blueplanet®
a division of Ciena

Introduction to Networking and Programmability

BPF002ILT, Revision 1.0.1

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
	1.0	Mar 2023	Initial release
	1.0.1	Apr 2023	Minor content and graphic update.

Contents

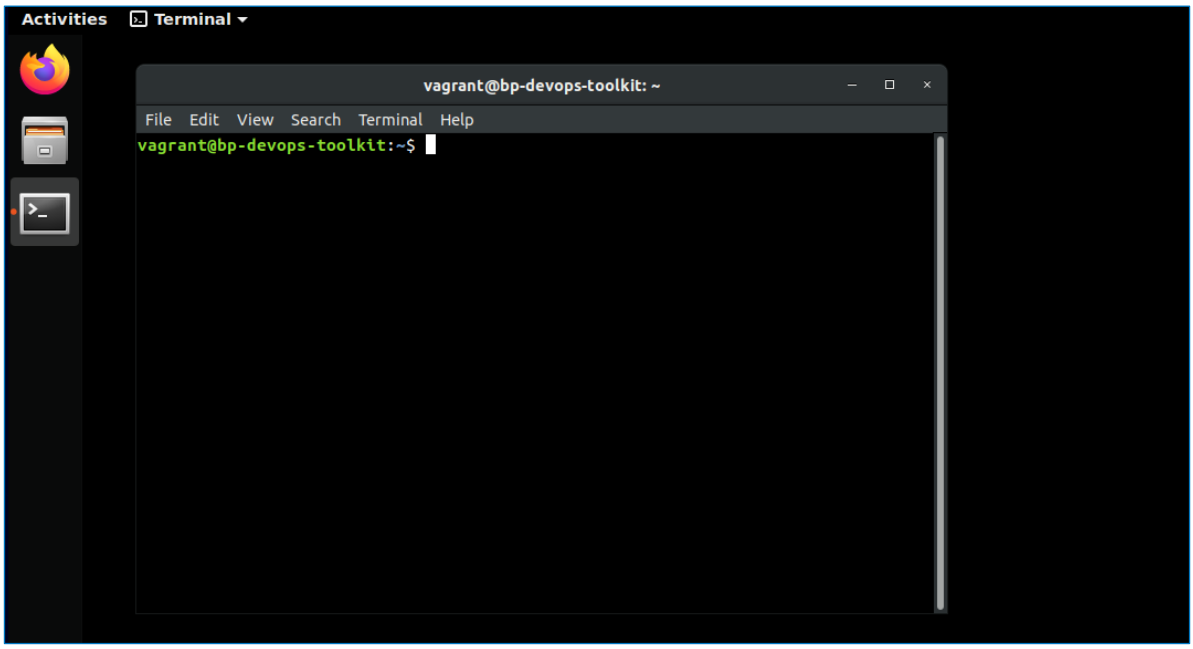
Lab 1: Prepare the Working Environment

Objectives

- Installation of the Java development kit (JDK) and Maven
- Setting up the development environment with the Git version control system
- Get to know the Microsoft Visual Studio Code (VS Code) editor and install extensions

Task 1: Install Java Development Kit and Maven

1. Open a new **terminal window** on your development machine.



2. Make sure that Java is not installed already.

```
$ java -version
bash: java: command not found
```

3. Install JDK. The binaries are already downloaded and located in your home directory.

```
$ sudo dpkg -i jdk-11.0.15.1_linux-x64_bin.deb
(Reading database ... 112228 files and directories currently installed.)
Preparing to unpack jdk-11.0.15.1_linux-x64_bin.deb ...
Unpacking jdk-11 (11.0.15.1-1) over (11.0.15.1-1) ...
Setting up jdk-11 (11.0.15.1-1) ...
```

In this lab environment, you install JDK 11 by unpacking the previously downloaded binaries. In other environments, you can install the JDK distributions using standard packaging tools such as **apt**.

4. Inspect the location of the unpacked binaries.

```
$ dpkg -L jdk-11
/.
/usr
/usr/lib
/usr/lib/jvm
/usr/lib/jvm/jdk-11
<… output omitted …>
```

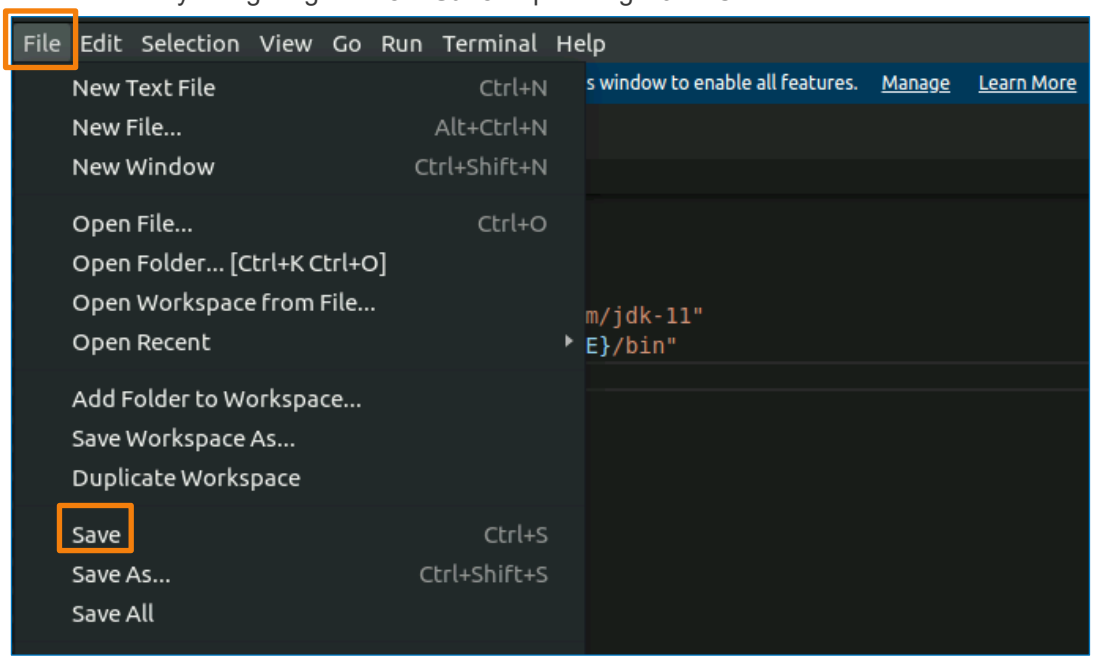
5. Open the ~/.bashrc file in visual studio code by executing the following command.

```
$ code ~/.bashrc
```

6. Scroll down and add the following lines to the end of the file.

```
export JAVA_HOME="/usr/lib/jvm/jdk-11"
export PATH="$PATH:${JAVA_HOME}/bin"
```

7. Save the file by navigating to **File > Save** or pressing **Ctrl + s**.



The ~/.bashrc file is executed each time a bash shell is opened.

8. Close the existing terminal window and open it again.
9. Inspect if Java was installed correctly.

```
$ java -version
java version "11.0.15.1" 2022-04-22 LTS
Java(TM) SE Runtime Environment 18.9 (build 11.0.15.1+2-LTS-10)
```

```
Java HotSpot(TM) 64-Bit Server VM 18.9 (build 11.0.15.1+2-LTS-10, mixed
mode)
```

10. Install the Maven build system by extracting the already downloaded binaries.

```
$ sudo tar xf ~/apache-maven-3.8.6-bin.tar.gz -C /opt
```

11. Go back to the ~/.bashrc file in Visual Studio Code. If you closed the previously opened window you can open it by executing the following command again.

```
$ code ~/.bashrc
```

12. Add the highlighted lines at the end of the file.

```
export JAVA_HOME="/usr/lib/jvm/jdk-11"
export PATH="$PATH:${JAVA_HOME}/bin"
export M2_HOME=/opt/apache-maven-3.8.6
export MAVEN_HOME=/opt/apache-maven-3.8.6
export PATH=${M2_HOME}/bin:${PATH}
```

13. Do not forget to save the file by navigating to **File > Save** or pressing **Ctrl + s**.
14. Close and reopen the terminal window and verify that the maven was successfully installed.

```
$ mvn -version
Apache Maven 3.8.6 (84538c9988a25aec085021c365c560670ad80f63)
Maven home: /opt/apache-maven-3.8.6
Java version: 11.0.15.1, vendor: Oracle Corporation, runtime:
/usr/lib/jvm/jdk-11
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "4.15.0-193-generic", arch: "amd64", family:
"unix"
```

You have successfully installed Java and its dependencies to develop and run Java applications.

Task 2: Prepare Development Environment with Git Version Control

1. Open a terminal window and create a new directory. Use the <***pod-name***>**-bpi-training** nomenclature, for example, **pod01-bpi-training**.

```
$ mkdir <pod name>-bpi-training
```

2. Enter the newly created directory and initialize a new git repository.

```
$ cd <pod name>-bpi-training
$:~/<pod name>-bpi-training$ git init --initial-branch=main
Initialized empty Git repository in /home/vagrant/<pod name>-bpi-training/.git/
```

3. Configure the git repository with your email and name.

```
$:~/name-bpi-training$ git config --global user.email you@example.com
$:~/name-bpi-training$ git config --global user.name "Your Name"
```

4. Copy the skeleton code from a temporary directory /lab1/ to the new git repository.

```
$:~/name-bpi-training$ cp -pr ~/lab1/bpi-prereq-training-java/* .
```

5. List the repository content.

```
$:~/name-bpi-training$ ls
Dockerfile  entrypoint.sh  java-netconf-client  README.md  yang
```

6. Check the files that are in the staging area and tracked by git.

```
$:~/name-bpi-training$ git status
On branch main
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        Dockerfile
        README.md
        entrypoint.sh
        java-netconf-client/
        yang/

nothing added to commit but untracked files present (use "git add" to track)
```

The git status command displays the state of the working directory and staging area. It allows you to see staged changes and files that are not tracked by Git. Currently, none of the files in the repository are tracked by Git.

7. Stage all untracked files in the repository.

```
$:~/name-bpi-training$ git add .
```

Using "**.**" (dot) you have staged all untracked changes in the repository. Sometimes you want to stage only a single file or folder. In that case, you must use the path to the file or folder instead of the dot, for example, **git add java-netconf-client** would stage only this directory together with its content. Adding untracked changes to the staged area is the first step to committing or taking a snapshot of the repository state.

8. Inspect again the state of the staged files.

```
$:~/name-bpi-training$ git status
On branch master
No commits yet
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   Dockerfile
        new file:   README.md
        new file:   entrypoint.sh
        new file:   java-netconf-client/pom.xml
<… output omitted …>
```

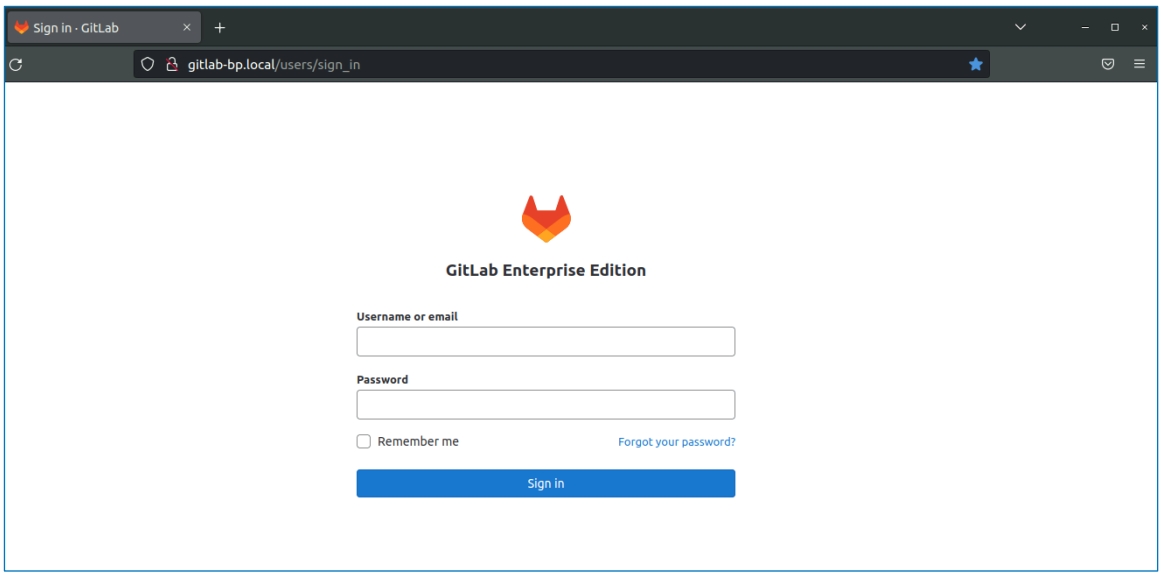
9. Commit the staged changes.

```
$:~/name-bpi-training$ git commit -m "initial repository changes"
[master (root-commit) 5d4c25a] initial repository changes
14 files changed, 875 insertions(+)
 create mode 100644 Dockerfile
 create mode 100644 README.md
 create mode 100755 entrypoint.sh
 create mode 100644 java-netconf-client/pom.xml
 <… output omitted …>
```

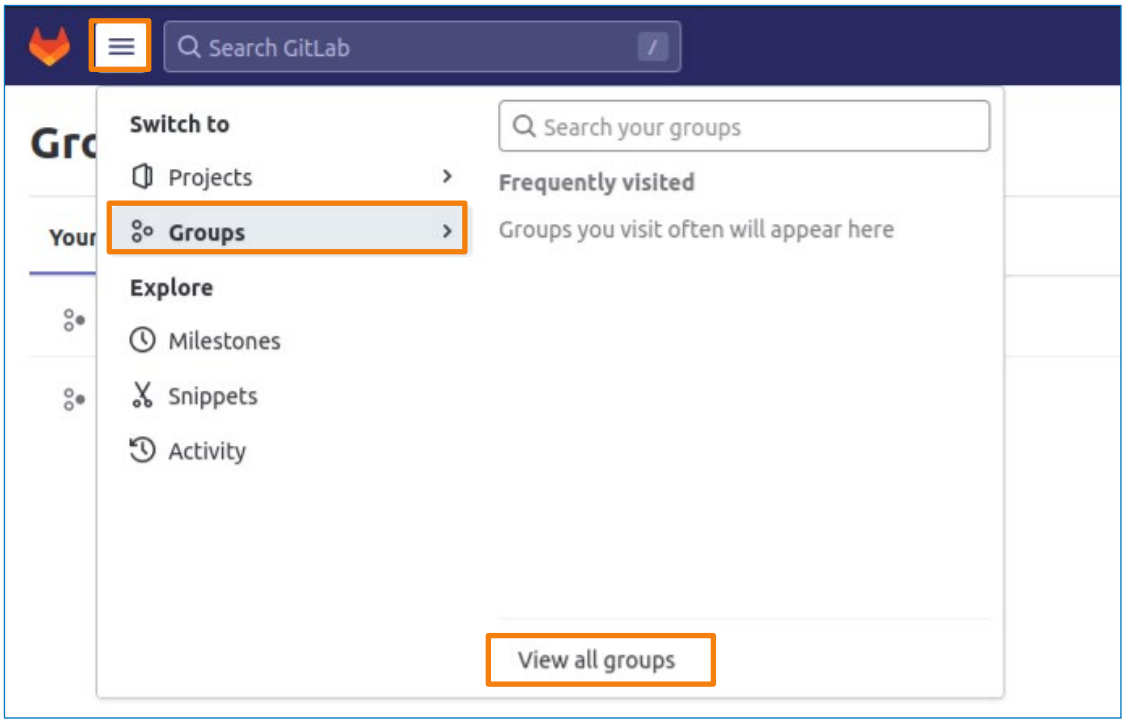
The git commit command creates a snapshot of the current changes to the project. Commit snapshots can be considered "safe" versions of a project. Only staged files are committed to the repository.

The changes are now safely stored with Git, but only within your local workspace. In a proper development environment, the committed changes are frequently uploaded to a central Git hosting service, where other developers can see your changes and integrate their own features into your code, and vice versa. Popular Git hosting services include GitHub, GitLab, BitBucket, and others.

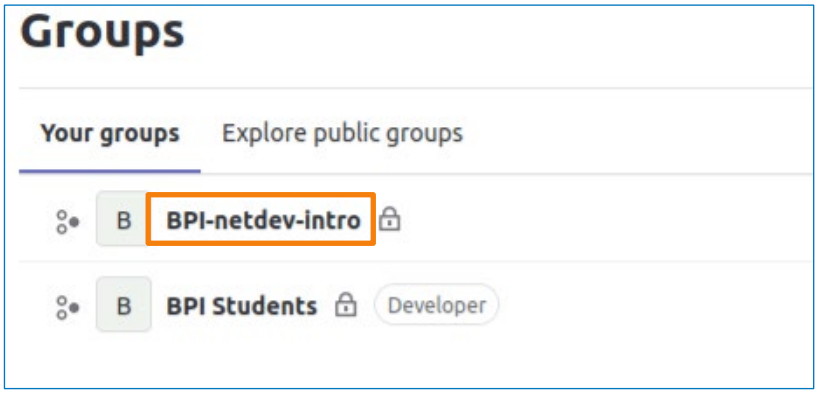
10. Open the Firefox browser on your development machine and navigate to *http://gitlab-bp.local/* or click the gitlab bookmark. Sign in using the credentials provided by the instructor.



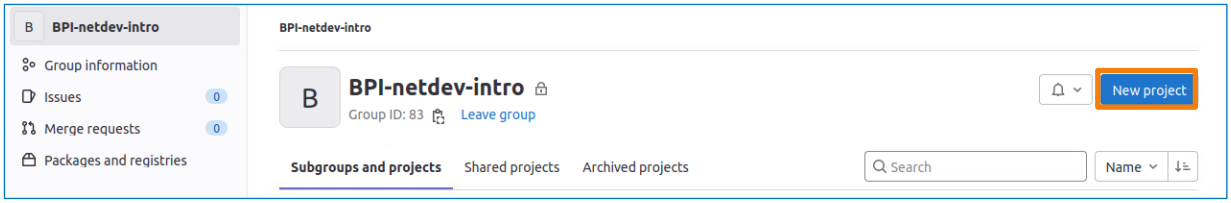
11. Find the group named **BPI-netdev-intro** in the **Groups > View all groups** menu.



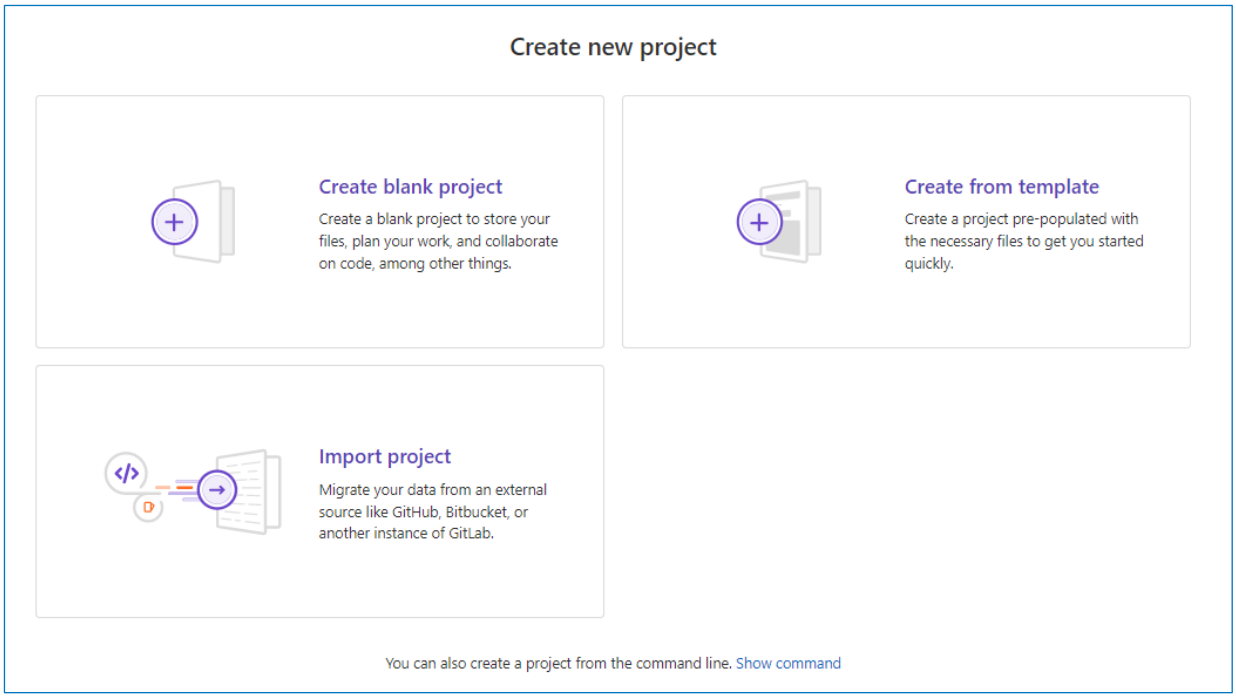
12. Click the **BPI-netdev-intro** group to open it.



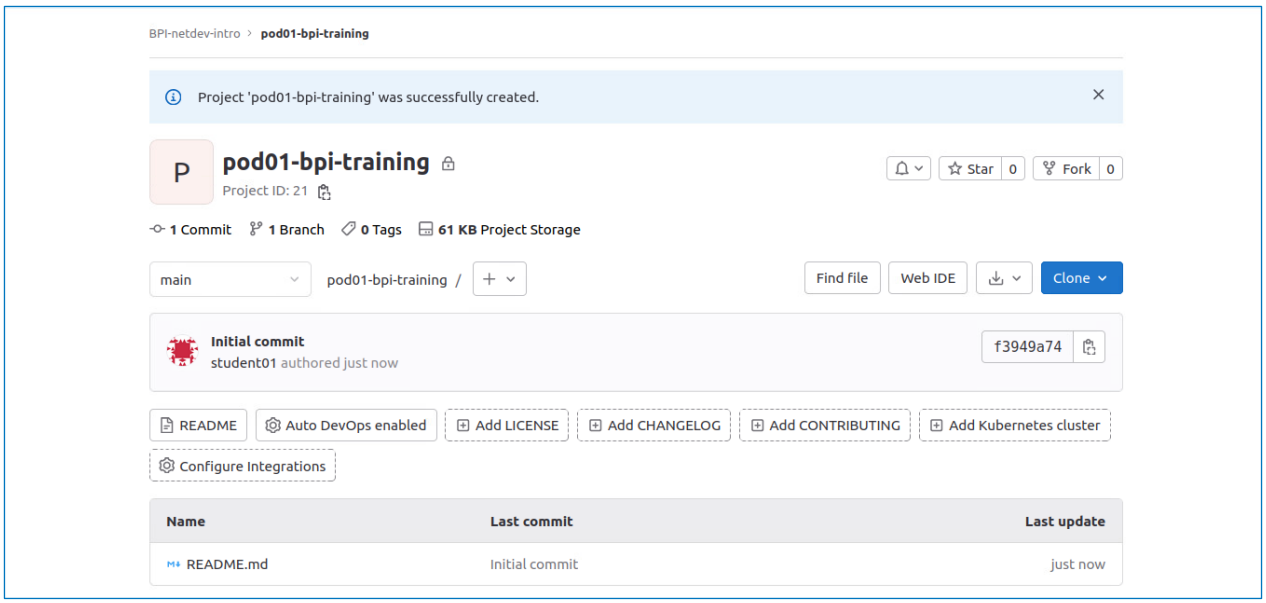
13. In the upper right corner click on the **New Project** button.



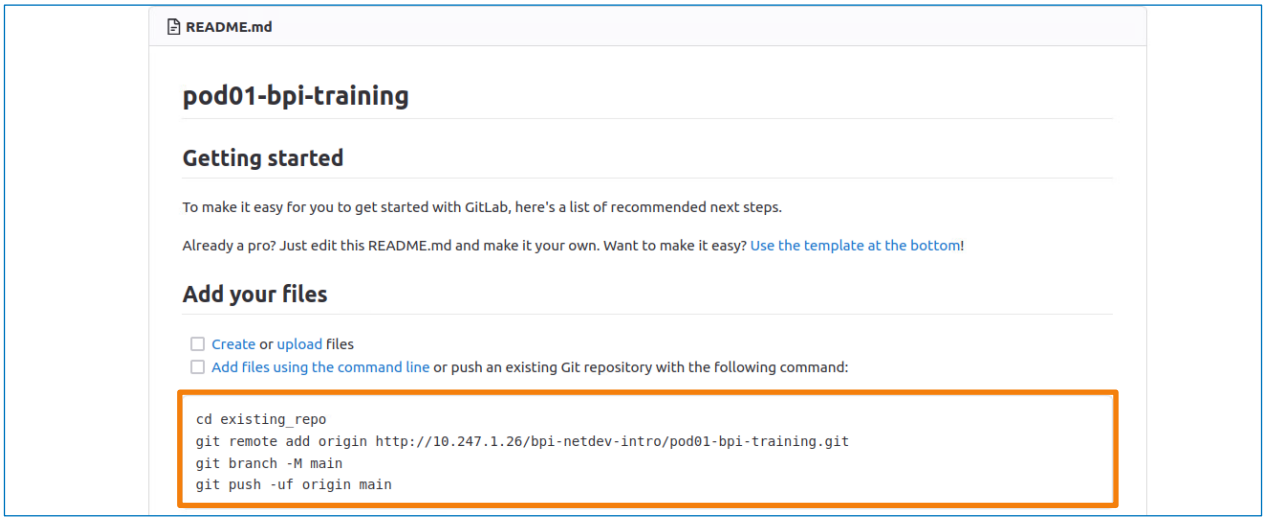
14. Select the **Create blank project** button.



15. As the project name type in the name of your local repository, ***pod-name*-bpi-training** (for example **pod01-bpi-training**), and then click **Create project**.



16. In the **Readme.md** file, you will notice the instructions for pushing an existing git repository into this remote location.



17. In the terminal on your development machine, navigate to the local git repository.

```
$ cd ~/<pod name>-bpi-training/
```

18. Configure a new remote repository that points to the newly created repository on GitLab.

```
$:~/pod-name-bpi-training$  git remote add origin http://gitlab-
   bp.local/bpi-netdev-intro/<pod-name>-bpi-training.git
```

A remote URL that starts with http:// indicates that the HTTP protocol is used for data transfer. If the URL starts with **git@**gitlab-bp.local/bpi-netdev-intro/, the transfer is done using the SSH protocol. The SSH protocol allows you to configure an RSA key pair for authentication to the remote repository.

19. Create a new branch named “initial-changes”.

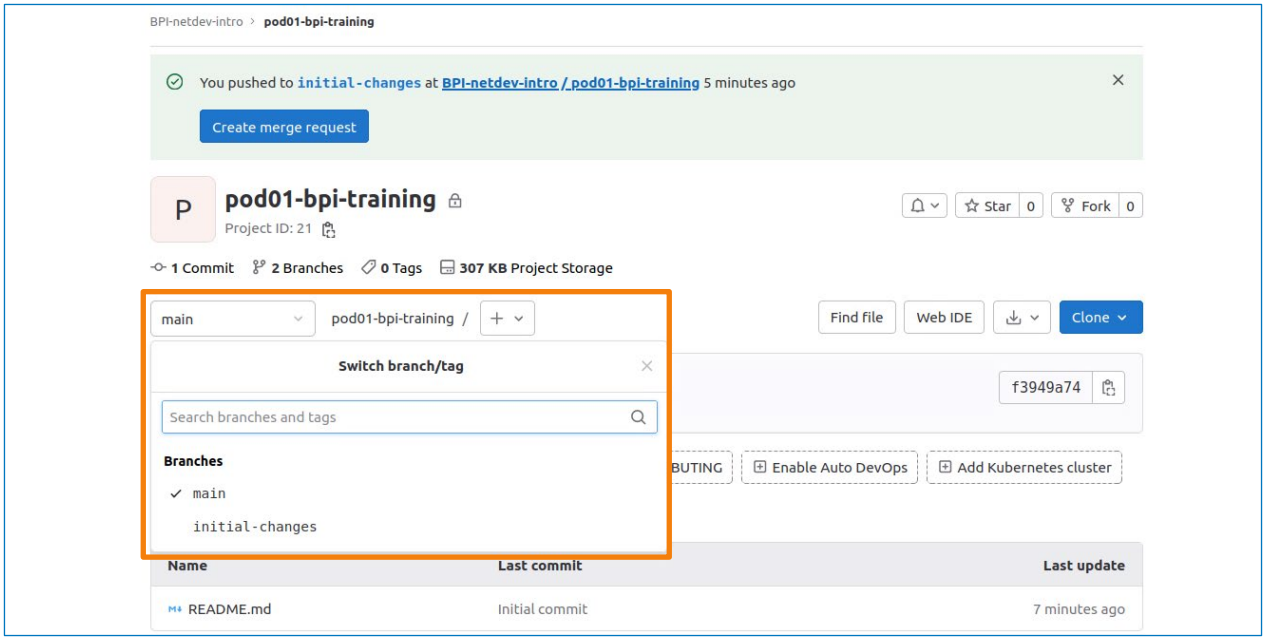
```
$:~/name-bpi-training$  git checkout -b initial-changes
Switched to a new branch ‘initial changes’
$:~/name-bpi-training$  git branch
* initial-changes
  main
```

You cannot push directly to the default "main" branch because the remote repository contains changes (for example, the README.md file) that you do not have locally. You also cannot force a push because the default branch is protected against forced git pushes. You can avoid this by pushing a custom local branch to the remote repository and then merging the changes with the main branch. In a repository shared by multiple developers, it is good practice to develop changes in a separate branch from the main branch and then merge the changes when they are reviewed and tested.

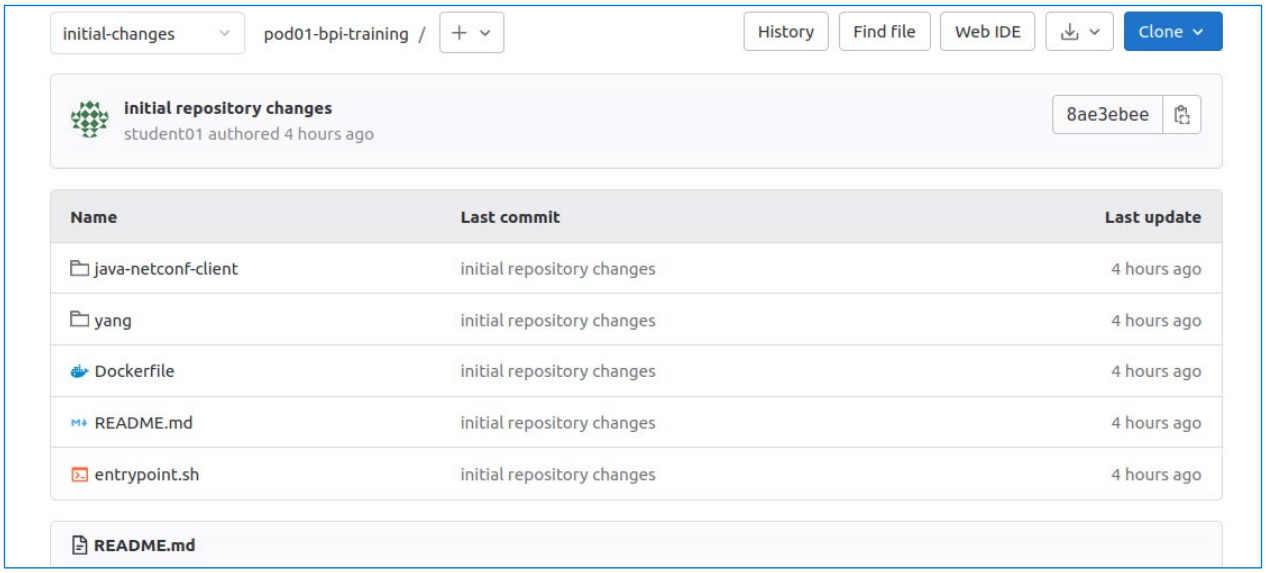
20. Push the local repository content to the remote repository. Enter the GitLab username and password when asked.

```
$:~/name-bpi-training$  git push -uf origin initial-changes
Username for 'https://gitlab-bp.local: <username>
Password for 'https:// <username>@gitlab-bp.local:
Counting objects: 31, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (23/23), done.
Writing objects: 100% (31/31), 11.39 KiB | 2.85 MiB/s, done.
Total 31 (delta 0), reused 0 (delta 0)
remote:
remote: To create a merge request for initial-changes, visit:
remote:   http://10.247.1.26/bpi-netdev-intro/pod01-bpi-training/-
   /merge_requests/new?merge_request%5Bsource_branch%5D=initial-changes
remote:
To http://gitlab-bp.local/bpi-netdev-intro/pod01-bpi-training.git
 * [new branch]      initial-changes -> initial-changes
Branch 'initial-changes' set up to track remote branch 'initial-
   changes' from 'origin'.
```

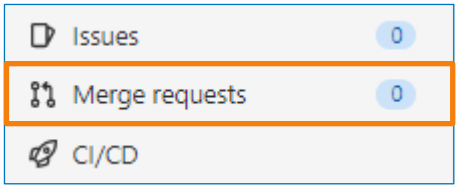
21. Using a browser, navigate to the remote repository in GitLab. You will notice that the branch you pushed is now visible in UI. Click on the branch dropdown menu and select **initial-changes**.



22. Review the state of the repository, the files should match your local repository.

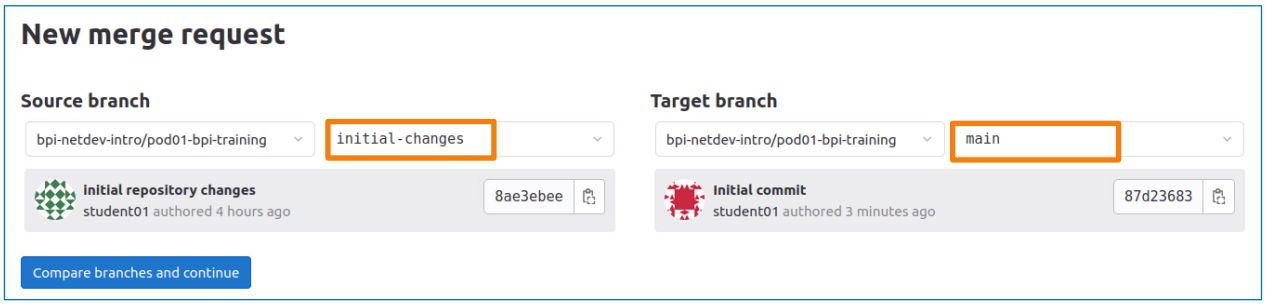


23. In the menu on the left, select **Merge Requests**.



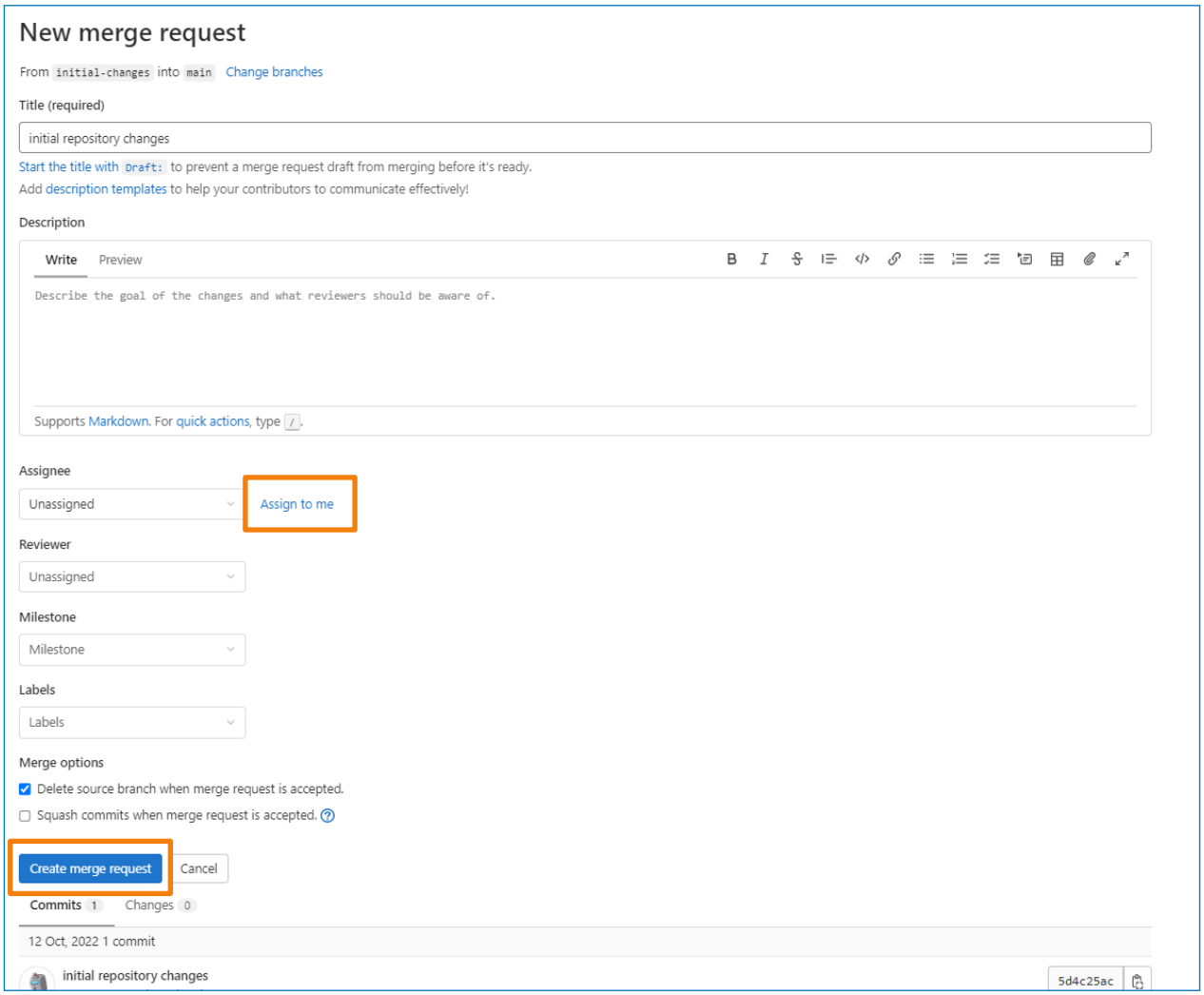
24. Click **New merge request**.

25. As the source branch, select the **initial-changes** branch, and for the target branch, select the **main** branch.



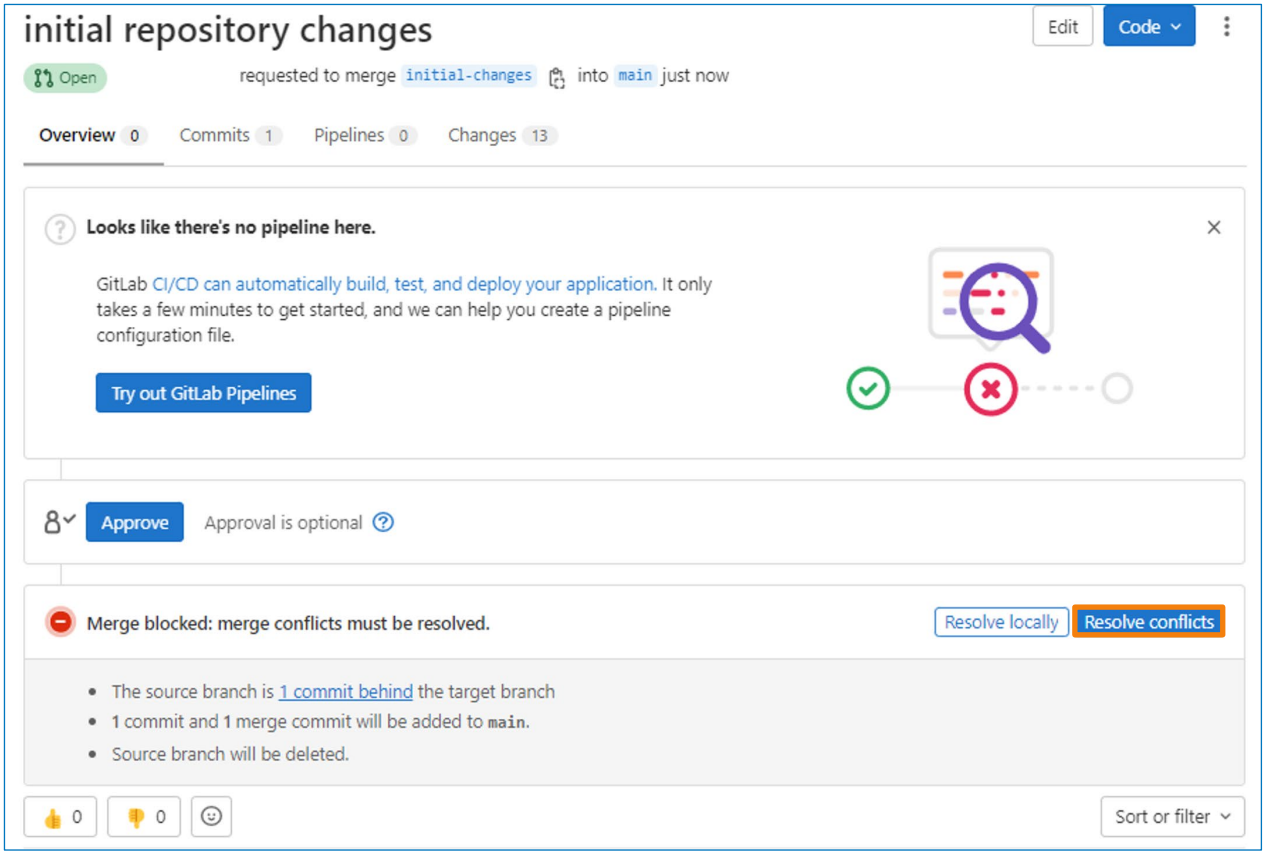
26. Click **Compare branches and continue**.

In the next window, you can describe the changes in your branch and adjust the settings of the merge request.



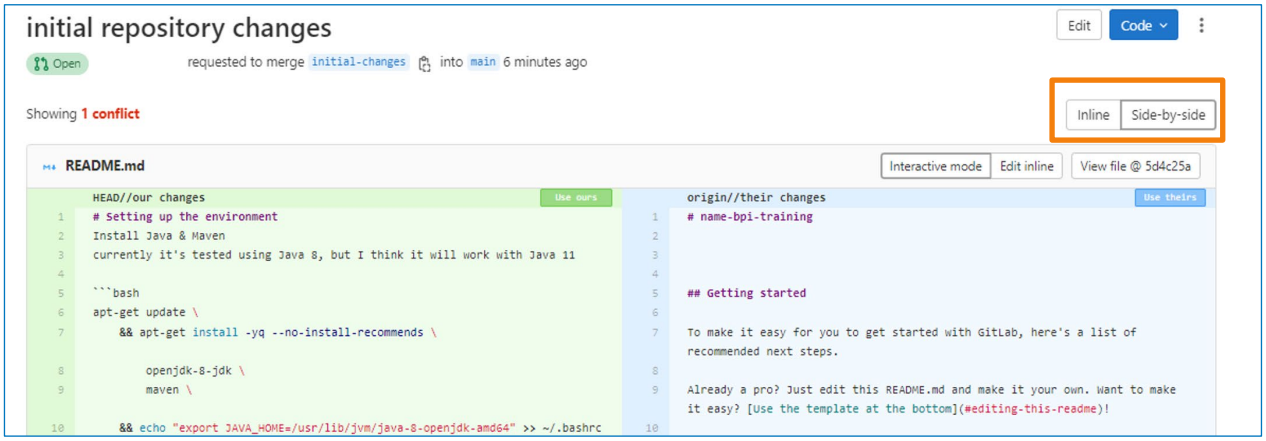
27. Click the **Assign to me** link and then the **Create merge request** button.

    The following window is your merge request page, which will exist until the branch is not merged into the target branch. Reviewers and developers can comment on the changes, and all commits related to the branch associated with the merge request are tracked here, a perfect place to follow the progress of the development of features.



    As you can see, the merge is blocked because of conflicts.

28. Click the **Resolve conflicts** button. Use the **Side-by-side** view.



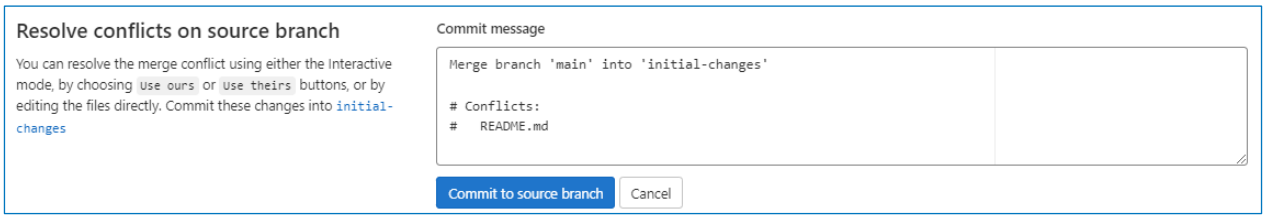
    The current conflict exists because of discrepancies in the README.md file in the two branches, that git cannot solve on its own. Your goal as a developer or reviewer is to decide which changes you want to keep in the target (main) branch. GitLab helps you do this by highlighting the changes

in the source branch (initial-changes) in green and the state of the file in the target branch (master) in blue.

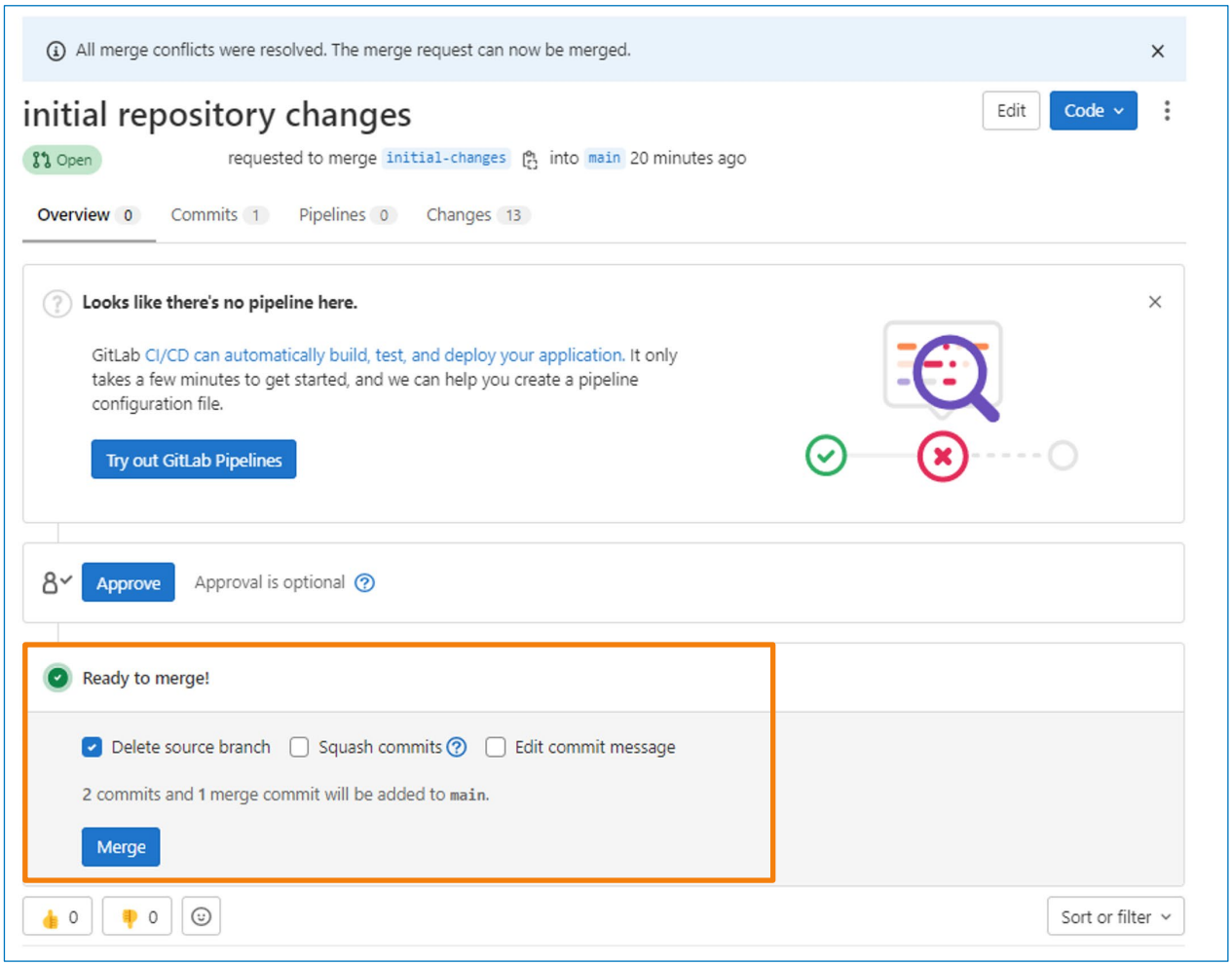
In this case, it is desired to override the default README.md with the version in the initial branch.

29. Click on **Use ours** in the upper-right corner of the section highlighted in green.

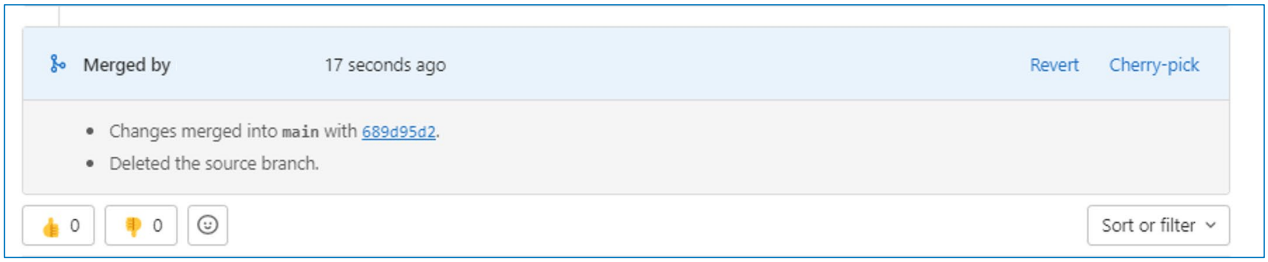
30. Scroll down and click **Commit to source branch**.



31. The merge request window suggests that all merge requests are resolved. Click on the **Merge** button.



By merging the branch, you have closed the merge request and deleted the source branch (initial-changes).



32. Open the terminal on your development machine. If you closed the previously opened window, navigate back to the local git repository.

```
$ cd ~/<your name>-bpi-training/
```

33. Switch to the main branch.

```
$:~/name-bpi-training$  git checkout main
```

```
Branch 'main' set up to track remote branch 'main' from 'origin'.
Switched to a new branch 'main'
```

34. Pull the changes from the remote repository.

```
$:~/name-bpi-training$  git pull origin main
Username for 'https://gitlab.flint.si': <username>
Password for 'https:// <username>@gitlab.flint.si':
remote: Enumerating objects: 7, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 5 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (5/5), done.
From https://gitlab.flint.si/ciena-blue-planet/bpi-academy/name-bpi-
   training
 * [new branch]      main       -> origin/main
```

35. Inspect the commits, and make sure the merge commit exists in your local branch.

```
$:~/name-bpi-training$  git log
commit 689d95d2a3e568d6bfb9abe4df48fd296ae14023 (HEAD -> main,
   origin/main)
Merge: 919e377 3987589
Author: <your name> <your email>
Date:   Mon Feb 27 11:00:24 2023 +0200
    Merge branch 'initial-changes' into 'main'
    initial repository changes
    See merge request bpi-netdev-intro/pod01-bpi-training!1
<… output omitted …>
```

Task 3: Configure VS Code editor for development

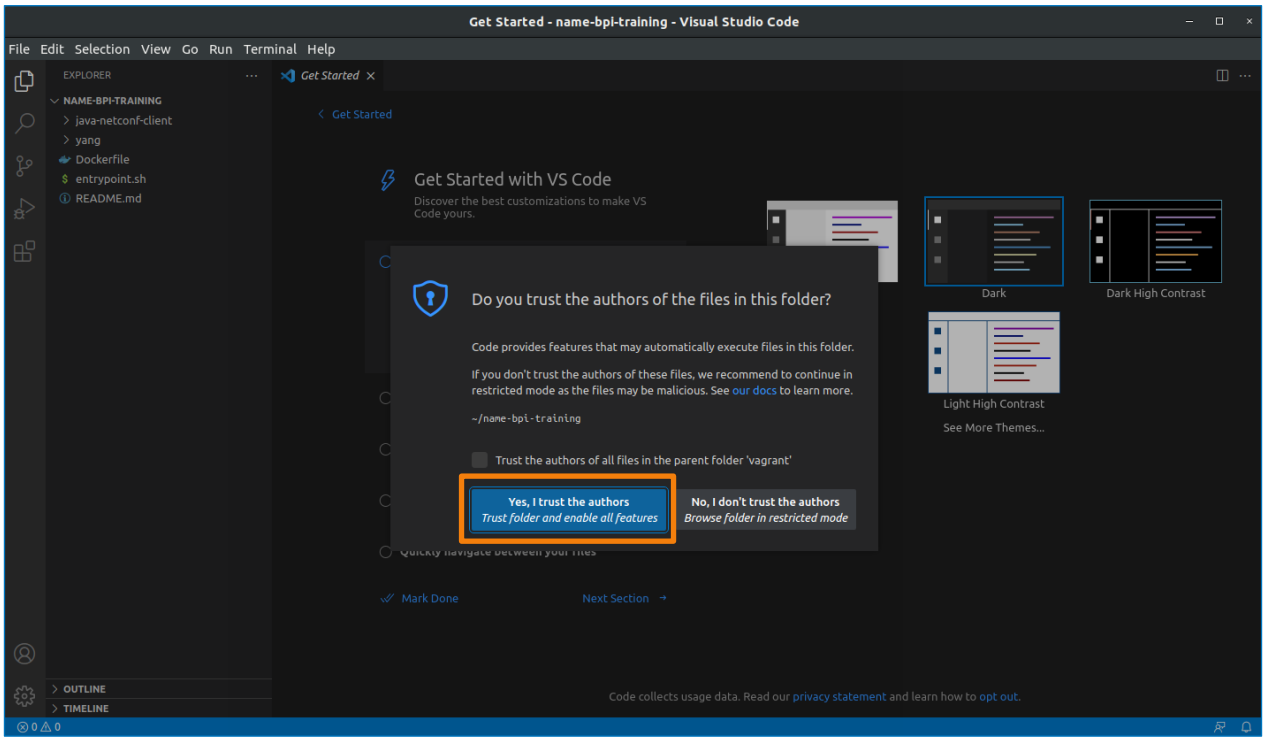
1. Using the terminal, open the repository directory in VS Code.

```
$:~/<pod name>-bpi-training$  code .
```

You can use a dot if you want to open the current directory, or the full path (for example, /home/vagrant/name-bpi-training).

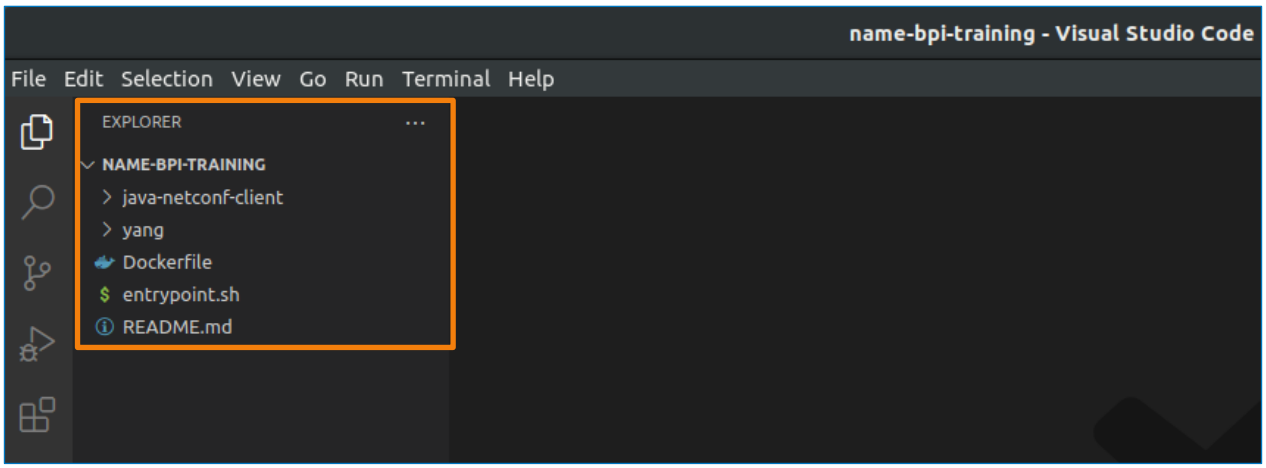
A new VS Code window should open with the repository files.

2. Click the **Yes, I trust the authors** button.



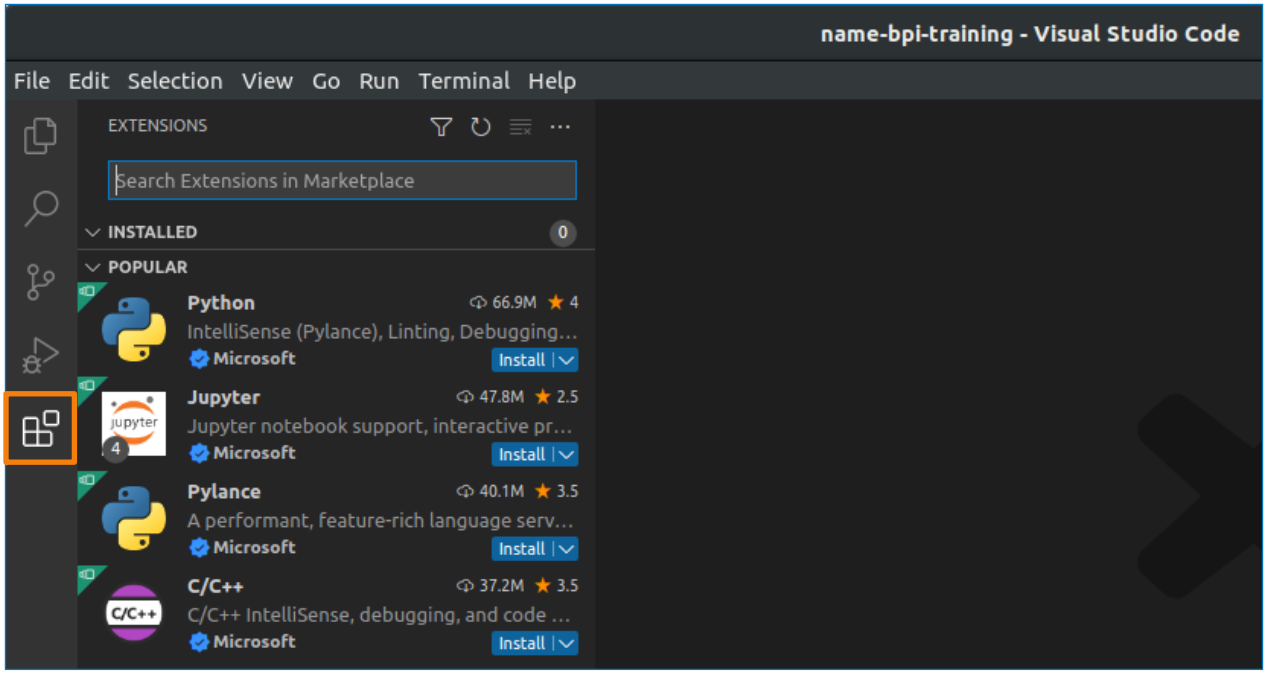
If you want, you can also choose an appearance with different themes.

3. Explore the opened workspace.



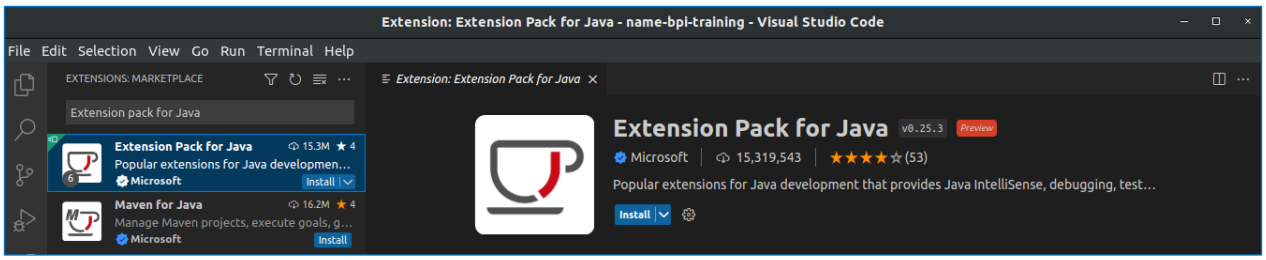
A Visual Studio Code "workspace" is the collection of one or more folders that are opened in a VS Code window (instance). In most cases, you have a single folder open as a workspace. Each workspace can have its own settings.

4. In the left vertical menu, select the **Extensions** tab.

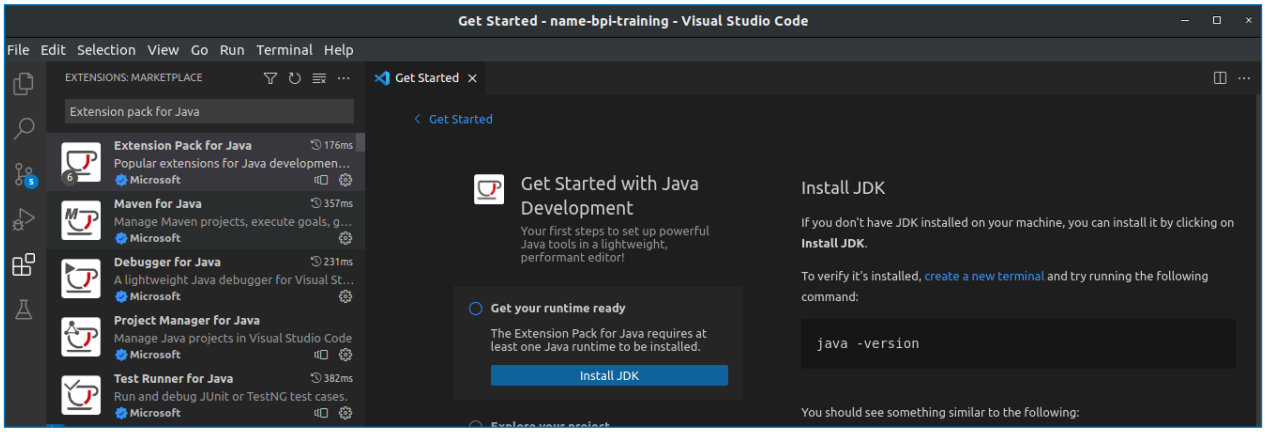


With VS Code extensions you can add languages, debuggers, and tools to your installation to support your development workflow.

5. Search for the **Extension pack for Java** developed by Microsoft.

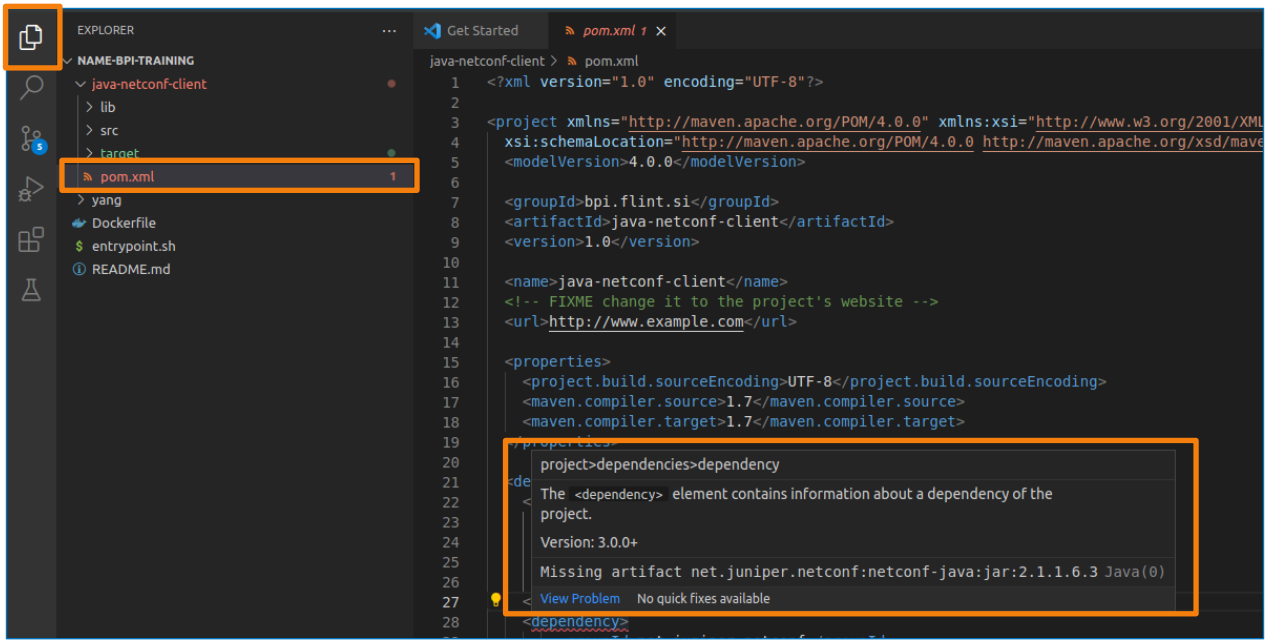


6. Click the **Install** button.



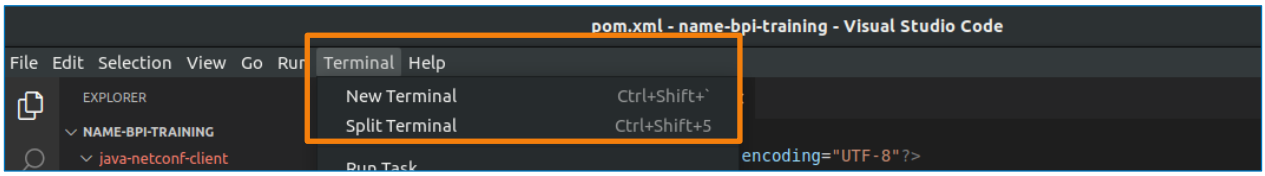
After a successful installation, a guide will suggest the installation of the JDK. You have already done this in the previous task.

7. Navigate to the workspace explorer. Notice that VS Code now suggests that there are some errors in your code.



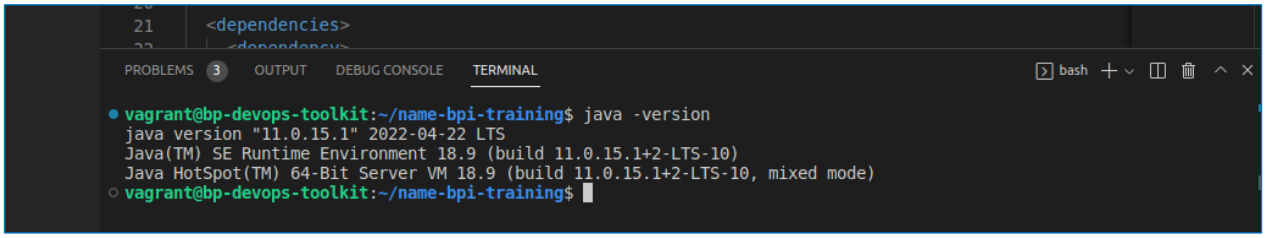
These issues will be solved in the following lab.

8. Open a new terminal window in VS Code. In the upper menu, select **Terminal > New Terminal**.

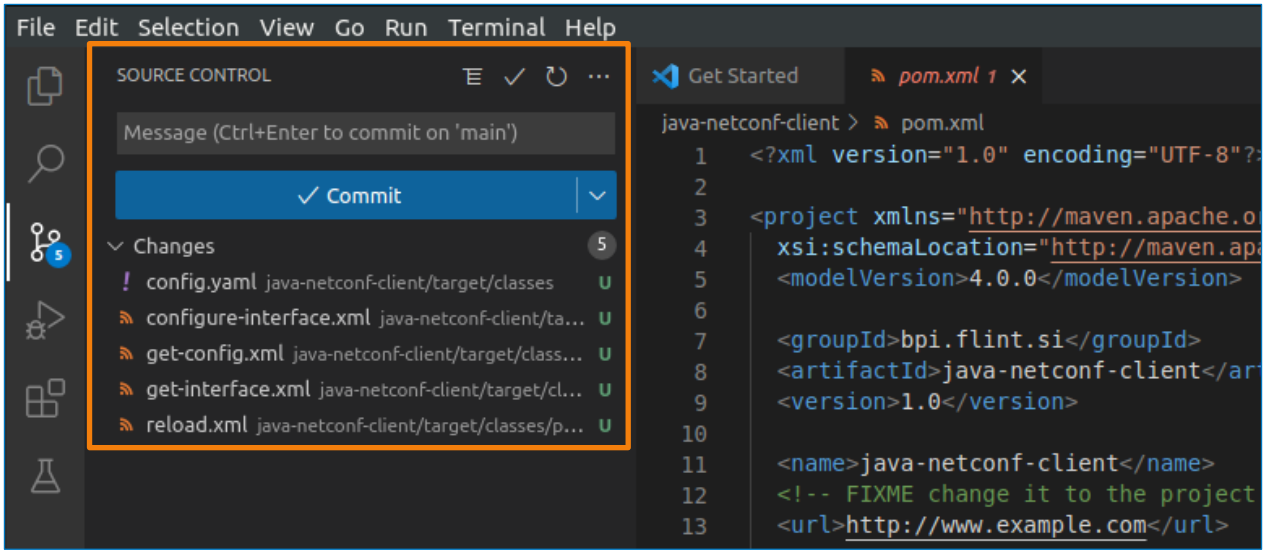


A new terminal window should open in the lower area of the VS code editor.

9. Check if Java is available in the VS Code terminal.

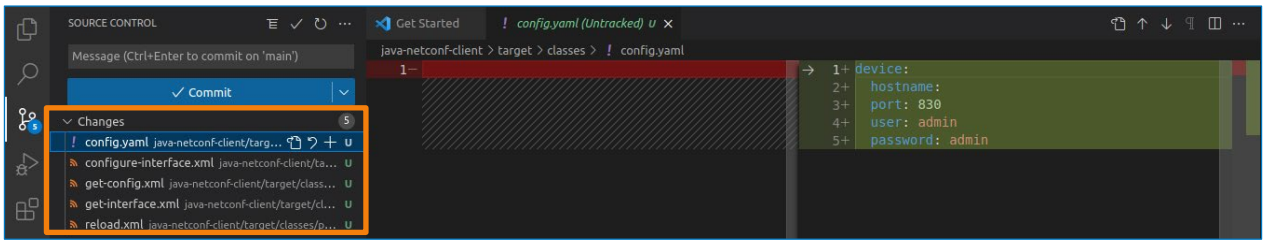


10. You can also use VS Code as a graphical user interface for Git. Click on the **Source Control** tab in the left vertical menu.



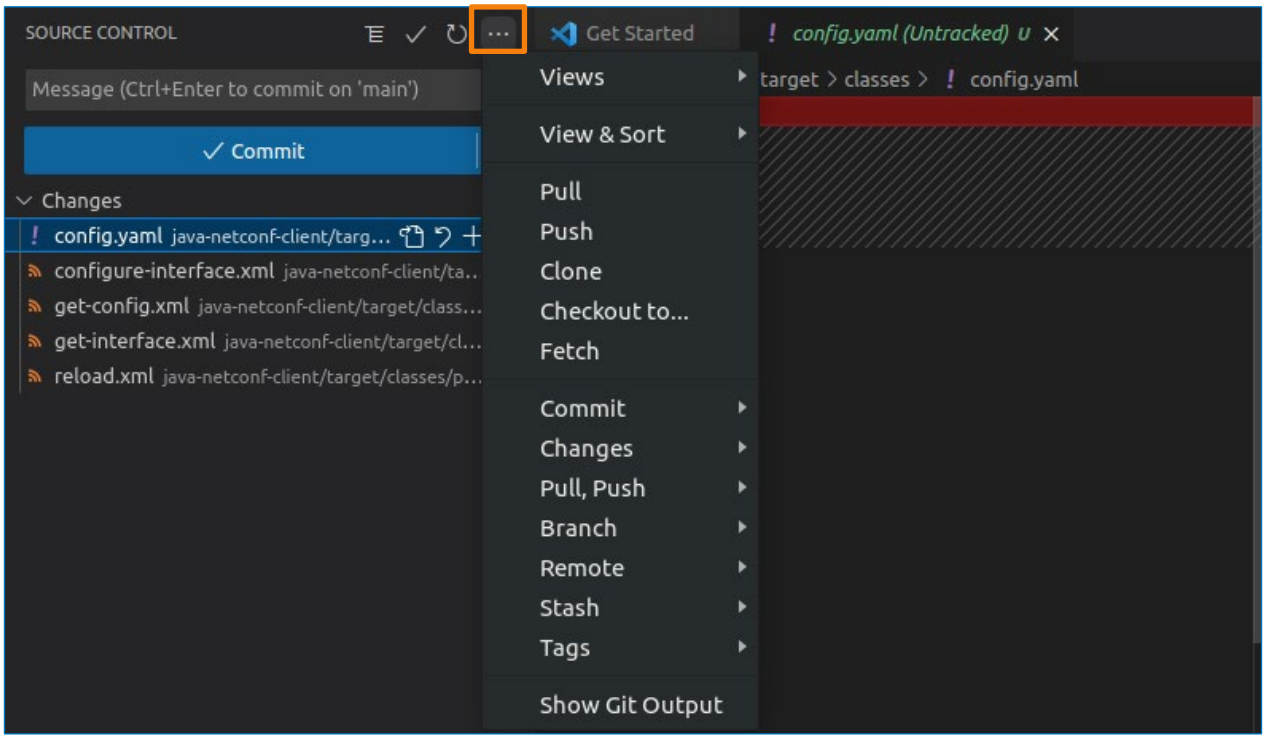
When you installed the Java extension, it automatically created a **target** directory within the repository where the compiled Java files and resources are typically installed. The VS Code Git extension is showing this as changes to the git repository.

11. Click one of the files within the **Changes** folder.



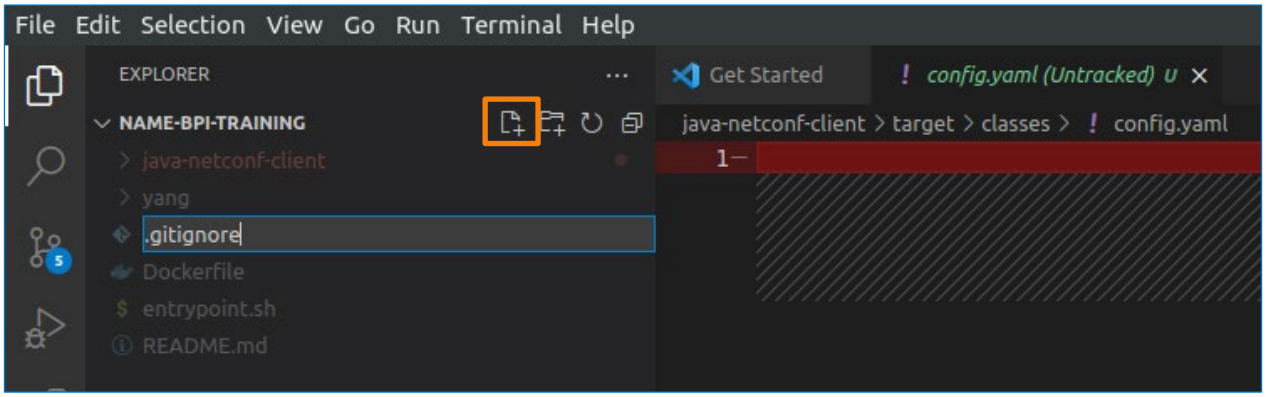
VS Code can show you the difference between the last commit in the branch and the change that is introduced. In this example, the files in the **target** directory did not exist before.

12. Click on the three dots in the source control window.

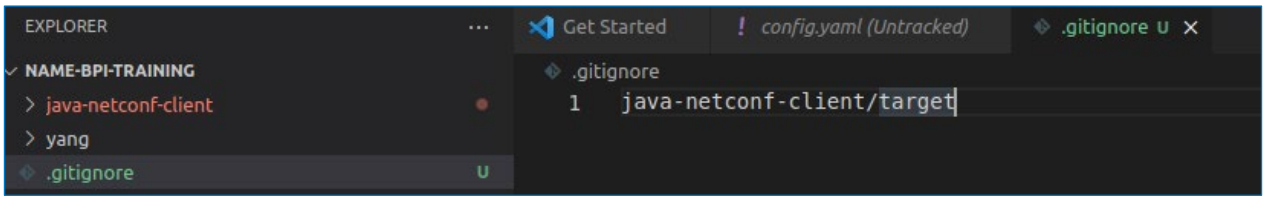


VS Code allows you to manage the Git repository through the graphical user interface with almost all the commands you would normally type in the command line.

13. Navigate back to the workspace, and at the root of the repository create a new file called **.gitignore**. You can right-click on the workspace and click **New File…**. Make sure that you create the file at the root of the workspace directory.

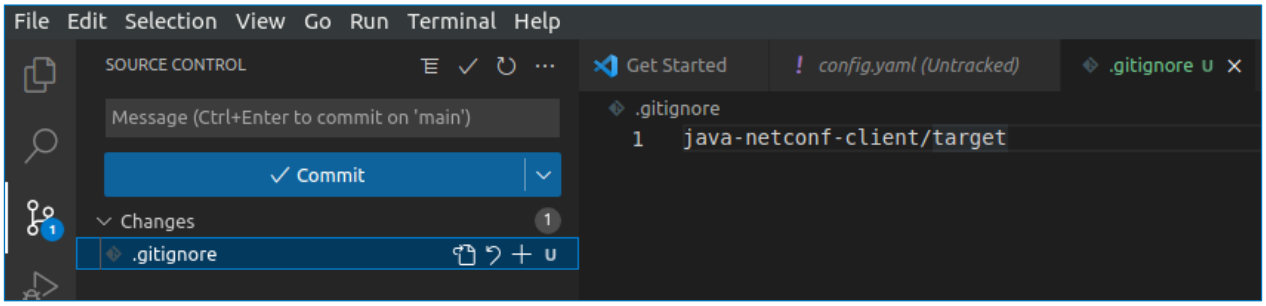


14. Open the .gitignore file by double-clicking it, and then type in “java-netconf-client/target”.

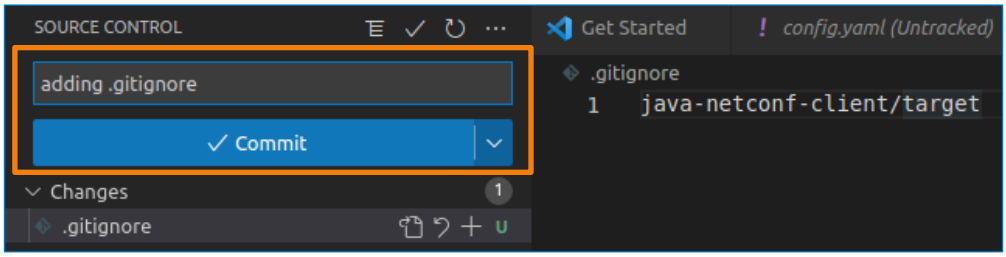


You can use the .gitignore file to specify intentionally untracked files or entire directories that can be ignored by Git. In this case, the target directory contains the compiled Java files that you should not track with Git.

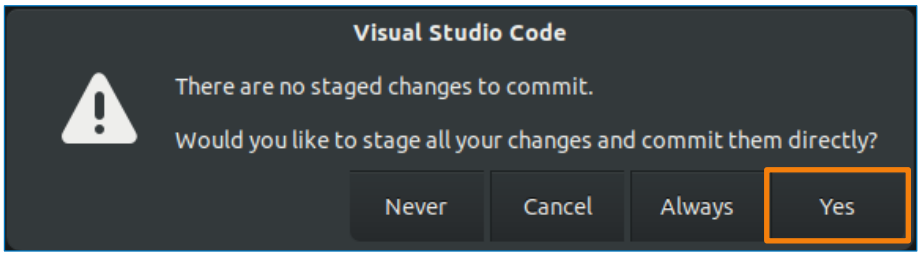
After you save the **.gitignore** file, you will notice on the Source Control tab that the changes do not include anymore the resource files from the *target* directory.



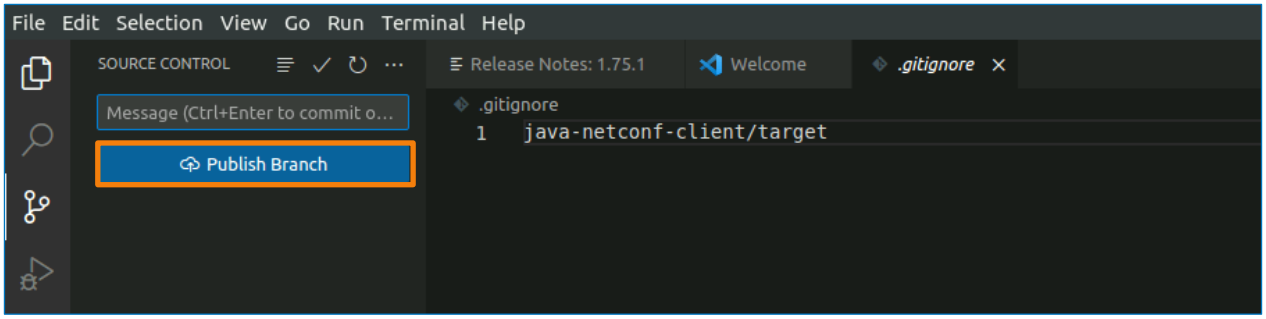
15. In the Source Control tab, type the message **adding .gitignore** and then click the **Commit** button.



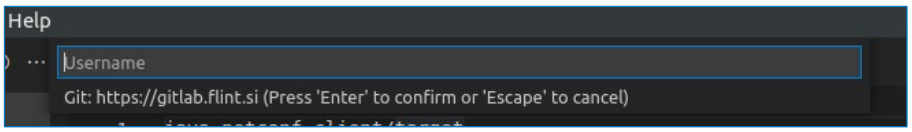
16. In the following prompt, select **Yes** to stage the .gitignore file. This is equivalent to the **git add .gitignore** command.



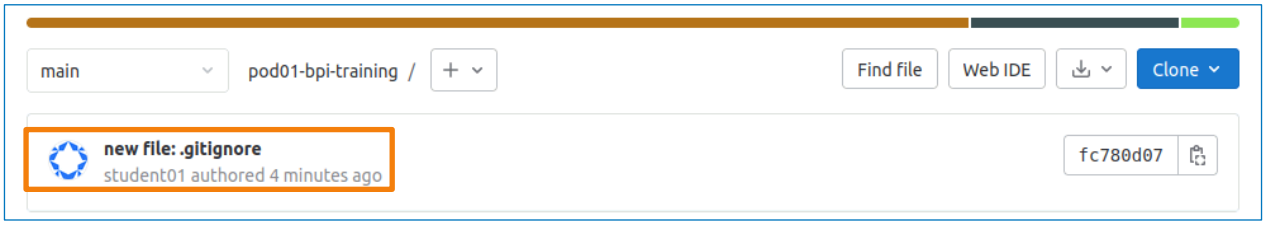
17. Save the changes then accept the commit message.
18. Push the changes to the remote repository (Gitlab) by clicking the **Publish Branch** button.



A pop-up window will appear at the top of the editor asking for your username and password. After you enter the correct credentials, the push to the remote repository will be successful.



19. Navigate to your remote repository using a browser. You should see that the local changes have been successfully added to the remote repository.



End of Lab

Lab 2: Manage Device Configurations with NETCONF/YANG

Objectives

- Creating loopback interface YANG model
- Install NETCONF client dependencies
- Implementing a get-config NETCONF operation in Java
- Implementing an edit-config NETCONF operation in Java

Task 1: Prepare loopback interface YANG module

1. Use VS Code to open the git repository that you created in the previous lab. As the following output suggests, you can use the terminal to open the required directory.

NOTE: Make sure that you type in your pod name instead of “<pod-number>”, for example, pod04-bpi-training/.

```
vagrant@BPF002-PODXX-Jumphost:~$ code <pod-number>-bpi-training/
```

2. In the Explorer window on the left side of the VS Code interface, navigate to the **yang** directory and open the **dev-sim.yang** file.

   You should see the following content in the module implementation.

```
module dev-sim {
  yang-version 1.1;
  namespace "http://flint.si/ns/yang/dev-sim";
  prefix dsim;
  import ietf-inet-types {
    prefix inet;
  }
  organization
    "Flint SI";

  description
    "NED YANG module for interfaces";

  revision 2022-11-02 {
    description
      "YANG module for interfaces";
  }

  //implement a type definition
```

```
  container interfaces {
      // implement a list of loopbacks
  }

  grouping interface-common {
    // implement interface common parameters
  }
}
```

The model already contains the basic metadata such as the namespace, prefix, organization, and description. It also imports an **ietf-inet-types** yang module that will help you with the parameter types.

First, you will focus on the **interface** container.

3. Within the curly braces of the **interfaces** container, implement a list of loopbacks with the name parameter as the key.

```
<… output omitted …>
  container interfaces {
      list loopback {
        description
        "Loopback interface";
        key "name";
        leaf name {
          type uint32 {
            range "0..2147483647";
          }
        }
      }
  }
<… output omitted …>
```

Make sure that all YANG elements that begin with a curly bracket are also closed with one. You can also install a YANG syntax highlighter on the VS Code extensions tab, which can help you with module development.

A loopback interface usually allows more than just the configuration of its name. To allow the configuration of other interface parameters on a NETCONF-enabled device, you should first model them. Different interface types (Ethernet, Loopback, ...) may have some parameters in common. To avoid duplication in your model, create a grouping of common parameters that can be used for all interface types.

4. Within the curly braces of the **interfaces-common** grouping, implement the IP address, description, and shutdown parameters.

```
<… output omitted …>
grouping interface-common {
    container ip {
      description
      "Set the IP address";
      leaf address {
        type inet:ipv4-address;
      }
    }
    leaf description {
      description
        "Interface specific description";
      type string {
        pattern ".*";
        length "0..200";
      }
    }
    leaf shutdown {
      default "no";
      type shutdown-option;
    }
  }
<… output omitted …>
```

YANG models define how clients (CLI, NETCONF, RESTCONF) interact with NETCONF devices or servers. How to model the parameters is a design choice that needs to consider the simplicity, usability, and parameter types that the device will be able to support.

For the *address* parameter, you have used a derived type defined in the imported *ietf-inet-types* module (see the use of *inet:* prefix). Defining the correct type is a first security measure because it prevents clients from entering invalid data. It is better to intercept them before the device tries to process them. In the **description** leaf, you used a built-in type string but restricted the input with a regular expression pattern that prevents line terminations. In the *shutdown* leaf, you have used a derived type *shutdown-option*, but note that it does not use a prefix, indicating that this module should implement this derived type definition.

5. Implement the **shutdown-option** typedef used by the **shutdown** leaf.

```
<… output omitted …>
  description
```

```
   "NED YANG module for interfaces";
  //implement a type definition
  typedef shutdown-option {
    type enumeration {
      enum "yes";
      enum "no";
    }
  }
<… output omitted …>
```

6. Add the previously implemented common parameters grouping to the list of loopbacks.

```
<… output omitted …>
  container interfaces {
      list loopback {
        description
        "Loopback interface";
        key "name";
        leaf name {
          type uint32 {
            range "0..2147483647";
          }
        }
        uses interface-common;
      }
  }
<… output omitted …>
```

The *uses* keyword makes the parameters implemented in the grouping available at the selected location.

7. Check the full implementation of the YANG module and make sure it matches the following module before proceeding to the next task.

```
module dev-sim {
  yang-version 1.1;
  namespace "http://flint.si/ns/yang/dev-sim";
  prefix dsim;
  import ietf-inet-types {
    prefix inet;
  }
```

```
organization
  "Flint SI";

description
  "YANG module for interfaces";

revision 2022-10-26 {
  description
    "YANG module for interfaces";
}

typedef shutdown-option {
  type enumeration {
    enum "yes";
    enum "no";
  }
}
container interfaces {
  list loopback {
    description
      "Loopback interface";
    key "name";
    leaf name {
      type uint32 {
        range "0..2147483647";
      }
    }
    uses interface-common;
  }
}

grouping interface-common {
  container ip {
    description
      "Set the IP address";
```

```
      leaf address {
        type inet:ipv4-address;
      }
    }
    leaf description {
      description
        "Interface specific description";
      type string {
        pattern ".*";
        length "0..200";
      }
    }
    leaf shutdown {
      default "no";
      type shutdown-option;
    }
  }
}
```

8. Use the **pyang** tool to validate your model.

```
~/$ cd <pod-name>-bpi-training/
~/<pod name>-bpi-training$ pyang --path yang/ yang/dev-sim.yang
~/<pod name>-bpi-training$
```

If there is no output, the model is valid.

Task 2: Start the NETCONF server

1. Copy the **dev-sim.yang** file to the NETCONF device directory.

```
$:~/<pod name>-bpi-training$ cp yang/dev-sim.yang ~/dev-sim/dev-1/yang/dev-sim/
```

2. Run the NETCONF device.

```
$:~/<pod name>-bpi-training$ cd ~/dev-sim

$:~/dev-sim$ docker-compose up -d

Building with native build. Learn about native build in Compose here: https://docs.docker.com/go/compose-native-build/

Creating network "dev-sim_dev-sim_net" with driver "bridge"

Creating dev-1 ... done
```

   The lab environment uses *Clixon*, a YANG-based configuration manager with CLI, NETCONF, and RESTCONF interfaces.

3. Inspect the device capabilities using the **netconf-console2** NETCONF client.

```
$ netconf-console2 --host $(docker inspect dev-1 --format '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}') --port 830 --hello

<?xml version='1.0' encoding='UTF-8'?>

<nc:hello xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">

  <nc:capabilities>

    <nc:capability>urn:ietf:params:netconf:base:1.1</nc:capability>

    <nc:capability>urn:ietf:params:netconf:base:1.0</nc:capability>

<nc:capability>urn:ietf:params:netconf:capability:candidate:1.0</nc:capability>
<nc:capability>urn:ietf:params:netconf:capability:validate:1.1</nc:capability>

    <… output omitted …>

  </nc:capabilities>

</nc:hello>
```

   Note that among the capabilities returned by the NETCONF device, there is no capability that provides direct write access to the running datastore (*writable-running*). This will be important later when you will use your NETCONF client to configure the device.

4. Inspect the initial device configuration.

```
$ netconf-console2 --host $(docker inspect dev-1 --format '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}') --port 830 --get-config

<?xml version='1.0' encoding='UTF-8'?>

<data xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
```

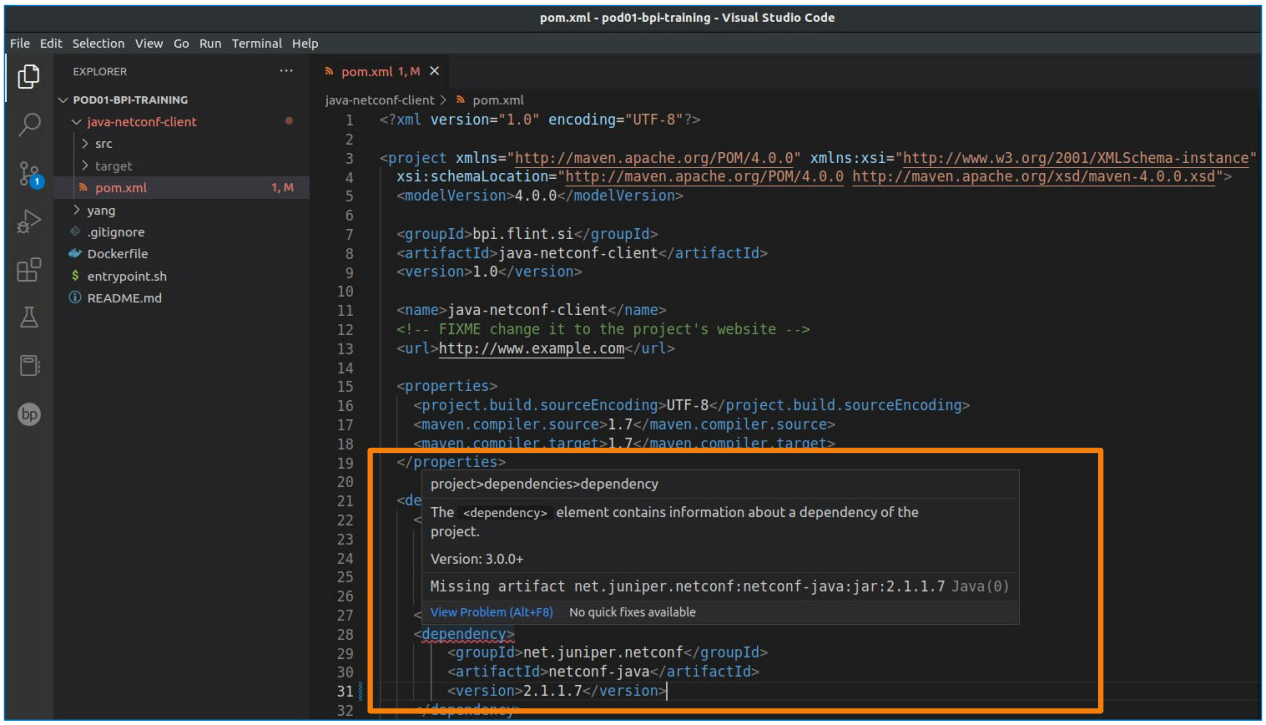
```
<interfaces xmlns="http://flint.si/ns/yang/dev-sim">
  <loopback>
    <name>0</name>
    <ip>
      <address>127.0.0.1</address>
    </ip>
    <... output omitted ...>
```

The NETCONF device was started with the initial configuration of the Loopback 0 interface.

Task 3: Install NETCONF Client Dependencies and Implement Device Connection

The NETCONF client that you are going to use is based on the *Juniper netconf-java library*.

1. In VS Code, open the **pom.xml** file located in the **java-netconf-client** directory.



Note that there are some problems in the file. The .net.juniper.netconf library, which is specified as a dependency, cannot be found. This is because the library is not present in the official Maven repository (*https://mvnrepository.com/*). The library should be downloaded and built manually.

2. In a terminal window, open the repository, navigate to the **java-netconf-client** directory, and clone the NETCONF library into the **lib** folder.

```
~/ cd ~/<pod-number>-bpi-training/java-netconf-client

~/name-bpi-training/java-netconf-client$ git clone http://gitlab-
bp.local/bpi-netdev-intro/netconf-java.git lib/netconf-client

Cloning into 'lib/netconf-client'...

<… output omitted …>
```

3. Install the NETCONF client library using Maven.

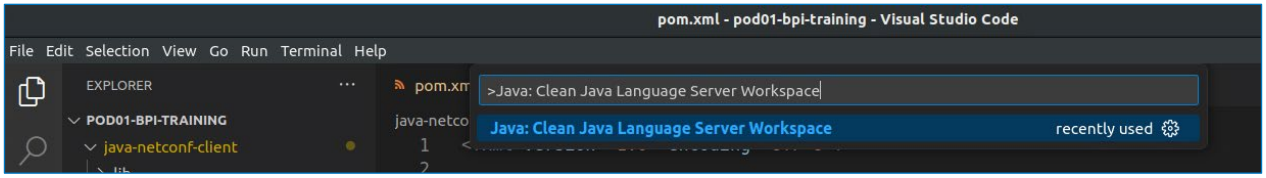
```
$:~/name-bpi-training/java-netconf-client$  mvn install -f lib/netconf-
client/pom.xml

[INFO] Scanning for projects...

<… output omitted …>

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

[INFO] BUILD SUCCESS

[INFO] ------------------------------------------------------------------------
```

```
<… output omitted …>
```

4. Install all dependencies in the **java-netconf-client** project.

```
$:~/name-bpi-training/java-netconf-client$  mvn install
[INFO] Scanning for projects...
<… output omitted …>
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
<… output omitted …>
```

5. If the **pom.xml** file still indicates a missing dependency in VS Code, open the command palette with **Ctrl + shift + p** key combination, and type in **Java: Clean Java Language Server Workspace**. Confirm the clean-up when the pop-up window appears by clicking the "**Reload and delete button**".



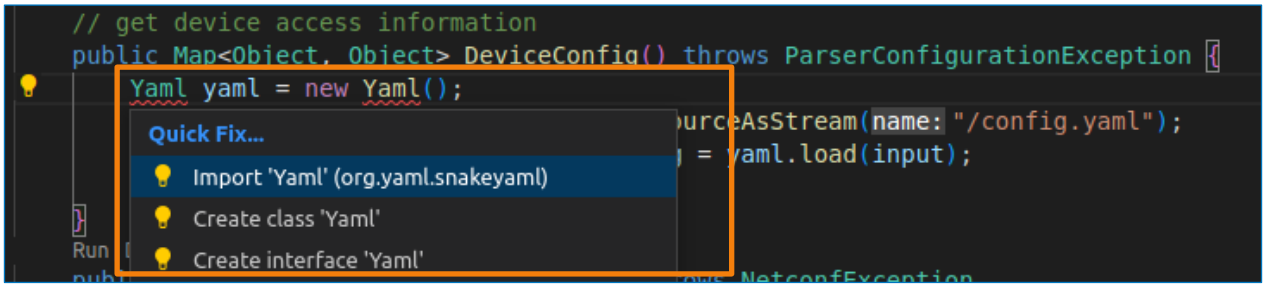
6. Start implementing the Java NETCONF client application. Implement the method for reading device access configuration. Modify the **App** class in the **src/main/java/*/App.java** path.

```
<… output omitted …>
public class App {
    <… output omitted …>
  // get device access information
  public Map<Object, Object> DeviceConfig() throws
ParserConfigurationException {
     Yaml yaml = new Yaml();
     InputStream input = getClass().getResourceAsStream("/config.yaml");
     Map<Object, Map<Object,Object>> config = yaml.load(input);
     return config.get("device");
  }

  public static void main(String args[]) throws NetconfException
<… output omitted …>
```

The configuration file is stored in YAML format, see **src/main/resources/config.yaml.** A Java YAML library (snakeyaml) is used to load the file into a **Map** object with a key value pair.

Not all libraries used in the code are imported. VS Code highlights portions of code that use objects from libraries that are not imported.

7. In VS Code, hover over the underlined **Yaml** variable declaration, click the offer "Quick fix" and select the option to import the snakeyaml library. You can also move the cursor over the line and use the key combination **Ctrl** + **.** to open the dropdown menu with the suggested corrections.



Repeat the process for all missing imports.

8. In the **main** method, create a new NETCONF Device object using the device configuration and the NETCONF library.

```
    <… output omitted …>
    public static void main(String args[]) throws NetconfException,
                ParserConfigurationException, SAXException, IOException {
        // read device access config
        Map<Object, Object> config = new App().DeviceConfig();
        // create new Device object
        Device device = DeviceBuilder.Create(config);
    <… output omitted …>
```

Again, there will be missing imports. Make sure you import the libraries before proceeding.

9. Save the **App.java** file.
10. Open the **DeviceBuilder.java** file and see how a new Device object is created.

    As input, the **Create** method receives the configuration for a device, reads in all the required parameters, and attempts to connect to the device. The device object returned by the method supports various NETCONF operations that you will use in the following steps.

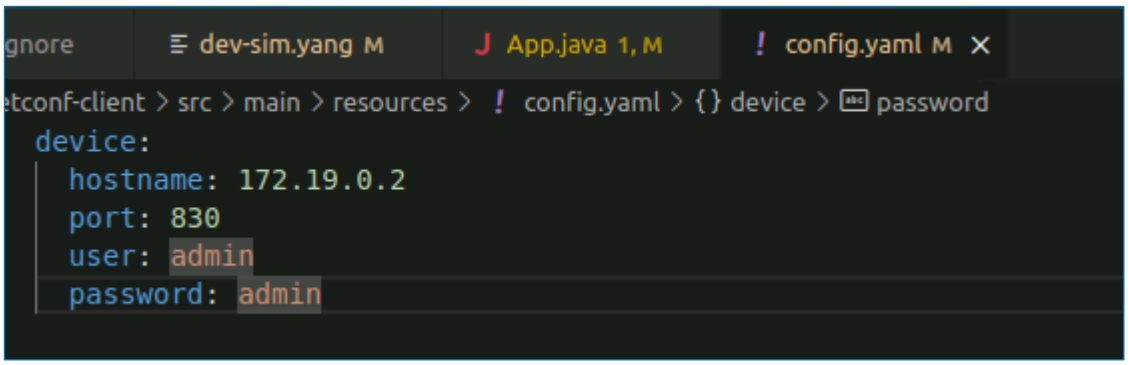
11. Open the **config.yaml** file, located in the **src/main/resources** directory.

```
device:
  hostname:
  port: 830
  user: admin
  password: admin
```

12. Modify the **config.yaml** access information by setting the hostname parameter to the IP address of the running NETCONF device container. Use the following command to get the address.

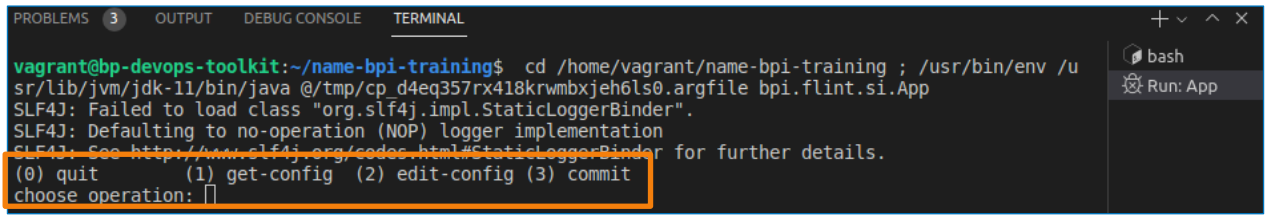
NOTE: Do not forget to save the file after adding the IP.

```
$ docker inspect dev-1 --format
'{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}'
172.19.0.2
```



13. Open the **App.java** file and run the application by clicking the **run** button above the **main** method.

In the terminal window that opens in VS Code, you can track the progress of the running application. When the client has successfully connected to the device, you should be prompted to select a NETCONF operation, as shown in the next figure.



14. Stop the program execution by initiating Ctrl + c key combination. In the next tasks, you will implement the NETCONF operations.

Task 4: Implement NETCONF get-config operation

To retrieve data from a NETCONF device, you can use a get-config operation. You can also provide filters that specify which data in the data tree you are interested in. In this task, you will implement the get-config operation with the XML payload and Java methods.

1. Open the **NetconfCommands.java** file.

```
<… output omitted …>
public class NetconfCommands {
    public Document parsePayload(String payload) {
        <… output omitted …>
    }
    public String DocumentToString(Document doc) {
        <… output omitted …>
    }
}
```

In the **NetconfCommands** class, you will implement the required NETCONF operations. The **parsePayload** and **DocumentToString** methods are helper methods for handling the XML payloads.

2. Open the **get-config.xml** file located in the **src/main/resources/payloads** path.

```
<get-config>
    <source>
    <!--append source datastore-->
    </source>
    <with-defaults xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-with-defaults">report-all</with-defaults>
</get-config>
```

In your Java implementation, you will use such XML payloads as templates for creating NETCONF messages. A get-config message begins with the XML “<get-config>” tag, followed by the “<source>” tag, to which a datastore (candidate or in use) is programmatically attached. The “<with-defaults>” tag is required when displaying the values of nodes that have not been set by a client, but whose default values are defined in the YANG model. This payload returns the entire data store because there are no filters defined.

3. Open the **get-interface.xml** file located in the src/main/resources/payloads path.

```
<get-config>
  <… output omitted …>
  <filter>
     <interfaces xmlns="http://flint.si/ns/yang/dev-sim">
        <loopback>
            <name></name>
```

```
          </loopback>
      </interfaces>
  </filter>
</get-config>
```

This payload is used when you want to limit the output to a single interface. The "<filter>" tag is used in a NETCONF message to indicate which parts of the datastore to output. In the case of this payload, the filter will be a specific loopback interface where the "<name>" tag is added programmatically. If you take a closer look at the filter, you will notice that the XML tags match the hierarchical structure of the YANG model you developed in the first task.

4. Open the **NetconfCommands.java** file and implement the first **getConfig** method as seen in the following code snippet.

```
<… output omitted …>
public class NetconfCommands {
        public void getConfig(Device device, String datastore, String
payload) throws ParserConfigurationException, SAXException, IOException
{
        Document document = parsePayload(payload);
        setDatastore(datastore, document, "source");
        XML message = device.executeRPC(DocumentToString(document));
        System.out.println(message);
    }
```

The method takes three arguments as input: the Device object containing the connection to the device, the datastore string indicating whether the client wants to retrieve candidate or running data, and the path to the XML payload used in the request.

The **setDatastore** method is required for configuring the source XML tag in the payload with the selected datastore (running or candidate).

When the XML message is prepared, it is sent to the device as a NETCONF message using the **executeRPC** method maintained in the NETCONF client library.

5. Implement the **setDatastore** method in the **NetconfCommands** class.

```
<… output omitted …>
    private void setDatastore(String datastore, Document document,
String type) {
        Node datastoreNode = null;
        Element typeElement = null;

        if (type == "target") {
            datastoreNode =
document.getElementsByTagName("target").item(0);
        } else if (type == "source"){
```

```
            datastoreNode =
document.getElementsByTagName("source").item(0);
        }
        if (datastore.equals("candidate")) {
            typeElement = document.createElement("candidate");
        } else if (datastore.equals("operational") &&
type.equals("source")) {
            typeElement = document.createElement("operational");
        } else {
            typeElement = document.createElement("running");
        }
        if (datastoreNode != null) {
            datastoreNode.appendChild(typeElement);
        }
    }
```

The **setDatastore** method takes the datastore (running or candidate), Document (XML payload), and the type (source or target) as input. The type differs when working with the get-config and edit-config operations. A datastore node is attached to the XML source or target node, and the payload is prepared for further modifications.

6. Implement the second **getConfig** method in the **NetconfCommands** class, which is used to query specific loopback interfaces instead of returning the entire configuration.

```
  <… output omitted …>
public void getConfig(Device device, String datastore, String payload,
String interface_id) throws ParserConfigurationException, SAXException,
IOException {
        Document document = parsePayload(payload);
        setDatastore(datastore, document, "source");
        Node interfaceNode =
document.getElementsByTagName("name").item(0);
        interfaceNode.setTextContent(interface_id);
        XML message = device.executeRPC(DocumentToString(document));
        System.out.println(message);
    }
<… output omitted …>
```

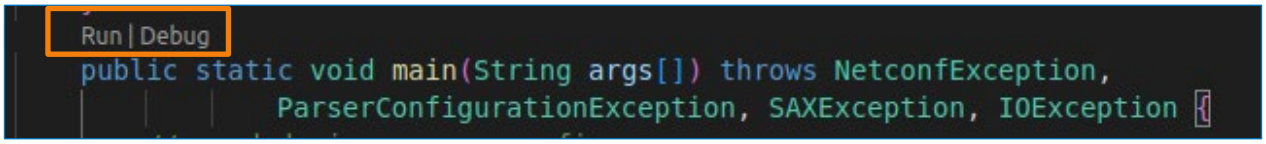
In this example, you used a method overloading technique because you used a method name that already exists, but this method has an additional parameter to filter the output to the desired interface.

7. Save the **NetconfCommands.java** file.
8. Open the **App.java** file, and in the main method, implement the user query prompt for the get-config operation.

```
        <… output omitted …>
         Device device = DeviceBuilder.Create(config);


         NetconfCommands netconf = new NetconfCommands();


         Scanner input = new Scanner(System.in);


         int command = -1;
         while(command != 0) {
             System.out.print("(0) quit\t(1) get-config\t(2) edit-
config\t(3) commit\nchoose operation: ");
             command = input.nextInt();
             if(command == 1) {
                 input.nextLine();
                 System.out.print("datastore: running/candidate [running]:
");
                 String datastore = input.nextLine();
                 System.out.print("loopback id: ");
                 String interface_id = input.nextLine();
                 if (interface_id != null && !interface_id.isEmpty()) {
                     netconf.getConfig(device, datastore,
getInterfaceConfigPayload, interface_id);
                 } else {
                     netconf.getConfig(device, datastore,
getConfigPayload);
                 }
             }
             else if(command == 2) {
                 <… output omitted …>
```

The implementation of the get-config interface allows users to either skip the interface ID or submit it as input. In the first case, the XML NETCONF message template is passed to the **getConfig** method without the filter; in the second case, the message is passed with the filter, along with the user-supplied interface ID.

9. Save the file **App.java** file and run the application from VS Code.



10. In the VS Code integrated terminal, type **1** to select the get-config operation, and press **Enter**.

```
PROBLEMS 2   OUTPUT   DEBUG CONSOLE   TERMINAL

vagrant@bp-devops-toolkit:~/name-bpi-training$  /usr/bin/env /usr/lib/jvm/jdk-11/bin/java @/tmp/cp_d4eq357rx418krwmbxjeh6ls
0.argfile bpi.flint.si.App
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
(0) quit        (1) get-config  (2) edit-config (3) commit
choose operation: 1
```

11. In the next step, you can press **Enter** to select the **running** datastore by default.

```
PROBLEMS 3   OUTPUT   DEBUG CONSOLE   TERMINAL

vagrant@bp-devops-toolkit:~/name-bpi-training$  cd /home/vagrant/name-bpi-training ; /usr/bin/env /usr/lib/jvm/jdk-11/
q357rx418krwmbxjeh6ls0.argfile bpi.flint.si.App
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
(0) quit        (1) get-config  (2) edit-config (3) commit
choose operation: 1
datastore: running/candidate [running]:
```

12. Skip entering ID by pressing **Enter**. This action queries the entire running datastore without filters.

```
PROBLEMS 3   OUTPUT   DEBUG CONSOLE   TERMINAL

vagrant@bp-devops-toolkit:~/name-bpi-training$  cd /home/vagrant/name-bpi-training ; /usr/bin/env /usr/lib/jvm/jdk-11
q357rx418krwmbxjeh6ls0.argfile bpi.flint.si.App
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
(0) quit        (1) get-config  (2) edit-config (3) commit
choose operation: 1
datastore: running/candidate [running]:
loopback id:
```

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-
id="2">
    <data>
        <interfaces xmlns="http://flint.si/ns/yang/dev-sim">
            <loopback>
                <name>0</name>
                <ip>
                    <address>127.0.0.1</address>
         <... output omitted ...>
    </data>
</rpc-reply>
```

13. Repeat the get-config operation but enter *10* as a loopback ID.

```
(0) quit        (1) get-config  (2) edit-config (3) commit
choose operation: 1
datastore: running/candidate [running]:
loopback id: 10
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="5">
    <data>
        <interfaces xmlns="http://flint.si/ns/yang/dev-sim"/>
    </data>
</rpc-reply>
```

An empty rpc-reply message is returned because the loopback with this ID is not configured. In the next task, you will implement the edit-config operation to configure the loopback interface.

Task 5: Implement NETCONF edit-config operation

1. Use the VS Code editor to open the **NetconfCommands.java** file and implement the **editConfig** method as part of the **NetconfCommands** class.

```
public class NetconfCommands {

<... output omitted ...>

    public void editConfig(Device device, String datastore, String
payload, String operation, String interface_id, String address, String
description, String shut) throws ParserConfigurationException,
SAXException, IOException {

        Document document = parsePayload(payload);

        setDatastore(datastore, document, "target");

        Node interfaceNode =
document.getElementsByTagName("name").item(0);

        Node addressNode =
document.getElementsByTagName("address").item(0);

        Node descriptionNode =
document.getElementsByTagName("description").item(0);

        Node shutdownNode =
document.getElementsByTagName("shutdown").item(0);


        if(!address.isEmpty()) {

            addressNode.setTextContent(address);

        }

        else {

            Node ipNode = document.getElementsByTagName("ip").item(0);

            ipNode.getParentNode().removeChild(ipNode);

        }

        if(!description.isEmpty()) {

            descriptionNode.setTextContent(description);

        }

        else {

            descriptionNode.getParentNode().removeChild(descriptionNode);

        }

        if(!shut.isEmpty()) {

            shutdownNode.setTextContent(shut);

        } else {

            shutdownNode.getParentNode().removeChild(shutdownNode);
```

```
        }

        if (operation.equals("delete")) {
            Element loopbackElement = (Element)
interfaceNode.getParentNode();
            loopbackElement.setAttribute("xc:operation", "delete");
        }


        interfaceNode.setTextContent(interface_id);
        XML message = device.executeRPC(DocumentToString(document));
        System.out.println(message);
    }
<... output omitted ...>
```

The **editConfig** method's purpose is to programmatically modify the configure-interface.xml payload with the user-provided input. Users may create new loopback interfaces and add the address, description, and shutdown state.

2. Open the **App.java** file, and in the main method, implement the user query prompt for the edit-config operation.

```
<... output omitted ...>
            else if(command == 2) {
                input.nextLine();
                System.out.print("datastore: running/candidate [running]:
");
                String datastore = input.nextLine().strip();
                System.out.print("loopback id [0]: ");
                String interface_id = input.nextLine();
                System.out.print("operation: merge/delete [merge]: ");
                String operation = input.nextLine();
                String address = "";
                String description = "";
                String shut = "";
                if (interface_id.isEmpty()) {
                    interface_id = "0";
                }
                if (operation.isEmpty()) {
                    operation = "merge";
                }
```

```
                if (operation.equals("merge")) {
                    System.out.print("ip address: ");
                    address = input.nextLine();
                    System.out.print("interface description: ");
                    description = input.nextLine();
                    System.out.print("shutdown: yes/no [no]");
                    shut = input.nextLine();
                }
                netconf.editConfig(device, datastore,
configureInterfacePayload, operation, interface_id, address,
description, shut);
            }
```

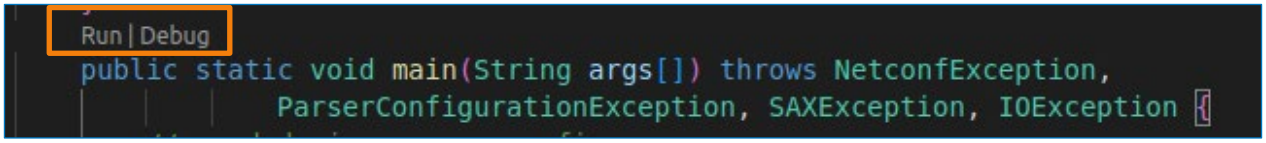
The user can choose to configure or delete an interface. When deleting the interface only the interface ID is required as input, when configuring an interface, the user has the option to specify the address, description, and shutdown status. The input is then passed to the **editConfig** method, which populates the XML payload and sends the message to the device.

3. Implement the third user input option, which is a NETCONF commit command.

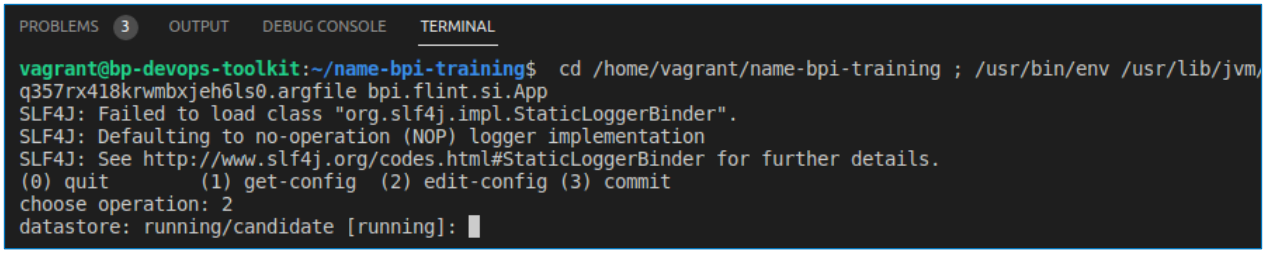
```
            else if(command == 3) {
                device.commit();
            }
```

In this implementation, the **commit** method provided by the NETCONF library is used.

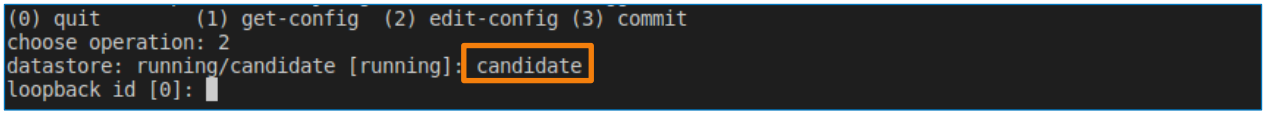
4. Save the file **App.java** file and run the application from VS Code.



5. In the VS Code integrated terminal, type **2** to select the edit-config operation, and press **Enter**.



6. Next, type **candidate** to select the candidate datastore as the target for configuration.



7. Choose a loopback ID for the loopback interface you want to configure.

```
(0) quit       (1) get-config  (2) edit-config (3) commit
choose operation: 2
datastore: running/candidate [running]: candidate
loopback id [0]: 10
operation: merge/delete [merge]:
```

8. Press **Enter** to select the default merge operation.
9. Enter the IP address, description, and shutdown option.

```
(0) quit       (1) get-config  (2) edit-config (3) commit
choose operation: 2
datastore: running/candidate [running]: candidate
loopback id [0]: 10
operation: merge/delete [merge]:
ip address: 10.10.10.10
interface description: test netconf message
shutdown: yes/no [no]:
```

10. Observe the rpc-reply message after the last input, it should say <ok/>, meaning that the configuration was successfully applied.

```
datastore: running/candidate [running]: candidate
loopback id [0]: 10
operation: merge/delete [merge]:
ip address: 10.10.10.10
interface description: test netconf message
shutdown: yes/no [no]:
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="2">
    <ok/>
</rpc-reply>
```

11. Select a **get-config** operation (operation=1) and look in the **running** datastore for the loopback (id=10) interface that you configured in the previous steps.

```
(0) quit       (1) get-config  (2) edit-config (3) commit
choose operation: 1
datastore: running/candidate [running]:
loopback id: 10
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="3">
    <data>
        <interfaces xmlns="http://flint.si/ns/yang/dev-sim"/>
    </data>
</rpc-reply>
```

The running datastore is empty because you used a candidate datastore when configuring the interface and have not yet committed the configuration. Alternatively, you could configure the running datastore directly, but not all devices support that.

12. Select a **get-config** operation and look in the **candidate** datastore for the loopback interface (id=10) that you configured in the previous steps.

```
(0) quit       (1) get-config  (2) edit-config (3) commit
choose operation: 1
datastore: running/candidate [running]: candidate
loopback id: 10
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="4">
    <data>
        <interfaces xmlns="http://flint.si/ns/yang/dev-sim">
            <loopback>
                <name>10</name>
                <ip>
                    <address>10.10.10.10</address>
                </ip>
                <description>test netconf message</description>
                <shutdown>no</shutdown>
```

The candidate datastore has the configuration you applied with the previous edit-config command.

13. Commit the configuration by selecting the **third** (3) operation in the program.

NOTE: The commit operation does not output any response.

```
(0) quit        (1) get-config  (2) edit-config (3) commit
choose operation: 3
```

14. Select a **get-config** operation and look in the **running** datastore for the loopback interface that you configured in the previous steps.

```
(0) quit        (1) get-config  (2) edit-config (3) commit
choose operation: 1
datastore: running/candidate [running]:
loopback id: 10
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="6">
    <data>
        <interfaces xmlns="http://flint.si/ns/yang/dev-sim">
            <loopback>
                <name>10</name>
                <ip>
                    <address>10.10.10.10</address>
                </ip>
                <description>test netconf message</description>
                <shutdown>no</shutdown>
            </loopback>
```

After the commit, the running datastore is configured with the loopback interface.

15. Delete the previously configured interface from the running datastore by using the **edit-config** operation.

```
(0) quit        (1) get-config  (2) edit-config (3) commit
choose operation: 2
datastore: running/candidate [running]:
loopback id [0]: 10
operation: merge/delete [merge]: delete
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="7">
    <rpc-error>
        <error-type>application</error-type>
        <error-tag>unknown-element</error-tag>
        <error-info>
            <bad-element>running</bad-element>
        </error-info>
        <error-severity>error</error-severity>
        <error-message>Failed to find YANG spec of XML node: running with parent: target in namespace: urn:ietf
```

An rpc-error message indicates that you cannot make direct configuration changes in the running datastore. Some NETCONF devices support this, but the device in your lab does not, as indicated in the hello message capabilities in Task 2.

16. Delete the interface using the **candidate** datastore and commit operation.

```
(0) quit        (1) get-config  (2) edit-config (3) commit
choose operation: 2
datastore: running/candidate [running]: candidate
loopback id [0]: 10
operation: merge/delete [merge]: delete
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="8">
    <ok/>
</rpc-reply>

(0) quit        (1) get-config  (2) edit-config (3) commit
choose operation: 3
(0) quit        (1) get-config  (2) edit-config (3) commit
```

17. Check the running datastore configuration and make sure that the deleted interface no longer exists.

```
(0) quit        (1) get-config  (2) edit-config (3) commit
choose operation: 1
datastore: running/candidate [running]:
loopback id: 10
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="9">
    <data>
        <interfaces xmlns="http://flint.si/ns/yang/dev-sim"/>
    </data>
</rpc-reply>
```

End of Lab

Lab 3: Using REST APIs and RESTCONF

Objectives

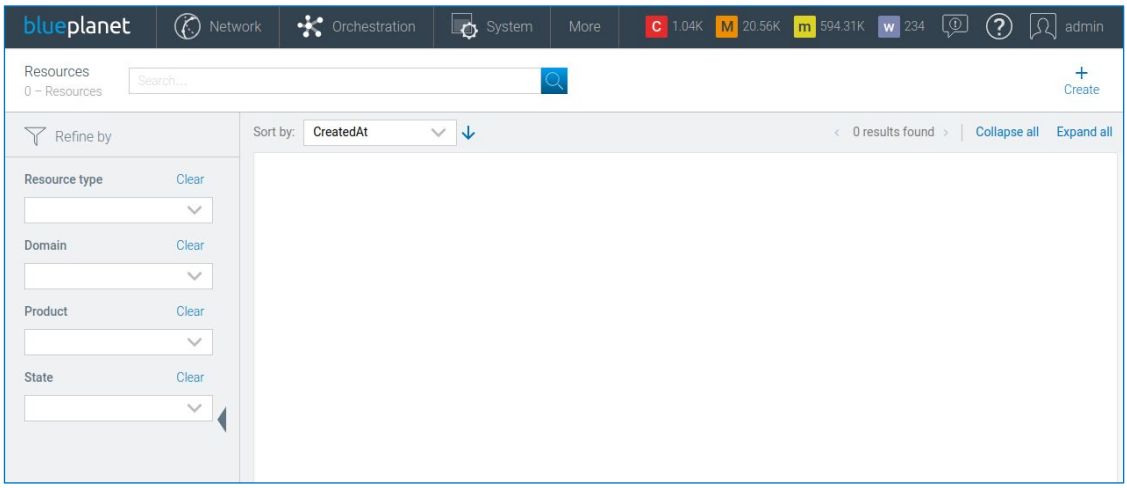
- Using Swagger UI to explore REST API calls
- Using Swagger UI to send requests
- Using Postman to test REST APIs
- Managing network devices with RESTCONF

Task 1: Using Swagger UI to Explore and Send API Calls

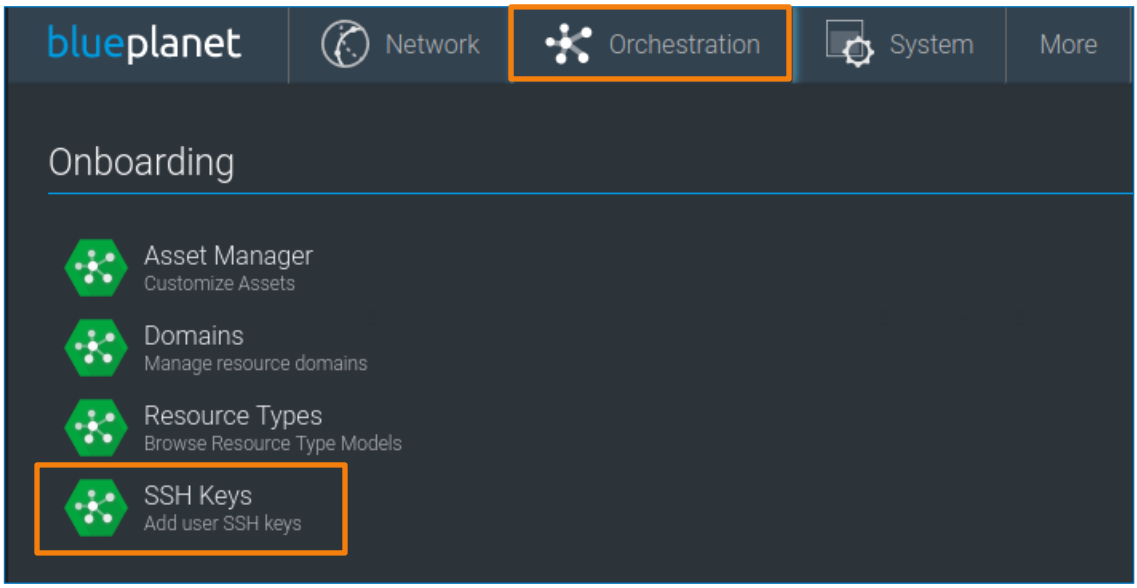
In this task, you leverage the REST API to communicate with the Blue Planet server deployed in your lab environment. The server has a REST API that can be used to perform certain tasks and it also supports Swagger UI that you use to explore and test the API.

In this example, you manage SSH keys that are stored on the server for authentication through the API. First, you will try to fetch a list of all currently stored SSH keys. Then, you will also try to add and remove additional keys.

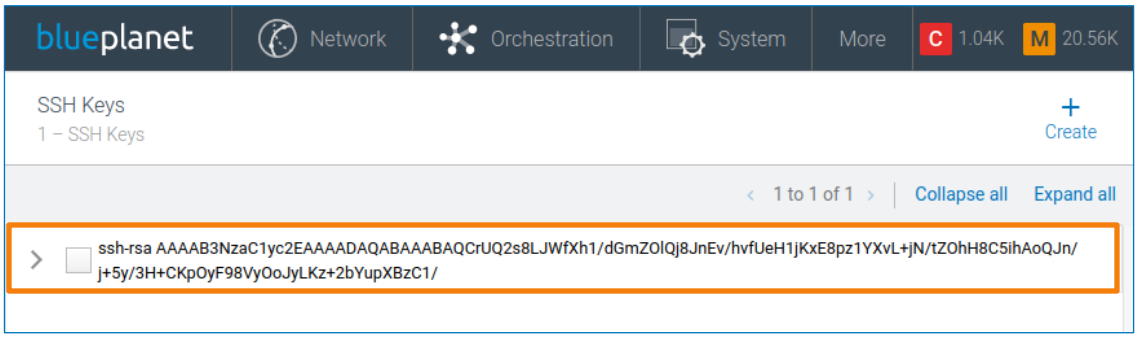
1. First, check the list of SSH keys through the web UI of the server. To open the web UI, open your web browser and navigate to *http://localhost:9980/orchestrate/*. The web UI of the server used in this lab opens.



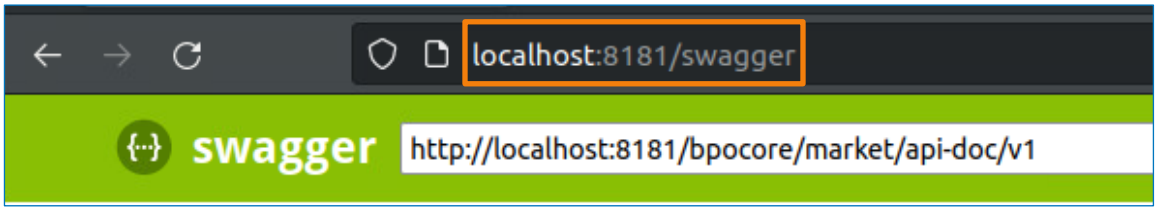
2. Navigate to the **Orchestration** tab and choose **SSH Keys** from the dropdown menu.



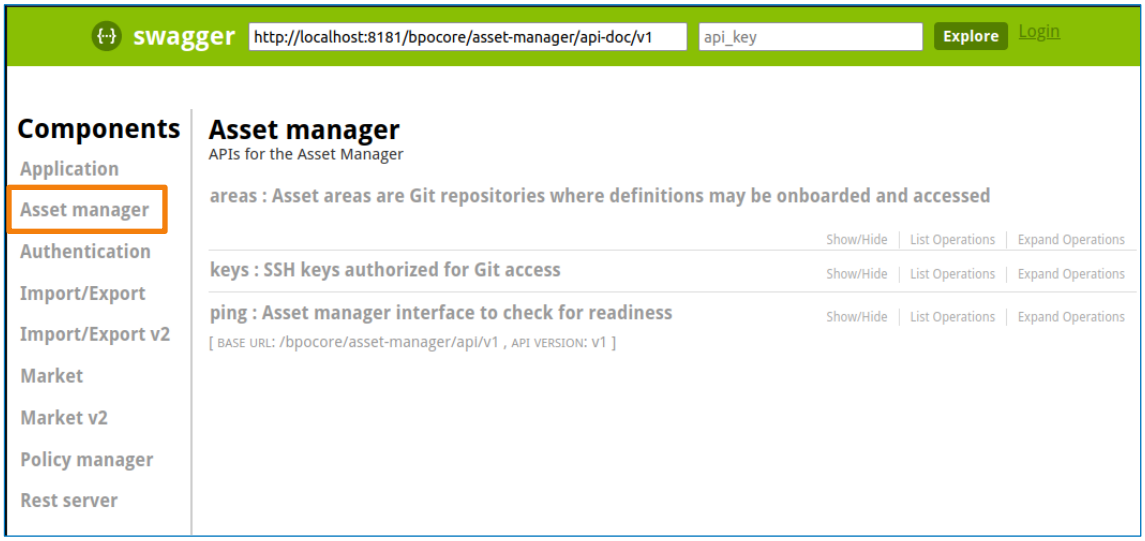
3. You can see that there is currently only one SSH public key stored on the server. Only the device with the corresponding private key can currently be authorized.



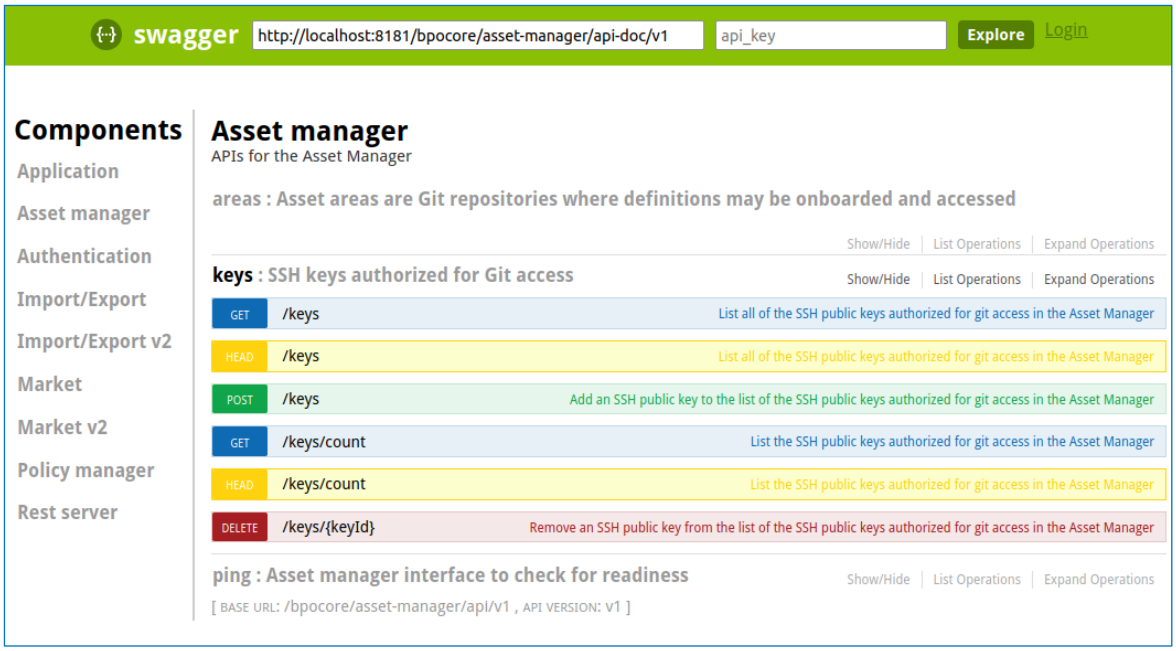
4. The same list can be retrieved using the REST API exposed on the server. Use Swagger UI to explore how the API call must be structured. Swagger UI is available at ***localhost:8181/swagger***.



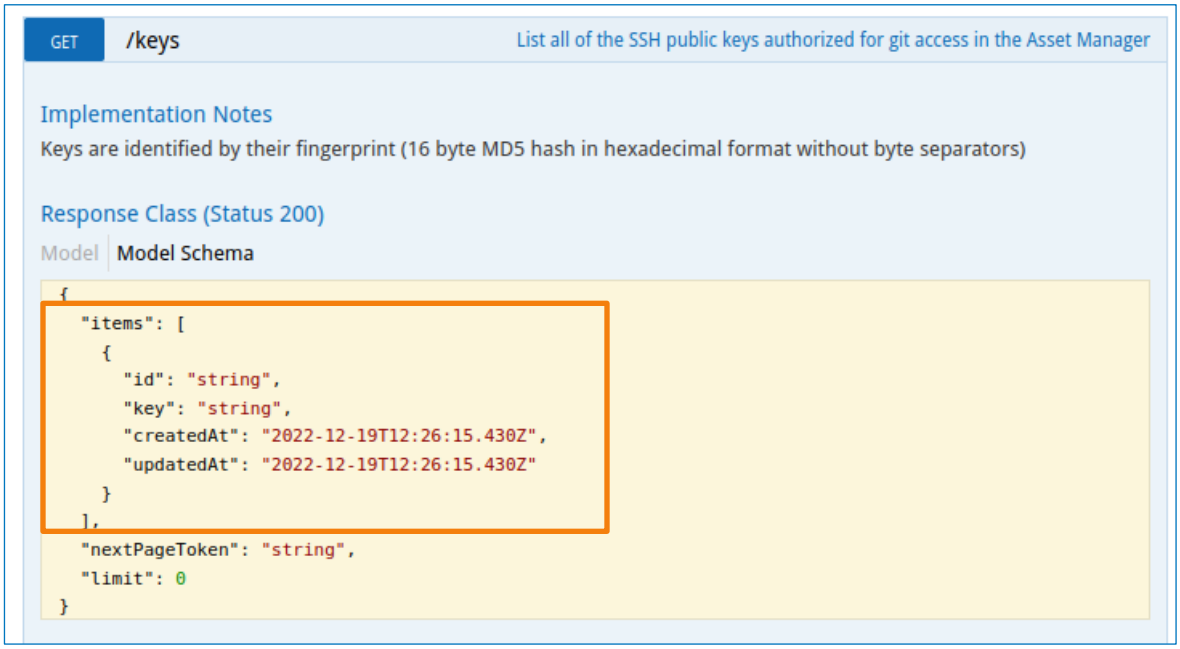
5. API Calls related to managing SSH keys can be found under the **Asset manager** menu.



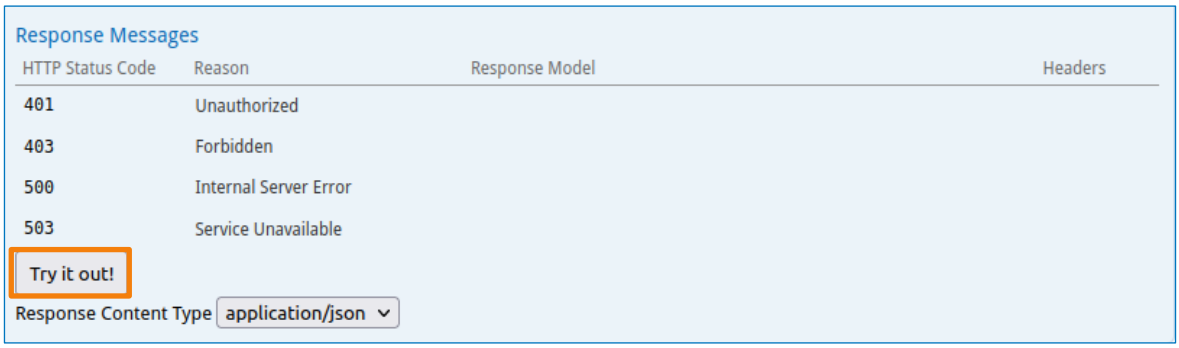
6. Expand the **keys** option to explore all the API calls. You can see that a list of all the stored SSH public keys can be retrieved by sending a **GET** request to the /keys endpoint.



7. When a GET request is received on the API, the server responds with the representation of the target resource. To see the details of the expected response, expand the mentioned GET request. You can see that the response body will include a list of SSH keys. Every SSH key also has an ID and a timestamp. Knowing the structure of a response is crucial when developing the code that retrieves that data.



8. Besides observing the structure of requests and responses, you can also send the request directly from the Swagger UI. Click the **Try it out** button to send the request.



9. The request should be successful (Response Code: 200 OK) and the response body should be similar to the following output.

```
{
  "items": [
    {
      "id": "b23c0df31f8eef47d02433c876d2aa8c",
      "key": "ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQCrUQ2s8LJWfXh1/dGmZOlQj8JnEv/hvfUeH1jKxE8p
z1YXvL+jN/tZOhH8C5ihAoQJn/j+5y/3H+CKpOyF98VyOoJyLKz+2bYupXBzC1/GJf5W8q0K
f+Gk0J5xebd7W8XvH9XJEtGthUmFgSV8LkR/H7VUJxgNd7V3z/n9X7V/QZ/uCByVsmQvLmhI
6U+ba6uDG6UiJemUaT6UjT6TnV7sA6vGmOe+U70/yU6T8U6pXU6dOm6TZT6SfU",
```

```
      "createdAt": "2022-12-19T13:39:53.851Z",
      "updatedAt": "2022-12-19T13:39:53.851Z"
    }
  ],
  "limit": 1000
}
```

10. Now you will upload your public SSH key to the Blue Planet server in your environment using the REST API. Open a new terminal window on your VM and navigate to **~/.ssh** to find your public key. Output the public key using the cat command.

NOTE: Your key is probably different from the one shown in the following output.

```
[~]$ cd /home/vagrant/.ssh/
[~/.ssh]$ cat id_rsa.pub
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQC/sHxu+Kjy6gMWR8jzKKB433SEwMZ7Uy4Zr9vwnG1F
XRJUi6wzdjn783JPf6r1lK4rLBPEJeALTKeGJAhR6lDyiMUG2gCEBAlqwxUJY2DNPUnnTJx9
A3E75s/jJaCSkke7LMZPgL5zYrr0VOKMvCOrU94RR5OMz3E7bDpHzKcqOrHjxrcd8V0Qk867
YcSyiVdwZCPSDjeAdgRAf00uxcj17kRLWIOzXbYqBFIFD19ARHB1uP7jIuQq+BHedTAtBeoP
/O9hYVnvkJ25xUwYCsvrCbaedFYzhpEhFsiJtej8nPUx8O3s43HhxM/eDcbWjpMQD2ciietp
ew91bagrttm3 vagrant@bp-devops-toolkit
```

11. Copy your key to the clipboard and make sure that you do not copy any additional whitespaces then navigate back to the Swagger UI. Expand the **keys** section again and choose the **POST request** that adds an SSH public key to the list.

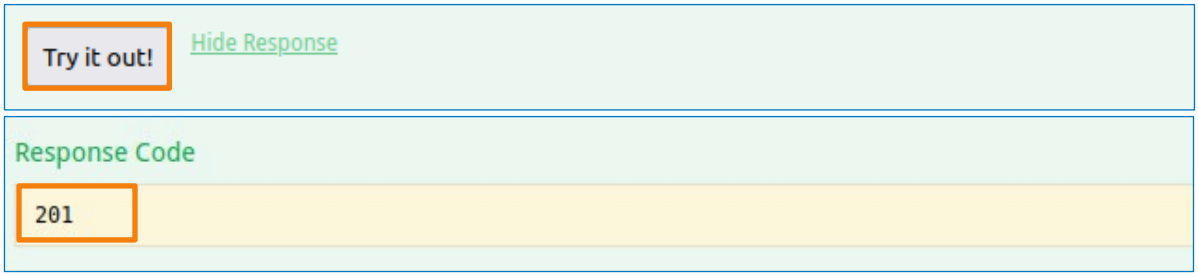
12. You can see that this request requires the **key** parameter as the body of the request. Paste your SSH key to the **Parameters** section. Make sure that your input follows the structure shown in the schema next to the empty field.

NOTE: Your key is probably different from the one shown in the following output.

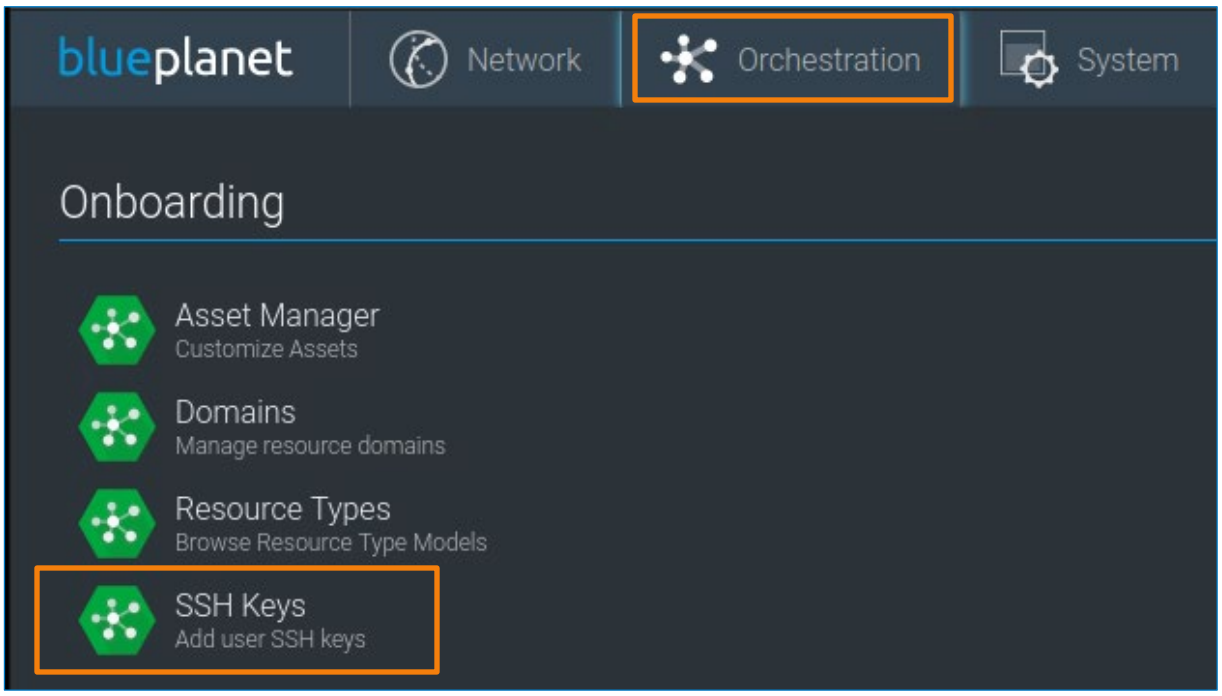
```
{
"key": "ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQC/sHxu+Kjy6gMWR8jzKKB433SEwMZ7Uy4Zr9vwnG1F
XRJUi6wzdjn783JPf6r1lK4rLBPEJeALTKeGJAhR6lDyiMUG2gCEBAlqwxUJY2DNPUnnTJx9
```

```
A3E75s/jJaCSkke7LMZPgL5zYrr0VOKMvCOrU94RR5OMz3E7bDpHzKcqOrHjxrcd8V0Qk867
YcSyiVdwZCPSDjeAdgRAf00uxcj17kRLWIOzXbYqBFIFD19ARHB1uP7jIuQq+BHedTAtBeoP
/O9hYVnvkJ25xUwYCsvrCbaedFYzhpEhFsiJtej8nPUx8O3s43HhxM/eDcbWjpMQD2ciietp
ew91bagrttm3 vagrant@bp-devops-toolkit"
}
```

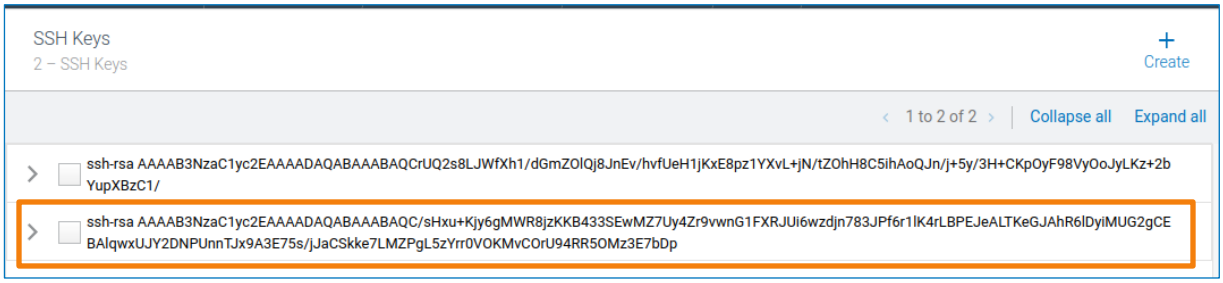
13. Click the **Try it out!** button to send the request. Verify that the response code is **201**.



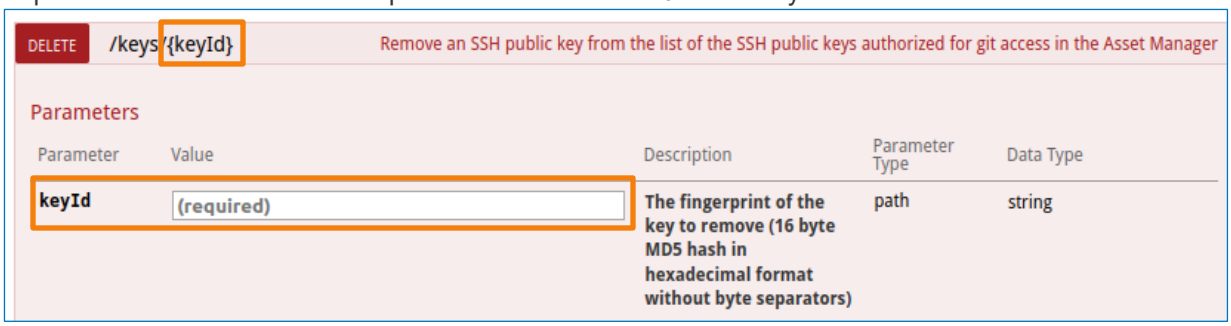
14. Go back to the web UI of the server to verify that the list of SSH keys now includes your public key as well. If you closed the previously opened tab, point it to the following URL: *http://localhost:9980/orchestrate/*. Click on the **Orchestration** tab and choose **SSH Keys** from the dropdown menu.



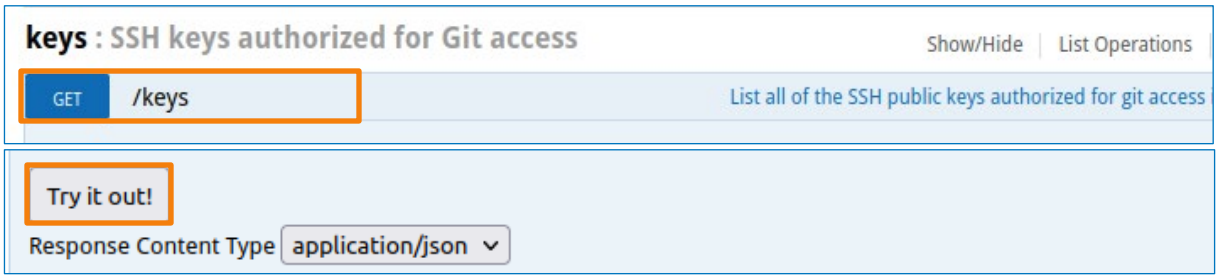
15. The list of currently stored keys opens. Your key should now be included on that list.



16. The SSH keys can also be deleted from the server through the API. Navigate back to the Swagger UI and observe how the request that removes a key should look like by expanding the **DELETE** request. You can see that the request must include the **Id** of the key that must be deleted.

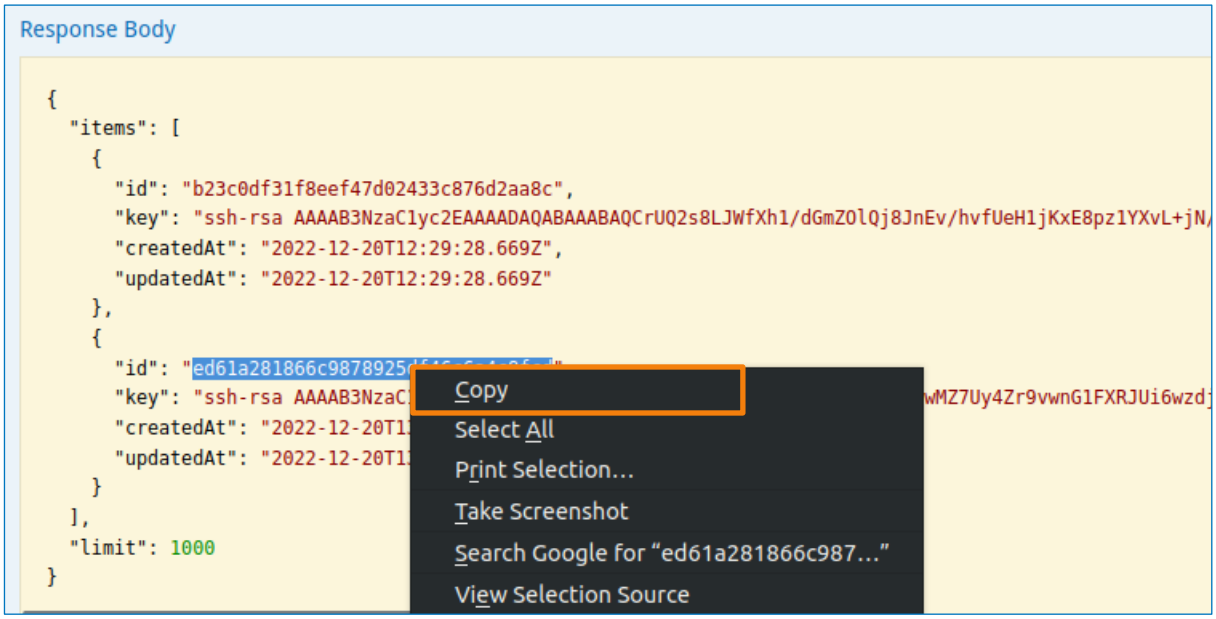


17. To find the Id of each key, fetch the list of keys again. Expand the first **GET** request again and click the **Try it out!** button.



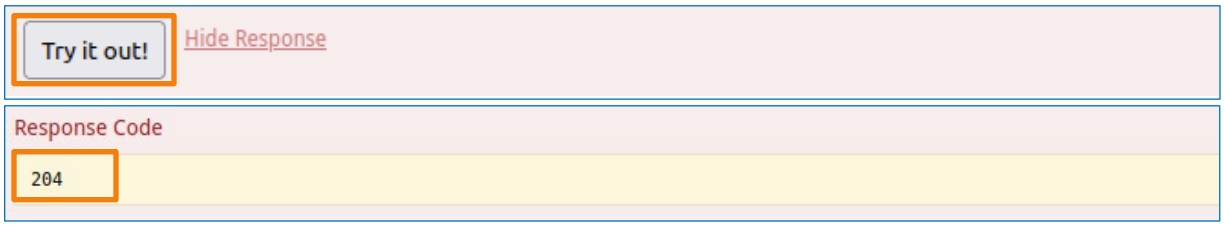
18. Copy the **id** of your public key to the clipboard.

NOTE: The id will probably be different from the one shown below.

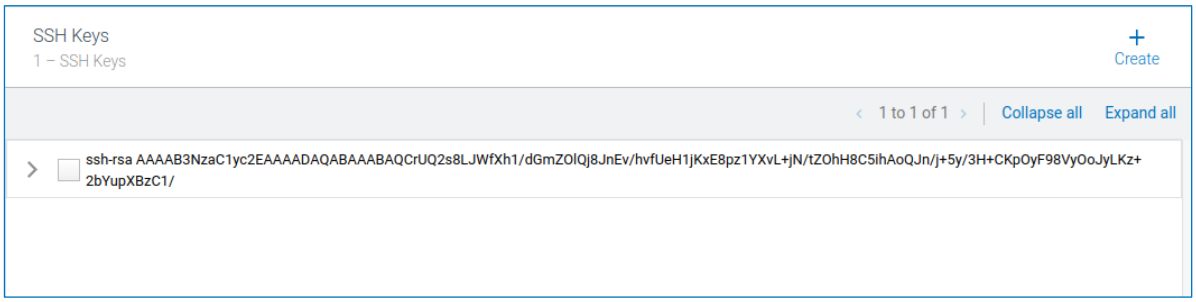


19. Expand the **DELETE** request again and paste the **id** to the **parameters** section.

20. Click the **Try it out!** button to send the request and delete your key from the server. The response code should be **204**.



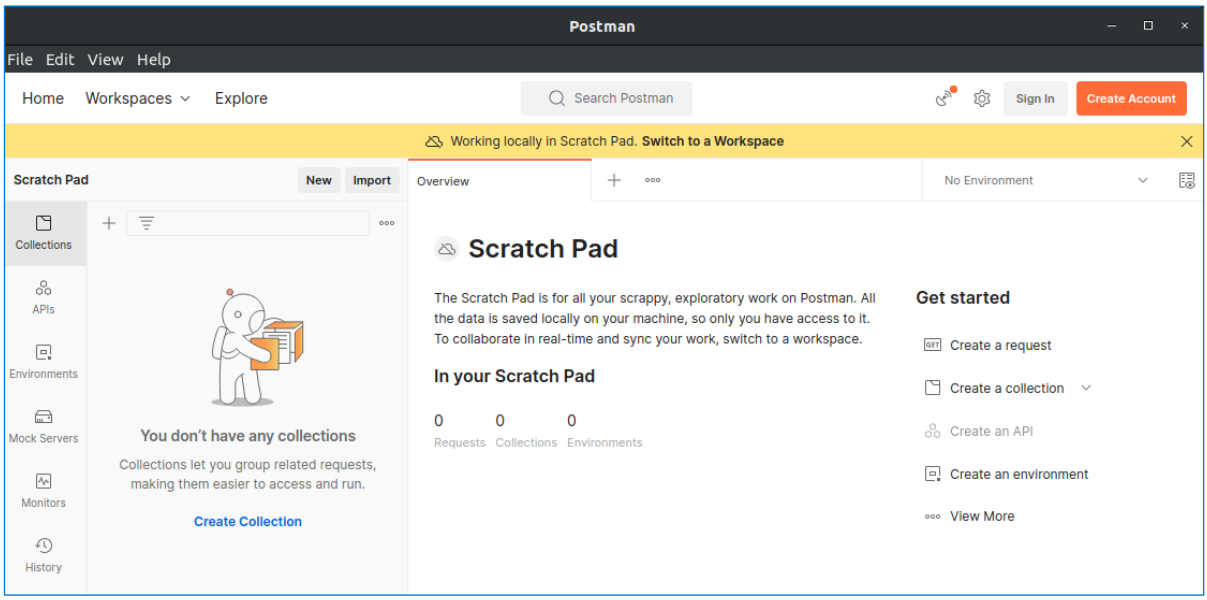
21. Go back to the list of SSH keys on the web UI and verify that your key is not present there anymore.



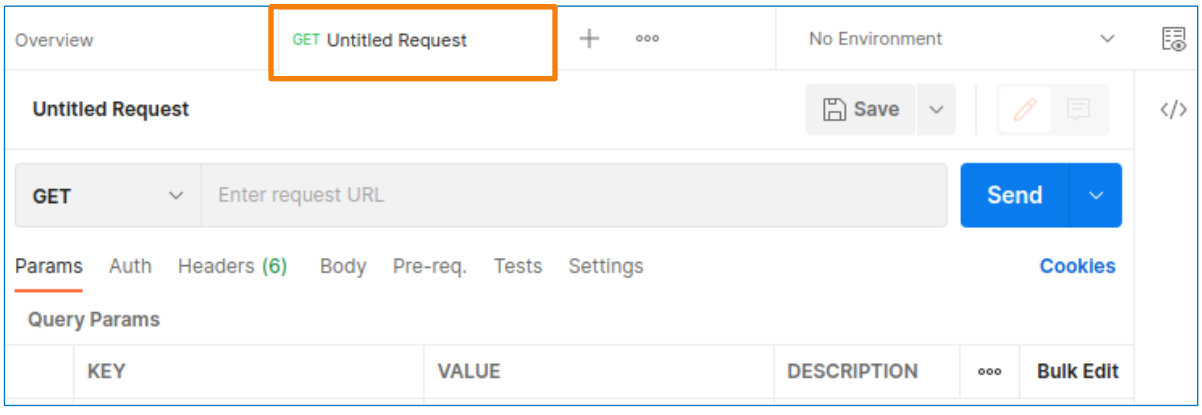
Task 2: Using Postman to Construct and Send API Calls

Using Swagger UI is usually just the first step when using the REST API. It is useful to explore the possible operations and the structure of the requests and responses. The next step is developing the code that will perform certain tasks through the API. Postman is a great tool that can help with the development process. In this task, you will use Postman to construct and send the requests and observe the responses.

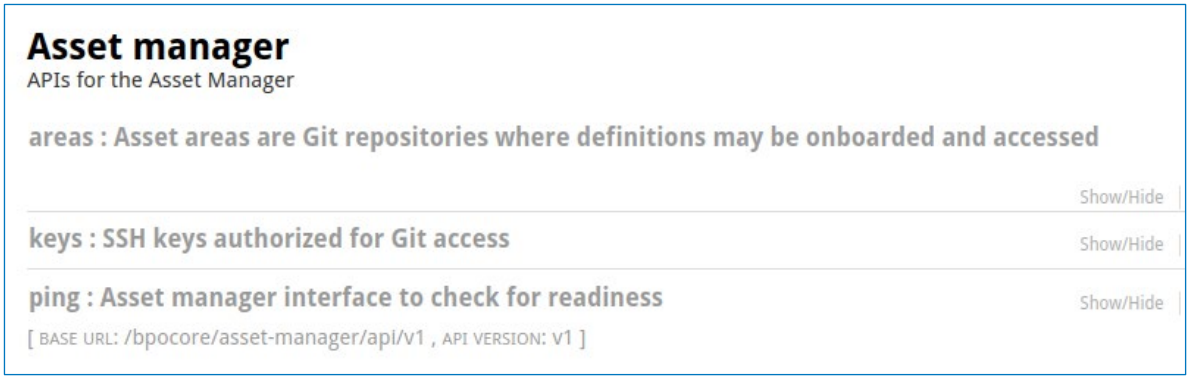
1. Open the Postman application from your desktop.



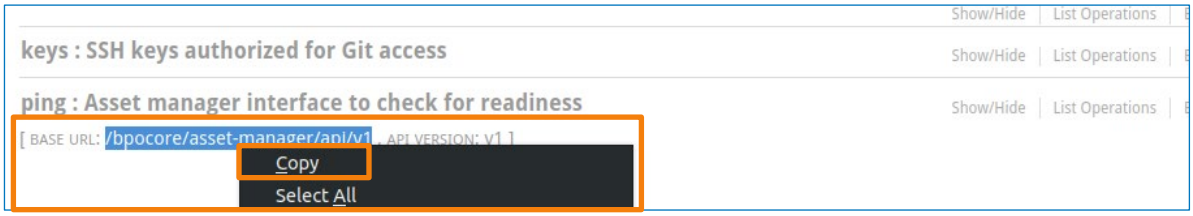
2. Click the **New** button and choose **HTTP Request** from the menu that opens to create a new request. A new blank request page should open.



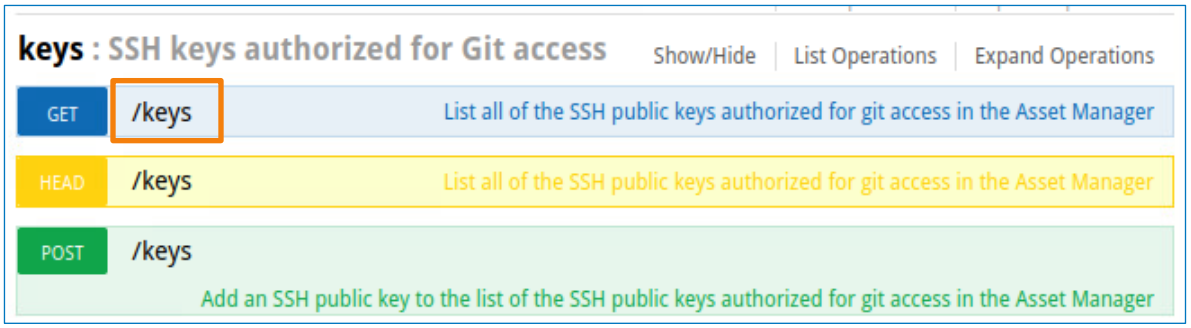
3. Now you construct a **GET** request that retrieves a list of the SSH keys stored on the server. Navigate back to the **Swagger UI** and open API Calls for **Asset Manager** to observe how the request must be structured.



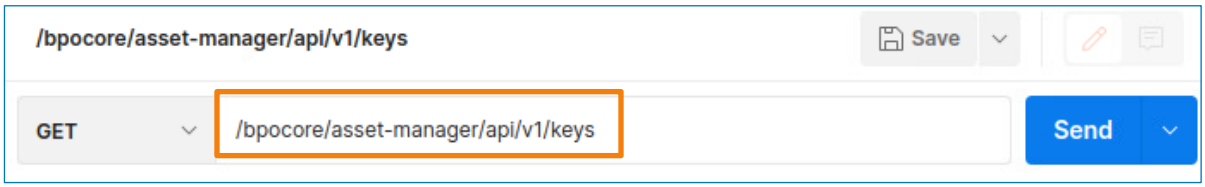
4. The first thing that you need to know is the URI of the resource that contains the list of SSH keys. The base URI for all the **Asset manager** resources can be found at the bottom of the page. Copy that URI to the clipboard as you will need it to send the request.



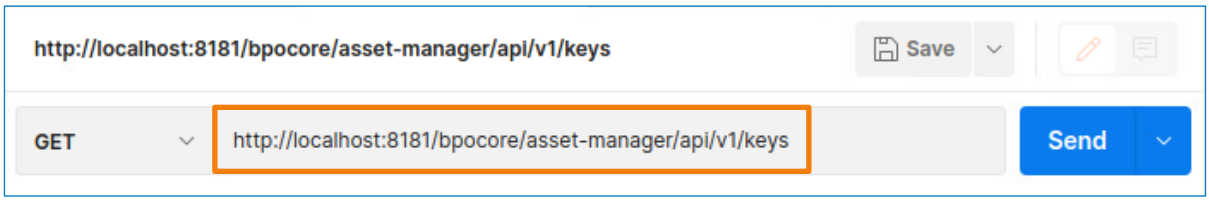
5. There are different resources stored under the base URI. In this case, you are interested in the resource that contains all the SSH keys. To find the URI of that resource, expand the **keys** section. You can see that SSH keys are stored under the **/keys** resource. This means that the URI of the target resource is **/bpocore/asset-manager/api/v1/keys**.



6. Navigate back to **Postman** and enter the target resource in the request URL field. Paste the base URI that you copied in step 4 and append the **/keys** path. You should obtain the following URI.



7. The request URL must also include the web address of the server that must receive the request. In this case, the application is running directly on the virtual machine that you are accessing, and the requests are expected on port **8181**. Prepend the address **localhost:8181** to the URI of the target resource.

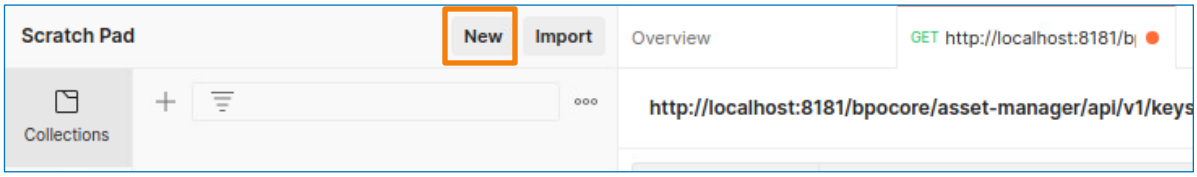


8. Make sure that the **GET** method is selected in the dropdown menu and **send the request**. The server should reply with a representation of the target resource. In this case, you should receive the list of existing SSH keys.

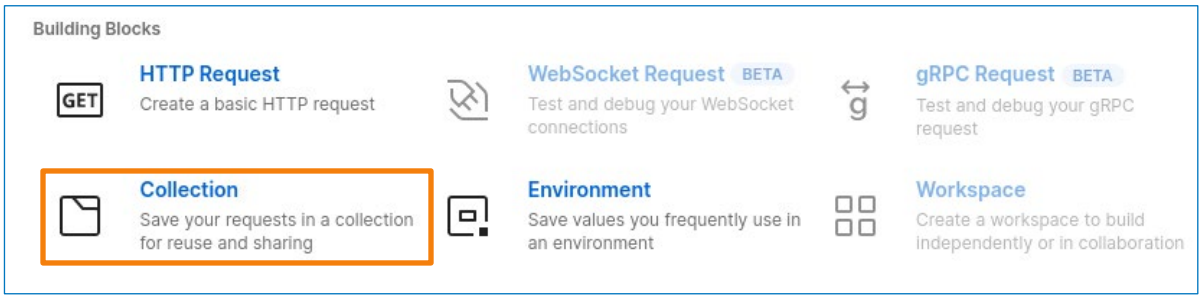
NOTE: There should currently be only one SSH key stored on the server.

9. Now you will save the created request to a Postman collection. Since there are currently no existing collections, click the **New** button to create a new one.

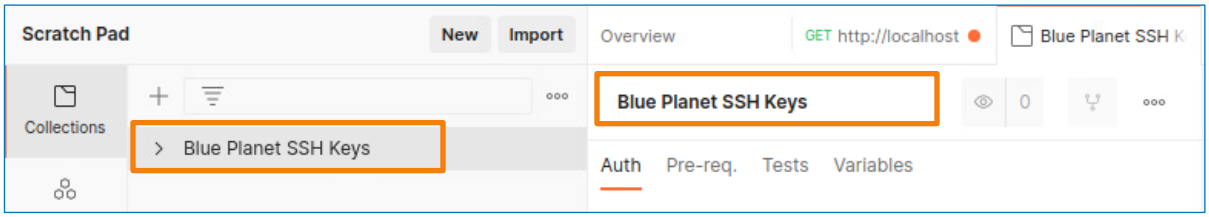
NOTE: There Different API calls are often related to the same workflow or feature. Postman enables you to group related API calls to **collections**.



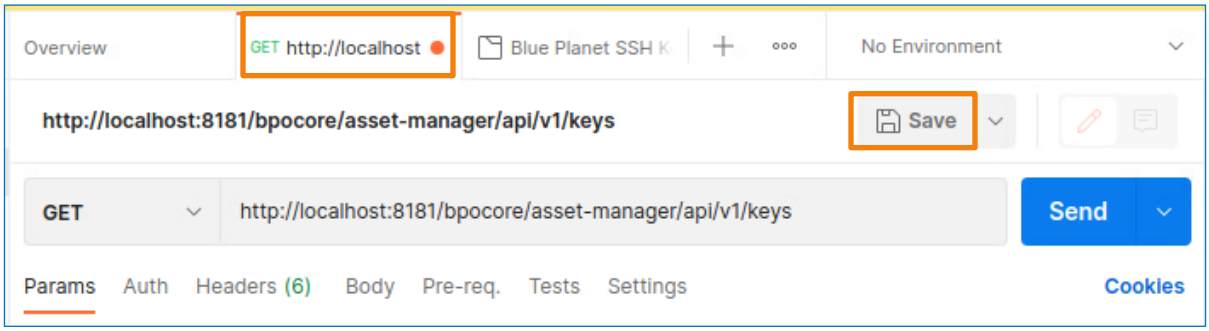
10. Choose **Collection** from the pop-up menu.



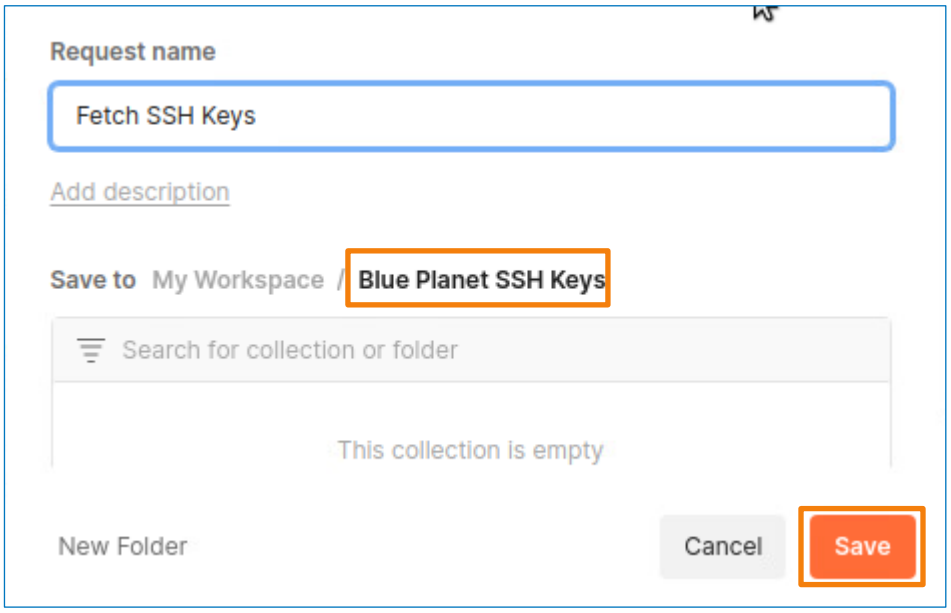
11. Name the collection '**Blue Planet SSH Keys**'.



12. Navigate back to the previously created **GET** request by clicking on it in the opened tabs menu. Click **Save** to save the request.

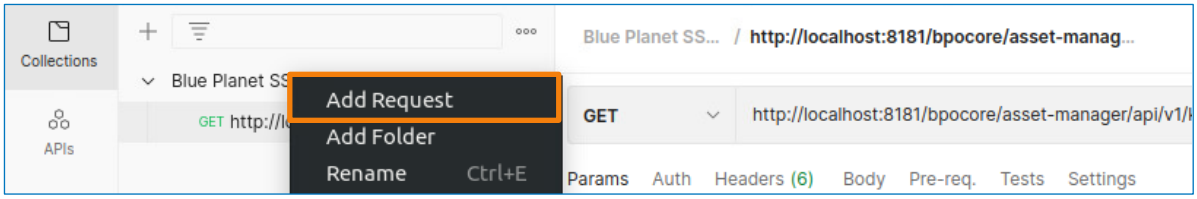


13. Name the request '**Fetch SSH Keys**' and choose your new **Blue Planet SSH Keys** collection from the pop-up menu and click **Save**.

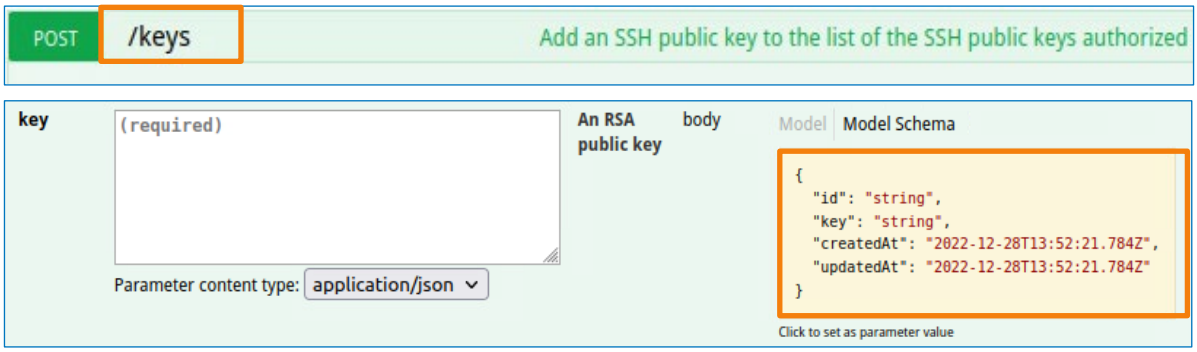


Your request is now saved to the collection. API calls stored at the same collection can for example share authentication data or pre-request and test scripts, they can also be executed at once.

14. You will now add a new related API call that adds a new SSH key to the collection. Right-click on your collection and select **Add Request**.

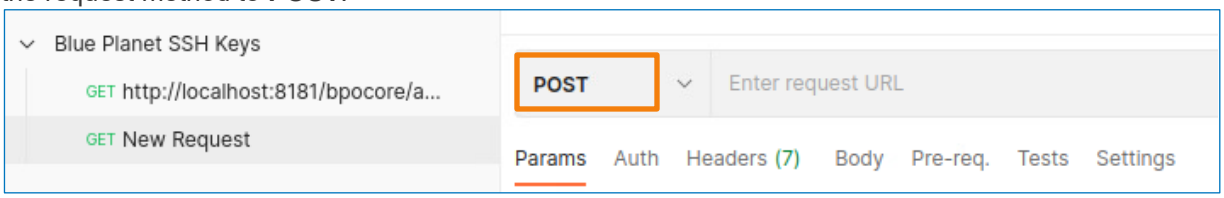


15. Navigate back to the Swagger UI to examine how the request to post a new key to the server must be structured. Expand the **POST** request that adds an SSH public key to the list.



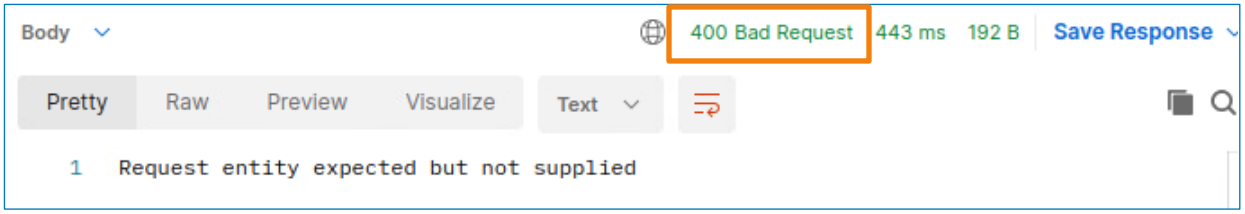
You can see that the *URI* is the same as in the GET request that you have used previously because you are working with the same resource. The difference is only that in this case, you want to write a new key to that resource. The structure of the body is also shown in Swagger UI. There are 4 different parameters but the only one required is the '**key'** parameter, which contains your public key.

16. Go back to Postman and construct the request according to Swagger UI documentation. First set the request method to **POST**.



17. Now copy the complete URL from the GET request you created previously and paste it into the request URL field.

18. As you can see in the documentation, this POST request also requires some parameters in the body. You can try sending the request with no payload first and observe that it will fail.

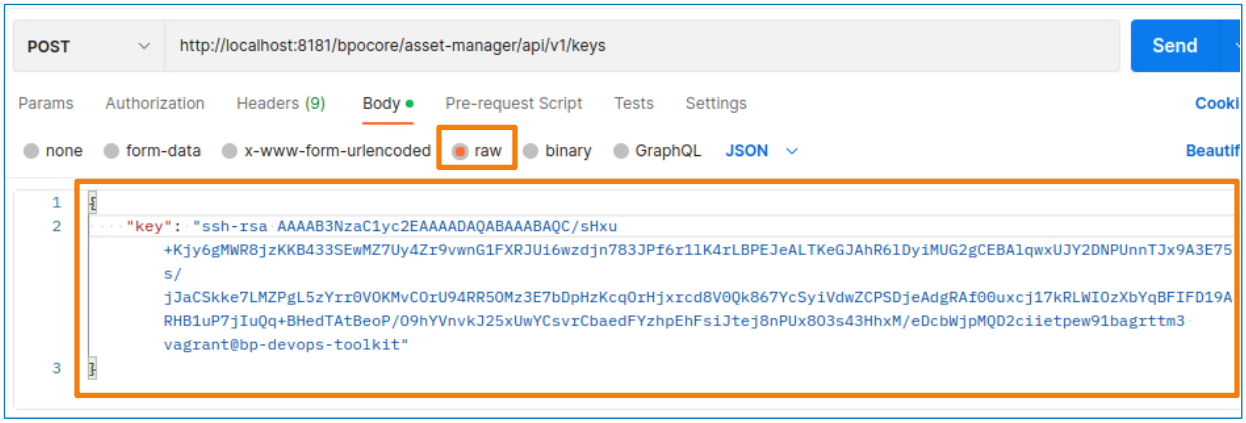


19. Add your public SSH key to the body of the request. Open a new terminal window and navigate to ~/.ssh to find your public key. Output the public key using the cat command.

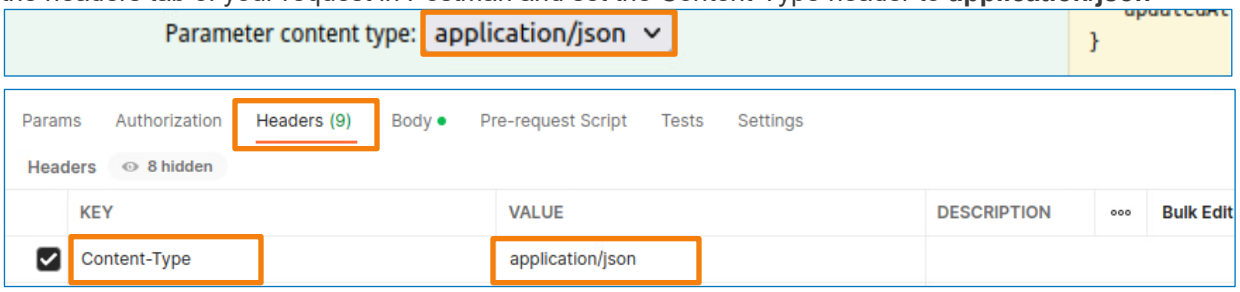
NOTE: Your key will probably be different from the one shown in the following output.

```
[~]$ cd /home/vagrant/.ssh/
[~/.ssh]$ cat id_rsa.pub
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQC/sHxu+Kjy6gMWR8jzKKB433SEwMZ7Uy4Zr9vwnG1F
XRJUi6wzdjn783JPf6r1lK4rLBPEJeALTKeGJAhR6lDyiMUG2gCEBAlqwxUJY2DNPUnnTJx9
A3E75s/jJaCSkke7LMZPgL5zYrr0VOKMvCOrU94RR5OMz3E7bDpHzKcqOrHjxrcd8V0Qk867
YcSyiVdwZCPSDjeAdgRAf00uxcj17kRLWIOzXbYqBFIFD19ARHB1uP7jIuQq+BHedTAtBeoP
/O9hYVnvkJ25xUwYCsvrCbaedFYzhpEhFsiJtej8nPUx8O3s43HhxM/eDcbWjpMQD2ciietp
ew91bagrttm3 vagrant@bp-devops-toolkit
```

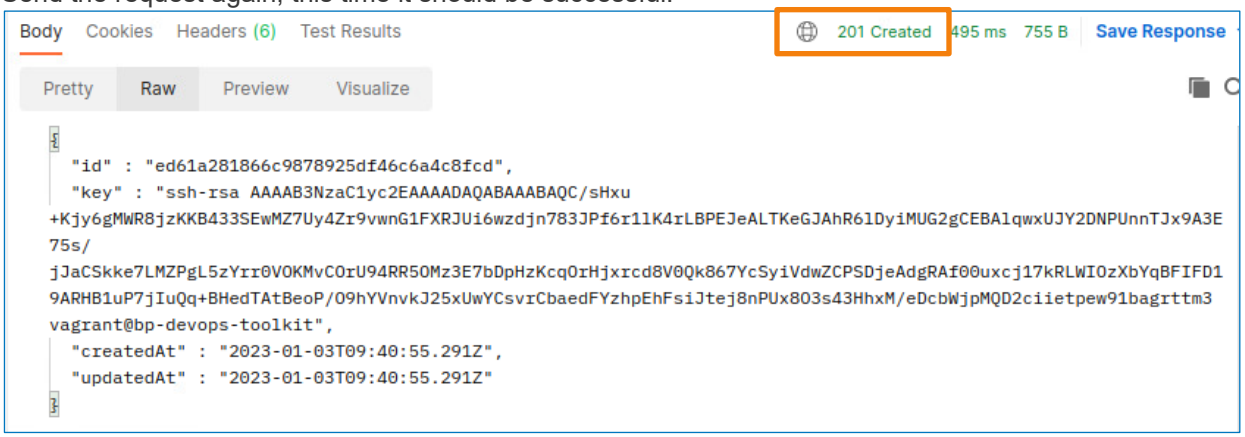
20. Copy the whole key and paste it to the body field in Postman. Set the type of encoding to **raw** and make sure that the payload follows the structure specified in the documentation.



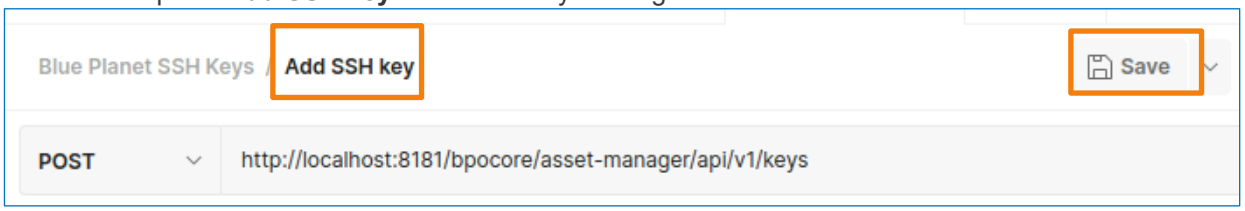
21. Before you send the request, you must set a new header that tells the server what is the type of content that was received. You can see that *content type* should be set to **application/json**. Open the headers tab of your request in Postman and set the Content-Type header to **application/json**



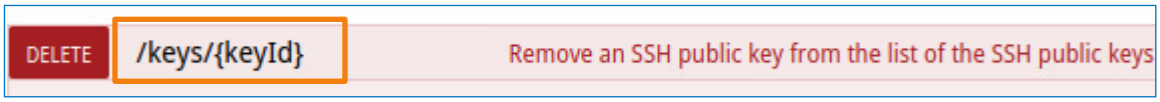
22. Send the request again, this time it should be successful.



23. Name the request '**Add SSH key**' and save it by clicking the save button.

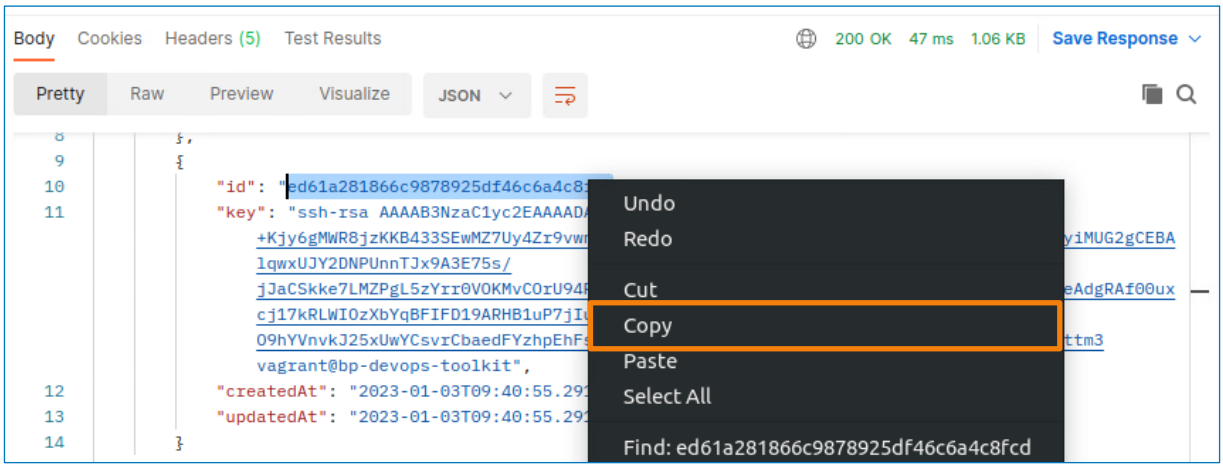


24. As you learned previously, the API can be used to delete the SSH keys as well. You will now use Postman to construct the **DELETE** request that you already used in Task 1. Navigate back to Swagger UI and examine the request to delete the SSH keys. Notice that the URI must include the id of the key that must be deleted.

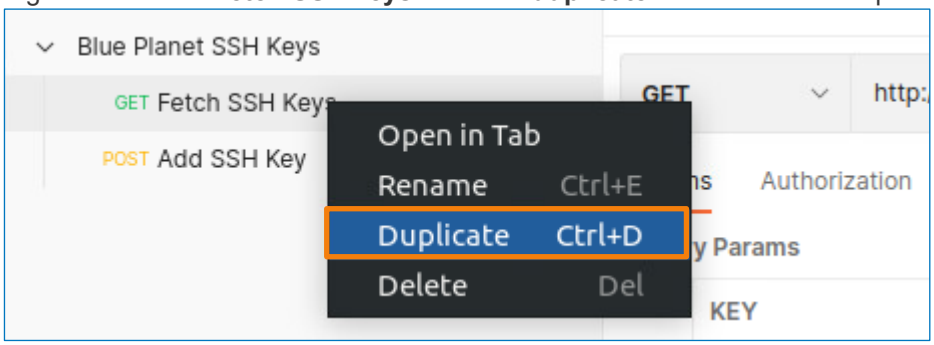


25. To get the keyId, send your **Fetch SSH Keys** request again. Look at the response body and copy the Id of your public SSH key that you added before.

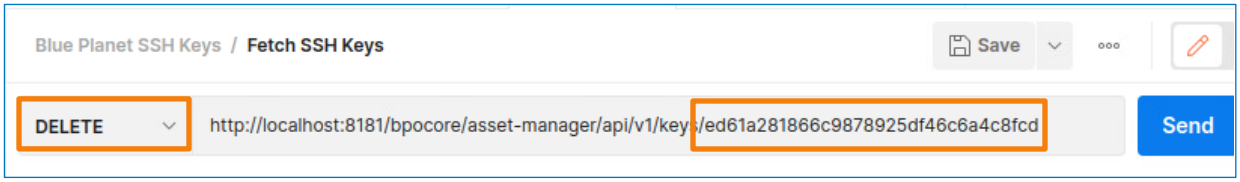
NOTE: The id in your case will probably be different from the one shown in the following figure.



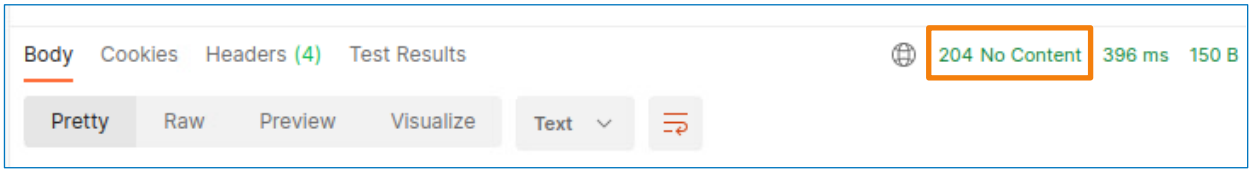
26. Right-click on the **Fetch SSH keys** and click **duplicate** to create a new duplicate request.



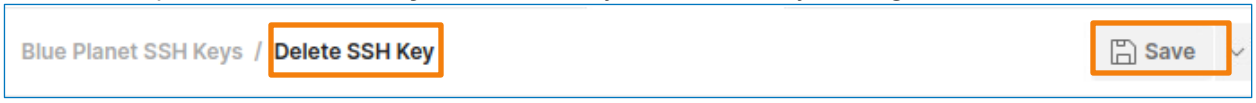
27. Open the new duplicate request, change the request method to **DELETE**, and paste the id of the key that must be deleted at the end of the URL, as shown in the following figure.



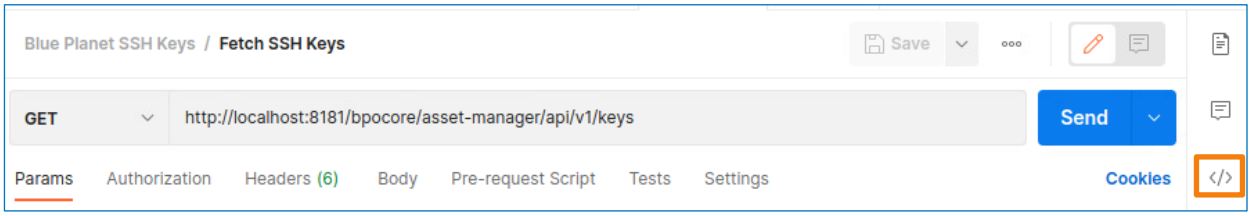
28. Send the request. The request should be successful, and the response code should be **204 No Content**.



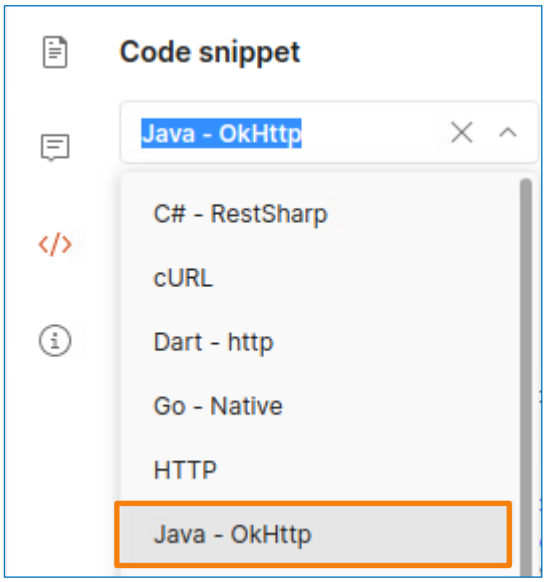
29. Name the request '**Delete SSH Key**' and save it to your collection by clicking the "Save" button.



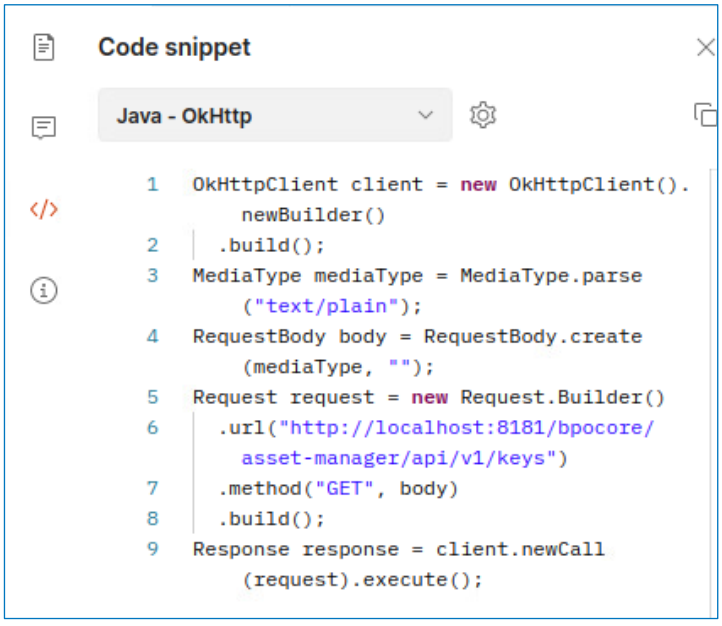
30. Besides providing you with a simple GUI to create and test API calls, Postman has another very useful feature that helps with the development process. Once you create the request, Postman can automatically generate a code that sends that request and stores the response. The code generator supports most of the languages that are popular today. To test the code generator feature, open your **Fetch SSH Keys** request and click the **Code** button.



31. Expand the dropdown menu and observe all the available languages. Choose **Java – OkHttp** option to generate Java code for the request.



32. Examine the code snippet provided by Postman. This code could be used as part of a Java program to send a GET request that fetches a list of SSH keys on the server.

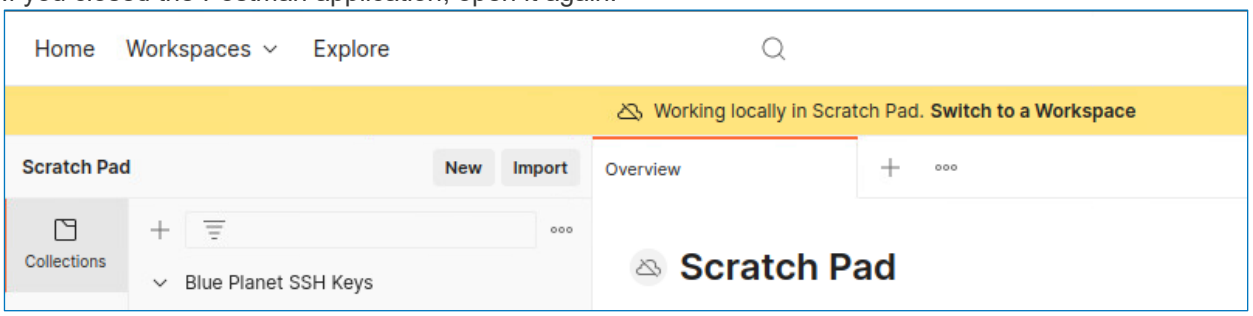


Task 3: Using RESTCONF to manage network devices

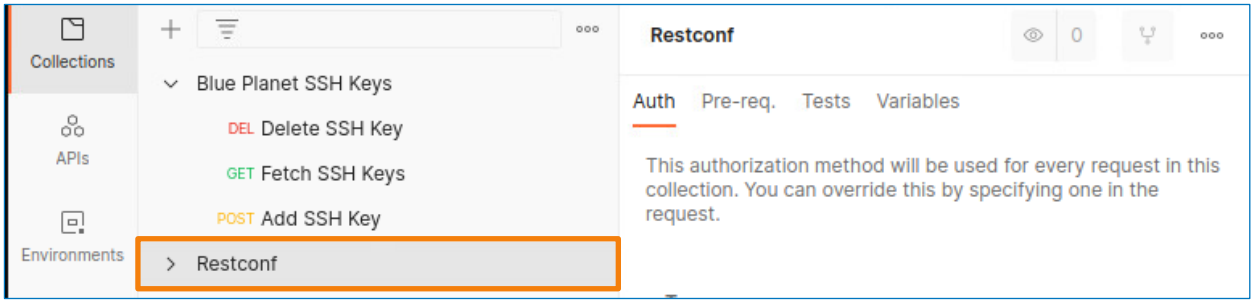
As you learned previously, your lab environment includes Clixon, a YANG-based device with CLI, NETCONF, and RESTCONF interface. In this task, you interact with the device through the RESTCONF interface.

1. One of the benefits of using RESTCONF is that HTTP(s) requests and responses are used for communication, and you can use Postman to test and explore the interface in the same way as you can use Postman to explore most of the RESTful APIs.

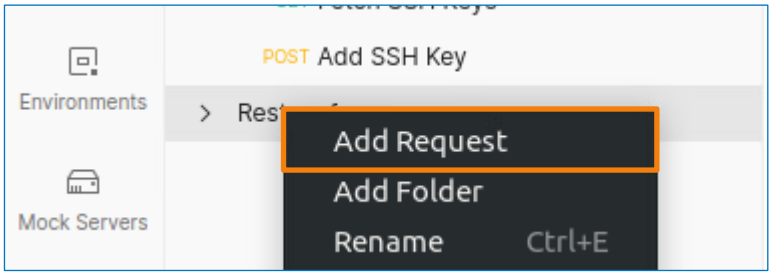
   If you closed the Postman application, open it again.



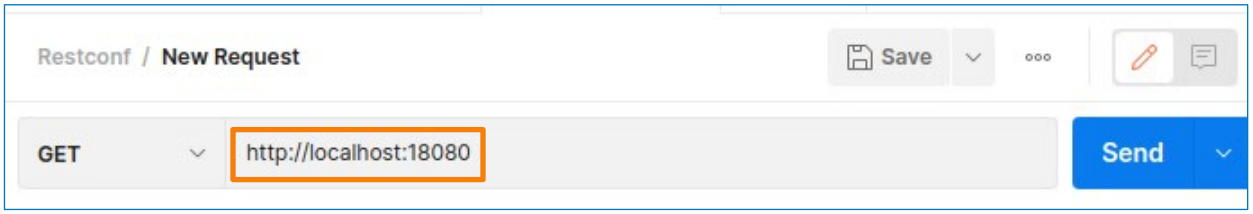
2. Create a new collection named **RESTCONF**. Click the **New** button and select **Collection** to create it.



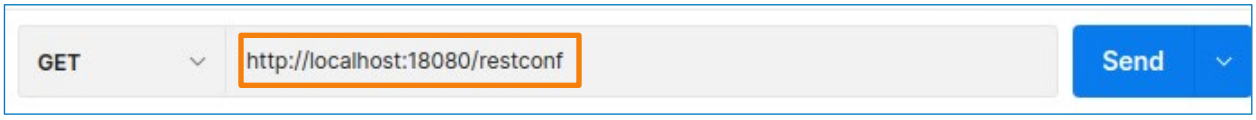
3. Right-click the created collection and select **Add request.**



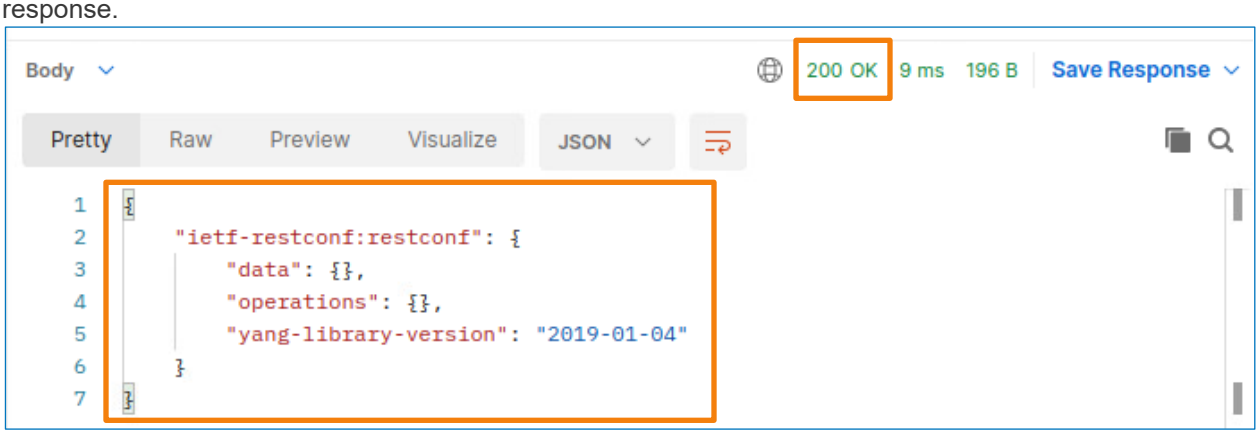
4. You will now construct a RESTCONF request that will fetch data from Clixon. First, you must find the web address of the device that should receive the request. Clixon is deployed as a container directly on the virtual machine that you are accessing so you can use **localhost** as the address. The port that you must use to reach the RESTCONF interface in this case is 18080. Enter the following URL in the request URL field: ***http://localhost:18080***.



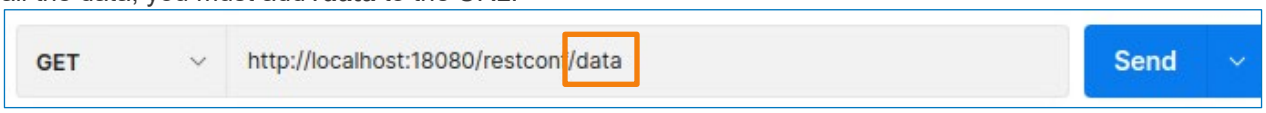
5. First you must check the connectivity and try to reach the RESTCONF interface on the device. Root URI where devices usually listen for RESTCONF requests is /restconf. Add **/restconf** to your URL.



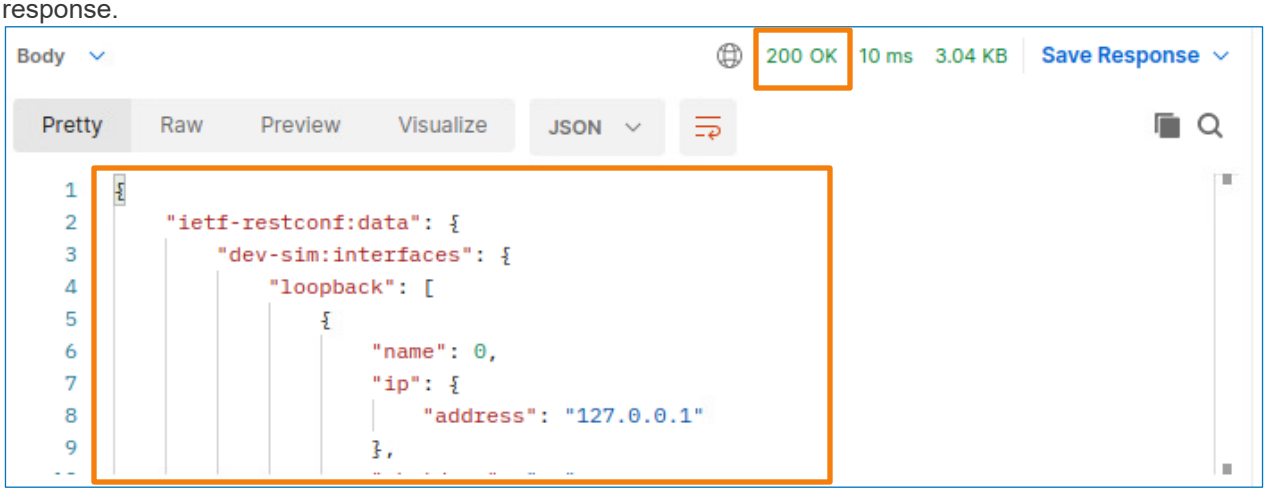
6. Send the request. The request should be successful, and you should receive the following response.



7. The response you received in step 6 describes the resources that you can access through RESTCONF. In this case, it tells you that all the data is located under the **/data** path, and custom operations available on this RESTCONF device are stored under the **/operations** path. To fetch all the data, you must add **/data** to the URL.



8. Send the request. You should receive all the data stored on the device in the body of the response.



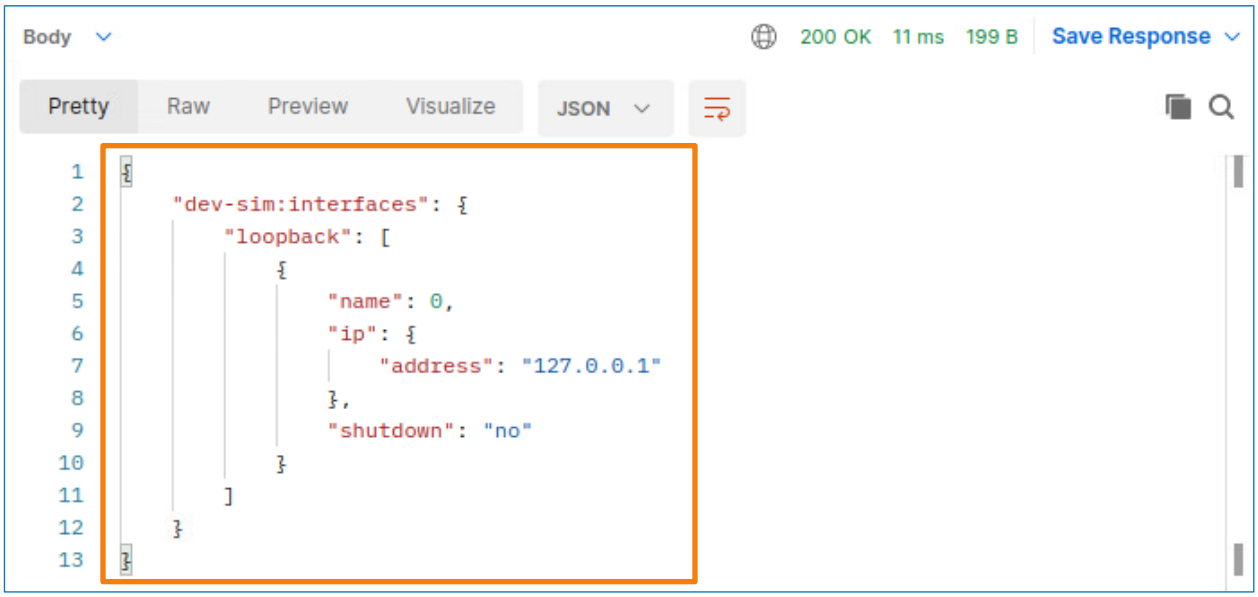
9. Usually, you only want to read certain configuration data and not the whole data store. To fetch only specific data, you must define the target resource with URI. Since RESTCONF devices usually do not have Swagger UI, you must rely on YANG models to explore the hierarchy and structure of the data. In this case, data follows the following YANG model.

```
module dev-sim {
  yang-version 1.1;
<... output omitted ...>
  container interfaces {
    list loopback {
      description
        "Loopback interface";
      key "name";
      leaf name {
        type uint32 {
          range "0..2147483647";
        }
      }
      uses interface-common;
    }
  }
  grouping interface-common {
    container ip {
      description
        "Set the IP address";
      leaf address {
```

```
        type inet:ipv4-address;
      }
    }
    leaf description {
      description
        "Interface specific description";
      type string {
        pattern ".*";
        length "0..200";
      }
    }
    leaf shutdown {
      default "no";
      type shutdown-option;
    }
  }
}
```

10. Observe the YANG model shown in the previous step. You can see that the name of the module is dev-sim and interface-related data is stored in the container named interfaces. To read only interface-related data, you must specify that the target resource is in the container named interfaces. To do that, add **module_name:resource_name** to the URL. In this case, you must add **dev-sim:interfaces** to the URL.

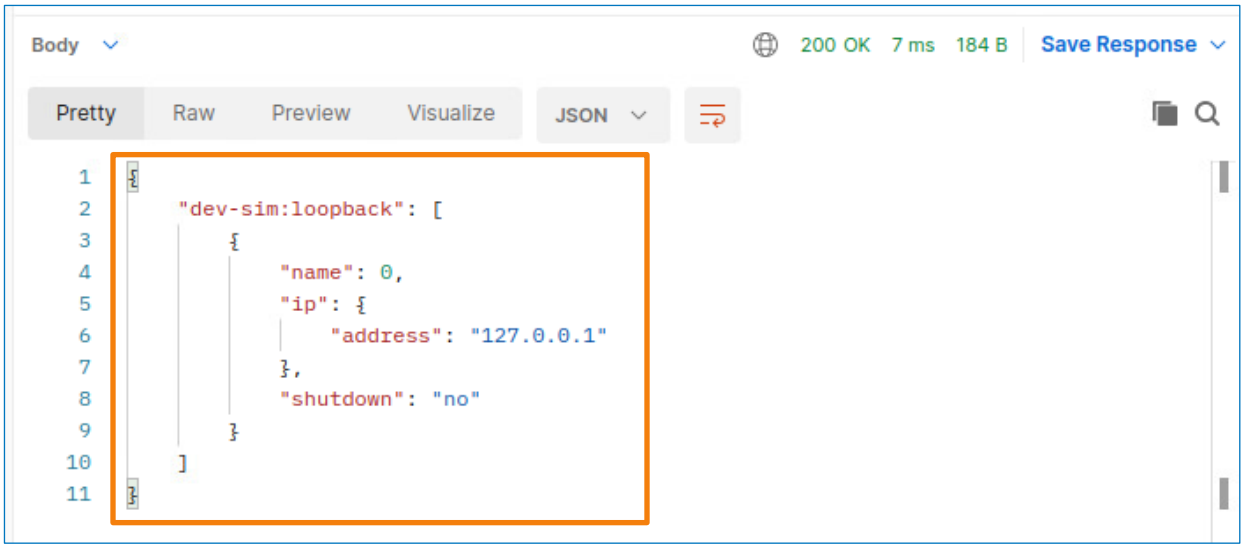
11. Send the request. You should receive only interface configuration data in the body of the response.



12. You can also request more specific data, for example only the IP address of the interface. To do that, you must define the path to the resource that contains the wanted data. In this case, you received a list of interfaces, and every interface has different parameters. As you can see from the YANG model the key parameter that is used to define a specific interface is the **name** parameter.

Therefore, you must add the name of the interface to the URL: **/loopback=0**.

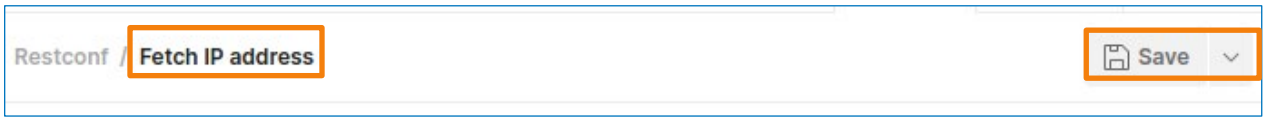
13. Send the request. The response should now include only configuration data for the interface **loopback 0**, even if there were other interfaces configured on the device.



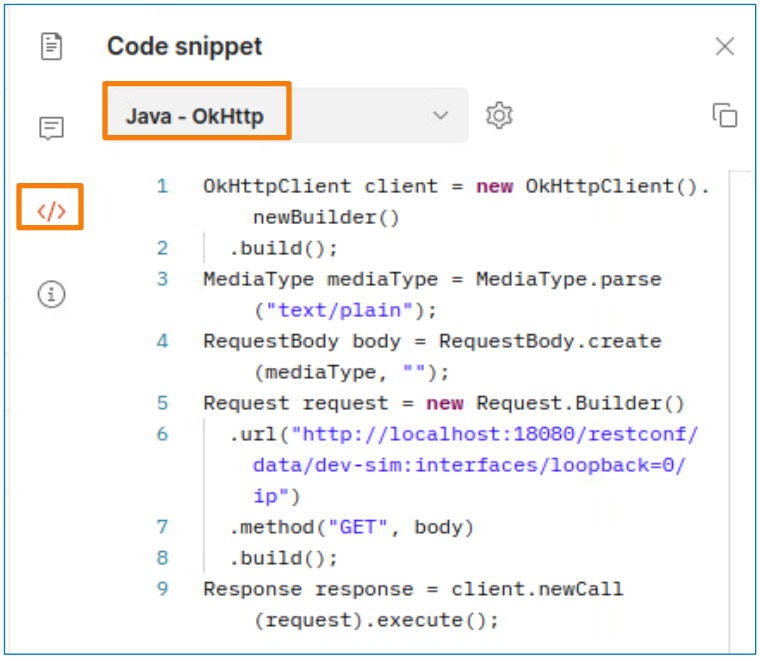
14. The resource contains 3 parameters (name, ip, and shutdown). To read only the IP address, you can specify that in the URL by adding the **/ip** path.

15. Send the request. You should now receive only the IP address of the loopback interface in the response.

16. Name the request **Fetch IP address** and save it.



17. Again, you could use the code-generator feature of Postman to generate code in different programming languages. To generate Java code that reads the IP address of the interface named loopback 0 on your device, click the **code** button and choose **Java – OkHttp** from the dropdown menu.



End of Lab

Lab 4: Using REST APIs with Java

Objectives

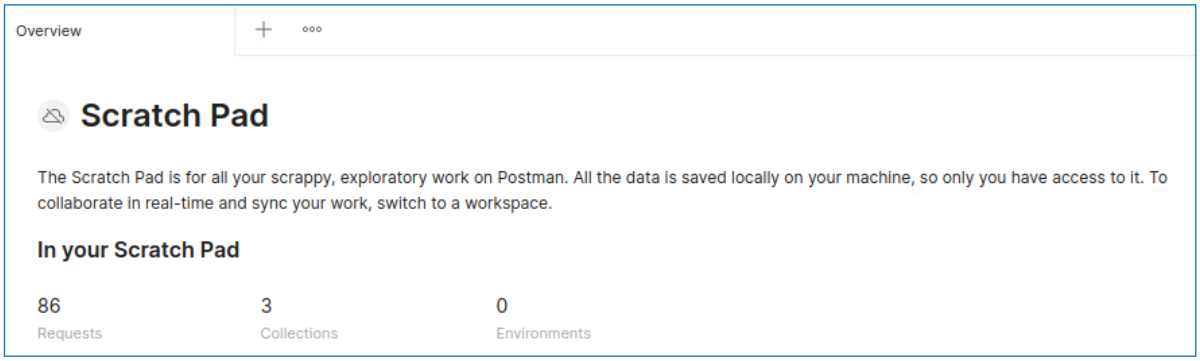
- Sending HTTP(s) requests to manage SSH keys on the lab server with Java
- Sending HTTP(s) requests to manage network devices with Java
- Creating code snippets with Postman

Task 1: Creating the Java project

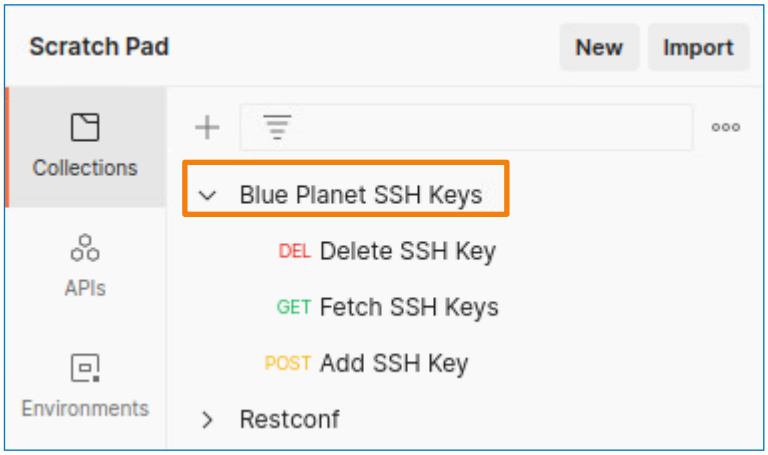
In this lab, you use Java to send the requests that you constructed in the previous lab. First, you develop Java code that can execute the requests for managing the SSH keys stored on the web server in your lab environment. Then you also develop the code that can fetch the IP address of a network device deployed in the lab environment.

First, you will prepare the Java project where you will develop the code that will manage the SSH keys stored on the Blue Planet server in your lab environment.

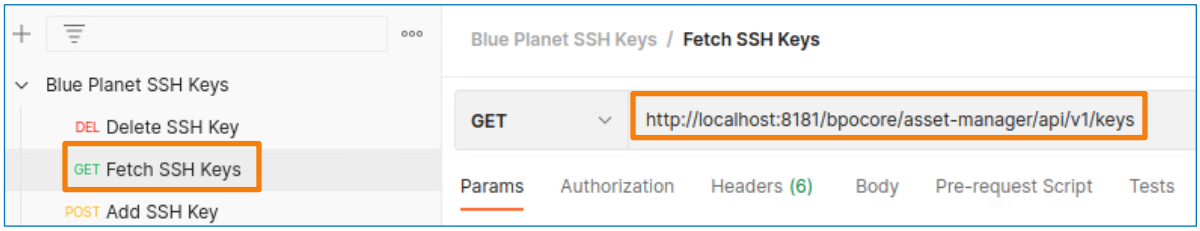
1. In this lab, you will use Postman as a tool that will help you develop the Java code. Open the Postman application.



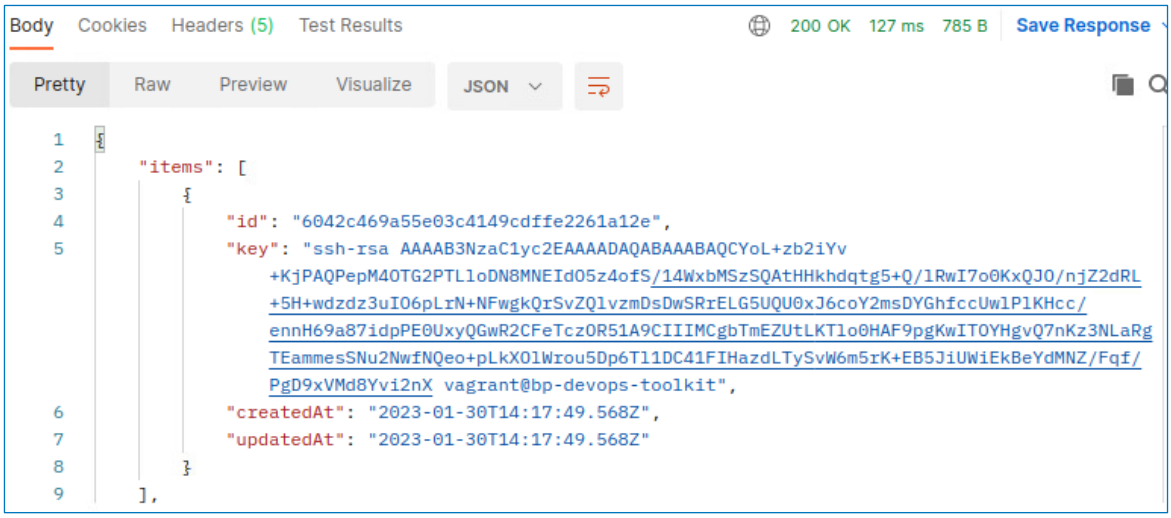
2. Expand the **Blue Planet SSH Keys** collection you created in the previous lab.



3. Select the **Fetch SSH Keys** request from the collection to open it.



4. Click **Send** and observe the response. The response includes a list of SSH keys stored on the server. You will now try to fetch this list with Java and print the existing key.



5. Click the **Code** button in Postman and select **Java – OkHttp** from the dropdown menu to generate a Java code that can be used to send the request.

   The code sets the content type and creates an empty body. Then it constructs the request by defining the URL and the method. Finally, it executes the request and stores it in the response variable.

6. Now you will create a new Java project where you will implement the autogenerated code. First, open a new terminal window and create a new directory named **lab4**.

```
[~]$ mkdir lab4
```

7. Navigate to the new directory and create a new Java project by executing the following commands:

NOTE: You can also generate the project using the Command Palette in VS Code.

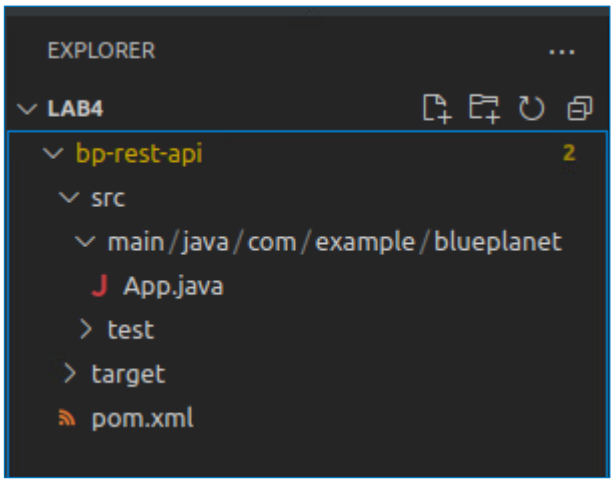
```
[~]$ cd lab4/
[~/lab4]$ mvn archetype:generate -DgroupId=com.example.blueplanet -
   DartifactId=bp-rest-api -DarchetypeArtifactId=maven-archetype-
   quickstart -DinteractiveMode=false


<... output omitted ...>
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] project created from Old (1.x) Archetype in dir:
   /home/vagrant/requests-java/bp-rest-api
[INFO] ------------------------------------------------------------------------
   -------
[INFO] BUILD SUCCESS
```

```
<... output omitted ...>
```

8. You now generated a new quick start Maven project with all the needed contents to start developing the code. Open the project using the **code .** command. A Visual Studio Code window with your project should open.

```
[~/lab4]$ code .
```

9. Expand the existing directories to explore the project contents.



10. Add the needed dependencies to the pom.xml file. In this case, you will be using the okHttp library to build and execute requests.

```
<... output omitted ...>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>com.squareup.okhttp3</groupId>
      <artifactId>okhttp</artifactId>
      <version>4.10.0</version>
    </dependency>
  </dependencies>
<... output omitted ...>
```

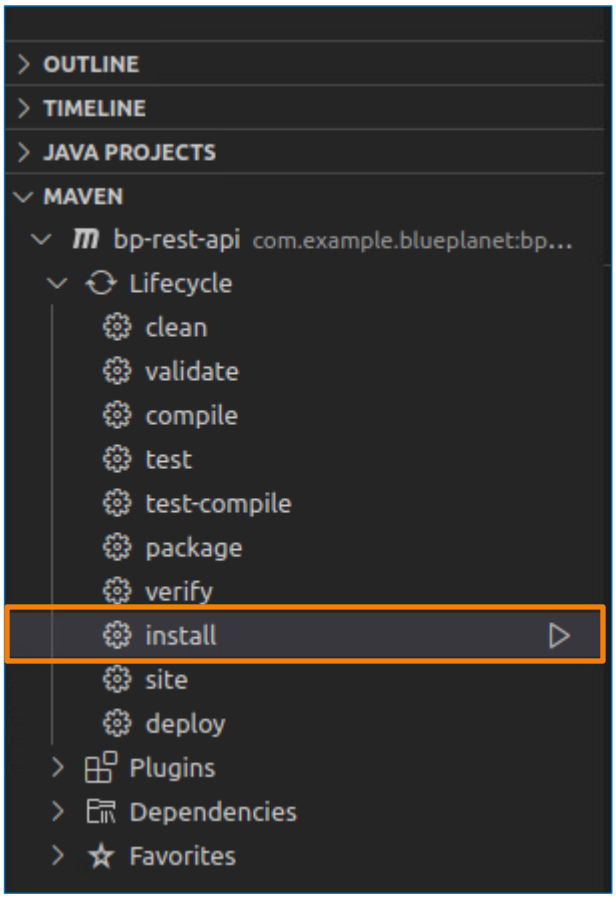
11. Add the following lines to the pom.xml file to change the default maven compiler and save the file.

```
<... output omitted ...>
```

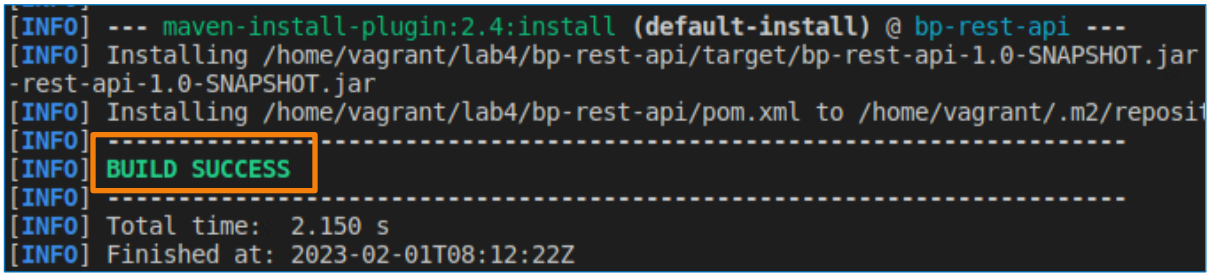
```
<dependency>
      <groupId>com.squareup.okhttp3</groupId>
      <artifactId>okhttp</artifactId>
      <version>4.10.0</version>
    </dependency>
  </dependencies>
  <properties>
     <maven.compiler.source>11</maven.compiler.source>
     <maven.compiler.target>11</maven.compiler.target>
  </properties>
</project>
```

12. Download and install the dependency by expanding the Maven menu and running the **install** command.

NOTE: You can also install the dependency by executing the following command from the terminal instead: mvn install -f "/home/vagrant/lab4/bp-rest-api/pom.xml"



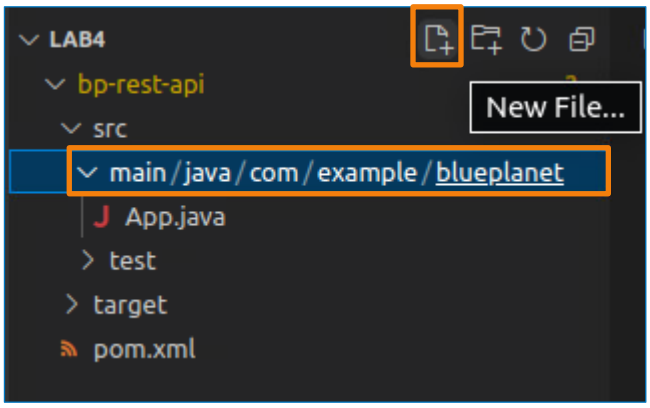
13. Make sure that the project is built successfully.



Task 2: Fetching the SSH Keys with Java

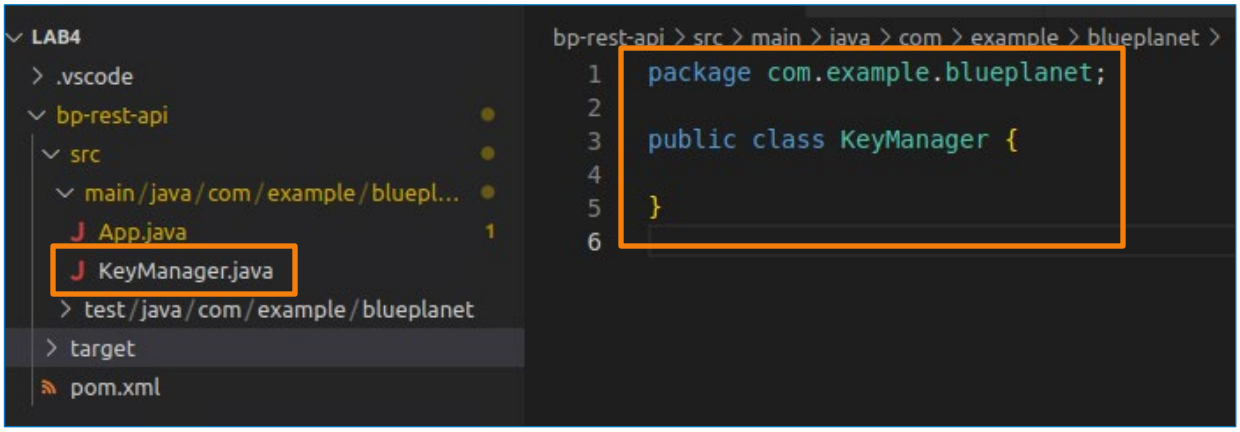
In this task, you create a new class with the FetchKeys method that can be used to fetch the list of all the stored SSH keys from the lab server.

1. Create a new Java file in the same directory as App.java by selecting the correct directory and clicking the **New file** button.



2. Name the file **KeyManager.java**, as you will add different methods to manage the SSH keys here.

NOTE: Make sure that you create the file in the correct directory of your project.s

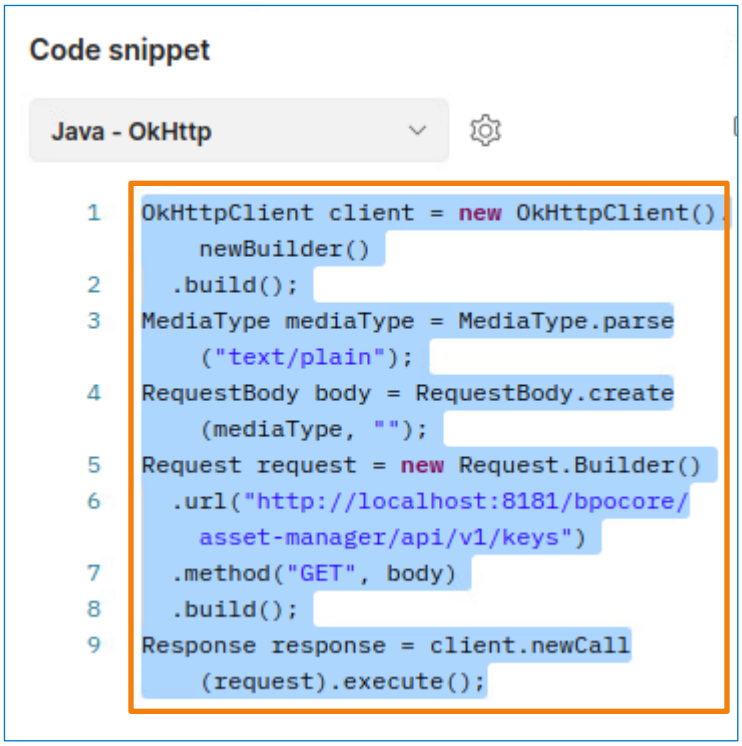


3. Create a new empty method named **FetchKeys**, where you will later add the code that executes the request.

```
package com.example.blueplanet;


public class KeyManager {
    public void FetchKeys() {


    }
}
```

4. Go back to Postman and copy the code snippet that you generated at the beginning of this task.

NOTE: If you closed the Code snippet, generate it again by following the first steps of this task.



5. Paste the code to the **fetchKeys.java** file that you just created. The method can now be used for sending a GET request to the /keys resource and storing the data that gets back in the response variable.

```
package com.example.blueplanet;


public class KeyManager {
    public void FetchKeys() {
        OkHttpClient client = new OkHttpClient().newBuilder()
       .build();
        MediaType mediaType = MediaType.parse("text/plain");
        RequestBody body = RequestBody.create(mediaType, "");
        Request request = new Request.Builder()
            .url("http://localhost:8181/bpocore/asset-
manager/api/v1/keys")
            .method("GET", body)
            .build();
        Response response = client.newCall(request).execute();
    }
```

```
}
```

6. Add the following import statements to the KeyManager.java file to use the OkHttp library in your code.

```
package com.example.blueplanet;


import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.Response;


public class KeyManager {
    public void FetchKeys() {
        OkHttpClient client = new OkHttpClient().newBuilder()
.build();
        MediaType mediaType = MediaType.parse("text/plain");
        RequestBody body = RequestBody.create(mediaType, "");
        Request request = new Request.Builder()
            .url("http://localhost:8181/bpocore/asset-
   manager/api/v1/keys")
            .method("GET", body)
            .build();
        Response response = client.newCall(request).execute();
    }
}
```

7. In this case, you are sending the GET request that should not contain the body. You will now change the autogenerated code from Postman to build a Get request without a body.
   First, remove the crossed-out lines.

```
package com.example.blueplanet;


import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.Response;


public class KeyManager {
    public void FetchKeys() {
        OkHttpClient client = new OkHttpClient().newBuilder()
.build();
```

```
        MediaType mediaType = MediaType.parse("text/plain");
        RequestBody body = RequestBody.create(mediaType, "");
        Request request = new Request.Builder()
            .url("http://localhost:8181/bpocore/asset-
   manager/api/v1/keys")
            .method("GET", body)
            .build();
        Response response = client.newCall(request).execute();
    }
}
```

8. Use the **get()** method instead of **method("GET", body)** since the latter requires body as a parameter and you want to send the request without it.

```
package com.example.blueplanet;


import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.Response;


public class KeyManager {
    public void FetchKeys() {
        OkHttpClient client = new OkHttpClient().newBuilder()
 .build();
        Request request = new Request.Builder()
            .url("http://localhost:8181/bpocore/assetmanager/api/v1/keys"
   )
            .method("GET", body)
            .get()
            .build();
        Response response = client.newCall(request).execute();
    }
}
```

9. Since you are now using the code generated by Postman as part of a method add the "throws IOException" declaration. Add the needed import statement as well.

NOTE: You can also use the **Quick Fix** button to handle the exception.

```
 package com.example.blueplanet;
```

```
import java.io.IOException;

import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.Response;

public class KeyManager {
    public void FetchKeys() throws IOException {
        OkHttpClient client = new OkHttpClient().newBuilder()
        .build();
        Request request = new Request.Builder()
            .url("http://localhost:8181/bpocore/assetmanager/api/v1/keys)
            .get()
            .build();
        Response response = client.newCall(request).execute();
    }
}
```

10. This code should now execute the request successfully and store the response in the “response” variable. Add another line to print the response.

```
package com.example.blueplanet;

import java.io.IOException;

import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.Response;

public class KeyManager {
    public void FetchKeys() throws IOException {
        OkHttpClient client = new OkHttpClient().newBuilder()
        .build();
        Request request = new Request.Builder()
            .url("http://localhost:8181/bpocore/asset-
   manager/api/v1/keys")
            .get()
            .build();
```

```
        Response response = client.newCall(request).execute();
        System.out.println(response.body().string());
    }
}
```

11. Your code should now look like the following code. Compare it to your code and make any changes needed for the codes to match.

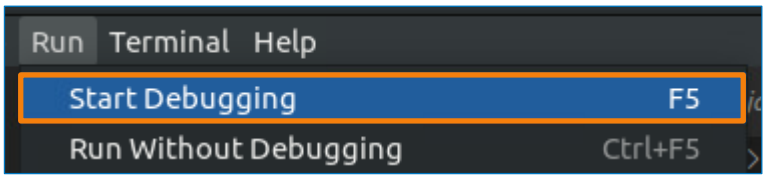
```
package com.example.blueplanet;


import java.io.IOException;


import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.Response;


public class KeyManager {
    public void FetchKeys() throws IOException {
        OkHttpClient client = new OkHttpClient().newBuilder()
        .build();
        Request request = new Request.Builder()
            .url("http://localhost:8181/bpocore/asset-
   manager/api/v1/keys")
            .get()
            .build();
        Response response = client.newCall(request).execute();
        System.out.println(response.body().string());
    }
}
```

12. Now open the **App.java** file and update the main method to create an instance of the KeyManager class and call its **fetchKeys** method.

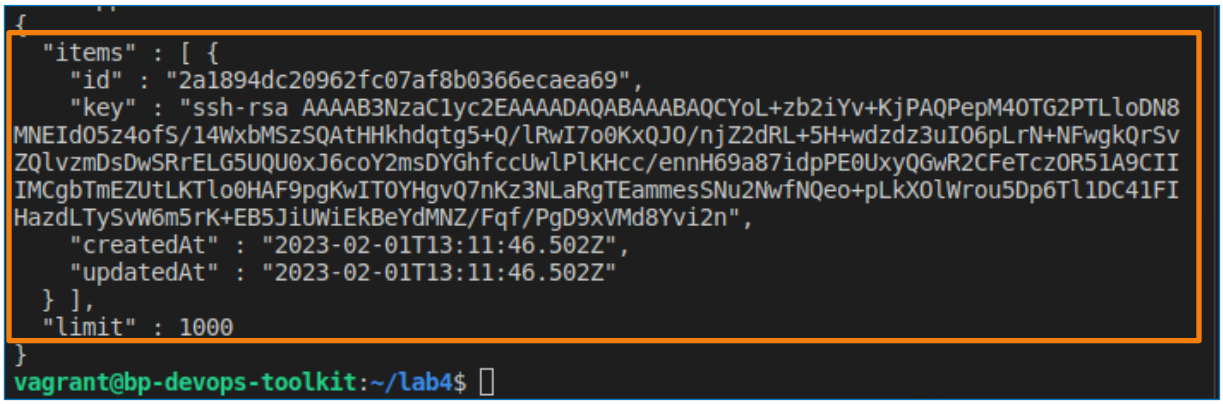
```
package com.example.blueplanet;


import java.io.IOException;


public class App
{
    public static void main( String[] args ) throws IOException
```

```
    {
        KeyManager km = new KeyManager();
        km.FetchKeys();
    }
}
```

13. Run the code from VS Code by clicking **Run > Start Debugging** or pressing the **F5** key,



14. The code should be executed successfully. Observe the response body printed in the terminal. The response should include a list of the SSH keys stored on the Blue Planet server in your environment.



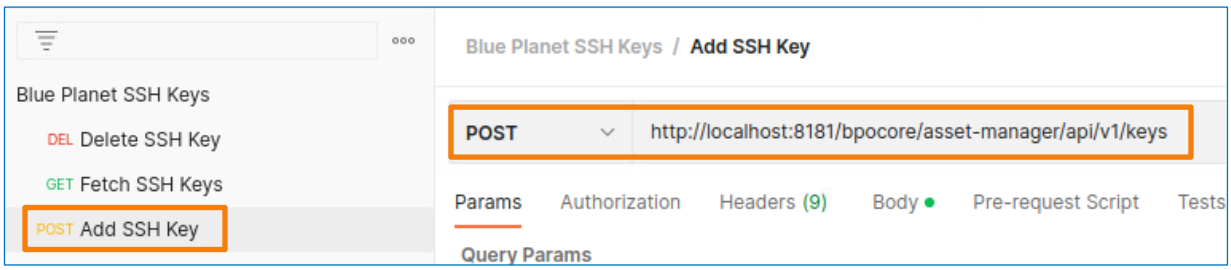
Task 3: Adding New SSH keys to the Server with Java

In this task, you will add another method that can be used to POST a key to the lab server, to the KeyManager class.

1. Create a new method named **PostKeys** in the KeyManager class.

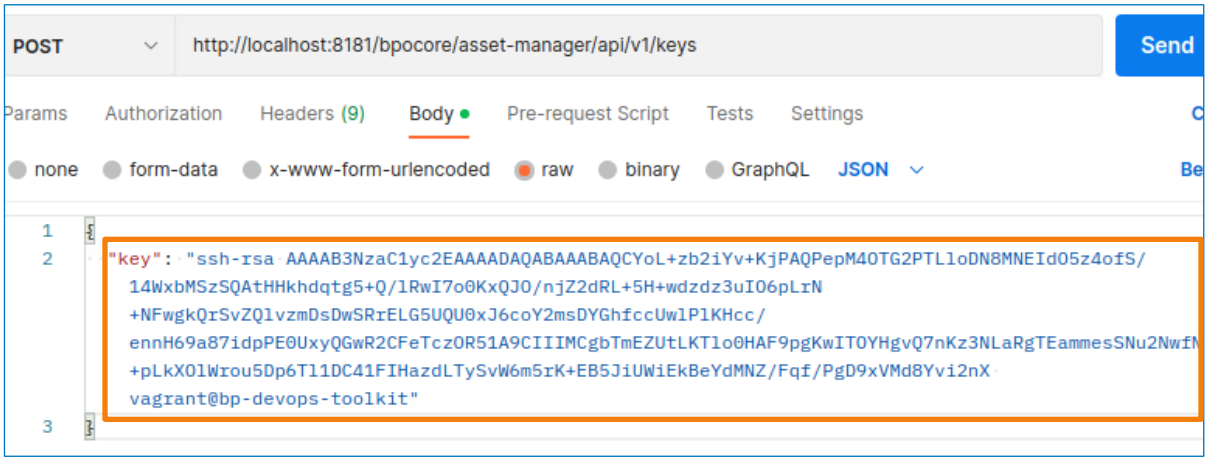
```
package com.example.blueplanet;

import java.io.IOException;

import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.Response;

public class KeyManager {
    public void FetchKeys() throws IOException {
        OkHttpClient client = new OkHttpClient().newBuilder()
        .build();
        Request request = new Request.Builder()
            .url("http://localhost:8181/bpocore/assetmanager/api/v1/keys"
    )
            .get()
            .build();
        Response response = client.newCall(request).execute();
        System.out.println(response.body().string());
    }

    public void PostKeys() throws IOException {

    }
}
```

2. Go back to the Postman application and open the request named **Add SSH Key**.

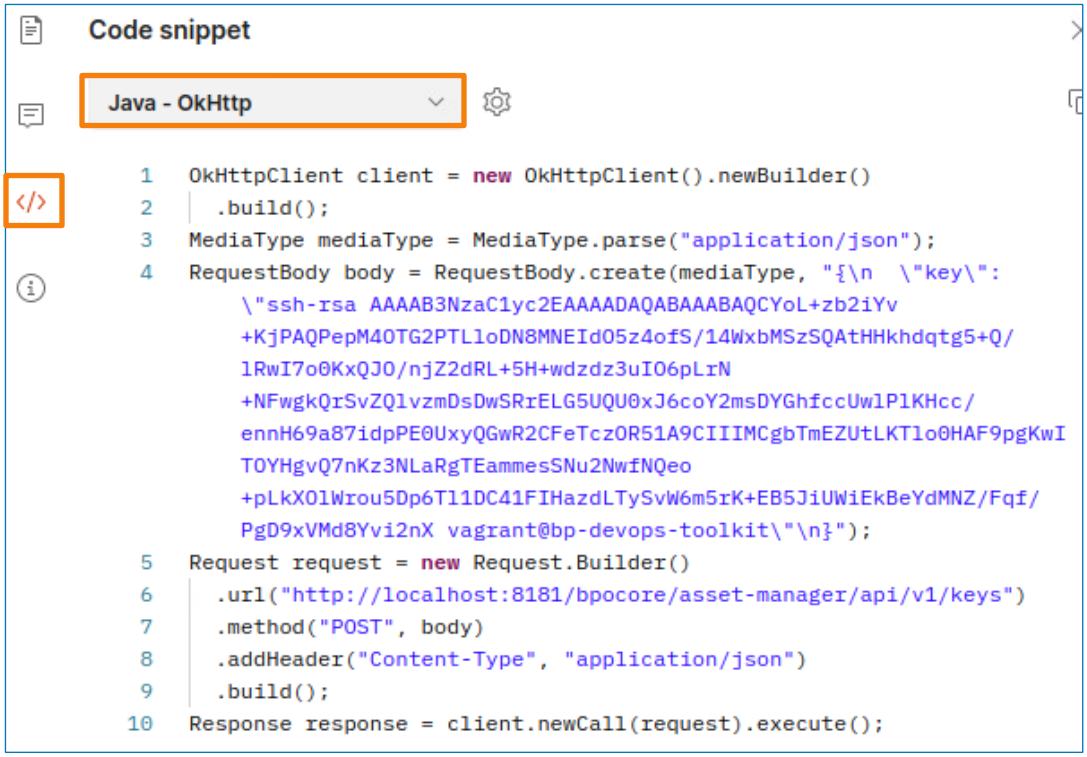


3. Make sure that the body of the request contains your public key.

NOTE: Your public key is probably different than the one shown in this example.



4. Generate the code snippet for this request by clicking the **Code** button and selecting **Java – OkHttp** from the dropdown menu.



5. Copy the code and paste it to the **PostKeys** method that you just created.

NOTE: Since this example is meant for demonstrational purposes, the public key is hardcoded directly in the code. In a production environment, you should find a better solution, for example, read the key from its file.

```
package com.example.blueplanet;
```

```
import java.io.IOException;


import okhttp3.MediaType;
import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.RequestBody;
import okhttp3.Response;


public class KeyManager {
    public void FetchKeys() throws IOException {
        OkHttpClient client = new OkHttpClient().newBuilder()
        .build();
        Request request = new Request.Builder()
            .url("http://localhost:8181/bpocore/asset-
   manager/api/v1/keys")
            .get()
            .build();
        Response response = client.newCall(request).execute();
        System.out.println(response.body().string());
    }


    public void PostKeys() throws IOException {
        OkHttpClient client = new OkHttpClient().newBuilder()
        .build();
      MediaType mediaType = MediaType.parse("application/json");
       RequestBody body = RequestBody.create(mediaType, "{\n  \"key\":
   \"ssh-rsa
   AAAAB3NzaC1yc2EAAAADAQABAAABAQCYoL+zb2iYv+KjPAQPepM4OTG2PTLloDN8MNEI
   dO5z4ofS/14WxbMSzSQAtHHkhdqtg5+Q/lRwI7o0KxQJO/njZ2dRL+5H+wdzdz3uIO6p
   LrN+NFwgkQrSvZQlvzmDsDwSRrELG5UQU0xJ6coY2msDYGhfccUwlPlKHcc/ennH69a8
   7idpPE0UxyQGwR2CFeTczOR51A9CIIIMCgbTmEZUtLKTlo0HAF9pgKwITOYHgvQ7nKz3
   NLaRgTEammesSNu2NwfNQeo+pLkXOlWrou5Dp6Tl1DC41FIHazdLTySvW6m5rK+EB5Ji
   UWiEkBeYdMNZ/Fqf/PgD9xVMd8Yvi2nX vagrant@bp-devops-toolkit\"\n}");
      Request request = new Request.Builder()
        .url("http://localhost:8181/bpocore/asset-manager/api/v1/keys")
        .method("POST", body)
        .addHeader("Content-Type", "application/json")
        .build();
```

```
        Response response = client.newCall(request).execute();
    }
}
```

6. Add the following import statements.

```
package com.example.blueplanet;

import java.io.IOException;

import okhttp3.MediaType;
import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.RequestBody;
import okhttp3.Response;

public class KeyManager {
<... output omitted ...>
```

7. Add the following line to print the response body and save the file.

```
<... output omitted ...>
    public void PostKeys() throws IOException {
        OkHttpClient client = new OkHttpClient().newBuilder()
        .build();
      MediaType mediaType = MediaType.parse("application/json");
      RequestBody body = RequestBody.create(mediaType, "{\n  \"key\":
   \"ssh-rsa
   AAAAB3NzaC1yc2EAAAADAQABAAABAQCYoL+zb2iYv+KjPAQPepM4OTG2PTLloDN8MNEI
   dO5z4ofS/14WxbMSzSQAtHHkhdqtg5+Q/lRwI7o0KxQJO/njZ2dRL+5H+wdzdz3uIO6p
   LrN+NFwgkQrSvZQlvzmDsDwSRrELG5UQU0xJ6coY2msDYGhfccUwlPlKHcc/ennH69a8
   7idpPE0UxyQGwR2CFeTczOR51A9CIIIMCgbTmEZUtLKTlo0HAF9pgKwITOYHgvQ7nKz3
   NLaRgTEammesSNu2NwfNQeo+pLkXOlWrou5Dp6Tl1DC41FIHazdLTySvW6m5rK+EB5Ji
   UWiEkBeYdMNZ/Fqf/PgD9xVMd8Yvi2nX vagrant@bp-devops-toolkit\"\n}");
      Request request = new Request.Builder()
        .url("http://localhost:8181/bpocore/asset-manager/api/v1/keys")
        .method("POST", body)
        .addHeader("Content-Type", "application/json")
        .build();
      Response response = client.newCall(request).execute();
      System.out.println(response.body().string());
```

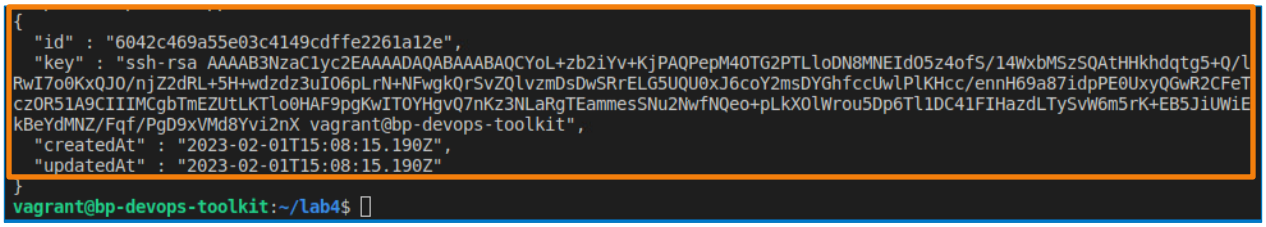
```
    }
}
```

8. Open the **App.java** file and update the main method to call the **PostKeys** instead of the **FetchKeys** method.

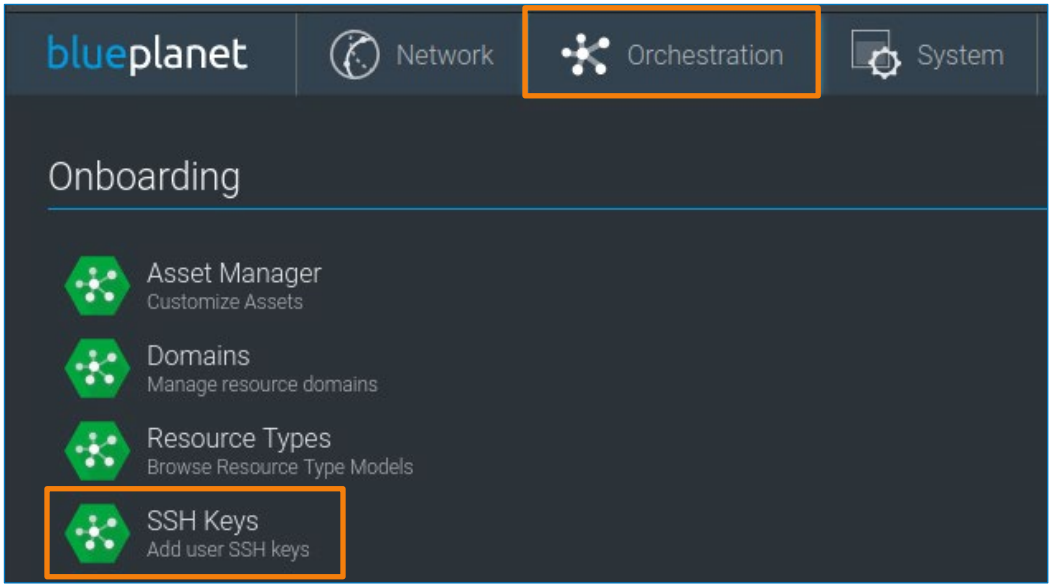
NOTE: For ease of use, you are just changing the name of the method that you are calling. In a real scenario, you would implement some additional logic to decide when to call which method.

```
package com.example.blueplanet;


import java.io.IOException;


public class App
{
    public static void main( String[] args ) throws IOException
    {
        KeyManager km = new KeyManager();
        ~~km.FetchKeys();~~
        km.PostKeys();
    }
}
```

9. Click **Run > Start Debugging** to execute the code. The request should execute successfully, and you should see the created key in the response.



10. Go to the web UI of the server to make sure that your key was added successfully. Open the web UI by clicking the **Blue Planet web UI** bookmark in your browser and open the **SSH Keys** menu.

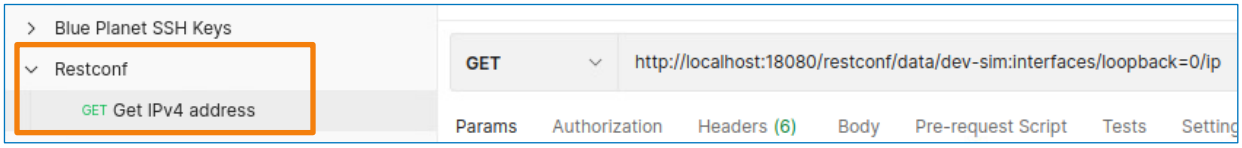


11. Find the key that you just added to the list of keys.

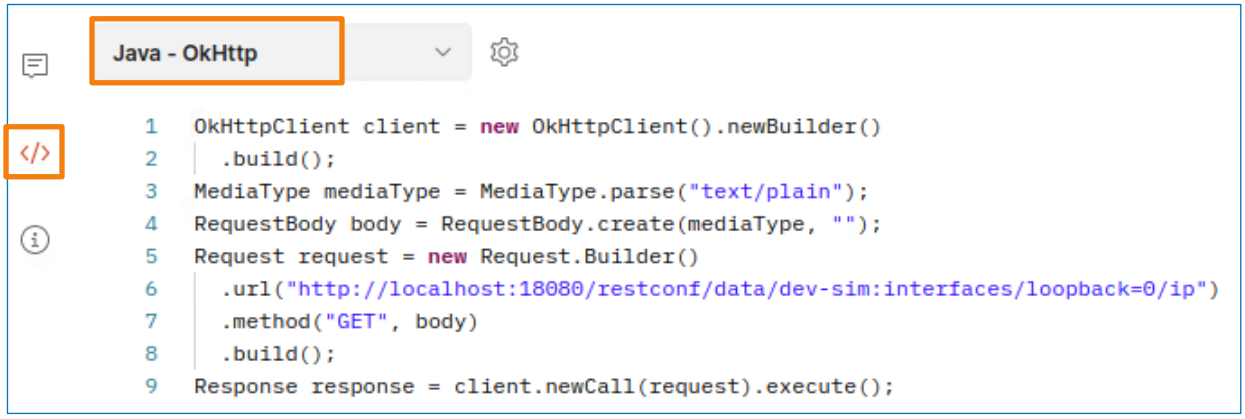
Task 4: Manage RESTCONF device with Java

Similarly, you can use Postman to create code snippets for sending the HTTP requests for managing a RESTCONF device, that you created in the previous lab. In this task, you will develop a new method that will read the IPv4 address on the RESTCONF device.

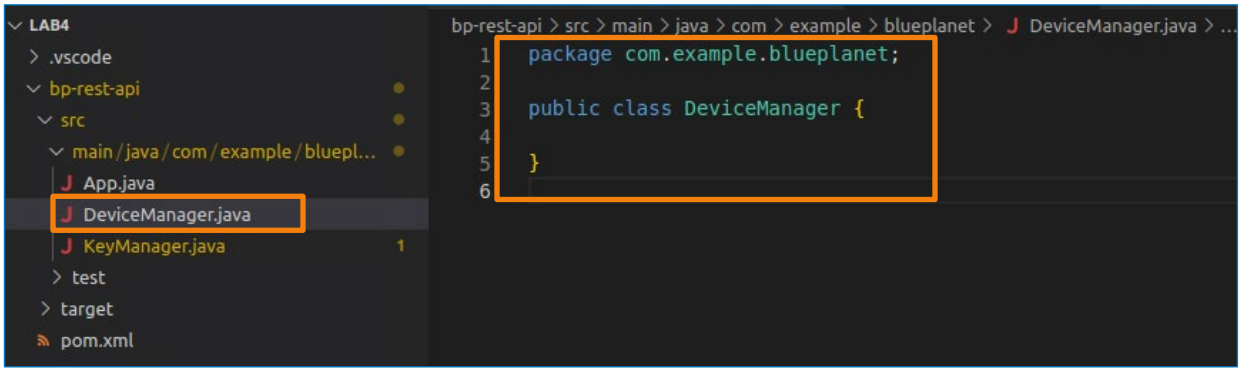
1. Go back to Postman and open the **GET** request that you saved to the **Restconf** collection in the previous lab.



2. Click the **Code** button and choose **Java – OkHttp** from the dropdown menu again.



3. Since you will use the same library to send the request you do not need to install any additional dependencies. Go back to VS Code and create a new Java class named **DeviceManager** in the same directory.



4. Create a new empty method named **GetConfig()**.

```
package com.example.blueplanet;


public class DeviceManager {


    public void GetConfig()  {
```

```
    }


}
```

5. Copy the code from Postman and paste it to the **GetConfig** method.

```
package com.example.blueplanet;


public class DeviceManager {


    public void GetConfig()  {
      OkHttpClient client = new OkHttpClient().newBuilder()
        .build();
      MediaType mediaType = MediaType.parse("text/plain");
      RequestBody body = RequestBody.create(mediaType, "");
      Request request = new Request.Builder()
        .url("http://localhost:18080/restconf/data/dev-
   sim:interfaces/loopback=0/ip")
        .method("GET", body)
        .build();
      Response response = client.newCall(request).execute();
    }


}
```

6. Modify the code to generate a GET request that does not include the body by deleting the following crossed-out lines.

```
package com.example.blueplanet;


public class DeviceManager {


    public void GetConfig()  {
      OkHttpClient client = new OkHttpClient().newBuilder()
        .build();
      MediaType mediaType = MediaType.parse("text/plain");
      RequestBody body = RequestBody.create(mediaType, "");
      Request request = new Request.Builder()
        .url("http://localhost:18080/restconf/data/dev-
   sim:interfaces/loopback=0/ip")
```

```
        .method("GET", body)
        .build();
      Response response = client.newCall(request).execute();
    }

}
```

7. Use the **.get()** method instead of the .method() method.

```
package com.example.blueplanet;

public class DeviceManager {

    public void GetConfig() {
      OkHttpClient client = new OkHttpClient().newBuilder()
        .build();
      Request request = new Request.Builder()
        .url("http://localhost:18080/restconf/data/dev-
   sim:interfaces/loopback=0/ip")
        .method("GET", body)
        .get()
        .build();
      Response response = client.newCall(request).execute();
    }

}
```

8. Import all the required dependencies.

```
package com.example.blueplanet;

import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.Response;

public class DeviceManager {

    public void GetConfig() {
      OkHttpClient client = new OkHttpClient().newBuilder()
```

```
        .build();
      Request request = new Request.Builder()
        .url("http://localhost:18080/restconf/data/dev-
   sim:interfaces/loopback=0/ip")
        .get()
        .build();
      Response response = client.newCall(request).execute();
    }

}
```

9. Again, use the **throws IOException** declaration and import **IOException**.

NOTE: You can also do that by using the offered “Quick Fix”

```
package com.example.blueplanet;

import java.io.IOException;

import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.Response;

public class DeviceManager {

    public void GetConfig() throws IOException  {
      OkHttpClient client = new OkHttpClient().newBuilder()
        .build();
      Request request = new Request.Builder()
        .url("http://localhost:18080/restconf/data/dev-
   sim:interfaces/loopback=0/ip")
        .get()
        .build();
      Response response = client.newCall(request).execute();

    }

}
```

10. The response data is stored in the “response” variable. Add the following line to print the body of the response.

```
package com.example.blueplanet;

import java.io.IOException;

import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.Response;

public class DeviceManager {

    public void GetConfig() throws IOException  {
      OkHttpClient client = new OkHttpClient().newBuilder()
        .build();
      Request request = new Request.Builder()
        .url("http://localhost:18080/restconf/data/dev-
   sim:interfaces/loopback=0/ip")
        .get()
        .build();
      Response response = client.newCall(request).execute();
      System.out.println(response.body().string());
    }

}
```

11. The method that executes the request and prints the IP address is now ready. Open the **App.java** file, delete the code that calls the FetchKeys method, and add the following code to call the **GetConfig()** method.

```
package com.example.blueplanet;

import java.io.IOException;

public class App
{
    public static void main( String[] args ) throws IOException
    {
```

```
        KeyManager km = new KeyManager();
        km.FetchKeys();
        DeviceManager dm = new DeviceManager();
        dm.GetConfig();
    }
}
```

12. Click **Run > Start Debugging** to execute the code. The request should be successful, and you should see the IP address of the loopback0 interface in the response.

Lab End

Lab 6: Camunda Introduction

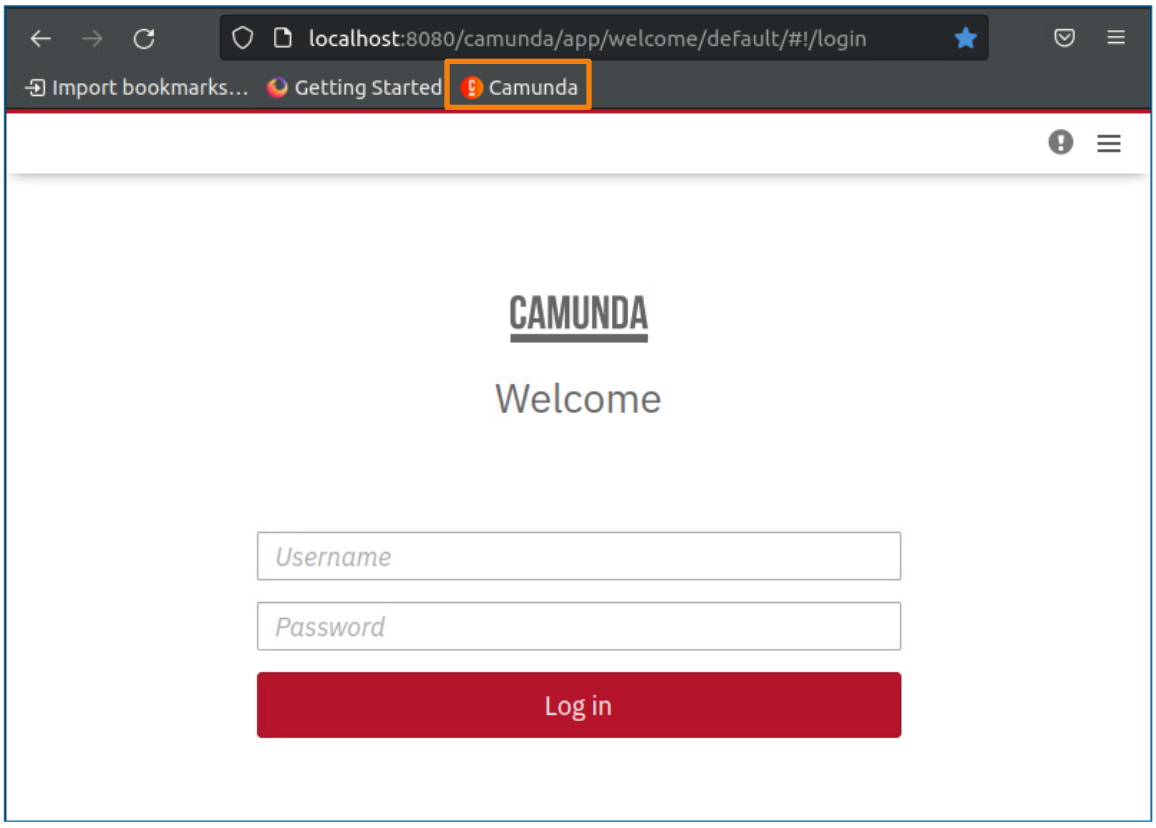
Objectives

- Exploring the Camunda platform
- Creating a simple Camunda process
- Deploying the process to Camunda

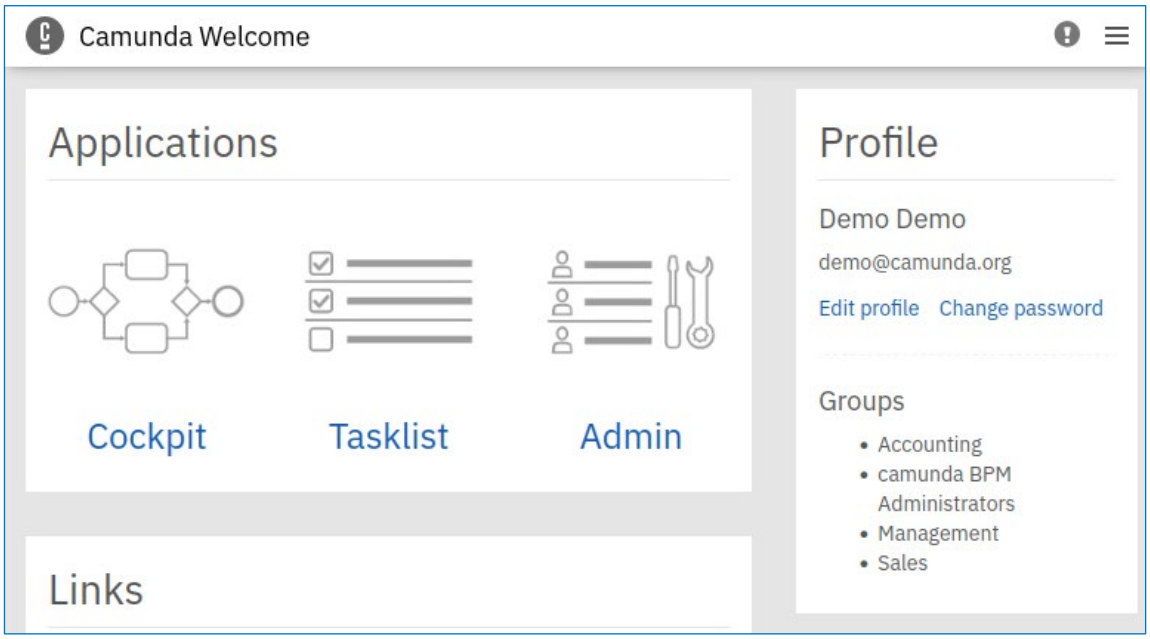
Task 1: Exploring the Camunda Platform

In this task, you explore the Camunda platform that is deployed in your Lab environment and learn about different components of the platform.

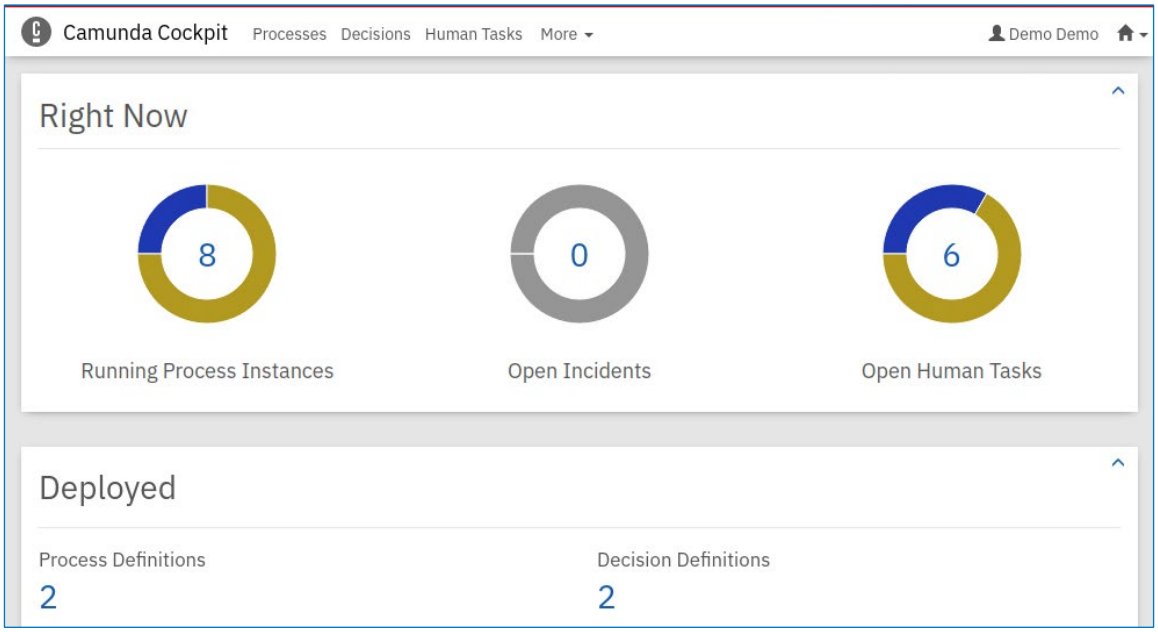
1. In this case, Camunda is deployed in a form of a container running on your virtual machine. To access the user interface, navigate to *http://localhost:8080/camunda/* or click on the **Camunda** bookmark.



2. Login with the following credentials to access the Camunda homepage:
   - Username: **demo**
   - Password: **demo**

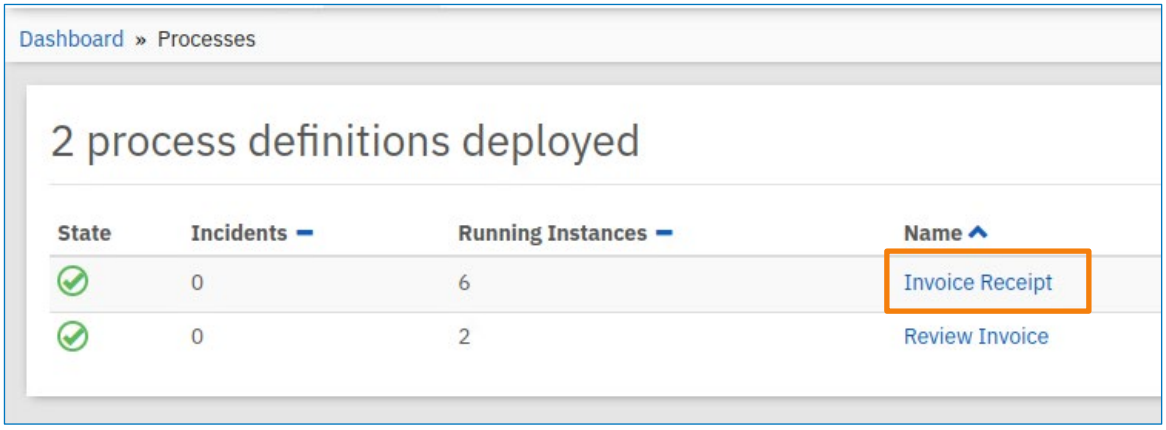


3. Open the **Cockpit** application by selecting it from the application menu. The Cockpit is a user application that provides a graphical interface to manage and monitor deployed processes. A dashboard with a summary of current processes will open.

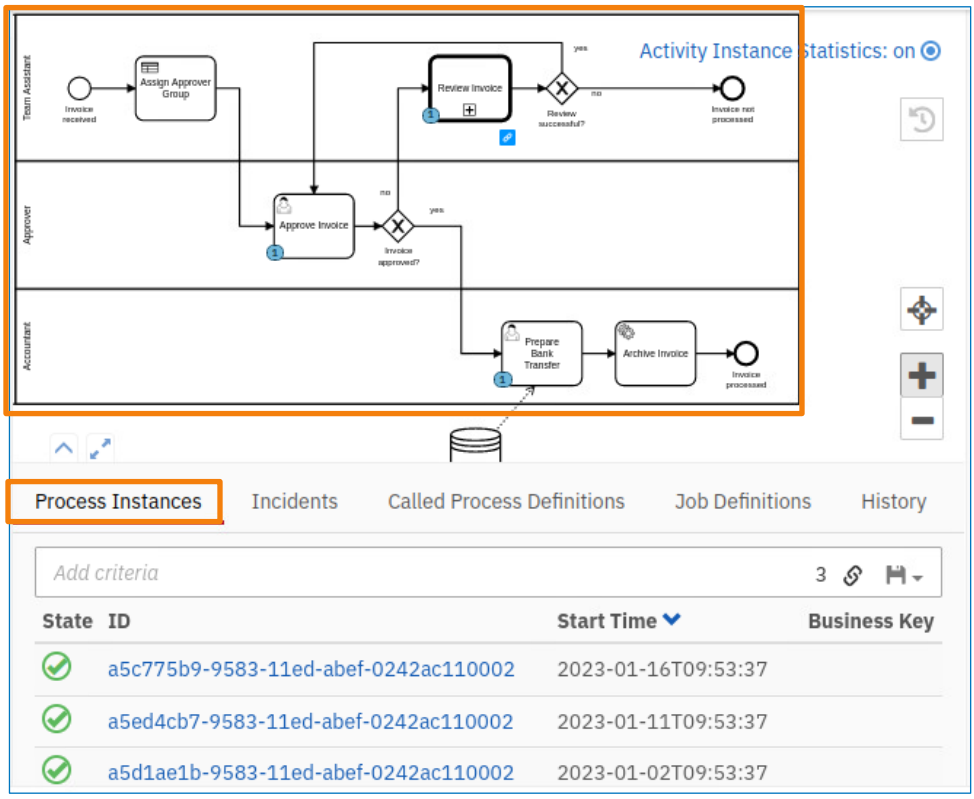


4. Click on the **Processes** tab in the menu bar, to open a list of currently deployed processes.

5. Some demo processes are already deployed on your instance of Camunda. Click on the name of the first process to find more information about that process.

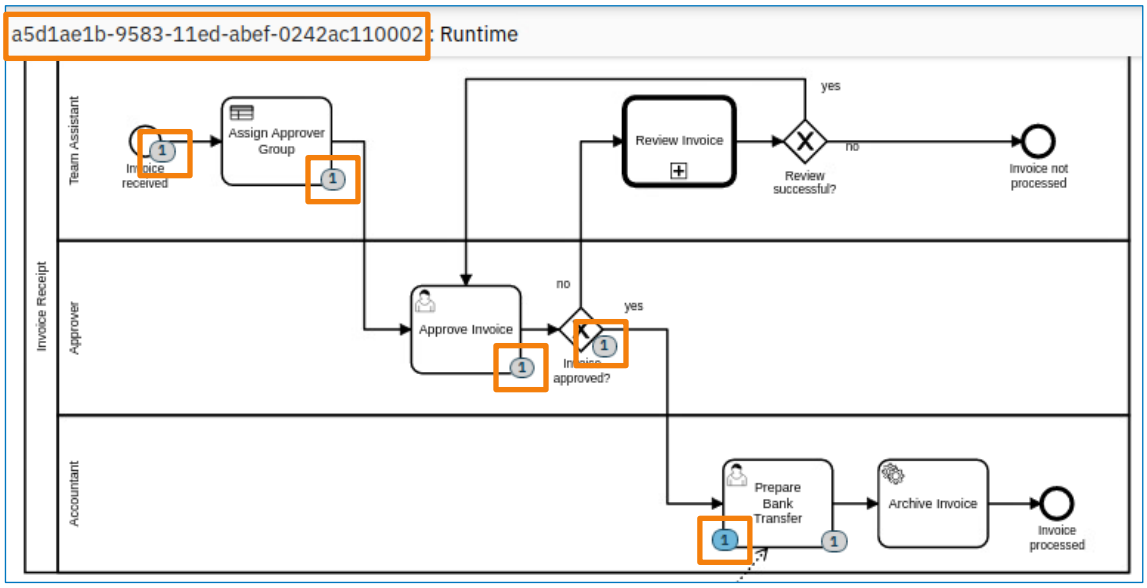


6. The page that opens shows you how many instances of this process are currently running. At the top of the page, there is also a Business Process Modeling Notation (BPMN) diagram of the process.

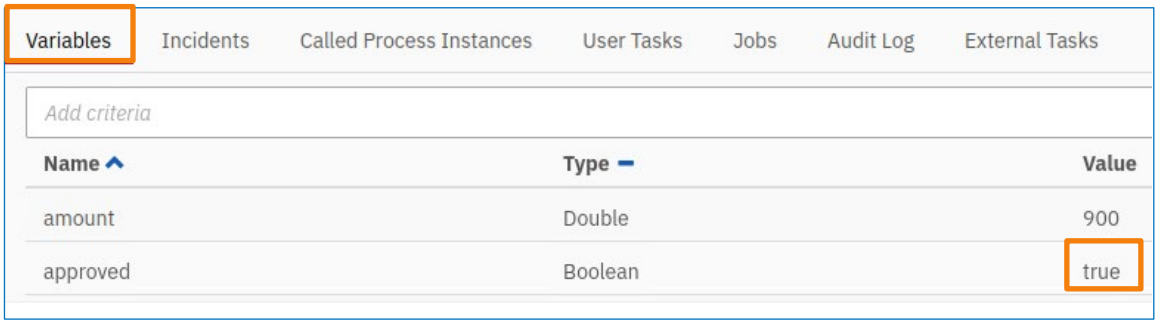


7. By clicking on one of the process instances you can check at which point of the process the selected instance currently is. This means that you can see which tasks were already performed, which decisions have been made, and what is the next step of the process.

NOTE: If you click on some other instance, it will probably be at a different point in the process flow.

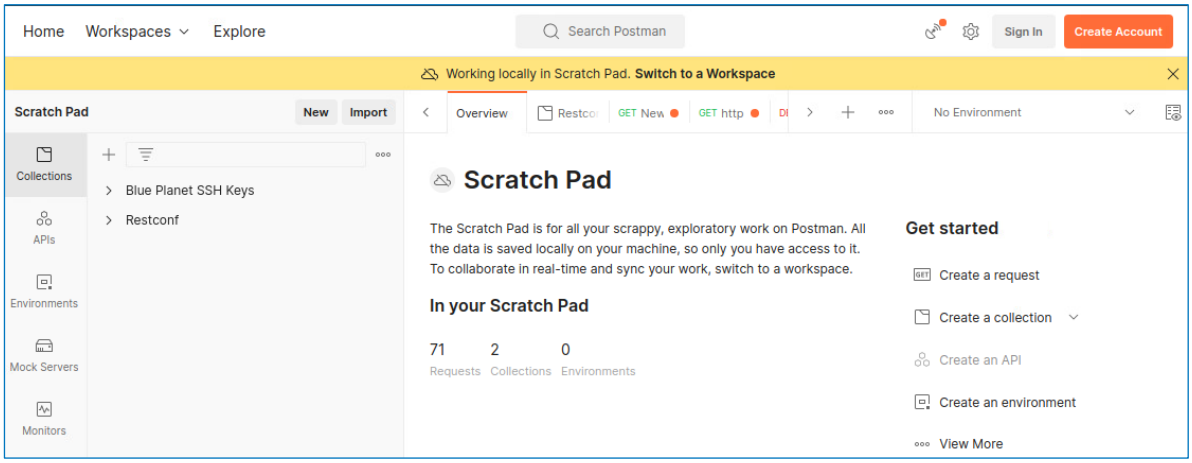


8. By opening the variables menu you can also check which variables were used or calculated in the process. For example, the value of the **approved** variable is **true** which is why the flow continued from the **invoice approved?** step to the **Prepare Bank Transfer** step. You can also change the values of the variables from Cockpit to test what happens.

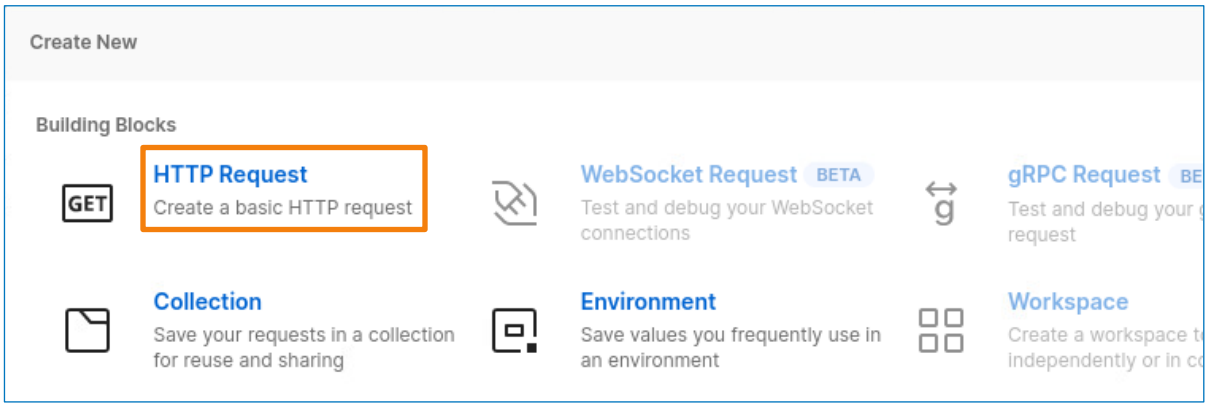


9. As mentioned, the Cockpit application provides features to monitor and manage processes that are deployed on the Camunda engine. The application communicates with the engine through a REST API and all of the functions that can be performed through the Cockpit web UI can also be performed through the API directly. This means that all the actions can be automated, for example, you can develop a program that constantly checks if there are any **Incidents** instead of manually checking in the UI.

   To test the communication with the Camunda engine REST API, open the **Postman** application.

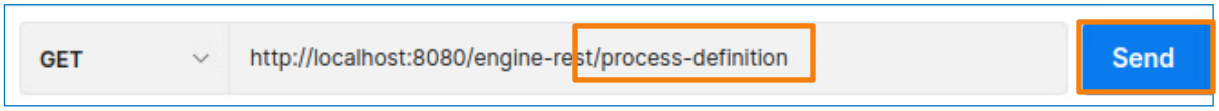


10. Click the **New** button and select **HTTP request** to create a new request.

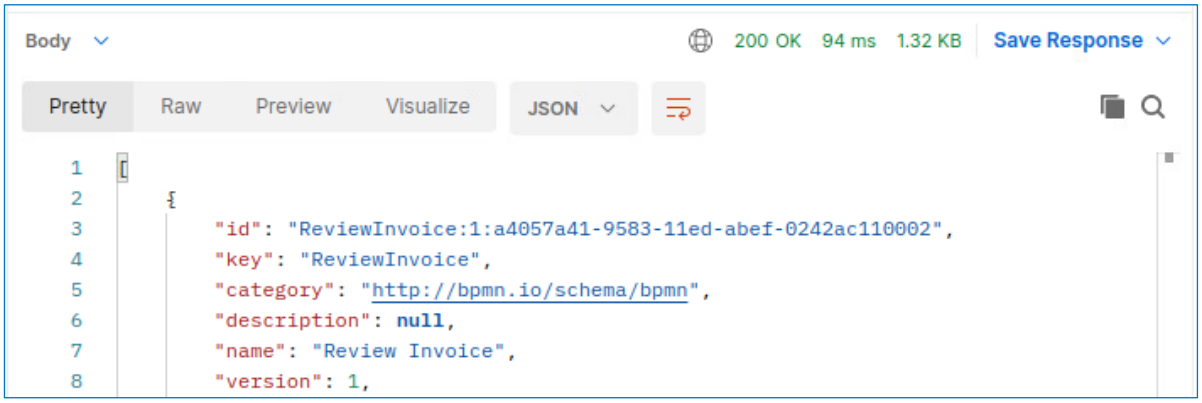


11. Start constructing the request by entering the URL in the **Enter request URL** field. The base URI in this case is **/engine-rest**. Since the engine is running directly on your VM and the requests received on port **8080** get to Camunda, the URL that you must enter is ***http://localhost:8080/engine-rest***.

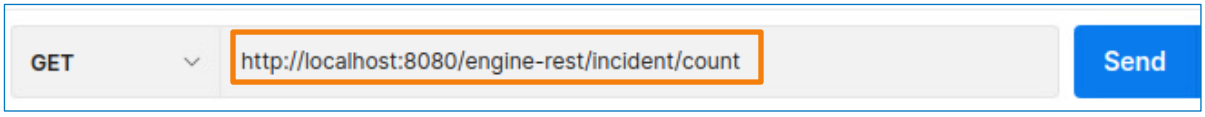
12. By exploring the Camunda REST API documentation, you can find out all the existing resources and possible operations. For example, you can fetch the list of existing process definitions by sending a get request on **/process-definition**. Add the path to the URL that you constructed in the previous request and click the **Send** button.



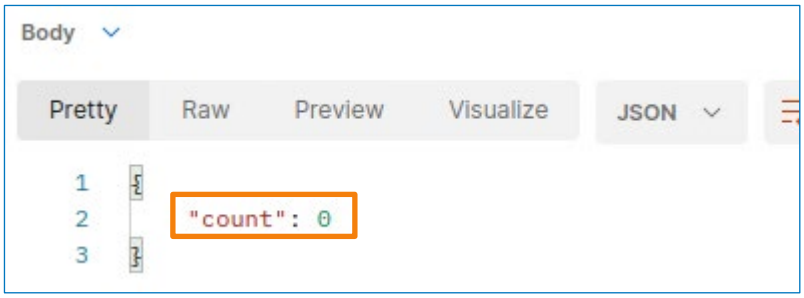
13. The response body should include a list of all process definitions.



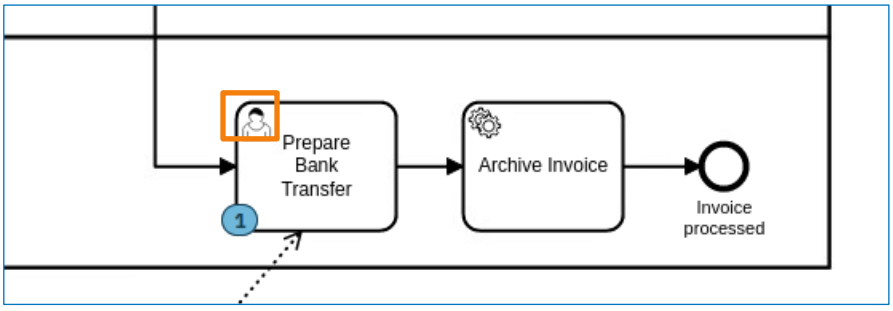
14. As mentioned, you can also check if there are any active incidents. To do that, change the URL to ***http://localhost:8080/engine-rest/incident/count*** and send the request.



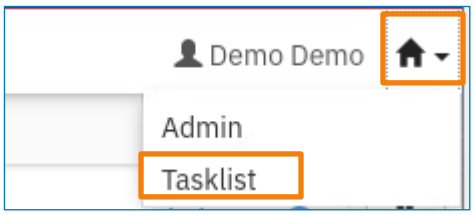
15. The response body shows the number of incidents. Currently, there should be no active incidents.



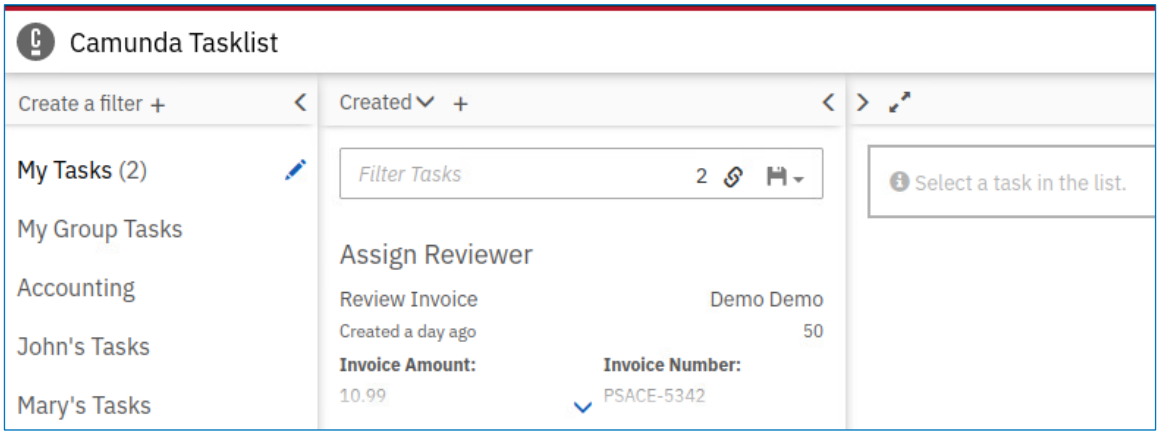
16. Return to the Cockpit application and open any of the present process definitions (choose **Processes** from the menu bar and choose one of the listed processes, for example, **Invoice Receipt**). A graphical representation of the process should open. Observe how some tasks are marked with a user symbol. Those tasks are user tasks and require some user action to be performed.



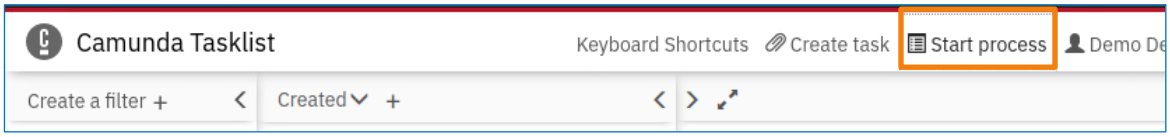
17. The application responsible for managing user tasks in Camunda is **Tasklist**. To access the tasklist application, click the **house-shaped button** in the top-right corner and choose **tasklist**.



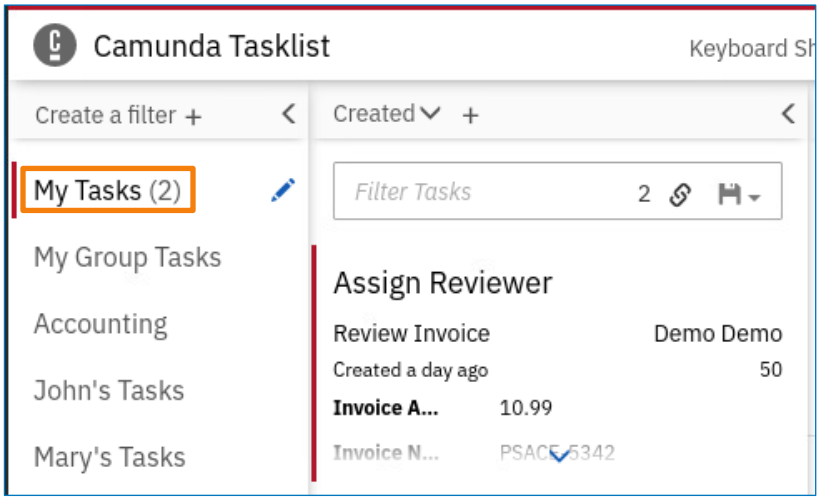
18. The Tasklist application web UI should open. The Tasklist application provides functionalities to manage the tasks, for example, tasks can be assigned and completed. Tasklist also enables users to start the deployed processes.



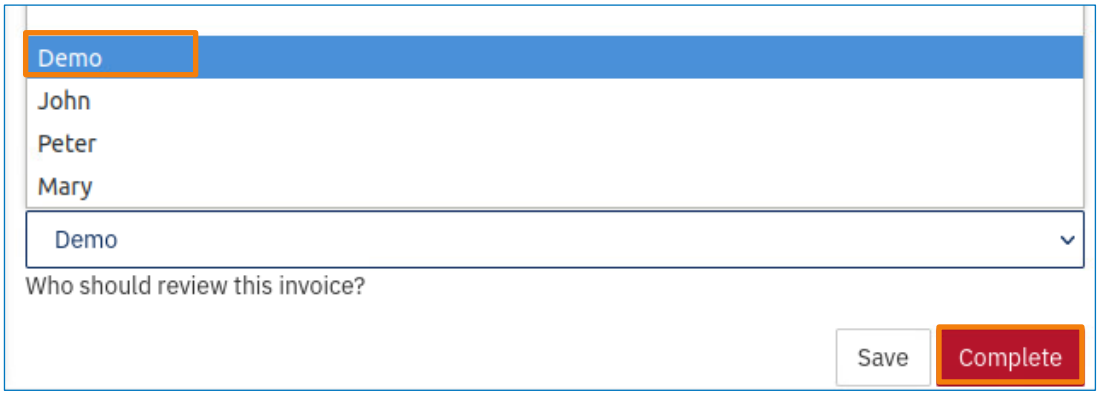
19. New process instances can be started by clicking the **Start process** button in the menu bar.



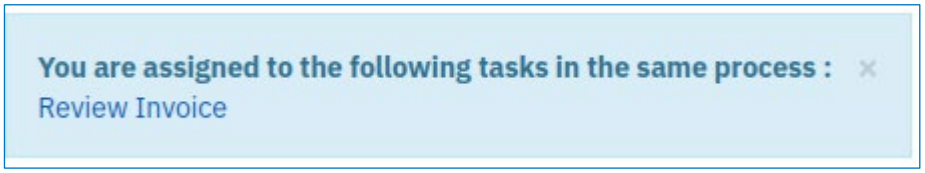
20. The tasklist application shows a list of all the user tasks that must be performed. Users can also use filters to display tasks assigned to different users and groups. The list of tasks assigned to you can be found by choosing the **My Tasks** filter.

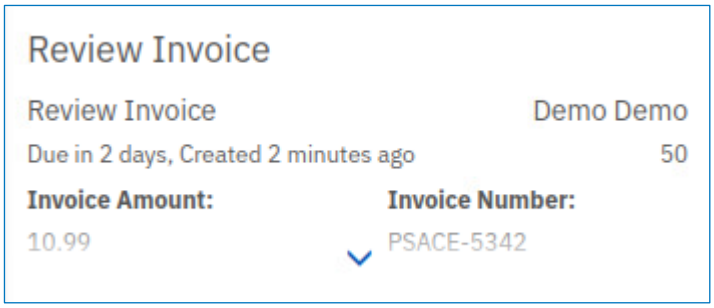


21. Click on the first task on the list. In this case, the task is to assign a reviewer that will then have to perform the following task. Choose the **demo** user (the user that you are currently using) to assign the next task to yourself and click the **Complete** button.

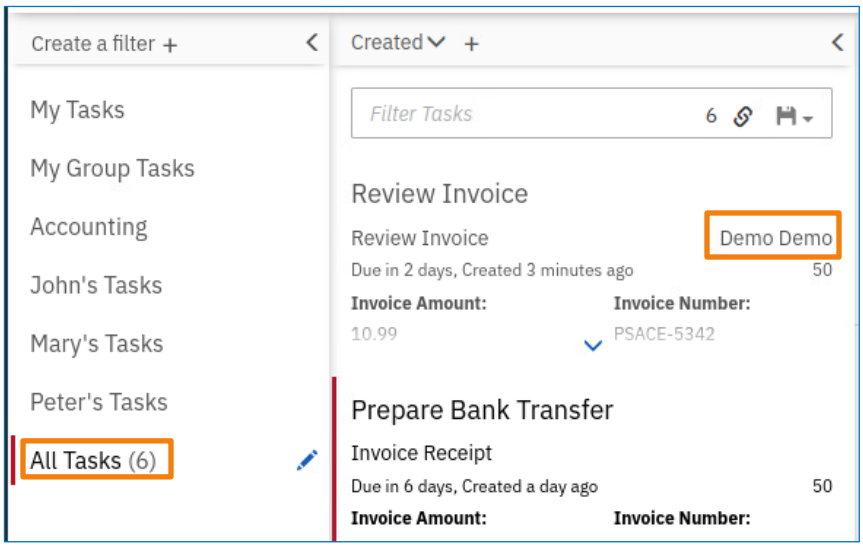


22. Since you chose the user that you are currently logged in with, the next user task is also assigned to you.





23. To check other tasks that are not yet assigned, click the **All Tasks** button. You can see that the task **Review Invoice** is assigned to your **Demo** user.



24. Click on one of the other tasks that are not yet assigned, for example, **Prepare Bank Transfer**. You can see that first you need to **claim** this task to complete it.

25. You now explored the two main Camunda applications that provide functionalities to manage the deployed processes. The other side of the processes is creating them. Camunda supports BPMN notation. The easiest way to create new BPMN processes for Camunda is by using the Camunda Modeler which provides a graphical interface for designing the workflows. Camunda Modeler is a separate application that you can open from your terminal.

    Open a new terminal window and navigate to the **camunda-modeler-5.6.0-linux-x64/** directory.

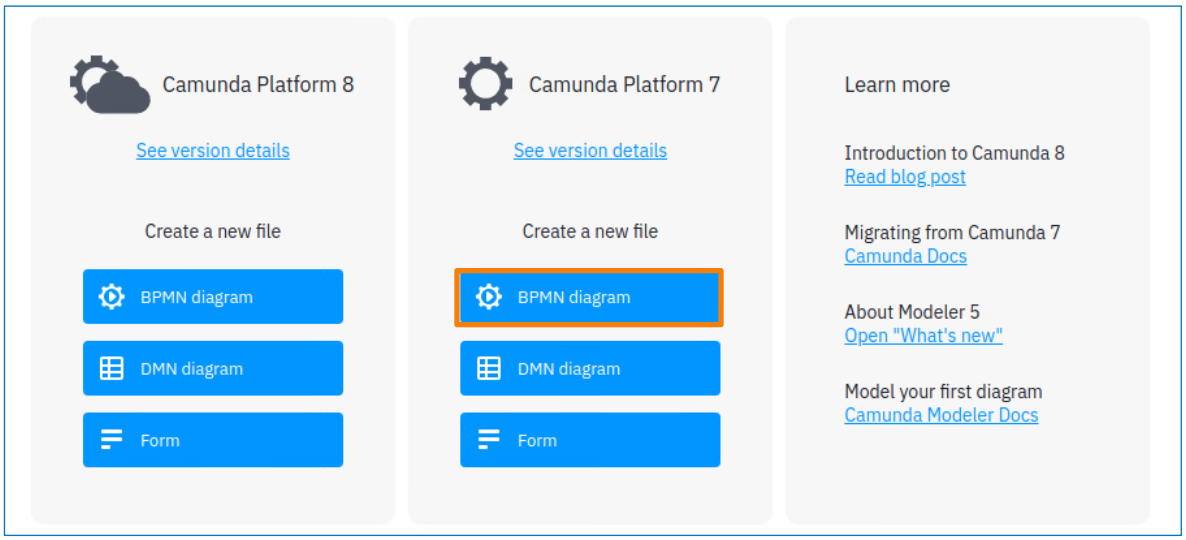
```
[~]$ [~]$ cd camunda-modeler-5.6.0-linux-x64/
[~/camunda-modeler-5.6.0-linux-x64]$
```

26. Start the Modeler application by executing the **./camunda-modeler** command.

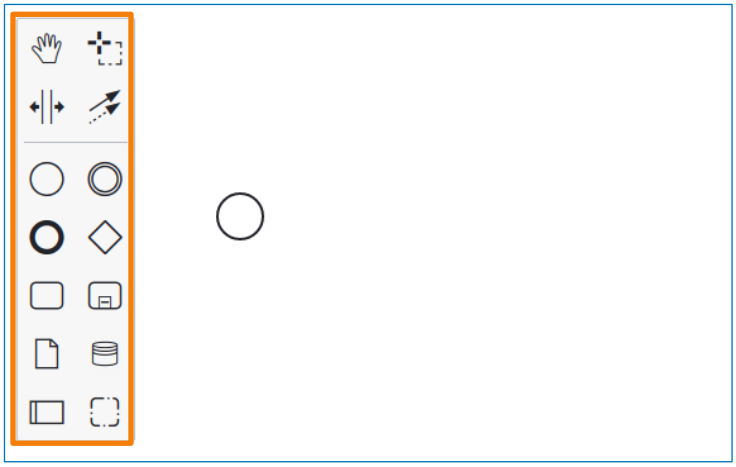
```
[~/camunda-modeler-5.6.0-linux-x64]$ ./camunda-modeler
```

27. Camunda Modeler should open. The application can be used for designing BPMN and DMN diagrams. In this case, you will focus on the BPMN workflows. Click the **BPMN diagram** button and make sure that you choose the **Camunda Platform 7 version** since your lab environment runs this version of the platform.

NOTE: If you get a pop-up window with the "New version available" message you can ignore it by pressing the **Skip this version** button.
Also, if you see the privacy preferences page first, you can close it by clicking the save button.

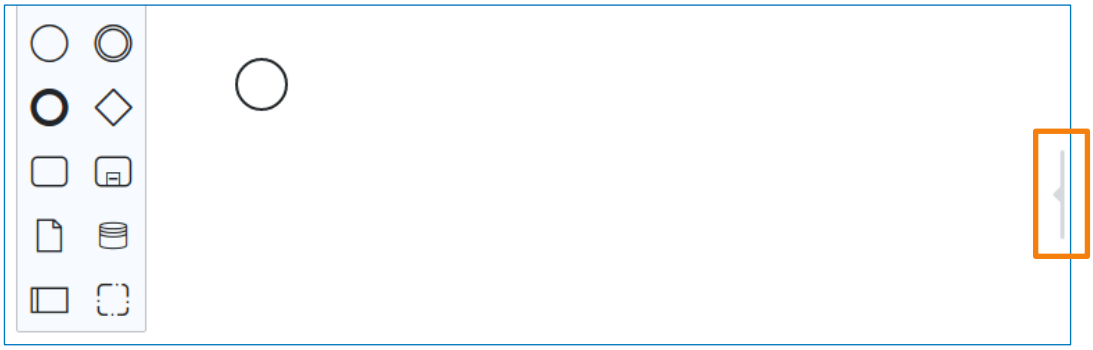


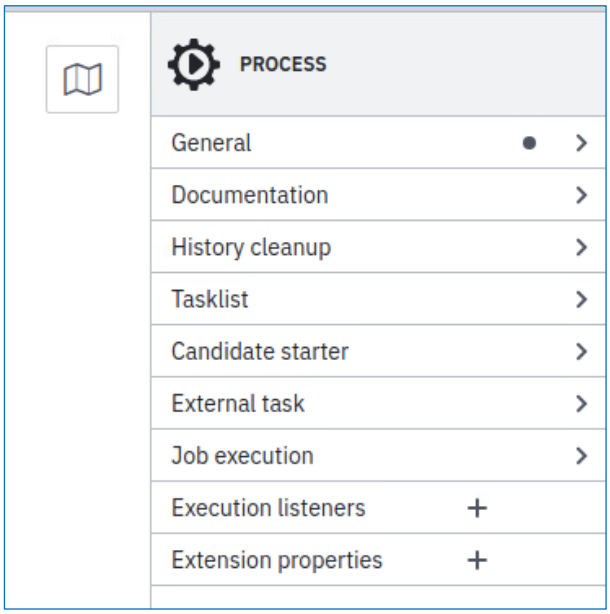
28. A graphical interface for designing BPMN diagrams should open. The menu on the left side of the screen includes icons that represent different actions. To construct the workflow you can simply drag and drop those icons on the screen and connect them.



29. On the right side of the screen there is a properties menu where you can define all the details required for the process to run.

NOTE: If the properties menu is not there you can open it by clicking the arrow button on the ride side of the editor.



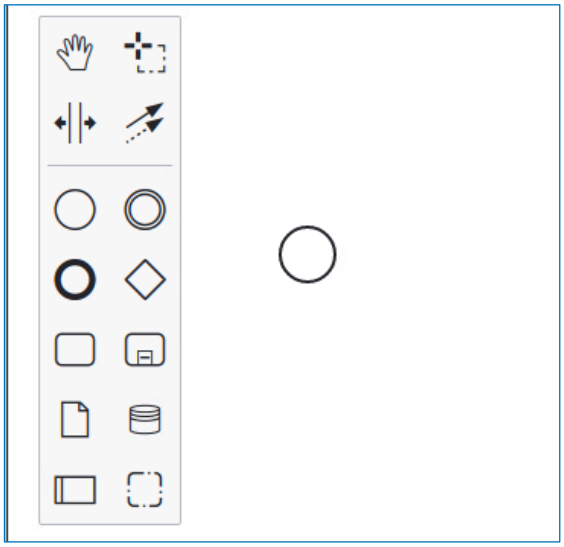


30. When the process is created you can deploy it to Camunda using the **Deploy** button in the Modeler app.

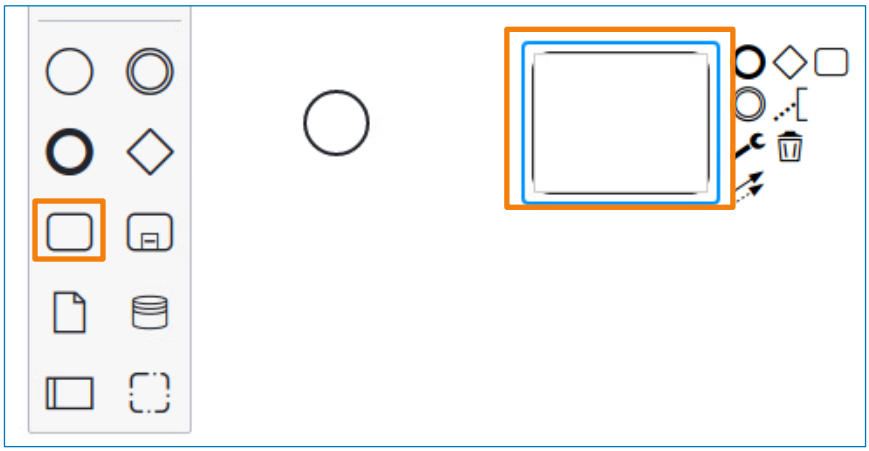
Task 2: Creating and Deploying Custom Camunda BPMN Process

In this task, you create a simple BPMN process using the Modeler application and then deploy it to Camunda.

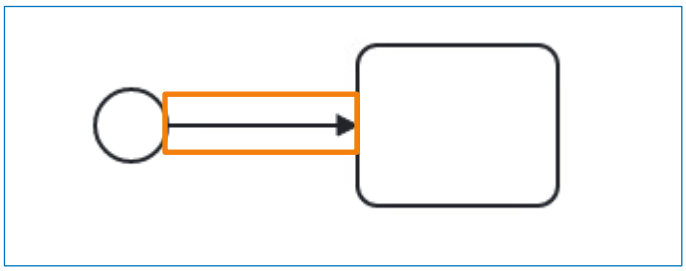
1. If you closed the Modeler open it again (follow the steps from the previous task to reopen it). You should see a blank process that only includes a start event (circle icon).



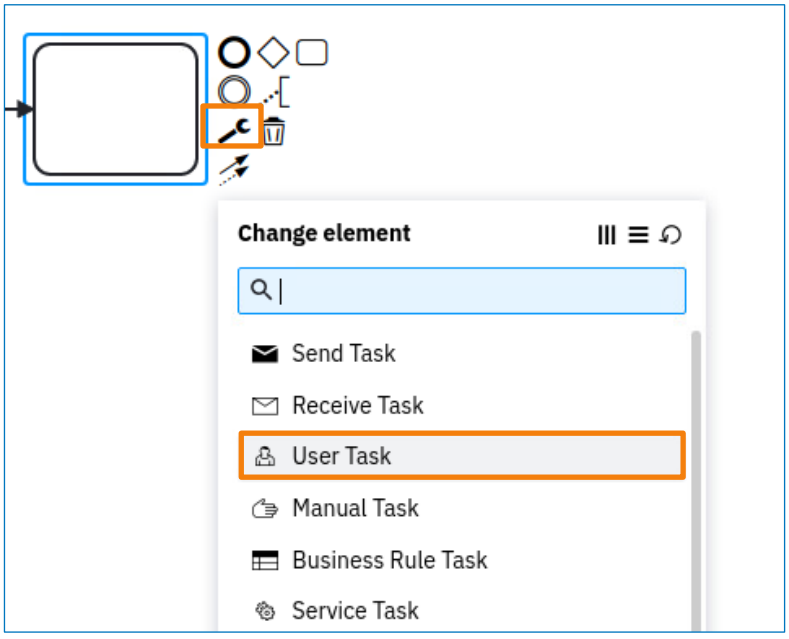
2. Add the first task of the process by dragging and dropping the **Create task** icon.



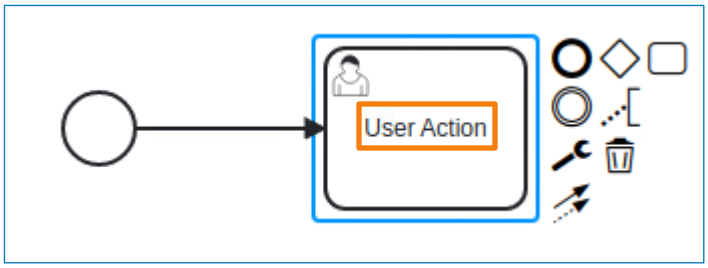
3. Connect the Start Event and the Task that you just added by clicking the **Start Event** and choosing the **arrows** icon from the side menu.



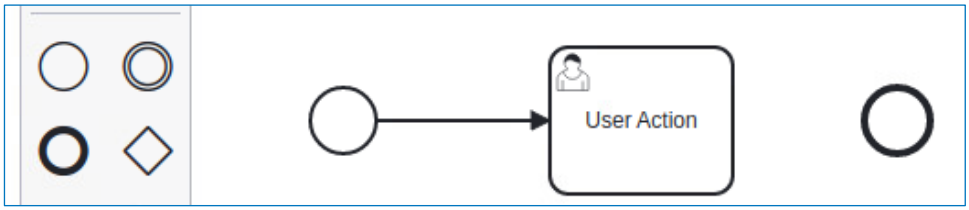
4. Different types of BPMN tasks can be used in Camunda. The most commonly used are User Tasks which require some kind of user action and Service Tasks which include some kind of logic and are executed automatically. In this example, set the type of task to **User Task**. Select the Create task icon and click the **wrench** button.



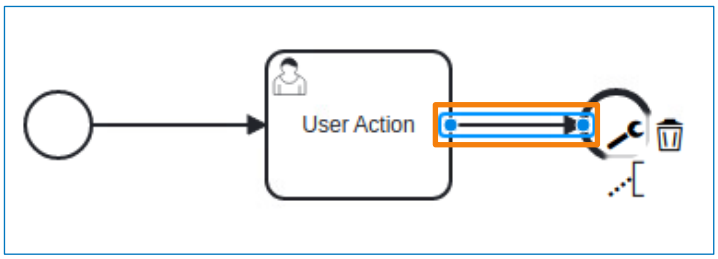
5. The task should be now marked as a user task (user symbol in the top-left corner). Name the task **User Action** by double-clicking the task and entering the name.



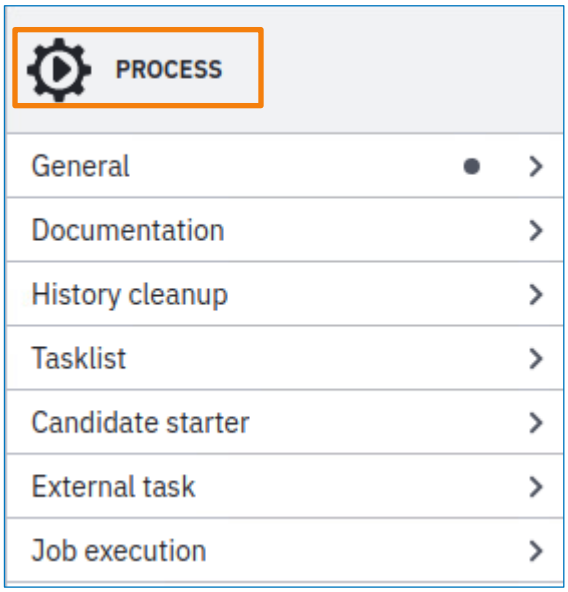
6. Since this is a simple task meant to demonstrate the development process, you can now add the **End Event** by dragging and dropping the **EndEvent** symbol (bold circle).



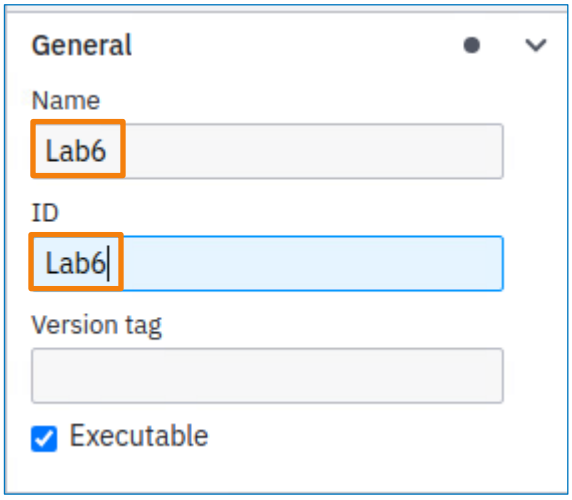
7. Connect the **User Action** task and **End Event** by clicking on the task and choosing the **arrows** function.



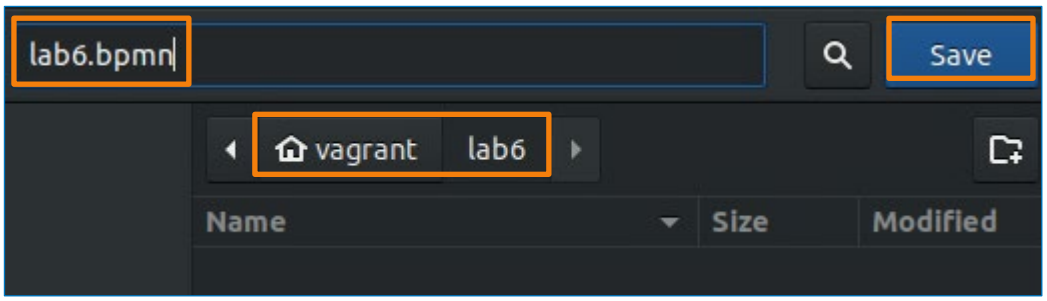
8. Now you only have to add some process information to be able to deploy it. Click on the empty space anywhere in the editor and make sure that you do not have any task selected. The Properties menu should show **Process** related settings.



9. Expand the **General** section of properties and enter **Lab6** as the **name** and **ID** of the process.



10. Save the process to the **Lab6** directory. Click **File > Save File As** and choose the **vagrant/lab6** directory. Name the file **lab6.bpmn** and click **Save**.



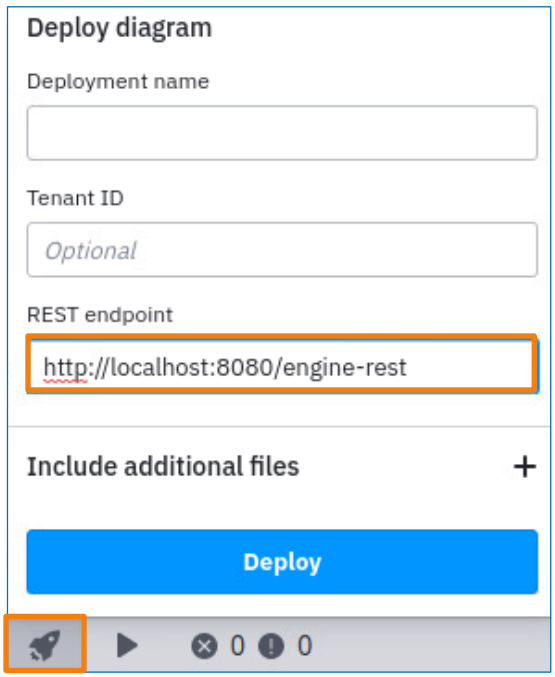
11. Before deploying the process to Camunda, explore what the .bpmn file that you created in the graphical editor looks like. Open a new terminal window and navigate to **lab6/** directory.

```
[~]$ [~]$ cd lab6/
[~/lab6]$
```

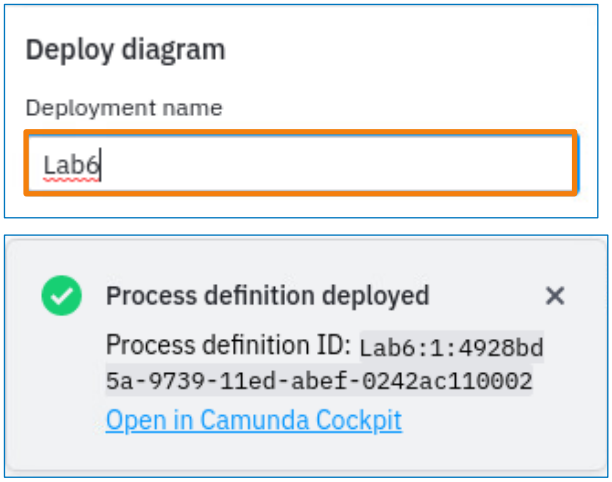
12. Display the contents of the file using the **cat** command. You should see your BPMN process encoded in XML.

```
[~/lab6]$ cat lab6.bpmn
<... output omitted ...>
  <bpmn:process id="Lab6" name="Lab6" isExecutable="true">
    <bpmn:startEvent id="StartEvent_1">
      <bpmn:outgoing>Flow_11hla0q</bpmn:outgoing>
    </bpmn:startEvent>
    <bpmn:sequenceFlow id="Flow_11hla0q" sourceRef="StartEvent_1"
targetRef="Activity_1e4wb1z" />
    <bpmn:userTask id="Activity_1e4wb1z" name="User Action">
      <bpmn:incoming>Flow_11hla0q</bpmn:incoming>
      <bpmn:outgoing>Flow_0n9cqp6</bpmn:outgoing>
    </bpmn:userTask>
    <bpmn:endEvent id="Event_1yqnzar">
<... output omitted ...>
```

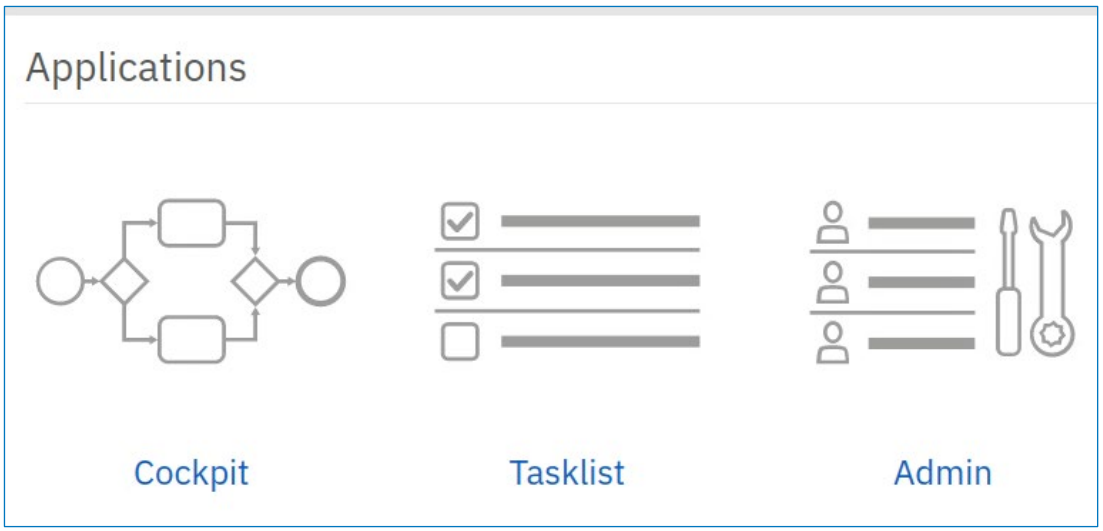
13. Navigate back to the Modeler and deploy the process to Camunda using the **Deploy process** button. Note that there is a REST endpoint field. This is because the Modeler interacts with the Camunda Engine through the REST API. You could also deploy the file manually by sending the request to this URL and including the .bpmn file.



14. Name the deployment **Lab6** and click deploy. You should see a **Process definition deployed** pop-up message.



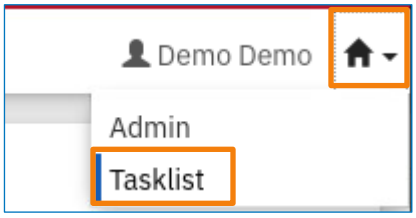
15. Navigate back to the dashboard of your Camunda platform. If you closed it, open a new browser window and click on the **Camunda bookmark**.



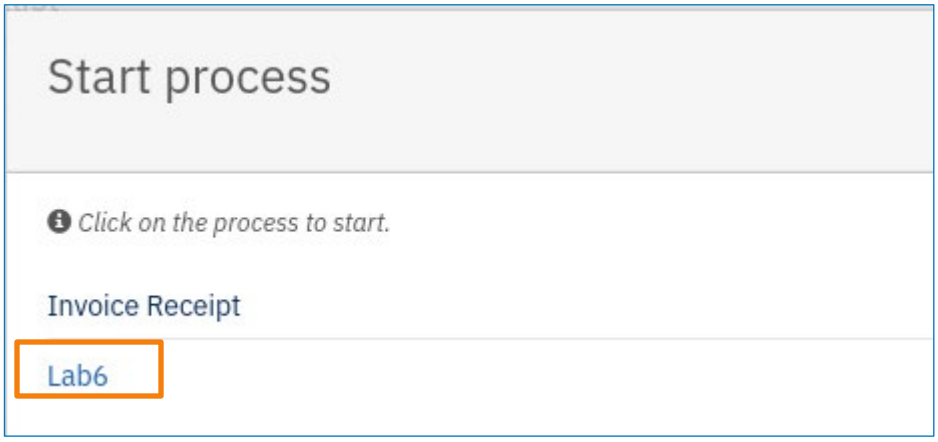
16. First open the cockpit application and click **Processes**. The list of deployed processes should now include your **Lab6** process.

NOTE: If you currently have the Tasklist application open click on the house-shaped button in the top right corner to find the Cockpit.

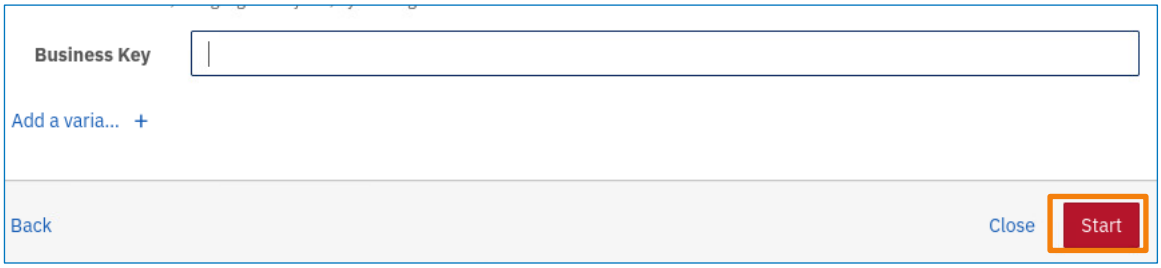
17. This process is not started yet. To do that, navigate to the **Tasklist** application.



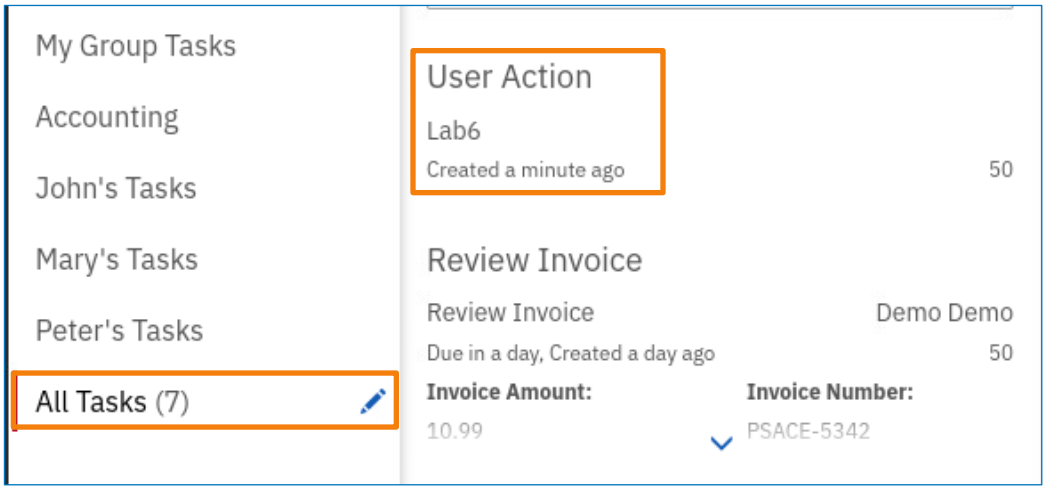
18. Click the **Start process** button from the top menu bar and select **Lab6** process.



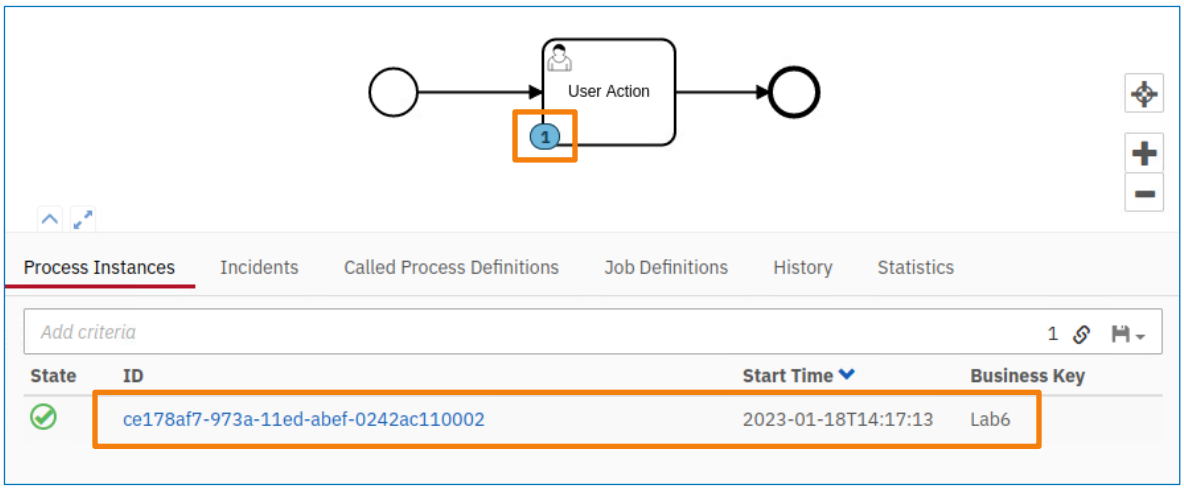
19. The Business key is not required so you can click the **Start** button.



20. Open the list of **All tasks**. Your **User Action** task should now be present on the list.



21. If you navigate back to the **Cockpit,** you should see that the process has started and is currently at the **User Action** task.



End of Lab

Lab 7: Camunda Workflow for VLAN ID

Objectives

- Creating a new Java project
- Designing a BPMN process and adding the Java code
- Deploying the project to Camunda
- Starting the process and observing the outcome

Task 1: Creating a Java Project

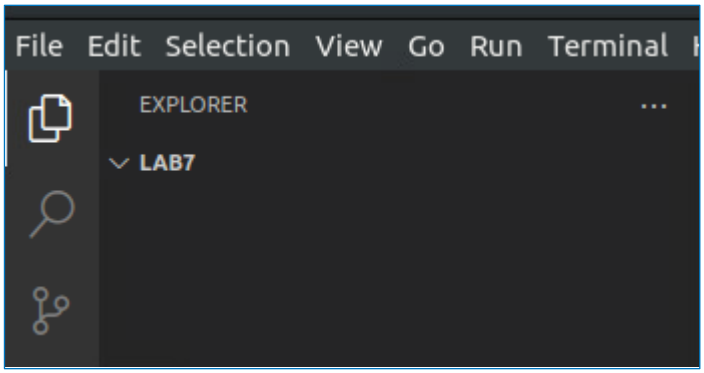
In this task, you prepare the working environment and create a Java project that will be used in a custom Camunda workflow.

1. First, you have to create a new Java project that you will use in your Camunda process. You will use Camunda's Maven Archetypes to generate the project.

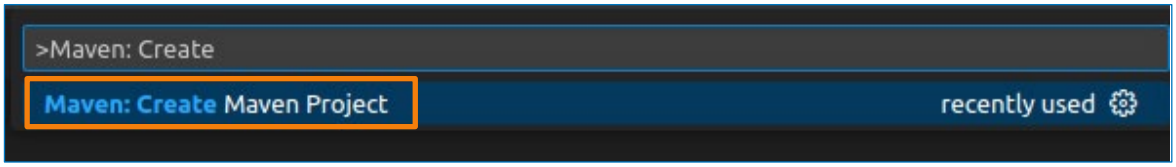
   Create a new **lab7** directory and open it with the **code .** command.

```
[~]$ mkdir lab7
[~]$ cd lab7/
[~/lab7]$ code .
```

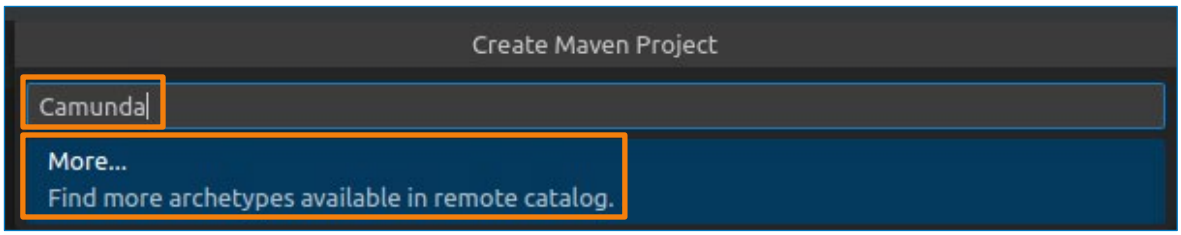
2. The empty lab7 directory should open in Visual Studio Code.



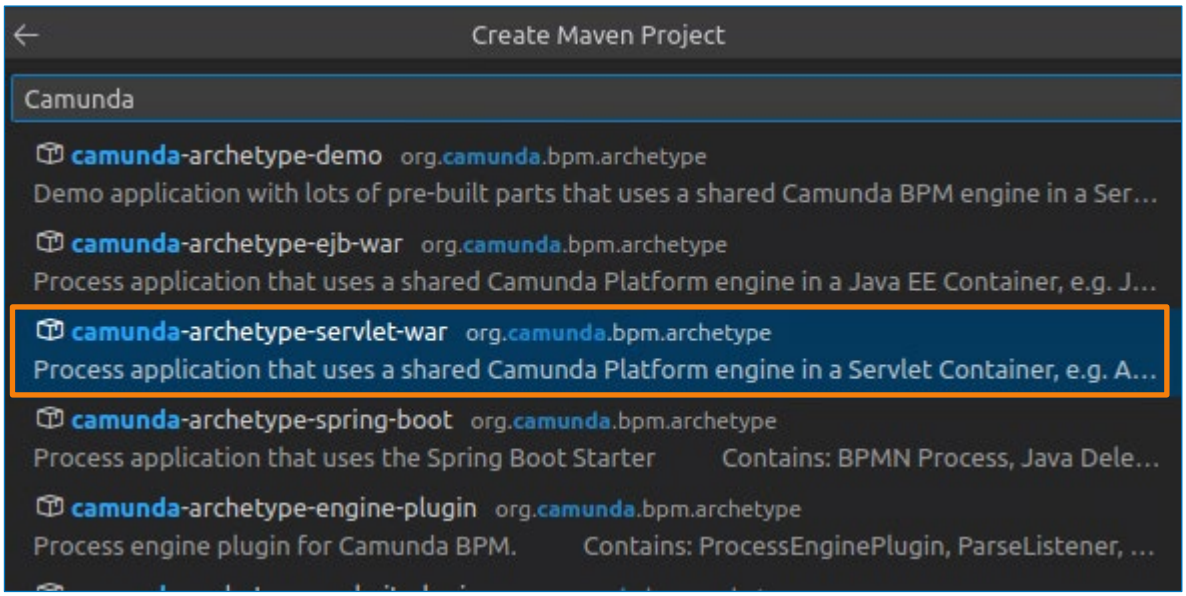
3. Create a new Maven project by pressing **Ctrl + Shift + P** to open the Command Palette and entering **Maven: Create Maven Project** in the search bar. Click the **Create Maven Project** button.



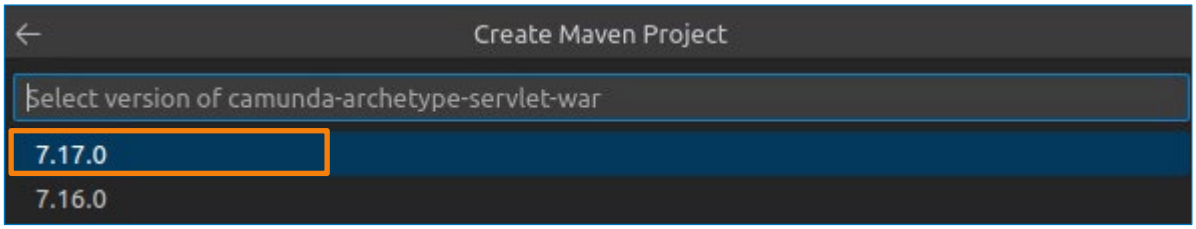
4. To find the correct Archetype, search for **Camunda** in the **Select an archetype** menu and click the **More...** button.



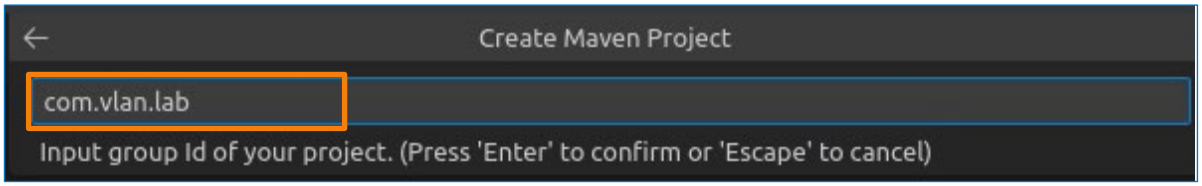
5. Choose the **camunda-archetype-servlet-war** from the dropdown menu. This archetype works well for most of the basic Camunda projects.



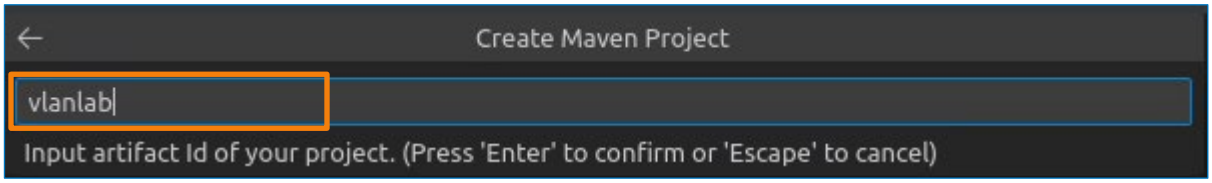
6. Choose the latest available version, in this case, the **7.17.0** version.



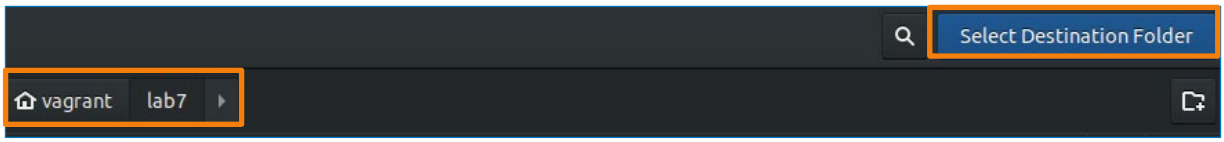
7. For the **group Id** of the project, enter **com.vlan.lab** and press **Enter**.



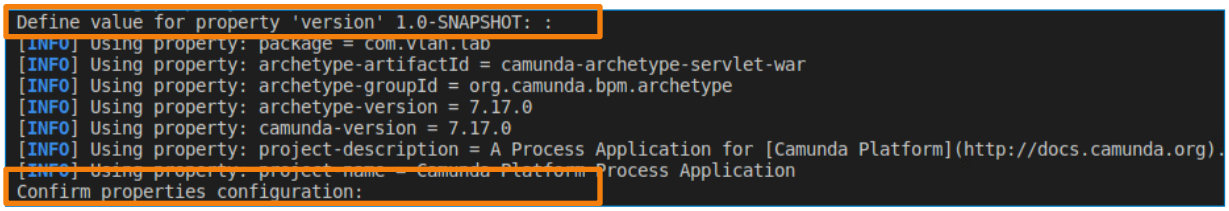
8. For the **artifact Id** of the project, enter **vlanlab** and press **Enter**.



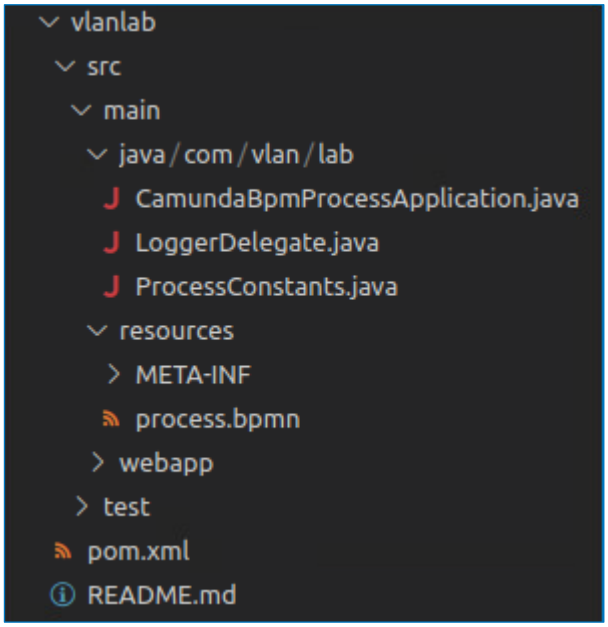
9. Choose **/home/vagrant/lab7** as the project location and click the **Select Destination Folder** button.



The createProject task will open in the VSC terminal. When prompted for **property version** and **properties configuration**, just press **Enter**.



10. The project should be built successfully. Expand the contents of the project and explore the created files.



Task 2: Creating the BPMN Process for VLAN Reservation

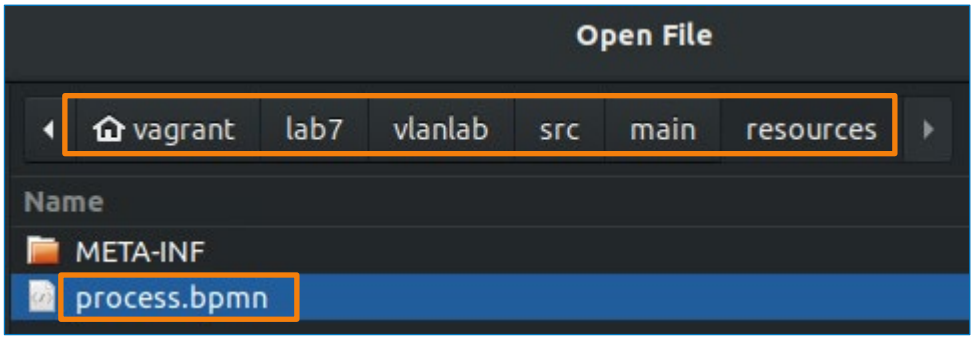
In this task, you create a custom BPMN workflow that reserves a VLAN ID if possible and gives feedback about the reservation status.

1. To design the workflow, use the Camunda Modeler application. To start the Modeler application, open a new terminal window and navigate to the **camunda-modeler-5.6.0-linux-x64/** directory then run the **./camunda-modeler** command.

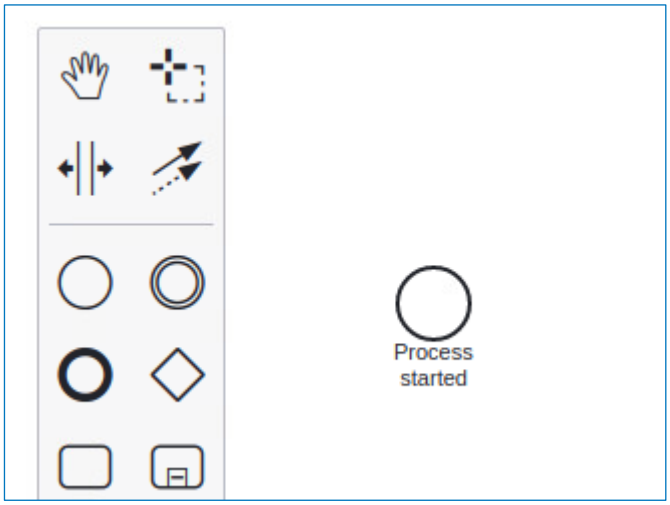
```
[~]$ [~]$ cd camunda-modeler-5.6.0-linux-x64/
[~/camunda-modeler-5.6.0-linux-x64]$ ./camunda-modeler
```

2. The Camunda Modeler main menu should open. Open the .bpmn file that was created automatically as a part of your project. Navigate to **File > Open File** and search for the **.bpmn** file in the directory of your Java project.

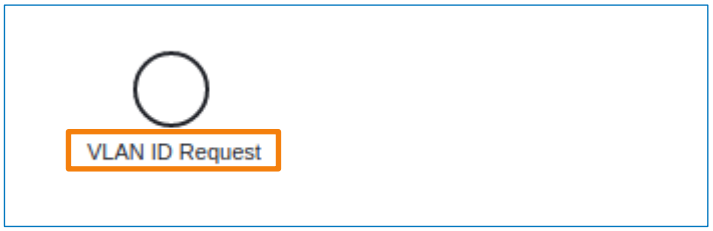
NOTE: If you did not close the .bpmn file from the previous lab, the modeler will open that file. You can close it by clicking the **x** button next to the file.



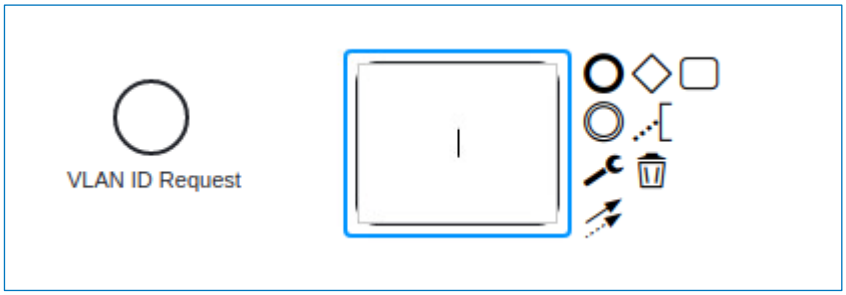
3. A process with only a start event and an end event should open. Delete the end event for now. Your process should now only include the start event.



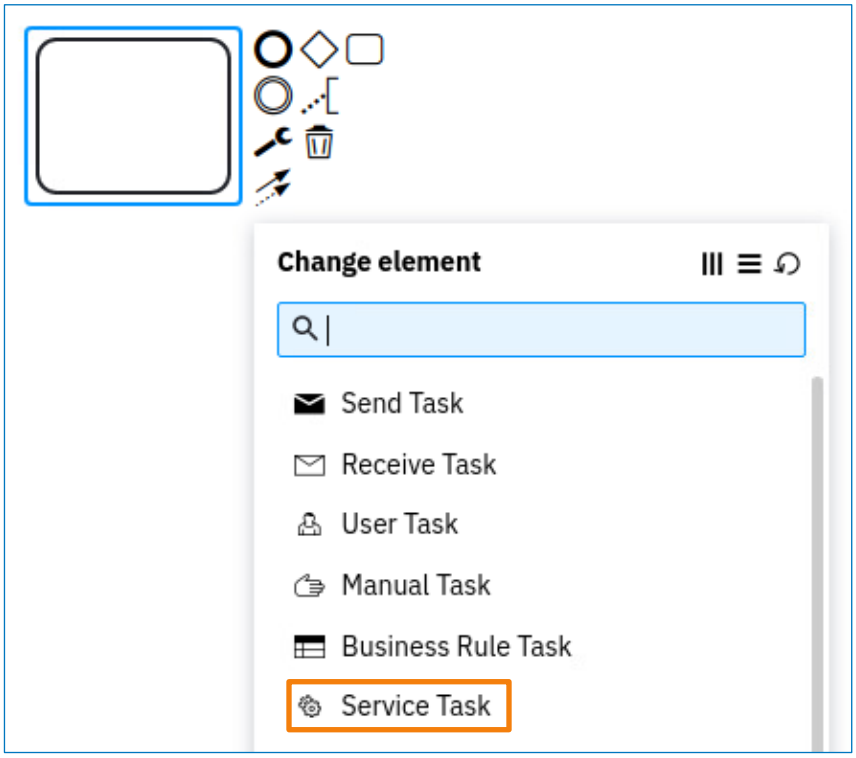
4. First name the start event **VLAN ID Request** as this process should be called when a request for a new VLAN ID is triggered. To change the name, double-click the start event icon and enter the name or change it under the general section of the properties menu.



5. Add the first task of the process by dragging and dropping the **Create task** icon.

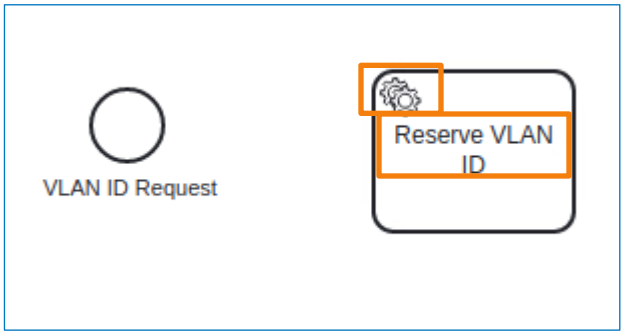


6. Set the type of task to **Service Task** by clicking the **wrench** button and choosing the type from the dropdown menu.

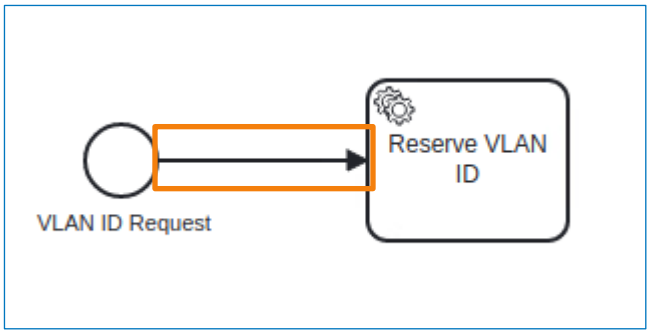


7. Name the task **Reserve VLAN ID** by double-clicking it and entering the name.

NOTE: Check that the task is now marked as a **service task**.

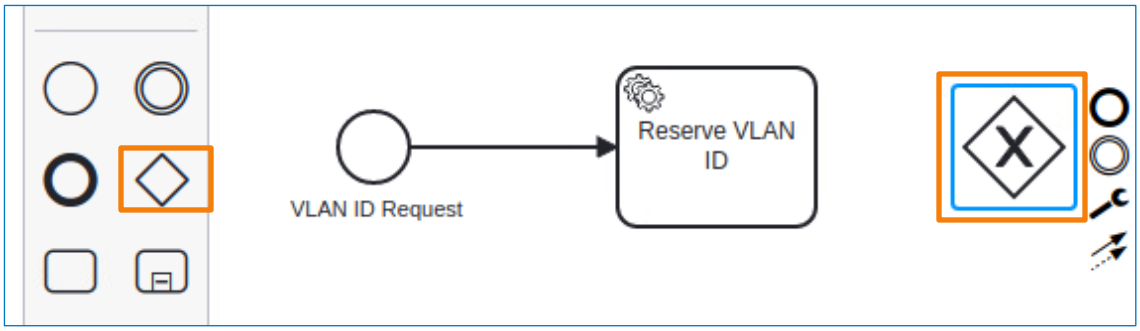


8. Connect the Start Event and the Task that you just added by clicking the **Start Event** and choosing the **arrows** icon from the side menu.

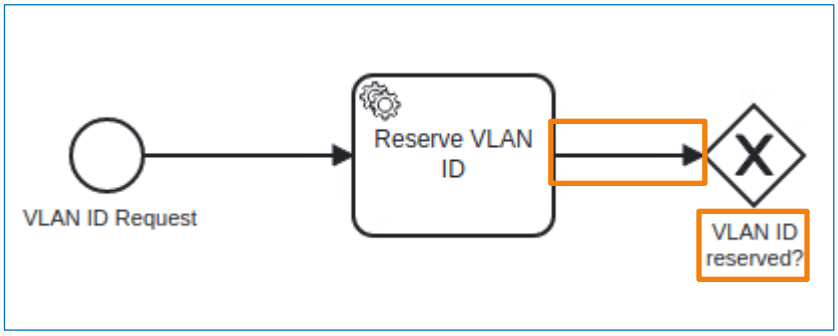


9. The goal of the **Reserve VLAN ID** task is to reserve one of the available IDs. Now you want to add the tasks that will give feedback about reservation status and notify the user whether the reservation was successful.

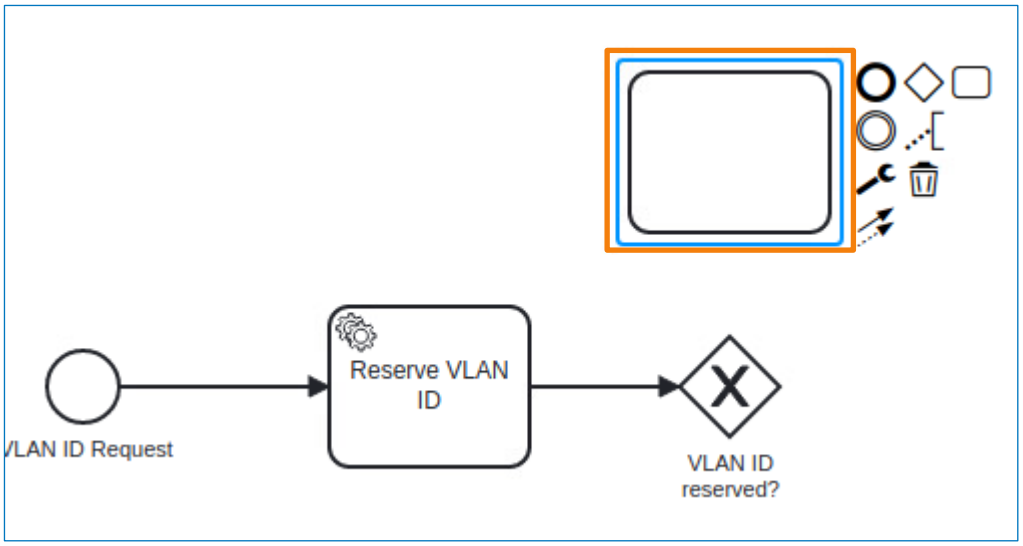
   First, add an XOR gateway to the process by dragging and dropping the **Create Gateway** icon.



10. Connect the **Reserve VLAN ID** with the Gateway and name the gateway **VLAN ID reserved?**.

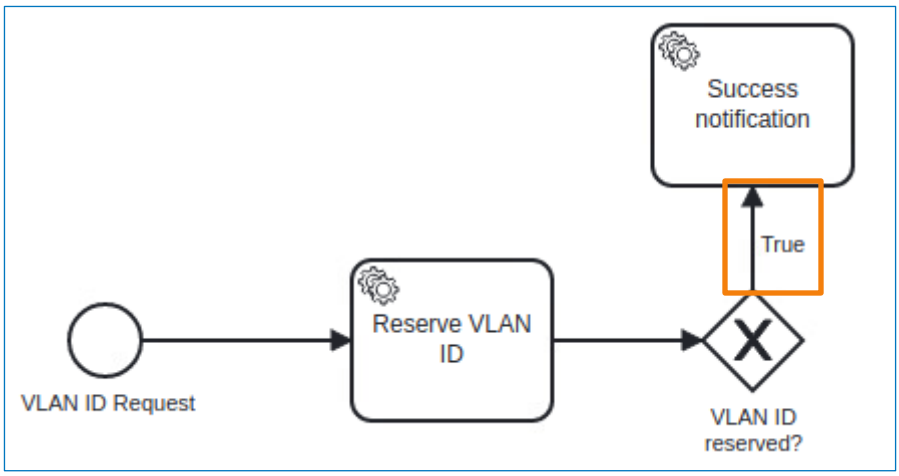


11. Now you will add two different paths to the workflow. First, add a task that will notify the user if the reservation was successful. Add another task to the process.

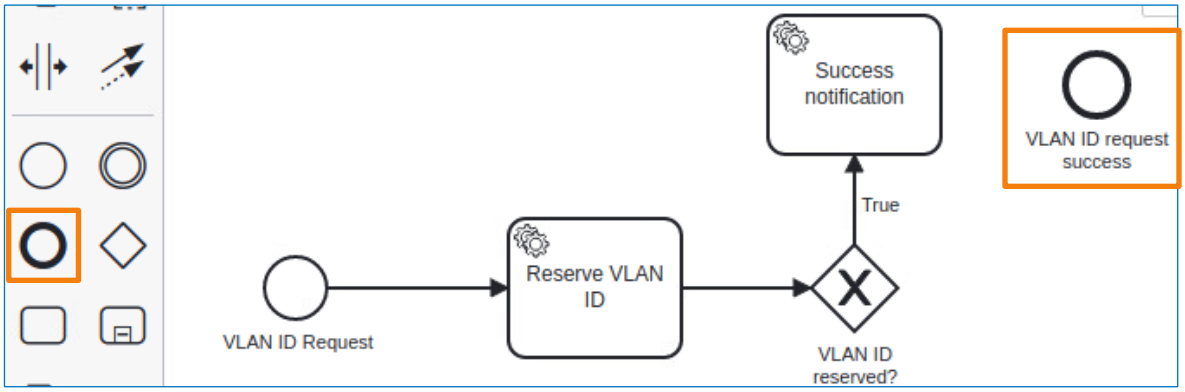


12. Change the type of the task to **service task** and name it **Success notification**. This task will notify the user when the VLAN ID is reserved successfully.

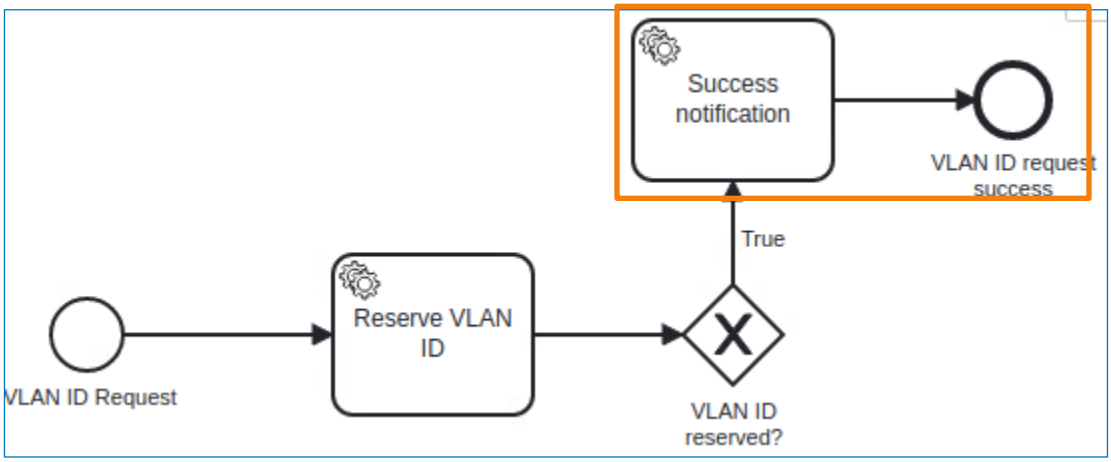
13. Connect the Gateway with the task that you created in the previous step and name the connection **True**, since this part of the workflow is only triggered when VLAN ID is reserved successfully.



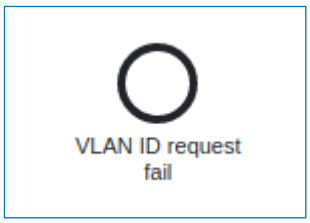
14. Add the **endEvent** to the workflow by dragging and dropping the **bold circle icon** to the screen. Name the event **VLAN ID request success**.



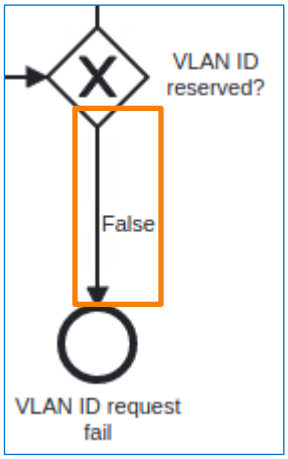
15. Connect the **Success notification** task with the **end event** as the process must end when the ID is reserved successfully.



16. Now you must take care of the other possible scenario – unsuccessful reservation. In this scenario, you will only add a new **endEvent** and name it **VLAN ID request fail**.

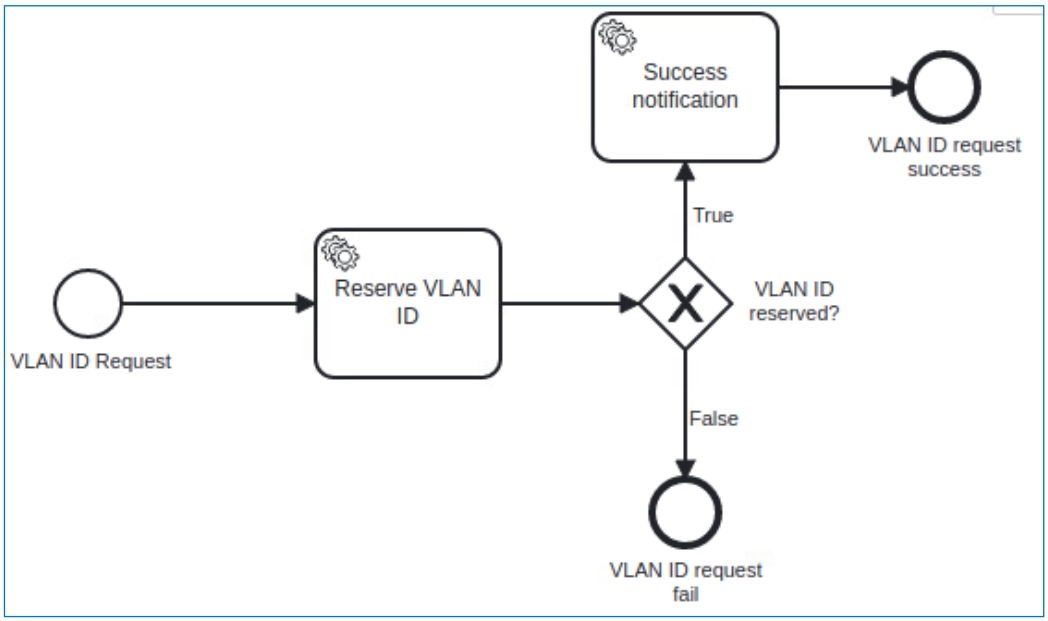


17. Add another connection to the gateway by connecting it to the **VLAN ID request fail** end event. Name the connection **False** as this path will be selected if the VLAN ID is not reserved.

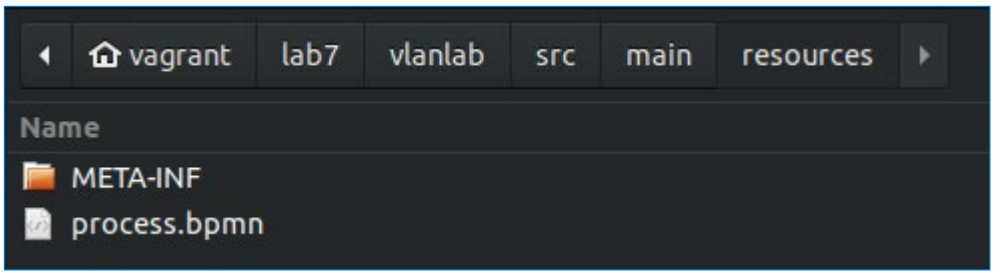


18. Your workflow should now look like the following figure:

NOTE: Make sure that your workflow looks exactly as shown here, otherwise the process will not be executed successfully in the next tasks.



19. Save the process (**File > Save File**). Make sure that you save the file to the correct location in your project.



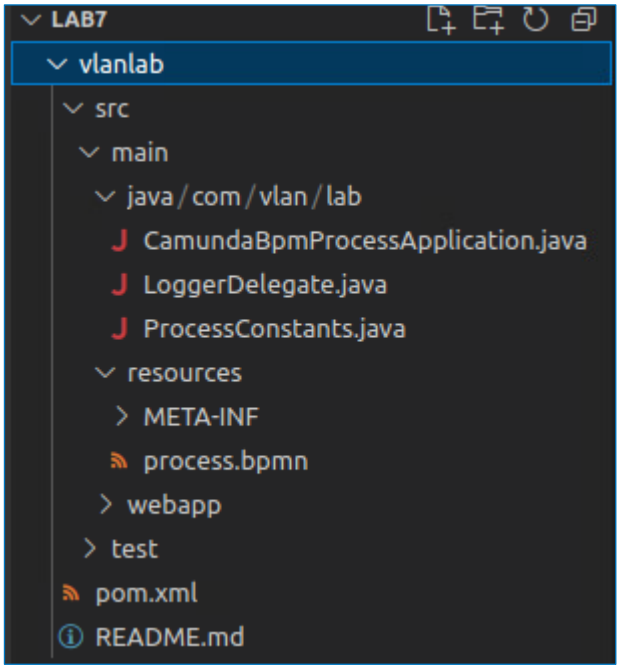
Task 3: Implementing the Process Logic

In the previous task, you designed a BPMN process for VLAN ID reservation. But at this point, the process does not perform anything because the logic is not yet implemented. In this task, you implement the needed logic by writing some Java code that will be executed by the service tasks.

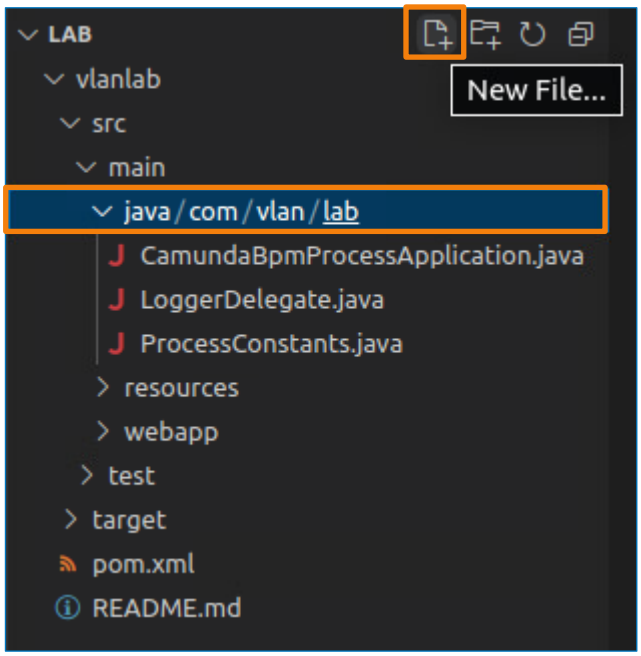
1. If you closed the Visual Studio Code with the project that you created in the first task, open it again from the terminal. Navigate to the **lab7/** directory and open the contents with the **code .** command.

```
[~]$ [~]$ cd lab7/
[~/lab7]$ code .
```

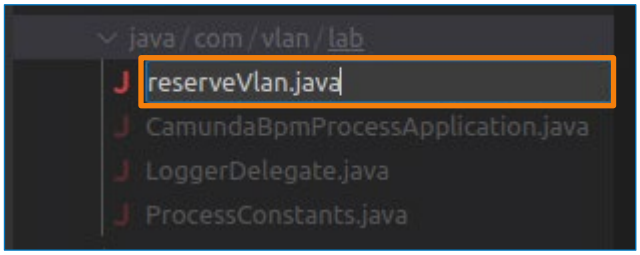
2. Make sure that you opened the correct directory and that you can access your **vlanlab** Java project.



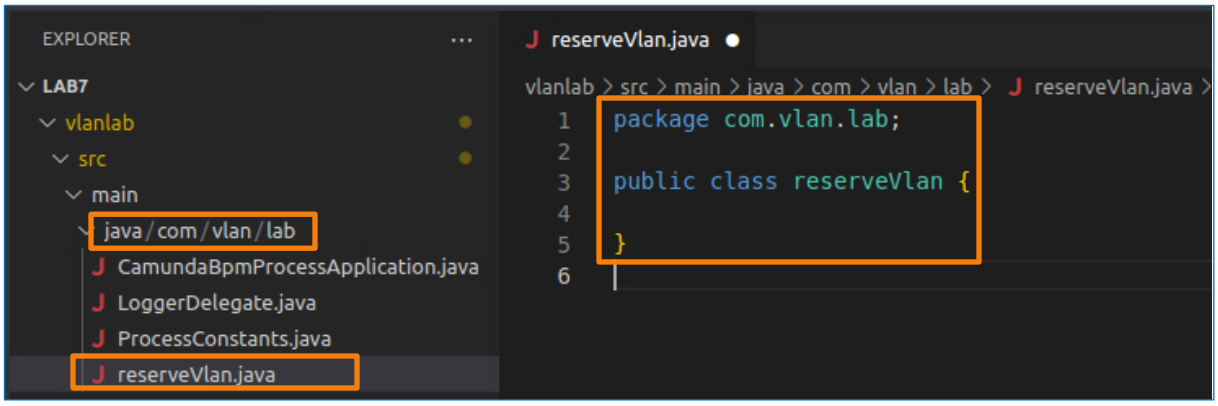
3. Since you used the Camunda archetype to create the project, most of the needed files are already prepared for you, and you only have to add the custom logic that you want to use in your process. In this example, you will add a new Java class that will reserve a VLAN ID if it is available.
   Select the **java/com/vlan/lab** directory and click the **New File…** button.



4. Name the file **reserveVlan.java** and press **Enter**.



5. A new reserveVlan.java class should be created. Make sure that it is saved to the correct directory (**main/java/com/vlan/lab**).



6. Before you start developing your custom logic, you must add the implementation of Java delegate. Add the following code to your Java class:

```
package com.vlan.lab;

import org.camunda.bpm.engine.delegate.DelegateExecution;
import org.camunda.bpm.engine.delegate.JavaDelegate;

public class reserveVlan implements JavaDelegate {

}
```

7. Create a method called **execute** that takes a parameter of type “DelegateExecution”. “DelegateExecution” is an interface in the BPMN package. It represents the state and context of the BPMN process and provides access to the current process state, variables, and other information.

```
package com.vlan.lab;

import org.camunda.bpm.engine.delegate.DelegateExecution;
import org.camunda.bpm.engine.delegate.JavaDelegate;

public class reserveVlan implements JavaDelegate {

    @Override
    public void execute (DelegateExecution execution) throws Exception {

    }
}
```

8. The method can be used to define variables that are used in the BPMN process. For example, you can set the value of the variable **name** to the context of a BPMN process.

```
package com.vlan.lab;

import org.camunda.bpm.engine.delegate.DelegateExecution;
import org.camunda.bpm.engine.delegate.JavaDelegate;

public class reserveVlan implements JavaDelegate {

    @Override
```

```
    public void execute (DelegateExecution execution) throws Exception {
        execution.setVariable("name", "student");
    }
}
```

9. The same method will be used to define the variable **vlanId**, but first, you must define which IDs are available. For the purposes of this demonstration, you will simply define a list with a couple of available IDs.

```
package com.vlan.lab;

import org.camunda.bpm.engine.delegate.DelegateExecution;
import org.camunda.bpm.engine.delegate.JavaDelegate;

import java.util.List;

public class reserveVlan implements JavaDelegate {

    private static List<Integer> availableIds = List.of(10,20,30,40);

    @Override
    public void execute (DelegateExecution execution) throws Exception {

        execution.setVariable("name", "student");
    }
}
```

10. Assign one of the available Ids to a variable named **vlanId.** Take a random Id from the list by leveraging the **Random** function.

```
package com.vlan.lab;

import org.camunda.bpm.engine.delegate.DelegateExecution;
import org.camunda.bpm.engine.delegate.JavaDelegate;

import java.util.List;
import java.util.Random;

public class reserveVlan implements JavaDelegate {
```

```
    private static List<Integer> availableIds = List.of(10,20,30,40);

    @Override
    public void execute (DelegateExecution execution) throws Exception {

        Random rando = new Random();
        int index = rando.nextInt(availableIds.size());
        int vlanId = availableIds.get(index);

        execution.setVariable("name", "student");
        execution.setVariable("vlanId", vlanId);
    }
}
```

11. Your BPMN process also includes a scenario where there are no available VLAN IDs. This means that your code must execute correctly even if you define an empty list of available IDs. Add some logic that will check if the list is empty before reserving an ID and assigning it to the variable.

```
package com.vlan.lab;

import org.camunda.bpm.engine.delegate.DelegateExecution;
import org.camunda.bpm.engine.delegate.JavaDelegate;

import java.util.List;
import java.util.Random;

public class reserveVlan implements JavaDelegate {

    private static List<Integer> availableIds = List.of(10,20,30,40);

    @Override
    public void execute (DelegateExecution execution) throws Exception {


        if(availableIds.isEmpty()){
            execution.setVariable("vlanId", null);
            return;
        }
```

```
        Random rando = new Random();
        int index = rando.nextInt(availableIds.size());
        int vlanId = availableIds.get(index);

        execution.setVariable("name", "student");
        execution.setVariable("vlanId", vlanId);
    }
}
```

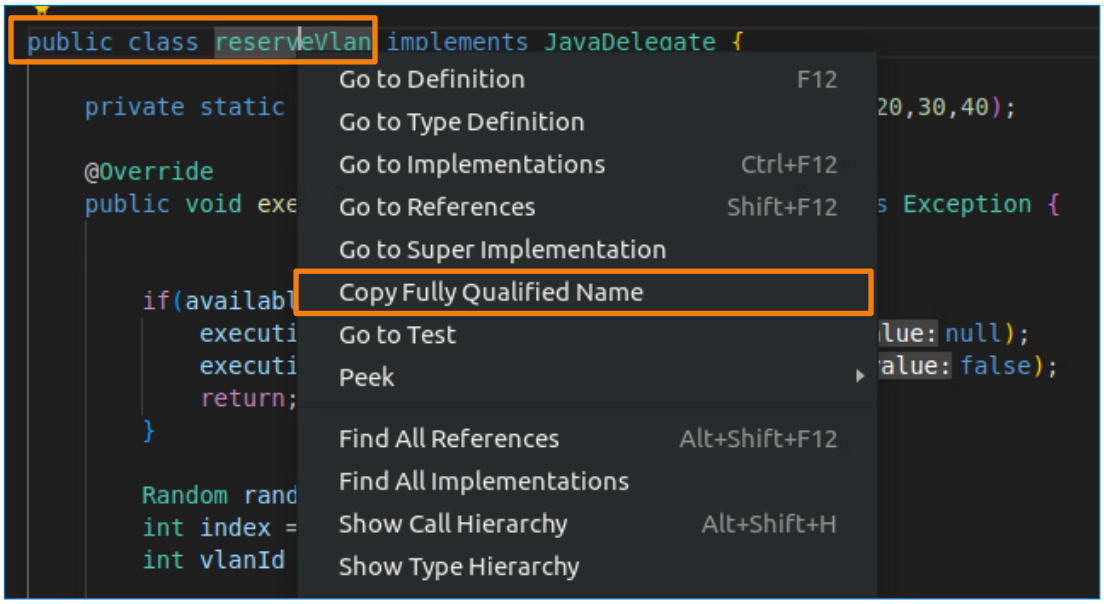
12. Define a new variable **vlanSet** and set it to **true** if the ID was reserved and to **false** if the ID was not reserved.

```
package com.vlan.lab;

import org.camunda.bpm.engine.delegate.DelegateExecution;
import org.camunda.bpm.engine.delegate.JavaDelegate;

import java.util.List;
import java.util.Random;

public class reserveVlan implements JavaDelegate {

    private static List<Integer> availableIds = List.of(10,20,30,40);

    @Override
    public void execute (DelegateExecution execution) throws Exception {

        if(availableIds.isEmpty()){
            execution.setVariable("vlanId", null);
            execution.setVariable("vlanSet", false);
            return;
        }

        Random rando = new Random();
        int index = rando.nextInt(availableIds.size());
        int vlanId = availableIds.get(index);
```

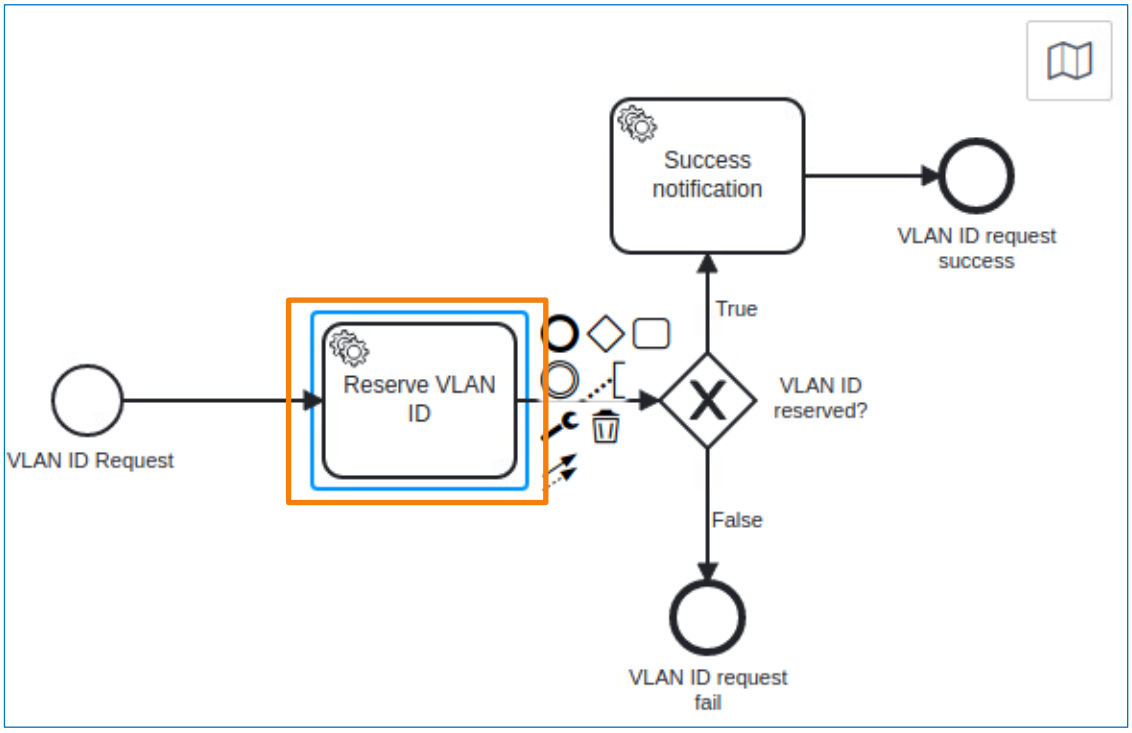
```
        execution.setVariable("name", "student");
        execution.setVariable("vlanId", vlanId);
        execution.setVariable("vlanSet", true);
    }
}
```

13. You developed all the logic for the first service task of your process. When the task is triggered, it will check the list of available VLAN IDs and if possible, it will reserve one of the available IDs. Now you have to attach this class to the service task in your process.

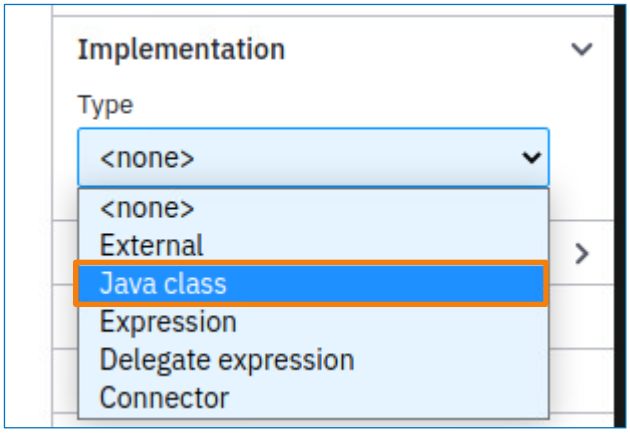
    Save the file and right-click the **reserveVlan** class and copy the fully qualified name.



14. Navigate back to the Modeler application and select the **Reserve VLAN ID** task.

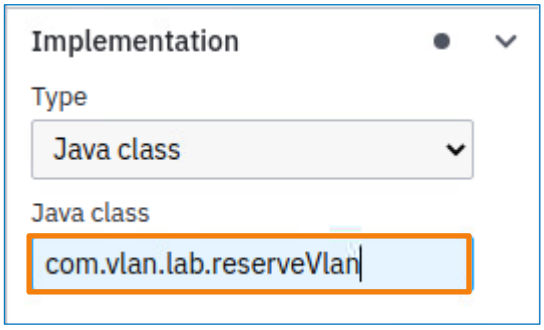


15. Expand the **Implementation** section of the properties menu and choose **Java class** as the implementation type.



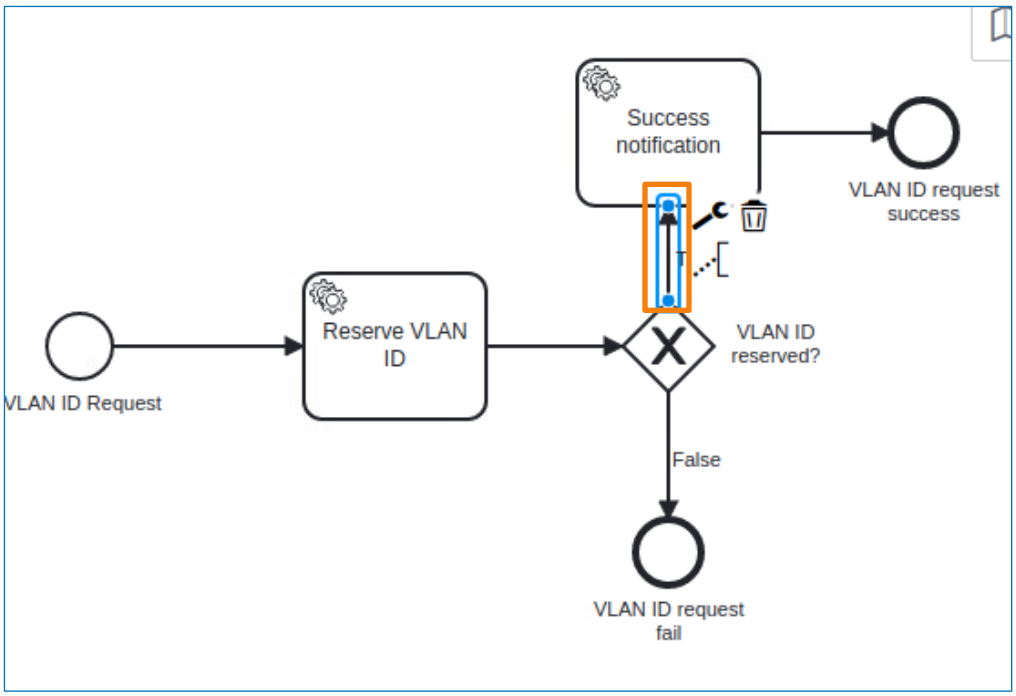
16. Paste the qualified name of your class to the **Java class** field.

NOTE: You must use the **Ctrl + V** shortcut as right-click menu is not available in this field.

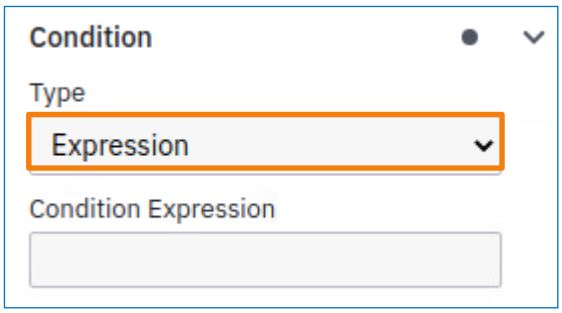


17. Now you covered the first task of your BPMN process. The next step that you must configure is the gateway that checks if the ID was reserved or not. Since you know that the first task will generate a variable named **vlanSet** with the value set to **true** or **false,** the gateway can simply check this variable.

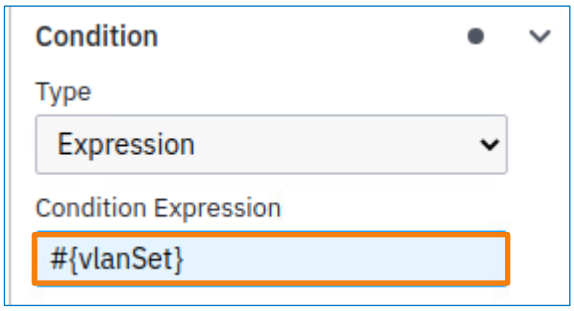
Click on the arrow named **True** that connects to the **Success notification task**.



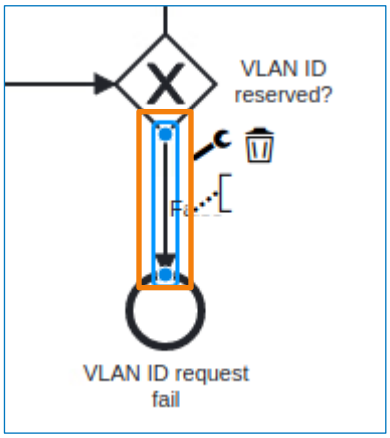
18. Expand the **Condition** section of the Properties menu. Here you can set under which condition the process will continue in this way. In this example, select **Expression** as the type of condition.



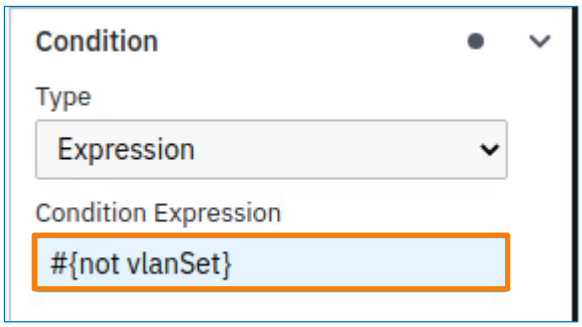
19. Enter **#{vlanSet}** in the **Condition Expression** field. This statement will check if the variable **vlanSet** is set to **true**.



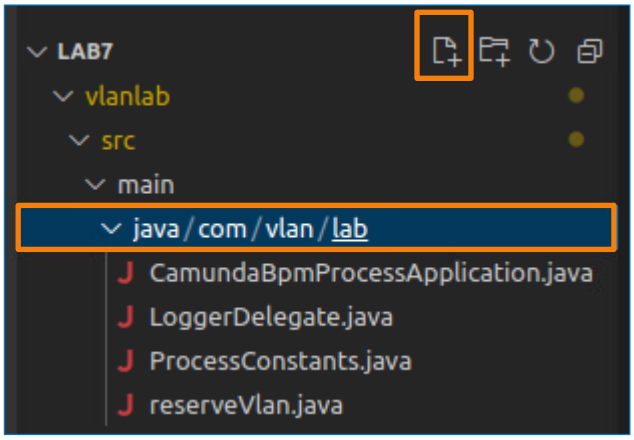
20. The same logic applies to the other possible path of the workflow. To define the other condition, click on the arrow named **False**.



21. This time, enter #{not vlanSet} in the **Condition Expression** field. This indicates that the workflow will proceed this way if the statement is not true.



22. There is only one more task (**success notification**) to configure. This task will implement some code again. Navigate back to VSC and create a new Java file in the same directory.



23. Name the file **successNotification.java** and add the following code. This is now just a definition of the class that does not perform anything yet.

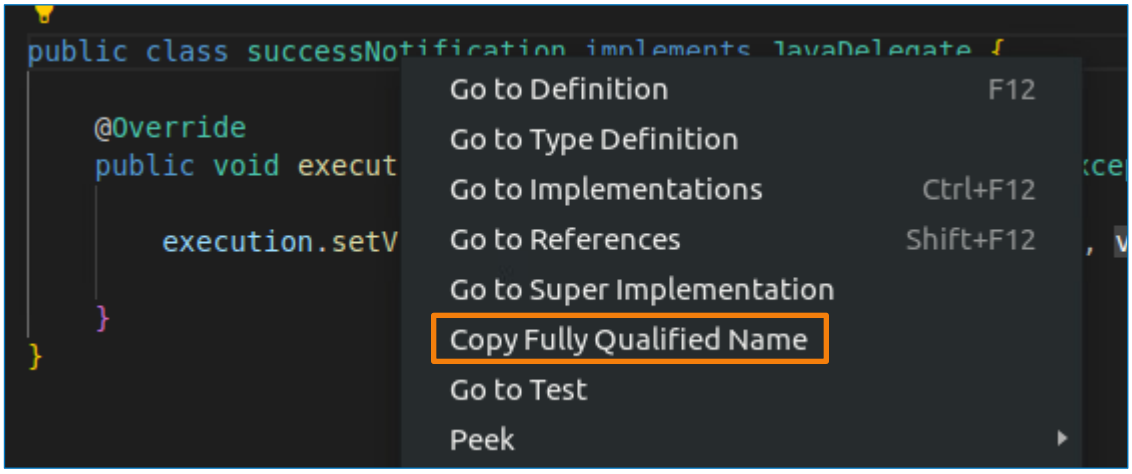
```
package com.vlan.lab;

import org.camunda.bpm.engine.delegate.DelegateExecution;
import org.camunda.bpm.engine.delegate.JavaDelegate;
public class successNotification implements JavaDelegate {

    @Override
    public void execute (DelegateExecution execution) throws Exception {



    }
}
```

24. This class will be used to simply assign a value of **true** to the variable named **reservationStatus**.

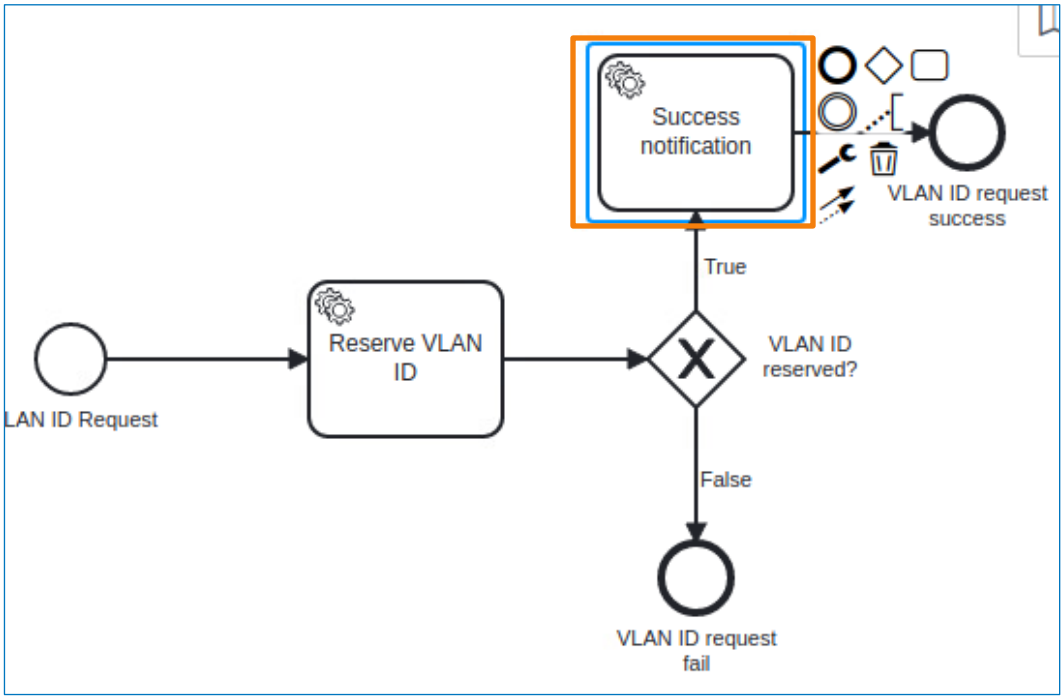
```
package com.vlan.lab;
```

```
import org.camunda.bpm.engine.delegate.DelegateExecution;
import org.camunda.bpm.engine.delegate.JavaDelegate;
public class successNotification implements JavaDelegate {

    @Override
    public void execute (DelegateExecution execution) throws Exception {

        execution.setVariable("reservationStatus", true);

    }
}
```

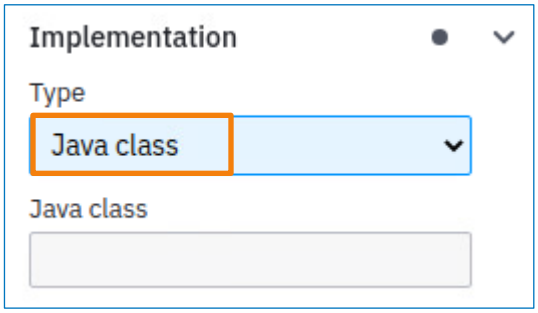
25. Save the file and right-click the class to copy the fully qualified name.



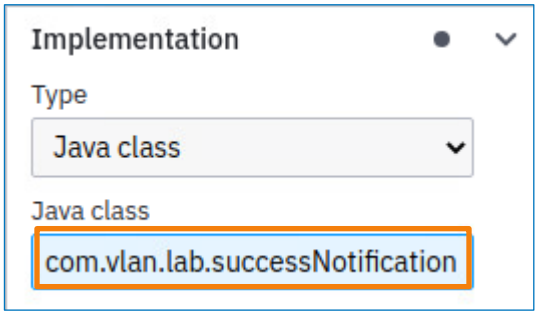
26. Navigate back to the modeler and click on the **Success notification** task.



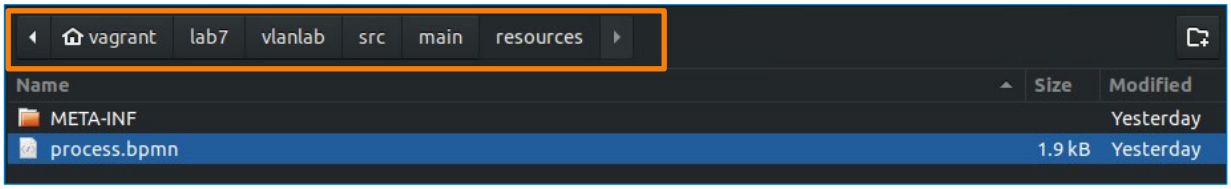
27. Open the implementation section of the properties menu and select **Java class** as the type.



28. Paste the fully qualified name in the **Java class** field.



29. Save the .bpmn file. Again, make sure that the file is saved to the correct location.

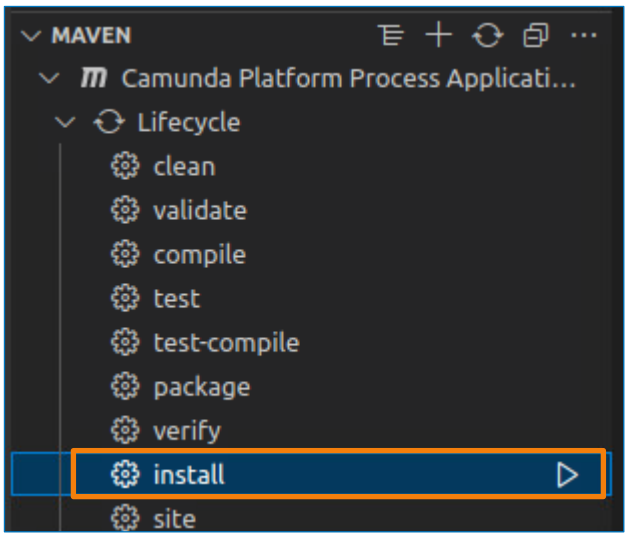


Task 4: Building the Project and Deploying it to Camunda

In the previous task, you developed the code needed for your workflow. You will now build the project using Maven and deploy it to Camunda.

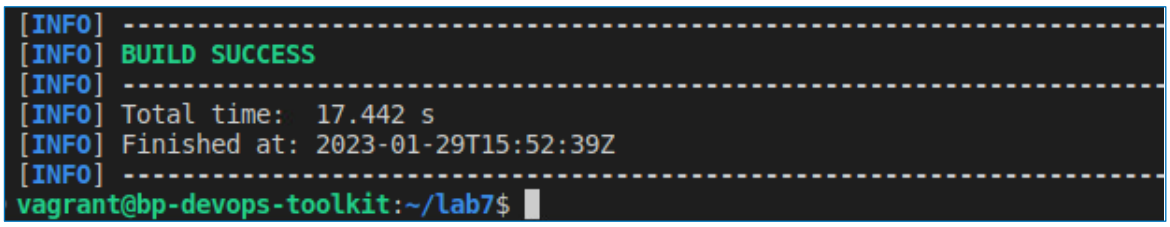
1. You will now build the project with Maven. The instance of VSC that you are using should have the Maven extension installed and you can manage the project through the GUI. In the bottom-left corner, expand the **Maven section** and expand the **lifecycle** commands. Run the **install** command.

NOTE: Instead of using the GUI, you can also run the following command in the terminal window instead: **mvn install -f "/home/vagrant/lab7/vlanlab/pom.xml"**.

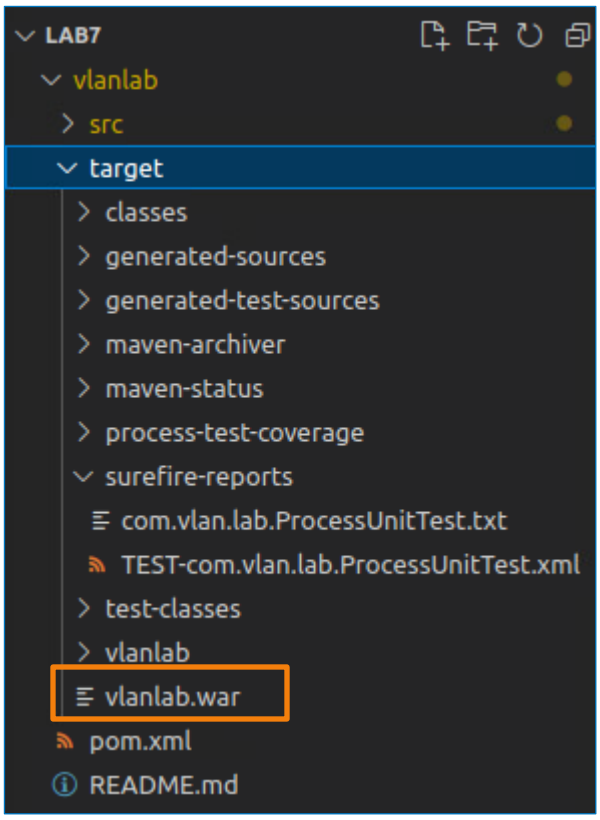


2. The project should be built successfully. If there are any errors, try to compare the code and the process shown in the lab guide with your code and BPMN process.

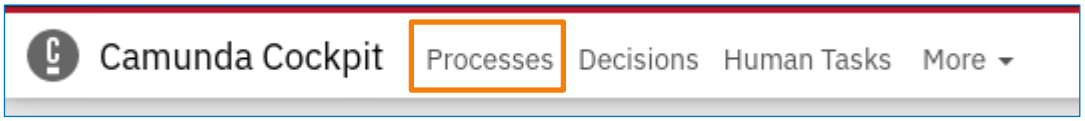
NOTE: If your process does not build successfully, you can also observe the logs in the **/target/surfire-reports** directory.



3. Expand the **/vlanlab/target/** directory. There should now be a **vlanlab.war** file. This is the file that contains all the information, java classes, and models needed to run the process in Camunda.



4. First, make sure that the process is not deployed to Camunda yet. Open the Cockpit application in your browser and click the **Processes** button.



5. Note that the vlan process is not listed under deployed process definitions.

6. You will now deploy your custom process to Camunda by copying the .war file to the correct directory in the Camunda container.

   Open a new terminal window and navigate to the directory that contains the .war file.

```
[~]$ [~]$ cd lab7/vlanlab/target/
```

```
[~/lab7/vlanlab/target]$ ls -l

<... output omitted ...>
drwxrwxr-x 2 vagrant vagrant  4096 Jan 28 15:41 surefire-reports
drwxrwxr-x 3 vagrant vagrant  4096 Jan 29 15:44 test-classes
drwxrwxr-x 5 vagrant vagrant  4096 Jan 28 15:46 vlanlab
-rw-rw-r-- 1 vagrant vagrant 54892 Jan 29 15:52 vlanlab.war
```

7. Copy the .war file to the **webapps** directory of the Camunda container using the **docker cp** command.

```
[~/lab7/vlanlab/target]$ docker cp  vlanlab.war camunda:/camunda/webapps
```

8. Connect to the container shell and check that the file was copied successfully and that the process was deployed to the Camunda Engine.

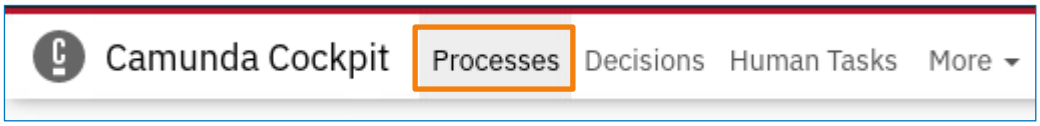
```
[~/lab7/vlanlab/target]$ docker exec -it camunda bash
bash-5.1$
```

9. Navigate to the **webapps** directory of the container and list the contents of the directory.

```
bash-5.1$ cd webapps/
bash-5.1$ ls -l
drwxr-x---    3 camunda  camunda      4096 Oct  4 08:06 ROOT
drwxr-xr-x    7 camunda  camunda      4096 Oct  4 08:06 camunda
drwxr-xr-x    6 camunda  camunda      4096 Oct  4 08:06 camunda-invoice
drwxr-xr-x    4 camunda  camunda      4096 Oct  4 07:11 camunda-welcome
<...output omitted...>
drwxr-x---    5 camunda  camunda      4096 Jan 30 09:23 vlanlab
-rw-rw-r--    1 camunda  camunda     54892 Jan 29 15:52 vlanlab.war
```

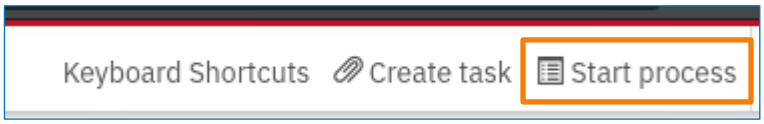
10. Go back to the Camunda web user interface and open the Cockpit application and open the list of process definitions.

NOTE: If you left the Camunda Cockpit application opened you should refresh it to see the newly added process.

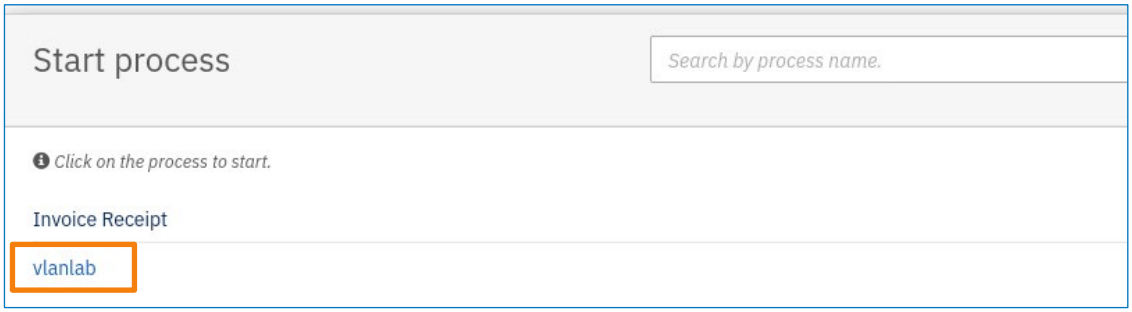


11. Make sure that the process is now available under the list of processes.

12. To start the process, go to the **Tasklist** application and click the **Start process** button.

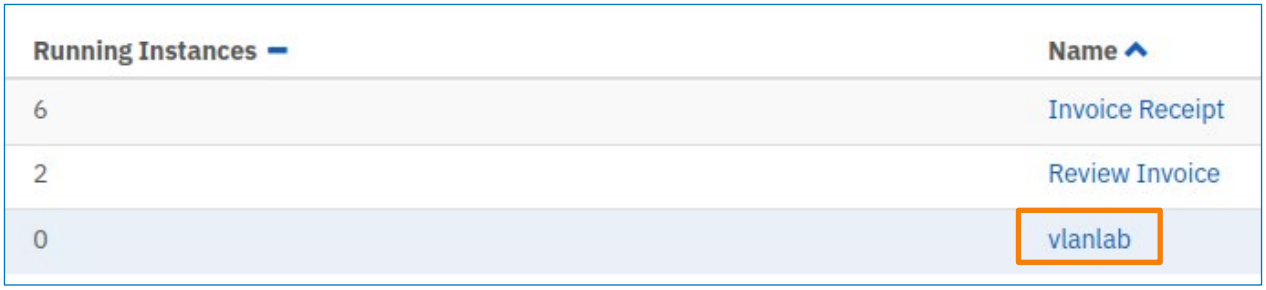


13. Choose your process from the pop-up menu and click **Start**.

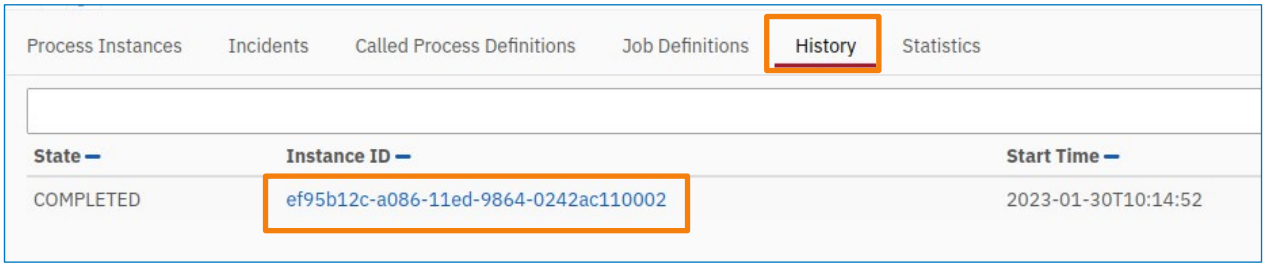


14. You should see a pop-up message saying that the process was started successfully. To explore the execution process, go back to the Camunda Cockpit and open the deployed processes.

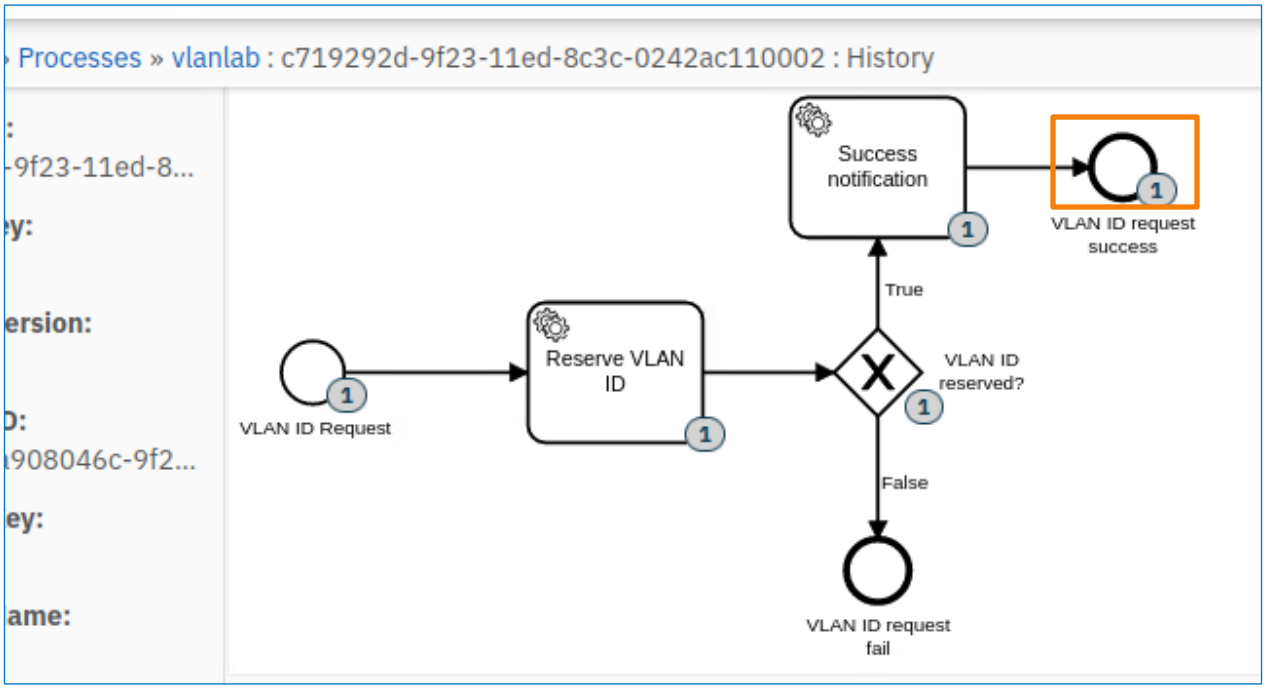
15. Select your **vlanlab** process from the list to find more information about the process.



16. Open the **History** tab and select the only available instance.

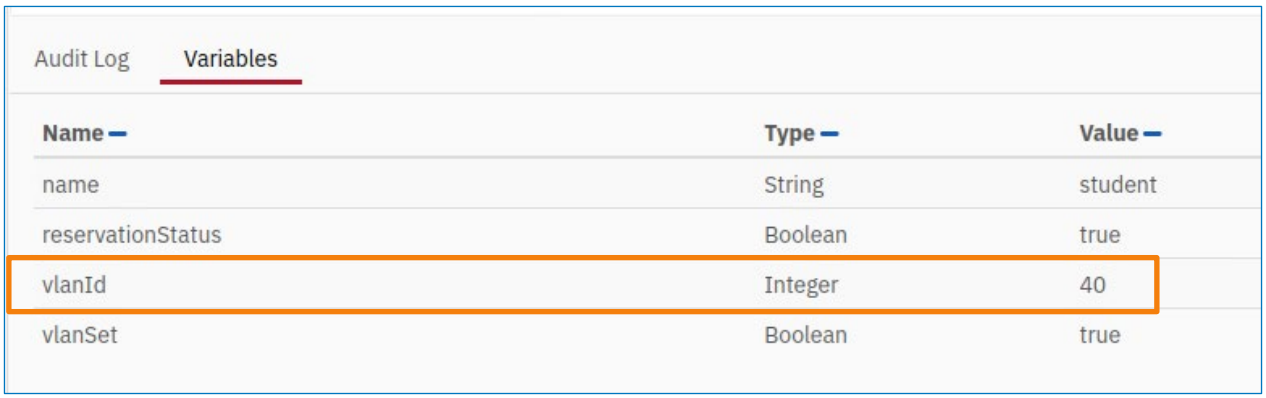


17. The history view should open. You can now see that the process ended on the “VLAN ID request success“ event meaning that the VLAN ID was reserved successfully. Otherwise, the gateway would select the other way and the process would have ended on the “VLAN ID request fail”.



18. To check which of the available IDs was reserved, open the **Variables** tab and find the **vlanId** variable. As you can see one of the IDs that you defined in the list of available IDs was selected.

NOTE: Since the ID is chosen from the list randomly, it is possible that in your case, one of the other available IDs was selected (10, 20, or 30).



End of Lab