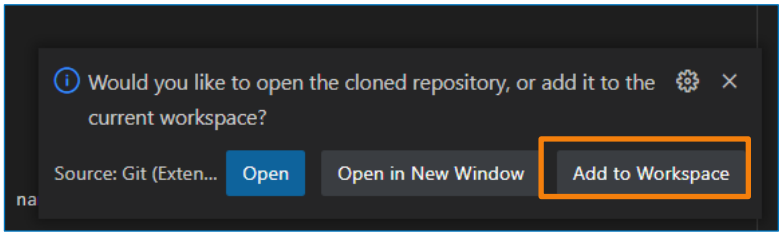


6. Click **Select Destination**.

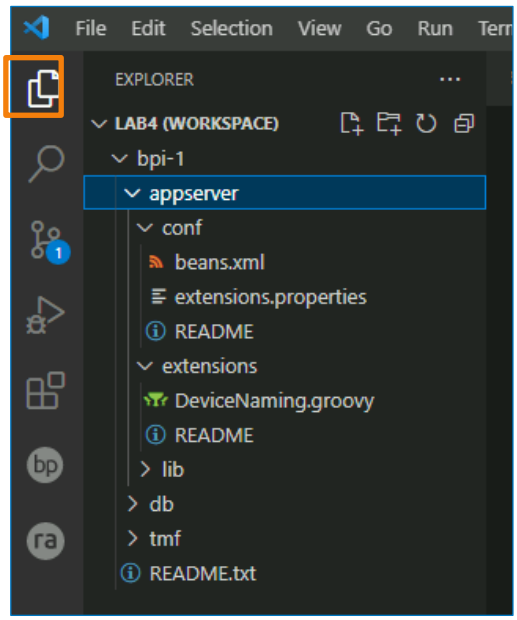
NOTE: It is important **not** to select “bpi”. Do not select any directory, VSC will create a new directory for you.



7. When asked, click on **Add to Workspace** to add the directory structure to the VSC workspace.

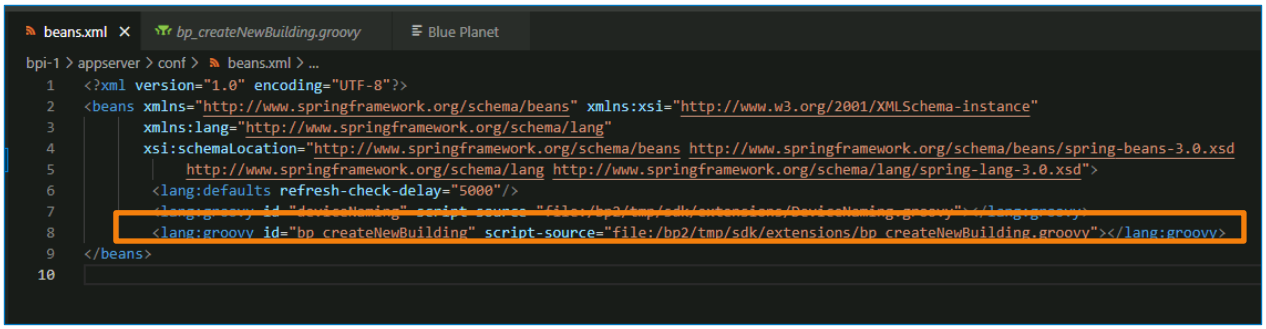


8. Your IDE is now ready. Click on **Explorer** from the left menu and expand **LAB4 (WORKSPACE)** to see the file structure. Expand **bpi > appserver > conf** and **bpi > appserver > extensions**.



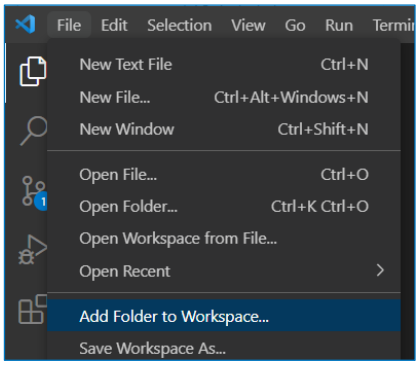
9. Under **bpi > appserver > conf** click the **beans.xml** file to open it. This file is used to register your groovy script (containing the service task code) so that the web application can execute the code when the workflow is executed. Add the following line before the last line:

```
<lang:groovy id="bp_createNewBuilding" script-
   source="file:/bp2/tmp/sdk/extensions/bp_createNewBuilding.groovy"></
   lang:groovy>
```

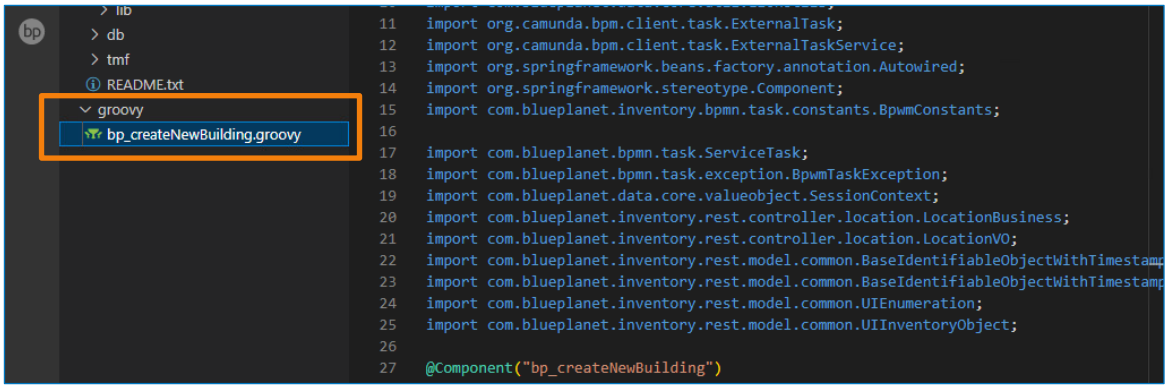


10. Press **CTRL+S** to save the changes.

11. Now add the groovy script to the project. The script has already been prepared for you, so you only need to import it and place it in the proper place in the file structure. First, add the folder with the script to the workspace by clicking the **File > Add Folder to Workspace…** menu option.

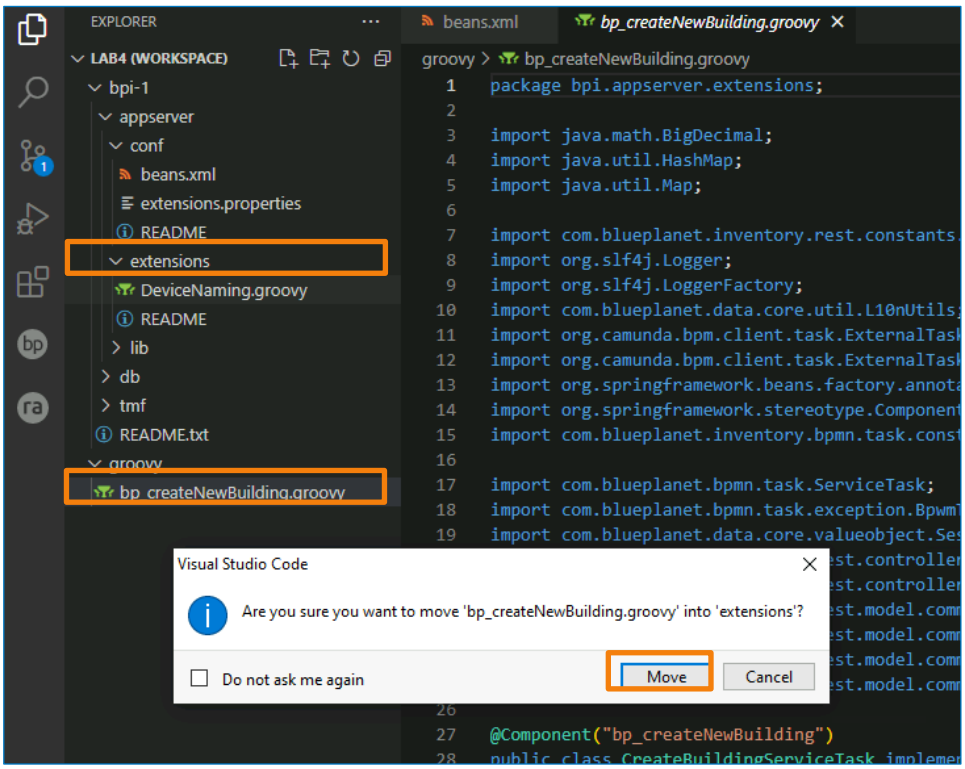


12. The file explorer will open. Select the directory **C:\Users\student\Workspaces\Lab4\groovy** and click on **Add**. Directory named **groovy** will be added to your directory structure. Expand it and click the file **bp_createNewBuilding.groovy** to open it.

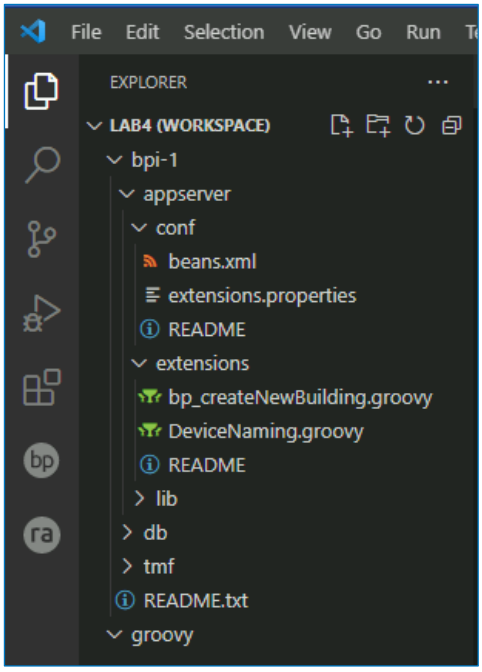


NOTE: Examine the script code. The UI inputs that you will enter when creating the workflow will be extracted to variables cityName and buildingName. Near the end of the try catch section, the UIInventoryObject is created by executing the business.createLocation method which is provided with all the required parameters.

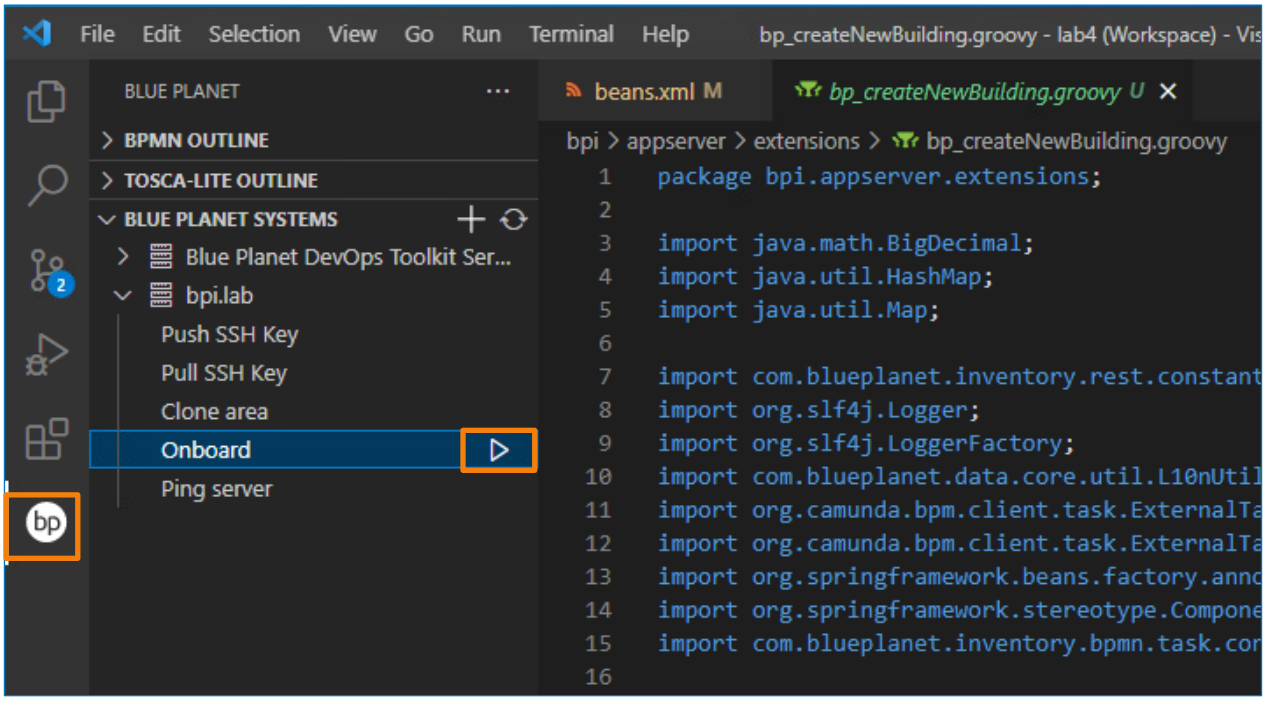
13. Move the groovy script from **groovy** to **bpi > appserver > extensions** by dragging and dropping it. Click on **Move** when asked.



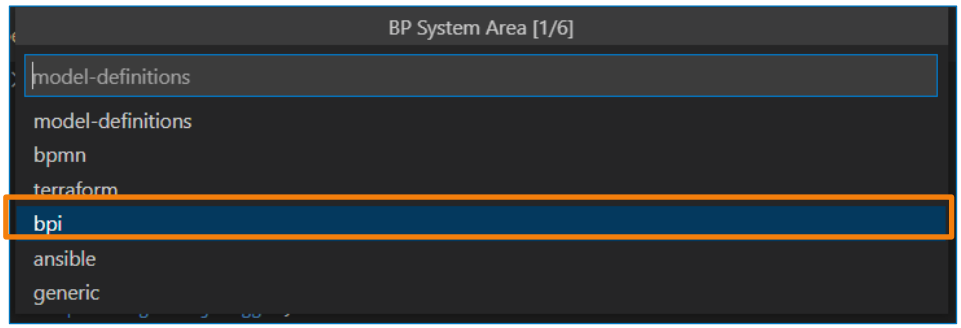
14. The structure now looks as follows:



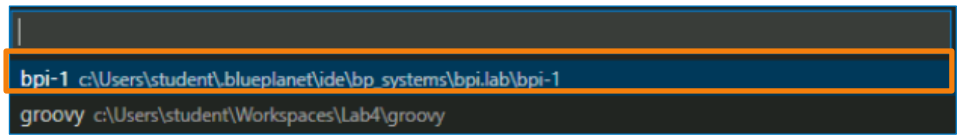
15. The application extension is now ready to be onboarded to the BPI server instance. Click **Blue Planet** from the menu and then click the **play icon** to the right of the **Onboard** menu option of your bpi.lab server.



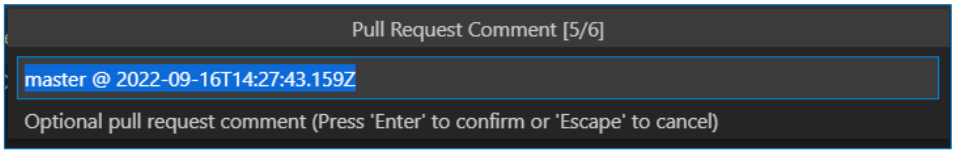
16. Select **bpi** from the dropdown box.



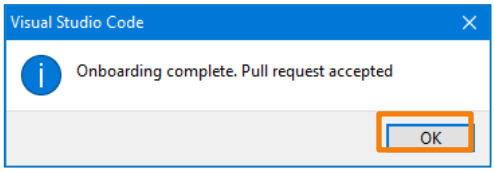
17. Click **bpi-1 (Primary Workspace Folder)** in the drop-down box.



18. Make sure that branch name is **master @ <timestamp>.** Press **ENTER** when asked for the Pull Request Comment.



19. The onboarding is successful, the files have been transferred to your BPI server. Click **OK**.



20. From your Student PC, open a terminal session to the BPI server. When asked for credentials use **bpadmin/bpadminpw**. You can use PuTTY to complete this step. Use the **kubectl delete pod** command to restart the web application container.

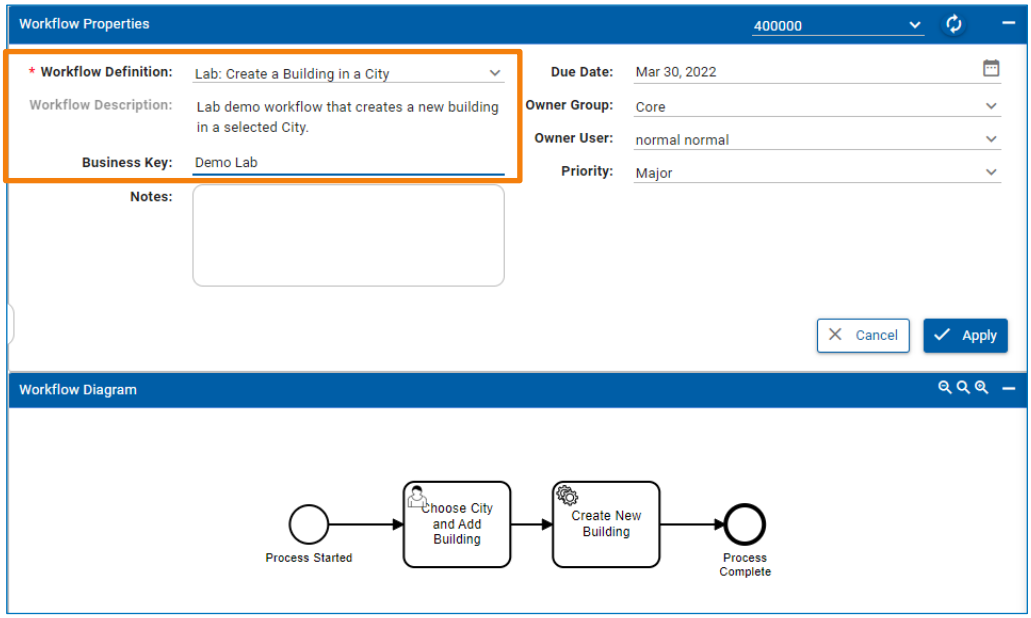
```
[bpadmin@bpi-2208 ~]$ kubectl delete pod web-0
pod "web-0" deleted
```

NOTE: When adding new scripts, Tomcat needs to be restarted. Later modifications on existing beans require no restart and should be picked up by Tomcat dynamically. Wait for several minutes for the container to be redeployed by Kubernetes before proceeding further. You can check the status of the web application with the **kubectl get pod web-0** command.

Task 5: Create a Workflow Instance and Verify Operation

In this task, you will create an instance of your newly deployed workflow definition and you will verify that the server is executing the implemented code by inspecting the log files on your pod's BPI server instance.

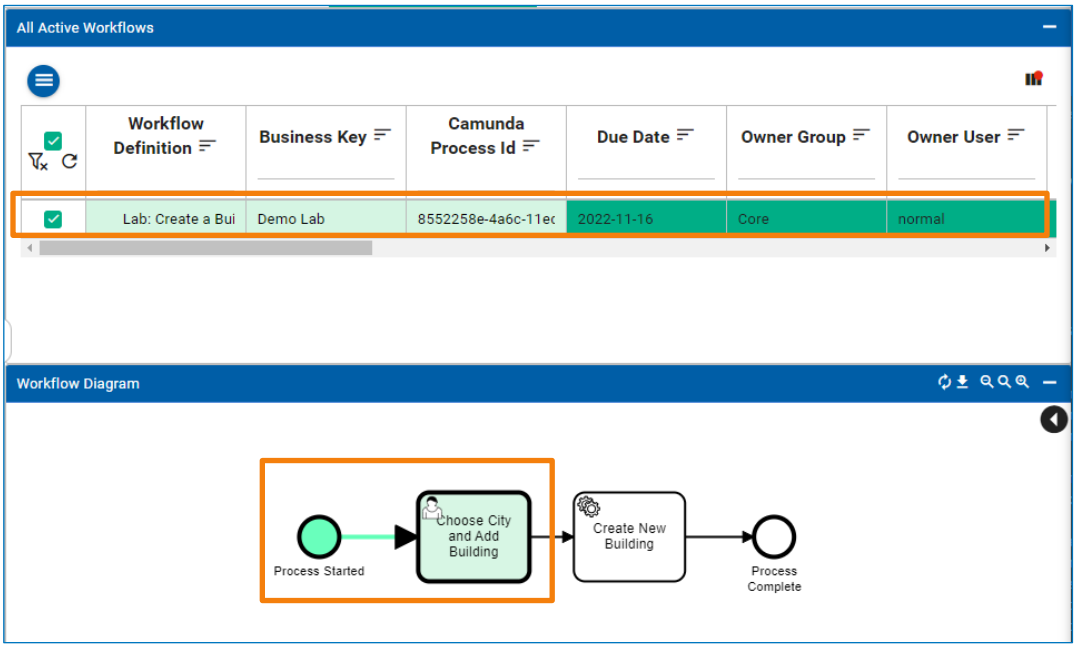
1. Open a web browser and log in to your BPI instance. From the Inventory main menu, select **Guided Operations > Create Workflow** to create a new workflow instance.
2. From the Workflow Definition drop-down box, select **Lab: Create a Building in a City**.
3. Specify **Demo Lab** as the Business Key and leave all other fields unchanged.



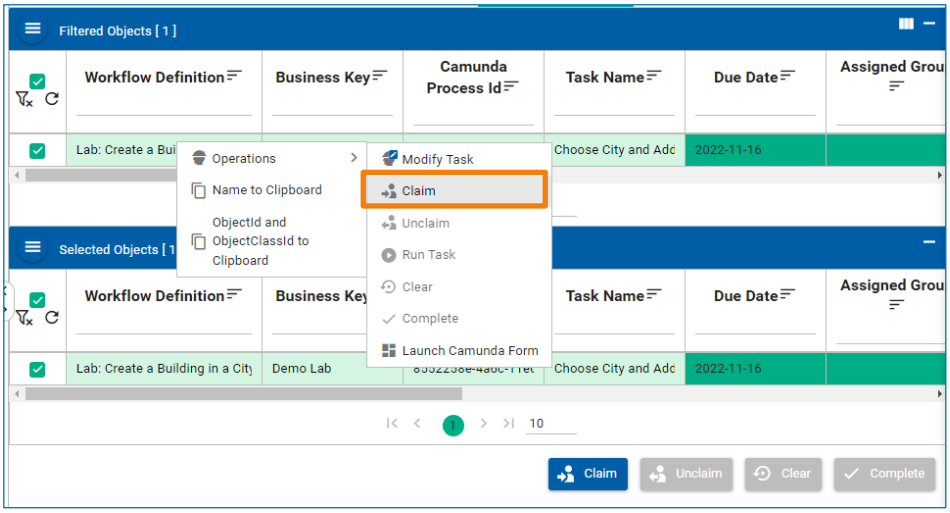
NOTE: The **Workflow Description** corresponds to the description from your BPMN file. Also, note the **Workflow Diagram** pane and observe that the diagram corresponds to the diagram that you created earlier in this exercise.

4. Click **Apply**.

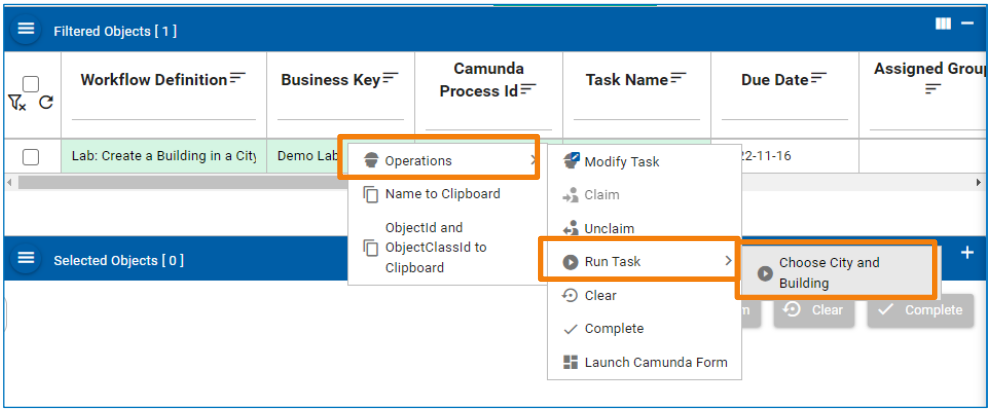
5. From the main menu, select **Guided Operations > Workflow Reports > All Active Workflows**. In the resulting table, click on the workflow that you just created. Observe in the lower pane that the workflow process has started, and the **Choose City and Add Building** user task is waiting for your input. The service task is not colored because it cannot run until the user task has been completed.



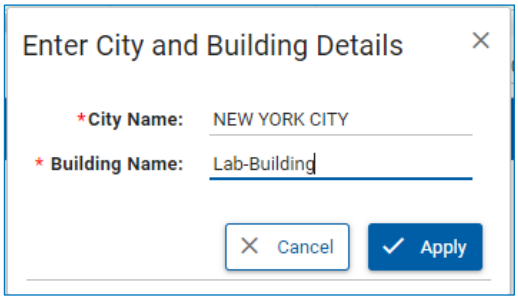
6. From the main menu, select **Guided Operations > Task Lists > All Active Tasks**. From the resulting table, right-click the first task (**Choose City and Add Building**) and select **Operations > Claim** to claim the task.



7. Right-click the task again and this time select **Operations > Run Task > Choose City and Building**.



8. The configured UI component for user input will pop up. Enter **NEW YORK CITY** under City Name and **Lab-Building** under Building Name. Note that the city name must be in capital letters.



NOTE: You can also drag the city from the locations pane on the left and drop it on the City Name in the form.

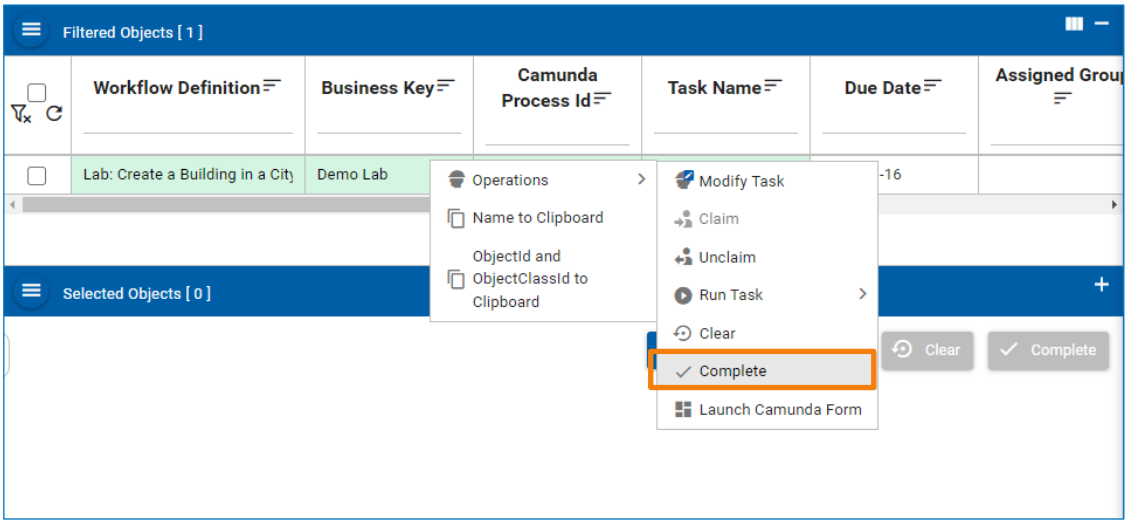
9. Click on **Apply**. The task input is now recorded but before you mark this user task as completed, set up log monitoring on your BPI server.
10. From your Student PC, use the server access information provided by your instructor to open the terminal session to the BPI server. When asked for credentials use **bpadmin/bpadminpw**. You can use PuTTY to complete this step.
11. Enter the following **kubectl** command in the BPI server terminal to enter the shell session of the **web-0** Kubernetes pod on your server:

```
[bpadmin@bpi-pod03 ~]$ kubectl exec -it web-0 -- bash
root@web-0:/dev/shm#
```

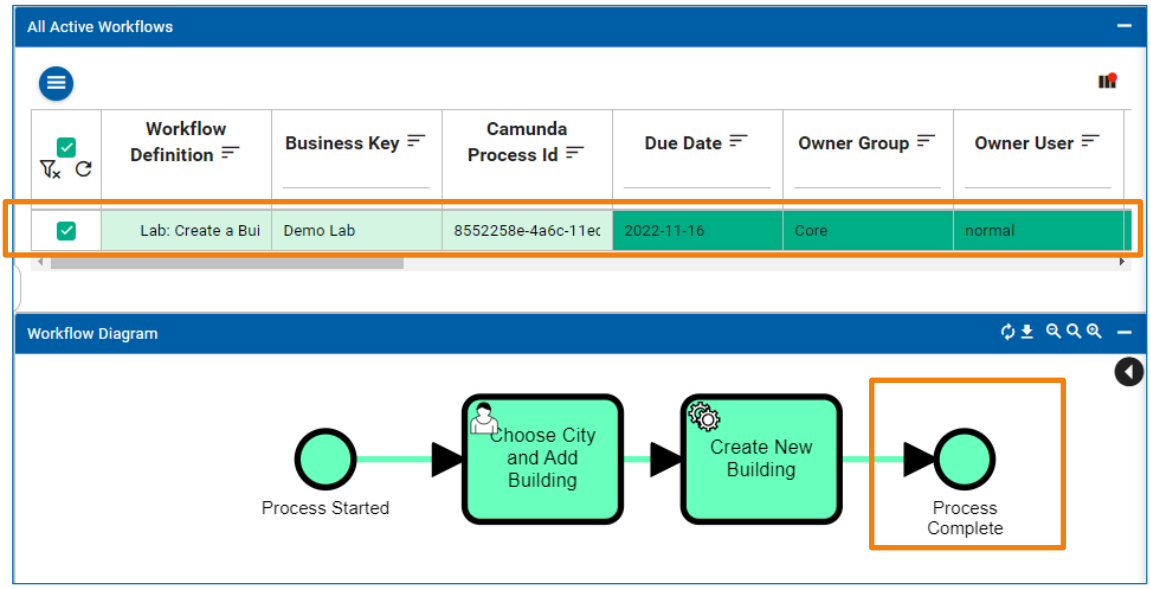
12. Once in the shell of the web-0 container, change the directory to **/bp2/log** and start monitoring the **catalina.out** log file.

```
root@web-0:/dev/shm# cd /bp2/log
root@web-0:/bp2/log# tail -f catalina.out | grep bpilab:
```

13. Return to your BP UI session in the browser and mark your user task as complete by right clicking the user task and selecting **Operations > Complete**. This will trigger the service task execution which, in turn, will result in the creation of the building as specified in the inputs.



14. From the main menu, select **Guided Operations > Workflow Reports > All Active Workflows**. In the resulting table, click on your workflow. In the **Workflow Diagram** pane, note that the process is now fully completed.



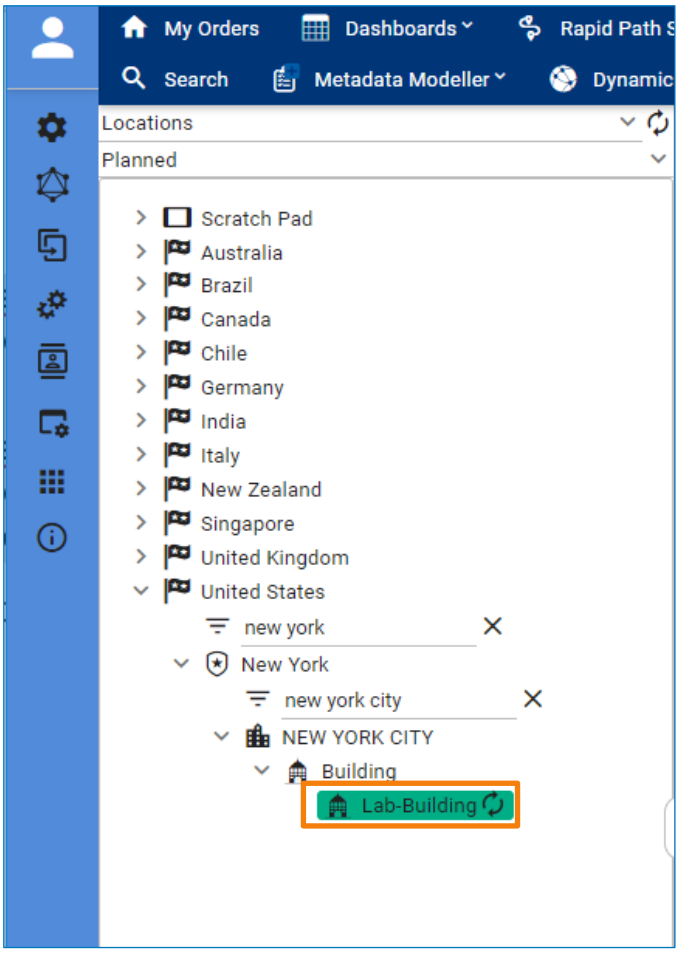
NOTE: It could take up to 10 minutes for the workflow to complete. Hit refresh until you can verify that all the steps have turned green (completed).

15. Return to your BPI server terminal session and verify that the events regarding the creation of your building were captured in the **catalina.out** log file.

```
<… output omitted …>
```

```
ERROR: 2023-02-22 11:43:41,037 : TopicSubscriptionManager bpi_tasks 1 :
   bpi.appserver.extensions.CreateBuildingServiceTask : bpilab: Executing Lab
   Create Building Service Task

ERROR: 2023-02-22 11:43:41,363 : TopicSubscriptionManager bpi_tasks 1 :
   bpi.appserver.extensions.CreateBuildingServiceTask : bpilab: Created
   building with name: Lab-building02
```

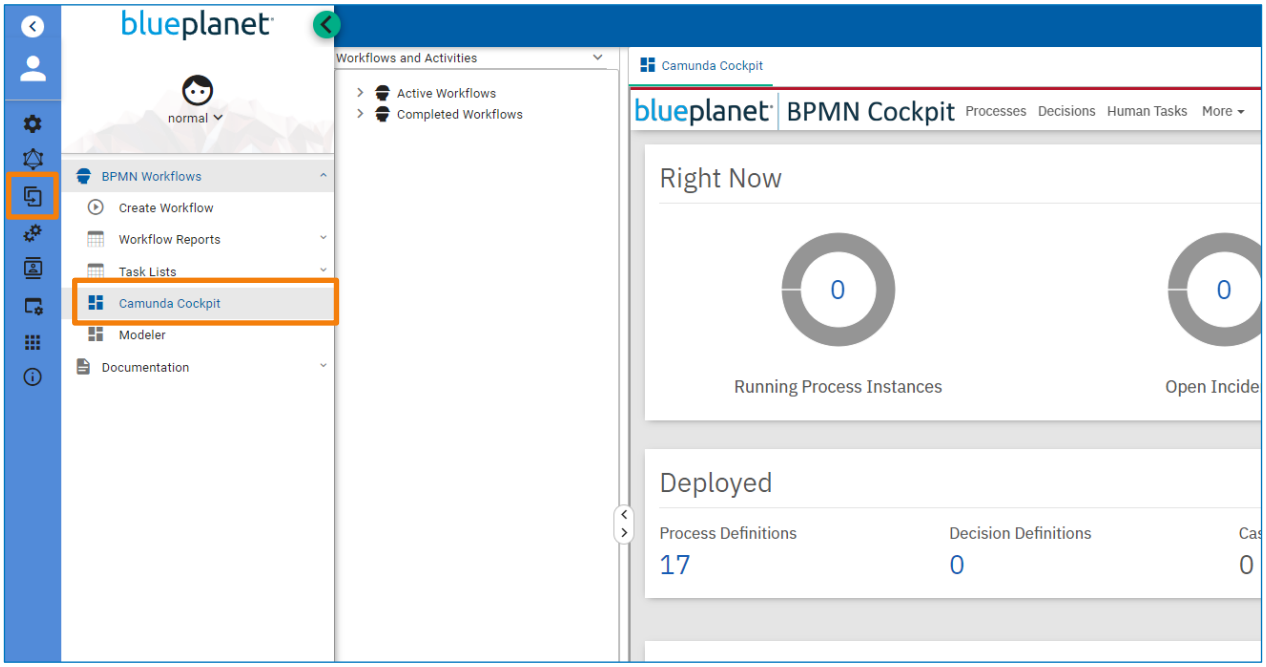
16. Finally, return to the BPI UI and confirm that the building was indeed created. In the **Locations** data tree navigate to **United States > New York > NEW YORK CITY > Building**. You will see a new building named Lab-Building.



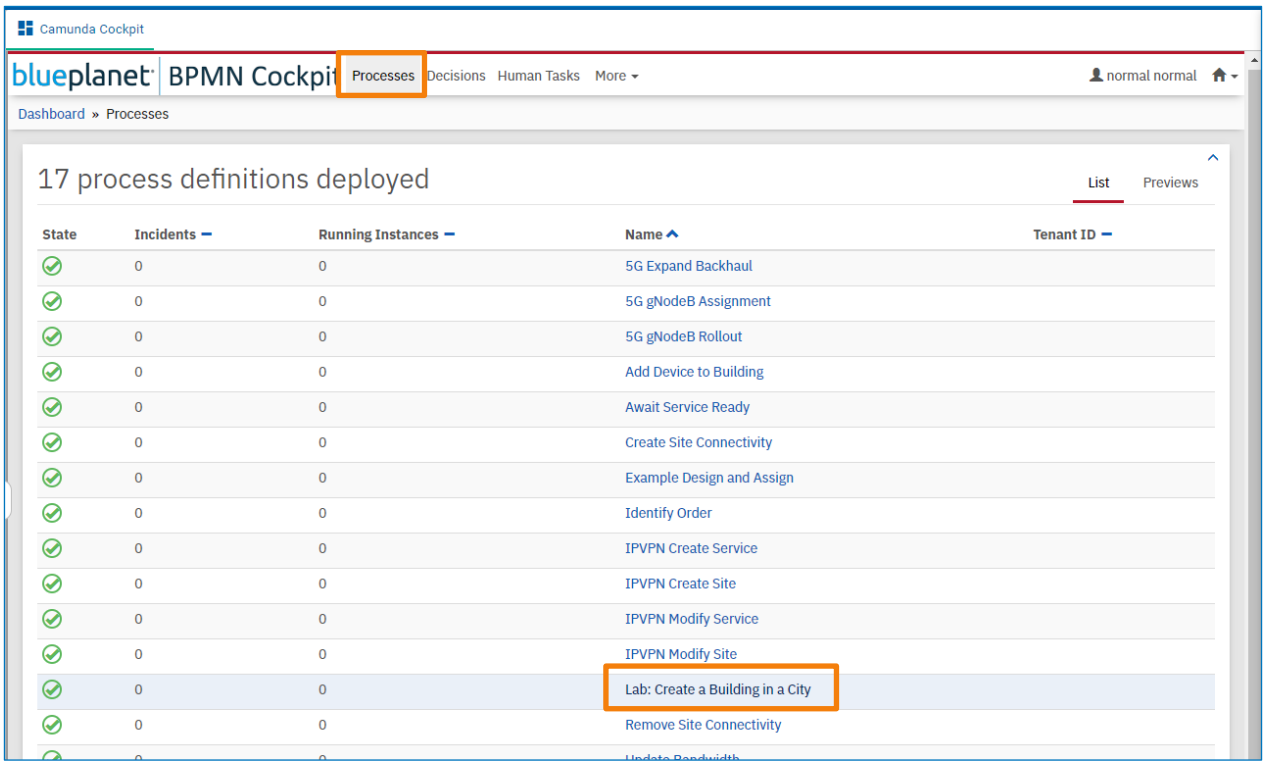
Task 6: Using BPMN APIs to Manage Workflows

In this task, you will create an instance of your deployed workflow definition, like the process in the previous task, but this time you will use the BPMN APIs to create the workflow instance.

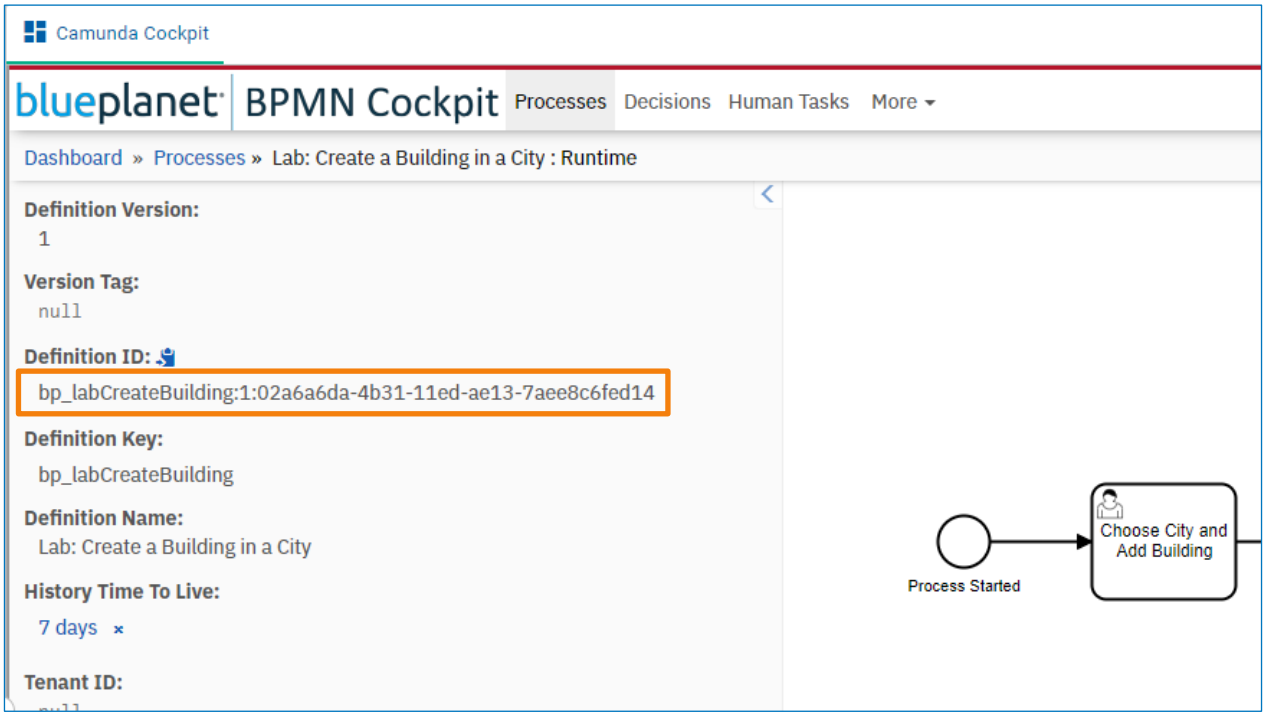
1. From the BPI UI session, in the navigation bar on the left, select **Workflow**, then **BPMN Workflows > Camunda Cockpit** to open the Camunda Cockpit.



2. In Camunda Cockpit, from the top menu, click **Processes** and then find and click your process definition **Lab: Create a Building in a City**.

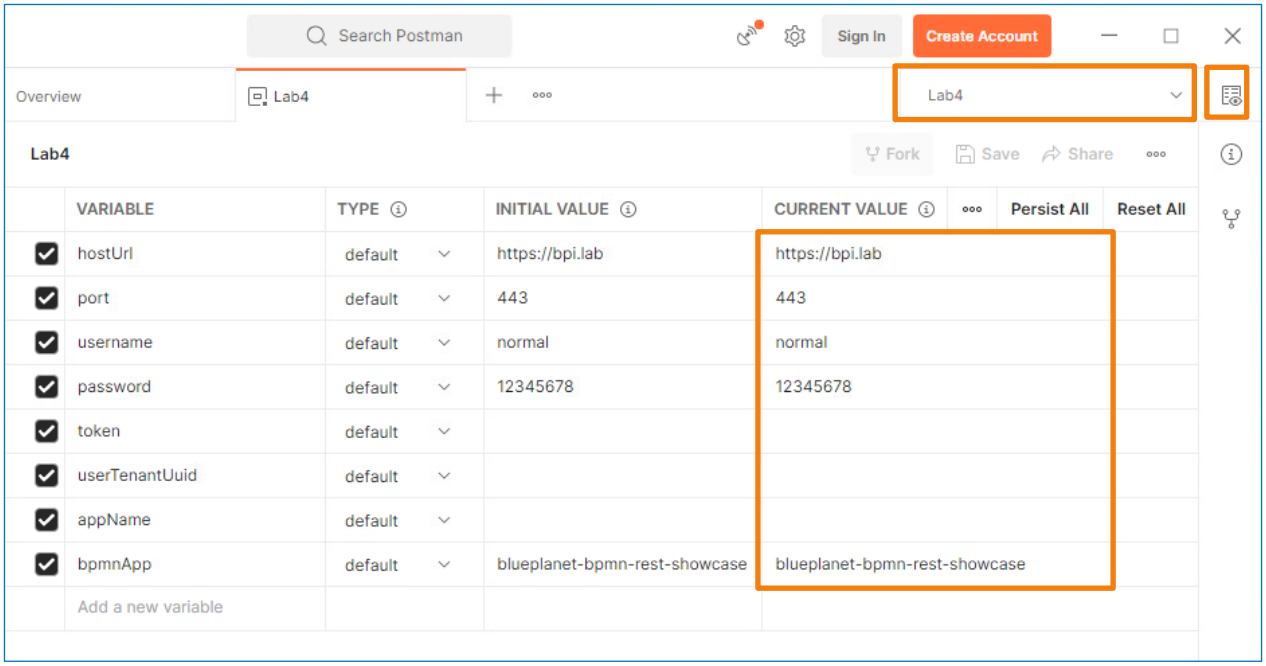


3. The workflow definition details page opens. Find the **Definition ID** and copy the text of it to the clipboard of your PC.

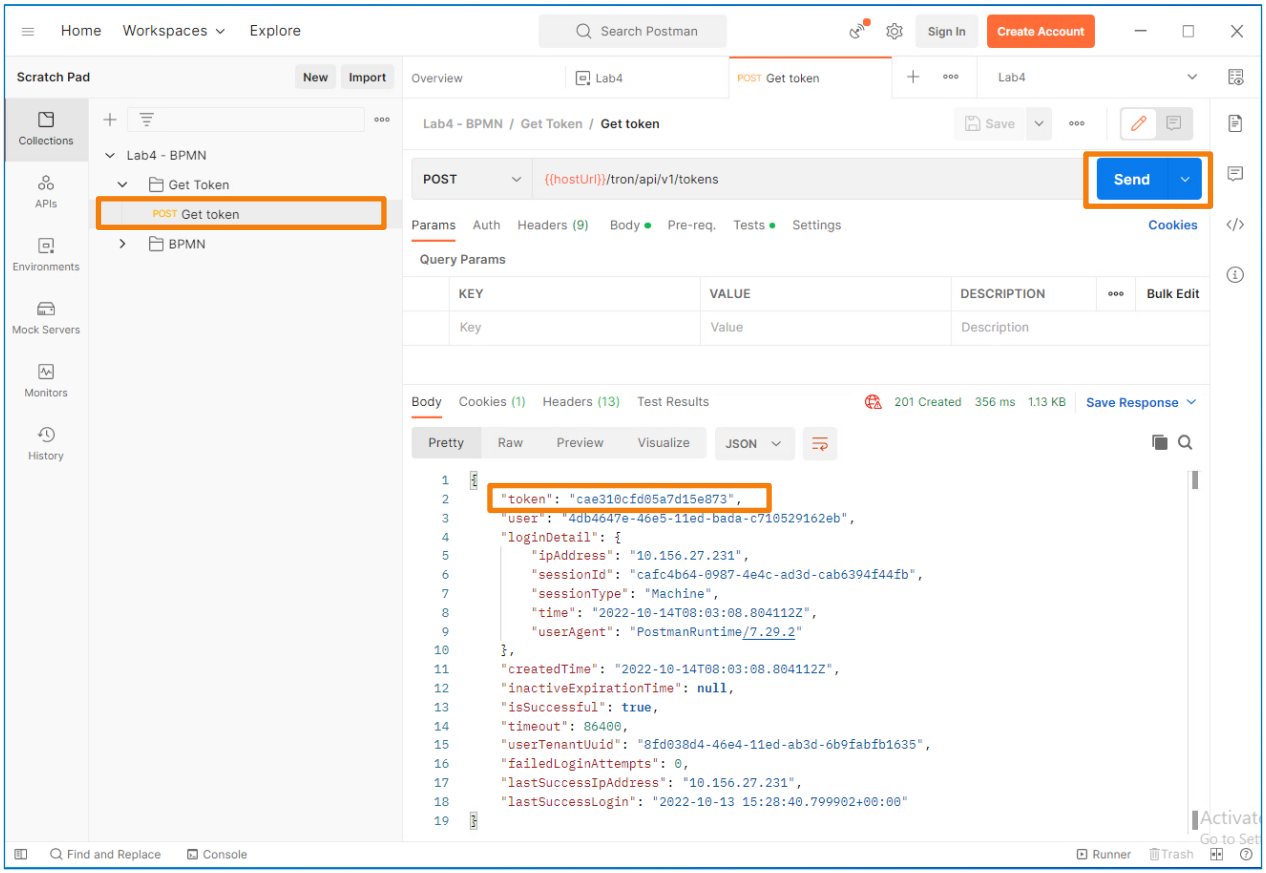


NOTE: Your Definition ID may be different than what is on the image in this lab guide.

4. Now from your Student PC desktop, open **Postman**. Verify that the environment selection is set to **Lab4**. Click the **Environment quick look** button to the right of Lab4 in the top right corner of Postman, then **Edit**. Verify that the current values in the environment are set as in the following image:

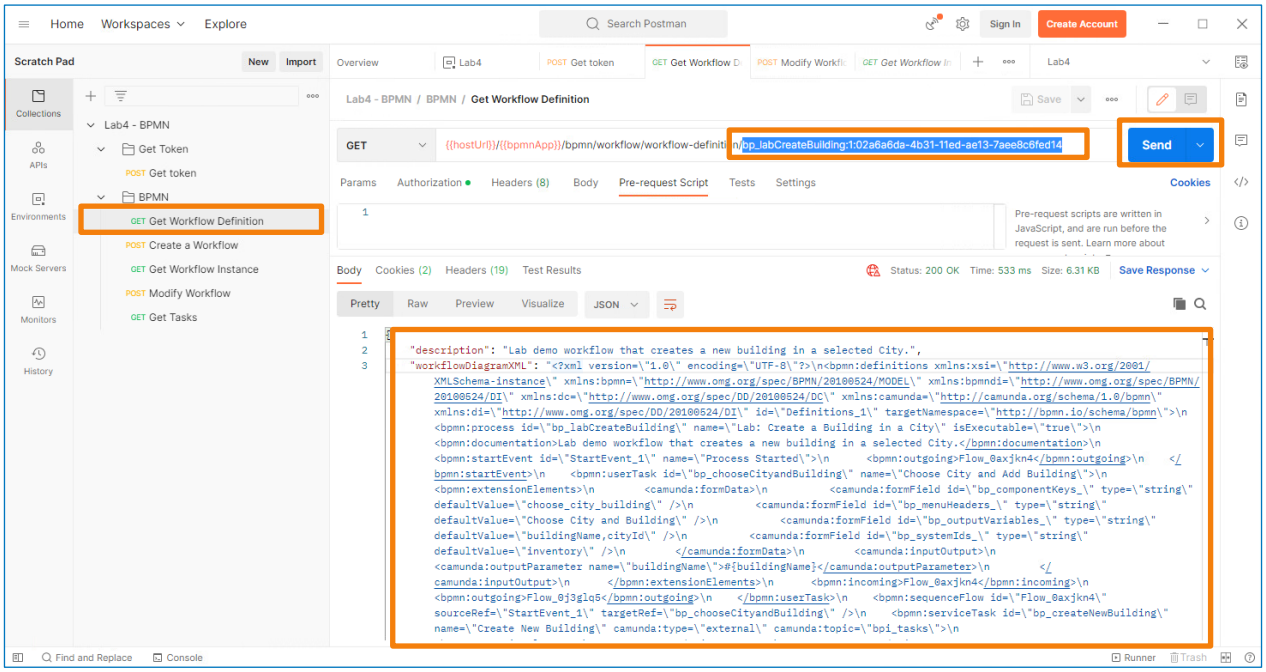


5. In the **Lab4 - BPMN** Postman collection in the left pane, expand **Get Token** and click the **Get token** POST request to get a token from the BPI server. This token will be stored locally and used to authorize all your subsequent API requests. The request details will open in the right pane. Now click **Send** and observe the output. You should get a token value back from the server.

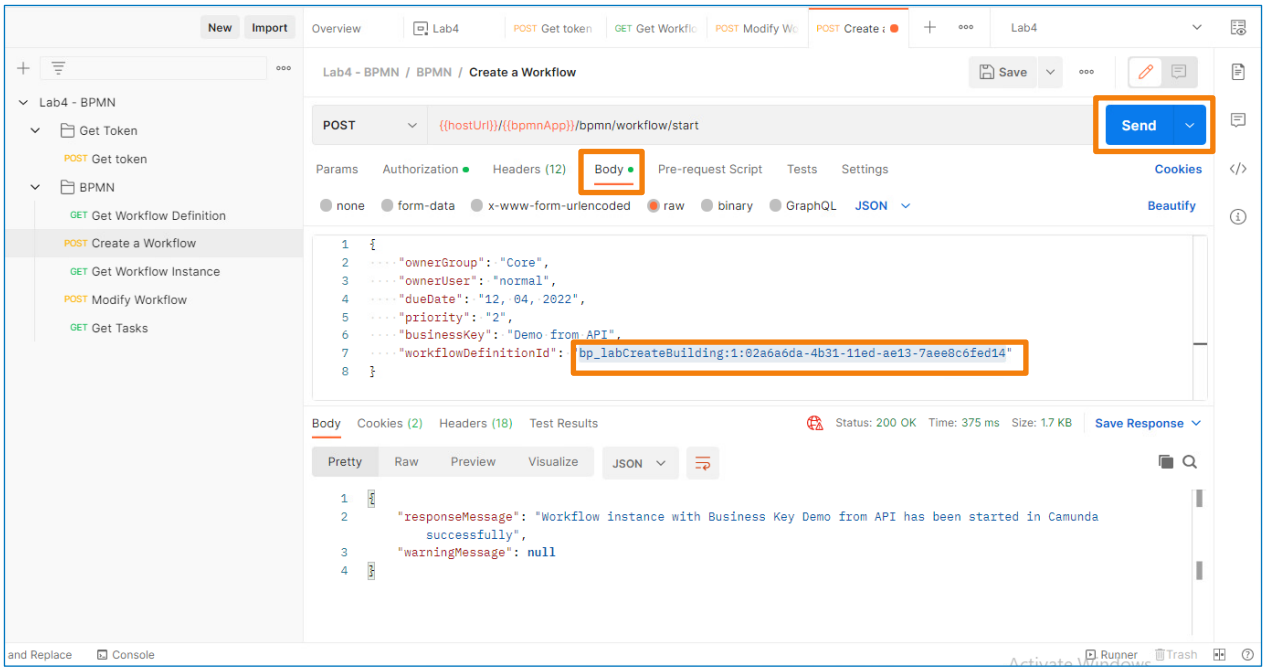


NOTE: The Get token request itself is authorized by the username and password that you defined in the Postman environment. You can verify this by inspecting the Body section of the request.

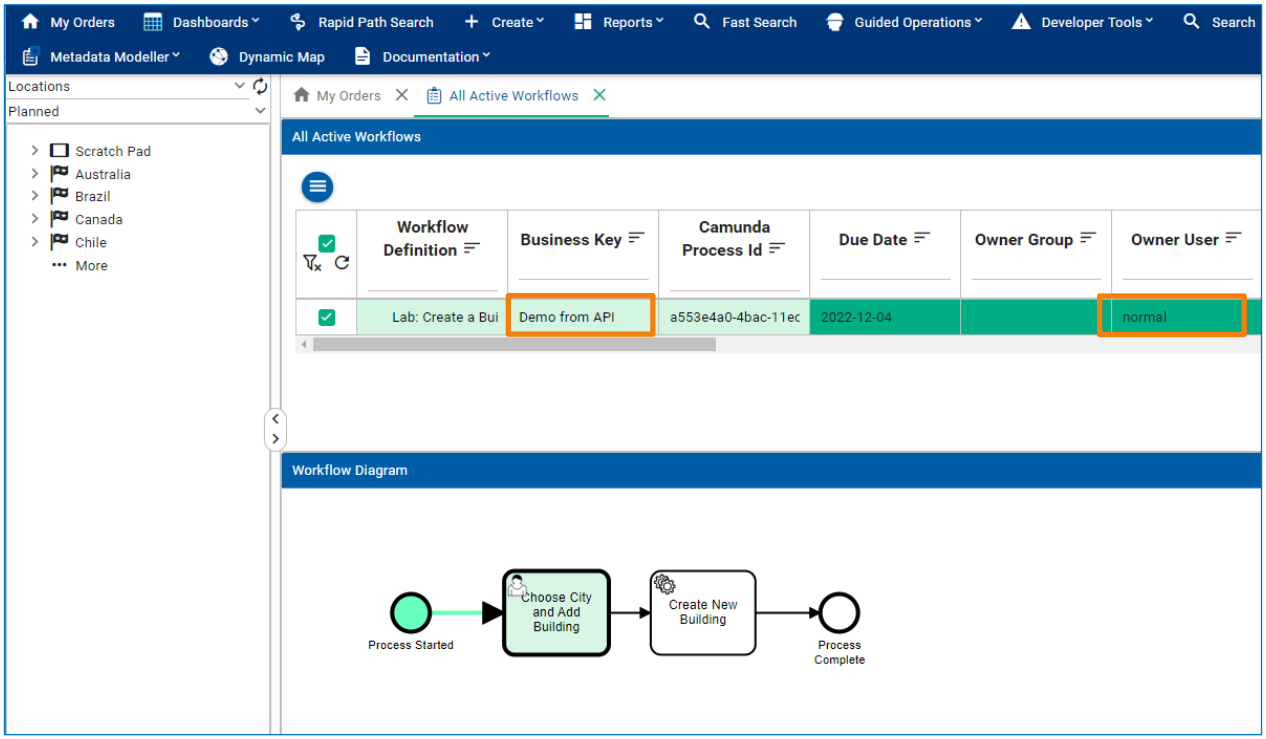
6. Now expand **BPMN** and click on the **Get Workflow Definition** request. Make sure you update the request URL with your own **workflowDefinitionId** value that you copied from Camunda Cockpit, e.g., *bp_labCreateBuilding:1:02a6a6da-4b31-11ed-ae13-7aee8c6fed14*. Click **Send**. The purpose of this task is to show how you can retrieve the workflow definition in the XML format via the BPMN API. You can then use the XML in the Camunda Modeler or BPI UI Modeler. The definition will be contained in the **workflowDiagramXML** field.



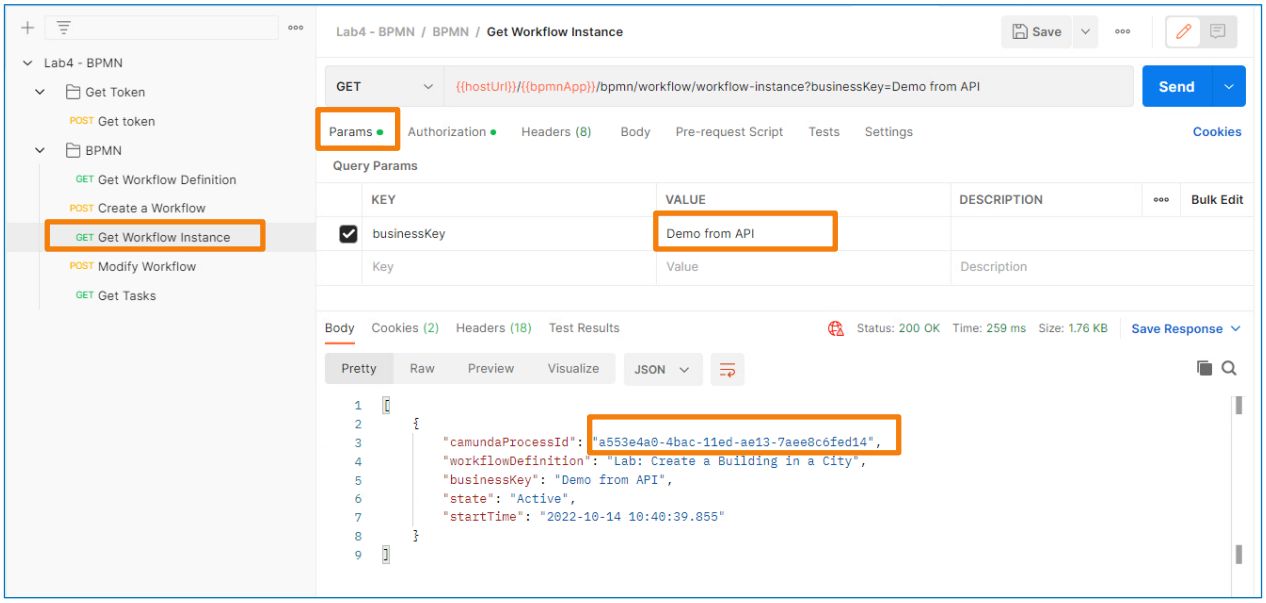
7. Create another workflow instance. In **BPMN**, click the **Create a Workflow** request. In the right pane, click **Body** to inspect the body of the request. In the Body text, update the **workflowDefinitionId** value with the value that you copied from Camunda Cockpit. Also make sure that the **dueDate** value is in the future of the current date, and then click on **Send**. The responseMessage shows a successful creation of the instance.



8. Return to your browser session and in the BPI UI, verify that the workflow instance was created as per the API request. Navigate to **Inventory > Guided Operations > All Active Workflows** and observe the workflow that you created via the API is in the table. Observe that the owner is user **normal**.

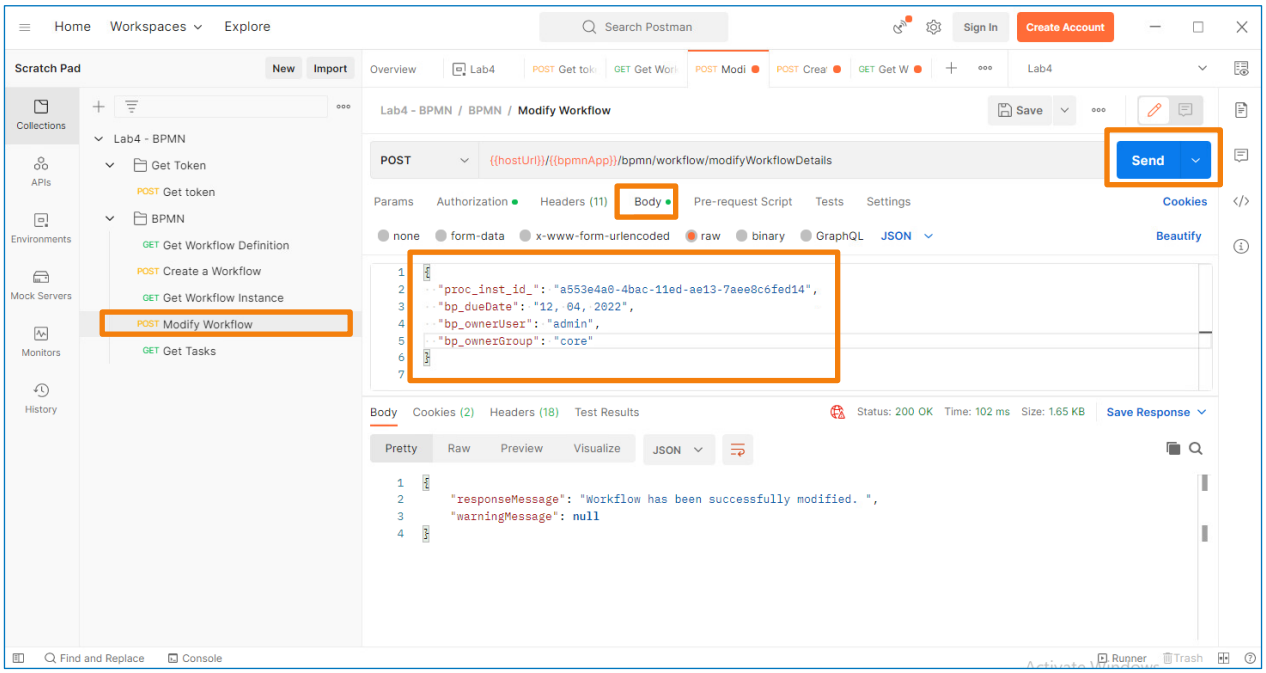


9. To retrieve specific workflow instance details, from Postman, use the **Get Workflow Instance** request. In the right pane click **Params**. For business Key, under VALUE, enter **Demo from API** and then click on **Send**. Observe that the resultMessage contains the details of the workflow instance you created earlier.



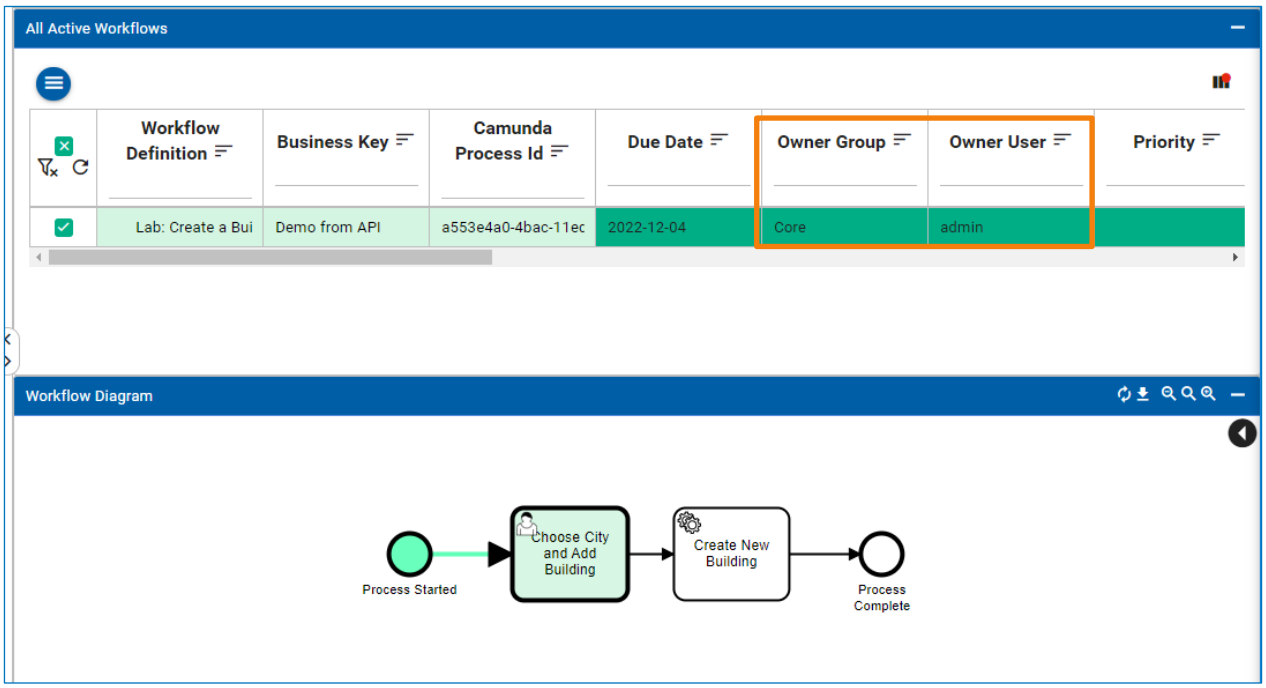
NOTE: Make a note of the **camundaProcessId** parameter value because you will need it in the next step.

10. Proceed to modify the workflow owner and group parameters of an active workflow. Use the workflow that you created in this exercise. To achieve this goal, use the **Modify Workflow** request from BPMN in the left pane. Click on **Body** to modify the parameters according to the information below and then click on **Send**:
    a. **proc_inst_id**: use the value of camundaProcessId from the previous step.
    b. **bp_due_date**: you must use the same due date from when you created the workflow.
    c. **bp_ownerUser**: “admin”
    d. **bp_ownerGroup**: “core”

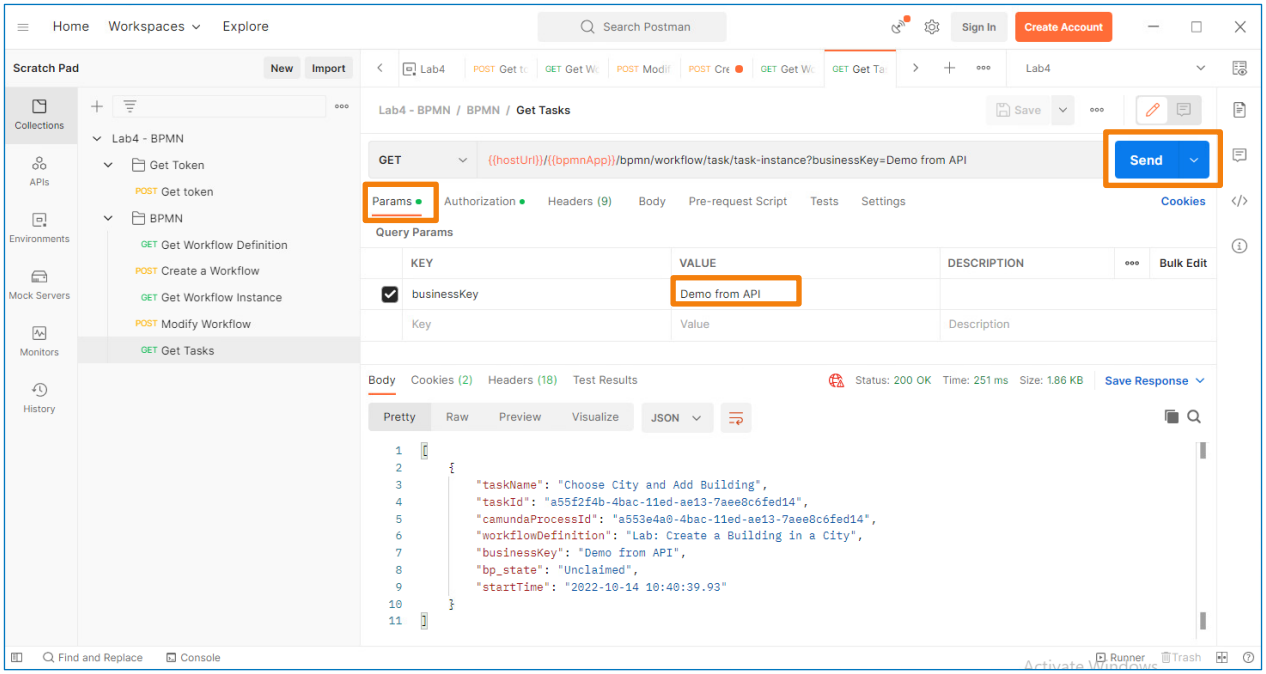


NOTE: Be careful to type “core” in lower case.

11. Return your browser UI session, refresh the table and verify that the workflow instance was modified. Now the owner user is admin.



12. To get a list of user tasks for a specific workflow instance, use the predefined **Get Tasks** request from Postman. Click **Params** and verify that the businessKey is set to “Demo from API” and click on **Send**. The response contains all user tasks defined for this workflow, in this instance there is only one task.



End of Lab

Lab 5: Rapid Path Search

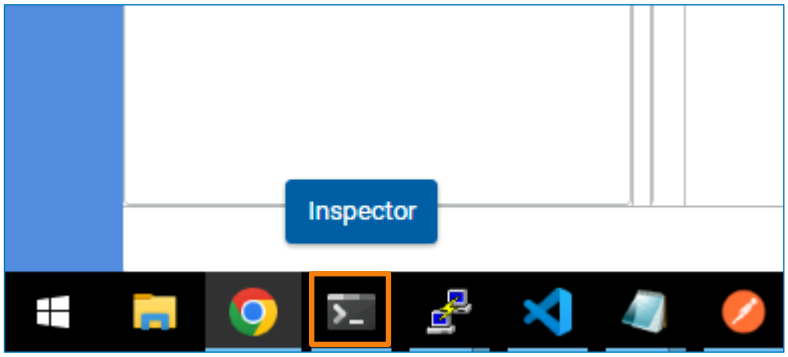
Objectives

- Create a new Neo4j plugin project
- Define new packages and classes using VS Code
- Create code that will prefer certain connection types over others, based on assigning different weights, and in that way influence the election of best routes
- Deploy the Neo4j plugin using the Custartifact side loader
- Test the new plugin against the same data to verify that the election of best paths has changed

Task 1: Create a New Neo4j Plugin Project

In this task, you set up the project in the Visual Studio Code (VS Code) integrated development environment (IDE) where you create your new business rule and respective classes.

1. From your Student PC desktop, open a Windows PowerShell application from the taskbar shortcut.



2. From PowerShell, change the directory to **Workspace\Lab5**.

```
PS C:\Users\student> cd .\Workspaces\Lab5
PS C:\Users\student\Workspaces\Lab5> dir


    Directory: C:\Users\student\Workspaces\Lab5


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         1/6/2023   2:52 AM            198 create_project.bat
-a----         1/6/2023   4:45 AM            584 deploy_project.bat
-a----         1/6/2023   4:18 AM           1370 pom.txt
```

NOTE: The folder contains scripts required to create and deploy projects, as well as the pom file that you will use to build the project.

3. In this exercise, you create a new Neo4j plugin Java project. Note that in your current directory, there is a script called **create_project.bat**. This script contains the Maven command used to create a new simple project, which will also create the appropriate file structure for the project. Display and observe the contents of the script using the **more** command.

```
PS C:\Users\student\Workspaces\Lab5> more .\create_project.bat

mvn archetype:generate ^

       "-DgroupId=com.blueplanet.lab.routing.neo4j.rules" ^

       "-DartifactId=lab-neo4j-plugin" ^

       "-DarchetypeArtifactId=maven-archetype-quickstart" ^

       "-DinteractiveMode=false"


PS C:\Users\student\Workspaces\Lab5>
```

4. Run the **create_project.bat** script now.

```
PS C:\Users\student\Workspaces\Lab5> .\create_project.bat


C:\Users\student\Workspaces\Lab5>mvn archetype:generate        "-
   DgroupId=com.blueplanet.lab.routing.neo4j.rules"     "-
   DartifactId=lab-neo4j-plugin"  "-DarchetypeArtifactId=maven-
   archetype-quickstart"      "-DinteractiveMode=false"

[INFO] Scanning for projects...

[INFO]

[INFO] ------------------< org.apache.maven:standalone-pom >------------
   -------

[INFO] Building Maven Stub Project (No POM) 1

[INFO] --------------------------------[ pom ]--------------------------
   -------

[INFO]

<… output omitted …>

[INFO] Parameter: basedir, Value: C:\Users\student\Workspaces\Lab5

[INFO] Parameter: package, Value: com.blueplanet.lab.routing.neo4j.rules

[INFO] Parameter: groupId, Value: com.blueplanet.lab.routing.neo4j.rules

[INFO] Parameter: artifactId, Value: lab-neo4j-plugin

[INFO] Parameter: packageName, Value:
   com.blueplanet.lab.routing.neo4j.rules

[INFO] Parameter: version, Value: 1.0-SNAPSHOT

[INFO] project created from Old (1.x) Archetype in dir:
   C:\Users\student\Workspaces\Lab5\lab-neo4j-plugin
```

```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  7.860 s
[INFO] Finished at: 2023-01-06T02:52:48-12:00
[INFO] ------------------------------------------------------------------------
```

5. Verify that the project directory **lab-neo4j-plugin** has been created in your current directory and that it contains the **src** directory as well as the **pom.xml** file.

```
PS C:\Users\student\Workspaces\Lab5> dir


    Directory: C:\Users\student\Workspaces\Lab5


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         1/6/2023   4:30 AM                lab-neo4j-plugin
-a----         1/6/2023   2:52 AM            198 create_project.bat
-a----         1/6/2023   4:45 AM            584 deploy_project.bat
-a----         1/6/2023   4:18 AM           1370 pom.txt

PS C:\Users\student\Workspaces\Lab5> dir lab-neo4j-plugin\


    Directory: C:\Users\student\Workspaces\Lab5\lab-neo4j-plugin


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         1/6/2023   2:52 AM                src
-a----         1/6/2023   2:52 AM            701 pom.xml
```

6. Next, from the Lab5 directory, copy the **pom.txt** file to lab-neo4j-plugin as pom.xml and overwrite the Maven-created pom.xml.

```
PS C:\Users\student\Workspaces\Lab5> copy pom.txt .\lab-neo4j-
   plugin\pom.xml
```

7. Verify that your **pom.xml** now appears as the following output. It should have four dependencies:
   - neo4j-kernel
   - routing-neo4j-core
   - routing-neo4j-rules
   - routing-neo4j-procedures

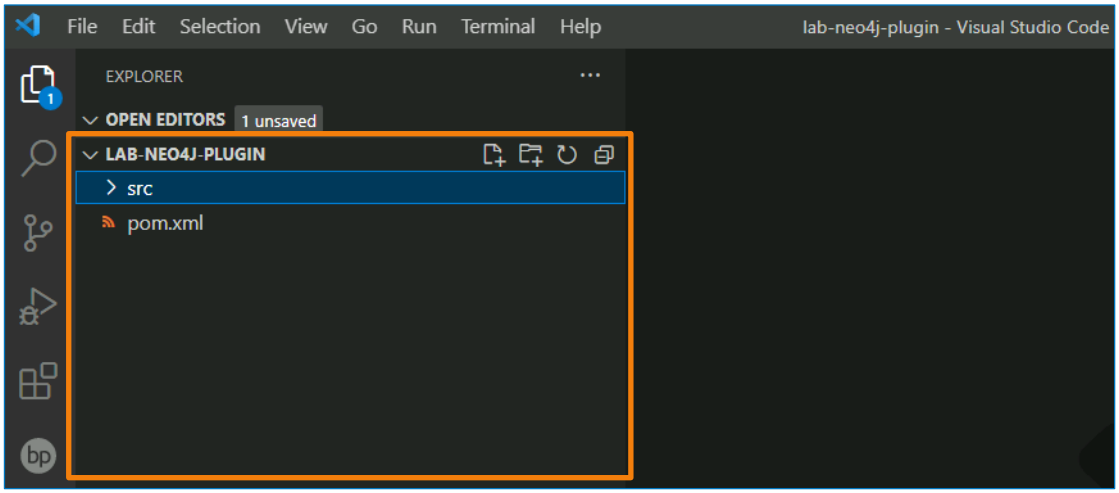
```
PS C:\Users\student\Workspaces\Lab5> more .\lab-neo4j-plugin\pom.xml


<project xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
   http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.blueplanet.lab.routing.neo4j.rules</groupId>
  <artifactId>lab-neo4j-plugin</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>lab-neo4j-plugin</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
        <groupId>org.neo4j</groupId>
        <artifactId>neo4j-kernel</artifactId>
        <version>3.3.2</version>
    </dependency>
    <dependency>
        <groupId>com.blueplanet.routing</groupId>
        <artifactId>routing-neo4j-core</artifactId>
        <version>22.8.2</version>
    </dependency>
    <dependency>
        <groupId>com.blueplanet.routing</groupId>
        <artifactId>routing-neo4j-rules</artifactId>
        <version>22.8.2</version>
```

```
    </dependency>
    <dependency>
        <groupId>com.blueplanet.inventory</groupId>
        <artifactId>routing-neo4j-procedures</artifactId>
        <version>22.8.2</version>
    </dependency>
  </dependencies>
  <properties>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
</properties>
</project>
```

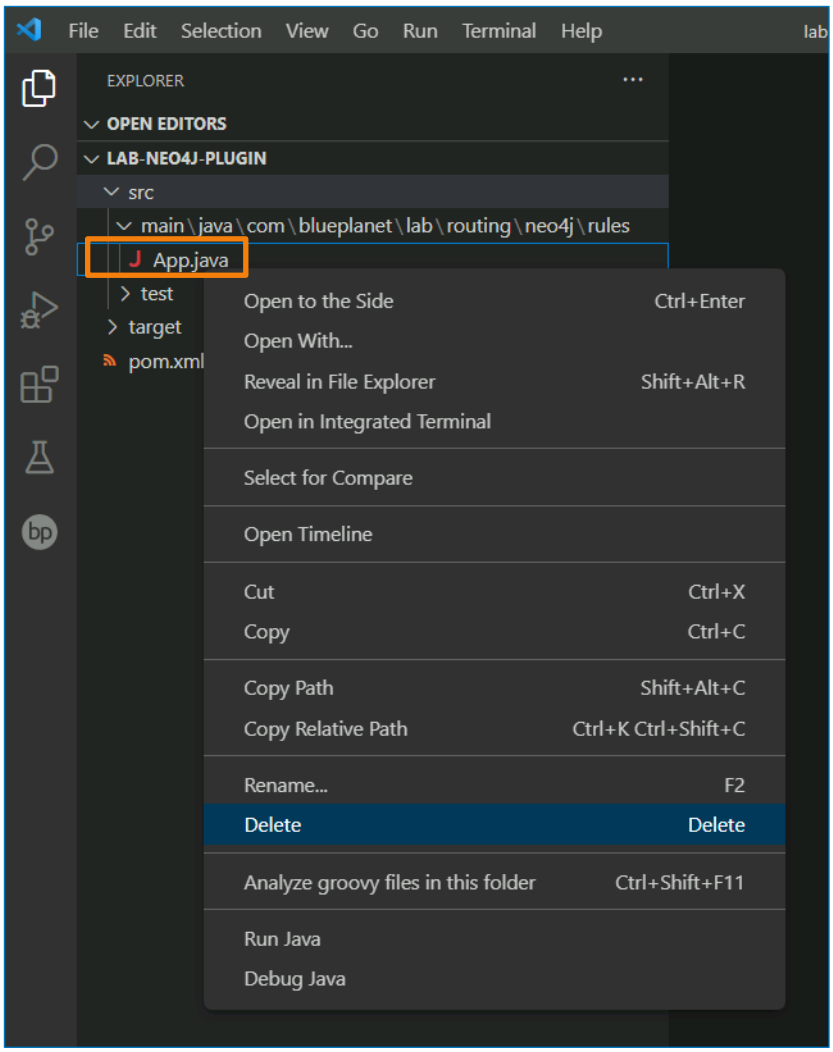
8. From PowerShell, while in directory **C:\Users\student\Workspaces\Lab5**, start VS Code and point it to your project directory to open the previously created Java project.

```
PS C:\Users\student\Workspaces\Lab5> code .\lab-neo4j-plugin\
```

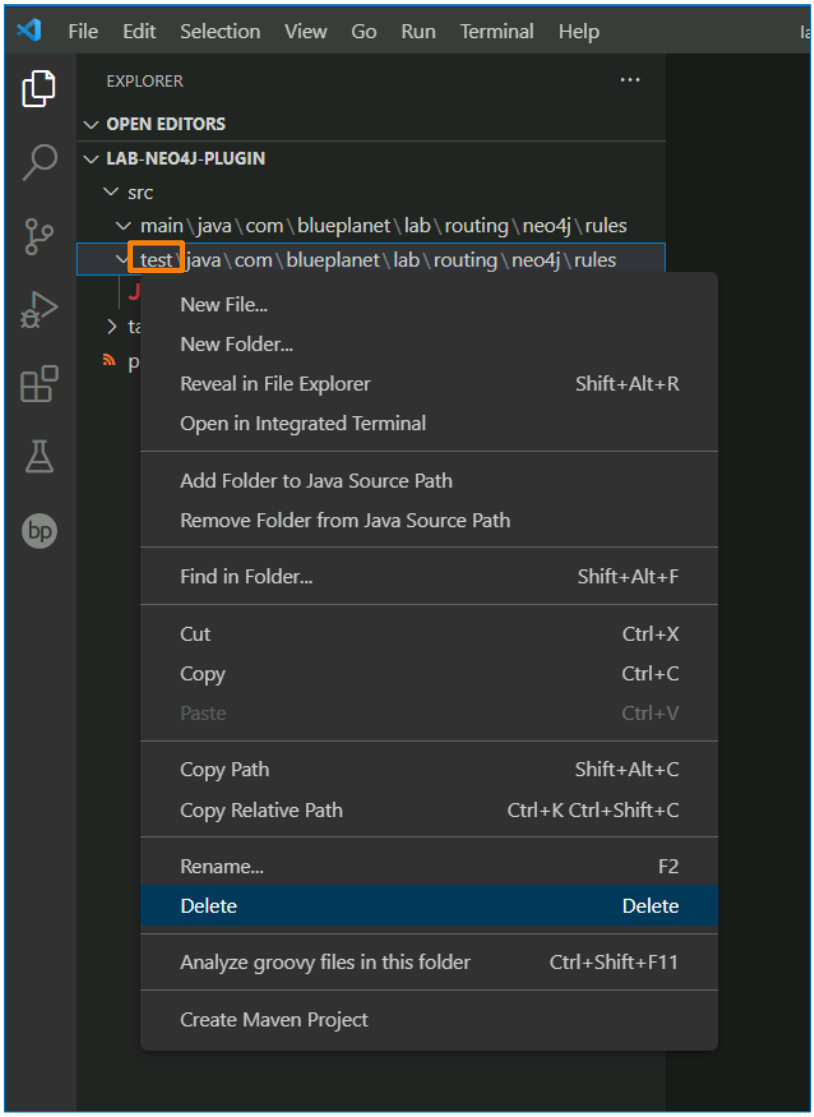
9. VS Code opens with your project structure on the left side in the Explorer pane.



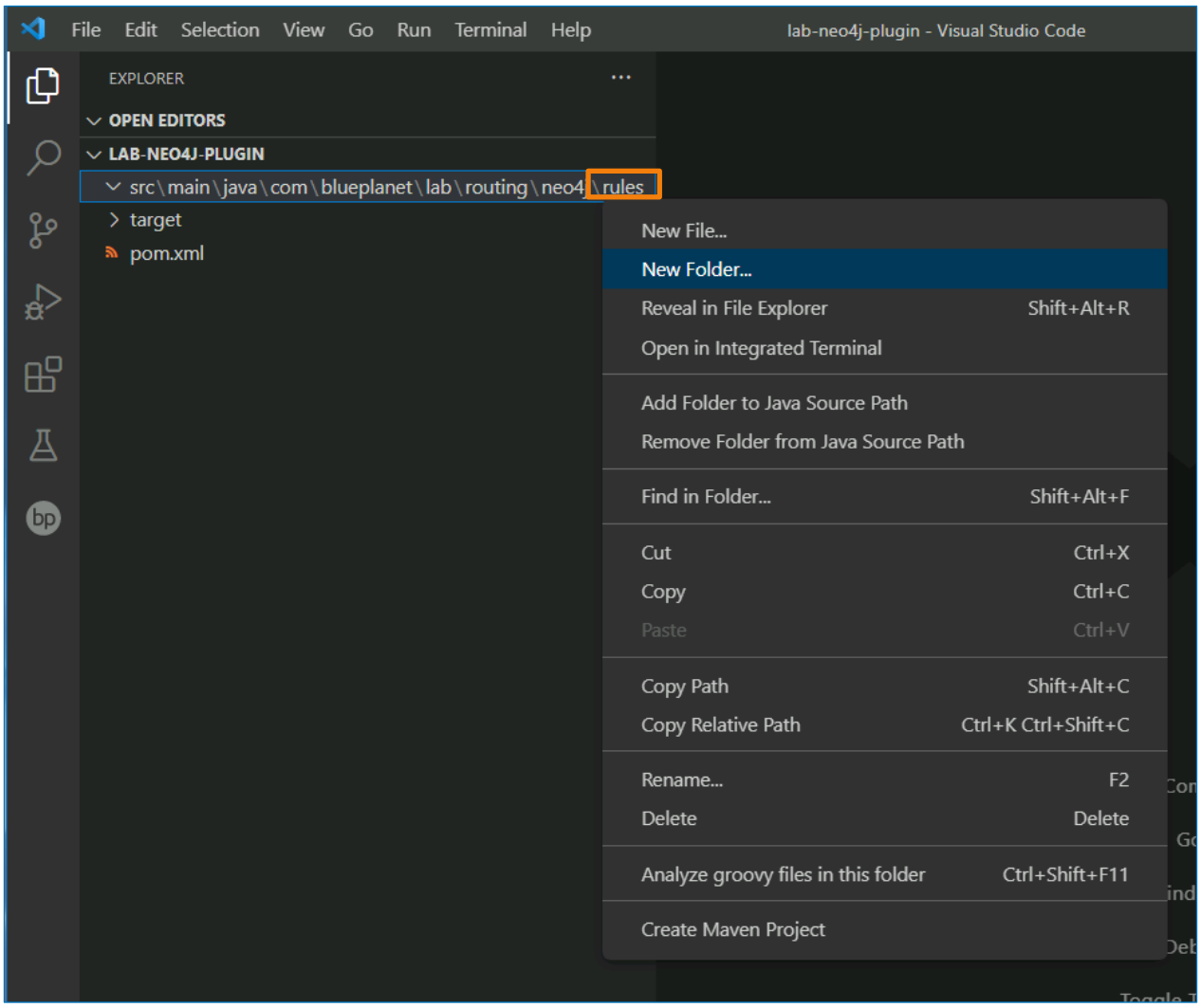
10. Delete the default **App.java** file as you don't require it for this exercise. Right-click the file and select **Delete**.



11. Also delete the test directory. Right-click **test** and select **Delete**.

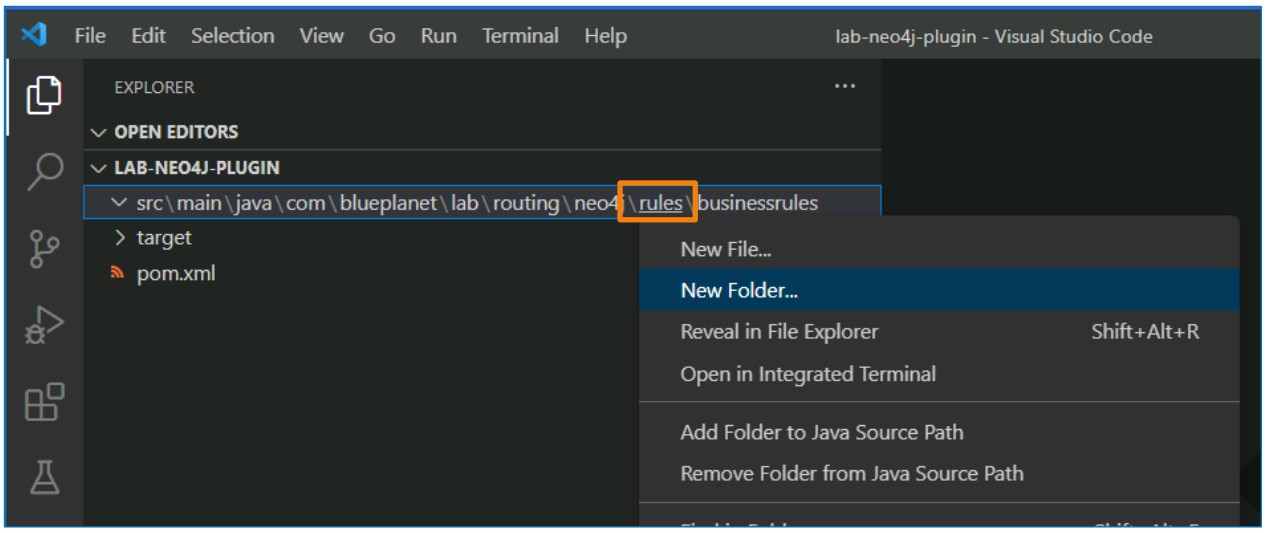


12. Next you will create the package structure for the project. Right-click the **rules** directory and select **New Folder…** from the menu.

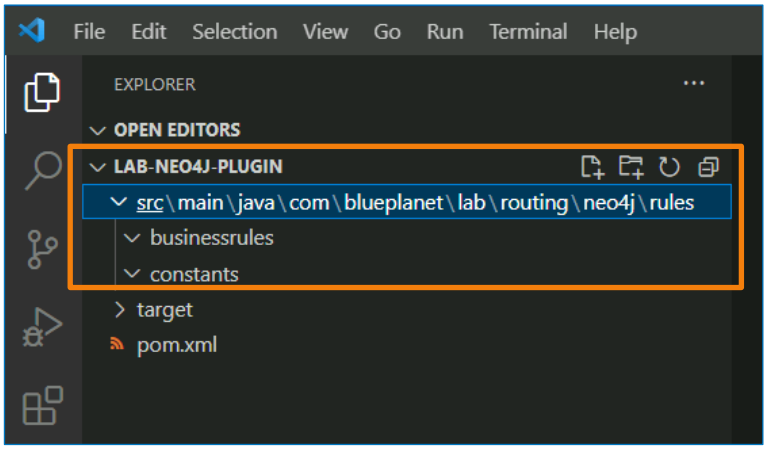


13. Enter **businessrules** and press the **Enter** key.

14. Right-click the **rules** directory again and select **New Folder…** from the menu to create another sub-directory.



15. Enter **constants** and press the **Enter** key. Your folder structure should look the same as in the following figure.



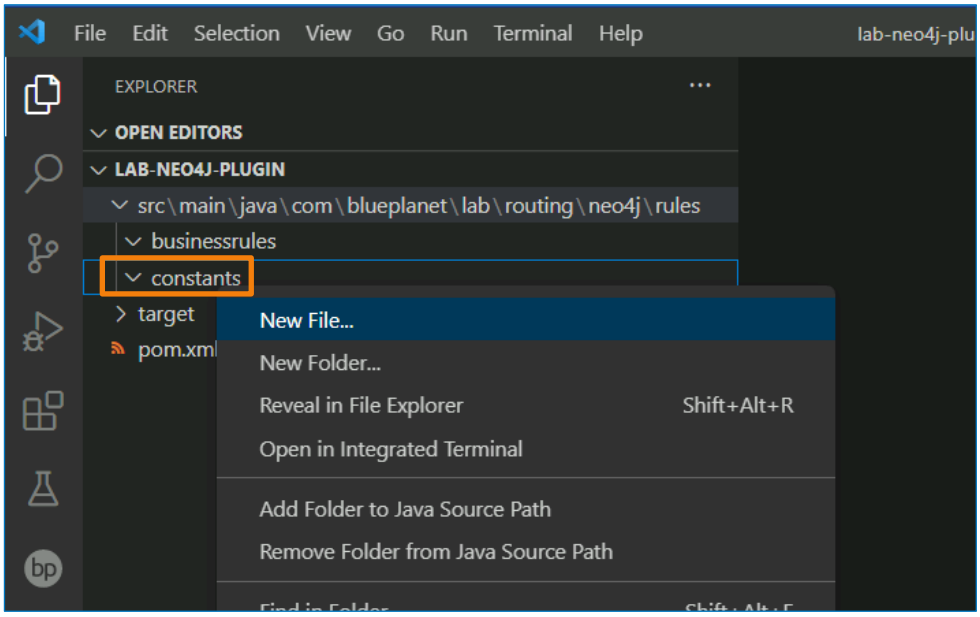
NOTE: If you do not currently see the target folder, ignore it for now. The target folder will appear the first time you build the project.

16. Your project has now been successfully set up. Leave the VS Code window open for the next task.

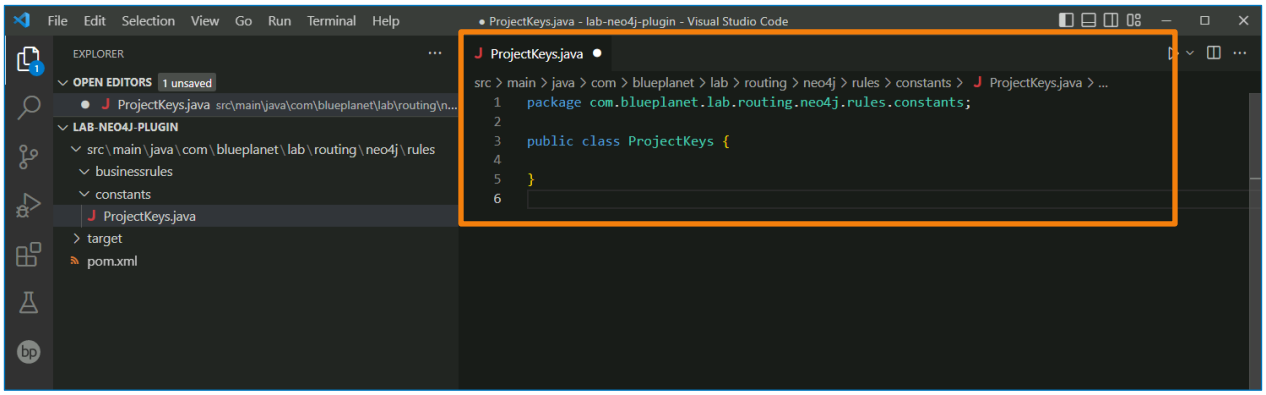
Task 2: Build and Deploy the Neo4j Plugin Project

In this task, you create the project files, enter the appropriate code, and then build and deploy the project to the BPI server in your lab infrastructure.

1. In the Explorer pane of VS Code, right-click the folder **constants** and select **New File…** to create a file named **ProjectKeys.java**.



2. Enter **ProjectKeys.java** for the file name and press the **Enter** key. The file is now created and populated with the package and class definitions.



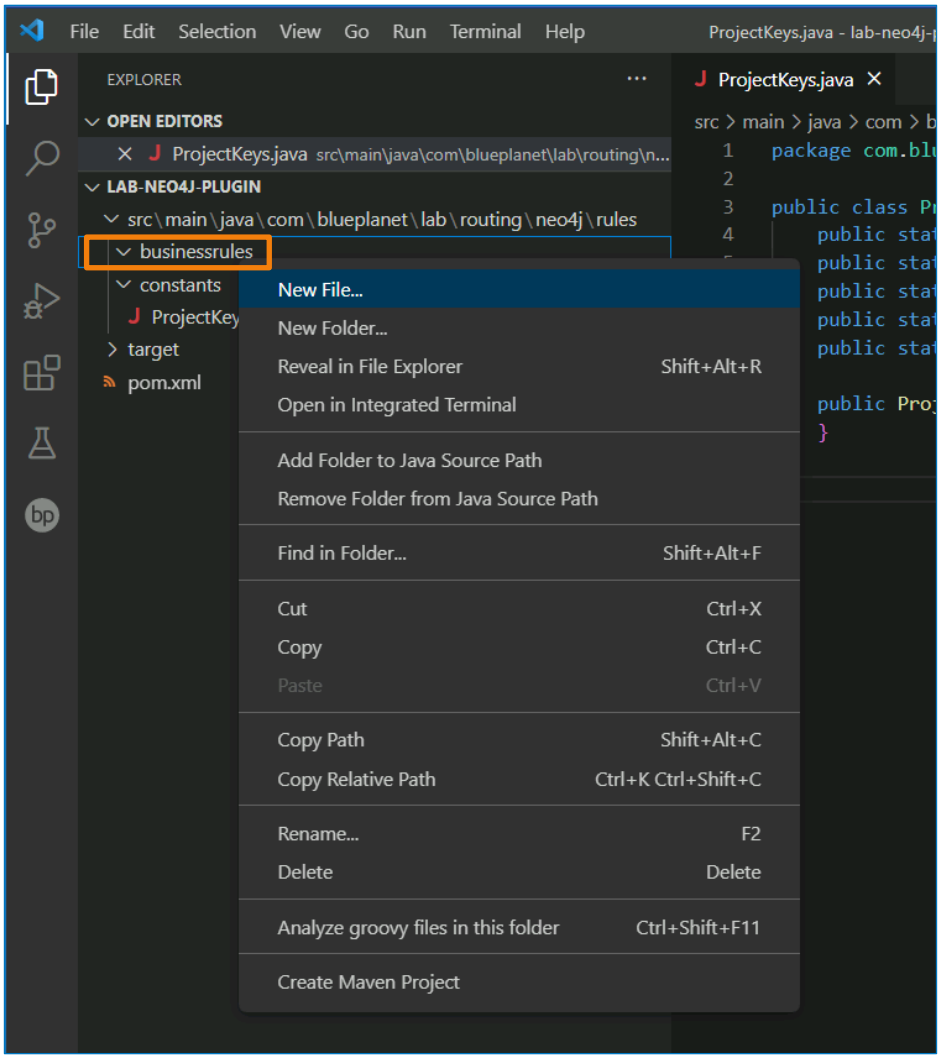
3. ProjectKeys.java is the place to define specific object property names used in the project. Enter the following code in the class so that the complete class code appears as the following.

```
package com.blueplanet.lab.routing.neo4j.rules.constants;


public class ProjectKeys {
    public static final String bandwidth = "bandwidth";
    public static final String siteType = "siteType";
    public static final String category = "category";
```

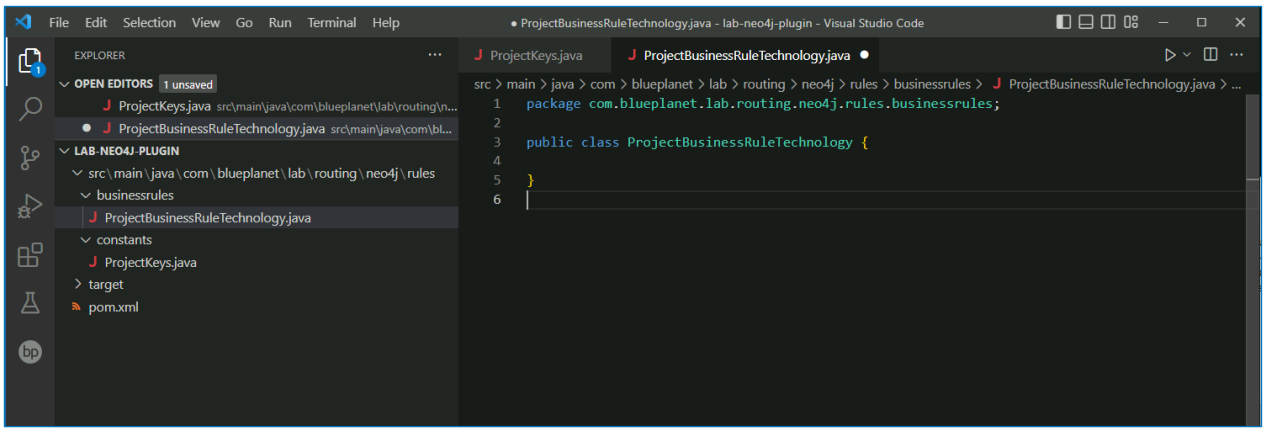
```
    public static final String connectionType = "connectionType";
    public static final String precomputedTypeName =
    "precomputedtypename";


    public ProjectKeys() {
    }
}
```

NOTE: In this exercise, you use only the precomputedTypeName property, other properties are here as an example of other keys you might use.

4. Press **CTRL+S** to save the changes to the file.
5. Next, in the project tree, right-click the folder **businessrules** and select **New File…** to create a new file.



6. Enter **ProjectBusinessRuleTechnology.java** for the file name and press the **Enter** key. The file opens with the basic package and class definitions.



7. In the file, enter the import statements immediately below the package statement. Next, extend the class by adding the “extends BusinessRuleTechnology”. This way your new class will inherit from the existing BusinessRuleTechnology class.

```
package com.blueplanet.lab.routing.neo4j.rules.businessrules;

import com.blueplanet.lab.routing.neo4j.rules.constants.ProjectKeys;
import com.blueplanet.routing.neo4j.constants.Labels;
import com.blueplanet.routing.neo4j.core.AnalysisInformationEnriched;
import com.blueplanet.routing.neo4j.core.BusinessRuleMap;
import com.blueplanet.routing.neo4j.core.RuleWeight;
import com.blueplanet.routing.neo4j.io.RoutingException;
import com.blueplanet.routing.neo4j.rules.BusinessRuleTechnology;
import org.neo4j.graphdb.Node;

import java.util.Objects;

public class ProjectBusinessRuleTechnology extends
   BusinessRuleTechnology {
```

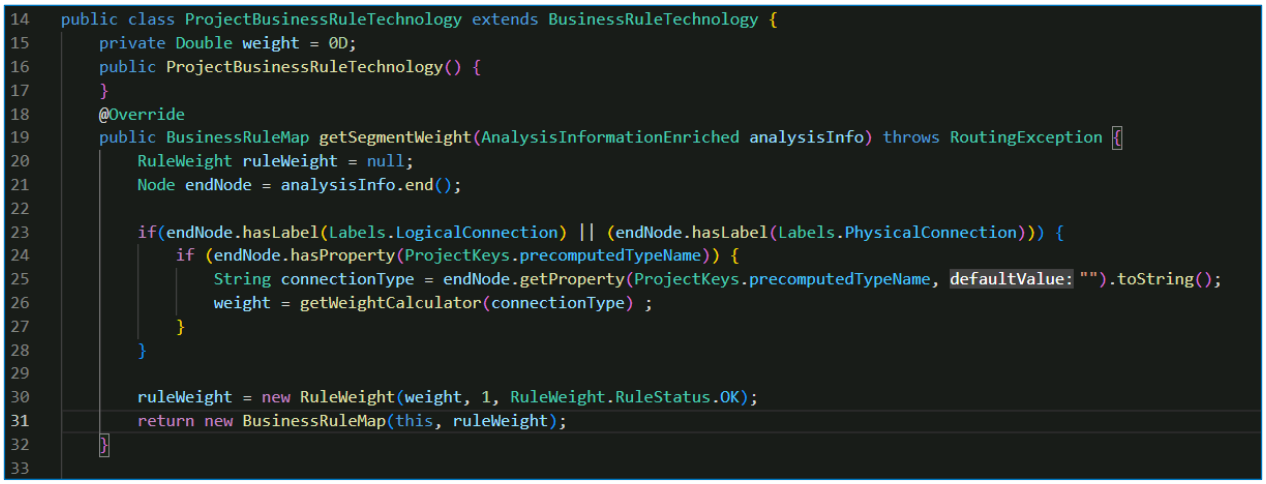
8. Next, in the public class, add the following RuleWeight and Node objects that you will use later in this class.

```
 private Double weight = 0D;
 public ProjectBusinessRuleTechnology() {
 }
```

9. Now implement the method getSegmentWeight by overriding the method with the same name as the class **BusinessRuleTechnology**. This method will provide different weights to the path segments, depending on which technology is used, basic Cable Fiber or OTN/OTU1. Enter the following code inside the class definition.

```
@Override
    public BusinessRuleMap getSegmentWeight(AnalysisInformationEnriched
    analysisInfo) throws RoutingException {
        RuleWeight ruleWeight = null;
        Node endNode = analysisInfo.end();

        if(endNode.hasLabel(Labels.LogicalConnection) ||
    (endNode.hasLabel(Labels.PhysicalConnection))) {
            if (endNode.hasProperty(ProjectKeys.precomputedTypeName)) {
                String connectionType =
    endNode.getProperty(ProjectKeys.precomputedTypeName, "").toString();
                weight = getWeightCalculator(connectionType);
           }
        }

        ruleWeight = new RuleWeight(weight, 1,
    RuleWeight.RuleStatus.OK);
        return new BusinessRuleMap(this, ruleWeight);
    }
```

10. Your class should now look the same as in the following figure.



11. Next, add the method **getWeightCalculator**, which is called from getSegmentWeight, and that will return different weights depending on the underlying connection type. If the connection type is “Cable Fiber” then the weight will be returned as 0, while if the connection type is “OTU1”, the weight will be returned as 15. In other cases, the method will return a Double value of 10. A lower number makes the segment more preferred. In this code, Cable Fiber is preferred over OTU1. Enter the following code below inside the class, below the getSegmentWeight method.

```
public Double getWeightCalculator(String connectionType){
    if (Objects.equals(connectionType, "Cable Fiber")){
        return 0D;
    } else if(Objects.equals(connectionType, "OTU1")){
        return 15D;
    }
    return 10D;
}
```

12. The completed class should look the same as in the following figure.

```
14  public class ProjectBusinessRuleTechnology extends BusinessRuleTechnology {
15      private Double weight = 0D;
16      public ProjectBusinessRuleTechnology() {
17      }
18      @Override
19      public BusinessRuleMap getSegmentWeight(AnalysisInformationEnriched analysisInfo) throws RoutingException {
20          RuleWeight ruleWeight = null;
21          Node endNode = analysisInfo.end();
22
23          if(endNode.hasLabel(Labels.LogicalConnection) || (endNode.hasLabel(Labels.PhysicalConnection))) {
24              if (endNode.hasProperty(ProjectKeys.precomputedTypeName)) {
25                  String connectionType = endNode.getProperty(ProjectKeys.precomputedTypeName, "").toString();
26                  weight = getWeightCalculator(connectionType) ;
27              }
28          }
29
30          ruleWeight = new RuleWeight(weight, 1, RuleWeight.RuleStatus.OK);
31          return new BusinessRuleMap(this, ruleWeight);
32      }
33      public Double getWeightCalculator(String connectionType){
34          if (Objects.equals(connectionType, "Cable Fiber")){
35              return 0D;
36          } else if(Objects.equals(connectionType, "OTU1")){
37              return 15D;
38          }
39          return 10D;
40      }
41  }
```

13. Press **CTRL+S** to save the changes.
14. Return to the PowerShell session or open a new one. Change the directory to your project location (C:\Users\student\Workspaces\Lab5\lab-neo4j-plugin) and run the **mvn clean install** command to build the jar file (artifact).

```
PS C:\Users\student\Workspaces\Lab5\lab-neo4j-plugin> mvn clean install
[INFO] Scanning for projects...
[INFO]
```

```
<… output omitted …>
[INFO] Building jar: C:\Users\student\Workspaces\Lab5\lab-neo4j-
   plugin\target\lab-neo4j-plugin-1.0-SNAPSHOT.jar
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ lab-
   neo4j-plugin ---
[INFO] Installing C:\Users\student\Workspaces\Lab5\lab-neo4j-
   plugin\target\lab-neo4j-plugin-1.0-SNAPSHOT.jar to
   C:\Users\student\.m2\repository\com\blueplanet\lab\routing\neo4j\rul
   es\lab-neo4j-plugin\1.0-SNAPSHOT\lab-neo4j-plugin-1.0-SNAPSHOT.jar
[INFO] Installing C:\Users\student\Workspaces\Lab5\lab-neo4j-
   plugin\pom.xml to
   C:\Users\student\.m2\repository\com\blueplanet\lab\routing\neo4j\rul
   es\lab-neo4j-plugin\1.0-SNAPSHOT\lab-neo4j-plugin-1.0-SNAPSHOT.pom
[INFO] ------------------------------------------------------------------
   -------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------
   -------
[INFO] Total time:  7.294 s
[INFO] Finished at: 2023-01-06T04:30:43-12:00
[INFO] ------------------------------------------------------------------
   -------
```

NOTE: If the result of the build is not “BUILD SUCCESS”, you might have introduced typos in the code. Carefully read the error message and try to find the errors.

15. Verify that the JAR file has been created in the **target** folder.

```
PS C:\Users\student\Workspaces\Lab5\lab-neo4j-plugin> dir .\target\


    Directory: C:\Users\student\Workspaces\Lab5\lab-neo4j-plugin\target


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         1/6/2023   4:30 AM                classes
d-----         1/6/2023   4:30 AM                generated-sources
d-----         1/6/2023   4:30 AM                maven-archiver
d-----         1/6/2023   4:30 AM                maven-status
```

```
-a----         1/6/2023   4:30 AM           5447 lab-neo4j-plugin-1.0-
   SNAPSHOT.jar
```

16. Finally, deploy the plugin to the server using the **deploy_project.bat** script. Run the script from the folder C:\Users\student\Workspaces\Lab5.

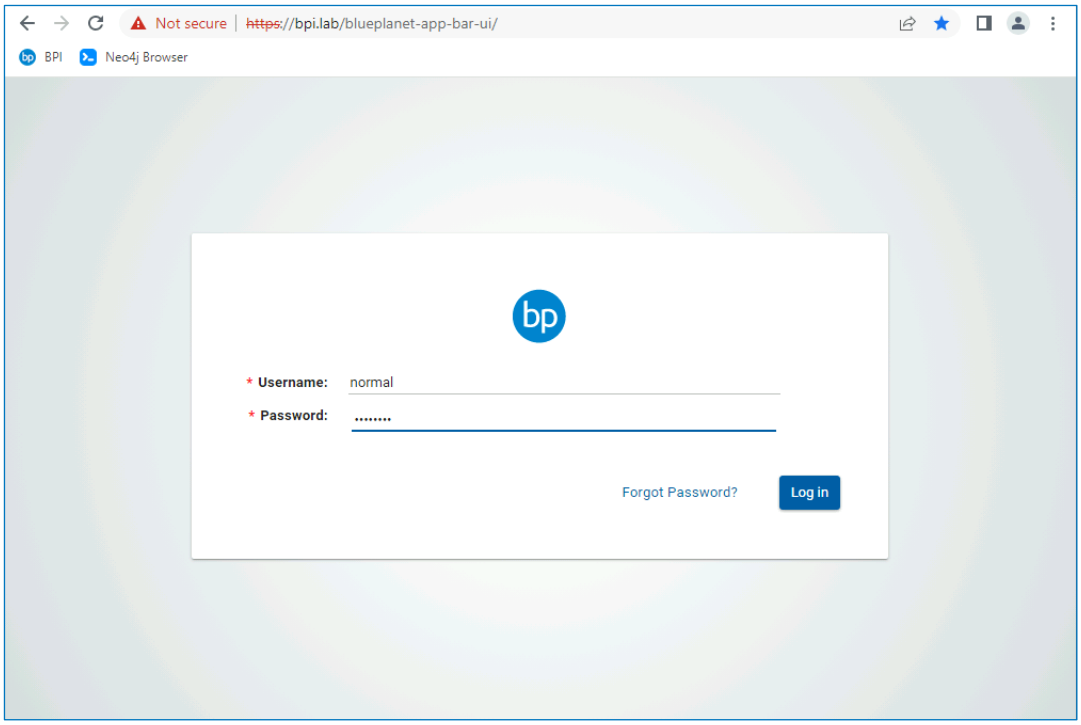
```
PS C:\Users\student\Workspaces\lab5\lab-neo4j-plugin> cd ..
PS C:\Users\student\Workspaces\Lab5> .\deploy_project.bat
"Copying plugin to the server."
lab-neo4j-plugin-1.0-SNAP | 5 kB |   5.3 kB/s | ETA: 00:00:00 | 100%
"Deploying custartifacts."
pod "custartifacts-0" deleted
"Restarting Neo4j container... please wait"
pod "neo4j-0" deleted
Completed.
PS C:\Users\student\Workspaces\Lab5>
```

17. Your Neo4j plugin is now deployed to the BPI instance in your lab.

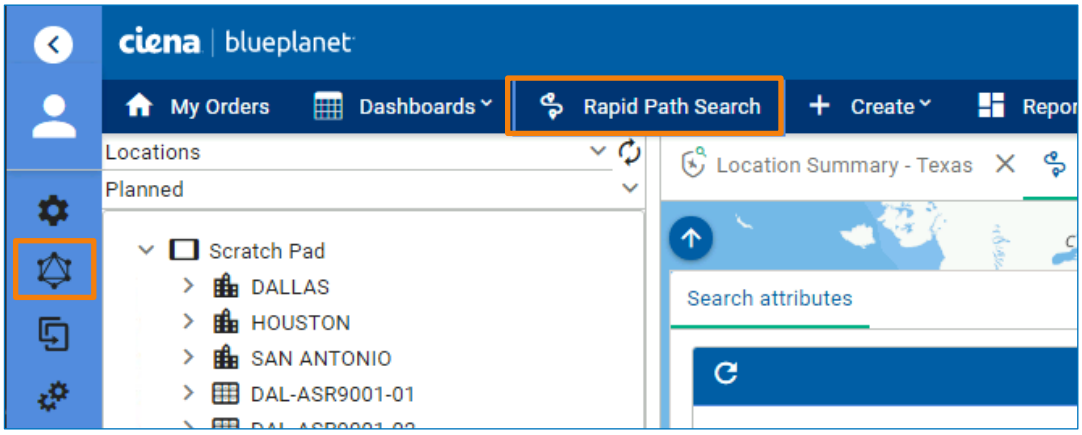
Task 3: Configure the Business Rule to Use the New Class

In this task, you first test the default rapid path search behavior with the default class (BusinessRuleTechnology) applied to the Technology rule. Then you reconfigure the Technology rule to use your newly deployed class (ProjectBusinessRuleTechnology) and then repeat the same rapid path search. The resulting path should now prefer a different underlying technology. Finally, you create an end-to-end Ethernet connection between the selected routers using RPS.

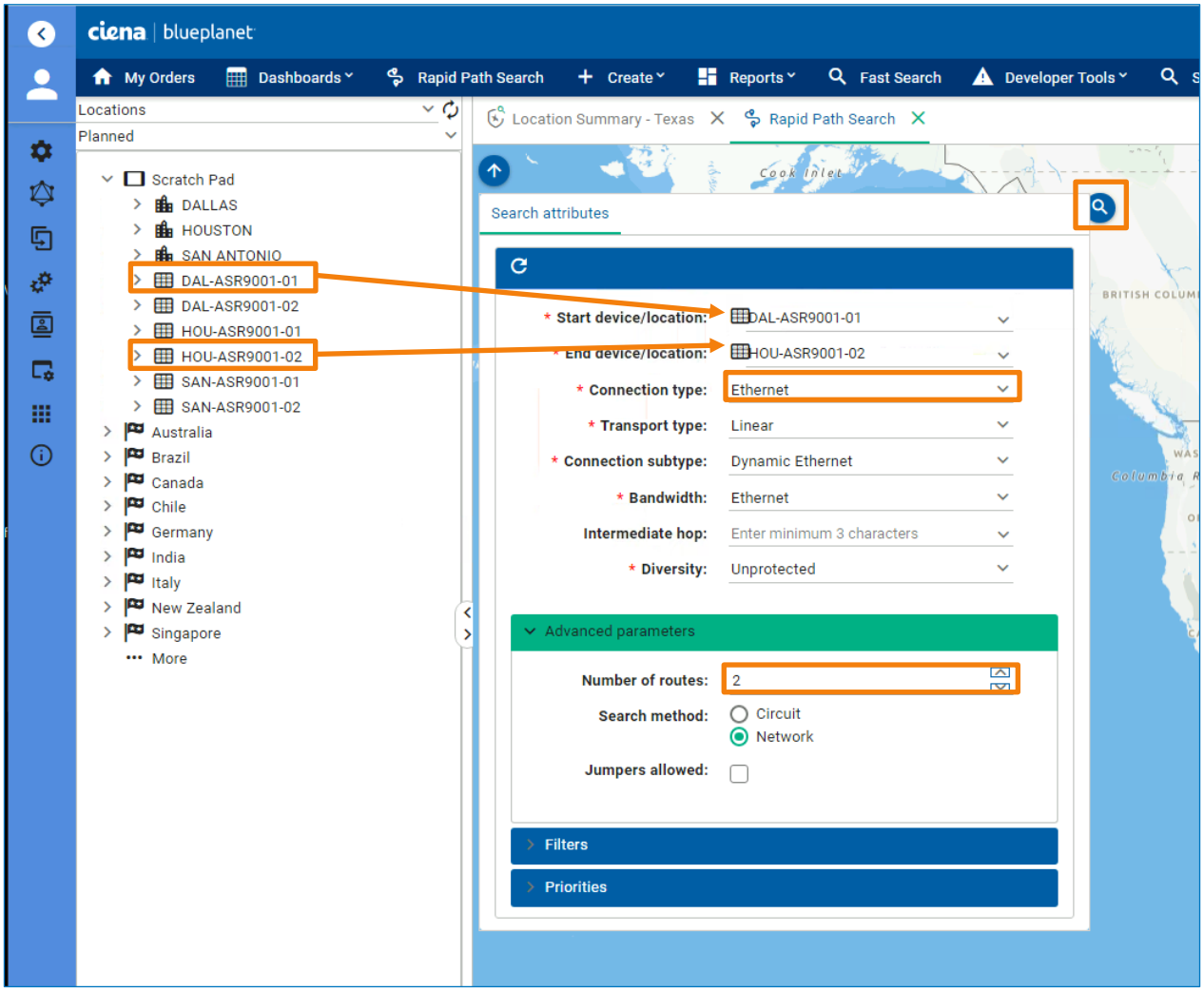
1. To log in to the Blue Planet Inventory UI, from your Student PC, open a Chrome browser session and go to *https://bpi.lab* or click on the BPI bookmark. Use the credentials **normal/12345678** to log in.



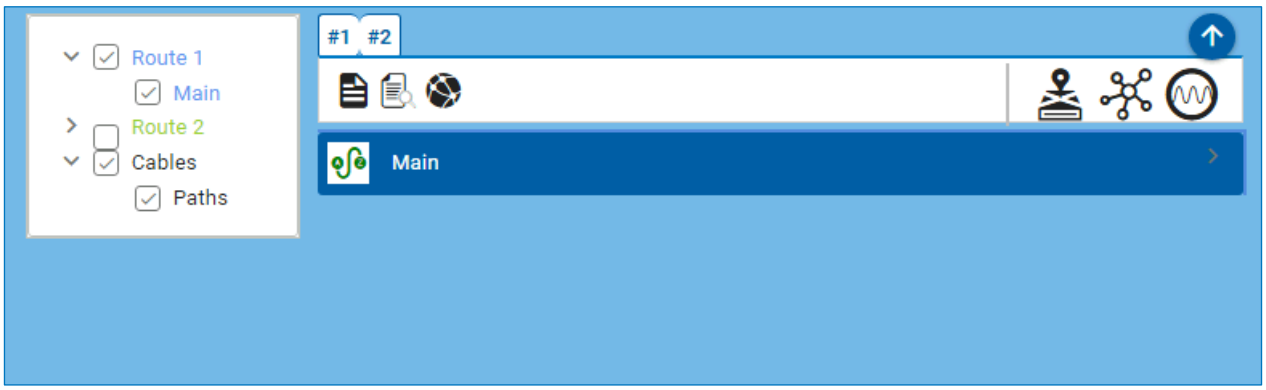
2. From the application bar on the left, select **Inventory**, then expand and select **Rapid Path Search** from the main menu.



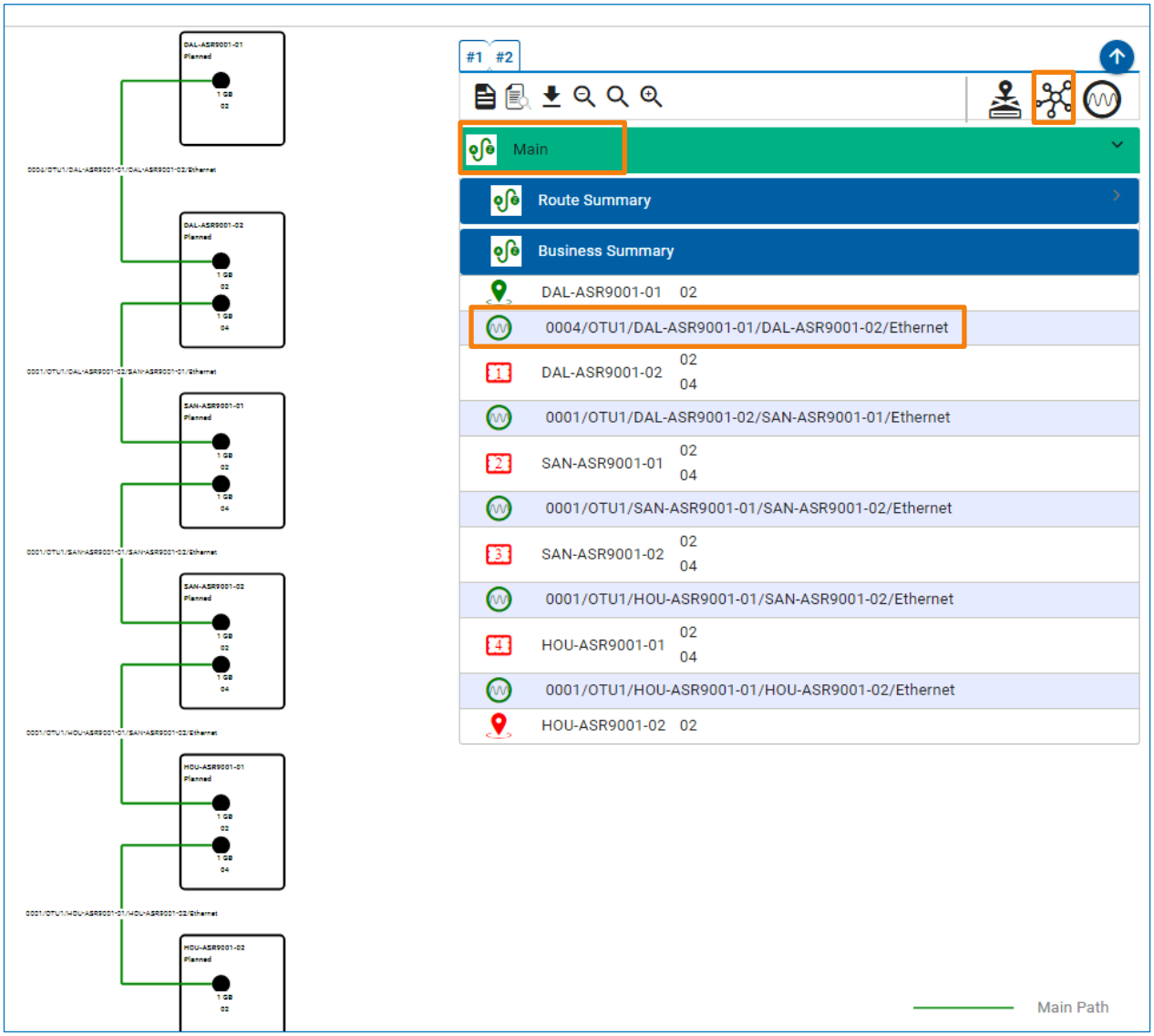
3. From the Scratch Pad, drag and drop **DAL-ASR9001-01** to the **Start device/location**. Drag **HOU-ASR9001-02** and drop it in the **End device/location**. For connection type, choose **Ethernet**. Expand the Advanced parameters and put **2** in the **Number of routes** field. Now click the **search** button (blue magnifying glass icon) to start the search.



4. The result shows two routes found (#1 and #2).

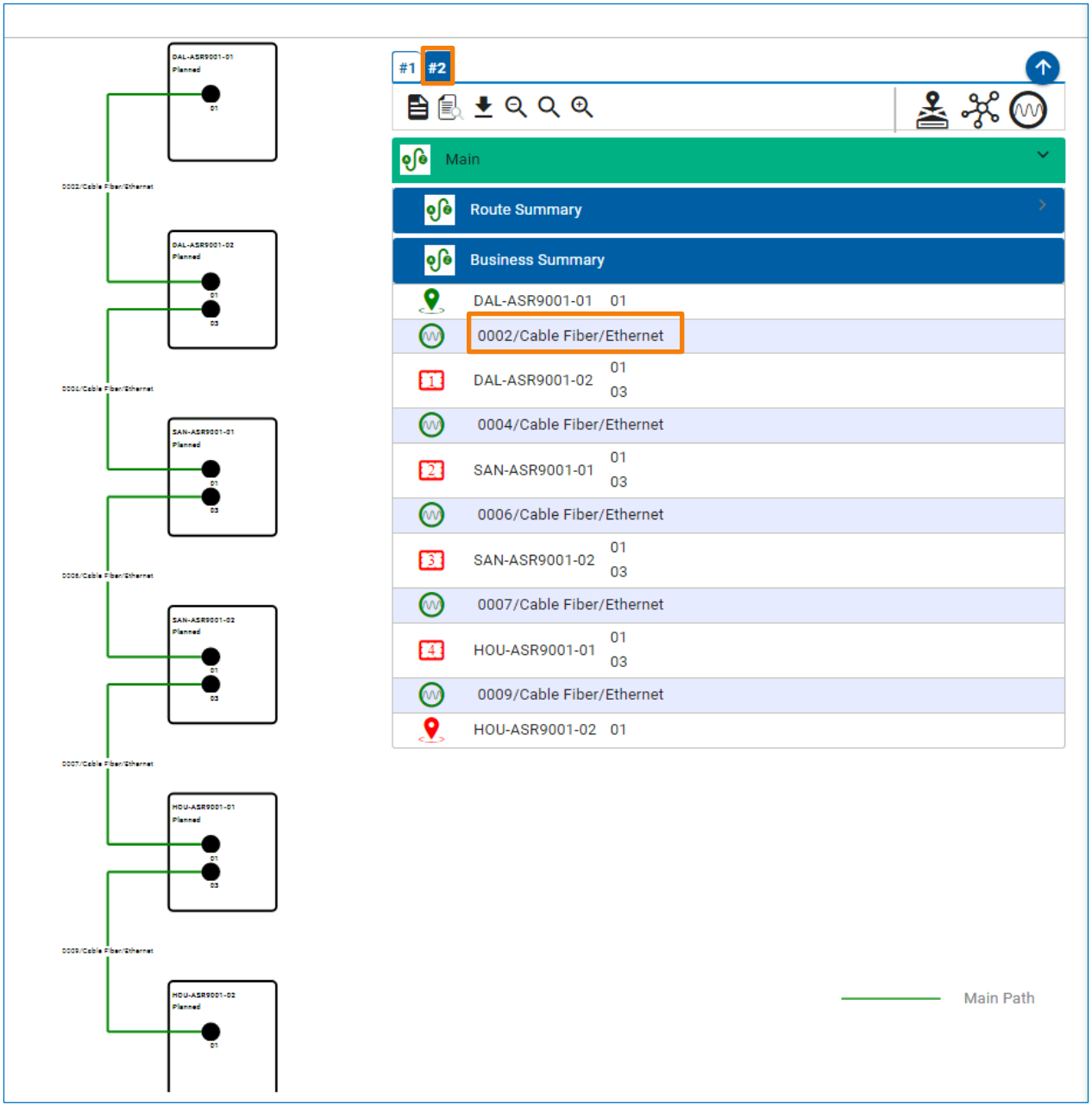


5. Click **Main** under route 1 (primary route), then the **Diagram** icon in the top right corner (middle icon). The main route expands and shows all the intermediate hops and connections along the way. Note that all the connections use the OTU1 technology because it is preferred to Cable Fiber connections.



NOTE: If the search is conducted with the Number of routes equal to 1, then only the route with OTU1 connections would show up in the result, as the best path (lowest weight).

6. Now click the route 2 button (#2 on the top left) and observe that the secondary path uses Cable Fiber connections.



NOTE: In your pod, the connection numbering may vary.

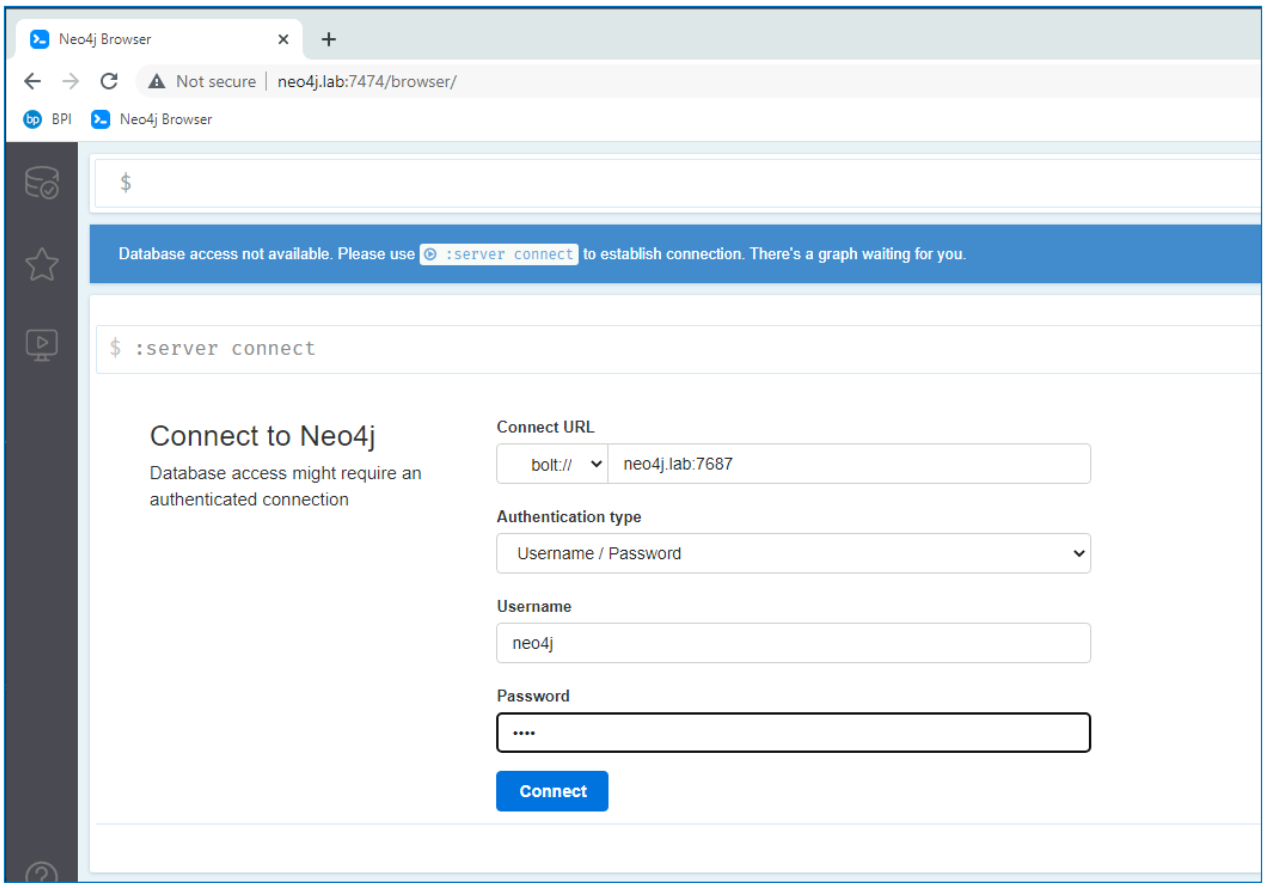
7. Now associate the new class, from the plugin that you deployed to the Technology business rule. You will do this by changing the **className** parameter for the business rule directly in the Neo4j database using a Cypher query. Open a new tab in the browser and point it to **http://neo4j.lab:7474/browser/**. When the page loads use the following access information and press **Connect**.

   Connect URL: **bolt:// neo4j.lab:7687**

   Authentication type: **Username/Password**

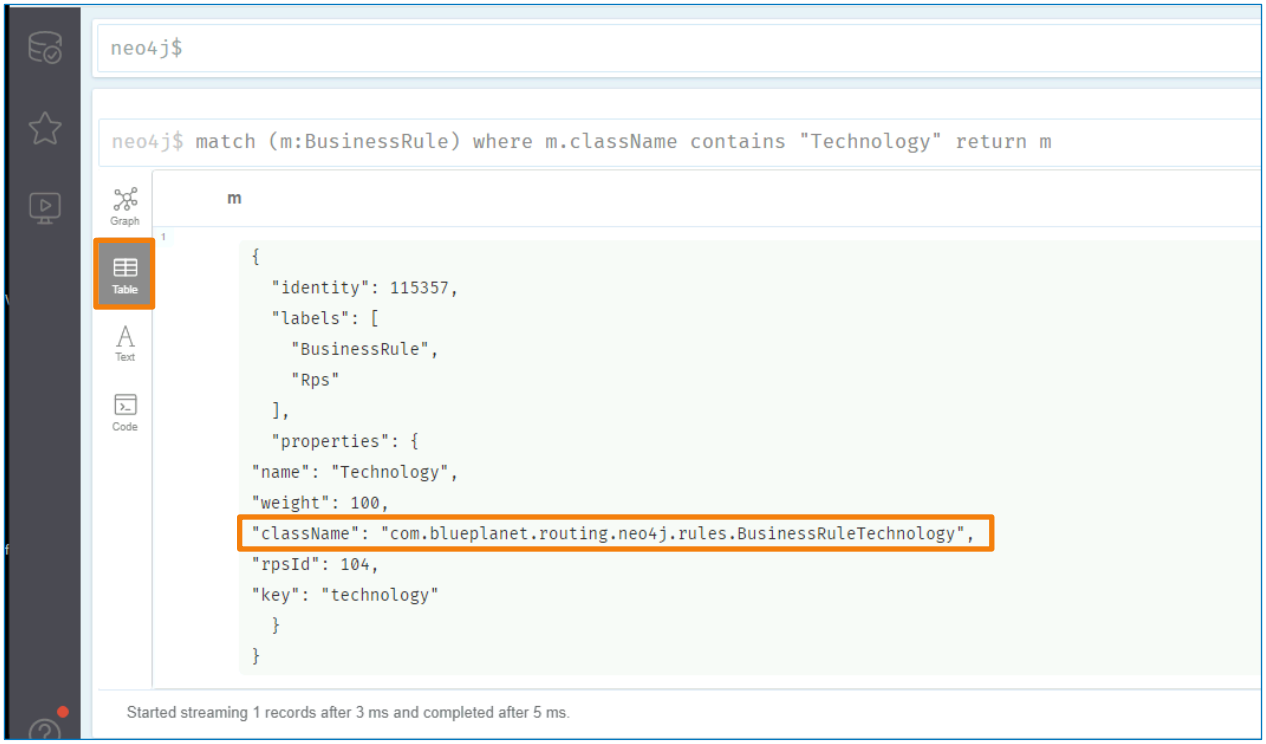
   Username: **neo4j**

   Password: **drni**



8. Run the following query to display the configuration of the Technology business rule.

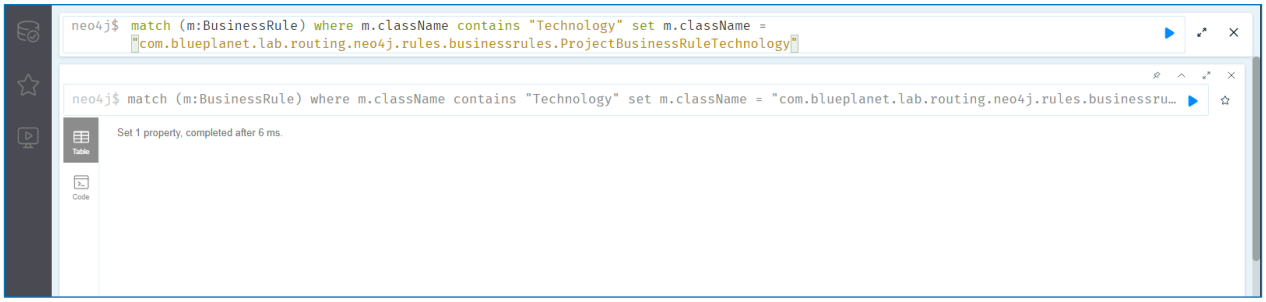
```
match (m:BusinessRule) where m.className contains "Technology" return m
```



Click **Table** and observe how the className property points to the original class **com.blueplanet.routing.neo4j.rules.BusinessRuleTechnology**. For your plugin to have an effect on the rule, you have to specify the class from your plugin in the business rule configuration.

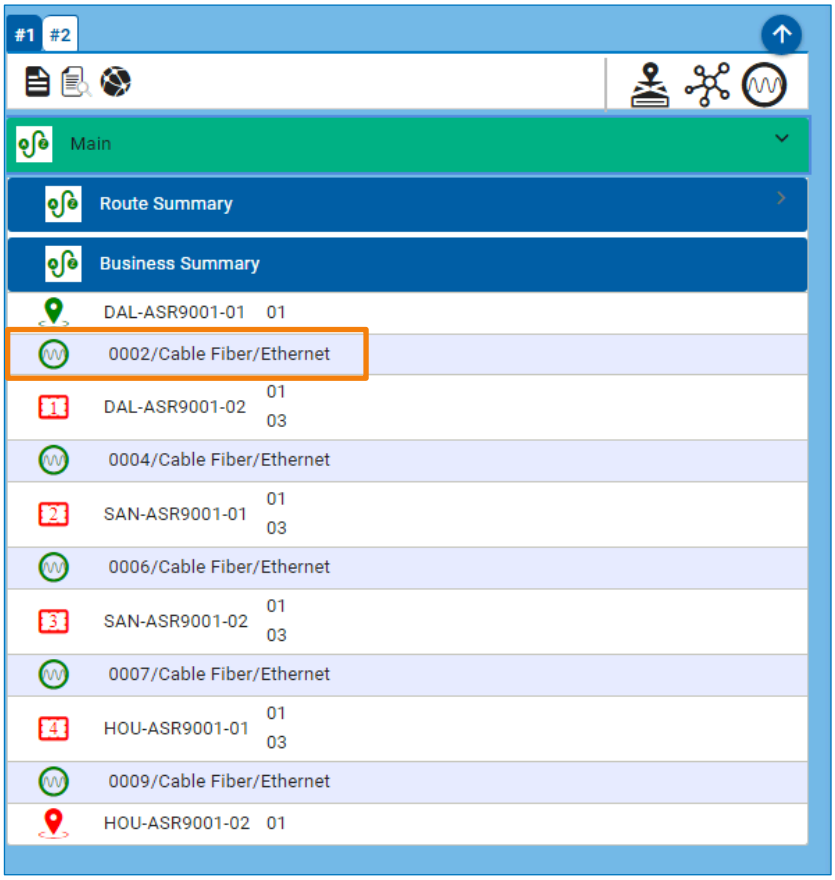
9. Execute the following Cypher query to modify the className of the Technology business rule. The new class name will point to the class from your project.

```
match (m:BusinessRule) where m.className contains "Technology" set
   m.className =
   "com.blueplanet.lab.routing.neo4j.rules.businessrules.ProjectBusines
   sRuleTechnology"
```

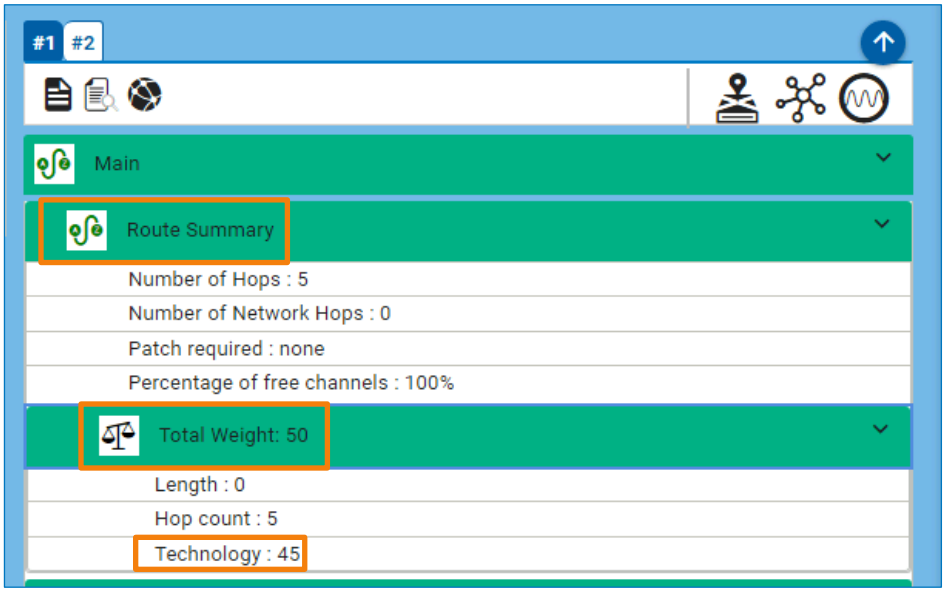


10. Execute the first query once again, to verify that the **className** has been changed.

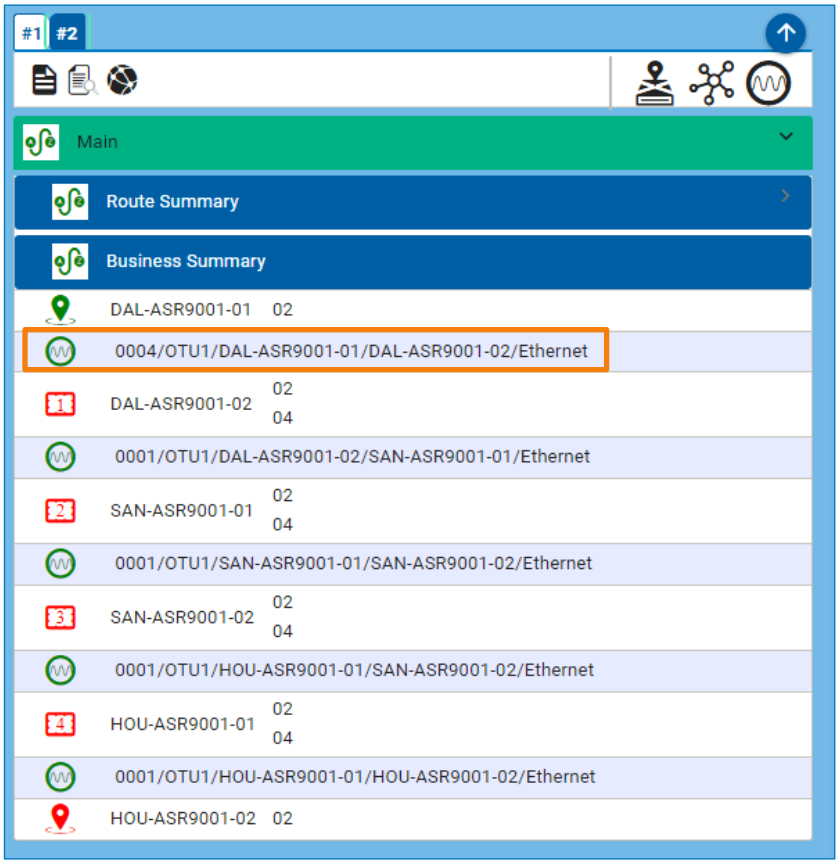
11. Return to the BPI server browser tab with RPS and run the search again with the same search parameters. Press the search button. This time route 1 returns with connections over Cable Fiber, meaning that your code influenced the route selection, as expected.



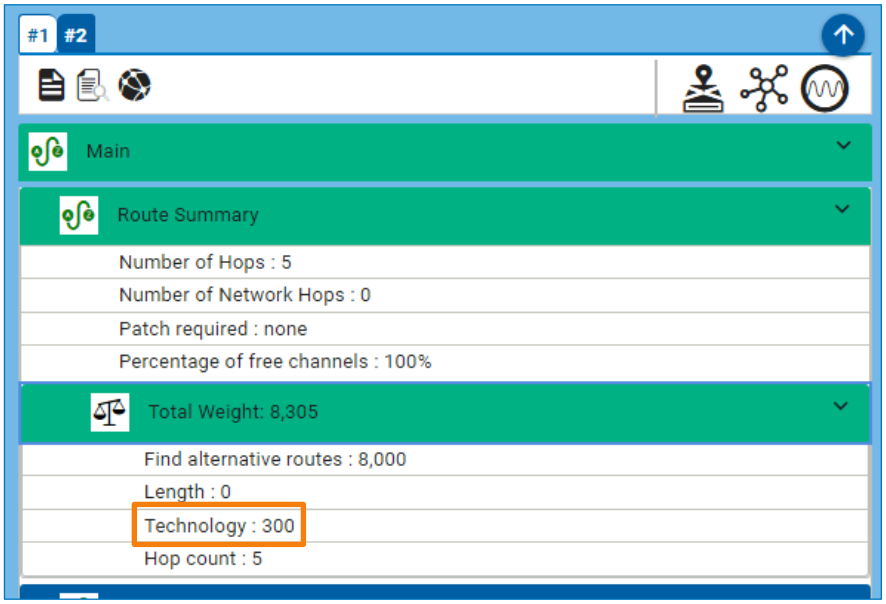
12. While in route 1, click **Route Summary** and then **Total Weight**. Note the Technology weight for route 1 is 45.



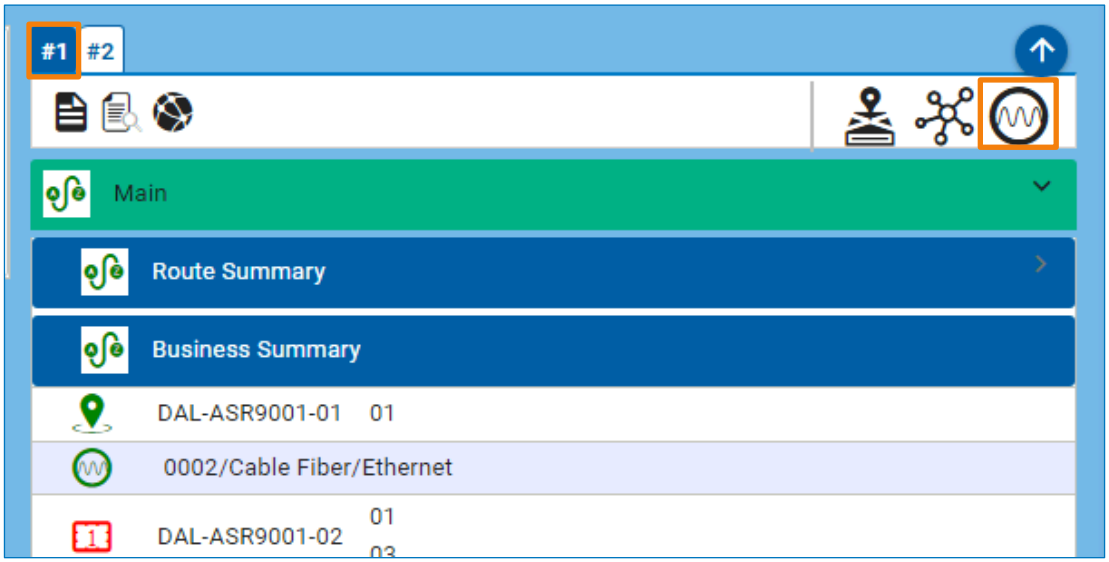
13. In the top left corner, switch to route 2 by clicking **#2**. Note how the secondary route now uses the OTU1 connections. You have successfully prioritized the election of the route using Cable Fiber connections over routes using OTU1 connections with the Neo4j plugin code.



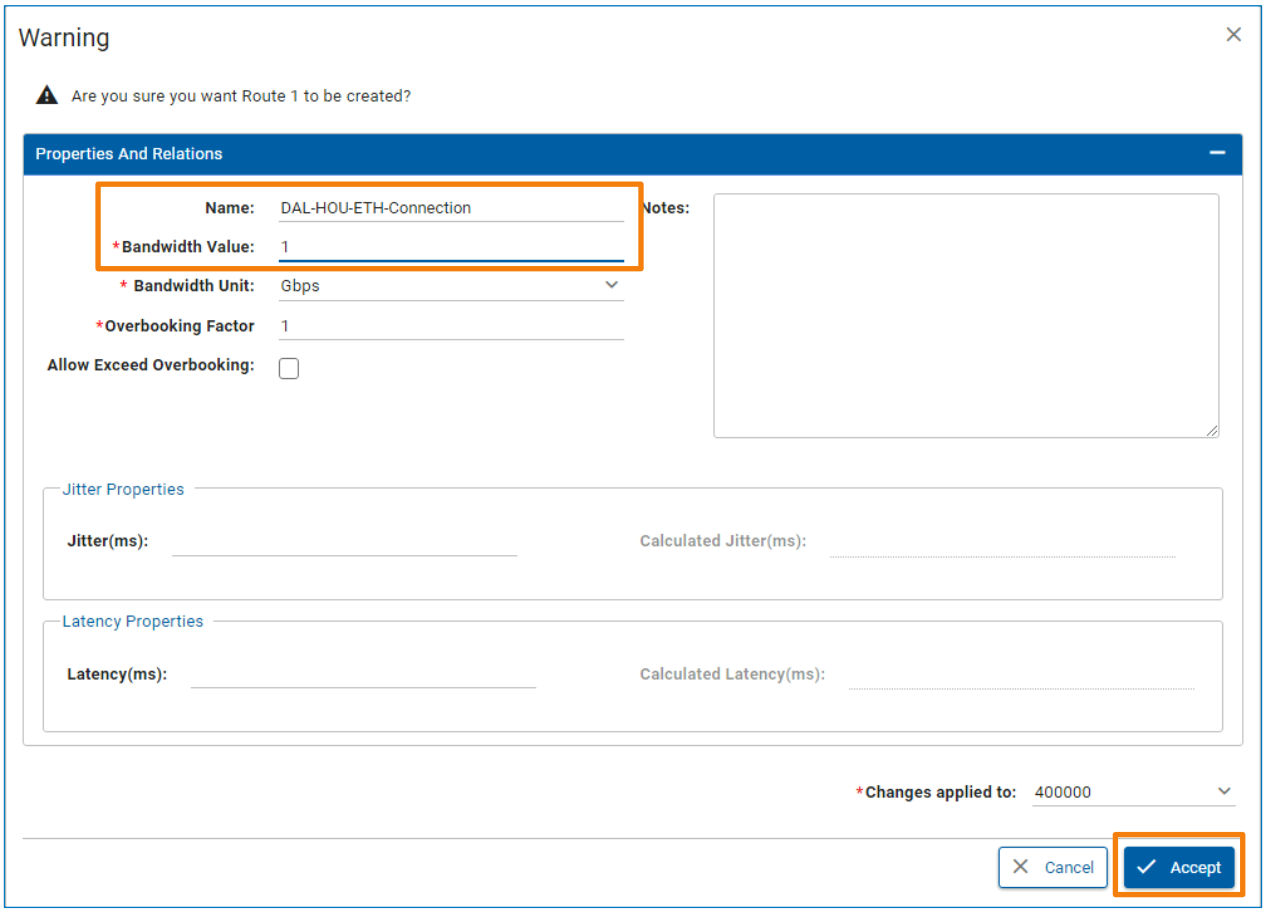
14. While in route 2, click **Route Summary** and then **Total Weight**. Note the Technology weight for route 2 is 300.



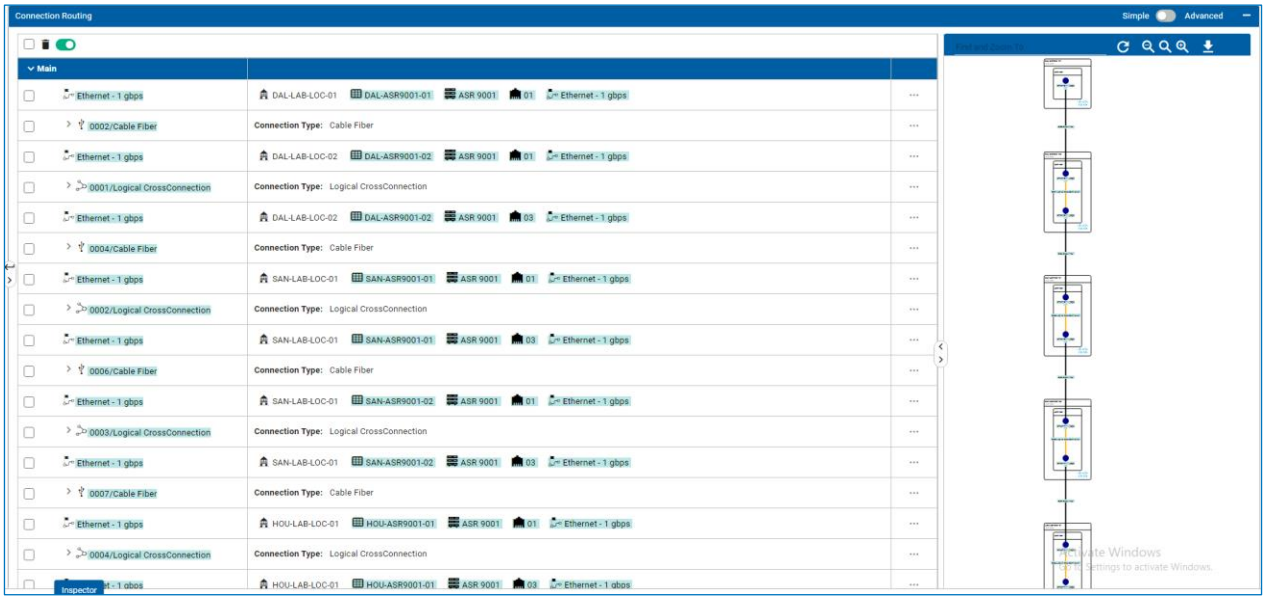
15. Now create the suggested Ethernet connections between DAL-ASR9001-01 and HOU-ASR90001-02. Switch back to route 1 by clicking **#1**, then click the **Apply** button in the top right (last icon).



16. The create connection page pops up. Enter **DAL-HOU-ETH-Connection** under Name, enter **1** under Bandwidth Value and click **Accept**.



17. The connection summary window opens after the connections have been successfully created. Observe the summary page. Scroll down to see the details of the connection in the **Connection Routing** section.



End of Lab

Lab 6: Data Ingestion Framework

Objectives

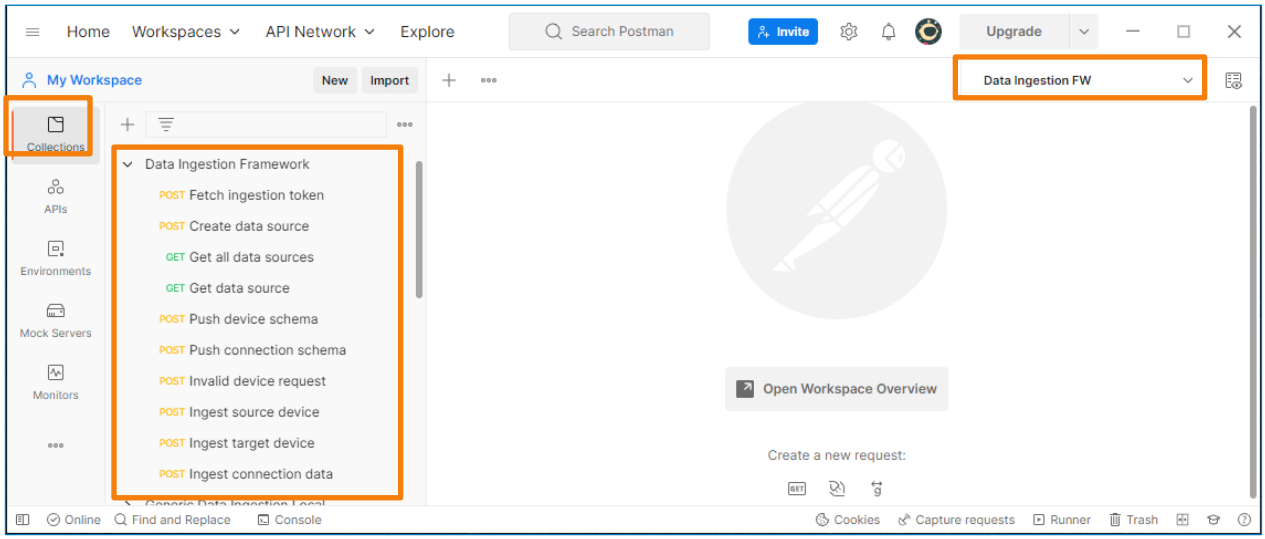
- Create a data source for the Ingestion Framework which will be used to create devices and connections in BPI
- Create a Drools rule to add a device to the specified location
- Create Drools Rule for creating physical connections between devices
- Apply JSON schema validation for events
- Create devices and cable fiber connections between devices
- Inspect applied changes in BPI UI and in Neo4J UI

Task 1: Configure Data Source

In this task, you will create a data source configuration which will be applied to Data Ingestion Framework. This data source will enable sending events through REST API to multiple endpoints. Each endpoint will be configured in a later task to accept different event message formats. You will use the Postman tool with a prepared collection to send REST requests.

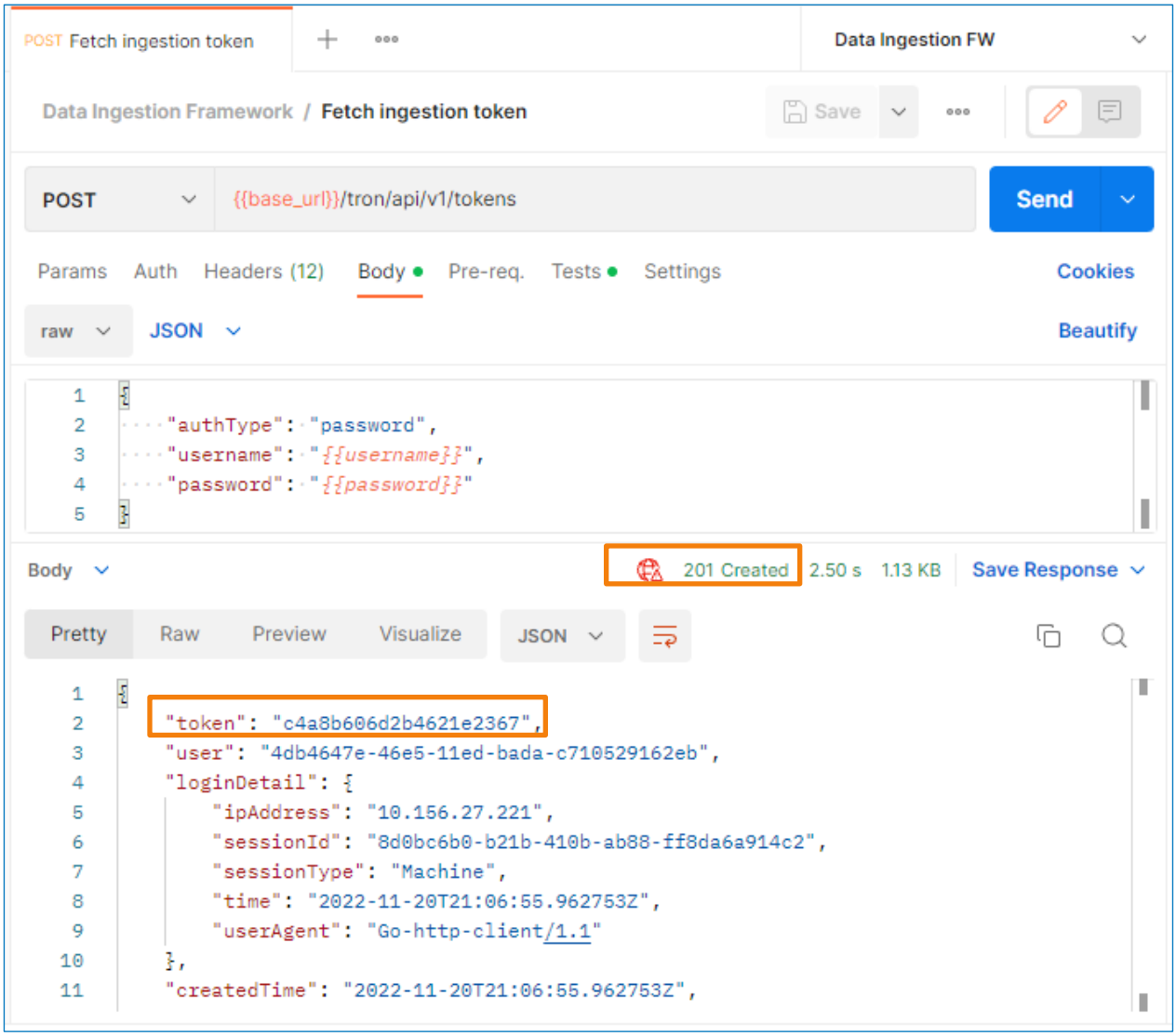
1. From your student PC, open Postman.

2. In Postman find and expand the collection **Data Ingestion Framework.**
3. Near the top right side of the Postman window, you should be able to see a dropdown menu that allows you to select an environment. Select **Data Ingestion FW** from the dropdown menu.



4. Inside the collection, find and open the **Fetch ingestion token** request. Click the **Send** button to send the request.

The request should return a response with the status **201 Created.** In the response message, you will get a token that will be used as the **Bearer Token** in subsequent requests directed to the Data Ingestion Framework.



NOTE: Fetch ingestion token request is configured to automatically store the token in environment variables. Other requests in this collection are configured to use this token so that you do not need to worry about configuring them to use the token.

5. From the collection, open the **Create data source** request. The body of the request is currently empty. You need to add the data source configuration to the request body. The data source will be used to ingest the device data as well as the physical connection data. To be able to see the data through the BPI UI, you will configure the data source to inject data to perspective 2(Plan data). The configuration for the data source should look like this:

```
{
    "name": "LabDataIngestionFWDataSource",
        "ingestionConfig": {
```

```
        "ingestionURL": "kafka0:9092?topics=data-ingestion-lab",
        "groupId": "consumer_test_group-test2",
        "authenticationType": "Kafka",
        "contentType": "application/json"
    },
    "retryConfig": {
        "ingestionURL": "localhost:9092?topics=retry-data-ingestion-
   lab",
        "groupId": "retry_test_group-test2",
        "authenticationType": "Kafka",
        "contentType": "application/json",
        "timeoutSec": 5,
        "retryCount": 3
    },
    "rules": [
        "Lab Data Ingestion - Create Device",
        "Lab Data Ingestion - Create Cable Fiber Connection"
    ],
    "perspectives": [ 2 ],
    "operations": [
        "createDevice",
        "createConnection"
    ],
    "ignoreCompatibilities": true,
    "createActivityCallout":"com.blueplanet.inventory.ingestionFramework
   .callouts.CreateActivityCallout.createActivity"
}
```

The configuration should contain:

a. **name:** Unique name for the data source. Data Ingestion Framework will throw an error if the name already exists.
b. **ingestionConfig:** Configuration for Kafka topic to which you can send event messages.
c. **retryConfig:** Configuration for Kafka retry topic.
d. **rules:** List of Drools rules which will be used by this data source when processing events. Currently, these rules do not exist yet. You will create these rules in later tasks.
e. **perspectives:** List of perspectives into which data will be ingested.

f. **operations:** List of operations which can be executed using this data source. This will create a REST endpoint for each operation. Each operation can have separate validation when sending events through REST API.

g. **createActivityCallout:** When applying changes to the BPI, we need to create an Activity which will be related to the changes. In Ingestion Data Framework, creating Activity is done outside of Drools rule definition. With this property, we can provide a custom callout that will handle the Activity creation. When this property is not configured, Data Ingestion Framework will provide a default behavior that will handle the Activity creation. When ingesting data into the Planned perspective, you also need to relate the changes to the order which is also done outside of Drools rule definition. This is not done in the default callback for activity creation but Data Ingestion Framework exposes custom callout implementation which can be used to relate changes to the order. This will require an order reference ID to be sent in the event message which will be explained later.

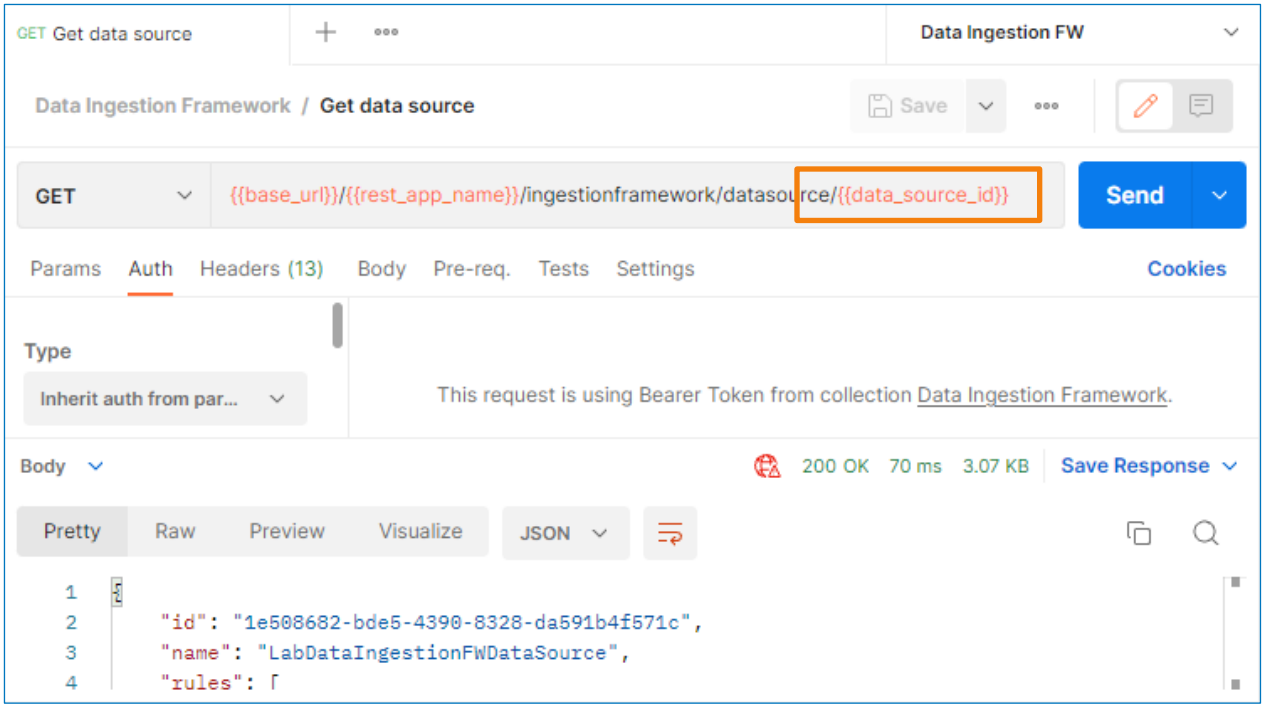
6. Copy the configuration from the previous step to the body of the **Create data source** request and send the request. The response should be returned with the status **200 Created.** The response body should look like this:


```
{
    "id": "1e508682-bde5-4390-8328-da591b4f571c",
    "name": "LabDataIngestionFWDataSource",
    "rules": [
        "Lab Data Ingestion - Create Device",
        "Lab Data Ingestion - Create Cable Fiber Connection"
    ],
    "ingestionConfig": {
        "ingestionURL": "kafka0:9092?topics=data-ingestion-lab",
        "groupId": "consumer_test_group-test2",
        "authenticationType": "Kafka",
        "contentType": "application/json",
        "concurrencyCount": 1
    },
    "retryConfig": {
        "ingestionURL": "localhost:9092?topics=retry-data-ingestion-
   lab",
        "groupId": "retry_test_group-test2",
        "authenticationType": "Kafka",
        "contentType": "application/json",
        "timeoutSec": 5,
        "retryCount": 3,
        "concurrencyCount": 1
    },
```

```
    "perspectives": [
        2
    ],
    "operations": [
        "createDevice",
        "createConnection"
    ],
    "ignoreCompatibilities": true,
    "ignoreRelationshipFailure": false,
    "createActivityCallout": "com.blueplanet.inventory.ingestionFramework.callouts.CreateActivityCallout.createActivity",
    "endpointDetails": [
        {
            "url": "https://bpi.lab:443/blueplanet-inventory-rest-showcase/ingestion/api/v1/LabDataIngestionFWDataSource/createDevice",
            "version": "v1",
            "schemaURL": "https://bpi.lab:443/blueplanet-inventory-rest-showcase/ingestion/api/v1/LabDataIngestionFWDataSource/createDevice/schema"
        },
        {
            "url": "https://bpi.lab:443/blueplanet-inventory-rest-showcase/ingestion/api/v1/LabDataIngestionFWDataSource/createConnection",
            "version": "v1",
            "schemaURL": "https://bpi.lab:443/blueplanet-inventory-rest-showcase/ingestion/api/v1/LabDataIngestionFWDataSource/createConnection/schema"
        }
    ],
    "active": true
}
```

In the response body, you can see that the id value is assigned to the data source. The response also returned information about REST endpoints. In the property **endpointDetails**, for each operation, you can see that the endpoint for sending event messages is configured as well as the endpoint to which we can send the JSON schema which will be used to validate events sent through that endpoint.

7. Send **Get all data sources** request from the collection. You should be able to find your data source configuration in the returned list.
8. Open the **Get data source** request from the collection and examine the URL of the request. Hover with your mouse over the Postman variables to see the values which will be used. Send the request to fetch the configuration of the data source you created in the previous steps.



Task 2: Adding JSON Validation Schema

In this task, you will define how the event message should look and what restrictions should be imposed on the event message parameters. For each operation configured with the data source, you will use a different event message structure. First, you will define the structure for applying the device configuration.

1. Inspect the event message which will be used to create the device:

```
{
    "version": 1,
    "header": {
        "envelopeId": "e2e9c706-3da6-4633-a9c7-063da62633d9",
        "timestamp": "2020-07-23T21:37:27.147Z"
    },
    "event": {
        "_type": "com.bp.inv.ResourceCreated",
        "resource": {
            "_type": "com.bp.inv.Resource",
            "label": "Ingestion Device 2",
            "resourceTypeId": "com.bp.inv.metamodel.Device",
            "properties": {
                "version":"1",
                "serialNumber": "TEST2",
                "modelNumber": "ASR9001",
                "roles": [ "Ethernet" ],
                "description": "Device description",
                "atType": "ASR 9001",
                "locationRefs": [ "255911256145162381" ],
                "orderRef":["255765155816568224"]
            }
        }
    }
}
```

2. Create the **deviceJSONSchema.json** file in the **C:\Users\student\Workspaces\Lab6** folder and open it with VSC. This file will be used as a temporary file to modify and edit the schema before copying it to the Postman request.
3. Copy the following JSON to the **deviceJSONSchema.json** file:

```
{
    "$schema": "http://json-schema.org/draft-07/schema#",
```

```
    "type": "object",
    "required": ["version", "header", "event"],
    "properties": { }
}
```

There are a few things that are set by this part of the schema:

a. **$schema:** Defines the schema standard which is used to describe attributes of the schema.
b. **type:** Defines the data type. At this top level, it defines that the event message must be a JSON object.
c. **required:** defines that properties **version**, **header**, and **event** must be present in the event message.
d. **properties:** more detailed restrictions and information about **version**, **header**, and **event** properties can be provided here.

4. Copy the following part of the schema to the **properties** attribute from the previous step to provide a specification for **version**, **header**, and **event** properties:

```
"version": {"type": "integer"},
"header": {
    "type": "object",
    "required": ["envelopeId", "timestamp"],
    "properties": {
        "envelopeId": {"type": "string"},
        "timestamp": {"type": "string"}
    }
},
"event": {
    "type": "object",
    "required": ["_type", "resource"],
    "properties": { }
}
```

The property **version** should be an integer. The property **header** is a JSON object which contains the properties **envelopeId** and **timestamp.** The property **event** is a JSON object which contains the properties **_type** and **resource**.

5. Copy the following part of the schema to the **properties.event.properties** (**properties** part of the **event** property schema definition):

```
"_type": {"enum": ["com.bp.inv.ResourceCreated",
   "com.bp.inv.ResourceUpdated", "com.bp.inv.ResourceDeleted"]},
"resource": {
    "type": "object",
    "required": ["_type", "label", "resourceTypeId", "properties"],
```

```
    "properties": { }
}
```

This part describes the **_type** and **resource** properties of the **event**:

a. **_type:** Value can be one of the values listed in the enumeration: "com.bp.inv.ResourceCreated", "com.bp.inv.ResourceUpdated" or "com.bp.inv.ResourceDeleted".

b. **resource:** An object which holds information about the device which needs to be created.

6. Inside the properties part of the resource, paste the following part of the schema:

```
{
    "_type": {"enum": ["com.bp.inv.Resource"]},
    "label": {"type": "string"},
    "resourceTypeId": {"enum": ["com.bp.inv.metamodel.Device"]},
    "properties": {
        "type": "object",
        "required": ["roles","atType","locationRefs","orderRef"],
        "properties": { }
    }
}
```

This part of the schema describes the properties of the resource:

a. **_type:** Can only be “com.bp.inv.Resource”.

b. **label:** This value will be used as the device name so it needs to be a string data type.

c. **resourceTypeId:** Since you will be configuring the device with this type of message, this can only take the value “com.bp.inv.metamodel.Device”.

d. **properties:** A JSON object which will contain values for attributes that will be stored on the Device node or as versioned properties. The information about the order and location references will be passed in here as well. The **createActivityCallout** method tries to find order references inside the properties of the resource.

7. Inside the **properties** part of the **properties** defined in the previous step, paste the following part of the JSON schema:

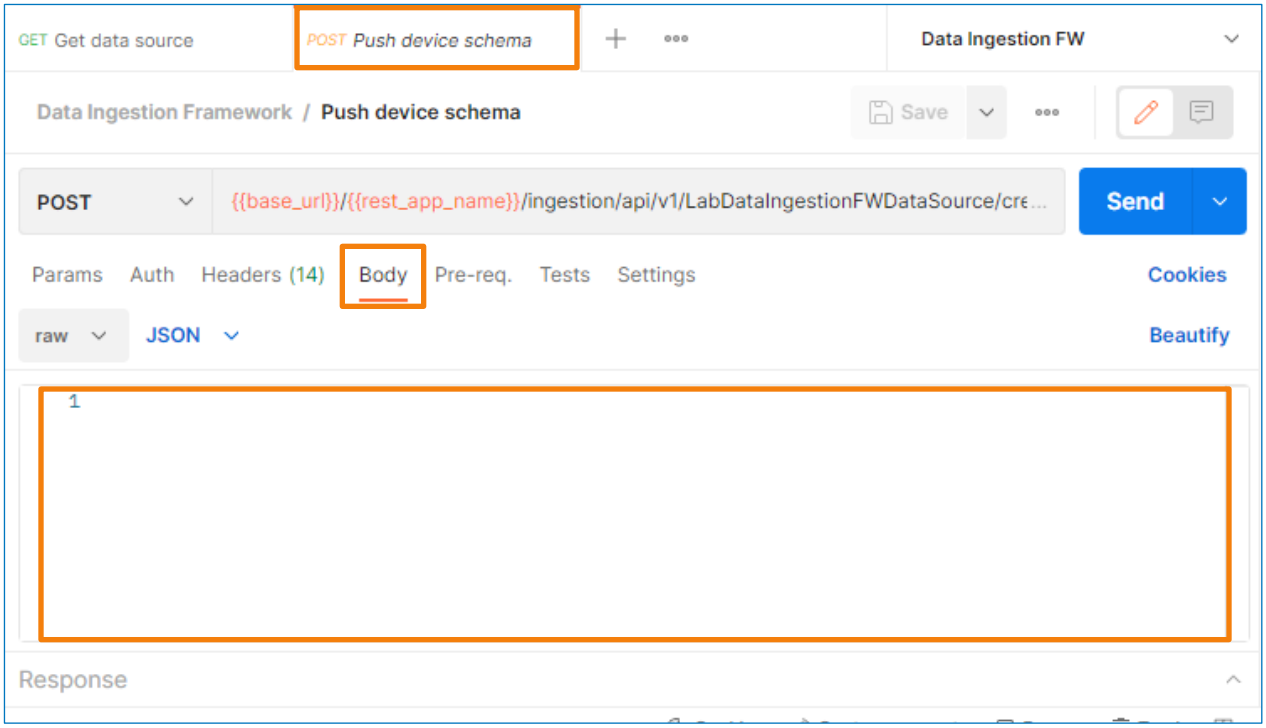
```
"ingestion_version": {"type": "string"},
"serialNumber": {"type": "string"},
"modelNumber": {"type": "string"},
"roles": {
    "type": "array",
    "items":{
        "type": "string",
        "enum": ["Ethernet", "Optical", "OTN", "MPLS", "gNodeB",
   "eNodeB", "Agregate", "Edge Router", "Access", "Compute"]
```

```
    }
},
"description": {"type": "string"},
"atType": {"type": "string"},
"locationRefs": {
    "type": "array",
    "items":{"type": "string"}
},
"orderRef": {
    "type": "array",
    "items":{"type": "string"}
}
```

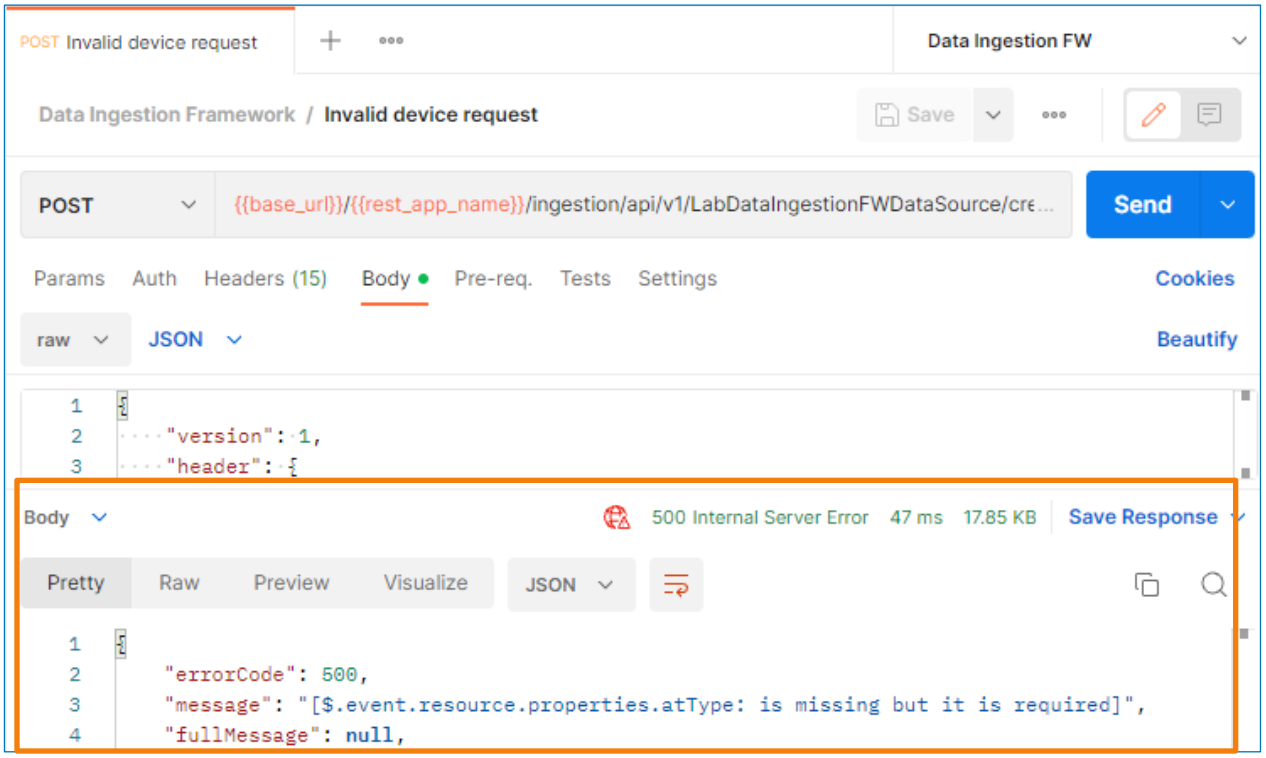
This part of the schema defines the properties of the **properties** JSON object:

a. **ingestion_version:** Not mandatory. This can be used to track with which version of the rule is the device deployed.
b. **serialNumber:** Non-mandatory versioned attribute.
c. **modelNumber:** Non-mandatory node property.
d. **roles:** List of device roles. Values in the list can only be one of the values specified in the enum part of the **roles** schema definition.
e. **description:** String description for a device.
f. **atType:** The name of the archetype for the device.
g. **locationRefs:** List of location references. Location reference is a location drnId passed as a string in the list. The device can be related to multiple locations.
h. **orderRef:** List of order references. The **createActivityCallout** method expects the property **orderRef** to be present and to be of type array.

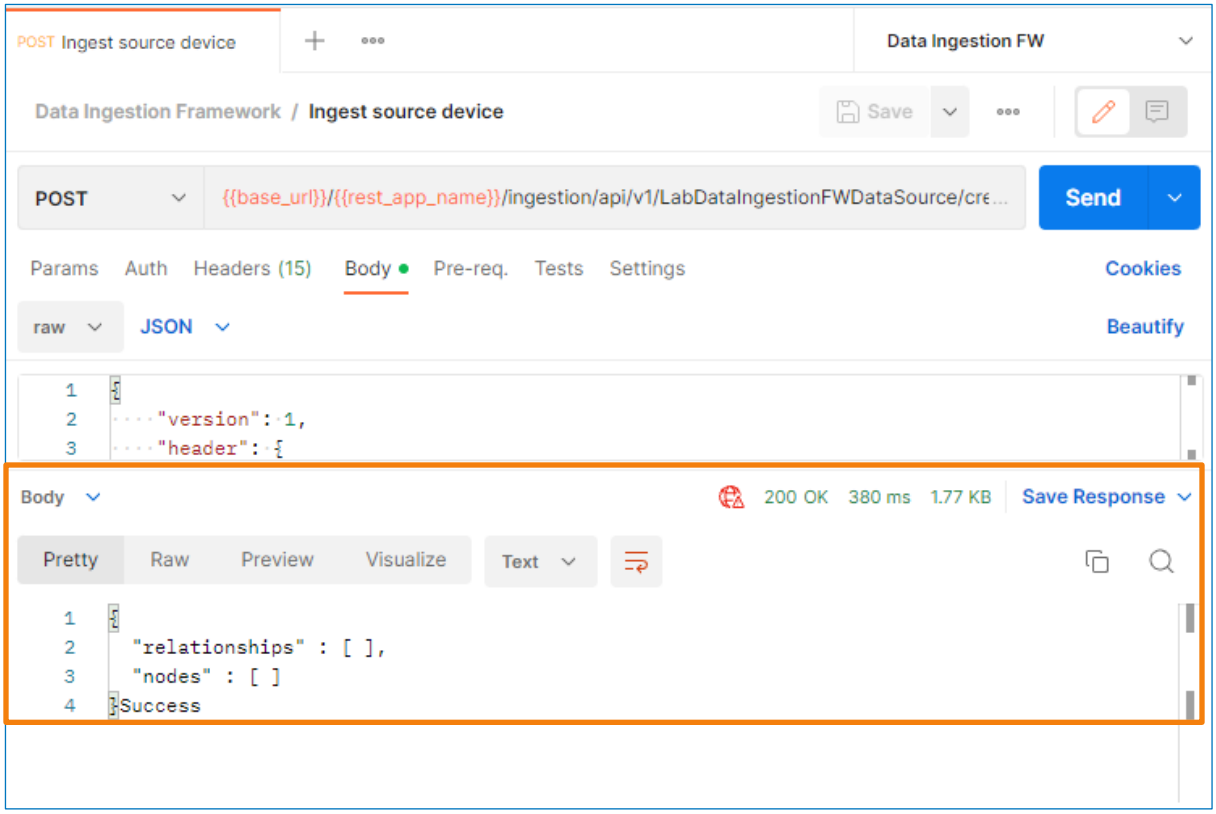
8. Now that you have a full schema for the event message for creating the device, go to Postman and open the **Push device schema** request from the collection. Copy the schema which you created from **deviceJSONSchema.json** to the request body. Send the request.



9. Execute the request **Invalid device request** from the collection to verify that validation is triggered properly. The request is missing the **atType** property in **event.resource.properties.**



10. Execute the request **Ingest source device** from the collection to verify that the event message passed the JSON schema validation.



11. Execute the request **Push connection schema** to apply the schema for the event message which is used to create a connection between devices. Examine the schema from the request body.

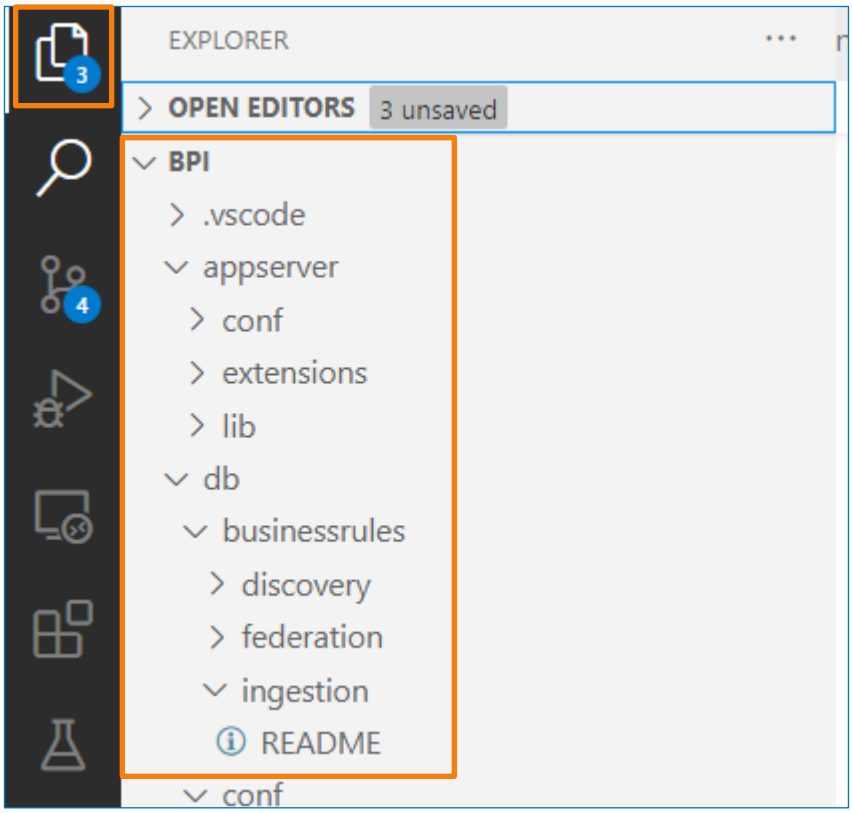
Task 3: Environment Setup

In this task, you write a Drools rule that creates a Device node in the database, and that generates a relationship to specified location node(s). You write drools rules in the Visual Studio Code(VSC) editor and you use VSC Blueplanet extensions to deploy Drools rules to the server.

1. From your student environment, open **VSC**.

2. Execute the action **Clone area**. In the dialogs that appear, select or fill in these values:
   a. BP System Area: **bpi**
   b. New Branch Name: **dataingestionfw**

   When prompted to select a folder, select the **C:\Users\student\Workspaces\Lab6** folder.
3. Once the repository is cloned, open the folder **C:\Users\student\Workspaces\Lab6\bpi** in VSC.
4. Inspect the folder structure. Your will place your custom Drools rules in the **C:\Users\student\Workspaces\Lab6\bpi\db\businessrules\ingestion** folder.



Task 4: Create a Drools Rule for Adding Devices to the Location

Now that you have the project set up you can start writing custom Drools rules for adding Devices to the inventory.

1. Create a file **LabDataIngestion_CreateDevice.drl** in the folder **C:\Users\student\Workspaces\Lab6\bpi\db\businessrules\ingestion**.

NOTE: Further instructions will assume that you have the folder **C:\Users\student\Workspaces\Lab6\bpi** opened in VSC and that the modifications to Drools rules will be done through VSC which will later be used to deploy the rules as well. In VSC, you should be able to expand the folder tree structure **db > businessrules > ingestion**. In this folder, you should be able to see the **LabDataIngestion_CreateDevice.drl** file. All the modifications to Drools rules in this task should be done in that file.

2. Copy the following code to the top of the file to define the packages and imports:

```
package org.bpi.demo;

/*

Data Ingestion Framework Lab

This is Ingestion Rule example for creating Device Node.

*/

import org.json.JSONObject;

import org.json.JSONArray;

import java.lang.String;

import java.util.Collections;

import java.util.Map;

import java.util.HashMap;

import java.util.List;

import java.util.ArrayList;

import
   com.blueplanet.inventory .ingestionFramework.decisionengine.vo.Inges
   tionDecisionResourceVO;

import
   com.blueplanet.inventory.ingestionFramework.decisionengine.vo.Ingest
   ionDecisionRelVO;

import
   com.blueplanet.inventory.ingestionFramework.decisionengine.vo.Ingest
   ionDecisionVO;
```

3. Declare the global variable **ingestionOutput** which will be used to pass the generated decision objects to Ingestion Data Framework for processing:

```
global
List<com.blueplanet.inventory.ingestionFramework.decisionengine.vo.Inges
tionDecisionVO> ingestionOutput;
```

Data Ingestion Framework will use the generated decision objects to modify the nodes and relationships in the inventory.

4. Drools rule definition is marked with the **rule/when/then/end** keywords. Create one Drools rule with the following parameters:
    a. **name:** Lab Data Ingestion - Create Device
    b. **dialect:** java
    c. **salience:** -1
    d. **activation-group:** Lab_Data_Ingestion

```
rule "Lab Data Ingestion - Create Device"
    dialect "java"
    salience -1
    activation-group "Lab_Data_Ingestion"

    when
        //condition is added here
    then
       //consequence is added here
end
```

5. Create the condition part of the rule. The event should be processed by this rule if it is of the type "com.bp.inv.ResourceCreated" and if the resource type is Device. Paste the following code to the condition part(“when” part) of the rule:

```
$event: JSONObject(
    has("event") &&
    getJSONObject("event").getString("_type") contains "ResourceCreated"
   &&
    getJSONObject("event").has("resource") &&
    getJSONObject("event").getJSONObject("resource").getString("resource
   TypeId") contains "Device"
)
```

When you are creating a rule condition, you should be aware that the event can be passed through the Kafka topic so it is possible that the event does not get validated by JSON schema.

When writing the consequence part of the rule you can either put the entire code in the “then” part or separate the code into smaller functions which will be called from other functions or from the “then” part of the rule. For easier separation of code in tasks and better readability, we will separate the code into smaller functions. Functions pre-defined outside of the rule definition and in your code, you should put them under the rule(below the end part)

6. Add the following code to the “then” part of the rule:

```
JSONObject deviceData =
   $event.getJSONObject("event").getJSONObject("resource");
String deviceName = deviceData.getString("label");
```

```
System.out.println("Creating device with name: " + deviceName);


        //create decision object for a device
IngestionDecisionVO createDeviceDecision = createDevice(deviceData);
ingestionOutput.add(createDeviceDecision);
```

This part of the code will extract information about the resource that you are trying to modify to the variable **deviceData.** This variable will be passed to the **createDevice** function which will generate the necessary decision object. Generated decision object will be appended to the global variable **ingestionOutput**

7. Below the **end** keyword add the **createDevice** function:

```
function IngestionDecisionVO createDevice(JSONObject deviceData) {
    IngestionDecisionResourceVO createDeviceDecision = new
    IngestionDecisionResourceVO();
    createDeviceDecision.setEventType("com.bp.inv.ResourceCreated");


    String deviceName = deviceData.getString("label");
    setDeviceQuery(createDeviceDecision, deviceName);


    String archetypeName =
    deviceData.getJSONObject("properties").getString("atType");
    setArchetypeInstanceQuery(createDeviceDecision, archetypeName);


    //set properties
    setNodeProperties(createDeviceDecision, deviceData);
    setVersionedProperties(createDeviceDecision,
    deviceData.getJSONObject("properties"), new
    String[]{"serialNumber"});


    //set callout
    createDeviceDecision.setCustomCallout("com.blueplanet.inventory.inge
    stionFramework.callouts.PostIngestionCallout.createOrModifyCallout")
    ;


    return createDeviceDecision;
}
```

This function does a couple of things:

a. Instantiate decision object of type **IngestionDecisionResourceVO**.
b. Set event type for the decision object.

c. Set instructions on the decision object on how to find the device in the database (if that device node exists). This is done with the **setDeviceQuery** function.
d. Set instructions on the decision object on how to find an archetype instance for the device. This is done with the **setArchetypeInstanceQuery** function.
e. Configure properties that will be stored on the node(**setNodeProperties**).
f. Configure properties that will be stored on the self-relationships on the node as versioned attributes(**setVersionedProperties**).
g. Configure the callout which will be used to process the resource after the node and properties have been created.
h. Return the decision object with set instructions to the caller.

8. Add the **setDeviceQuery** function:

```
function void setDeviceQuery(IngestionDecisionResourceVO
   createDeviceDecision, String deviceName) {

    //set query parameters

    HashMap matchingQueryListParams = new HashMap<String, Object>();

    matchingQueryListParams.put("ingestion_uuid", deviceName);

    matchingQueryListParams.put("deviceUuidPropName", "name");

    createDeviceDecision.setMatchingQueryListParams(Collections.singleto
   nList(matchingQueryListParams));


    // set device query

    createDeviceDecision.setMatchingQueryList(Collections.singletonList(
   "MATCH(d:Device) WHERE d[$deviceUuidPropName]=$ingestion_uuid RETURN
   d AS output limit 1"));


    //set retry message

    createDeviceDecision.setRetryMessage("Retry Exception : Unable to
   find the resource node with given parameters in DB");

}
```

The decision object is passed as an argument to the function. Function needs to configure the query which will be used to find the device in the database. The **IngestionDecisionResourceVO** class exposes the method **setMatchingQueryList** to provide a list of queries that can be used to find the resource node in the database. Depending on the inventory state and the queries you provide, this could match multiple nodes. If multiple nodes are matched, then decision object instructions will affect only the first matched node. For each query you provide, the Ingestion Framework processor expects a map of query parameters which can be provided with the **setMatchingQueryListParams** method. Query parameters are marked with a dollar sign in the query string. If the number of provided maps does not match the number of provided queries, then the processor will throw an error. If the map of parameters is not needed for the query, then an empty map should be passed. Query string and query map are matched according to their position in their respective lists. For example, the query at position 0 in the query string list will use the query parameters map at position 0 of the query parameters list to populate parameter values.

9. Add the **setArchetypeInstanceQuery** function:

```
function void setArchetypeInstanceQuery(IngestionDecisionResourceVO
   createDeviceDecision, String archetypeName) {

    // configure query parameters

    HashMap<String, Object> archetypeQueryParams = new HashMap<String,
   Object>();

    archetypeQueryParams.put("archetypeNameForCreation", archetypeName);

    createDeviceDecision.setArchetypeInstanceQueryParams(archetypeQueryP
   arams);


    // set archetype instance query

    createDeviceDecision.setArchetypeInstanceQuery("MATCH
   (Archetype{name:$archetypeNameForCreation})<-[:HAS_ARCHETYPE]-
   (archetypeInstance:ArchetypeInstance) return
   archetypeInstance.drniId as output limit 1");

}
```

There are three steps that the Data Ingestion Framework processor does to find out archetype instance id:

a. Get archetype instance ID directly from the decision object if it is configured.
b. Get archetype instance ID from the working memory from the cache.
c. Execute query to figure out archetype instance ID.

If the archetype instance ID is not configured directly on the decision object using the method **IngestionDecisionResourceVO.setArchetypeInstanceIdForCreation**, then the processor will try to find it in the cache. Archetype instance ID can be stored in the cache if the query for finding archetype instance ID was already executed beforehand. If the archetype instance ID was not found in the cache then the query which is set on the decision object through the method **setArchetypeInstanceQuery** will be executed. If the archetype instance ID is found, it will be stored in the cache using a key value that consists of a date when the query is executed and the archetype instance query string.

If the parameter map is not provided through the method **setArchetypeInstanceQueryParams**, Ingestion Framework Processor will generate an empty map and will not throw an error.

10. Add the **setNodeProperties** and **setVersionedProperties** functions:

```
function void setNodeProperties(IngestionDecisionResourceVO
   createDeviceDecision, JSONObject deviceData){

    HashMap<String, Object> deviceProperties = new HashMap<String,
   Object>(); //create object to store node properties


    JSONObject resourceProperties =
   deviceData.getJSONObject("properties");


    String deviceName = deviceData.getString("label");

    deviceProperties.put("name", deviceName);
```

```
    deviceProperties.put("lab_ingestion", true);
    for(String property: new String[]{"ingestion_version",
   "description", "modelNumber"}){
        if (resourceProperties.has(property)){
            deviceProperties.put(property,
   resourceProperties.getString(property));
        }
    }
    deviceProperties.put("role",
   resourceProperties.getJSONArray("roles").toList());
    String[] typeParts =
   deviceData.getString("resourceTypeId").split("\\.");
    deviceProperties.put("type", typeParts[typeParts.length - 1]);


    createDeviceDecision.setNodeProperties(deviceProperties);
    createDeviceDecision.setUpdateProperties(true);
}


function void setVersionedProperties(IngestionDecisionResourceVO
   createDeviceDecision, JSONObject resourceProperties, String[]
   properties){
    HashMap<String, Object> versionedProperties = new HashMap<>();
    for(String property: properties){
        if(resourceProperties.has(property)){
            Object propertyValue = resourceProperties.get(property);
            if(propertyValue != null){
                versionedProperties.put(property,
   propertyValue.toString());
            }
        }
    }
    createDeviceDecision.setVersionedProperties(versionedProperties);
}
```

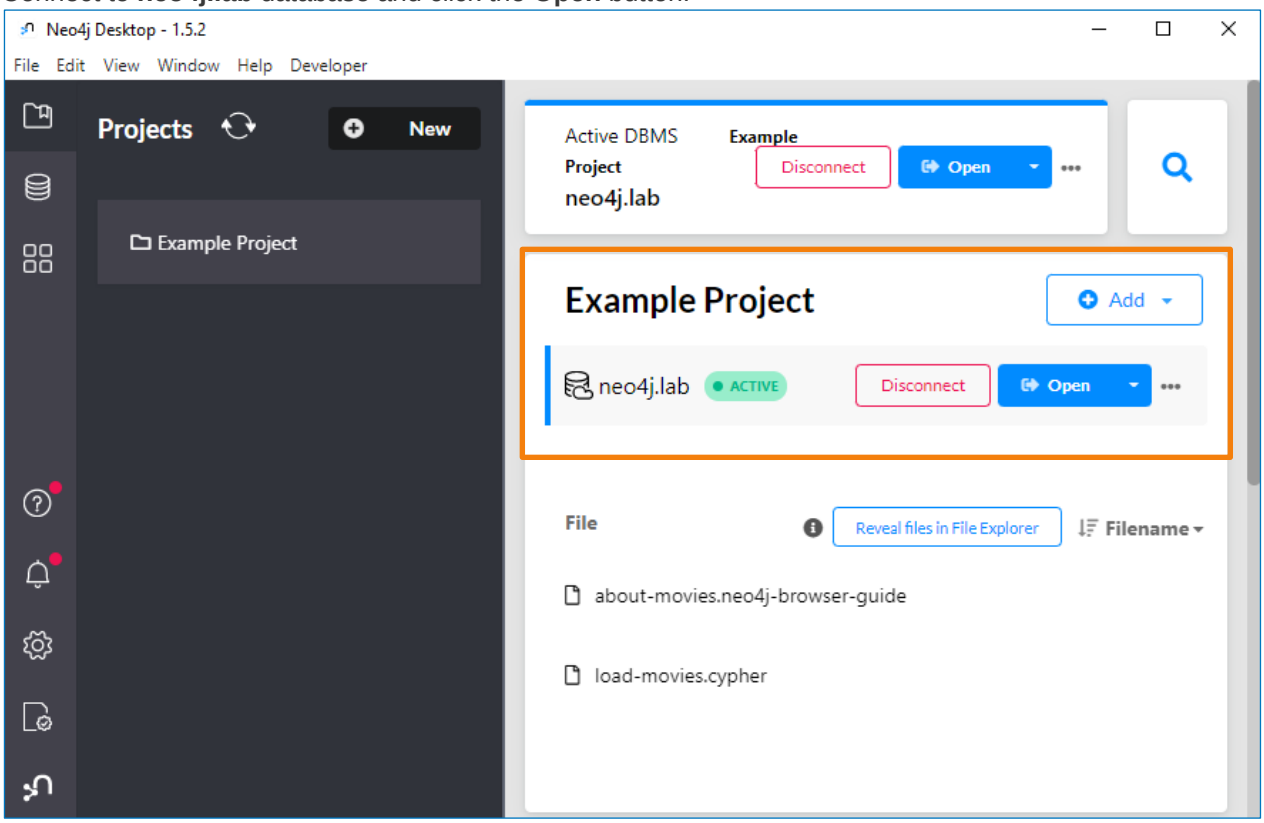
If you have done everything correctly, you should be able to deploy the Drools rule and create the device through the data source.

11. In VSC, open the **Blueplanet** extension and expand the actions for the **bpi.lab** system in the section **BLUE PLANET SYSTEMS**.
12. Execute the **Onboard** action.

13. In the dropdown menu, select the **bpi** option and when prompted for Pull Request Comment, press **Enter** to apply the default comment.

Even though you pushed the changes to the server you still cannot use the rule because the Drools session does not see the changes yet. You need to connect to the database to execute queries to refresh the database context and to refresh the Drools session.

On the desktop in your student environment, locate the **Neo4J Desktop** application and open it. Connect to **neo4j.lab** database and click the **Open** button.



14. In the new window that appears, log in with the following credentials:
    a. Username: **drni**
    b. Password: **neo4j**
15. To see the changes made to the Drools rule, you need to refresh the database context. Execute the following query:

```
CALL com.blueplanet.inventory.sdk.callouts.reloadDBContext();
```

16. To refresh Drools session, execute the following query:

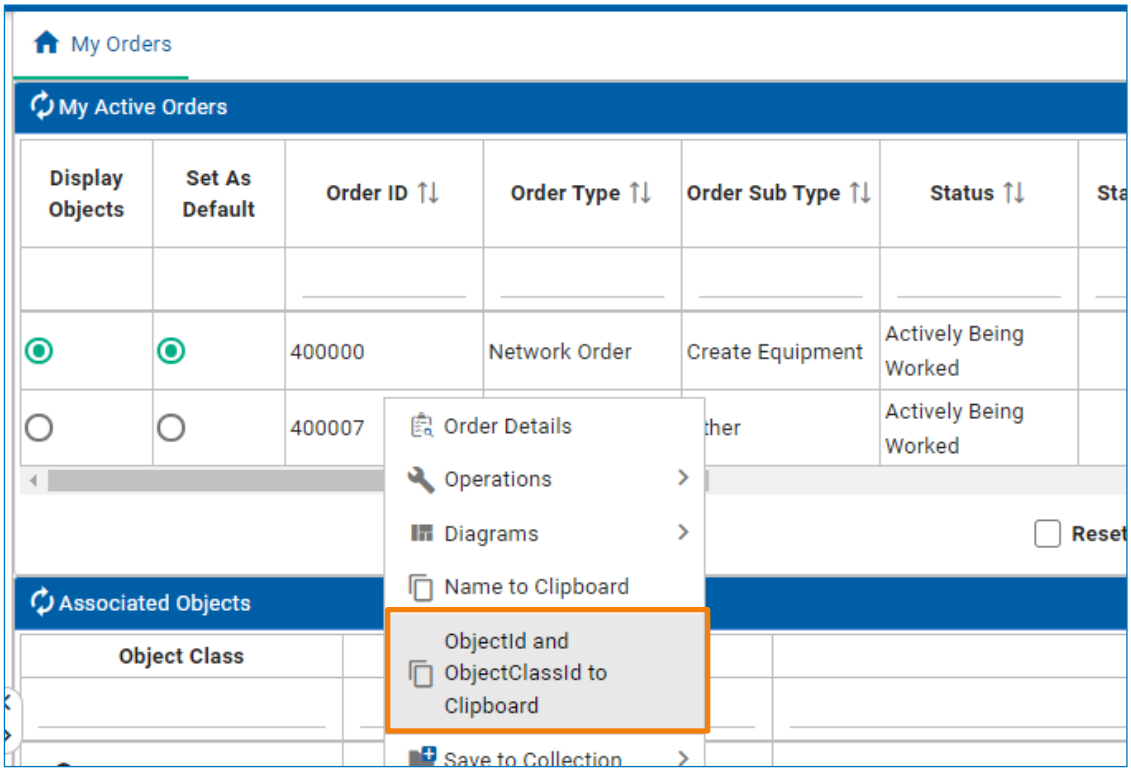
```
CALL com.blueplanet.inventory.ingestionFramework.refreshDroolsSession();
```

NOTE: Whenever you push changes to Drools rules to the server you need to execute both of those two steps.

It is time to create a device and see the changes created by the Drools rule. In order to do that you first need to create a new order and get the drnId of the order.

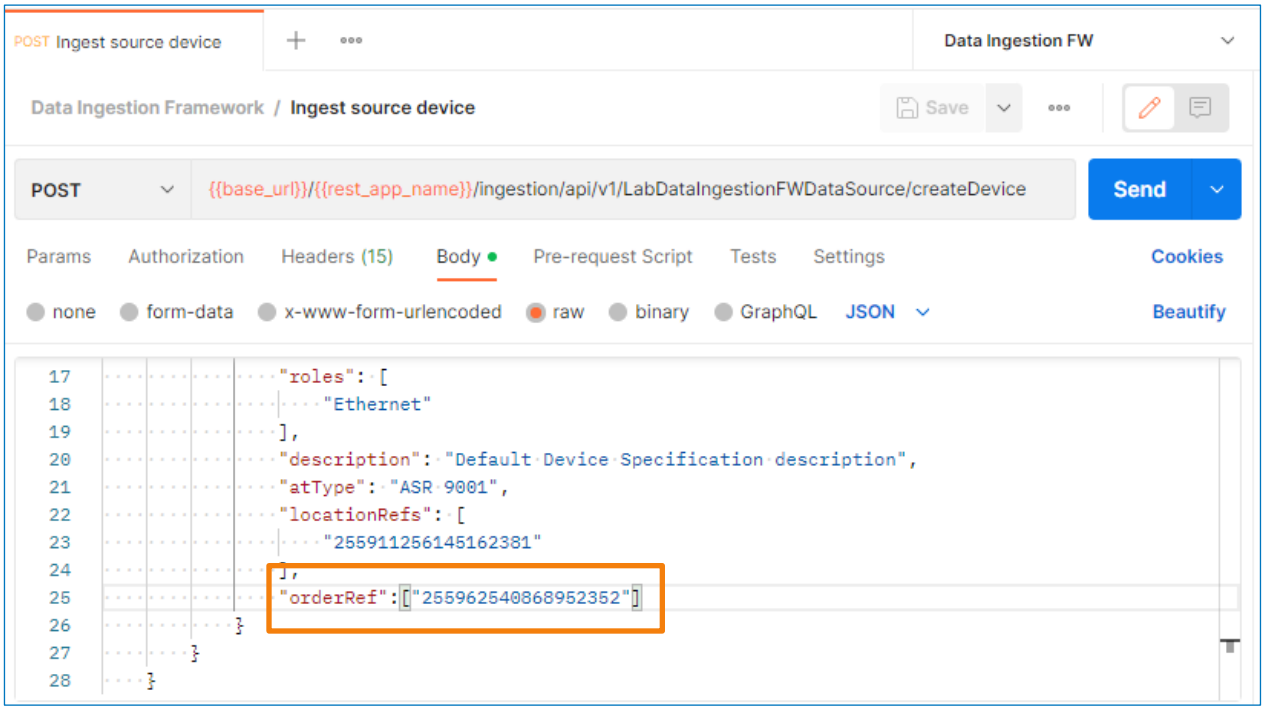
17. Open the Chrome browser in your student environment and go to *https://bpi.lab* and log in with credentials provided by the instructor.

18. Create new simple order.
19. Find your order in the **My Orders** tab and right-click on the order row. From the context menu select the option **ObjectId and ObjectClassId to Clipboard**.

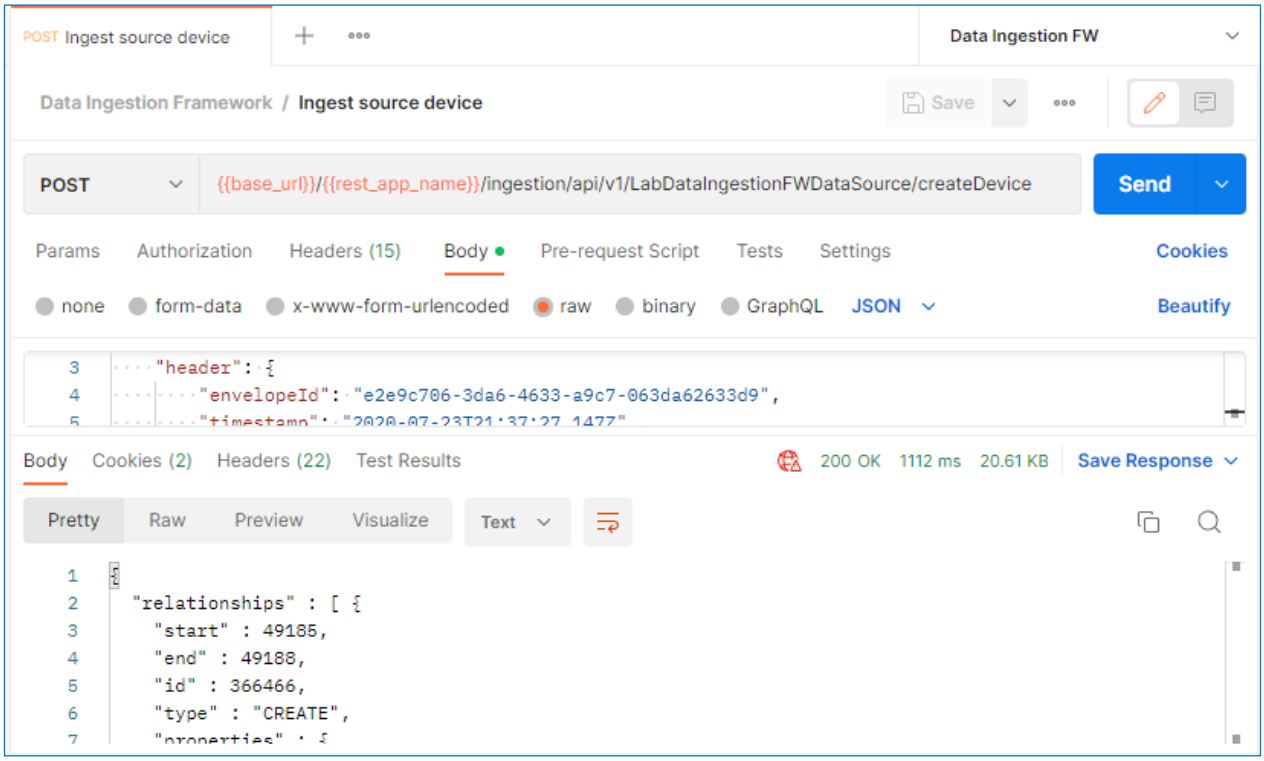


In the clipboard, you will now have a string that contains two parts separated by a colon. The first part will be the drnId of the order and the second part will be the drnId of the order Metamodel. Paste the string from the clipboard somewhere and copy only the first part (the part before the colon character).

20. In the Postman, open the request **Ingest source device** and go to the request body. In the request body, find the **orderRef** array and paste the order drnId as a string inside the array. Save the changes.

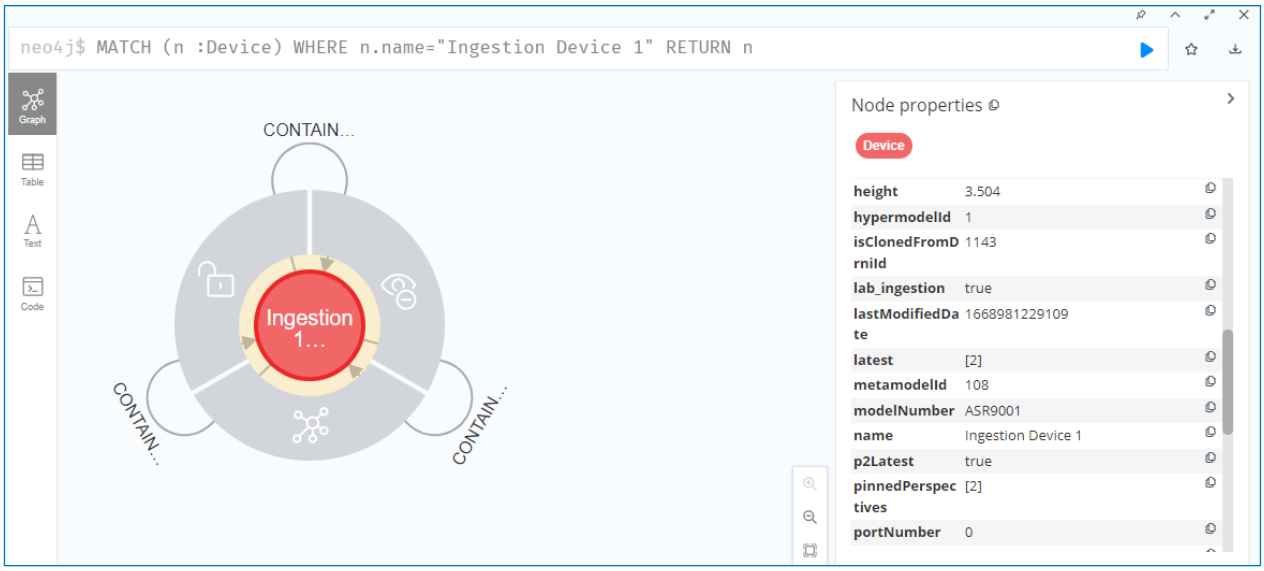


21. Execute the request **Ingest source device**.



22. If everything went ok you should be able to see the device in the database. In Neo4J, execute the following query to find the device:

```
MATCH (n :Device) WHERE n.name="Ingestion Device 1" RETURN n
```



Currently, the device can be ingested into the database but it is not assigned to any location. In the next few steps, you will add code to relate the device node to a specific location.

23. In the **LabDataIngestion_CreateDevice.drl** file, add the following code to the end of the consequence part (right above the **end** keyword):

```
IngestionDecisionVO putDeviceInLocationsDecision =
   putDeviceInLocations(deviceData);

ingestionOutput.add(putDeviceInLocationsDecision);
```

This calls the function which will generate the decision object with instructions to create a relationship between the device and specified locations.

24. Add the **putDeviceInLocations** function:

```
function IngestionDecisionVO putDeviceInLocations(JSONObject
   deviceData){

    IngestionDecisionRelVO putDeviceInLocationsDecision = new
   IngestionDecisionRelVO();

    putDeviceInLocationsDecision.setEventType("com.bp.inv.RelationshipCr
   eated");


    String deviceName = deviceData.getString("label");


    setLocationRelationshipSourceQuery(putDeviceInLocationsDecision,
   deviceData.getJSONObject("properties").getJSONArray("locationRefs"))
   ;
```

```
    setLocationRelationshipTargetQuery(putDeviceInLocationsDecision,
    deviceName);


    //set error and warning messages
    putDeviceInLocationsDecision.setRetryMessage("Unable to find the
    Location with given id(s) or Device with id " + deviceName + " in
    DB");
    putDeviceInLocationsDecision.setExceptionMessage("Exception occurred
    while trying to add Device with id " + deviceName + " in
    Locations");


    // set relationship type
    putDeviceInLocationsDecision.setRelationshipType("HAS");


    // set relationship properties
    Map<String, Object> relProps = new HashMap<String, Object>();
    putDeviceInLocationsDecision.setRelProperties(relProps);
    return putDeviceInLocationsDecision;
}
```

This function does the following:

a. Instantiate decision object of the type **IngestionDecisionRelVO**
b. Set event type for the decision object
c. Set instructions on the decision object on how to find locations for which references are provided (**setLocationRelationshipSourceQuery**). Locations are the starting points of the relationships which need to be created.
d. Set instructions on the decision object on how to find the device node in the database (**setLocationRelationshipTargetQuery**). The device is the end point of the relationships which need to be created.
e. Set warnings and error messages.
f. Set the relationship type.
g. Configure properties that will be stored on the created relationship.
h. Return the decision object with set instructions to the caller.

25. Add the **setLocationRelationshipSourceQuery** function:

```
function void setLocationRelationshipSourceQuery(IngestionDecisionRelVO
    relationshipDecision, JSONArray locationRefs){
    relationshipDecision.setSingleMatchSource(false);
    relationshipDecision.setMatchingQueryListForSource(Collections.singl
    etonList("Match(l:Location) where l.drniId in $locationIds return l
    as output"));


    List<Long> locationIds = new ArrayList<>();
```

```
    for (int k = 0; k < locationRefs.length(); k++) {
        locationIds.add(Long.valueOf(locationRefs.getString(k)));
    }
    Map<String, Object> matchingQueryListParamsSource = new
    HashMap<String, Object>();
    matchingQueryListParamsSource.put("locationIds", locationIds);
    relationshipDecision.setMatchingQueryListParamsForSource(Collections
    .singletonList(matchingQueryListParamsSource));
}
```

The device can be related to multiple locations. That is why you need to make sure that the query to fetch location nodes can return multiple nodes. This can be done by using the **setSingleMatchSource** method and sending the boolean value **false.**

26. Add the **setLocationRelationshipTargetQuery** function:

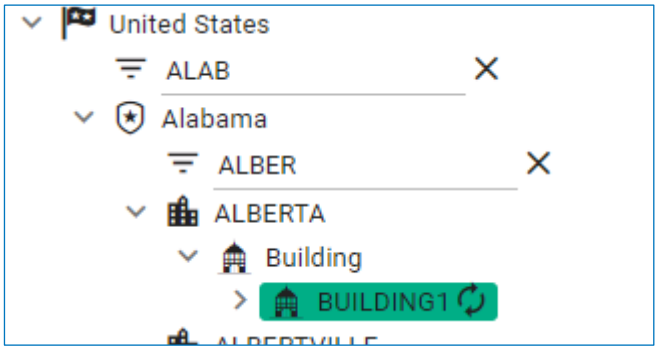
```
function void setLocationRelationshipTargetQuery(IngestionDecisionRelVO
    relationshipDecision, String deviceName){
    relationshipDecision.setSingleMatchTarget(true);
    relationshipDecision.setMatchingQueryListForTarget(Collections.singl
    etonList("Match(d:Device) where
    d[$deviceUuidPropName]=$ingestion_uuid return d as output"));

    Map<String, Object> matchingQueryListParamsTarget = new
    HashMap<String, Object>();
    matchingQueryListParamsTarget.put("ingestion_uuid", deviceName);
    matchingQueryListParamsTarget.put("deviceUuidPropName", "name");
    relationshipDecision.setMatchingQueryListParamsForTarget(Collections
    .singletonList(matchingQueryListParamsTarget));
}
```

27. Push the changes to the server and refresh the database context and Drools session.

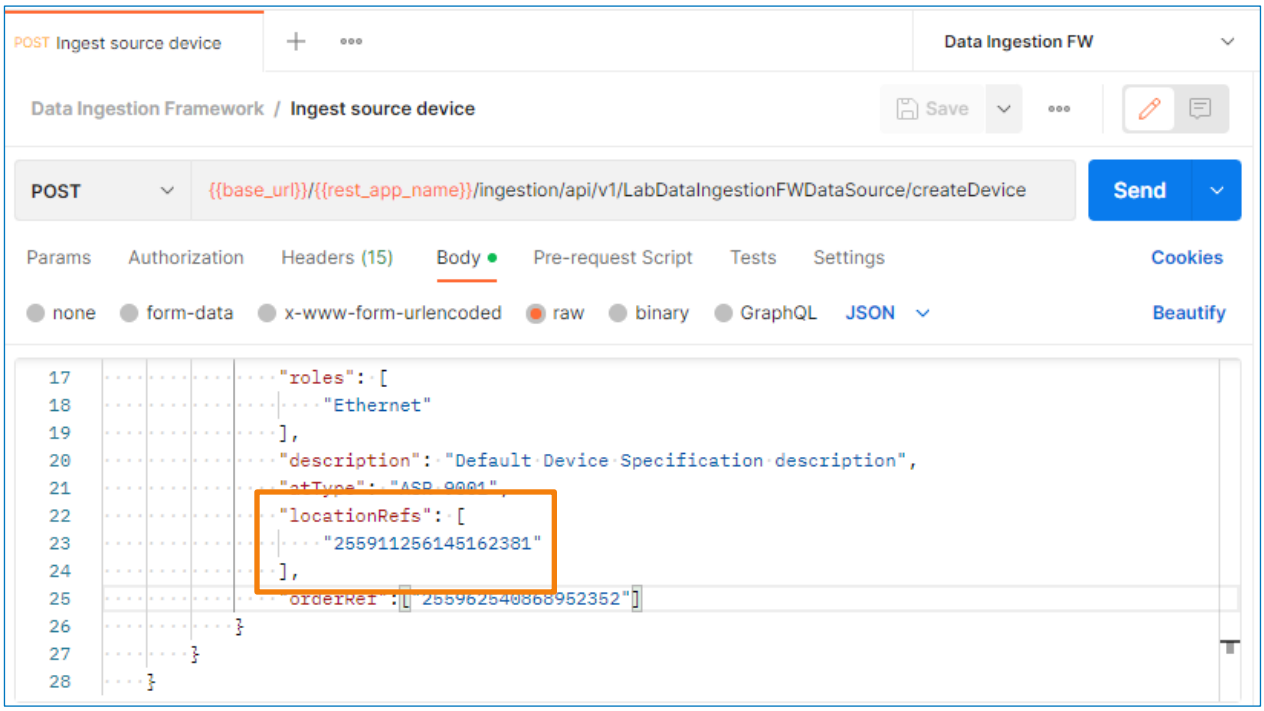
Now you can create or modify the device and relate it to a location.

28. From the BPI UI, expand the location tree view to see **United States > Alabama > Alberta**.

29. Add a new building to Alberta with the name **BUILDING1.** Refresh **Alberta** in the Locations tree view and expand **Alberta > Building > BUILDING1**. (Add BUILDING1 to Scratch Pad for quicker access in the future.)

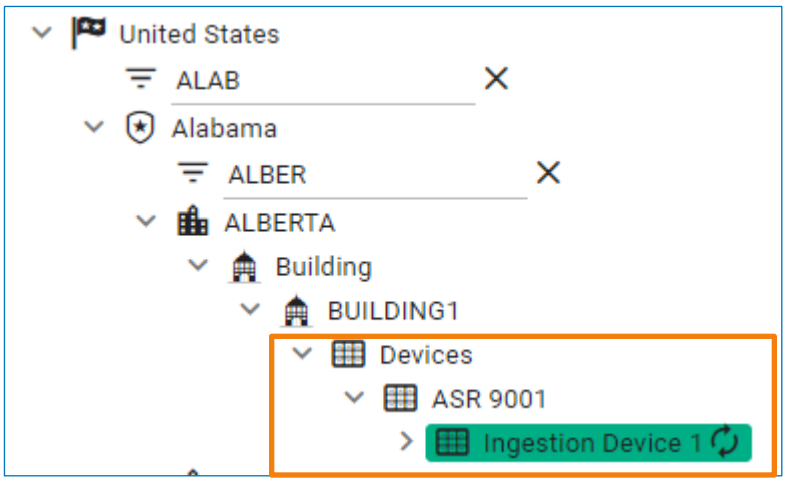


30. Right-click on **BUILDING1** and from the context menu, select the option **ObjectId and ObjectClassId to Clipboard.** In the clipboard, you will now have a string that contains two parts separated by the colon. The first part will be the drnId of the building and the second part will be the drnId of the building Metamodel. Paste the string from the clipboard somewhere and copy only the first part (the part before the colon character).
31. In the Postman, open the body of the request **Ingest source device**. In the request body, find the **locationRefs** array and paste the building drnId as a string inside the array. Save the changes.



32. Execute the **Ingest source device** request.

33. From the BPI UI, refresh **BUILDING1** in the location tree view. You should be able to see **Ingestion Device 1** listed under the devices in **BUILDING1**.



34. From the Postman, execute the request **Ingest target device** to create a second device which will be used for connection configuration.

Task 5: Create Drools rule for creating Cable Fiber connection between devices

In this task, you will create and deploy the rule which will allow you to ingest information about the cable fiber connection between two devices. The cable fiber connection is represented as the PhysicalConnection node in the database model. Therefore, the event message to create a cable fiber connection will be an event message to create the resource.

1. Create a file named **LabDataIngestion_CreateConnection.drl** in the folder **C:\Users\student\Workspaces\Lab6\bpi\db\businessrules\ingestion**.

NOTE: All the modifications to Drools rules in this task should be done in that file.

2. Copy the following code to the top of the file to define the package and imports:

```
package org.bpi.demo;

/*

Data Ingestion Framework Lab

This is Ingestion Rule example for creating Device Node.

*/

import org.json.JSONObject;

import org.json.JSONArray;

import java.lang.String;

import java.util.Collections;

import java.util.Map;

import java.util.HashMap;

import java.util.List;

import java.util.ArrayList;

import
   com.blueplanet.inventory .ingestionFramework.decisionengine.vo.Inges
   tionDecisionResourceVO;

import
   com.blueplanet.inventory.ingestionFramework.decisionengine.vo.Ingest
   ionDecisionRelVO;

import
   com.blueplanet.inventory.ingestionFramework.decisionengine.vo.Ingest
   ionDecisionVO;
```

3. Declare the global variable **ingestionOutput** which will be used to pass the generated decision objects to the Ingestion Data Framework for processing:

```
global
List<com.blueplanet.inventory.ingestionFramework.decisionengine.vo.Inges
tionDecisionVO> ingestionOutput;
```

4. Create one Drools rule with the following parameters:
    a. name: **Lab Data Ingestion - Create Cable Fiber Connection**
    b. dialect: **java**
    c. salience: **-1**
    d. activation-group: **Lab_Data_Ingestion**

```
rule "Lab Data Ingestion - Create Cable Fiber Connection"
    dialect "java"
    salience -1
    activation-group "Lab_Data_Ingestion"

    when
        //condition is added here
    then
       //consequence is added here
end
```

5. To the rule condition part, add the following condition:

```
$event: JSONObject(
    has("event") &&
    getJSONObject("event").getString("_type") contains "ResourceCreated"
   &&
    getJSONObject("event").has("resource") &&
    getJSONObject("event").getJSONObject("resource").getString("resource
   TypeId") contains "PhysicalConnection"
)
```

The cable fiber connection is the PhisycalConnection so with this condition all the messages that do not modify the PhysicalConnection will be filtered out.

6. Add the following code to the rule consequence part (between the **then** and the **end** keywords):

```
JSONObject connectionData =
   $event.getJSONObject("event").getJSONObject("resource");
String connectionName = connectionData.getString("label");
System.out.println("Creating connection with name: " + connectionName);


IngestionDecisionVO createConnectionDecision =
   createConnection(connectionData);
ingestionOutput.add(createConnectionDecision);
```

```
IngestionDecisionVO relateConnectionSourceDecision =
   relateConnectionToDevice(connectionData, true);

ingestionOutput.add(relateConnectionSourceDecision);


IngestionDecisionVO relateConnectionTargetDecision =
   relateConnectionToDevice(connectionData, false);

ingestionOutput.add(relateConnectionTargetDecision);
```

This part of the code will call functions to generate decision objects with instructions to create a connection node and instructions to create relationships between the connection node and the source and target devices. Because creating the decision object is very similar when it comes to creating the relationship between the connection node and source and target devices, we can use the same **relateConnectionToDevice** function to generate those two decision objects.

7. Add the **createConnection** function to the file:

```
function IngestionDecisionVO createConnection(JSONObject connectionData)
   {
    IngestionDecisionResourceVO createConnectionDecision = new
   IngestionDecisionResourceVO();
    createConnectionDecision.setEventType("com.bp.inv.ResourceCreated");


    String connectionName = connectionData.getString("label");
    setConnectionQuery(createConnectionDecision, connectionName);


    final Long CABLE_FIBER_ARCHETYPE_INSTANCE_ID = 5005L;
    createConnectionDecision.setArchetypeInstanceIdForCreation(CABLE_FIB
   ER_ARCHETYPE_INSTANCE_ID);


    setNodeProperties(createConnectionDecision, connectionData);
    createConnectionDecision.setCustomCallout("com.blueplanet.inventory.
   ingestionFramework.callouts.PostIngestionCallout.createOrModifyCallo
   ut");


    return createConnectionDecision;
}
```

This function is very similar to the function createDevice from the previous task. There are a couple of differences:

a. The function **createConnection** is not setting a query to fetch archetype instance ID from the database. The archetype instance id is directly configured on the decision object using the **setArchetypeInstanceIdForCreation** method

b. There is no need to configure any versioned properties for the connection node so skip this step in this method.

8. Add the function **setConnectionQuery**:

```
function void setConnectionQuery(IngestionDecisionResourceVO
   createConnectionDecision, String connectionName) {

    //set query parameters

    HashMap matchingQueryListParams = new HashMap<String, Object>();

    matchingQueryListParams.put("ingestion_uuid", connectionName);

    matchingQueryListParams.put("connectionUuidPropName", "name");

    createConnectionDecision.setMatchingQueryListParams(Collections.sing
   letonList(matchingQueryListParams));

   createConnectionDecision.setMatchingQueryList(Collections.singletonLi
   st("MATCH(c:PhysicalConnection) WHERE
   c[$connectionUuidPropName]=$ingestion_uuid RETURN c AS output limit
   1"));


    //set retry message

    createConnectionDecision.setRetryMessage("Retry Exception : Unable
   to find the resource node with given parameters in DB");

}
```

This function sets instructions on how to find this connection node in the database. The query which is used here is very similar to the query used to find the device node in the database.

9. Add the function **setNodeProperties**:

```
function void setNodeProperties(IngestionDecisionResourceVO
   createConnectionDecision, JSONObject connectionData){

    HashMap<String, Object> connectionProperties = new HashMap<String,
   Object>(); //create object to store node properties


    String connectionName = connectionData.getString("label");

    connectionProperties.put("name", connectionName);


    connectionProperties.put("precomputedtypename", "Cable Fiber");

    connectionProperties.put("userDefinedName", "Y");

    connectionProperties.put("connectionPageRoutingPanelView",
   "simple");

    connectionProperties.put("lab_ingestion", true);


    createConnectionDecision.setNodeProperties(connectionProperties);

        createConnectionDecision.setUpdateProperties(true);

}
```

All the node properties except the connection name are predetermined.

The code for creating the connection node is added. To complete the rule you need to add the code to create relationships between the connection node and the devices. Information about devices is stored resource properties in the event message. That is, the names of the devices which need to be related to the connection node are passed in the event message.

10. To make it easier to fetch device names, add the **getDeviceName** helper method to the file. This method should take as an argument resource JSONObject and a boolean flag if it needs to return the name of the source device. The function returns the name of either the source device or the name of the target device:

```
function String getDeviceName(JSONObject resourceProperties, boolean
   isSource){
    if(isSource){
        return resourceProperties.getString("source");
    }
    return resourceProperties.getString("target");
}
```

11. Add implementation for the **relateConnectionToDevice** function:

```
function IngestionDecisionVO relateConnectionToDevice(JSONObject
   connectionData, boolean isSource){
    IngestionDecisionRelVO relateConnectionToDeviceDecision = new
   IngestionDecisionRelVO();
    relateConnectionToDeviceDecision.setEventType("com.bp.inv.Relationsh
   ipCreated");

    String connectionName = connectionData.getString("label");
    setRelationshipSourceQuery(relateConnectionToDeviceDecision,
   connectionName);

    String deviceName =
   getDeviceName(connectionData.getJSONObject("properties"), isSource);
    setRelationshipTargetQuery(relateConnectionToDeviceDecision,
   deviceName);

    //set error and warning messages
    relateConnectionToDeviceDecision.setRetryMessage("Unable to find the
   Connection or Device node with given parameters");
    relateConnectionToDeviceDecision.setExceptionMessage("Exception
   occurred while trying to relate Device with id " + deviceName + " to
   PhysicalConnection with id " + connectionName);

    // set relationship type
```

```
    relateConnectionToDeviceDecision.setRelationshipType("HAS_CONNECTION
    _COMPONENT");


    // set relationship properties
    setRelationshipProperties(relateConnectionToDeviceDecision,
    connectionData, isSource);
    return relateConnectionToDeviceDecision;
}
```

This method is very similar to the **putDeviceInLocations** method from the previous task. There are a couple of differences:

a. For source query we are expecting only one node as the result output.
b. Relationship type is HAS_CONNECTION_COMPONENT.
c. Relationship properties are now provided because they will give information to the BPI on which device is the source device and which is the target device.

12. Add the functions **setRelationshipSourceQuery** and **setRelationshipTargetQuery**:

```
function void setRelationshipSourceQuery(IngestionDecisionRelVO
    relationshipDecision, String connectionName){
    relationshipDecision.setSingleMatchSource(true);
    relationshipDecision.setMatchingQueryListForSource(Collections.singl
    etonList("MATCH(c:PhysicalConnection) WHERE
    c[$connectionUuidPropName]=$ingestion_uuid RETURN c AS output"));


    Map<String, Object> matchingQueryListParamsSource = new
    HashMap<String, Object>();
    matchingQueryListParamsSource.put("ingestion_uuid", connectionName);
    matchingQueryListParamsSource.put("connectionUuidPropName", "name");
    relationshipDecision.setMatchingQueryListParamsForSource(Collections
    .singletonList(matchingQueryListParamsSource));
}


function void setRelationshipTargetQuery(IngestionDecisionRelVO
    relationshipDecision, String deviceName){
    relationshipDecision.setSingleMatchTarget(true);
    relationshipDecision.setMatchingQueryListForTarget(Collections.singl
    etonList("Match(d:Device) where
    d[$deviceUuidPropName]=$ingestion_uuid return d as output"));


    Map<String, Object> matchingQueryListParamsTarget = new
    HashMap<String, Object>();
    matchingQueryListParamsTarget.put("ingestion_uuid", deviceName);
```

```
    matchingQueryListParamsTarget.put("deviceUuidPropName", "name");
    relationshipDecision.setMatchingQueryListParamsForTarget(Collections
    .singletonList(matchingQueryListParamsTarget));
}
```

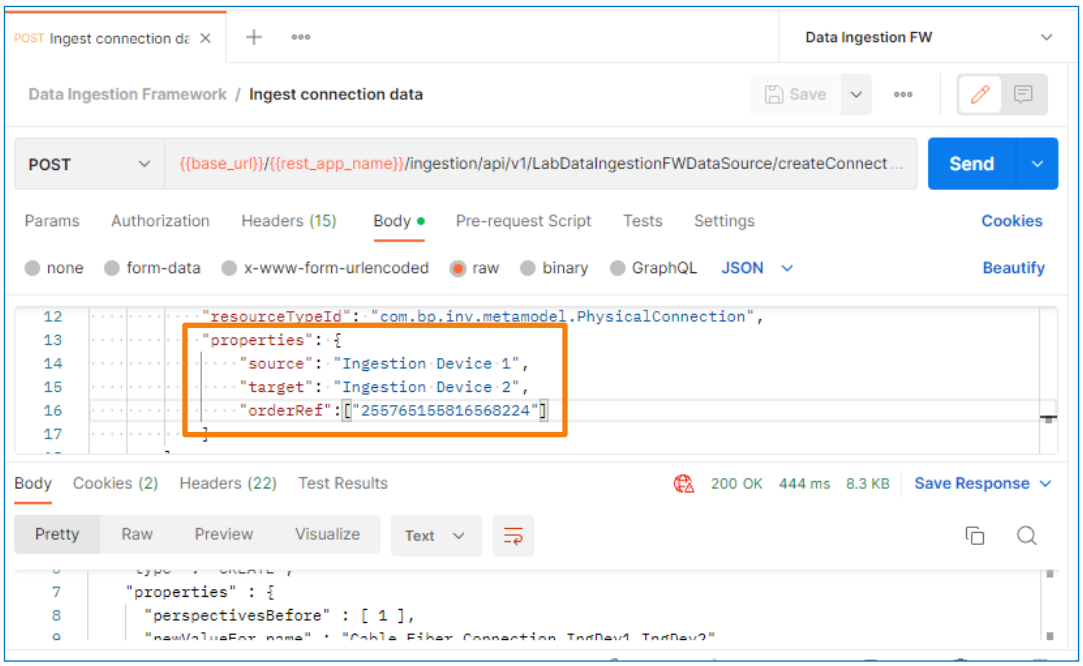
13. Add the function **setRelationshipProperties**:

```
function void setRelationshipProperties(IngestionDecisionRelVO
   relationshipDecision, JSONObject connectionData, boolean isSource){
    Map<String, Object> relProps = new HashMap<String, Object>();

    relProps.put("isEndOfConnection", "Y");
    relProps.put("isNotInRoute", "N");
    relProps.put("sectionNumber", 1);
    relProps.put("segmentNumber", 1);
    if (isSource){
        relProps.put("seq", 1);
    } else{
        relProps.put("seq", 2);
    }
    relationshipDecision.setRelProperties(relProps);
}
```

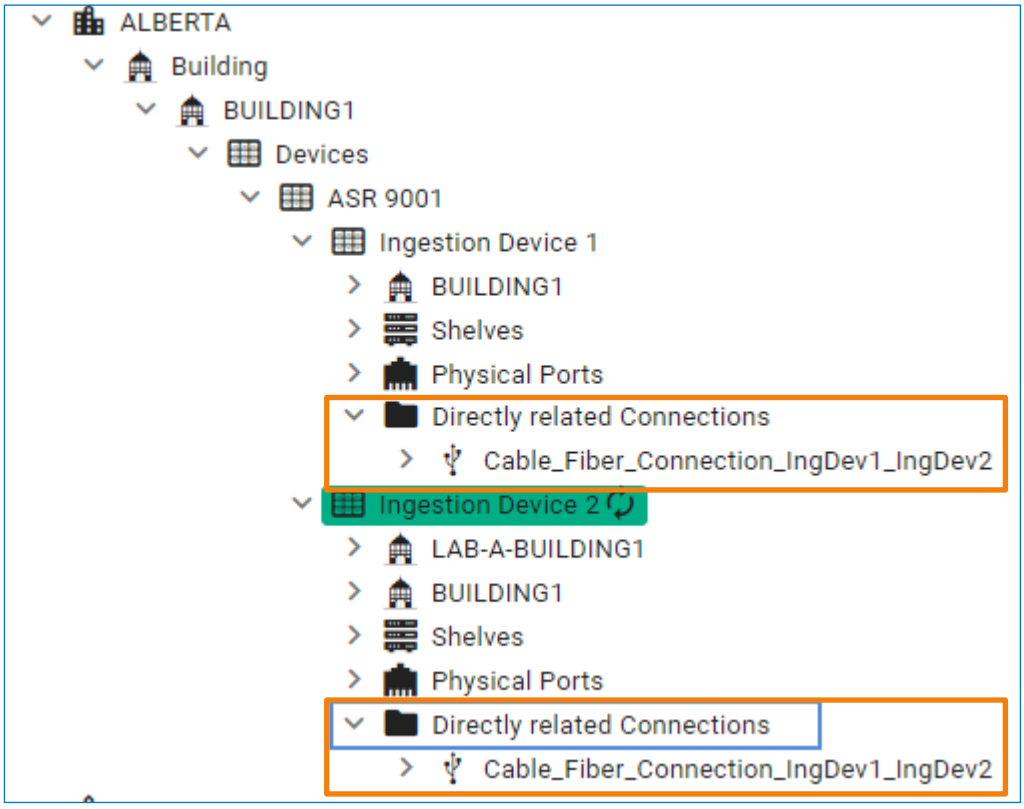
All relationship properties are predetermined. The value of property **seq** will be set to 1 if the relationship connects the source device node with the connection node or 2 otherwise.

14. If you have added all the code to the **LabDataIngestion_CreateConnection.drl** file, Onboard the rule to the **bpi.lab** server.
15. Refresh the database context and Drools session.

16. Open the **Ingest connection data** request in the Postman and set property orderRef in the request body. Execute the request.



17. If the request passed successfully with status 200, you should be able to see the connection listed under the **Directly related Connections** section for the device in the locations tree view.



End of Lab

Lab 8: Additional Extensions

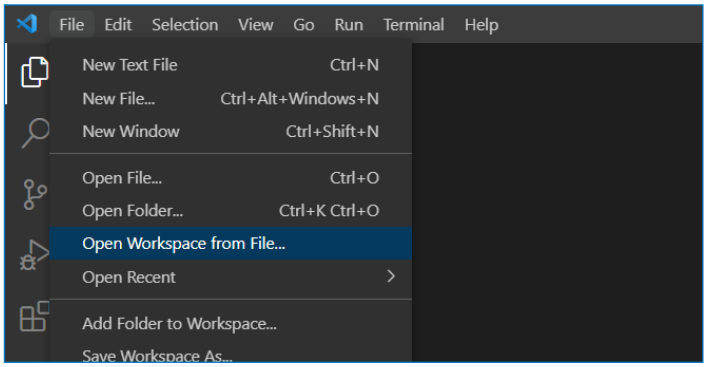
Objectives

- Implement extensions of the connection naming business logic by onboarding a Groovy script

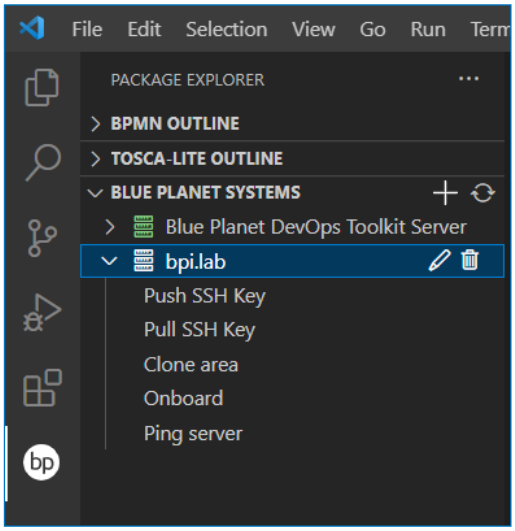
Task 1: Set Up the Environment

In one of the previous labs, you learned how to implement business logic for naming devices. In the following tasks, you will examine and update the code which implements more complex logic for the naming of connections. You will customize the naming logic by onboarding a new Groovy script through the Visual Studio Code Blue Planet Extension. First, you will use the VSC IDE to clone the bpi area from the Asset Manager and add it to your Workspace.

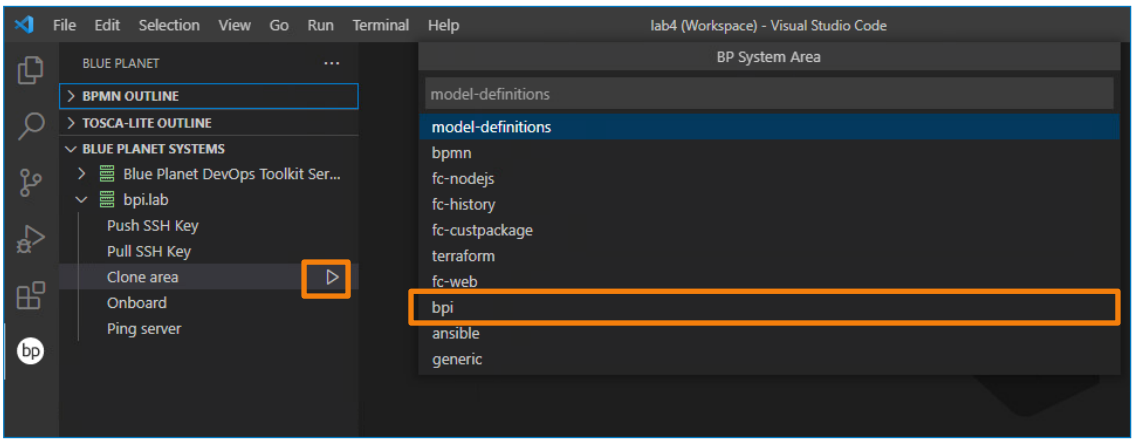
1. Open VSC from your desktop by clicking the **VSC** icon.
2. From the main menu, select **File > Open Workspace from File…** to open the workspace for this lab.



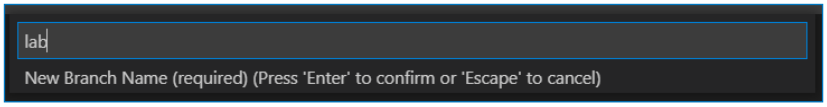
3. In the file browser, find the workspace file **C:\Users\student\Workspaces\Lab8\lab8.code-workspace** and click **Open**.
4. Click the **Blue Planet** menu option in the VSC left sidebar and expand the **bpi.lab** server you added in a previous lab.



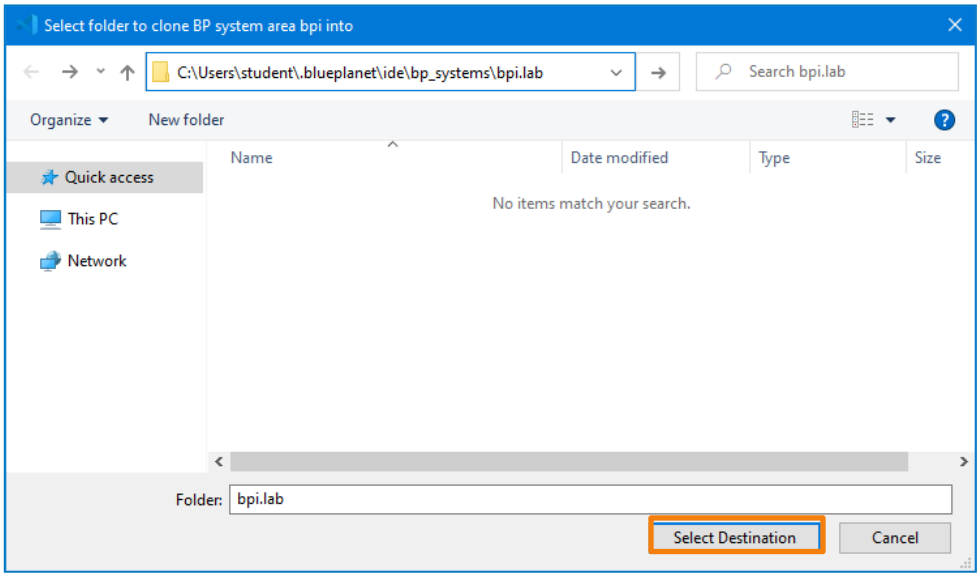
5. Next, clone the directory structure from the server to your VSC workspace. Click on the **play icon** next to **Clone area** and then select **bpi** when asked for BP System Area.



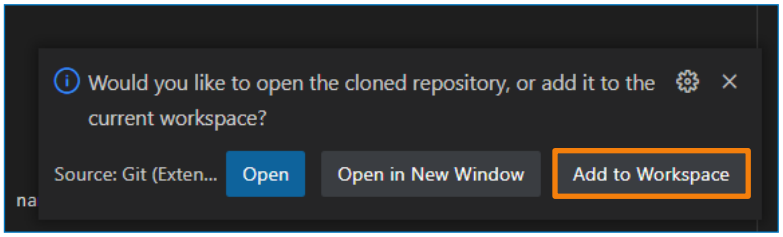
6. For the branch name, enter **lab** and press the **ENTER** key.



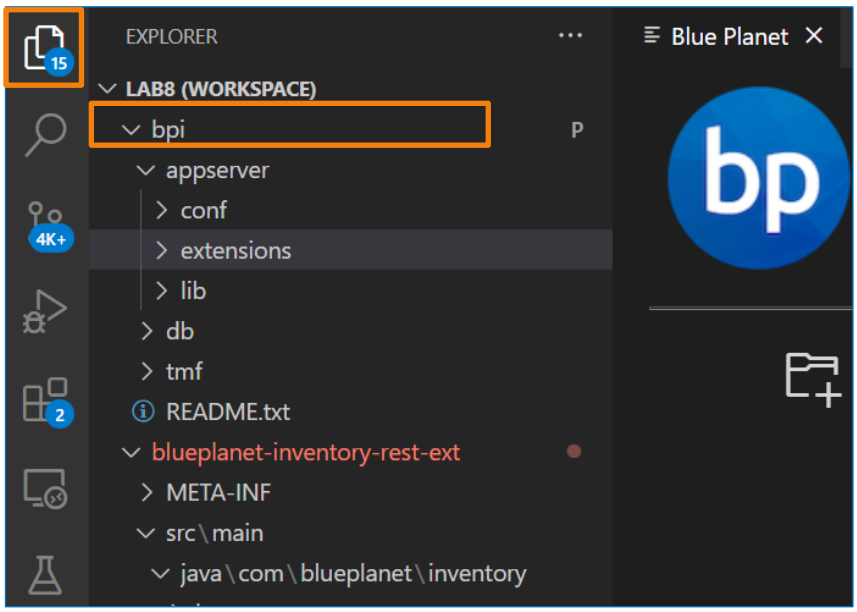
7. Select the offered default location by clicking on **Select Destination**. No other selection is required.



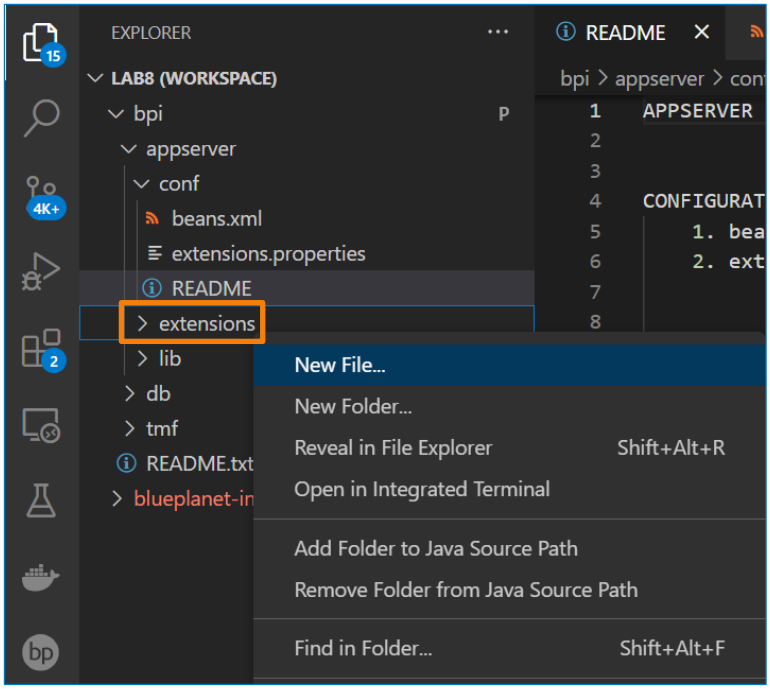
8. When asked, click the **Add to Workspace** button to add the directory structure to the VSC workspace.



9. The cloning process is now complete. Click **Explorer** from the left menu and expand **LAB8 (WORKSPACE)** to see the file structure. Expand **bpi > appserver > conf** and **bpi > appserver > extensions** to explore the directory structure. The Groovy script with your new connection naming business logic will be placed in the **extensions** directory.

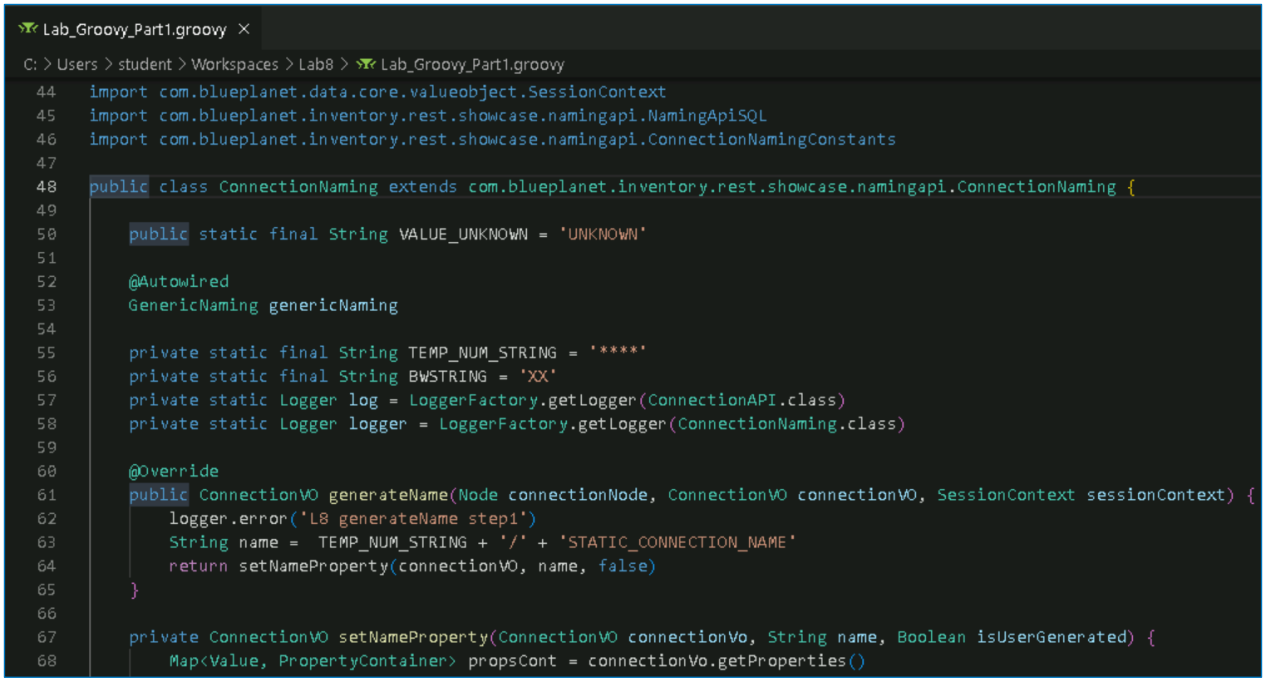


10. You will now create the Groovy file, which will override the default ConnectionNaming class. Right-click on **bpi/appserver/extensions** and select the **New File…** option.



11. For the file name, enter **ConnectionNaming.groovy**. Be careful to correctly enter the name in the proper case.

12. Some files with code snippets are already prepared on your Student PC. From the VSC Explorer panel, under the LAB8 workspace, open the **Lab_Groovy_Part1.groovy** file.



This is the minimal required code for creating a name of a connection, which also includes all the required import statements. Naming a connection essentially means creating logic for constructing a string that would be applied as the value of the **name** property of the connection node in the graph database.

1. This class extends the existing ConnectionNaming class.
2. Enable additional logging, which you monitor on the server.

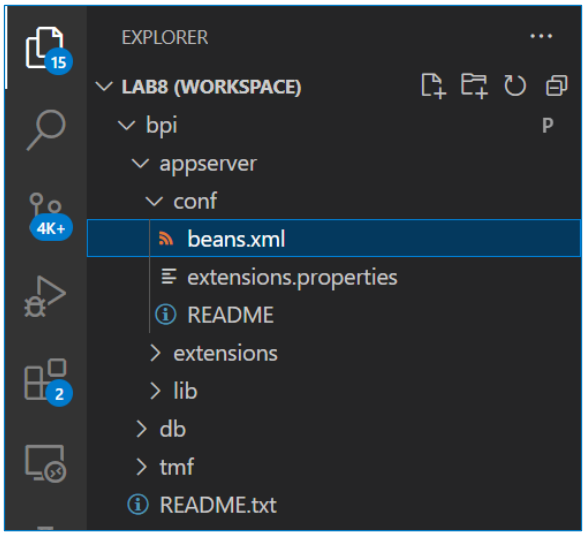
3. The generateName() method is called when a connection is created.
4. Logging messages will start with the L8 prefix.
5. The connection name is statically defined. The TEMP_NUM_STRING **** is later replaced with a 4-digit sequential number. The generated **name** is ****/STATIC_CONNECTION_NAME.
6. The connection VO and the **name** variable are passed to the setNameProperty method, which sets the **name** property of the connection.

13. Examine the code of the **setNameProperty** method.

```
private ConnectionVO setNameProperty(ConnectionVO connectionVo, String name, Boolean isUserGenerated) {
1   Map<Value, PropertyContainer> propsCont = connectionVo.getProperties()
    PropertyContainer props = propsCont.get(connectionVo.getArchetypeInstanceIdForCreation()) 2
3   props = new PropertyContainerBuilder().build()
    logger.error('L8 setNameProperty step1 connectionVO: ' + connectionVo.getProperties())
4   props.put(ProjectPropertyKeys.name, new Value(name))
    propsCont.put(new Value(connectionVo.getArchetypeInstanceIdForCreation()), props) 5
6   connectionVo.setProperties(propsCont)
    logger.error('L8 setNameProperty step2 connectionVO: ' + connectionVo.getProperties())
    return connectionVo 7
}
```

1. Define the **propsCont** map, which contains the PropertyContainer object.
2. **props** is assigned the value of the Archetype Instance ID of the connection VO, which can be used to retrieve channelization (logical interface) names.
3. Create a PropertyContainer, so that you can add the name property to it.
4. Add a new property “name” and assign it the value of the **name** variable.
5. Add the **props** property container to the **propsCont** map. The key is the Archetype Instance ID, and the value is **props**.
6. Set the properties of the connection VO from the contents of propsCont.
7. Return the connection VO, which now contains the new **name** property.

14. Before you onboard the Groovy file, you need to register the new bean. From your workspace, navigate to **bpi > appserver > conf** and open the **beans.xml** file.

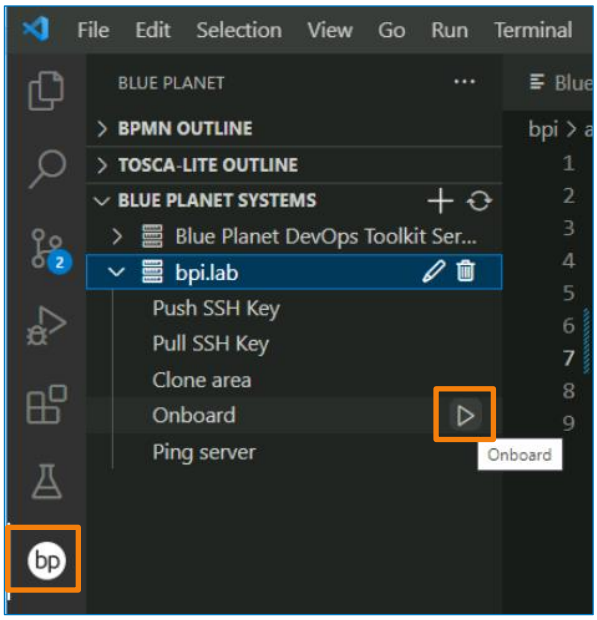


15. Enter the following statements in lines **6-7**:

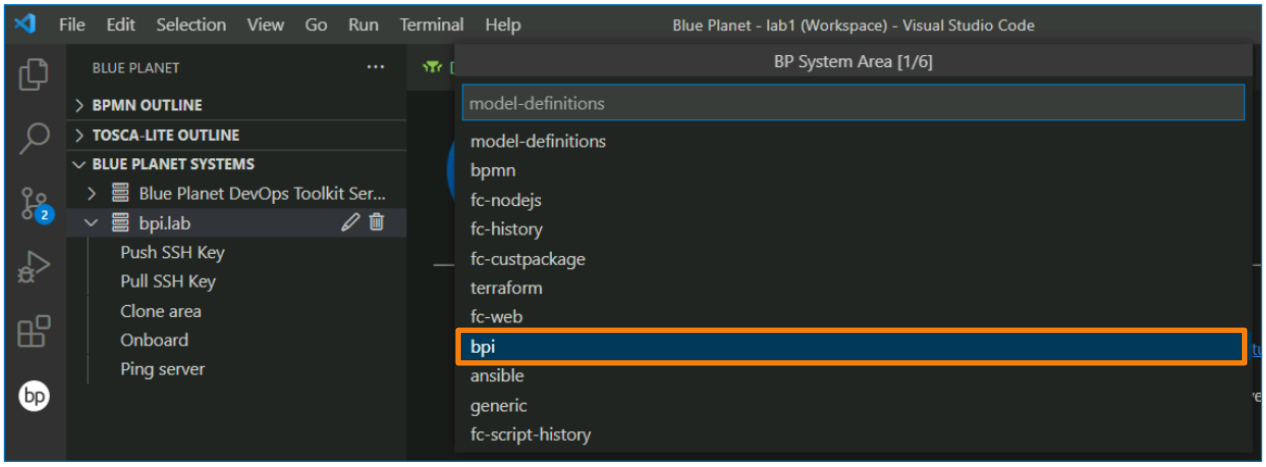
```
<lang:defaults refresh-check-delay="5000"/>

<lang:groovy id="connectionNaming" script-
   source="file:/bp2/tmp/sdk/extensions/ConnectionNaming.Groovy"></lang
   :groovy>
```

```
bpi > appserver > conf > beans.xml > beans
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3         xmlns:lang="http://www.springframework.org/schema/lang"
4         xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
5             http://www.springframework.org/schema/lang http://www.springframework.org/schema/lang/spring-lang-3.0.xsd">
6     <lang:defaults refresh-check-delay="5000"/>
7     <lang:groovy id="connectionNaming" script-source="file:/bp2/tmp/sdk/extensions/ConnectionNaming.groovy"></lang:groovy>
8  </beans>
```

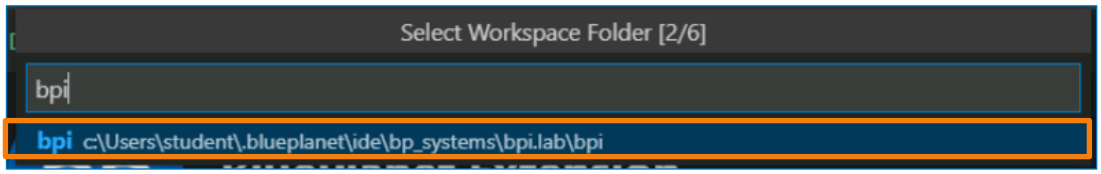
16. Save the changes by pressing **CTRL+S**. Local files are now ready to be onboarded.
17. From the left sidebar, click the **Blue Planet** icon. Under BLUE PLANET SYSTEMS, expand **bpi.lab**, and click the **triangle** icon next to the **Onboard** option to onboard the files to the server.



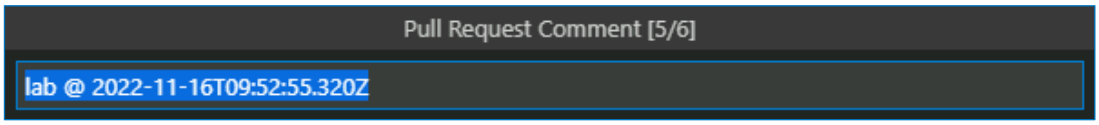
18. Click **bpi** to select the appropriate BP System Area.



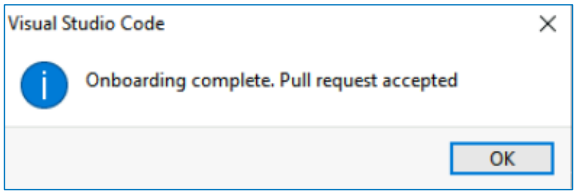
19. Select **bpi** for Select Workspace Folder. Be careful not to select other options if available.



20. When asked for Pull Request Comment, just press the **ENTER** key.



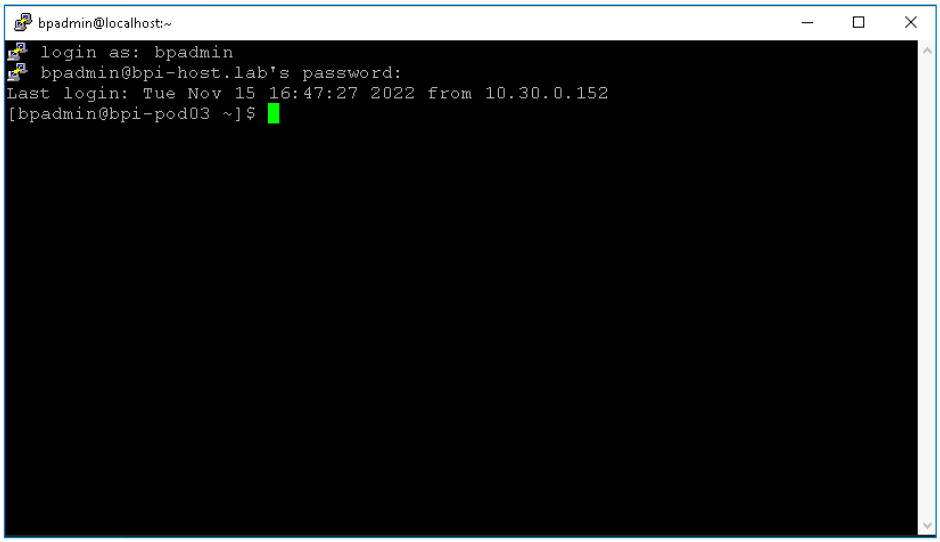
21. When onboarding is complete you will be notified. Press **OK**.



Task 2: Set Up and Monitor Logging

Before you create new connections, you will set up real-time log monitoring, which will provide you with insight into how the code is being executed. A script is prepared on the BPI host to help you with that.

1. From your Student PC, open Putty and open an SSH session to your BPI host **bpi-host.lab**.



2. View the contents of the **get_log.sh** script.

```
[bpadmin@bpi-pod03 ~]$cat get_log.sh

kubectl exec -it web-0 -- tail -fn100 /bp2/log/catalina.out | awk -F ' :
    '  '{print $1 ": " $4}' | grep L8
```

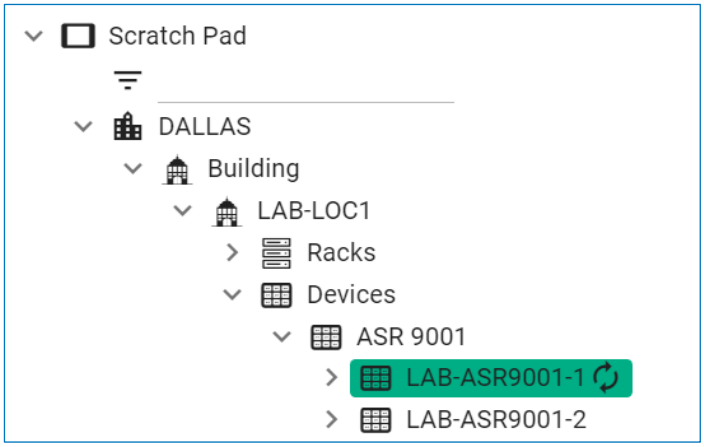
   The script monitors the Tomcat catalina.out log in the web-0 pod and prints just the lines that contain the string “L8”. You have set up the logging statements in your Groovy script to start with this string.

3. Run the **get_log.sh** script.

```
[bpadmin@bpi-pod03 ~]$./get_log.sh


```

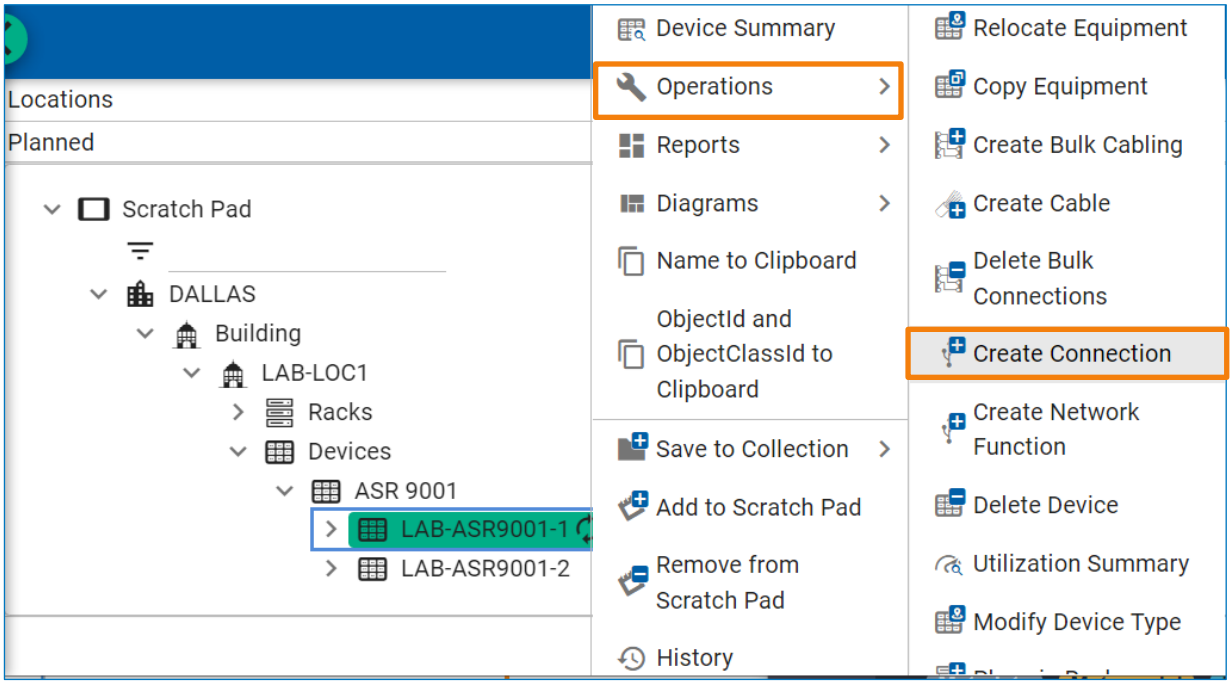
   It is expected that the script is currently returning nothing because it waits for the log entries you added to your Groovy script. Leave the script running, you will return to the terminal later.

4. You will now create a new connection in the BPI UI. From your web browser, log in to the BPI UI. In your Scratch Pad, navigate to **DALLAS > Building > LAB-LOC1 > Devices > ASR 9001**.

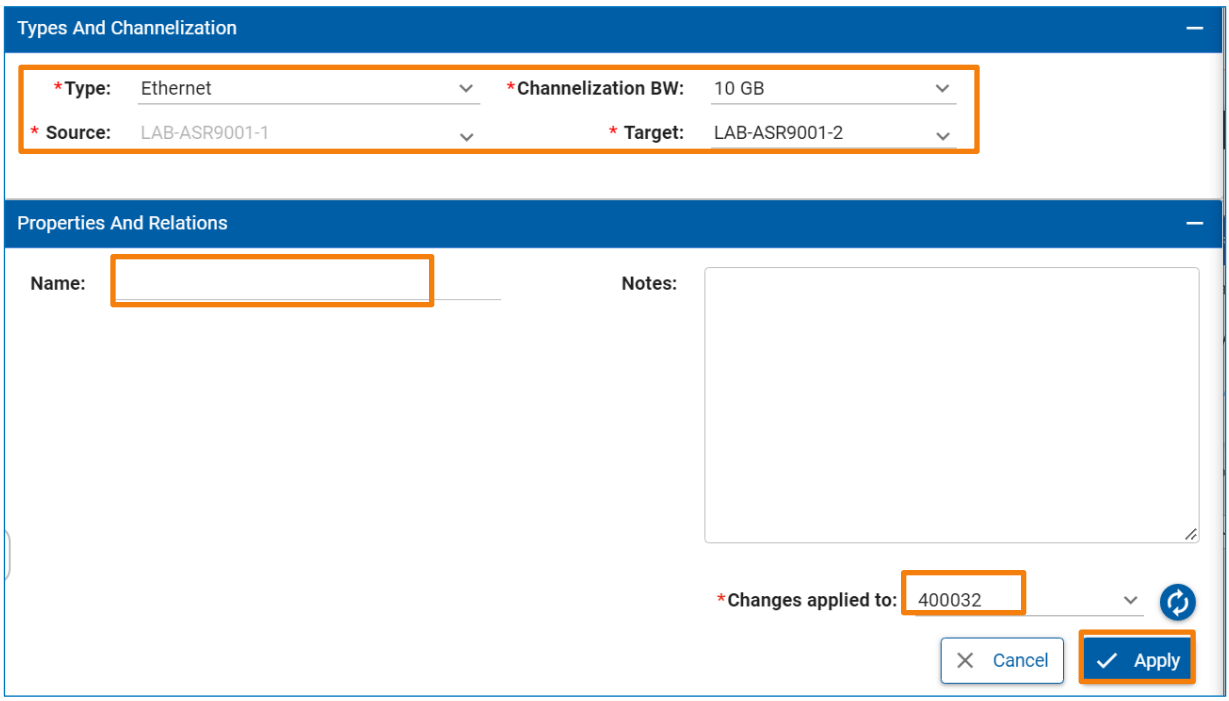


You will be creating Ethernet connections between these two devices.

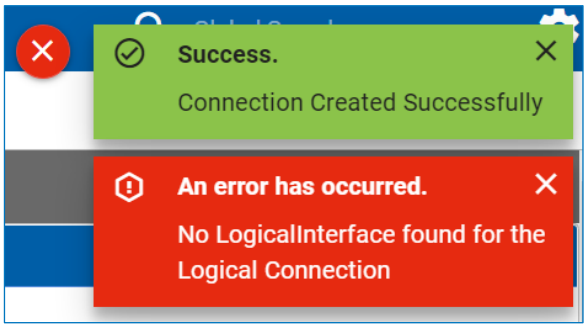
5. Right-click the **LAB-ASR9001-1** device, expand **Operations** and choose **Create Connection**.



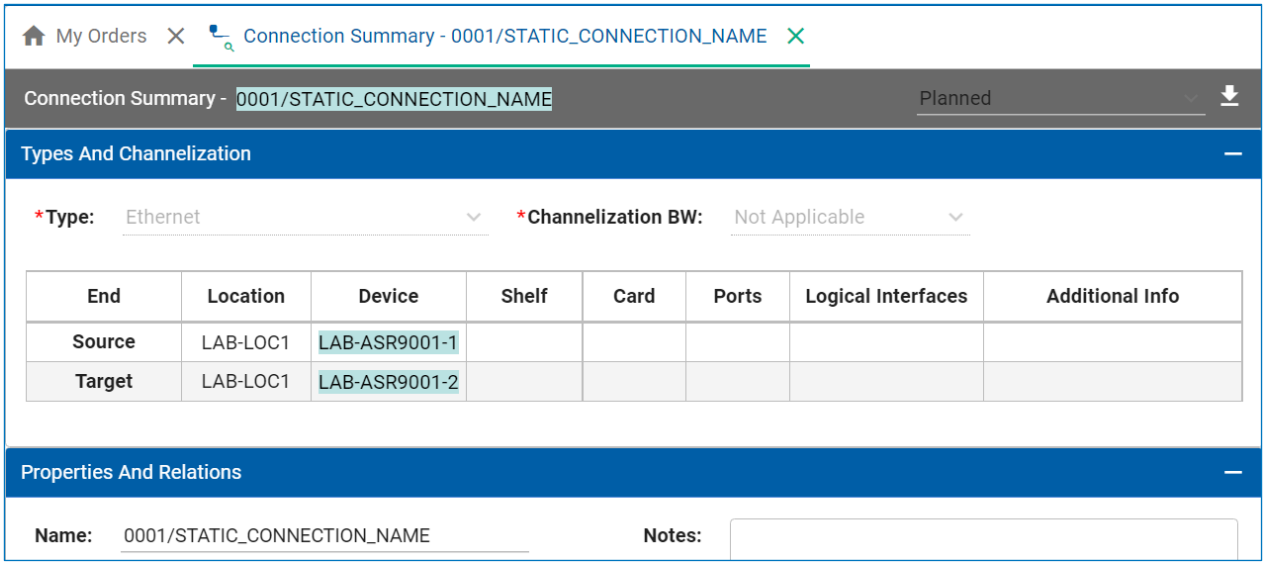
6. On the **Create Connection** page, choose **Ethernet** from the **Type** dropdown menu and **10 GB** from the **Channelization BW** dropdown. For the **Target** device, choose **LAB-ASR9001-2**, or drag that device from the Scratch Pad. Leave the **Name** field empty. Choose your order number and click **Apply**.



You should see a message that the connection was created successfully. You can ignore the error message for now. For the purpose of this exercise, we are not concerned with the connection being valid and having proper termination points. You can see that the naming logic is applied when a connection is created.



7. You can see the generated connection name at the top of the Connection Summary page, and in the Name field in the Properties and Relations panel.



NOTE: Your four-digit prefix in the connection name could be different.

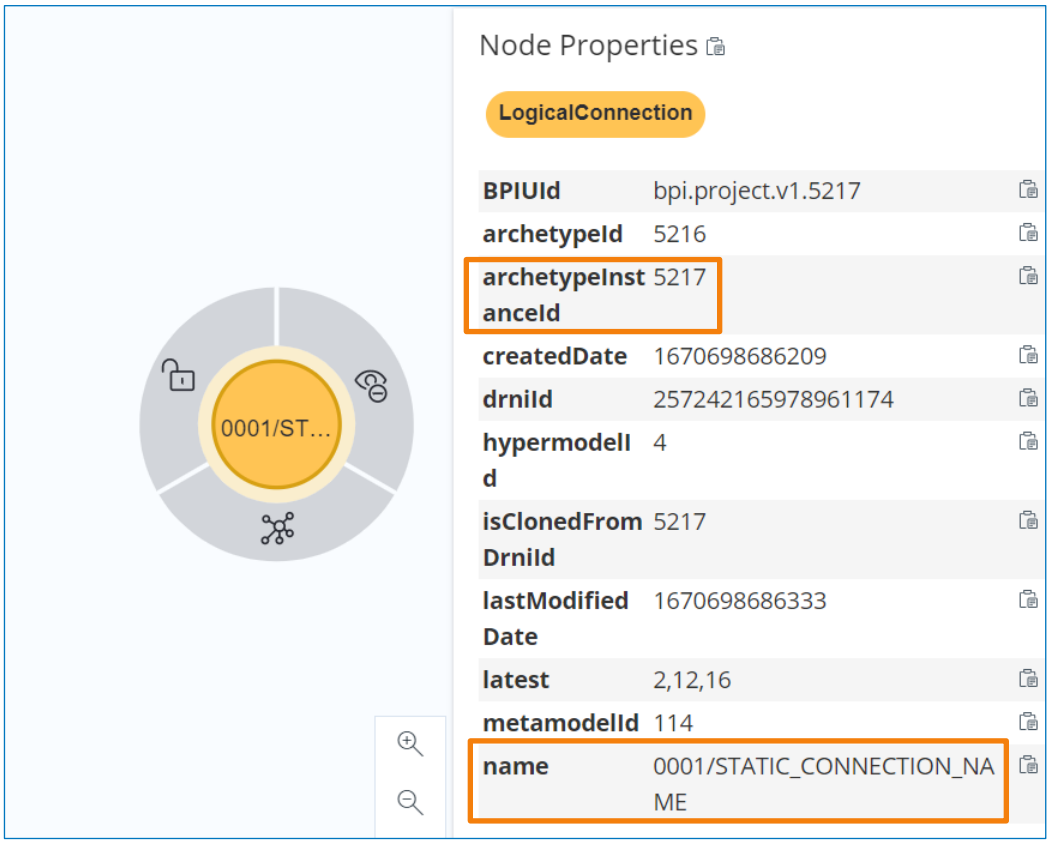
8. Go back to your terminal and observe the log output, parsed by the get_log script.

```
[bpadmin@bpi-pod01 ~]$./get_log.sh

ERROR: L8 generateName step1

ERROR: L8 setNameProperty step1 connectionVO: ['5217':[notes:null,
   channelization:6011, usage:null, name:null]]

ERROR: L8 setNameProperty step2 connectionVO:
   ['5217':[name:'****/STATIC_CONNECTION_NAME']]
```

You can see that the generateName() method was called first. The generated name is ”*****/STATIC_CONNECTION_NAME”. Remember that the *****/ prefix is replaced with the consecutive four-digit number outside the ConnectionNaming class. This default behavior ensures that connection names are unique throughout the inventory. Also notice the 5217 value of the archetypeInstanceId (the key of the properties container), and the 6011 channelization property value, which is the archetypeInstanceId of the 10 GB channelization (logical interface).

9. From your Student PC desktop, open the Neo4j Browser Desktop application. Run the following query to retrieve the LogicalConnection node you just created.

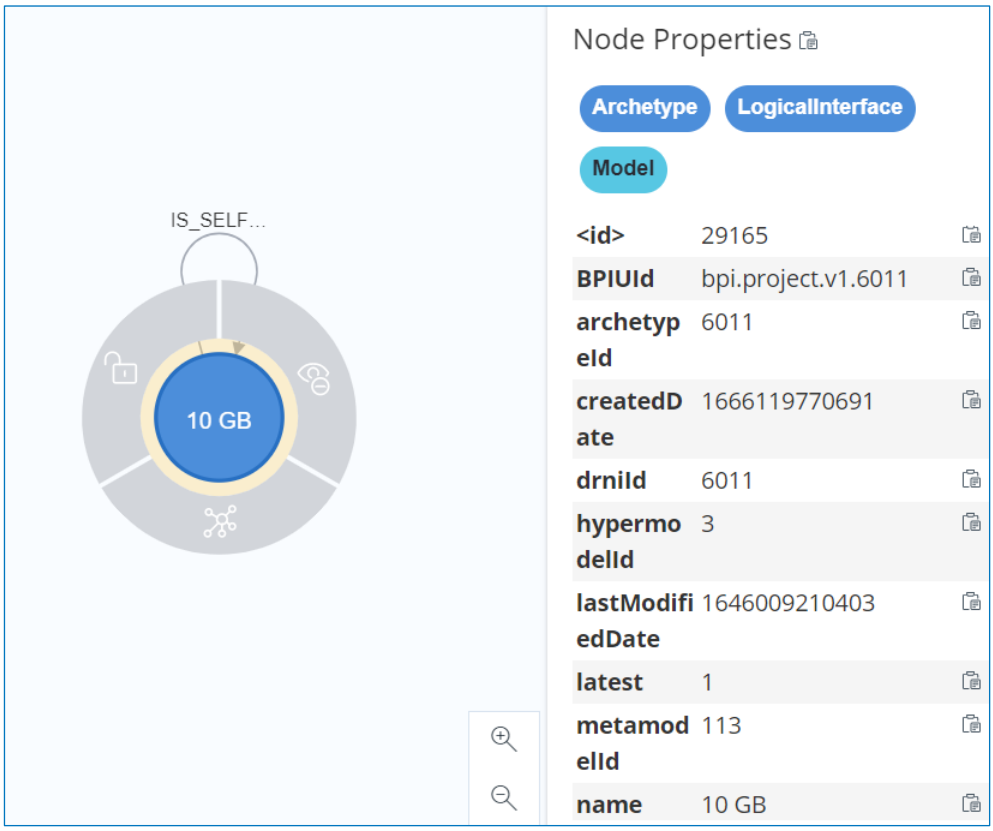
```
match (m{name:"0001/STATIC_CONNECTION_NAME"}) return m
```



Notice the **archetypeInstanceId** and the **name** property values, which match what you observed in the logs.

10. Run another query to return the Model node with the drniId **6011**, which was the value of the **channelization** property of the ConnectionVO instance you observed in the logs.

```
match(m{drniId:6011}) return m
```

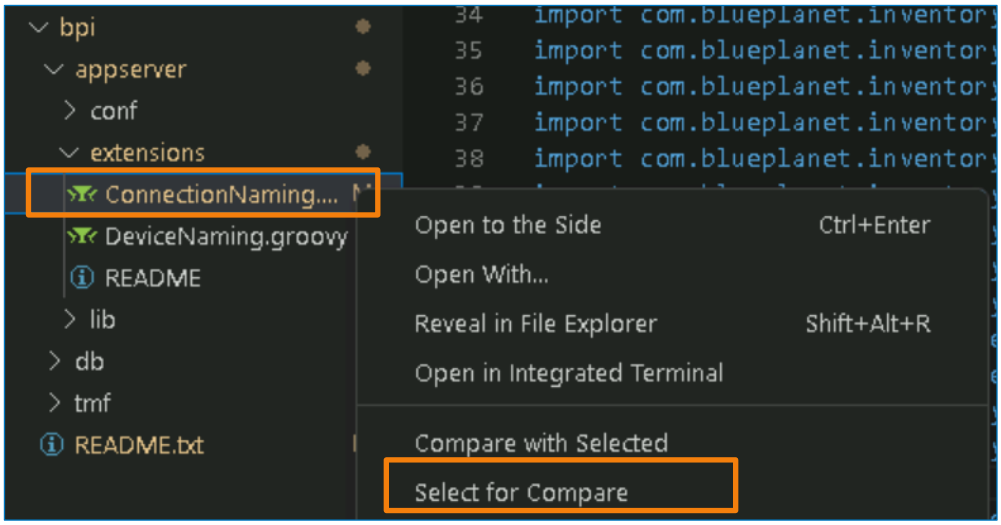


The returned node is a LogicalInterface Archetype with the name **10 GB**, which is what you selected as the Channelization BW when creating the connection in the UI.

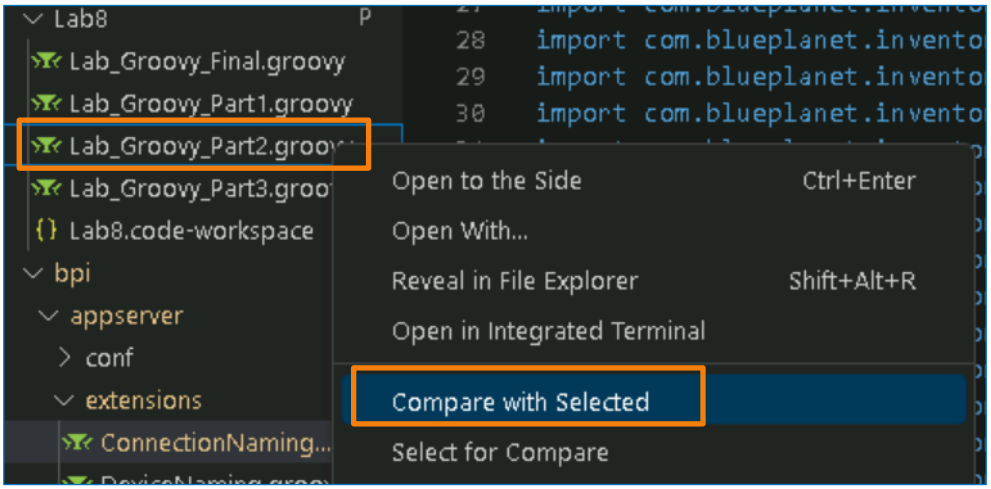
Task 3: Extend the Connection Naming Business Logic

In this task, you will add code to the ConnectionNaming Groovy script, which will generate the connection name from the connection type and bandwidth. Then you will extend that code to retrieve some properties of the connection end-points and add those values to the connection name.

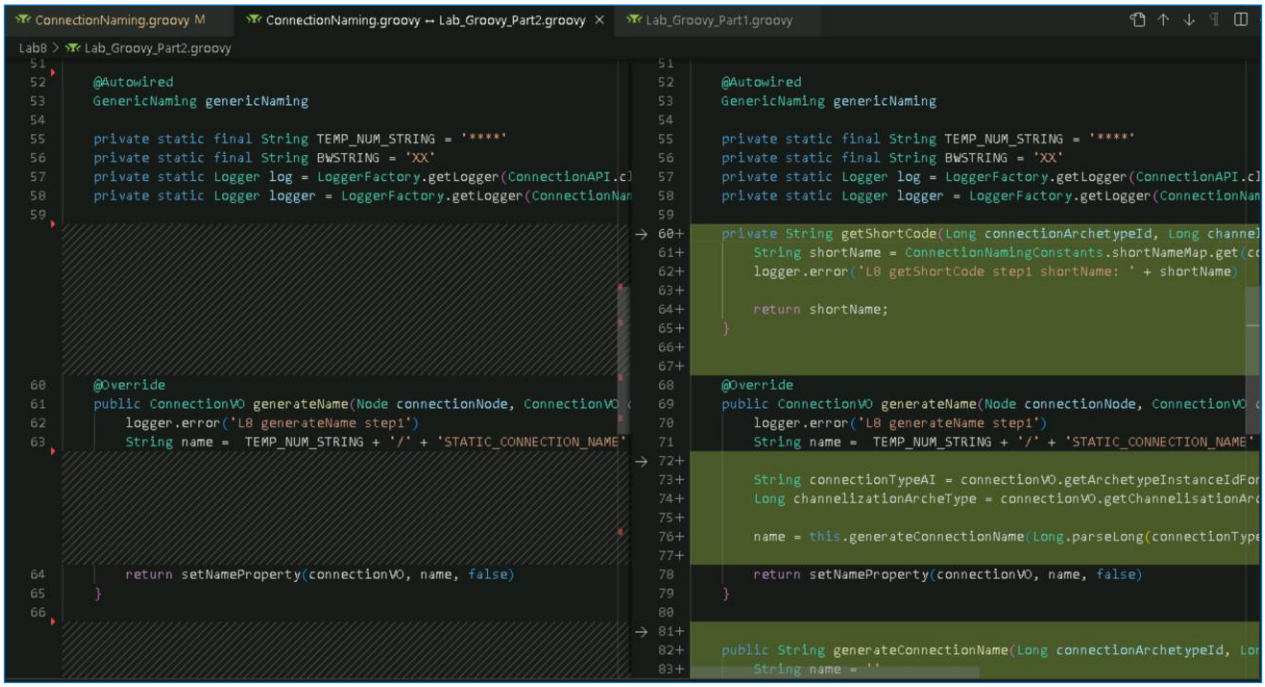
1. There are additional Groovy files prepared in your Workspace. The **Lab_Groovy_Part2.groovy** extends your current script with additional logic to retrieve the short name of the connection type and bandwidth and uses that as the name of the connection. Compare the current ConnectionNaming.groovy file with Lab_Groovy_Part2.groovy. From the workspace, right-click the **ConnectionNaming.groovy** file and choose **Select for Compare**.



2. Right-click the **Lab_Groovy_Part2.groovy** file and choose **Compare with Selected**.



3. Both files now open in separate panels, ConnectionNaming.groovy on the left, and Lab_Groovy_Part2.groovy on the right. You can see the additional code on the right marked in green.



Two new methods were added, and the generateName method was extended.

4. Examine the updated code in the **generateName** method.

```
public ConnectionVO generateName(Node connectionNode, ConnectionVO connectionVO, SessionContext
sessionContext) {
    logger.error('L8 generateName step1')
    String name =  TEMP_NUM_STRING + '/' + 'STATIC_CONNECTION_NAME'

1   String connectionTypeAI = connectionVO.getArchetypeInstanceIdForCreation();
2   Long channelizationArcheType = connectionVO.getChannelisationArchetype();

3   name = this.generateConnectionName(Long.parseLong(connectionTypeAI), channelizationArcheType,
    sessionContext);

    return setNameProperty(connectionVO, name, false)
}
```

1. Get the Archetype Instance ID of the connection VO.
2. Get the channelization (logical interface) Archetype of the VO.
3. Call the new **generateConnectionName** method, which will apply additional logic.

5. Examine the code of the new **generateConnectionName** method.

```
public String generateConnectionName(Long connectionArchetypeId, Long channelizationArcheType,
SessionContext sessionContext) {
    String name = ''

    logger.error('L8 generateConnectionName: ' + connectionArchetypeId.toString())

1   String shortCode = getShortCode(connectionArchetypeId, channelizationArcheType, sessionContext)

    name = "****/" + shortCode
2   return name
}
```

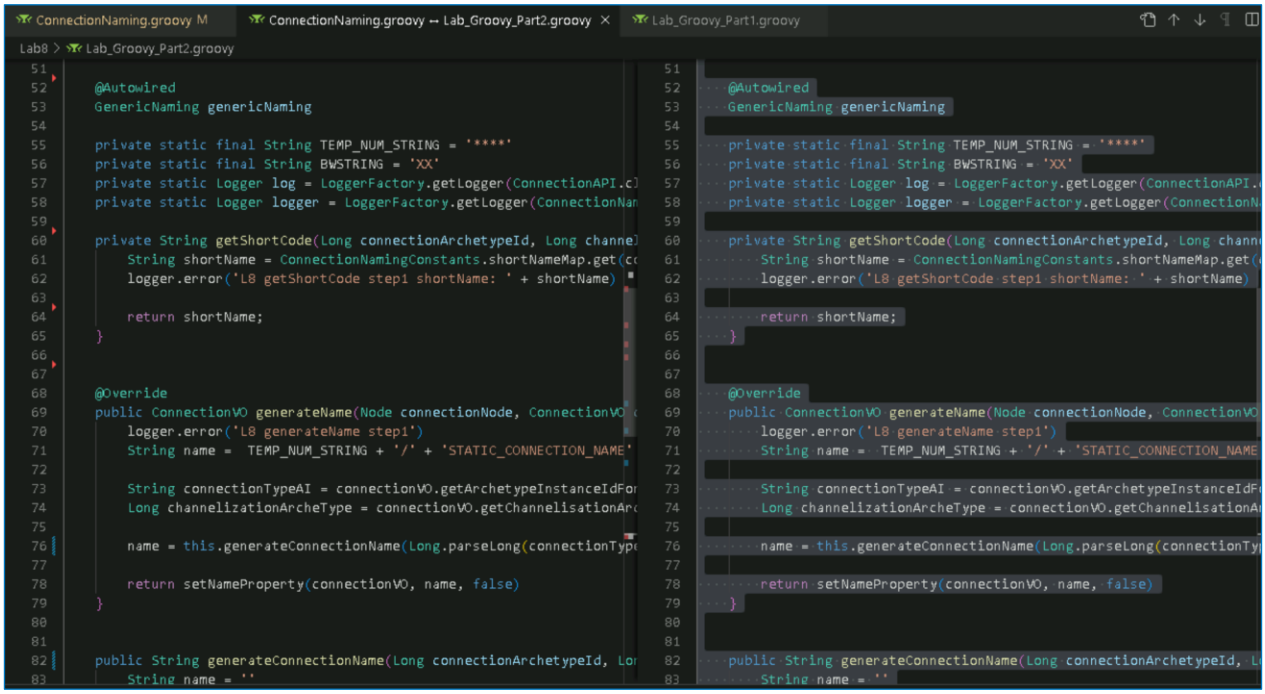
1. Call the new **getShortCode** method to retrieve the short name from the connection Archetype Instance ID and the channelization (LogicalInterface) Archetype name.
2. Add the temporary string, which will be replaced by the number sequence, and return the name.

6. Examine the code of the new **getShortCode** method.

```
private String getShortCode(Long connectionArchetypeId, Long channelizationArcheType,
SessionContext sessionContext) {
1   String shortName = ConnectionNamingConstants.shortNameMap.get(connectionArchetypeId);
    logger.error('L8 getShortCode step1 shortName: ' + shortName)

    return shortName;
}
```

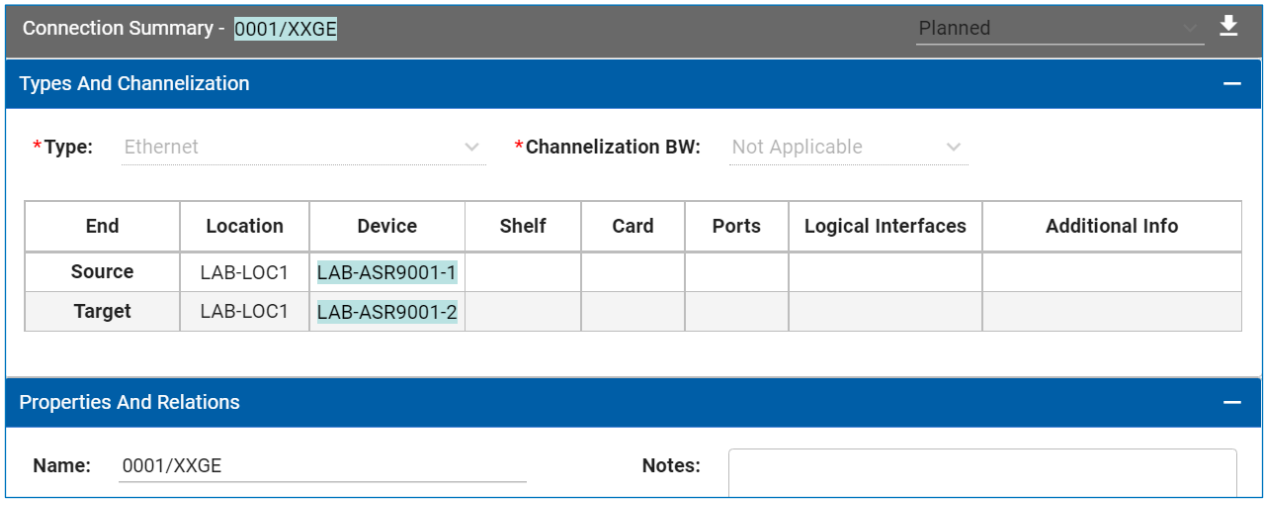
1. This simple method retrieves the short name of the connection type from the Archetype ID mapping in the **ConnectionNamingConstants** class. In your example, this will be “XXGE”.

7. You will now replace the contents of your **ConnectionNaming.groovy** file. **Select and copy** to the clipboard all the content of the **Lab_Groovy_Part2.groovy** file on the right (**CTRL+A**, **CTRL+C**). Then move to the left panel, **select and delete** all content of the ConnectionNaming.groovy file and **paste** the contents of the clipboard (**CTRL+A**, **CTRL+V**).



Both files should now have the same content and Compare should show no differences.

8. Save the changes to the ConnectionNaming.groovy file by pressing **CTRL+S**.
9. You can now repeat the steps from a previous task to onboard the changes to the BPI server. From the left sidebar, click the **Blue Planet** icon. Under BLUE PLANET SYSTEMS, expand **bpi.lab**, click the triangle icon next to the **Onboard** option, and confirm the appropriate options to onboard the bpi area to the server.
10. From the BPI UI, repeat the steps from the previous task to create a new 10 GB Ethernet connection between LAB-ASR9001-1 and LAB-ASR-9001-2.



The name of the connection was now constructed with the string XXGE, which is the value, collected from the ConnectionNamingConstants class. You can also see that in the log file.

```
ERROR: L8 getShortCode step1 shortName: XXGE
```

11. You will now add logic that will replace the “XX” part of the connection name with the actually selected bandwidth of the connection. In VSC, compare the current contents of the ConnectionNaming.groovy file with the **Lab_Groovy_Part3.groovy** file.
12. The new code extends the logic in the **getShortCode** method.

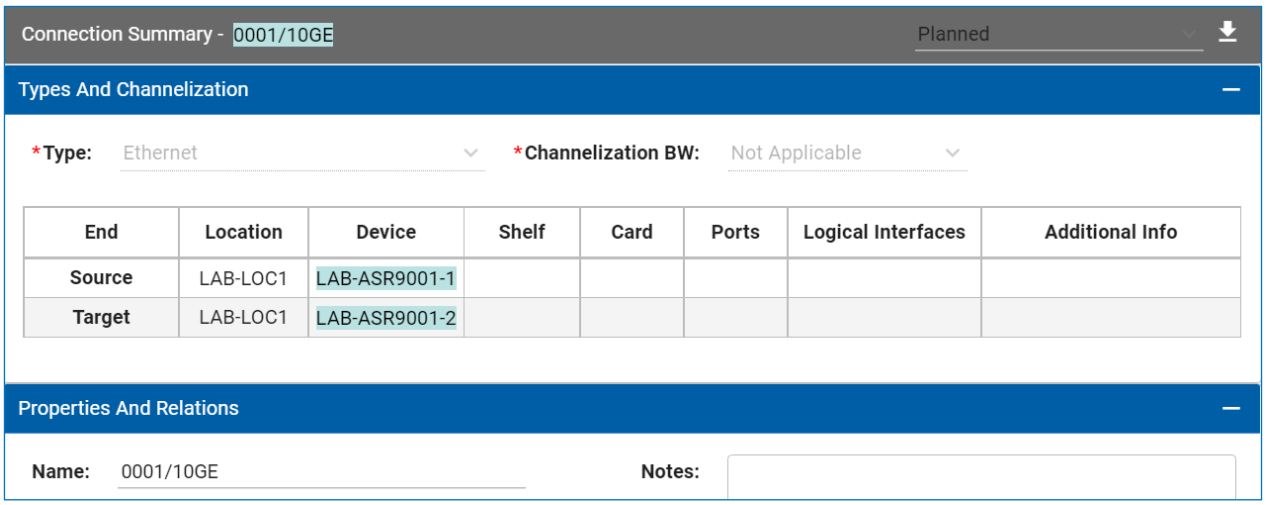
```
private String getShortCode(Long connectionArchetypeId, Long channelizationArcheType, SessionContext sessionContext) {
    String shortName = ConnectionNamingConstants.shortNameMap.get(connectionArchetypeId);
    logger.error('L8 getShortCode step1 shortName: ' + shortName)

    if ((channelizationArcheType != null) && ((shortName != null) && (shortName.contains('XX')))) {
1       Node bw = inventoryDao.fetchNode(ProjectLabels.Archetype, new Value(channelizationArcheType), SecurityUtils.get
        if (bw != null) {
2           String bandwidthName = bw.getPropertyAsString(ProjectPropertyKeys.name)
            logger.error('L8 getShortCode step2 bandwidthName: ' + bandwidthName)
            String bwName = ''
            //the bandwidth name should have the number followed by a space
            //if not we will just take the first two char.
            if (bandwidthName.length() > 1) {
                if (bandwidthName.contains(' ')) {
3                   bwName = bandwidthName.substring(0, bandwidthName.indexOf(' ') + 1)
                    } else {
                    bwName = bandwidthName.substring(0, 2)
                }
                } else {
                bwName = bandwidthName
            }
            bwName = bwName.trim()
4           shortName =  shortName.replace(BWSTRING, bwName)
        }
    }
    logger.error('L8 getShortCode step3 shortName: ' + shortName)

    return shortName;
}
```

1. Use the **fetchNode** method of the **inventoryDao** object to retrieve the LogicalInterface Archetype where the name matches the **channelizatonArcheType** variable value. In your example, this will be the “10 GB” LogicalInterface Archetype.
2. Set the **bandwidthName** variable to the name of the Archetype (“10 GB”).
3. The name of the Archetype is then parsed to return just the numerical bandwidth value (“10”).
4. This numerical value then replaces the “XX” part of the shortName variable.

13. **Select and copy** to the clipboard all the content of the **Lab_Groovy_Part3.groovy** file on the right (**CTRL+A**, **CTRL+C**). Then move to the left panel, **select and delete** all content of the **ConnectionNaming.groovy** file and **paste** the contents of the clipboard (**CTRL+A**, **CTRL+V**).

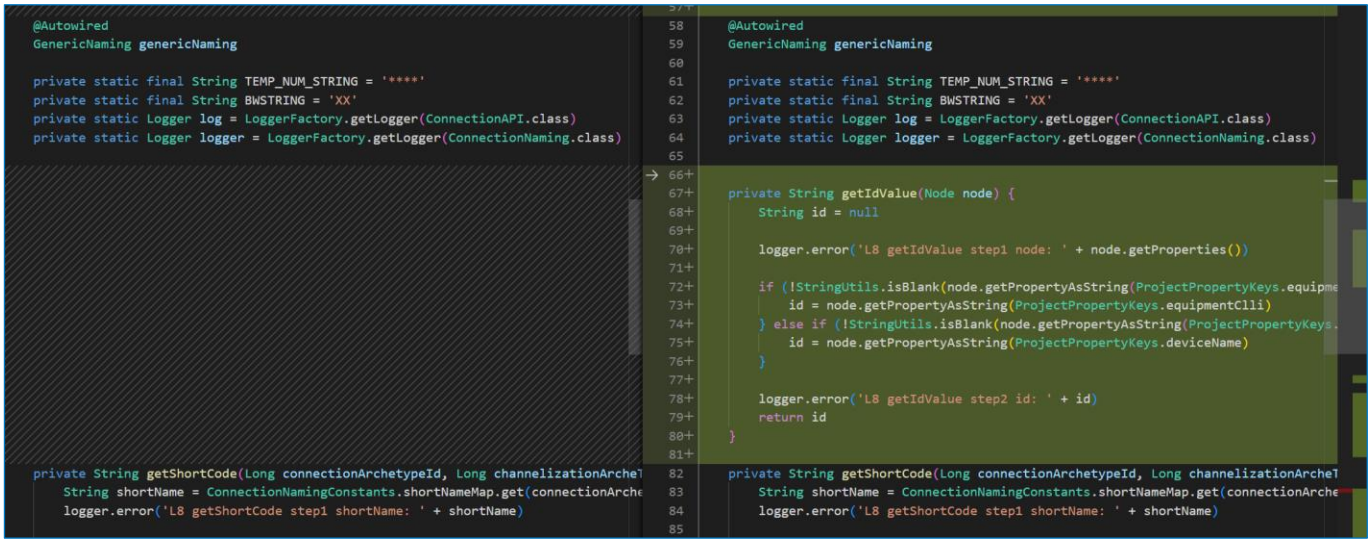
    Both files should now have the same content and Compare should show no differences.
14. Save the changes to the ConnectionNaming.groovy file by pressing **CTRL+S**.
15. Onboard the changes to the BPI server again.

16. From the BPI UI, repeat the steps from the previous task to create a new 10 GB Ethernet connection between LAB-ASR9001-1 and LAB-ASR-9001-2.



You can see that the connection name now contains the whole bandwidth value, which is 10GE.

17. When naming connections, you can also retrieve information from the connection end points (A-end and Z-end) and add that to the connection name. In VSC, compare the current contents of the ConnectionNaming.groovy file with the **Lab_Groovy_Part4.groovy** file.
18. Examine the differences between the two files.

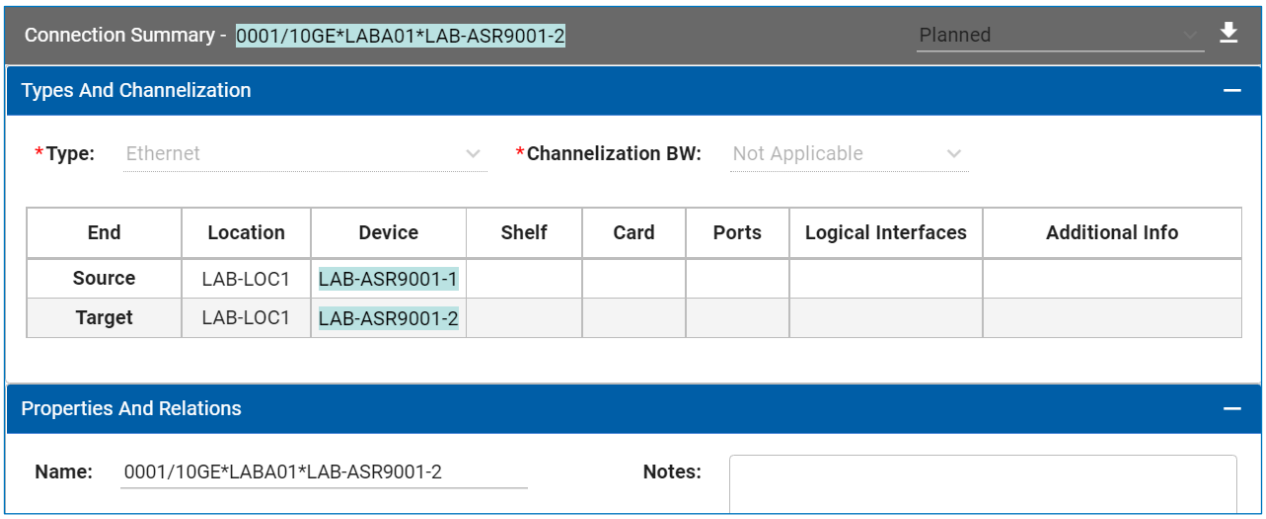


New methods are defined to get the ends (Source and Target device) of the connection and provide the ID of the end devices.

19. **Select and copy** to the clipboard all the content of the **Lab_Groovy_Part4.groovy** file on the right (**CTRL+A**, **CTRL+C**). Then move to the left panel, select all content of the ConnectionNaming.groovy file and paste the contents of the clipboard (**CTRL+A**, **CTRL+V**).

    Both files should now have the same content and Compare should show no differences.
20. Save the changes to the ConnectionNaming.groovy file by pressing **CTRL+S**.
21. Onboard the changes to the BPI server again.
22. Note that you could retrieve any device property and use it in the connection name.

23. From the BPI UI, repeat the steps from the previous task to create a new 10 GB Ethernet connection between LAB-ASR9001-1 and LAB-ASR-9001-2.



The connection name is now generated in the following format:

<unique number prefix>/<connection bandwidth>*<source device CLLI>*<target device name>

Notice that the target device does not have the equipmentCLLI property, so the device name was chosen as the ID.

24. Note that the current code is sparse and covers a specific example. You can refer to the **Lab_Groovy_Full.groovy** file for more mature code that includes sanity checks and error handling. Copy the contents of this file to your ConnectionNaming.groovy script and onboard it to complete this lab.