

va



a division of Ciena



## Introduction to Networking and Programmability

BPF002ILT, Revision 1.0.3

### **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](http://www.ciena.com).

## **Copyright® 2023 Ciena® Corporation – All Rights Reserved**

The material contained in this document is also protected by copyright laws of the United States of America and other countries. It may not be reproduced or distributed in any form by any means, altered in any fashion, or stored in a database or retrieval system, without the express written permission of Ciena Corporation.

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

For additional office locations and phone numbers, please visit the Ciena website at [www.ciena.com](http://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.
	1.0.2	May 2023	Minor content update.
	1.0.3	June 2023	Minor content update.

# Contents

Lab 1: Prepare the Working Environment .....	5
Lab 2: Manage Device Configurations with NETCONF/YANG .....	18
Lab 3: Using REST APIs and RESTCONF .....	39
Lab 4: Using REST APIs with Java .....	63
Lab 6: Camunda Introduction.....	88
Lab 7: Camunda Workflow for VLAN ID .....	107

## Lab 1: Prepare the Working Environment

### Objectives

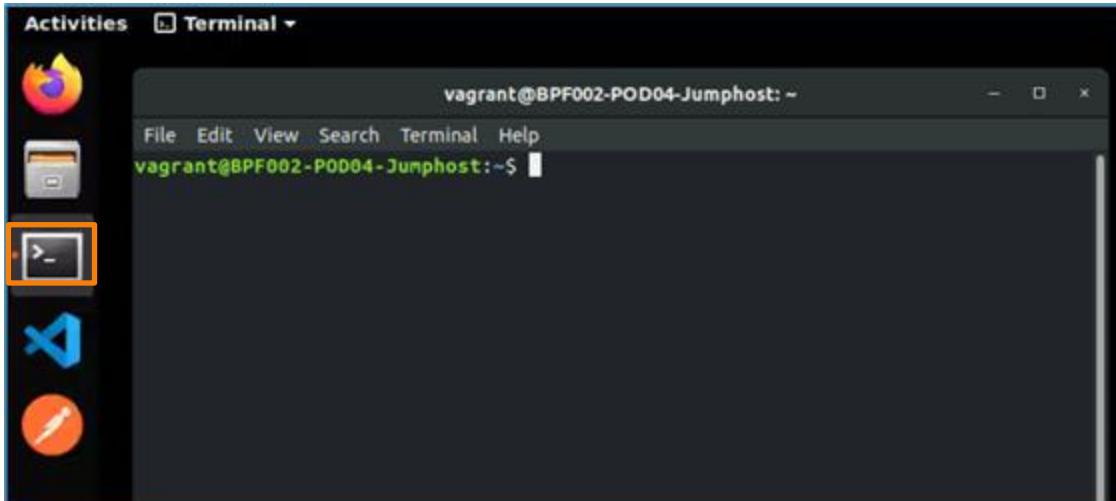
- Setting up the development environment with the Git version control system
- Get to know the Microsoft Visual Studio Code (VS Code) editor and install extensions

### Task 1: Prepare the Development Environment with Visual Studio Code and Git Version Control

Your lab environment includes a development machine with all the tools that you will need to perform the tasks in the following labs, including Visual Studio Code, git, java and maven. The lab environment also includes a shared GitLab server that will be used to host git repositories for all the students.

In this lab you will prepare a git repository that will store the files for some of the following labs. You will explore a basic git workflow, including both local and remote repository and learn how Visual Studio Code can be used in the development process.

1. Open a new terminal window on your development machine by clicking the terminal icon from the desktop.



2. Make sure that java is installed on your machine by typing the **java -version** command in the terminal.

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

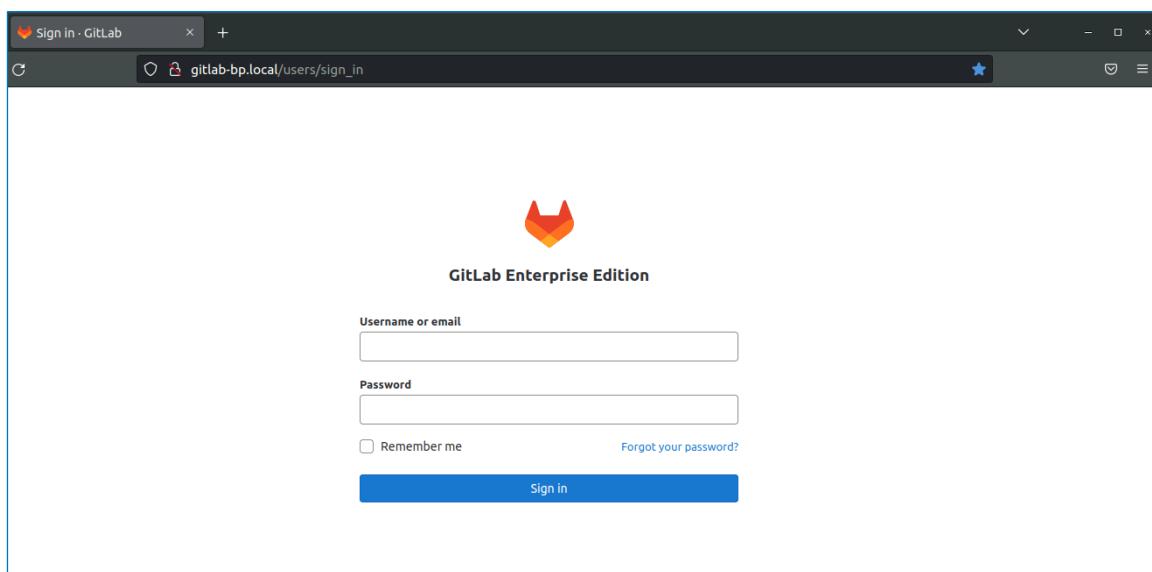
3. Make sure that maven is installed on your machine by typing the **mvn -version** command in the terminal.

```
$:~/<pod-name>-bpi-training$ mvn -version
```

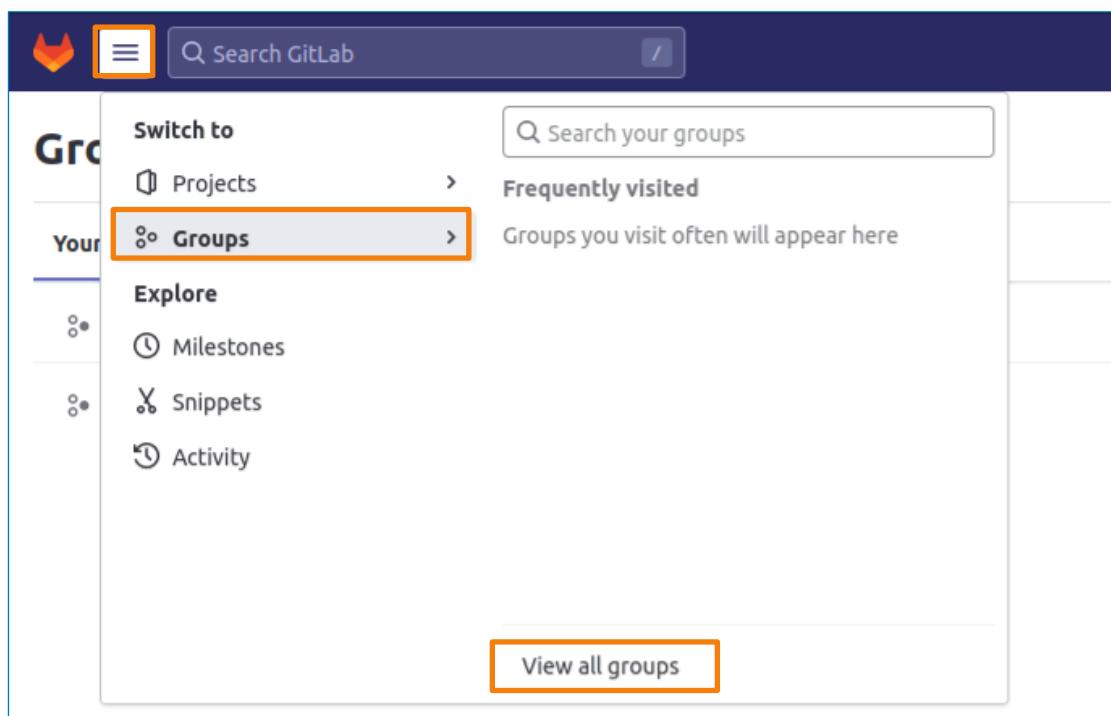
```
Apache Maven 3.8.6 (84538c9988a25aec085021c365c560670ad80f63)
Maven home: /opt/apache-maven-3.8.6
Java version: 11.0.15.1, vendor: Oracle Corporation, runtime:
/usr/lib/jvm/jdk-11
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "4.15.0-201-generic", arch: "amd64", family:
"unix"
```

4. Open the Firefox browser by clicking the icon on the desktop and navigate to <http://gitlab-bp.local/> or click the GitLab bookmark. Sign in using the following credentials:

- Username: **studentXX** (XX = pod number, for example student01)
- Password: **12345678**

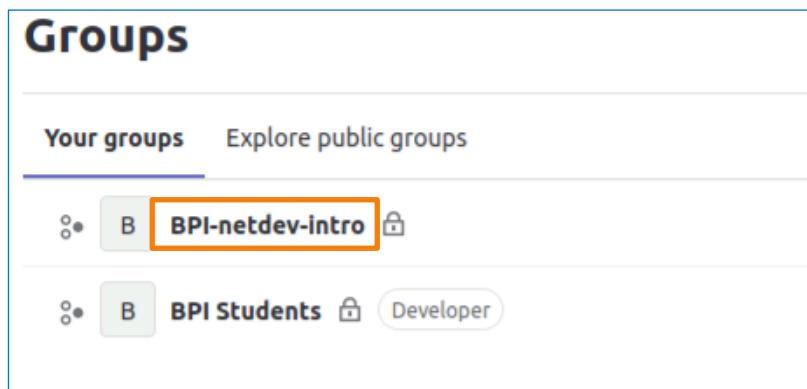


5. Find the group named **BPI-netdev-intro** in the **Groups > View all groups** menu.



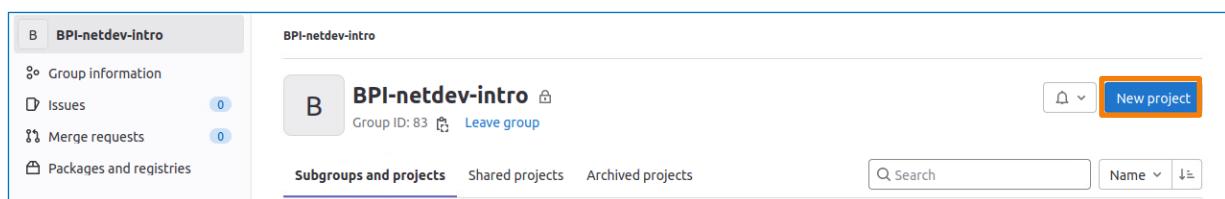
The screenshot shows the GitLab interface with the 'Groups' menu open. The 'Groups' option is highlighted with an orange box. A 'View all groups' button at the bottom right of the menu is also highlighted with an orange box.

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



The screenshot shows the 'Groups' page with the 'Your groups' tab selected. The 'BPI-netdev-intro' group is listed, with its name highlighted by an orange box.

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



The screenshot shows the 'BPI-netdev-intro' group page. In the top right corner, there is a 'New project' button, which is highlighted with an orange box.

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

The screenshot shows a 'Create new project' interface with three main options:

- Create blank project**: Create a blank project to store your files, plan your work, and collaborate on code, among other things.
- Create from template**: Create a project pre-populated with the necessary files to get you started quickly.
- Import project**: Migrate your data from an external source like GitHub, Bitbucket, or another instance of GitLab.

At the bottom, it says "You can also create a project from the command line. Show command".

9. As the project name type in **podXX-bpi-training** (XX = pod number, for example **pod01-bpi-training**), leave all other fields on the default setting, and then click **Create project**. You should see a successfully created project with a default README.md file.

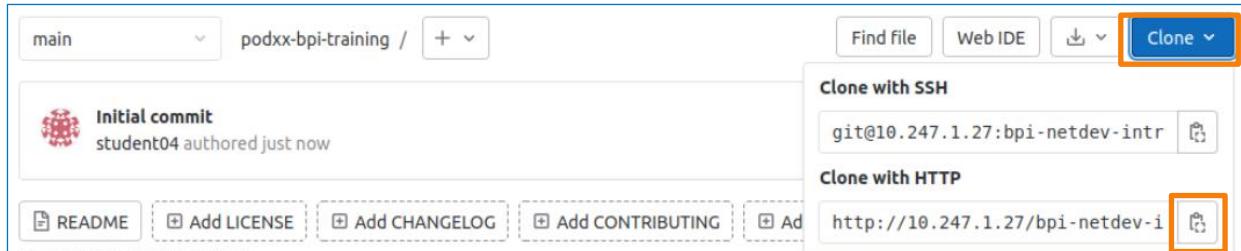
The screenshot shows a GitLab project page for 'podXX-bpi-training'. The top bar indicates 'Project 'podXX-bpi-training' was successfully created.' The project summary includes:

- Project ID: 287
- 1 Commit
- 1 Branch
- 0 Tags
- 61 KB Project Storage

The repository navigation bar shows 'main' and 'podxx-bpi-training / +'. Action buttons include 'Find file', 'Web IDE', 'Clone', and a download icon. A commit card for 'Initial commit' by 'student04' is shown with a timestamp and a copy link. Below are buttons for README, LICENSE, CHANGELOG, CONTRIBUTING, Kubernetes cluster, CI/CD setup, and integration configuration. A table at the bottom lists the README.md file with its last commit information.

Name	Last commit	Last update
README.md	Initial commit	just now

10. You will now clone the git repository to your student VM. Click the clone button and copy the URL of the repository by clicking the **Copy URL** button next to the **Clone with HTTP** option. You will use this URL in the next step to clone the git repository.



11. Navigate back to the terminal window (or open a new one if you closed it) and clone the GitLab repository to your Student VM using the **git clone <URL>** command. Use the URL that you copied in the previous step.

Enter the GitLab username and password when prompted.

```
$:~/ $ git clone http://10.247.1.27/bpi-netdev-intro/podXX-bpi-training.git
Cloning into 'podXX-bpi-training'...
Username for 'http://10.247.1.27': studentXX
Password for 'http://student02@10.247.1.27': password
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused
$:~/<pod-name>-bpicd pod -training$
Receiving objects: 100% (3/3), done.
```

12. You have now cloned the project to your local development environment. Enter the cloned directory and observe the contents. You should see the README.md file that was created by default when you created the project on GitLab.

```
$ cd <pod name>-bpi-training
$:~/<pod name>-bpi-training$ ls
README.md
$:~/<pod-name>-bpi-training$
```

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

**NOTE:** The email will only be used for working with this git repository. You can come up with any example email.

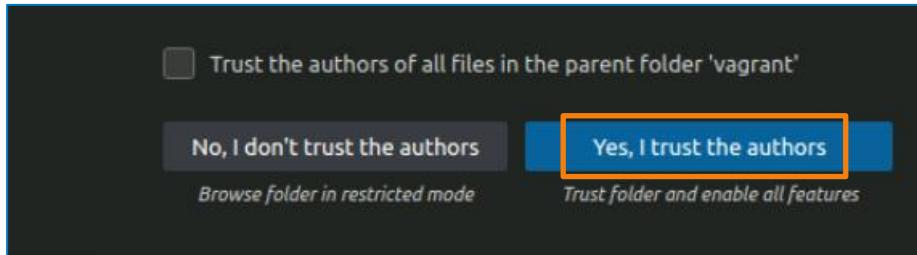
```
$:~/<pod name>-bpi-training$ git config --global user.email
you@example.com
$:~/<pod-name>-bpi-training$ git config --global user.name "Your Name"
```

14. Open the directory in Visual Studio Code using the **code .** command.

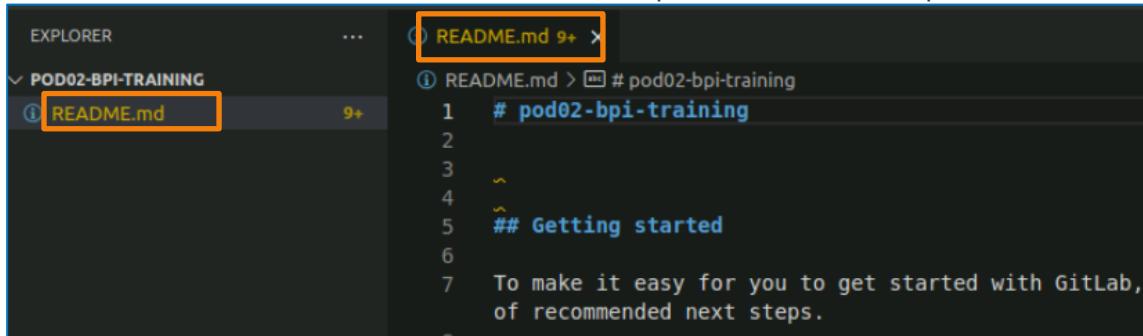
**NOTE:** Make sure that you also include the dot (".") in the command as this instructs VSC to open the current working directory.

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

15. Choose **Yes, I trust the authors** from the pop-up menu in VSC.



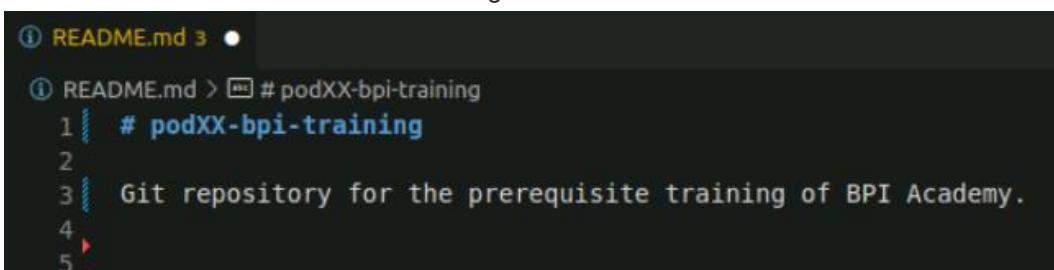
16. Double-click the README.md file in the side menu to open it in the text editor pane.



17. Delete all the contents of the README file then add the following lines. (Replace XX with your pod number.)

```
# podXX-bpi-training  
  
Git repository for the prerequisite training of BPI Academy
```

18. Your file should now look like the following.



19. Save the file by navigating to **File > Save** or by pressing **Ctrl + S**.
20. Navigate back to the terminal window and execute the **git status** command. This will display the state of the working directory and staging area. You should see that some changes have been done on the README.md file since the previous commit.

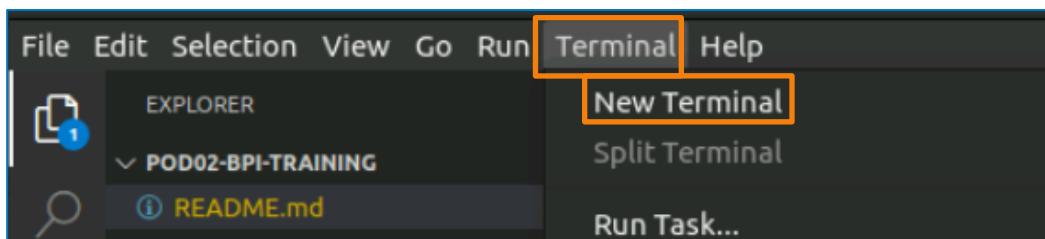
```
$:~/<pod-name>-bpi-training$ git status
```

```
On branch main
Your branch is up to date with 'origin/main'.

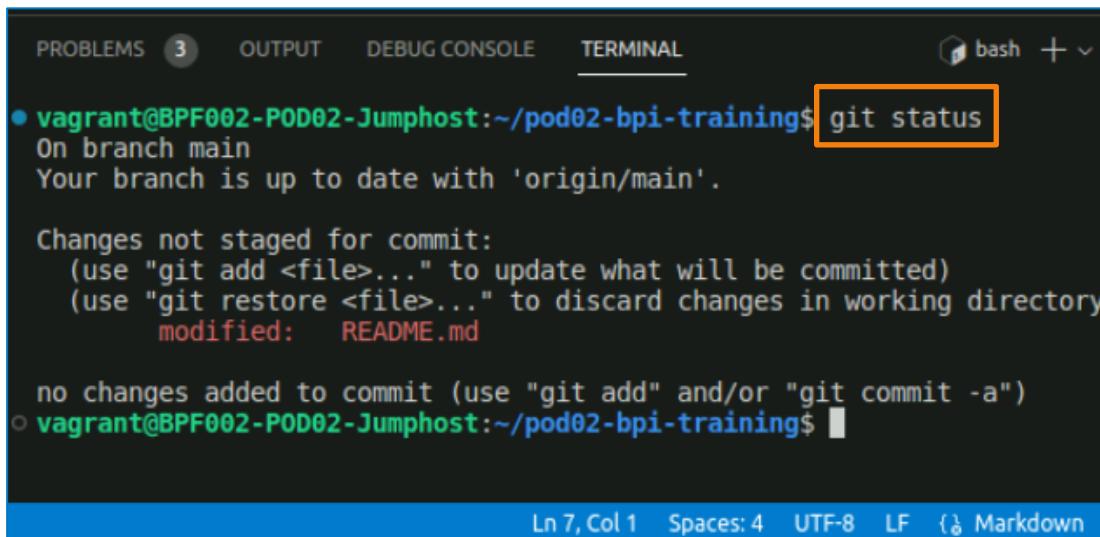
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
$:/<pod-name>-bpi-training$
```

21. Instead of accessing the terminal in a separate window you can also open the terminal directly in VSC. Navigate to **Terminal > New Terminal** to open a terminal window in VSC UI.



22. You should now see a terminal that opened directly in VSC. Execute the same **git status** command from this terminal window. You should see the same output.



```
● vagrant@BPF002-POD02-Jumphost:~/pod02-bpi-training$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
○ vagrant@BPF002-POD02-Jumphost:~/pod02-bpi-training$
```

23. Try to commit the changes that you performed with **git commit** command.

```
$:/<pod-name>-bpi-training$ git commit
On branch main
Your branch is up to date with 'origin/main'.
```

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
$:/<pod-name>-bpi-training$
```

Only the files that are added to the staging area first can be committed. In this case you didn't yet add the README.md file to staging area that's why nothing was committed.

24. Add the README.md file to the staging area with **git add** command.

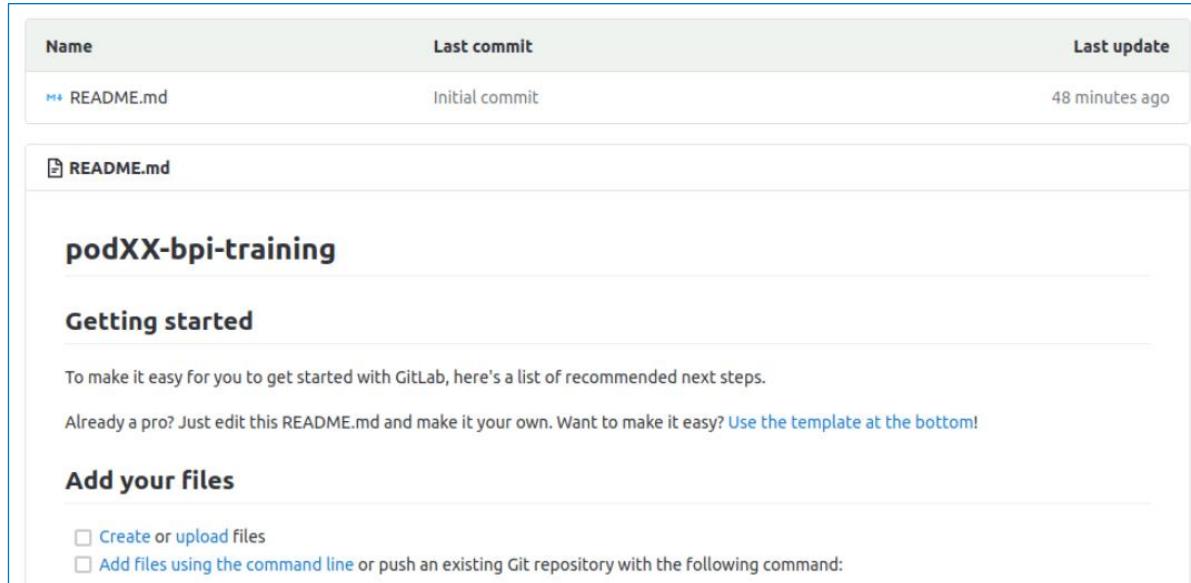
```
$:/<pod-name>-bpi-training$ git add README.md
$:/<pod-name>-bpi-training$
```

25. Commit the staged changes. Use the -m flag to add the commit message.

```
$:/<pod-name>-bpi-training$ git commit -m "Changed README.md"
[main 6ef9963] changed README.md
 1 file changed, 1 insertion(+), 90 deletions(-)
$:/<pod-name>-bpi-training$
```

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 repository. In this case the central git repository is hosted on a GitLab server in your lab environment.

26. Before pushing the changes, navigate back to GitLab in your browser and refresh the window to make sure that nothing has changed on the remote repository yet.



Name	Last commit	Last update
 README.md	Initial commit	48 minutes ago

**podXX-bpi-training**

### Getting started

To make it easy for you to get started with GitLab, here's a list of recommended next steps.

Already a pro? Just edit this README.md and make it your own. Want to make it easy? [Use the template at the bottom!](#)

### Add your files

[Create or upload files](#)  
 [Add files using the command line](#) or push an existing Git repository with the following command:

You can see that the README.md file is still not changed in the repository stored on GitLab.

27. Push the changes to remote repository using **git push** command. Type in the GitLab credentials when prompted.

**NOTE:** If you are using the terminal in VSCode, make sure that you enter the username and pop-up menu at the top of the UI when prompted.

```
$:~/<pod-name>-bpi-training$ git push
Username for 'http://10.247.1.27': studentXX
Password for 'http://student04@10.247.1.27': password
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 321 bytes | 321.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To http://10.247.1.27/bpi-netdev-intro/podxx-bpi-training.git
    7768715..6ef9963 main -> main
$:~/<pod-name>-bpi-training$
```

28. Navigate back to GitLab and refresh the browser window again. Your README.md file should now include all your changes.

The screenshot shows a GitLab repository page. At the top, there is a commit message: "changed README.md" by "vagrant" 13 minutes ago, with a commit hash "6ef99634". Below the commit message are several buttons: "README", "Add LICENSE", "Add CHANGELOG", "Add CONTRIBUTING", "Enable Auto DevOps", "Add Kubernetes cluster", "Set up CI/CD", and "Configure Integrations". A table below shows the file "README.md" with its last commit information: "Name" (README.md), "Last commit" (changed README.md), and "Last update" (13 minutes ago). The "README.md" file content is displayed in a box with an orange border, showing the text "podXX-bpi-training" and "Git repository for the prerequisite training of BPI Academy".

Your git repository now includes your custom README.md file. Now you will also add some additional files to this repository. Those files are currently stored in a temporary directory on your Student VM and include some skeleton code that is going to be used in the following labs.

29. Navigate back to the terminal window and copy the skeleton code from a temporary directory /lab1/ to the git repository that you created.

**NOTE:** Make sure that you are in the git repository (podXX-bpi-training) when you issue the command. Also, make sure that you include the dot “.” in the command.

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

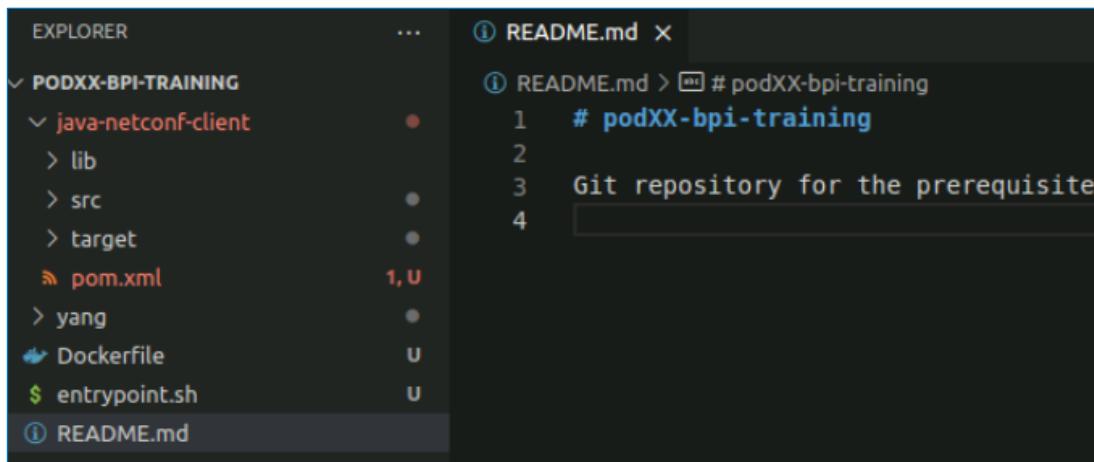
30. List the contents of the repository and observe the files.

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

Notice that there are now some additional files besides README.md. The files that you added to your repository are going to be used in the following lab.

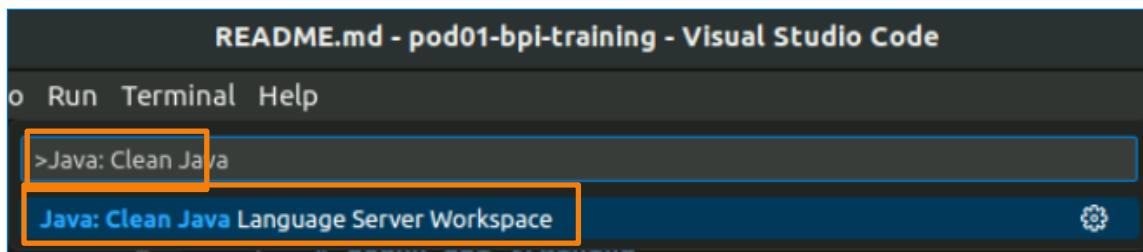
31. Navigate back to VSCode and observe the repository contents.

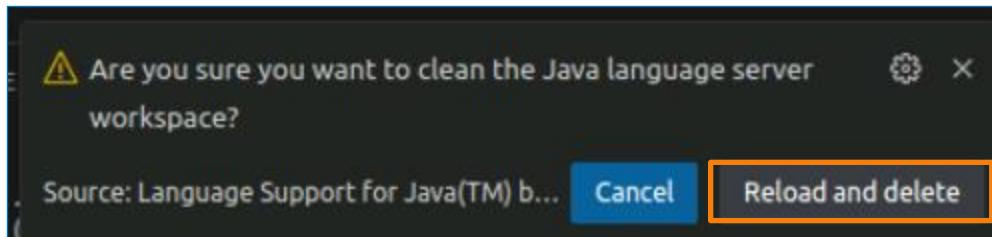
**NOTE:** If you closed the VSCode window, open it again using the **code .** command.



Note that there are some problems with one of the files. This is because one of the dependencies specified in the pom.xml file is not present in the official Maven repository. The library has to be downloaded and built manually. You will solve this problem in the next lab.

32. If VSCode doesn't indicate any problems with pom.xml file, 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”.





33. Because you added new files to your repository since the last commit you should commit the changes. First use the **git status** command again to check the state of the repository.

```
$:~/<pod-name>-bpi-training$ git status
```

On branch main

Your branch is up to date with 'origin/main'.

Untracked files:

(use "git add <file>..." to include in what will be committed)

Dockerfile

entrypoint.sh

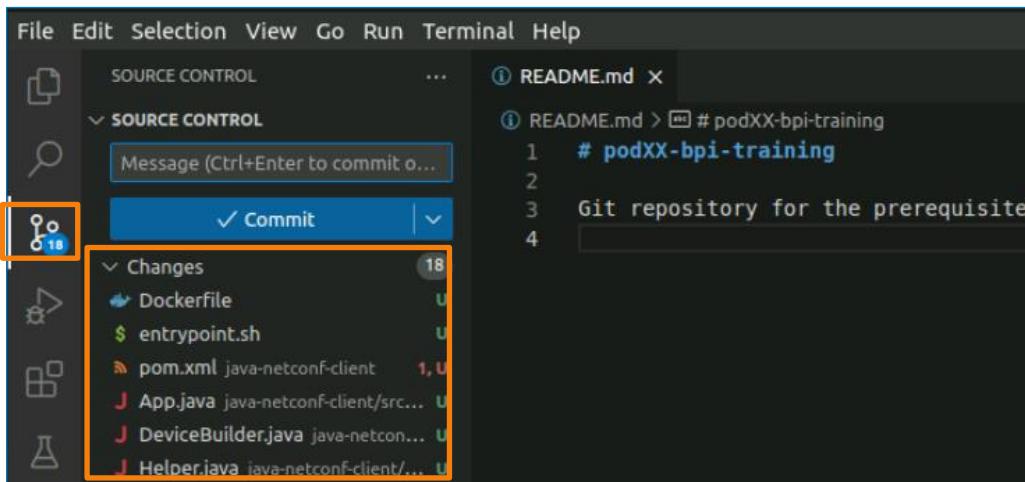
java-netconf-client/

yang/

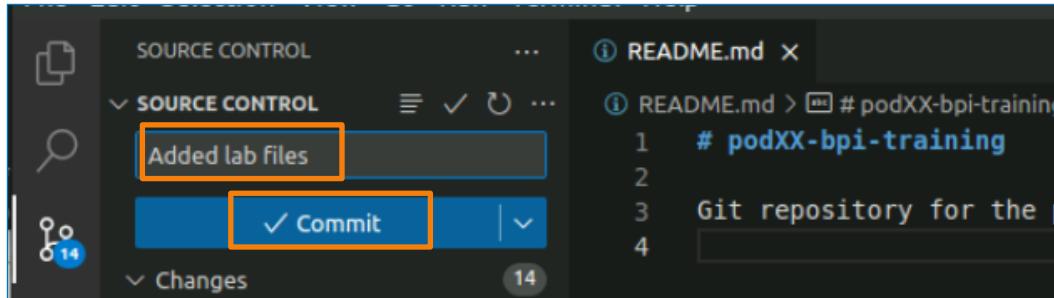
nothing added to commit but untracked files present (use "git add" to track)

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

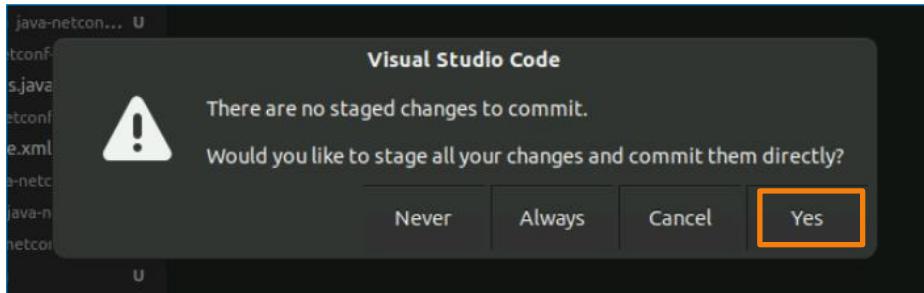
34. You can also use VSCode graphical interface for git to check the state of the repository. Click on the **Source Control** tab in the left side menu and observe all the displayed changes.



35. Commit the changes by entering a commit message in the **Message** tab and clicking the **Commit** button. For the commit message use "**Added lab files**".



36. Since the changed files were not added to staging area before committing, VSCode will ask you if all the files must be staged first. Choose **Yes** in the pop-up menu.



In this case we used VSCode git integration to commit the changes instead of typing git commands in the terminal.

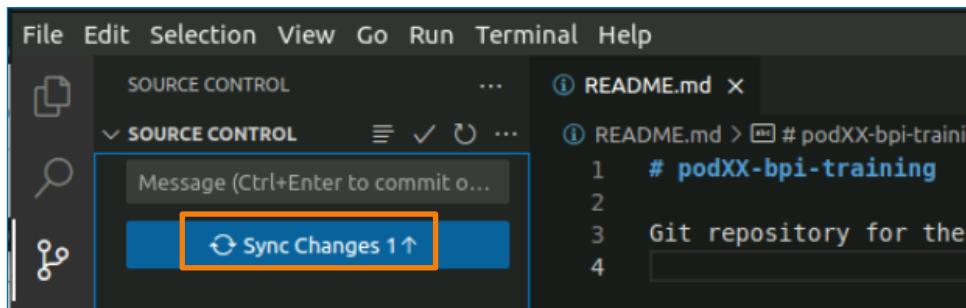
37. Check the status of the repository again.

```
$:~/<pod-name>-bpi-training$ git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)

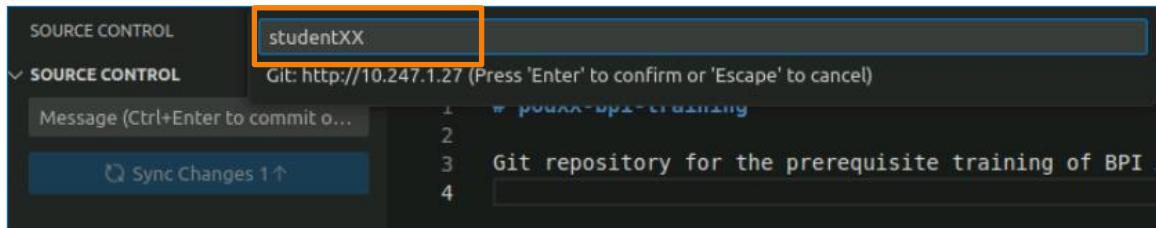
nothing to commit, working tree clean
```

Your changes are now safely stored in this commit but are not yet pushed to the central repository on GitLab.

38. Push the changes to GitLab by clicking the **Sync changes** button in VSCode. Choose **Ok** in the pop-up menu explaining the action.



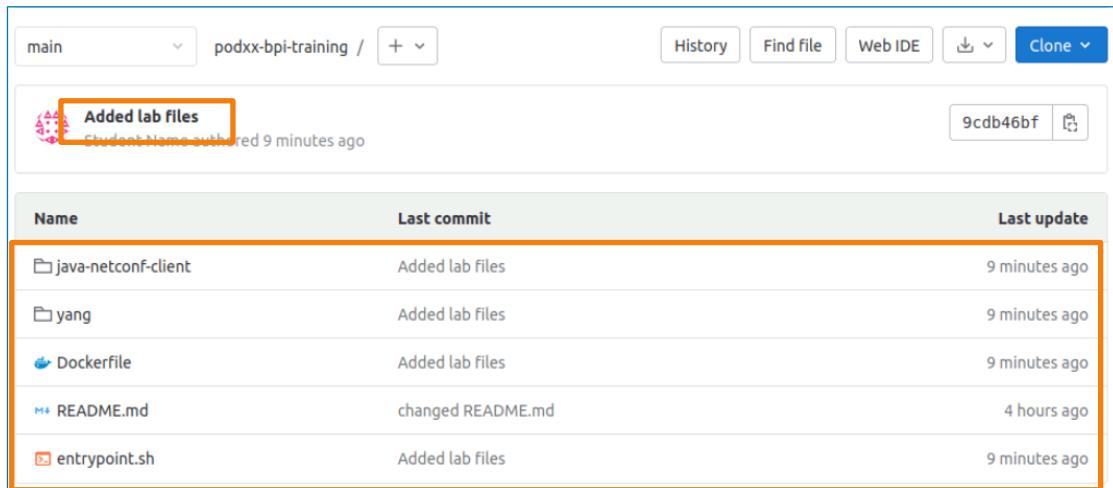
39. Enter the GitLab username and password when prompted.



A screenshot of a terminal window titled "SOURCE CONTROL". The title bar has the text "studentXX" highlighted with an orange box. Below the title bar, the URL "Git: http://10.247.1.27" is displayed. A message box says "Message (Ctrl+Enter to commit o...)" with a "Sync Changes 1↑" button. The main area shows a commit log with four entries:

```
1  # podxx-bpi-training
2
3 3 Git repository for the prerequisite training of BPI A
4
```

40. Navigate back to GitLab, refresh the browser window, and review the state of the repository. The files should match your local repository.



A screenshot of a GitLab repository page for "podxx-bpi-training". The top navigation bar shows "main" and "podxx-bpi-training /". On the right, there are buttons for "History", "Find file", "Web IDE", and "Clone". A "Clone" button is highlighted with an orange box. Below the navigation, a commit message "Added lab files" is shown, with the author "Student Name" and timestamp "9 minutes ago". The commit hash "9cdb46bf" is also visible. The main content area displays a table of files with their last commit details:

Name	Last commit	Last update
java-netconf-client	Added lab files	9 minutes ago
yang	Added lab files	9 minutes ago
Dockerfile	Added lab files	9 minutes ago
README.md	changed README.md	4 hours ago
entrypoint.sh	Added lab files	9 minutes ago

---

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

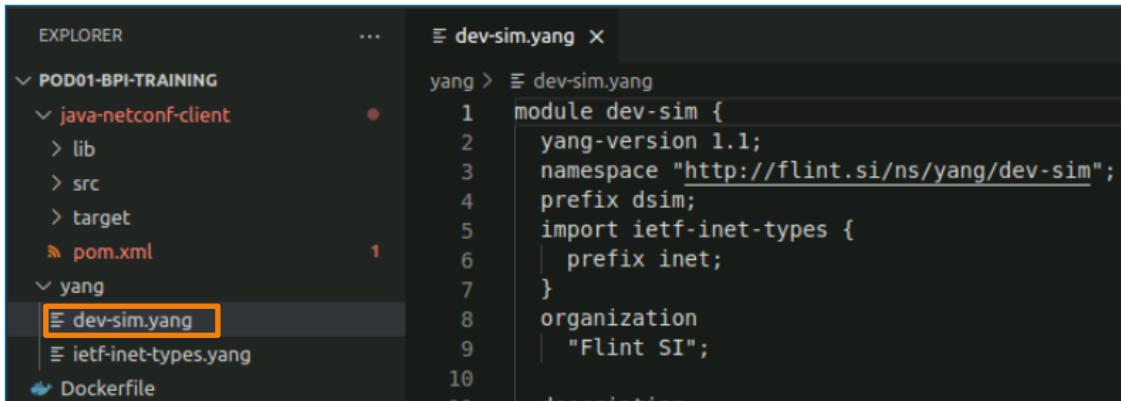
### Task 1: Prepare loopback interface YANG module

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

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 by double-clicking it.



```
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";
```

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
  "YANG module for interfaces";

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

//type definition
/*
typedef shutdown-option {
  type enumeration {
    enum "yes";
    enum "no";
  }
}
*/
// interfaces container with a list of loopbacks
/*
container interfaces {
  list loopback {
    description
    "Loopback interface";
    key "name";
    leaf name {
      type uint32 {
        range "0..2147483647";
      }
    }
    uses interface-common;
  }
}
*/
*/
```

```
// interface common parameters
/*
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;
    }
}
*/
}
```

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.

Key parts of the model are commented out for now. You will uncomment them in the following steps to implement the elements that describe the loopback interfaces of a networking device.

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

3. Uncomment the **interfaces** container by deleting the “/\*” and “\*/” before and after the container statement. This will implement a container with a list of loopbacks that uses the name parameter as the key.

```
<... output omitted ...>
// interfaces container with a list of loopbacks
```

```

/*
container interfaces {
    list loopback {
        description
        "Loopback interface";
        key "name";
        leaf name {
            type uint32 {
                range "0..2147483647";
            }
        }
        uses interface-common;
    }
}
*/

```

A loopback interface usually allows more than just the configuration of its name. Since different types of interfaces (loopback, ethernet, ...) usually have some common parameters (IP, description, status) you will create an **interface-common grouping** element describing those elements outside of the **list loopback** and then use the created **interface-common grouping** in the list **interfaces** with the **uses** keyword. This approach helps you avoid duplication in the YANG model.

4. Uncomment the **interfaces-common** grouping to implement the IP address, description, and shutdown parameters.

```

<... output omitted ...>
// interface common parameters
/*
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 by uncommenting the following code.

```

<... output omitted ...>

description
"YANG module for interfaces";
}

//type definition
/*
typedef shutdown-option {
    type enumeration {
        enum "yes";
        enum "no";
    }
}
*/
```

&lt;... output omitted ...&gt;

6. 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;
    }
}
}
```

7. Save the **dev-sim.yang** file by navigating to **File > Save** or pressing **Ctrl + S**.
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.

```
:~/dev-sim$ 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 use your NETCONF client to configure the device.

4. Inspect the initial device configuration.

```
:~/dev-sim$ 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  
:~/<pod 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.

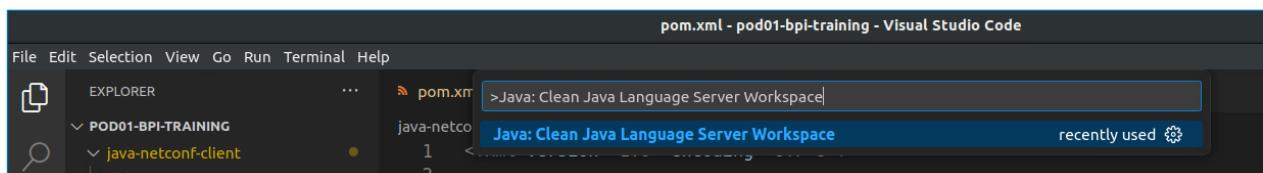
```
:[~/<pod name>-bpi-training/java-netconf-client$ mvn install -f  
lib/netconf-client/pom.xml  
[INFO] Scanning for projects...  
<... output omitted ...>  
[INFO] -----  
[INFO] BUILD SUCCESS  
[INFO] -----
```

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

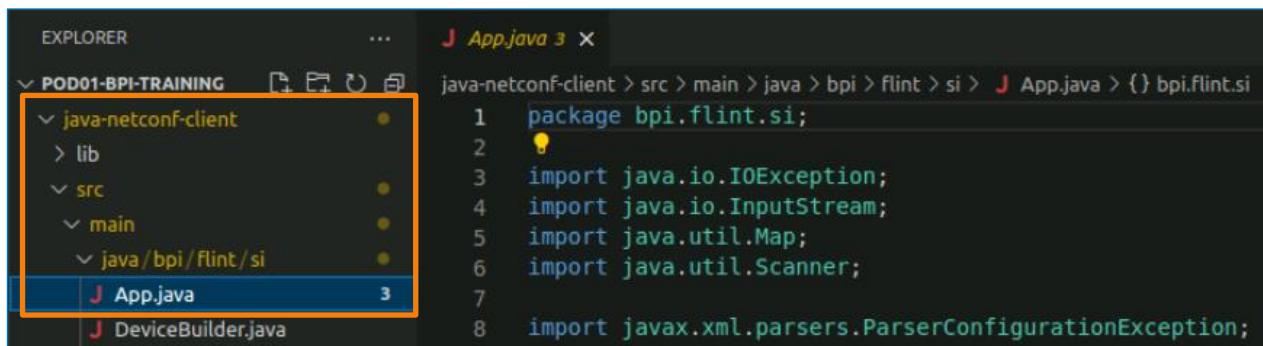
4. Install all dependencies in the **java-netconf-client** project.

```
$:~/<pod-name>-bpi-training/java-netconf-client$ 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. Open the **App** class in the **src/main/java/\*/App.java** path.



7. Observe the code. The **DeviceConfig** method reads the device access information.

```
<... 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.

8. In the **main** method, uncomment the highlighted lines to create a new NETCONF Device object using the device configuration and the NETCONF library. Uncomment the code by removing the “//” at the beginning of the highlighted lines.

```
<... 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 ...>
```

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

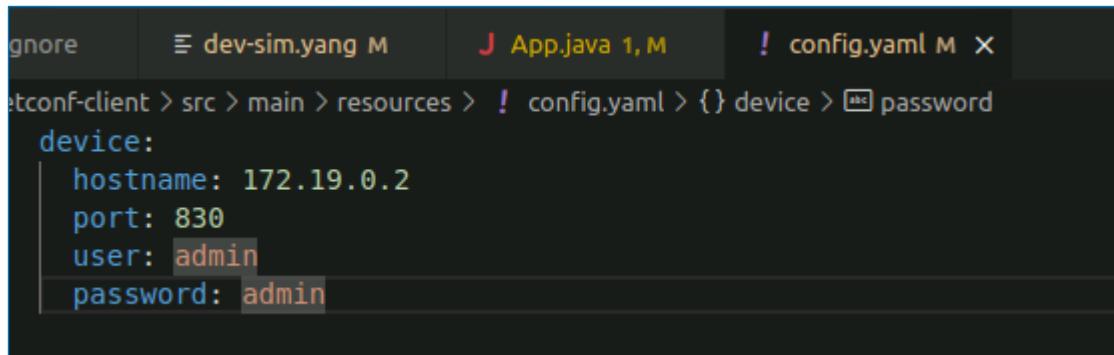
As input, the **Create** method receives the configuration for a device, reads in all the required parameters, and attempts to connect to the device. The device object returned by the method supports various NETCONF operations that you will use in the following steps.

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. Execute the following command from the terminal to get the address.

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



```

gnore      ⌂ dev-sim.yang M    J App.java 1,M    ! config.yaml M X
netconf-client > src > main > resources > ! config.yaml > {} device > password
device:
  hostname: 172.19.0.2
  port: 830
  user: admin
  password: admin

```

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

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

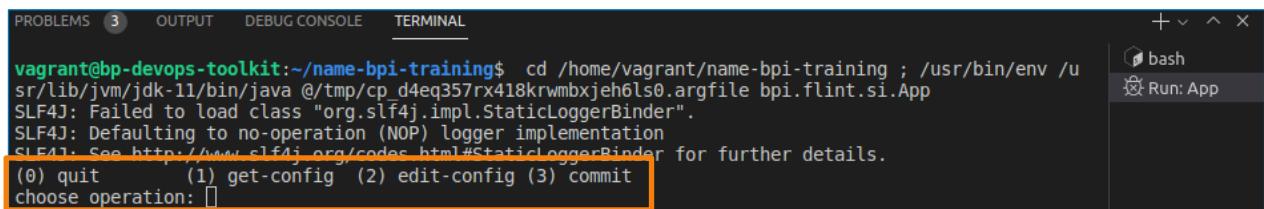


```

Run | Debug
public static void main(String args[]) throws NetconfException,
    |           ParserConfigurationException, SAXException, IOException {
    |           // read device access config
    Map<Object, Object> config = new App().DeviceConfig();
}

```

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.



PROBLEMS    3    OUTPUT    DEBUG CONSOLE    TERMINAL

```

vagrant@bp-devops-toolkit:~/name-bpi-training$ cd /home/vagrant/name-bpi-training ; /usr/bin/env /u
sr/lib/jvm/jdk-11/bin/java @/tmp/cp_d4eq357rx418krwmbxjeh6ls0.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: 

```

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 a simple get-config operation with the XML payload and Java methods.

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

```
<get-config>
    <source>
        <running/>
    </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 running) node is added. 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.

2. Open the **NetconfCommands.java** file and implement the **getConfig** method by uncommenting the following lines.

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

The method takes two arguments as input: the Device object containing the connection to the device, and the path to the XML payload used in the request.

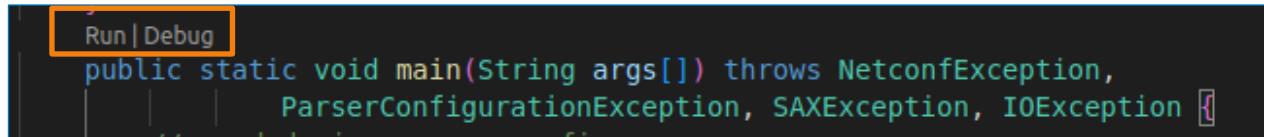
When the XML message is prepared, it is sent to the device as a NETCONF message using the **executeRPC** method maintained in the NETCONF client library.

3. Save the file.
4. Open the **App.java** file, and in the main method, implement the user query prompt for the get-config operation by uncommenting the following line.

```
<... output omitted ...>
Device device = DeviceBuilder.Create(config);
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) {
        //getConfig(device, getConfigPayload);
    }
    else if(command == 2) {
        <... output omitted ...>
```

5. The static method **getConfig** is imported into App.java, and you can use it directly in the main method. Save the file **App.java** file and run the application from VS Code.



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

A screenshot of the VS Code integrated terminal. The terminal window shows the following output:

```
SLF4J: No SLF4J providers were found.
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See https://www.slf4j.org/codes.html#noProviders for further details.
(0) quit      (1) get-config  (2) edit-config (3) commit
choose operation: 1
<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>
```

The get-config operation returns the entire running datastore. By default, a loopback 0 is 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. Open the configure-interface.xml file located in the src/main/resources/payloads path.

```
<edit-config>
  <target>
    <candidate/>
  </target>
  <config xmlns:xc="urn:ietf:params:xml:ns:netconf:base:1.0">
    <interfaces xmlns="http://flint.si/ns/yang/dev-sim">
      <loopback>
        <name></name>
        <ip>
          <address></address>
        </ip>
      </loopback>
    </interfaces>
  </config>
</edit-config>
```

In your Java implementation, you will use this XML payload as a template for creating a NETCONF edit-config message. An edit-config message begins with the XML tag "<edit-config>", followed by the tag "<target>", to which a datastore (candidate or running) is added. Because the NETCONF device you are using does not have a writable-running capability, the target is statically set to **candidate**.

2. Use the VS Code editor to open the **NetconfCommands.java** file and implement the **editConfig** method by uncommenting the following lines. In this case, remove the "\*" and "/" before and after the highlighted lines.

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

  public static void editConfig(Device device, String payload, String
operation, String interface_id, String address) throws
ParserConfigurationException, SAXException, IOException {
    /*
      Document document = Helper.parsePayload(payload);
      Node interfaceNode =
document.getElementsByTagName("name").item(0);
      Node addressNode =
document.getElementsByTagName("address").item(0);
      ...
      if(!address.isEmpty()) {
```

```
        addressNode.setTextContent(address);

    }

    else {

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

        ipNode.getParentNode().removeChild(ipNode);

    }

}

if (operation.equals("delete")) {

    Element loopbackElement = (Element)
interfaceNode.getParentNode();

    loopbackElement.setAttribute("xc:operation", "delete");

}

interfaceNode.setTextContent(interface_id);

XML message =
device.executeRPC(Helper.DocumentToString(document));

System.out.println(message);

#}

}
```

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.

3. Save the file.
  4. Open the **App.java** file, and in the main method uncomment the following lines to implement the user query prompt for the edit-config operation.

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

    else if(command == 2) {
        /*
        input.nextLine();
        System.out.print("loopback id [0]: ");
        String interface_id = input.nextLine();
        System.out.print("operation: merge/delete [merge]: ");
        String operation = input.nextLine();
        String address = "";
        if (interface_id.isEmpty()) {
            interface_id = "0";
        }
    }
}
```

```

        }

        if (operation.isEmpty()) {
            operation = "merge";
        }

        if (operation.equals("merge")) {
            System.out.print("ip address: ");
            address = input.nextLine();
        }

        editConfig(device, configureInterfacePayload, operation,
interface_id, address);
    */
}

```

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.

5. Implement the third user input option, which is a NETCONF commit command by uncommenting this line of code.

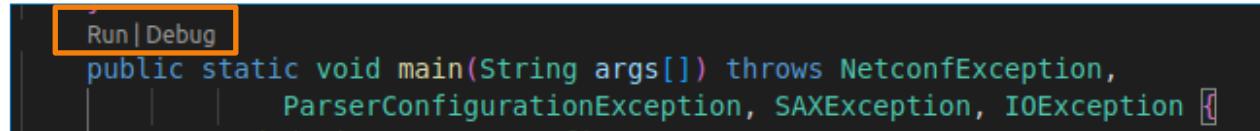
```

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

```

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

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

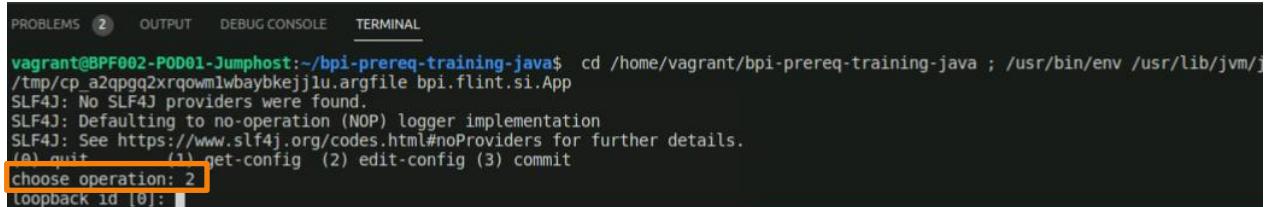


```

Run | Debug
public static void main(String args[]) throws NetconfException,
ParserConfigurationException, SAXException, IOException {
}

```

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

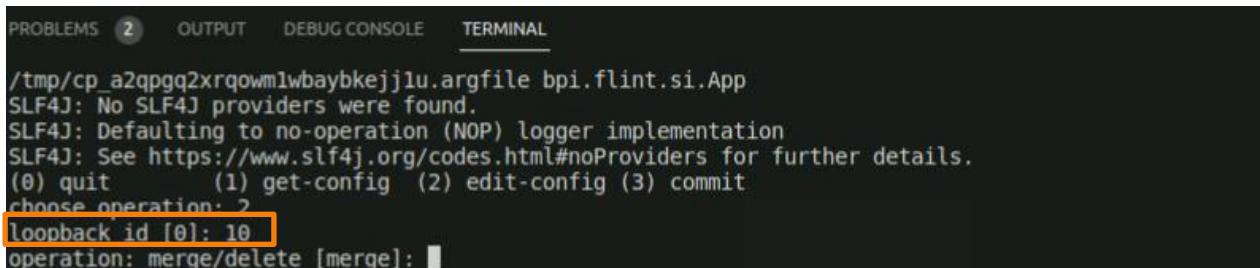


```

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL
vagrant@BPF002-POD01-Jumphost:~/bpi-prereq-training-java$ cd /home/vagrant/bpi-prereq-training-java ; /usr/bin/env /usr/lib/jvm/jdk-11.0.11+9-Ubuntu-1ubuntu1/jre/bin/java -jar bpi.flint.si.App
SLF4J: No SLF4J providers were found.
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See https://www.slf4j.org/codes.html#noProviders for further details.
(0) quit      (1) get-config  (2) edit-config (3) commit
choose operation: 2
loopback id [0]: 

```

8. Choose a loopback ID for the loopback interface you want to configure. If you don't provide the ID, Loopback 0 will be used.



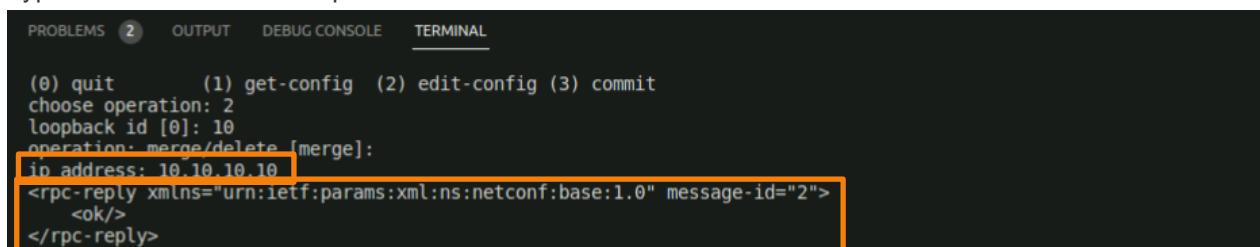
```

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL

/tmp/cp_a2qpgq2xrqowmlwbaybkejjlu.argfile bpi.flint.si.App
SLF4J: No SLF4J providers were found.
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See https://www.slf4j.org/codes.html#noProviders for further details.
(0) quit      (1) get-config  (2) edit-config (3) commit
choose operation: 2
loopback id [0]: 10
operation: merge/delete [merge]: 

```

9. Press **Enter** to select the default merge operation.
10. Type in the IP address and press enter.



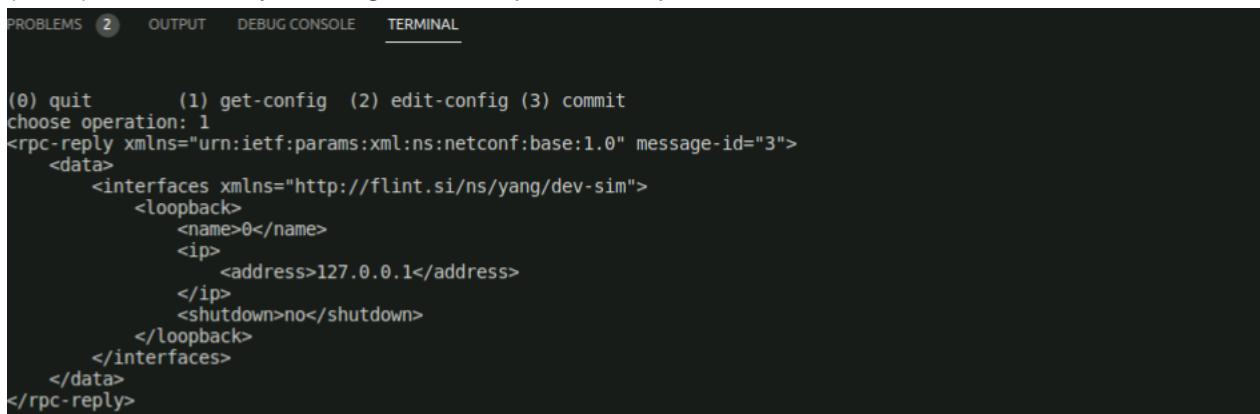
```

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL

(0) quit      (1) get-config  (2) edit-config (3) commit
choose operation: 2
loopback id [0]: 10
operation: merge/delete [merge]:
in address: 10.10.10.10
<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.



```

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL

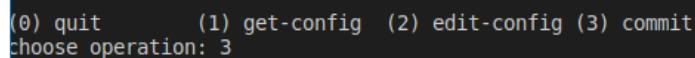
(0) quit      (1) get-config  (2) edit-config (3) commit
choose operation: 1
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="3">
  <data>
    <interfaces xmlns="http://flint.si/ns/yang/dev-sim">
      <loopback>
        <name>0</name>
        <ip>
          <address>127.0.0.1</address>
        </ip>
        <shutdown>no</shutdown>
      </loopback>
    </interfaces>
  </data>
</rpc-reply>

```

Notice that the running datastore does not have any configuration changes, because you used a candidate datastore when configuring the interface and have not yet committed the configuration.

12. 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

```

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



```

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL
(0) quit      (1) get-config (2) edit-config (3) commit
choose operation: 1
<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">
      <loopback>
        <name>0</name>
        <ip>
          <address>127.0.0.1</address>
        </ip>
        <shutdown>no</shutdown>
      </loopback>
      <loopback>
        <name>10</name>
        <ip>
          <address>10.10.10.10</address>
        </ip>
        <description/>
        <shutdown/>
      </loopback>
    </interfaces>
  </data>
</rpc-reply>

```

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

14. Exit the program by pressing **Ctrl + c**.
15. Since you are done with developing the code, add and commit all the changes.

**NOTE:** You will see a warning about including a git repository inside of this git repository. This is because you have cloned one of the dependencies from another git repository. For now, you can ignore this warning

```

$ cd ~/pod01-bpi-training
~/pod01-bpi-training$ git add .
~/pod01-bpi-training$ git commit -m "Add Java NETCONF manager"

```

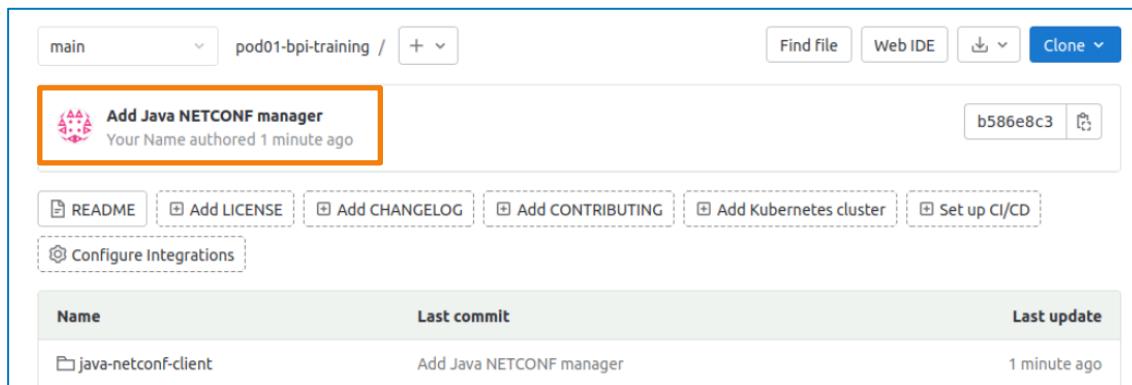
16. Push the changes to GitLab.

```

~/pod01-bpi-training$ git push
Username for 'http://10.247.1.27': studentxx
Password for 'http://student01@10.247.1.27':
Enumerating objects: 58, done.
Counting objects: 100% (58/58), done.
Delta compression using up to 4 threads
Compressing objects: 100% (29/29), done.
Writing objects: 100% (44/44), 16.90 KiB | 3.38 MiB/s, done.
Total 44 (delta 3), reused 0 (delta 0), pack-reused 0
To http://10.247.1.27/bpi-netdev-intro/pod01-bpi-training.git
  1d74c05..b586e8c main -> main

```

17. All your changes are now safely stored in a central repository hosted on the GitLab server.



The screenshot shows a GitLab repository interface. At the top, there are dropdown menus for 'main' and 'pod01-bpi-training / +'. To the right are buttons for 'Find file', 'Web IDE', a download icon, and a 'Clone' button. Below this is a commit card with a user icon, the title 'Add Java NETCONF manager', and the note 'Your Name authored 1 minute ago'. To the right of the card is a commit hash 'b586e8c3' and a copy icon. Below the card is a horizontal bar with several dashed buttons: 'README', 'Add LICENSE', 'Add CHANGELOG', 'Add CONTRIBUTING', 'Add Kubernetes cluster', and 'Set up CI/CD'. Underneath this bar is a 'Configure Integrations' button. At the bottom of the card is a table with three columns: 'Name', 'Last commit', and 'Last update'. The table contains one row for 'java-netconf-client' with the commit message 'Add Java NETCONF manager' and a timestamp of '1 minute ago'.

Name	Last commit	Last update
java-netconf-client	Add Java NETCONF manager	1 minute ago

End of Lab

## Lab 3: Using REST APIs and RESTCONF

### Objectives

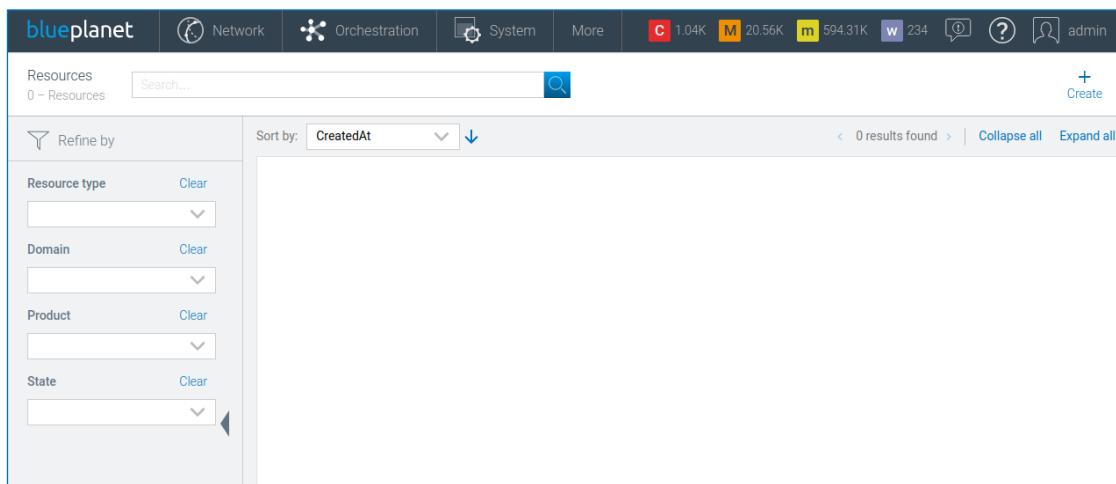
- Using Swagger UI to explore REST API calls
- Using Swagger UI to send requests
- Using Postman to test REST APIs
- Managing network devices with RESTCONF

### 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.

The screenshot shows the blueplanet Orchestration interface. At the top, there is a navigation bar with tabs: Network, Orchestration (which is highlighted with an orange border), System, and More. Below the navigation bar, the page title is "Onboarding". Under the "Onboarding" title, there are four items listed: "Asset Manager" (Customize Assets), "Domains" (Manage resource domains), "Resource Types" (Browse Resource Type Models), and "SSH Keys" (Add user SSH keys). The "SSH Keys" item is also highlighted with an orange border.

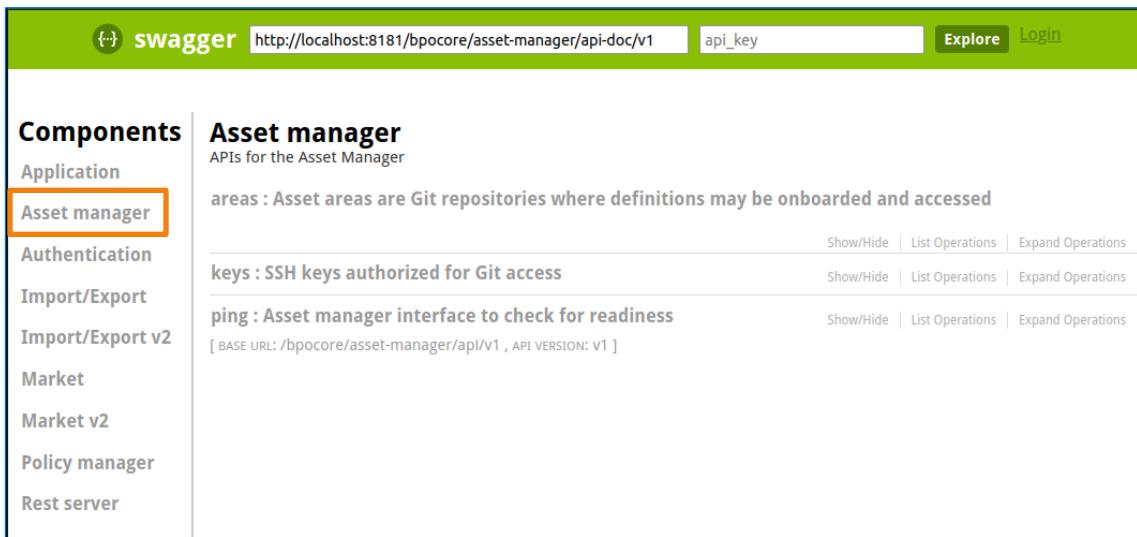
3. You can see that there is currently only one SSH public key stored on the server. Only the device with the corresponding private key can currently be authorized.

The screenshot shows the "SSH Keys" list interface. At the top, there is a header with the title "SSH Keys" and a count of "1 – SSH Keys". To the right of the title is a "Create" button with a plus sign. Below the header, there is a table with one row. The table has columns for a checkbox icon, a preview of the SSH key, and a "Delete" button. The preview column contains the SSH key value: "ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQCrUQ2s8LJWfXh1/dGmZ0lQj8JnEv/hvfUeH1jKxE8pz1YXvL+jN/tZohH8C5ihAoQJn/j+5y/3H+CKpOyF98VyOoJyLKz+2bYupXBzC1/". The entire row is highlighted with an orange border.

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](http://localhost:8181/swagger).

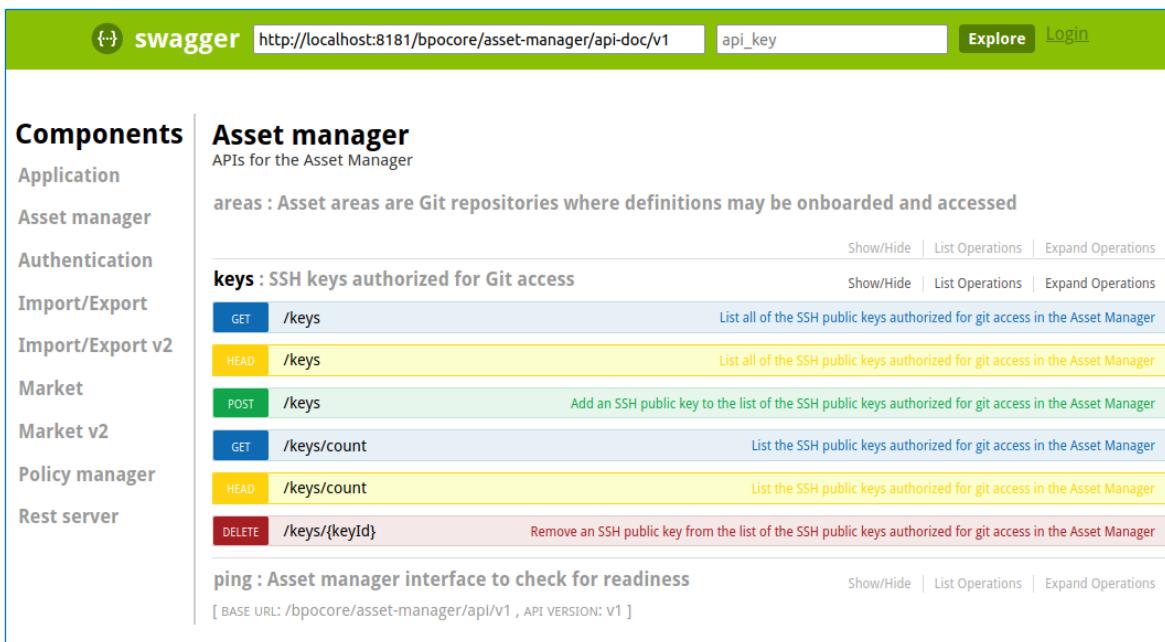
The screenshot shows the Swagger UI interface. At the top, there is a search bar with the text "localhost:8181/swagger". Below the search bar, there is a green header bar with the text "swagger" and the URL "http://localhost:8181/bpocore/market/api-doc/v1".

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



The screenshot shows the Swagger UI interface for the Asset manager API. On the left, a sidebar lists various components: Application, Asset manager (which is selected and highlighted in orange), Authentication, Import/Export, Import/Export v2, Market, Market v2, Policy manager, and Rest server. The main content area is titled "Asset manager" and describes it as "APIs for the Asset Manager". It contains three sections: "areas", "keys", and "ping". Each section has "Show/Hide", "List Operations", and "Expand Operations" buttons. Below the "ping" section is a note: "[ BASE URL: /bpocore/asset-manager/api/v1 , API VERSION: V1 ]".

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.



This screenshot shows the expanded API documentation for the "keys" endpoint under the "Asset manager" component. The sidebar remains the same. The "keys" section is expanded, showing five methods: GET /keys, HEAD /keys, POST /keys, GET /keys/count, and HEAD /keys/count. Each method is described with its purpose: "List all of the SSH public keys authorized for git access in the Asset Manager", "Add an SSH public key to the list of the SSH public keys authorized for git access in the Asset Manager", "List the SSH public keys authorized for git access in the Asset Manager", and "List the SSH public keys authorized for git access in the Asset Manager". The "DELETE /keys/{keyId}" method is shown but lacks a description. A note at the bottom states: "[ BASE URL: /bpocore/asset-manager/api/v1 , API VERSION: V1 ]".

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.



```

GET /keys
List all of the SSH public keys authorized for git access in the Asset Manager

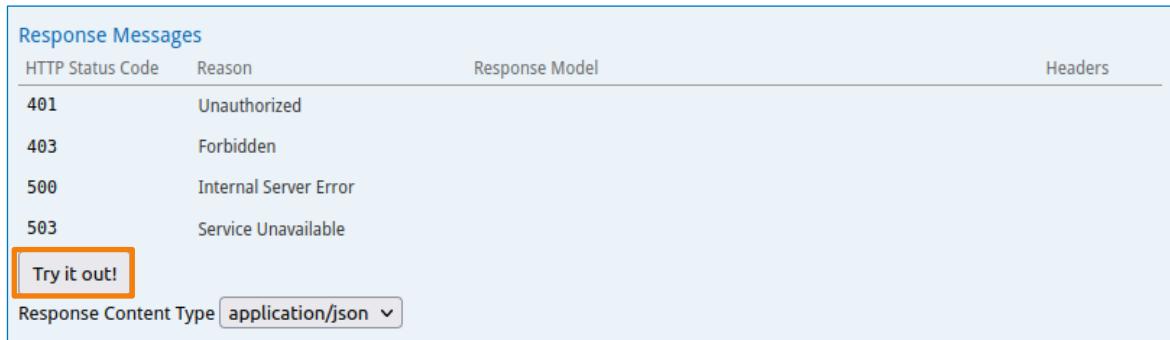
Implementation Notes
Keys are identified by their fingerprint (16 byte MD5 hash in hexadecimal format without byte separators)

Response Class (Status 200)
Model Model Schema

{
  "items": [
    {
      "id": "string",
      "key": "string",
      "createdAt": "2022-12-19T12:26:15.430Z",
      "updatedAt": "2022-12-19T12:26:15.430Z"
    }
  ],
  "nextPageToken": "string",
  "limit": 0
}

```

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.



HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
500	Internal Server Error		
503	Service Unavailable		

**Try it out!**

Response Content Type application/json ▾

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"
      AAAAB3NzaC1yc2EAAAQABAAQCrUQ2s8LJWfXh1/dGmZO1Qj8JnEv/hvfUeH1jKxE8p
      z1YXvL+jN/tZ0hH8C5ihAoQJn/j+5y/3H+CKp0yF98VyOoJyLKz+2bYupXBzC1/GJf5W8q0K
      f+Gk0J5xebd7W8XvH9XJEtGthUmFgSV8LkR/H7VUJxgNd7V3z/n9X7V/QZ/uCByVsmQvLmhI
      6U+ba6uDG6UiJemUaT6UjT6TnV7sA6vGm0e+U70/yU6T8U6pXU6d0m6TzT6sfU",
    }
  ]
}
```

```

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

```

- 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.

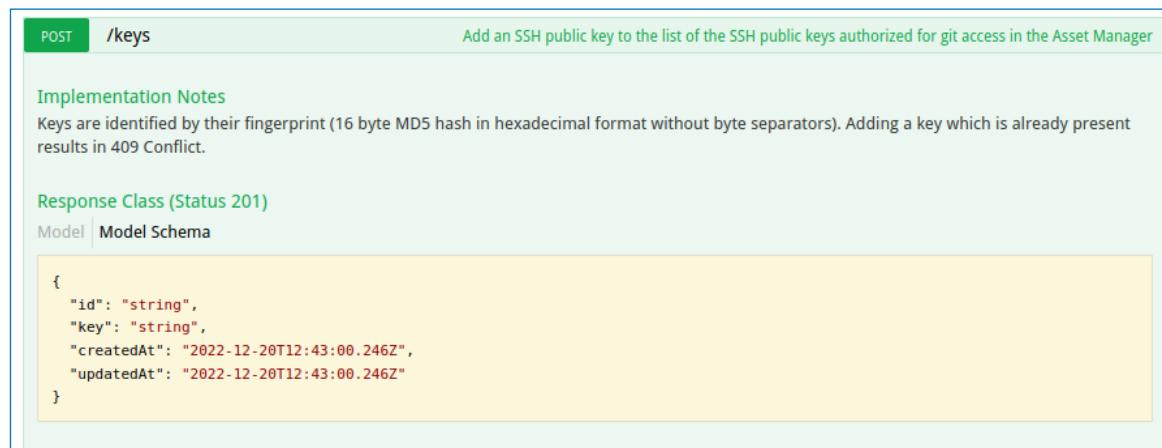
**NOTE:** Your key is probably different from the one shown in the following output.

```

[~]$ cd /home/vagrant/.ssh/
[~/ssh]$ cat id_rsa.pub
ssh-rsa
AAAAB3NzaC1yc2EAAAQABAAQ/sHxu+Kjy6gMWR8jzKKB433SEwMZ7Uy4Zr9vwnG1F
XRJUi6wzdn783JPf6r1lK4rLBPEJeALTKeGJAhR61DyiMUG2gCEBA1qwxUJY2DNPUnnTJx9
A3E75s/jJaCSkke7LMZPgL5zYrr0VOKMvCorU94RR50Mz3E7bDpHzKcqOrHjxrcd8V0Qk867
YcSyiVdwZCPSDjeAdgRAf00uxcj17kRLWI0zXbYqBFIFD19ARHB1uP7jIuQq+BHedTAtBeoP
/09hYVnvkJ25xUwYCsvrCbaedFYzhpEhFsiJtej8nPux803s43HhxM/eDcbWjpMQD2ciertp
ew91bagrttm3 vagrant@bp-devops-toolkit

```

- 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.



**POST /keys** Add an SSH public key to the list of the SSH public keys authorized for git access in the Asset Manager

**Implementation Notes**  
Keys are identified by their fingerprint (16 byte MD5 hash in hexadecimal format without byte separators). Adding a key which is already present results in 409 Conflict.

**Response Class (Status 201)**

Model Model Schema

```
{
  "id": "string",
  "key": "string",
  "createdAt": "2022-12-20T12:43:00.246Z",
  "updatedAt": "2022-12-20T12:43:00.246Z"
}
```

- 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
AAAAB3NzaC1yc2EAAAQABAAQ/sHxu+Kjy6gMWR8jzKKB433SEwMZ7Uy4Zr9vwnG1F
XRJUi6wzdn783JPf6r1lK4rLBPEJeALTKeGJAhR61DyiMUG2gCEBA1qwxUJY2DNPUnnTJx9
```

```
A3E75s/jJaCSkke7LMZPgL5zYrr0VOKMvCOrU94RR50Mz3E7bDpHzKcq0rHjxrcd8V0Qk867
YcSyiVdwZCPSDjeAdgRAf00uxcj17kRLWI0zXbYqBFIFD19ARHB1uP7jIuQq+BHedTAtBeoP
/09hYVnvkJ25xUwYCsvrCbaedFYzhpEhFsiJtej8nPux803s43HhxM/eDcbWjpMQD2ciietp
ew91bagrttm3 vagrant@bp-devops-toolkit"
}
```

**Parameters**

Parameter	Value	Description	Parameter Type	Data Type
key	<pre>{   "key": "ssh-rsa AAAAB3NzaC1yc2EAAQABAAQCS/5Hxx+Kjy6 gMR8jzKB433SEwMZ7Uy4Zr9vnG1FXRJUi6wdzj n783JPf6rlK4rLBPEJeALTKeGJAhR6lDyiMUG2gC EBA1lqxUJY2DNPUunnTx9A3E75s /iJaCSkke7LMZPgL5zYrr0VOKMvCOrU94RR50Mz3E"</pre>	An RSA public key	body	<a href="#">Model</a>   <a href="#">Model Schema</a> <pre>{   "id": "string",   "key": "string",   "createdAt": "2022-12-20T12:42:59.914Z",   "updatedAt": "2022-12-20T12:42:59.914Z" }</pre>
Parameter content type:	application/json	▼		Click to set as parameter value

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

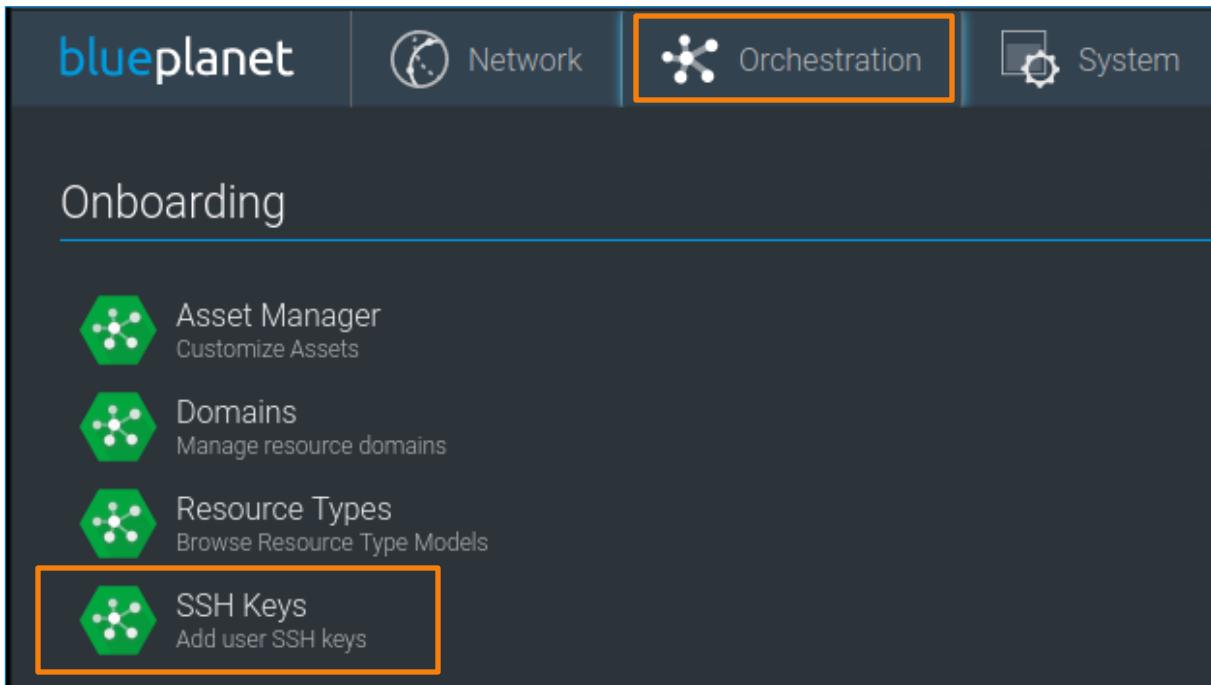
**Try it out!** [Hide Response](#)

---

**Response Code**

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.



The screenshot shows the blueplanet web interface. At the top, there are four tabs: Network, Orchestration (which is highlighted with an orange border), and System. Below the tabs, the page title is "Onboarding". Under the "Onboarding" section, there are four items listed: Asset Manager (Customize Assets), Domains (Manage resource domains), Resource Types (Browse Resource Type Models), and SSH Keys (Add user SSH keys). The "SSH Keys" item is also highlighted with an orange box.

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



The screenshot shows a list of SSH keys. The header says "SSH Keys" and "2 - SSH Keys". There are two entries in the list:

- >  ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQCrUQ2s8LJWfXh1/dGmZ0lQj8JnEv/hvfUeH1jKxE8pz1YXvL+jN/tZ0hH8C5ihAoQJn/j+5y/3H+CKpOyF98VyOoJyLKz+2b  
YupXBzC1/
- >  ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQCs/hXu+Kjy6gMWR8jzKKB433SEwMZ7Uy4Zr9vwmG1FXRJUI6wzdjn783JPf6r1Ik4rLBPEJeALTKeGJAhR6lDyIMUG2gCE  
BAIqwxUJY2DNPUnnTJx9A3E75s/JaCSkke7LMZPgL5zYrr0VOKMvCorU94RR50Mz3E7bDp

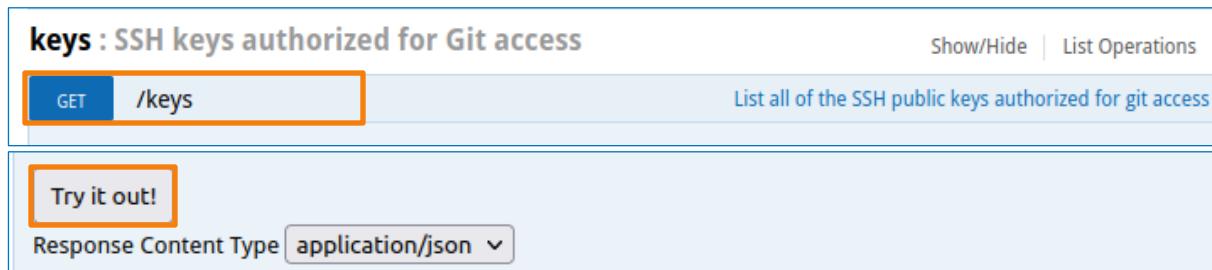
16. The SSH keys can also be deleted from the server through the API. Navigate back to the Swagger UI and observe how the request that removes a key should look like by expanding the **DELETE** request. You can see that the request must include the **Id** of the key that must be deleted.



The screenshot shows the Swagger UI for the "/keys/{keyId}" endpoint. The endpoint description is "Remove an SSH public key from the list of the SSH public keys authorized for git access in the Asset Manager". The parameters section shows a table:

Parameter	Value	Description	Parameter Type	Data Type
keyId	(required)	The fingerprint of the key to remove (16 byte MD5 hash in hexadecimal format without byte separators)	path	string

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.



**keys : SSH keys authorized for Git access**

GET /keys

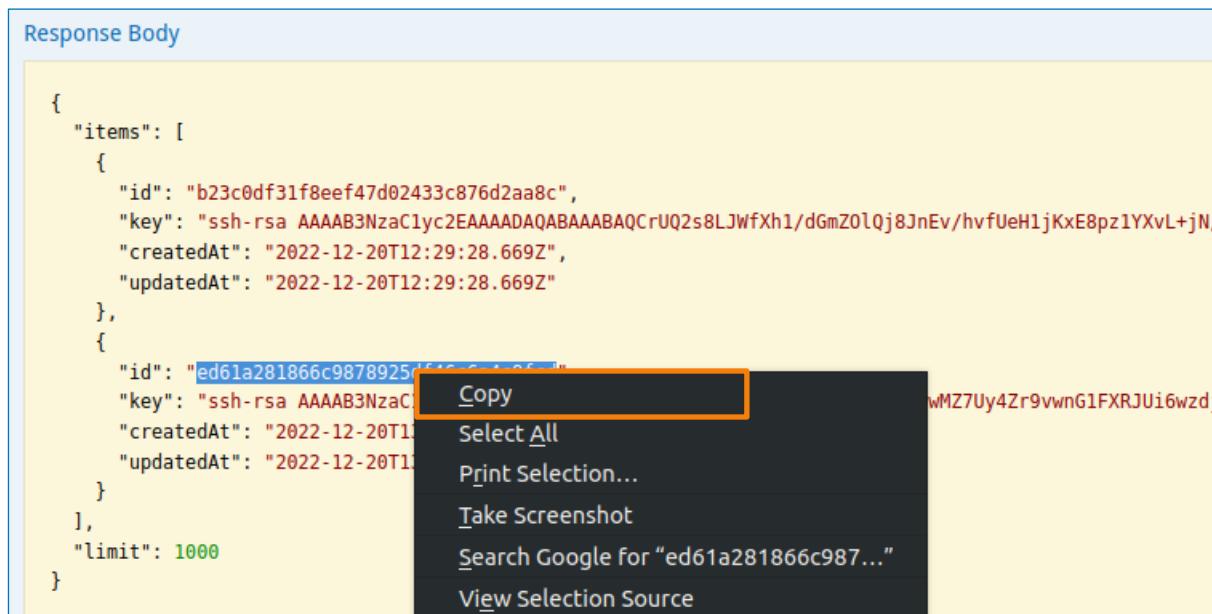
Try it out!

List all of the SSH public keys authorized for git access

Response Content Type application/json ▾

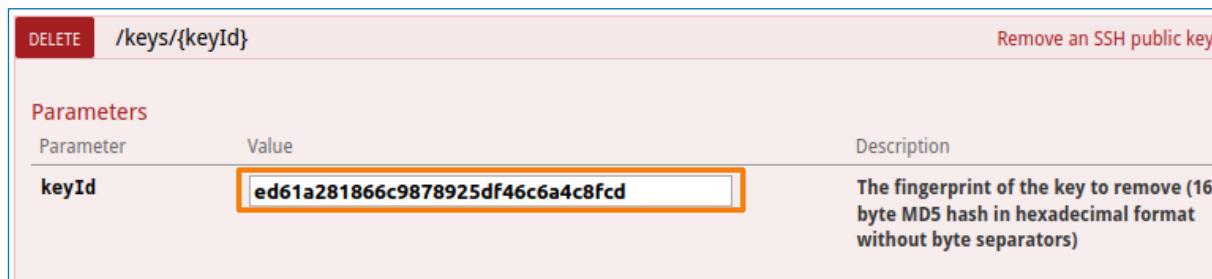
18. Copy the **id** of your public key to the clipboard.

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



```
{
  "items": [
    {
      "id": "b23c0df31f8eef47d02433c876d2aa8c",
      "key": "ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQCrUQ2s8LJWfxh1/dGmZ0l0j8JnEv/hvfUeH1jKxE8pz1YXvL+jN",
      "createdAt": "2022-12-20T12:29:28.669Z",
      "updatedAt": "2022-12-20T12:29:28.669Z"
    },
    {
      "id": "ed61a281866c9878925df46c6a4c8fc",
      "key": "ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQCrUQ2s8LJWfxh1/dGmZ0l0j8JnEv/hvfUeH1jKxE8pz1YXvL+jN",
      "createdAt": "2022-12-20T12:29:28.669Z",
      "updatedAt": "2022-12-20T12:29:28.669Z"
    }
  ],
  "limit": 1000
}
```

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



DELETE /keys/{keyId}

Remove an SSH public key

Parameters

Parameter	Value	Description
keyId	ed61a281866c9878925df46c6a4c8fc	The fingerprint of the key to remove (16 byte MD5 hash in hexadecimal format without byte separators)

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

The screenshot shows a user interface for testing API endpoints. At the top, there is a button labeled "Try it out!" with an orange border. To its right is a link labeled "Hide Response". Below this, a section titled "Response Code" displays the value "204" in a yellow-highlighted box.

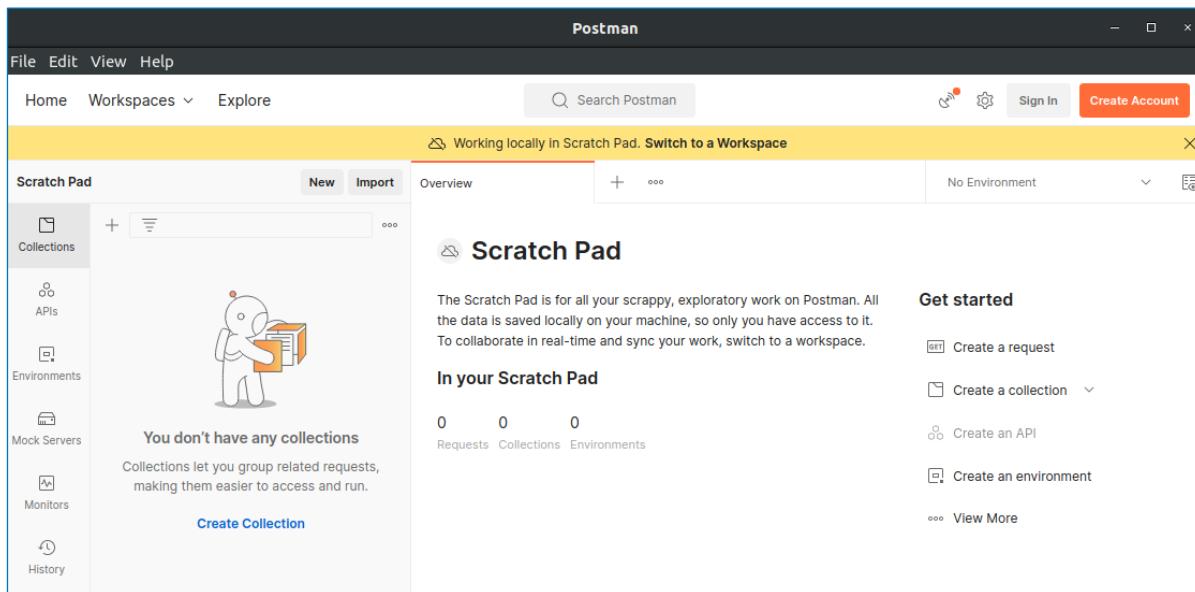
21. Go back to the list of SSH keys on the web UI and verify that your key is not present there anymore.

The screenshot shows a list of SSH keys. The header includes "SSH Keys" and "1 - SSH Keys" on the left, and a "Create" button with a plus sign on the right. Below the header, there is a single key entry: "ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQCrUQ2s8LJWfxh1/dGmZ0lQj8JnEv/hvfUeH1jKxE8pz1YXvL+jN/tZOhH8C5lhAoQJn/j+5y/3H+CKpOyF98VyOoJyLKz+2bYupXBzC1/". The entire list area is enclosed in a light gray box.

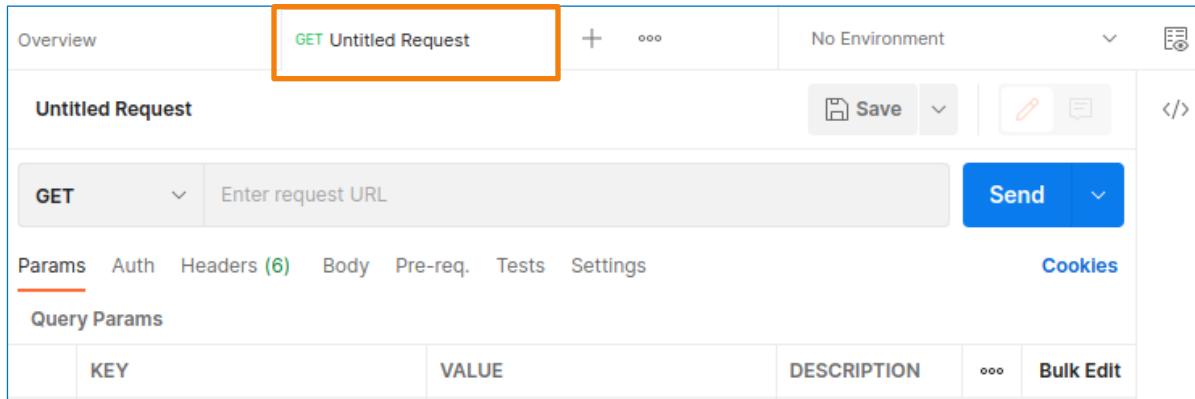
## Task 2: Using Postman to Construct and Send API Calls

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.

## Asset manager

APIs for the Asset Manager

**areas** : Asset areas are Git repositories where definitions may be onboarded and accessed

Show/Hide

**keys** : SSH keys authorized for Git access

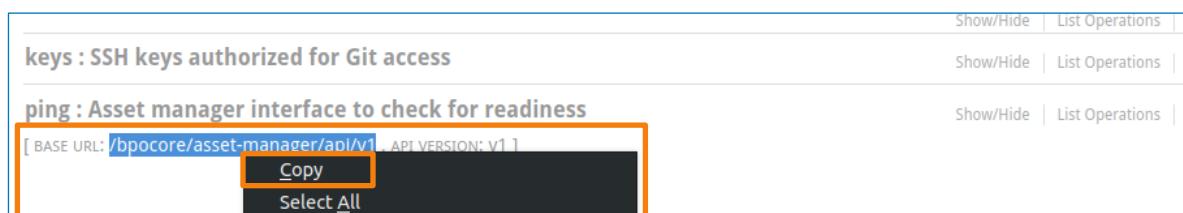
Show/Hide

**ping** : Asset manager interface to check for readiness

Show/Hide

[ BASE URL: /bpocore/asset-manager/api/v1 , API VERSION: v1 ]

- 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.



keys : SSH keys authorized for Git access

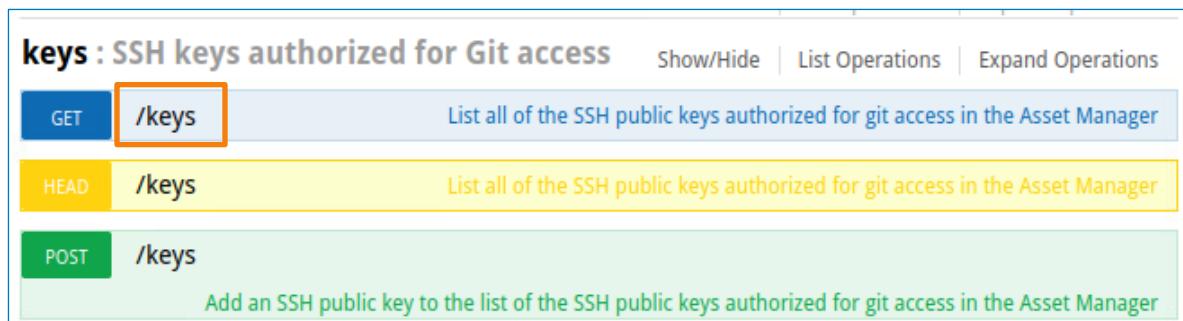
ping : Asset manager interface to check for readiness

[ BASE URL: /bpocore/asset-manager/api/v1 , API VERSION: v1 ]

Copy

Select All

- 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**.



**keys** : SSH keys authorized for Git access

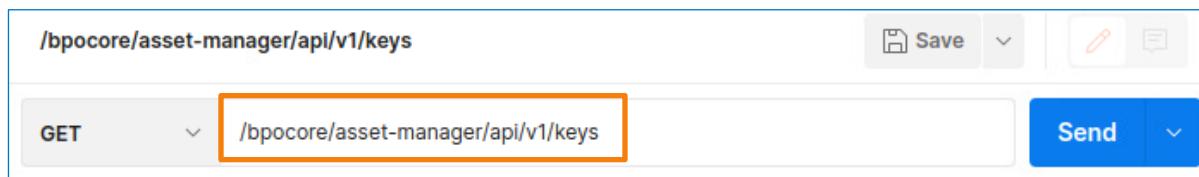
Show/Hide | List Operations | Expand Operations

GET /keys List all of the SSH public keys authorized for git access in the Asset Manager

HEAD /keys List all of the SSH public keys authorized for git access in the Asset Manager

POST /keys Add an SSH public key to the list of the SSH public keys authorized for git access in the Asset Manager

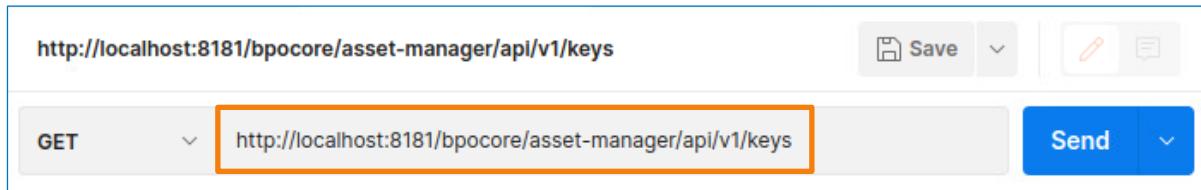
- Navigate back to **Postman** and enter the target resource in the request URL field. Paste the base URI that you copied in step 4 and append the **/keys** path. You should obtain the following URI.



/bpocore/asset-manager/api/v1/keys

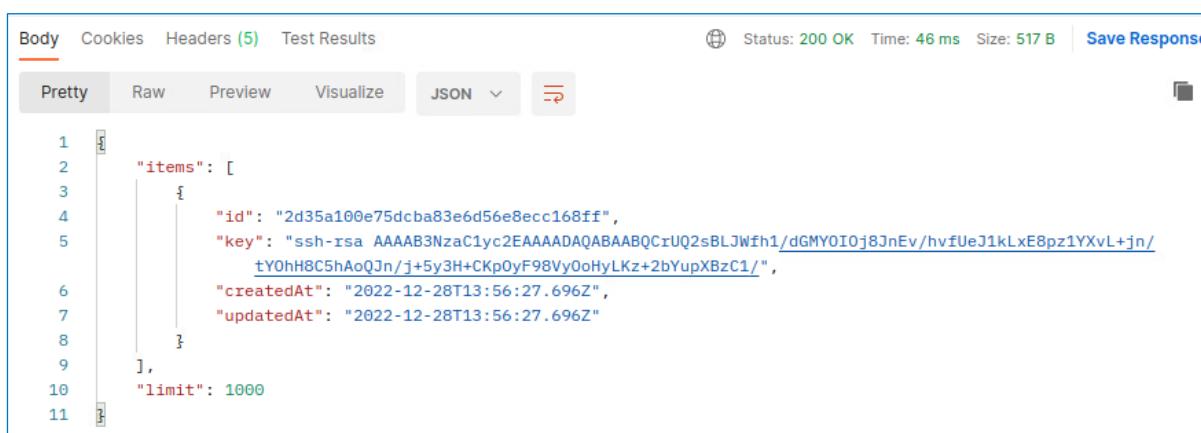
GET /bpocore/asset-manager/api/v1/keys Send

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.



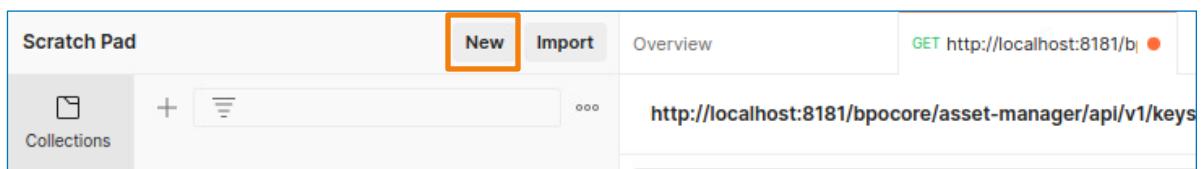
8. Make sure that the **GET** method is selected in the dropdown menu and **send the request**. The server should reply with a representation of the target resource. In this case, you should receive the list of existing SSH keys.

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

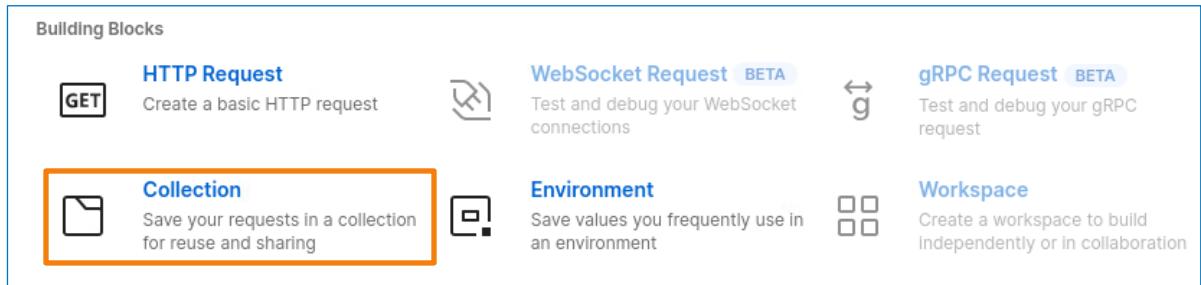


9. Now you will save the created request to a Postman collection. Since there are currently no existing collections, click the **New** button to create a new one.

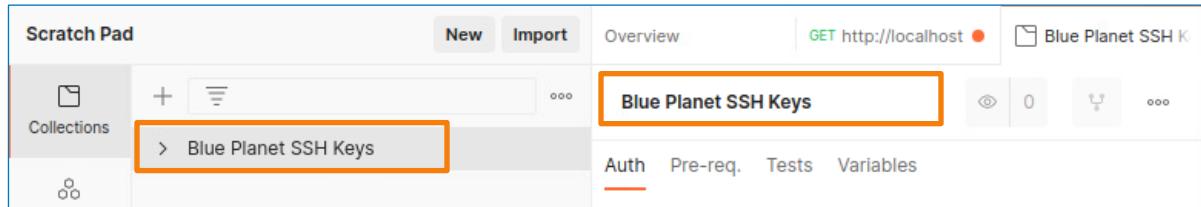
**NOTE:** Different API calls are often related to the same workflow or feature. Postman enables you to group related API calls to **collections**.



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

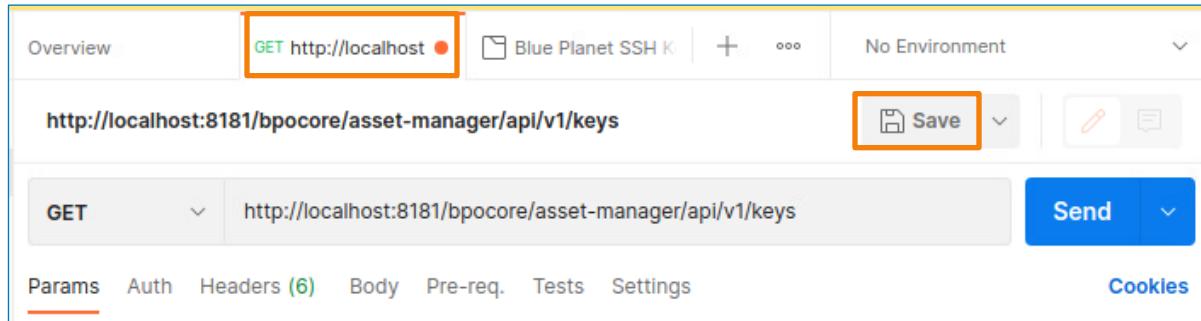


11. Name the collection '**Blue Planet SSH Keys**'.



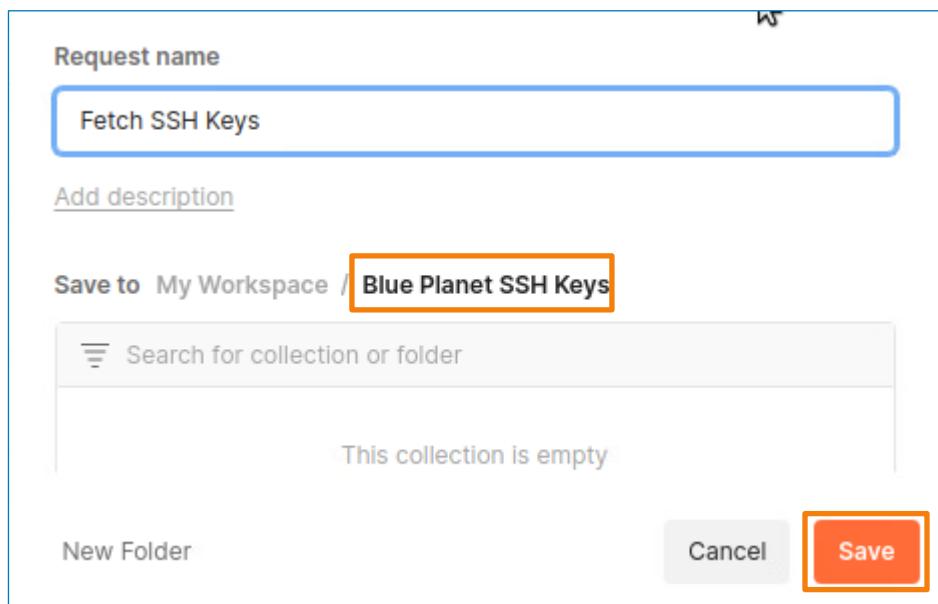
The screenshot shows the 'Scratch Pad' interface with the 'Collections' tab selected. A list of collections is shown, with 'Blue Planet SSH Keys' highlighted by an orange box. The top navigation bar includes 'New', 'Import', 'Overview', and a tab for 'GET http://localhost'. Below the list are tabs for 'Auth', 'Pre-req.', 'Tests', and 'Variables', with 'Auth' currently active.

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



The screenshot shows the API request configuration screen. At the top, the URL 'GET http://localhost' is highlighted with an orange box. Below the URL, the request method is set to 'GET' and the target URL is 'http://localhost:8181/bpocore/asset-manager/api/v1/keys'. On the right, there are buttons for 'Save', 'Send', and other options. The 'Params' tab is selected at the bottom.

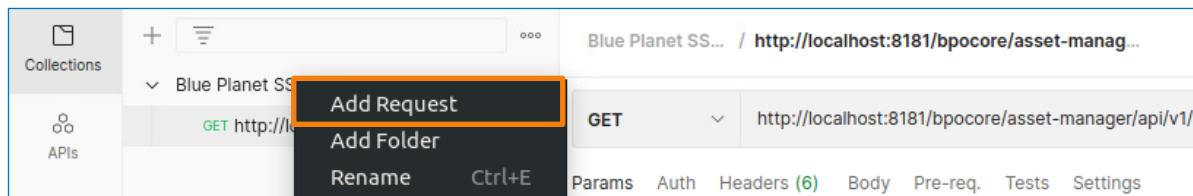
13. Name the request '**Fetch SSH Keys**' and choose your new **Blue Planet SSH Keys** collection from the pop-up menu and click **Save**.



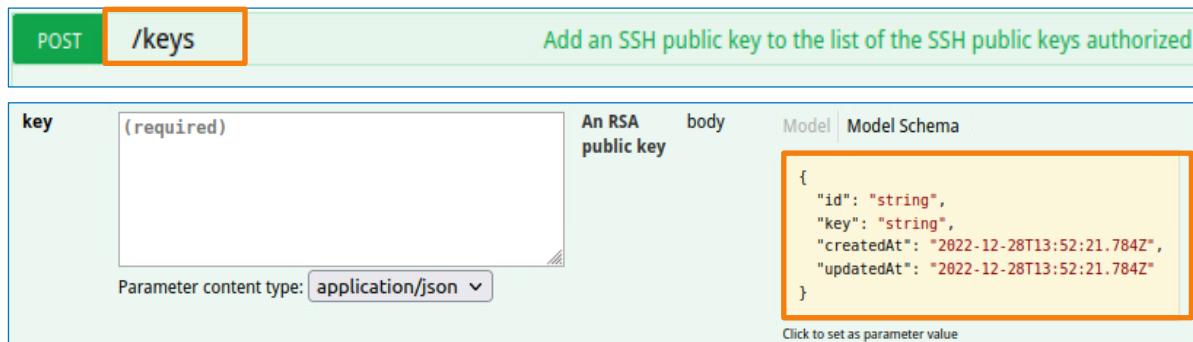
The screenshot shows a 'Request name' dialog box. It has fields for 'Request name' (containing 'Fetch SSH Keys') and 'Add description'. Below these, a 'Save to' dropdown menu is open, showing 'My Workspace / Blue Planet SSH Keys' with an orange box around 'Blue Planet SSH Keys'. There is also a search bar for collections and folders. At the bottom are 'Cancel' and 'Save' buttons, with 'Save' highlighted by an orange box.

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.

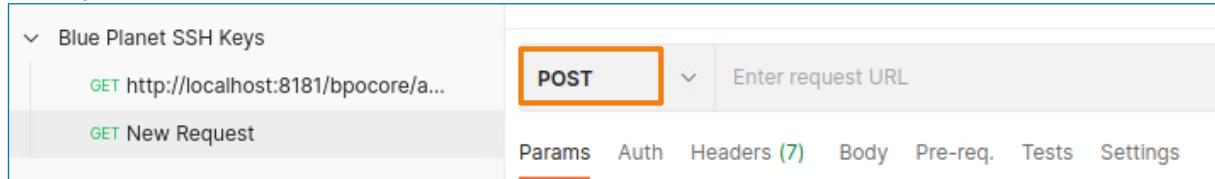


The screenshot shows the Swagger UI for the `POST /keys` endpoint. The `key` parameter is marked as required and is described as an RSA public key. The body schema is defined as follows:

```
{
  "id": "string",
  "key": "string",
  "createdAt": "2022-12-28T13:52:21.784Z",
  "updatedAt": "2022-12-28T13:52:21.784Z"
}
```

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.

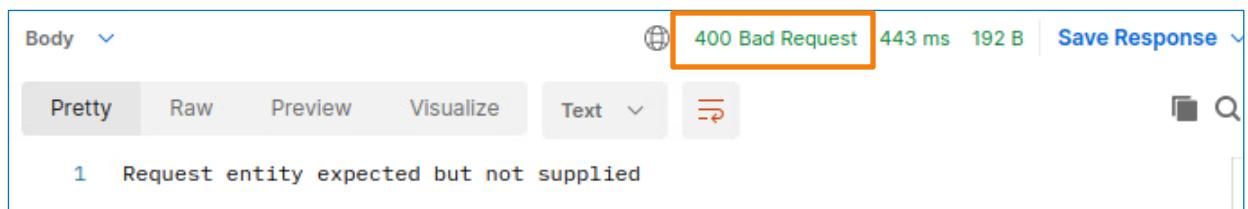
16. Go back to Postman and construct the request according to Swagger UI documentation. First set the request method to **POST**.



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.



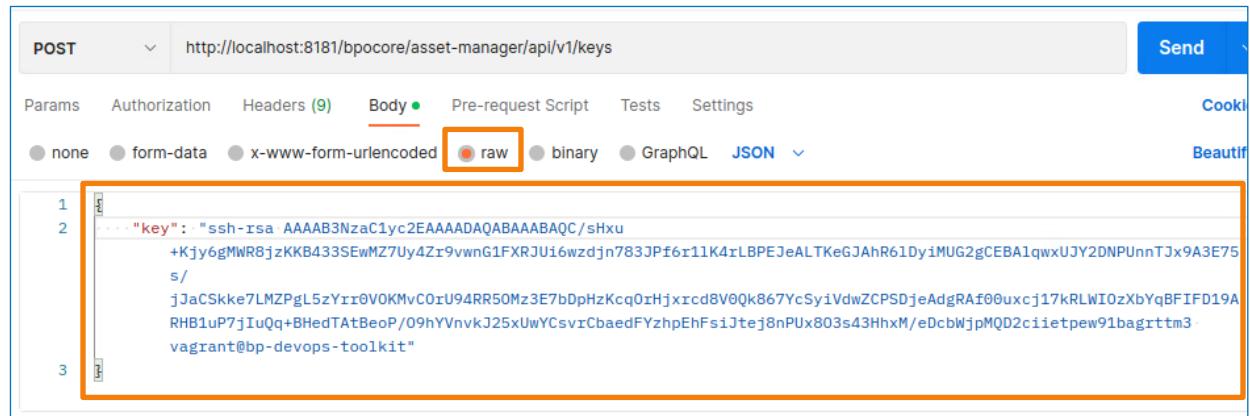
The screenshot shows the Postman interface after sending a POST request to the `/keys` endpoint. The response status is `400 Bad Request`. The error message is: `Request entity expected but not supplied`.

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
AAAAB3NzaC1yc2EAAAQABAAQ/sHxu+Kjy6gMWR8jzKKB433SEwMZ7Uy4Zr9vvnG1F
XRJUi6wzdn783JPf6r1lK4rLBPEJeALTKeGJAHR61DyiMUG2gCEBA1qwxUJY2DNPUnnTJx9
A3E75s/jJaCSkke7LMZPgL5zYrr0VOKMvCorU94RR50Mz3E7bDpHzKcqOrHjxrcd8V0Qk867
YcSyiVdwZCPSDjeAdgRAf00uxcj17kRLWIOzXbYqBFIFD19ARHB1uP7jIuQq+BHedTAtBeoP
/09hYVnvkJ25xUwYCsvrCbaedFYzhpEhFsiJtej8nPUs803s43HhxM/eDcbWjpMQD2cietp
ew91bagrttm3 vagrant@bp-devops-toolkit
```

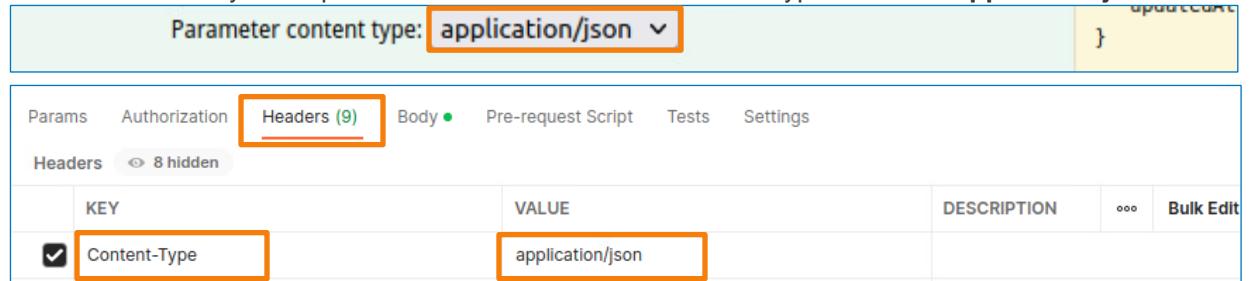
20. Copy the whole key and paste it to the body field in Postman. Set the type of encoding to **raw** and make sure that the payload follows the structure specified in the documentation.



The screenshot shows the Postman interface for a POST request to `http://localhost:8181/bpocore/asset-manager/api/v1/keys`. The **Body** tab is selected, and the **raw** radio button is selected. The input field contains the copied SSH key:

```
1 "key": "ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQ/sHxu
2 +Kjy6gMWR8jzKKB433SEwMZ7Uy4Zr9vvnG1F
3 XRJUi6wzdn783JPf6r1lK4rLBPEJeALTKeGJAHR61DyiMUG2gCEBA1qwxUJY2DNPUnnTJx9
4 A3E75s/jJaCSkke7LMZPgL5zYrr0VOKMvCorU94RR50Mz3E7bDpHzKcqOrHjxrcd8V0Qk867
5 YcSyiVdwZCPSDjeAdgRAf00uxcj17kRLWIOzXbYqBFIFD19ARHB1uP7jIuQq+BHedTAtBeoP
6 /09hYVnvkJ25xUwYCsvrCbaedFYzhpEhFsiJtej8nPUs803s43HhxM/eDcbWjpMQD2cietp
7 ew91bagrttm3 vagrant@bp-devops-toolkit"
```

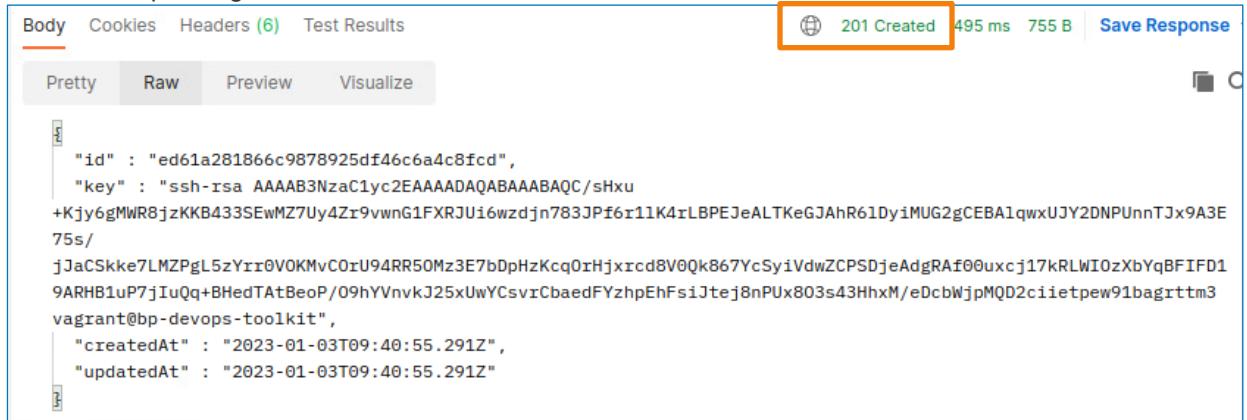
21. Before you send the request, you must set a new header that tells the server what is the type of content that was received. You can see that *content type* should be set to **application/json**. Open the headers tab of your request in Postman and set the Content-Type header to **application/json**



The screenshot shows the Postman interface with the **Headers** tab selected. The **Content-Type** header is set to `application/json`.

KEY	VALUE	DESCRIPTION	Bulk Edit
<input checked="" type="checkbox"/> Content-Type	application/json		

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



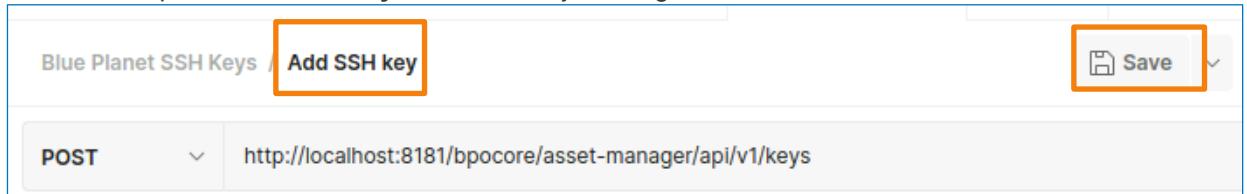
The screenshot shows a Postman request response. The status bar at the top right indicates a 201 Created response with 495 ms and 755 B. The response body is a JSON object representing an SSH key:

```

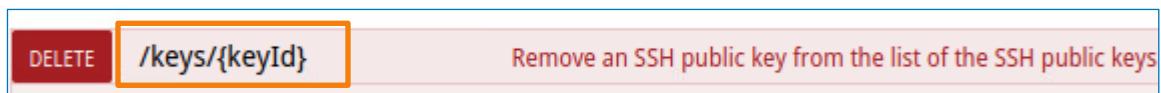
{
  "id": "ed61a281866c9878925df46c6a4c8fcd",
  "key": "ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQ/sHxu
+Kjy6gMWR8jzKKB433SEwMZ7Uy4Zr9vwnG1FXRJU16wdzjn783JPf6r11K4rLBPEJeALTKeGJAhR61Dy1MUG2gCEBA1qwxUJY2DNPUnnTJx9A3E
75s/
jJaCSkke7LMZPgL5zYrr0VOKMvC0rU94RR50Mz3E7bDpHzKcq0rHjxrcd8V0Qk867YcSyiVdwZCPSDjeAdgRAf00uxcj17kRLWI0zXbYqBFIFD1
9ARHB1uP7jIuQq+BHedTAtBeoP/09hVvnkJ25xUwYCsvrCbaedFYzhpEhFsiJtej8nPux803s43HhxM/eDcbWjpMQD2ciertpew91bagrttm3
vagrant@bp-devops-toolkit",
  "createdAt": "2023-01-03T09:40:55.291Z",
  "updatedAt": "2023-01-03T09:40:55.291Z"
}

```

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

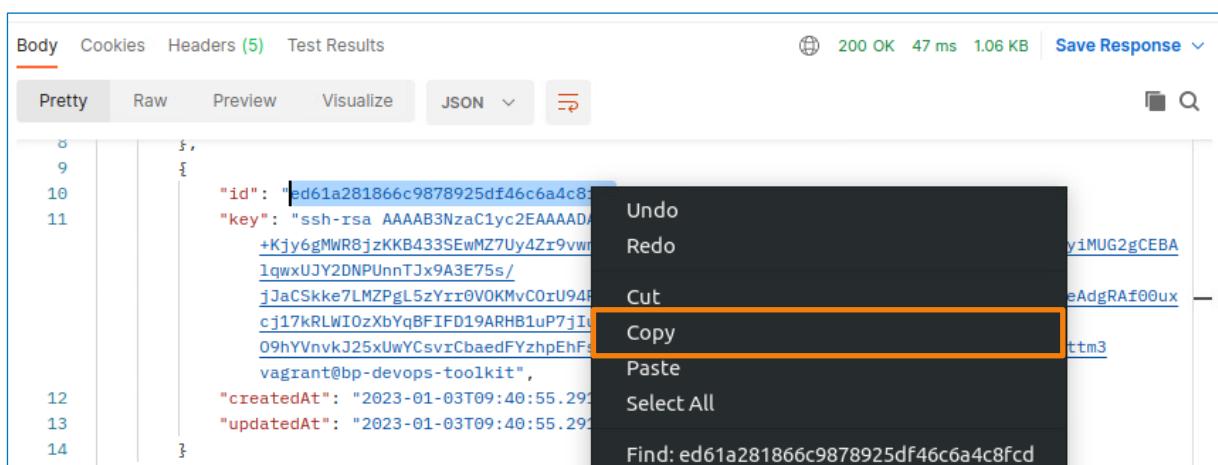


24. As you learned previously, the API can be used to delete the SSH keys as well. You will now use Postman to construct the **DELETE** request that you already used in Task 1. Navigate back to Swagger UI and examine the request to delete the SSH keys. Notice that the URL must include the id of the key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



The screenshot shows the response body in Postman. The response is a 200 OK with 47 ms and 1.06 KB. The response body is a JSON object representing an SSH key:

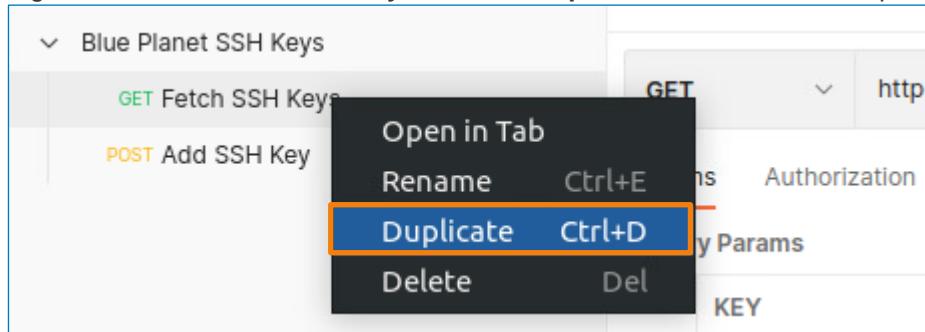
```

{
  "id": "ed61a281866c9878925df46c6a4c8fcd",
  "key": "ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQ/sHxu
+Kjy6gMWR8jzKKB433SEwMZ7Uy4Zr9vwnG1qwxUJY2DNPUnnTJx9A3E75s/
jJaCSkke7LMZPgL5zYrr0VOKMvC0rU94RR50Mz3E7bDpHzKcq0rHjxrcd8V0Qk867YcSyiVdwZCPSDjeAdgRAf00uxcj17kRLWI0zXbYqBFIFD1
9ARHB1uP7jIuQq+BHedTAtBeoP/09hVvnkJ25xUwYCsvrCbaedFYzhpEhFsiJtej8nPux803s43HhxM/eDcbWjpMQD2ciertpew91bagrttm3
vagrant@bp-devops-toolkit",
  "createdAt": "2023-01-03T09:40:55.291Z",
  "updatedAt": "2023-01-03T09:40:55.291Z"
}

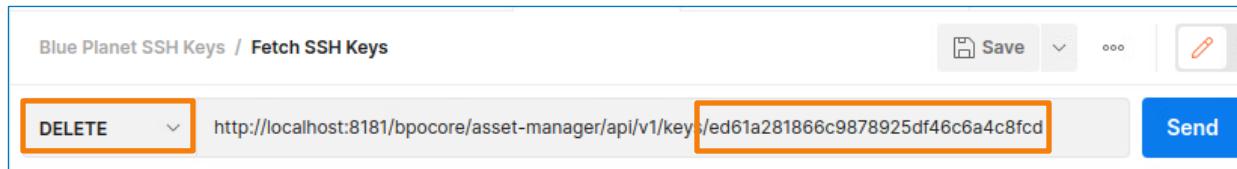
```

A context menu is open over the 'key' field, with the 'Copy' option highlighted with an orange box.

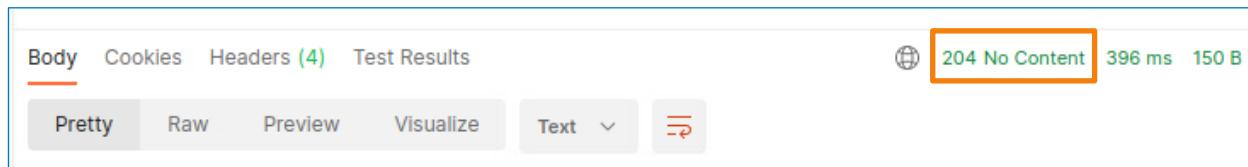
26. Right-click on the **Fetch SSH keys** and click **duplicate** to create a new duplicate request.



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.



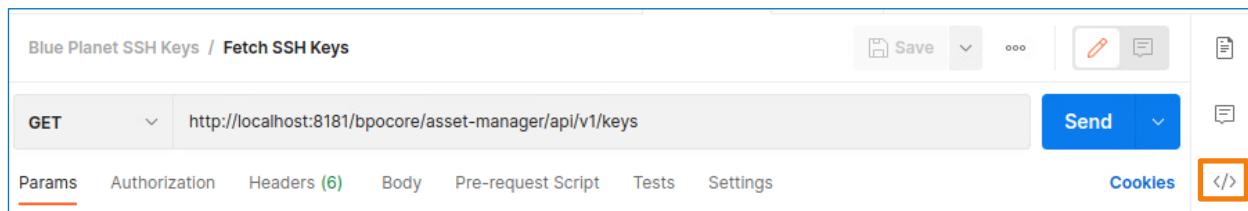
28. Send the request. The request should be successful, and the response code should be **204 No Content**.



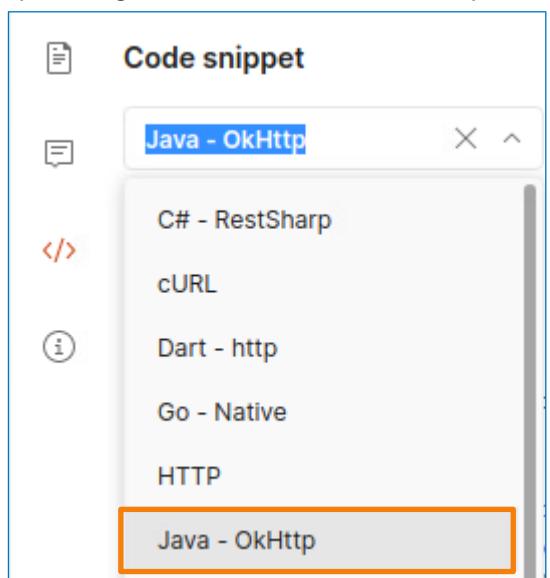
29. Name the request '**Delete SSH Key**' and save it to your collection by clicking the "Save" button.



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.

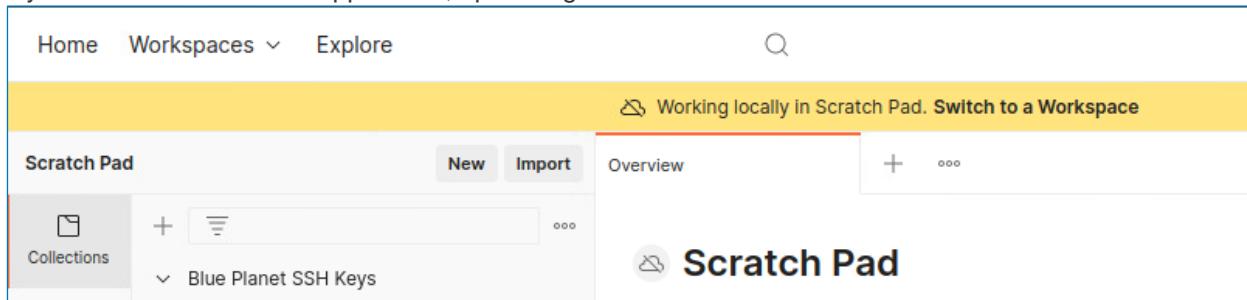
```
OkHttpClient client = new OkHttpClient().newBuilder()
    .build();
MediaType mediaType = MediaType.parse("text/plain");
RequestBody body = RequestBody.create(mediaType, "");
Request request = new Request.Builder()
    .url("http://localhost:8181/bpocore/
        asset-manager/api/v1/keys")
    .method("GET",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

## Task 3: Using RESTCONF to manage network devices

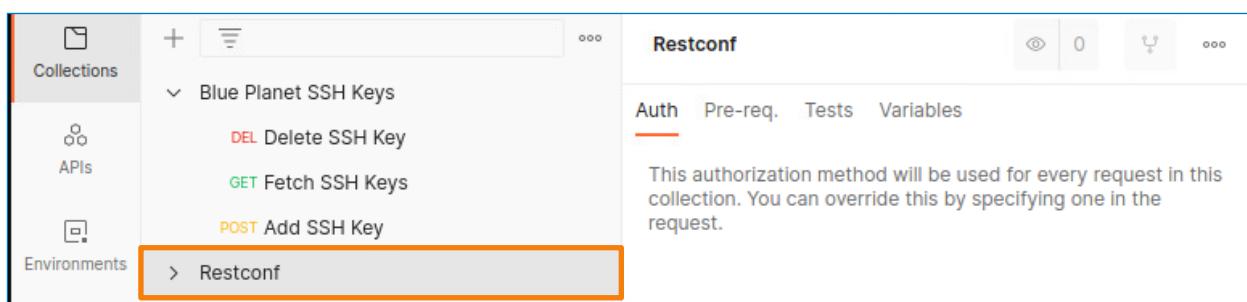
As you learned previously, your lab environment includes **Clixon**, a YANG-based device with CLI, NETCONF, and RESTCONF interface. In this task, you interact with the device through the RESTCONF interface.

- 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.

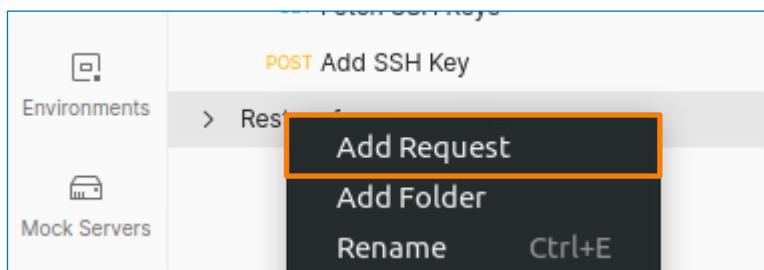
If you closed the Postman application, open it again.



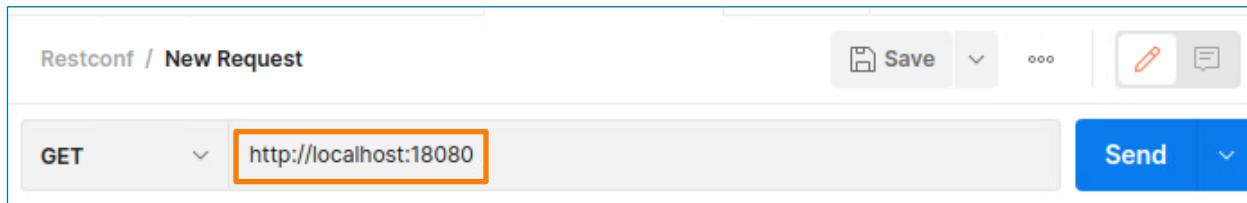
- Create a new collection named **RESTCONF**. Click the **New** button and select **Collection** to create it.



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



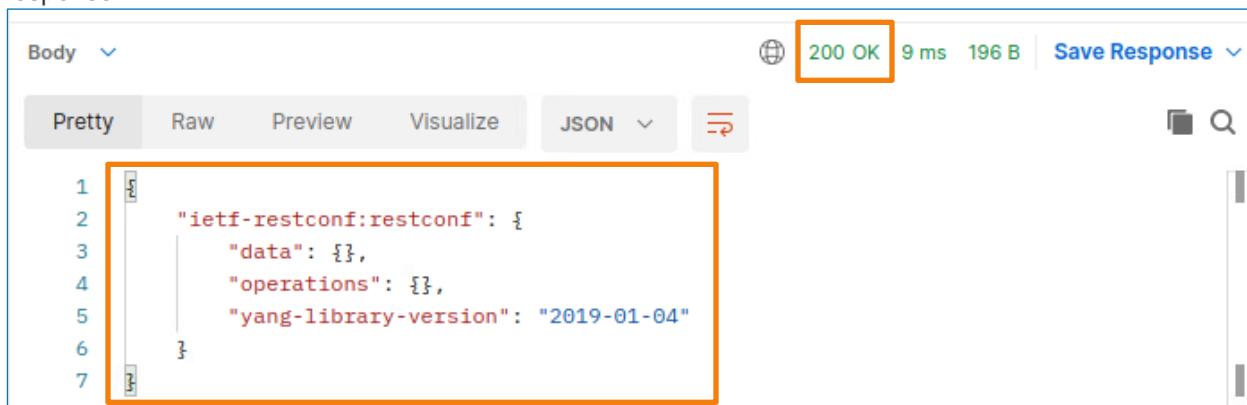
4. You will now construct a RESTCONF request that will fetch data from Clixon. First, you must find the web address of the device that should receive the request. Clixon is deployed as a container directly on the virtual machine that you are accessing so you can use **localhost** as the address. The 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>.



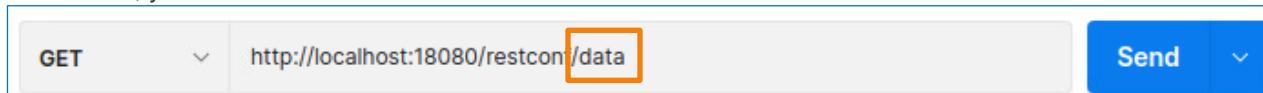
5. First you must check the connectivity and try to reach the RESTCONF interface on the device. Root URI where devices usually listen for RESTCONF requests is /restconf. Add /restconf to your URL.



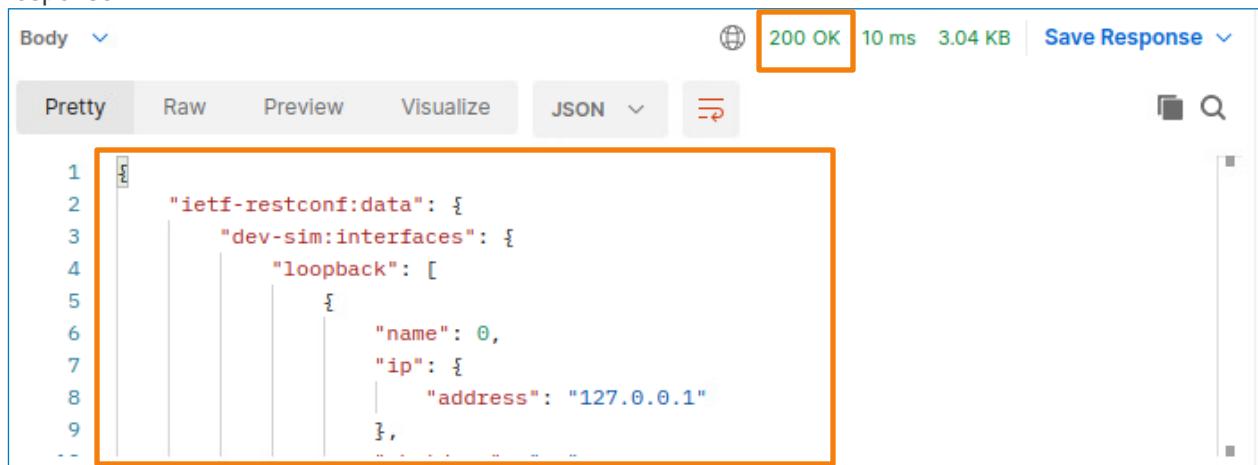
6. Send the request. The request should be successful, and you should receive the following response.



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.



The screenshot shows a REST API response in a browser-like interface. At the top, it says "Body" with a dropdown arrow. To the right, there are status indicators: a globe icon, "200 OK", "10 ms", "3.04 KB", and "Save Response" with a dropdown arrow. Below this, there are tabs: "Pretty", "Raw", "Preview", "Visualize", and "JSON" with a dropdown arrow. The "Pretty" tab is selected. The JSON body is displayed with line numbers on the left. An orange box highlights the first few lines of the JSON output:

```

1
2   "ietf-restconf:data": {
3     "dev-sim:interfaces": [
4       "loopback": [
5         {
6           "name": 0,
7           "ip": {
8             "address": "127.0.0.1"
9           }
10          ...
11        ]
12      ]
13    }
14  }
15 }

```

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;
}
}
}
}

```

- 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.



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

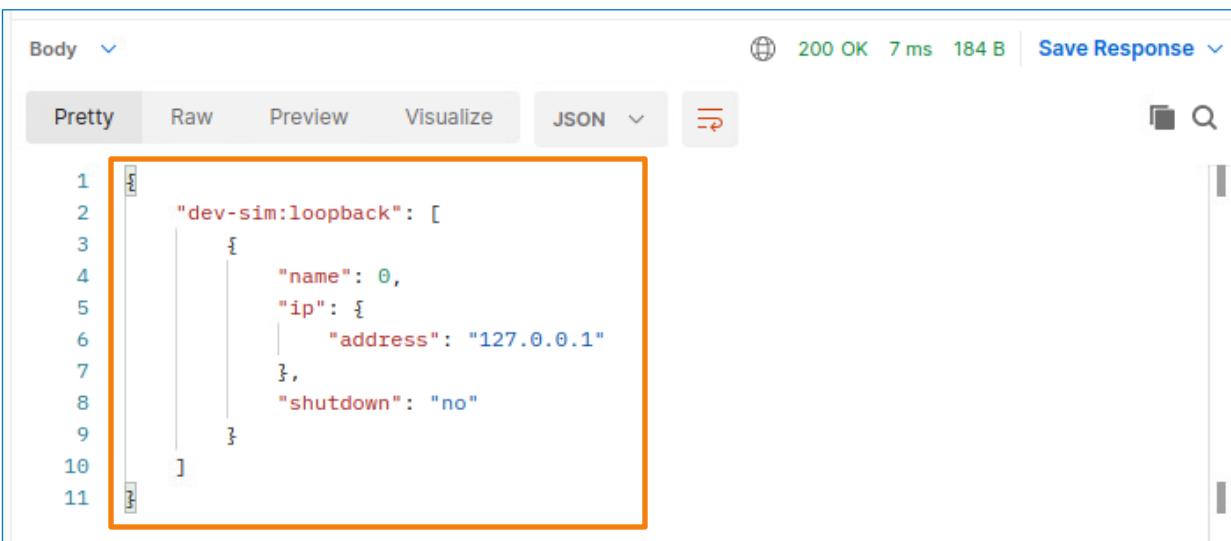


- You can also request more specific data, for example only the IP address of the interface. To do that, you must define the path to the resource that contains the wanted data. In this case, you

receive 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 the configuration data for the interface **loopback 0**.



```

1
2   "dev-sim:loopback": [
3     {
4       "name": 0,
5       "ip": {
6         "address": "127.0.0.1"
7       },
8       "shutdown": "no"
9     }
10  ]
11

```

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.

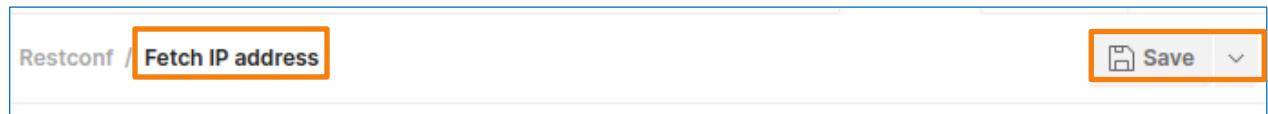


```

1
2   "dev-sim:ip": {
3     "address": "127.0.0.1"
4   }
5

```

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.

```
Code snippet
Java - OkHttp
</>
1 OkHttpClient client = new OkHttpClient().  
    newBuilder()  
    .build();  
2 MediaType mediaType = MediaType.parse  
    ("text/plain");  
3 RequestBody body = RequestBody.create  
    (mediaType, "");  
4 Request request = new Request.Builder()  
    .url("http://localhost:18080/restconf/  
        data/dev-sim:interfaces/loopback=0/  
        ip")  
    .method("GET", body)  
    .build();  
5 Response response = client.newCall  
    (request).execute();
```

---

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.

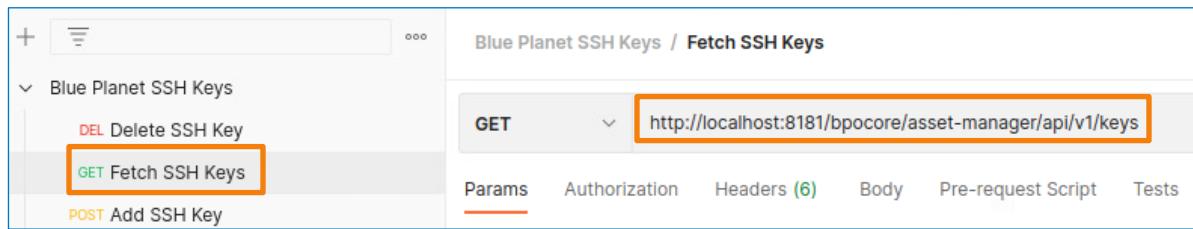
1. In this lab, you will use Postman as a tool that will help you develop the Java code. Open the Postman application.

The screenshot shows the Postman Scratch Pad interface. At the top, there's a navigation bar with 'Overview' and a '+' button. Below it is a section titled 'Scratch Pad' with a cloud icon. A descriptive text states: 'The Scratch Pad is for all your scrappy, exploratory work on Postman. All the data is saved locally on your machine, so only you have access to it. To collaborate in real-time and sync your work, switch to a workspace.' Below this, a heading 'In your Scratch Pad' is followed by three counts: '86 Requests', '3 Collections', and '0 Environments'. The main area is currently empty, showing a light gray background.

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

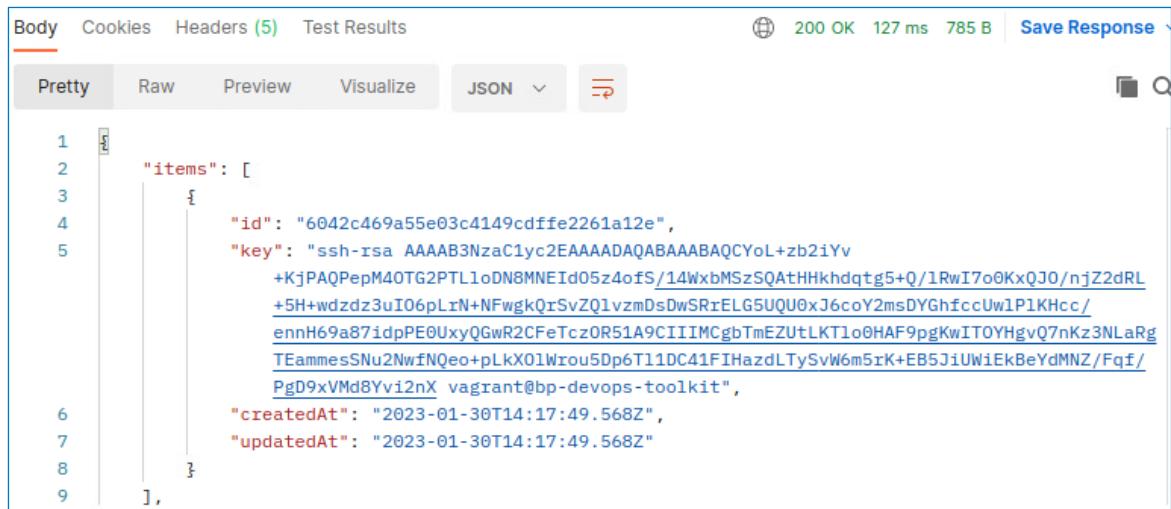
The screenshot shows the Postman Scratch Pad interface with the 'Blue Planet SSH Keys' collection expanded. On the left, there's a sidebar with 'Collections' (highlighted with a red border), 'APIs', and 'Environments'. The main area shows the collection structure with three items: 'Delete SSH Key' (red text, likely a placeholder or error), 'Fetch SSH Keys' (green text), and 'Add SSH Key' (yellow text). There are also 'New' and 'Import' buttons at the top right of the main area.

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



The screenshot shows the Postman interface with the 'Blue Planet SSH Keys' collection expanded. The 'Fetch SSH Keys' request is selected, and its details are shown in the right panel. The URL is highlighted with an orange box: `http://localhost:8181/bpocore/asset-manager/api/v1/keys`.

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.



The screenshot shows the Postman interface displaying the response body of the 'Fetch SSH Keys' request. The response is a JSON object with the following structure:

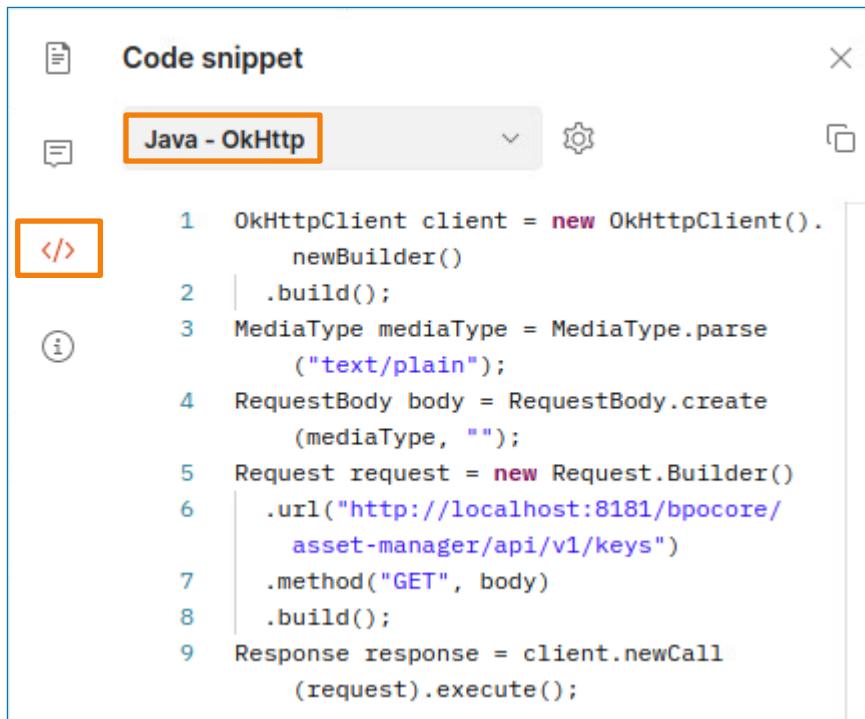
```

1   "items": [
2     {
3       "id": "6042c469a55e03c4149cdffe2261a12e",
4       "key": "ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQCYoL+zB2iYv
+KjPAQPepM40TG2PTL1oDN8MNEId05z4ofS/14WxbMSzSQAtHHkhdtg5+Q/1RwI7o0KxQJ0/njZ2dRL
+5H+wdzdz3uIO6pLrN+NFWgkQrSvZQ1vzmDsDwSRrELG5UQU0xJ6coY2msDYGhfccUwlP1KHcc/
ennH69a87idpPE0UxyQGwR2CFeTczOR51A9CIIIMCgbTmEZUtLKT1o0HAF9pgKwITOYHgvQ7nKz3NLaRg
TEammesSNu2NwfNQeo+pLkX01Wzou5Dp6T11DC41FIHazdLTySvW6m5rK+EB5JiUWiEkBeYdMNZ/Fqf/
PgD9xVMd8Yv12nX vagrant@bp-devops-toolkit",
5       "createdAt": "2023-01-30T14:17:49.568Z",
6       "updatedAt": "2023-01-30T14:17:49.568Z"
7     }
8   ],
9 ]

```

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.



The screenshot shows the Postman interface with the 'Code snippet' tab selected. A dropdown menu at the top is set to 'Java - OkHttp'. Below the dropdown, a code editor displays the following Java code:

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

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.

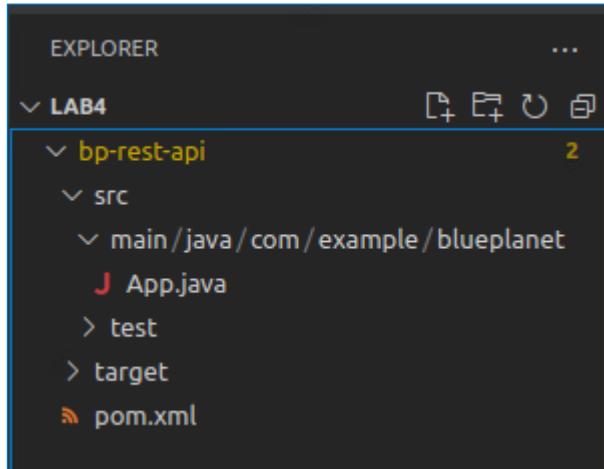
```
[~]$ cd lab4/  
[~/lab4]$ mvn archetype:generate -DgroupId=com.example.blueplanet -  
    DartifactId=bp-rest-api -DarchetypeArtifactId=maven-archetype-  
    quickstart -DinteractiveMode=false  
  
<... output omitted ...>  
[INFO] Parameter: version, Value: 1.0-SNAPSHOT  
[INFO] project created 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 ...>
```

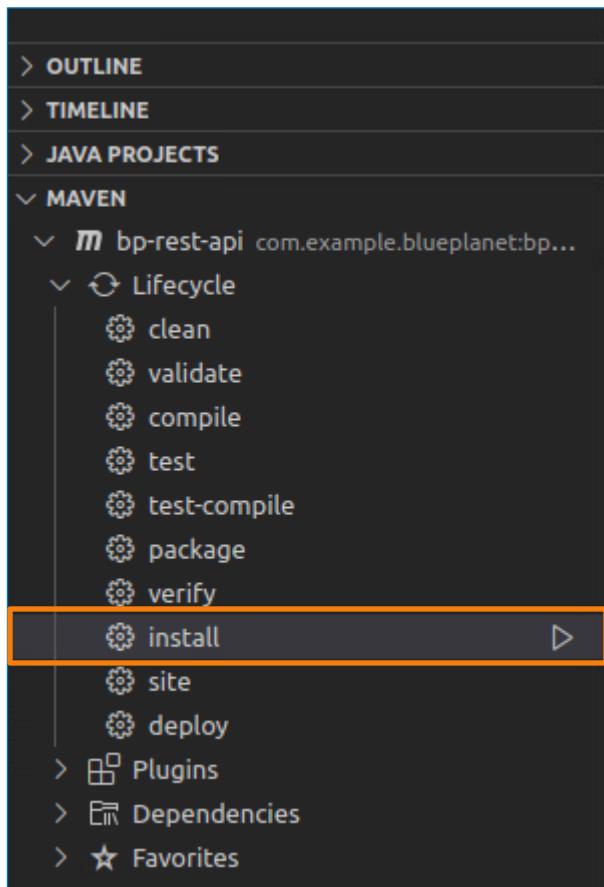
11. Add the following lines to the pom.xml file to change the default maven compiler and save the file.

```
<... 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"



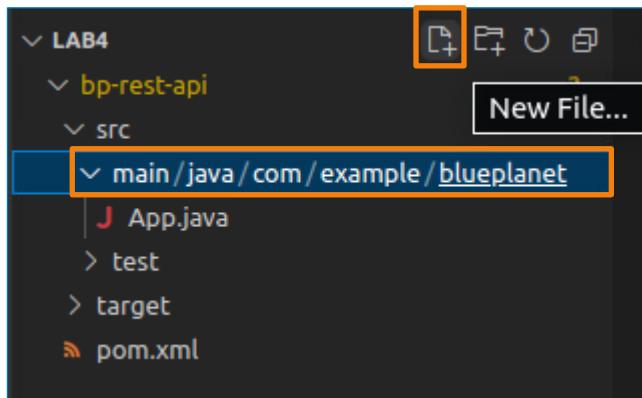
13. Make sure that the project is built successfully.

```
[INFO] --- maven-install-plugin:2.4:install (default-install) @ bp-rest-api ---
[INFO] Installing /home/vagrant/lab4/bp-rest-api/target/bp-rest-api-1.0-SNAPSHOT.jar
[INFO] Installing /home/vagrant/lab4/bp-rest-api/pom.xml to /home/vagrant/.m2/repository
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time:  2.150 s
[INFO] Finished at: 2023-02-01T08:12:22Z
```

## Task 2: Fetching the SSH Keys with Java

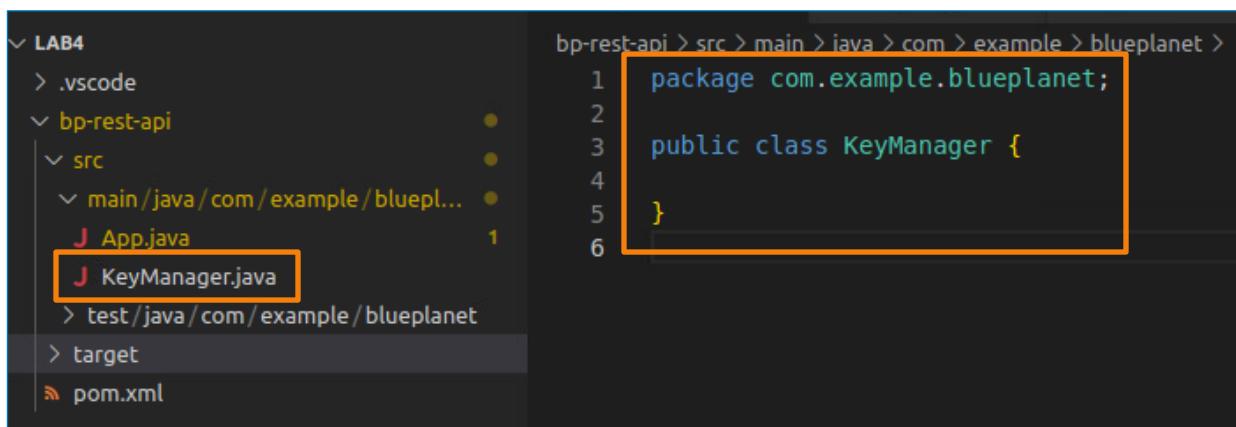
In this task, you create a new class with the **FetchKeys** method that can be used to fetch the list of all the stored SSH keys from the lab server.

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.



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.

**Code snippet**

Java - OkHttp

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

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.

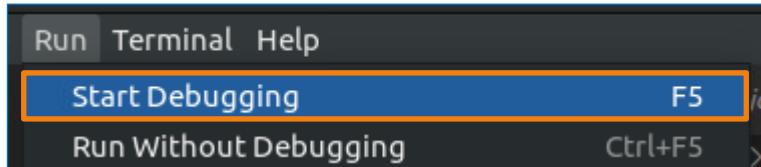
```
package com.example.blueplanet;

import java.io.IOException;

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

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

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



- 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.

```
{
  "items" : [ {
    "id" : "2a1894dc20962fc07af8b0366ecaea69",
    "key" : "ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQCYoL+zb2iYv+KjPAQPeM40TG2PTLloDN8
MNEId05z4ofs/14WxbMSzS0AtHHkdqtg5+0/lRwI7o0KxQJ0/njZ2dRL+5H+wdzdz3uI06pLrN+NFWgkQrSv
ZQlvzmDsDwSRrELG5UQU0xJ6coY2msDYGhfcUwlPlKHcc/ennH69a87idpPE0UxyQGwR2CFeTcz0R51A9CII
IMCgbTmEZUTLKTlo0HAF9pgKwITOYHgvQ7nKz3NLaRgTEammesSNu2NwfNQeo+pLkX0lWrou5Dp6Tl1DC41FI
HazdLTySvW6m5rK+EB5JiUWiEkBeYdMNZ/Fqf/PgD9xVmD8Yvi2n",
    "createdAt" : "2023-02-01T13:11:46.502Z",
    "updatedAt" : "2023-02-01T13:11:46.502Z"
  }],
  "limit" : 1000
}
vagrant@bp-devops-toolkit:~/lab4$ 
```

### 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;

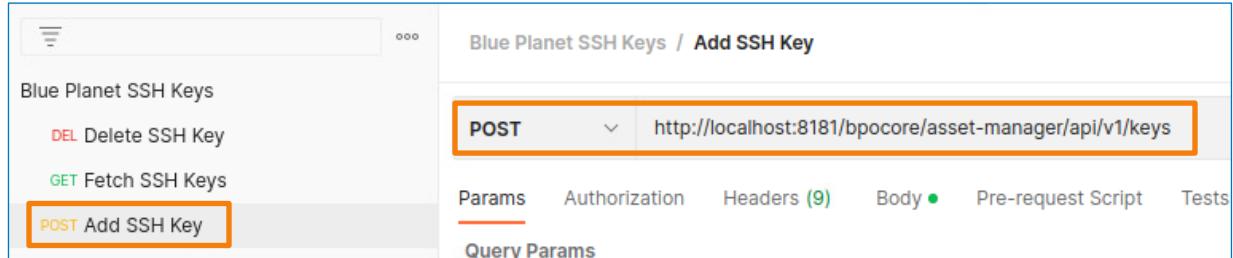
import java.io.IOException;

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());
    }

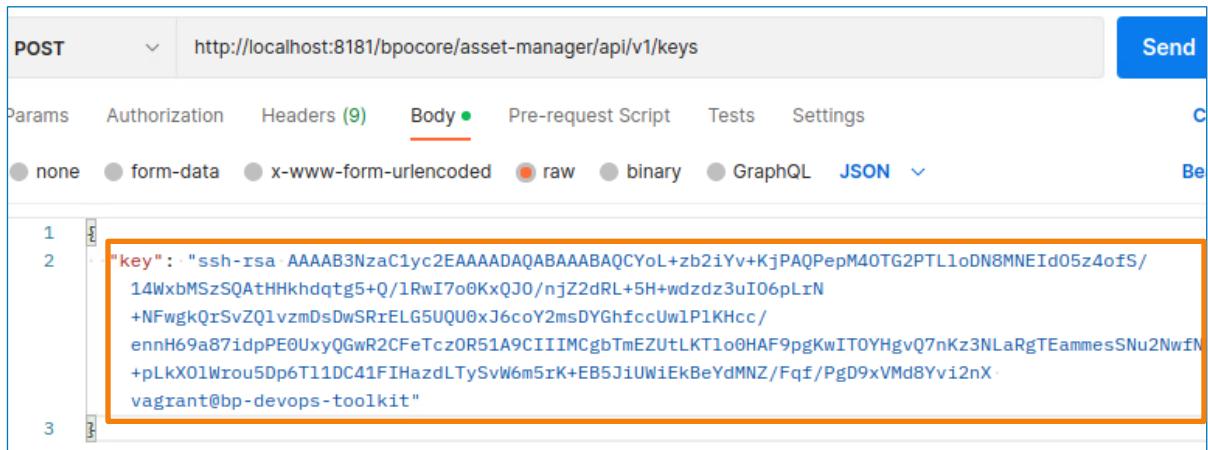
    public void PostKeys() throws IOException {
    }
}
```

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.

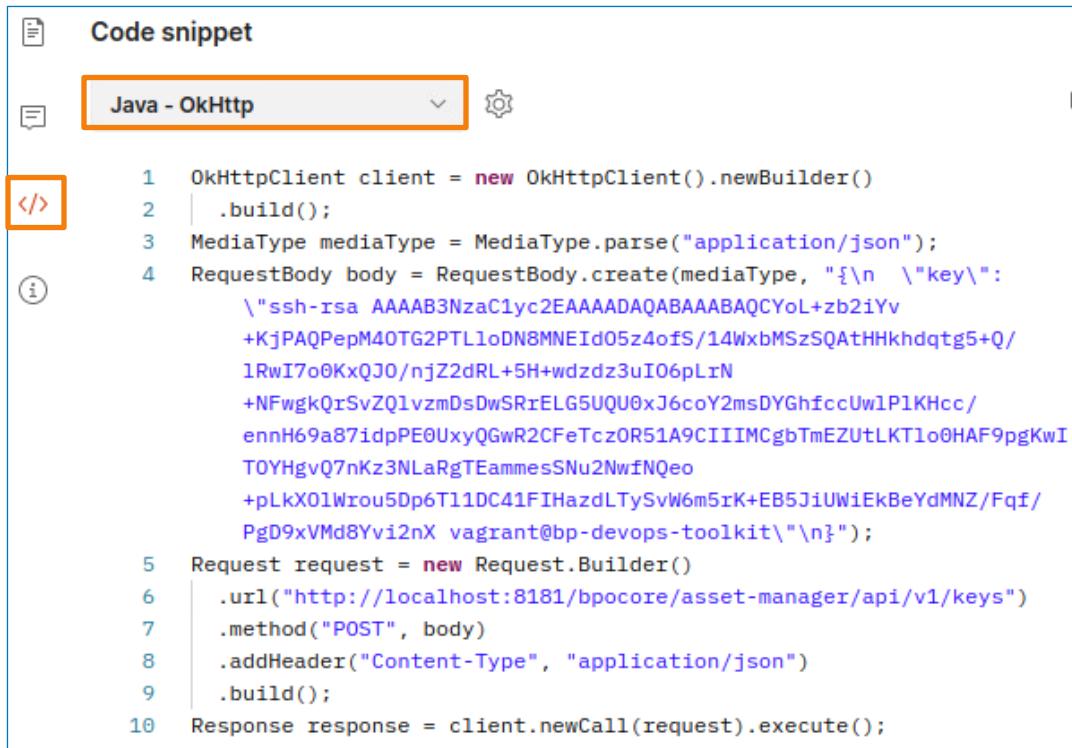


The screenshot shows the Postman interface with a POST request to `http://localhost:8181/bpocore/asset-manager/api/v1/keys`. The **Body** tab is selected, showing a JSON payload with a `key` field containing a long public key string. The public key is highlighted with an orange box.

```

1  "key": "ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQCYoL+zB2iYv+KjPAQPepM40TG2PTLloDN8MNEId05z4ofS/
2   14WxbMSzSQAtHHkhdqtg5+0/RwI7o0KxQJ0/njZ2dRL+5H+wdzd3uI06pLrN
3   +NFwgkQrSvZQ1vzmDsDwSRrELG5UQU0xJ6coY2msDYGhfccUwlP1KHcc/
4   ennH69a87idpPE0UxyQGwR2CFeTczOR51A9CIIIMCgbTmEZUtLKTlo0HAF9pgKwITOYHgvQ7nKz3NLaRgTEammesSNu2NwfN
5   +pLkX01Wrou5Dp6Tl1DC41FIHazdLTySvW6m5rK+EB5JiUWiEkBeYdMNZ/Fqf/PgD9xVMd8Yvi2nX
6   vagrant@bp-devops-toolkit"
7
8
9
10
  
```

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



The screenshot shows the 'Code snippet' panel in Postman, displaying Java code generated for the previous request. The code uses the OkHttp library to build a POST request with a JSON body containing the public key. The code is highlighted with an orange box.

```

1 OkHttpClient client = new OkHttpClient().newBuilder()
2     .build();
3 MediaType mediaType = MediaType.parse("application/json");
4 RequestBody body = RequestBody.create(mediaType, "{\"key\": \"ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQCYoL+zB2iYv
5   +KjPAQPepM40TG2PTLloDN8MNEId05z4ofS/14WxbMSzSQAtHHkhdqtg5+0/
6   1RwI7o0KxQJ0/njZ2dRL+5H+wdzd3uI06pLrN
7   +NFwgkQrSvZQ1vzmDsDwSRrELG5UQU0xJ6coY2msDYGhfccUwlP1KHcc/
8   ennH69a87idpPE0UxyQGwR2CFeTczOR51A9CIIIMCgbTmEZUtLKTlo0HAF9pgKw
9   TOYHgvQ7nKz3NLaRgTEammesSNu2NwfNQeo
10  +pLkX01Wrou5Dp6Tl1DC41FIHazdLTySvW6m5rK+EB5JiUWiEkBeYdMNZ/Fqf/
11  PgD9xVMd8Yvi2nX vagrant@bp-devops-toolkit\"}");
12 Request request = new Request.Builder()
13     .url("http://localhost:8181/bpocore/asset-manager/api/v1/keys")
14     .method("POST", body)
15     .addHeader("Content-Type", "application/json")
16     .build();
17 Response response = client.newCall(request).execute();
  
```

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.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());
    }

    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\nAAAAB3NzaC1yc2EAAAQABAAQCYoL+zb2iYv+KjPAQPepM40TG2PTLloDN8MNEI
d05z4ofS/14WxbMSzSQAtHHkhdtg5+Q/lRwI7o0KxQJ0/njZ2dRL+5H+wdzdz3uI06p
LrN+NFWgkQrSvZQ1vzmDsDwSRrELG5UQU0xJ6coY2msDYGhfccUwlPlKHcc/ennH69a8
7idpPE0UxyQGwR2CFeTcz0R51A9CIIIMCgbTmEZUtLKTlo0HAF9pgKwITOYHgvQ7nKz3
NLaRgTEammesSNu2NwfNQeo+pLkX0lWrou5Dp6Tl1DC41FIHazdLTySvW6m5rK+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.RequestBody;
import okhttp3.OkHttpClient;
import okhttp3.Request;
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\":\n        \"ssh-rsa\nAAAAB3NzaC1yc2EAAAQABAAQCYoL+zb2iYv+KjPAQPepM40TG2PTLloDN8MNEI\n        d05z4ofS/14WxbMSzSQAtHHkhdtg5+Q/lRwI7o0KxQJ0/njZ2dRL+5H+wdzdz3uI06p\n        LrN+NFWgkQrSvZQ1vzmDsDwSRrELG5UQU0xJ6coY2msDYGhfccUwlP1KHcc/ennH69a8\n        7idpPE0UxyQGwR2CFeTczOR51A9CIIIMCgbTmEZUtLKT1o0HAF9pgKwIT0YHgvQ7nKz3\n        NLaRgTEammesSNu2NwfNQeo+pLkX0lWrou5Dp6T11DC41FIHazdLTySvW6m5rK+EB5Ji\n        UWiEkBeYdMNZ/Fqf/PgD9xVMd8Yvi2nX vagrant@bp-devops-toolkit\\\"\\n\\\"}");\n\n    Request request = new Request.Builder()\n        .url("http://localhost:8181/bpocore/asset-manager/api/v1/keys")\n        .method("POST", body)\n        .addHeader("Content-Type", "application/json")\n        .build();\n\n    Response response = client.newCall(request).execute();\n    System.out.println(response.body().string());\n}
```

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;

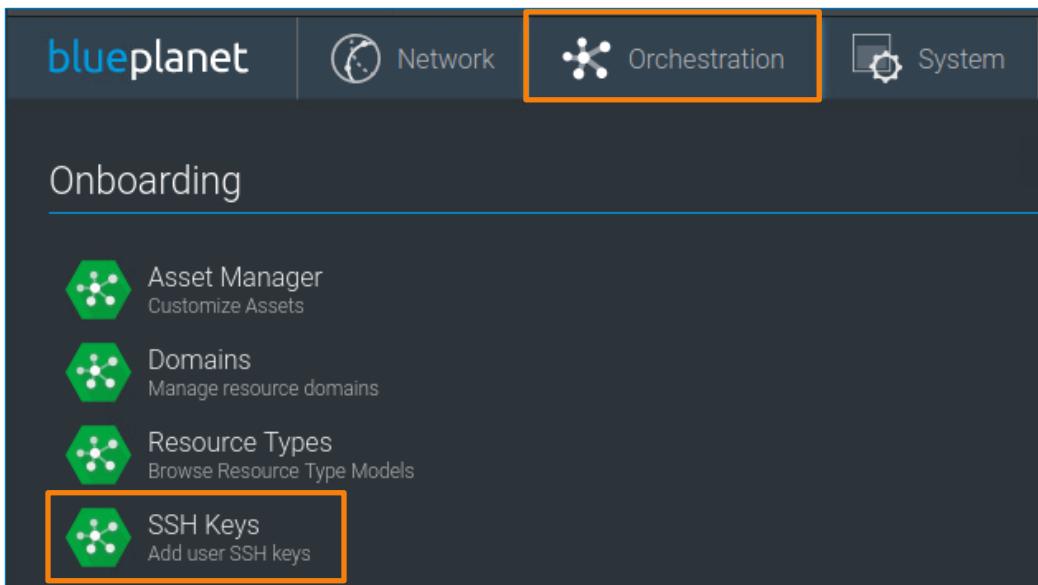
import java.io.IOException;

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.

```
{
  "id" : "6042c469a55e03c4149cdffe2261a12e",
  "key" : "ssh-rsa AAAAB3NzaC1yc2EAAQABAAQCYoL+zB2iYv+KjPAQPepM40TG2PTLloDN8MNEid05z4ofS/14WxbMSzSQAtHHkdqtg5+Q/l
RwI7o0KxQJ0/njZ2dRL+5H+wdzd23uI06pLrN+NfWgkQrsvZQlvzmDsDwSRrELG5UQU0xJ6coY2msDYGhfcUwLPlKhcc/ennh69a87idpPE0UxyQGwR2CFcET
cz0R51A9CIIIMCbTmEZUtLKTlo0HAF9pgKwIT0YHgvQ7nKz3NLaRgTEammesSNu2NwfNQeo+pLkX0lWrou5Dp6Tl1DC41FIHazdLTySvw6m5rK+EB5JiUW1E
kBeYdMNZ/Fqf/PgD9xVm8Yyi2nX vagrant@bp-devops-toolkit",
  "createdAt" : "2023-02-01T15:08:15.190Z",
  "updatedAt" : "2023-02-01T15:08:15.190Z"
}
vagrant@bp-devops-toolkit:~/lab4$ 
```

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.

**NOTE:** The UI may not display the full length of the SSH key that you added unless you click the “Expand all” button.



The screenshot shows a list of SSH keys. There are two keys listed, each preceded by a checkbox. The first key's value is partially visible: "ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQCYoL+z...". The second key's value is fully visible: "ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQCYoL+z...". Below the second key, there is a "Copy" button with a clipboard icon. The entire list area is enclosed in a blue border.

```

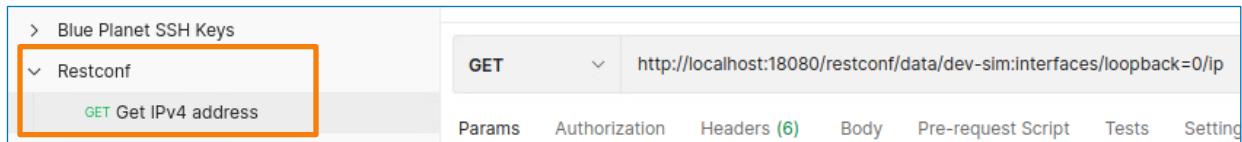
 ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQCYoL+zb2iYv+KJPAQPepM40TG2PTLioDN8MNEld05z4ofS/14WxbMSzSQAtHHkhdqg5+Q/IrwI7o0KxQJO/njZ2dRL+5H+wdzdz3ui0
6pLrN+NfwgkQrSvZQlvzmDsDwSrELG5UQU0xJ6coY2msDY
 ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQCYoL+zb2iYv+KJPAQPepM40TG2PTLioDN8MNEld05z4ofS/14WxbMSzSQAtHHkhdqg5+Q/IrwI7o0KxQJO/njZ2dRL+5H+wdz
d23uI06pLrN+NfwgkQrSvZQlvzmDsDwSrELG5UQU0xJ6coY2msDYGhfccUwlPIKHcc/ennH69a87idpPEOUxyQGwR2CFeTczOR51A9CIIIMCgbTmEZUtLKTlo0HAF9pgKwITOVHg
v07nKz3NLaRgTEammesSNu2NwfNOeo+pLkXOIWrou5Dp6Tl1DC41FIHazdLTySvW6m5rK+EB5JIUWEkBeYdMNZ/Fqf/PgD9xVmD8Yvi2nX vagrant@bp-devops-toolkit

```

## Task 4: Manage RESTCONF device with Java

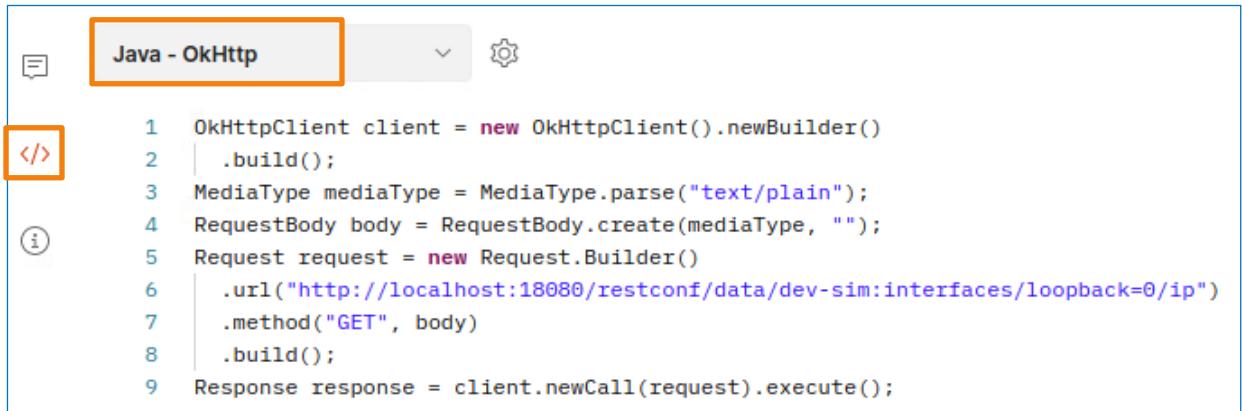
Similarly, you can use Postman to create code snippets for sending the HTTP requests for managing a RESTCONF device, that you created in the previous lab. In this task, you will develop a new method that will read the IPv4 address on the RESTCONF device.

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



The screenshot shows the Postman interface with a collection named "Restconf". Under "Restconf", there is a request named "Get IPv4 address". The URL is set to `http://localhost:18080/restconf/data/dev-sim:interfaces/loopback=0/ip`. The "Headers (6)" tab is selected. Other tabs include Params, Authorization, Body, Pre-request Script, Tests, and Setting.

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



```

</>
1 OkHttpClient client = new OkHttpClient().newBuilder()
2     .build();
3 MediaType mediaType = MediaType.parse("text/plain");
4 RequestBody body = RequestBody.create(mediaType, "");
5 Request request = new Request.Builder()
6     .url("http://localhost:18080/restconf/data/dev-sim:interfaces/loopback=0/ip")
7     .method("GET", body)
8     .build();
9 Response response = client.newCall(request).execute();

```

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.



The screenshot shows the VS Code file structure for LAB4. It includes a folder for `bp-rest-api` containing `src`, `test`, `target`, and `pom.xml`. Inside `src/main/java/com/example/blueplanet`, there are files `App.java`, `DeviceManager.java` (highlighted with an orange box), and `KeyManager.java`. The `DeviceManager.java` content is:

```

1 package com.example.blueplanet;
2
3 public class DeviceManager {
4
5 }
6

```

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.

```
file com.example.blueplanet.App  
{"dev-sim:ip":{"address":"127.0.0.1"}  
Vagrant@bp-devops-TOOLKIT:~/Lab4$
```

---

Lab End

## Lab 6: Camunda Introduction

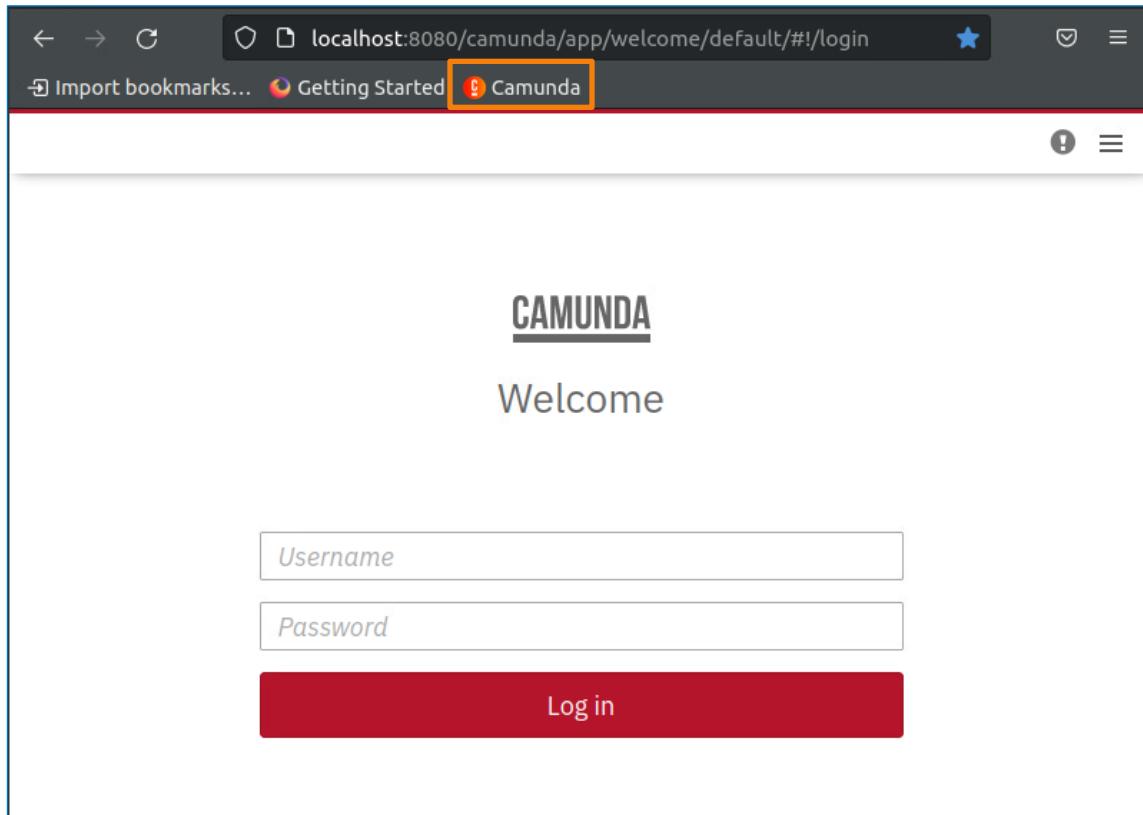
### Objectives

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

### Task 1: Exploring the Camunda Platform

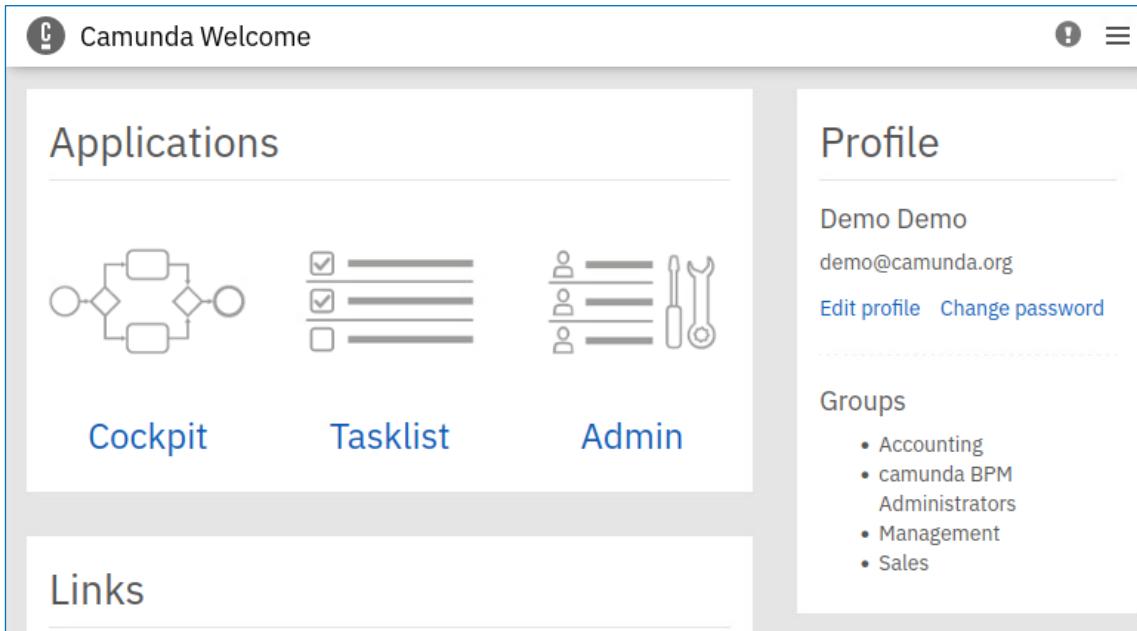
In this task, you explore the Camunda platform that is deployed in your Lab environment and learn about different components of the platform.

1. In this case, Camunda is deployed in the 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.



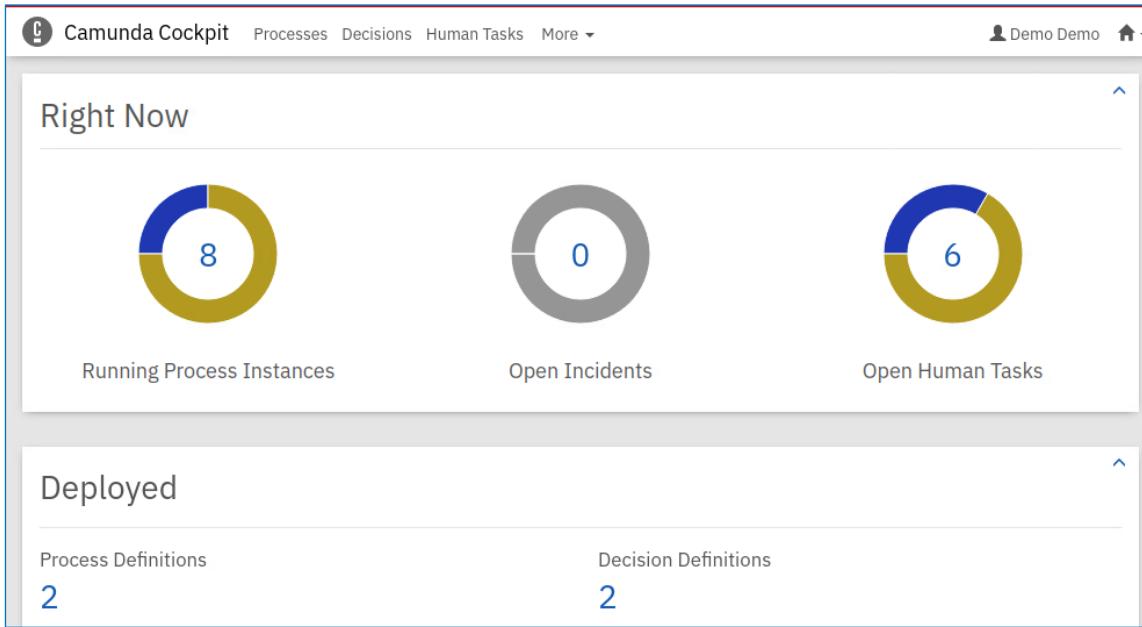
2. Login with the following credentials to access the Camunda homepage:

- Username: **demo**
- Password: **demo**



The screenshot shows the Camunda Welcome page. At the top left is the "Camunda Welcome" logo. On the right are three small icons: a person, a gear, and a list. Below the header is a "Applications" section with three items: "Cockpit" (represented by a process diagram icon), "Tasklist" (represented by a checklist icon), and "Admin" (represented by a user and wrench icon). To the right is a "Profile" sidebar showing "Demo Demo" and "demo@camunda.org", with links to "Edit profile" and "Change password". Below the profile is a "Groups" section listing "Accounting", "camunda BPM Administrators", "Management", and "Sales". At the bottom left is a "Links" section.

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.



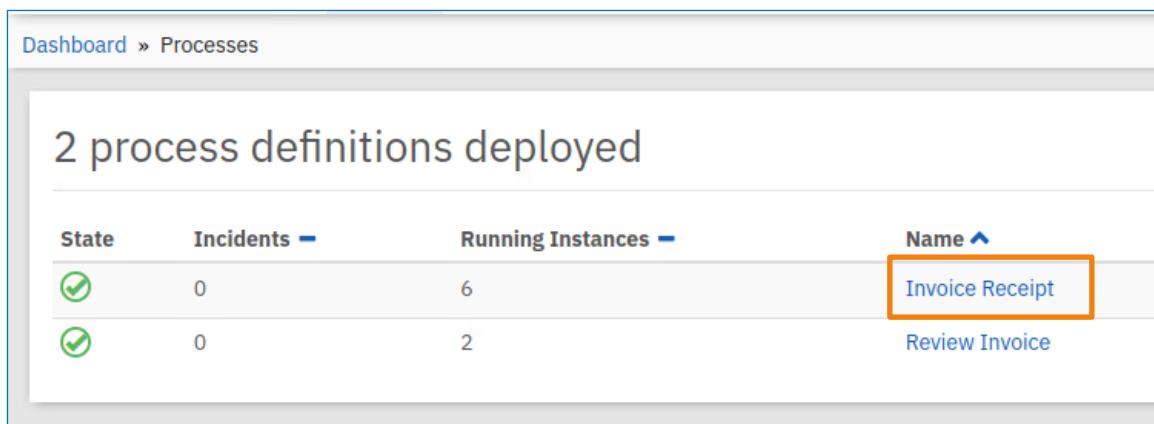
The screenshot shows the Camunda Cockpit dashboard. At the top is a navigation bar with the "Camunda Cockpit" logo, "Processes", "Decisions", "Human Tasks", and a "More" dropdown. On the right is a user profile for "Demo Demo" and a home icon. The main area has a "Right Now" section with three donut charts: "Running Process Instances" (8), "Open Incidents" (0), and "Open Human Tasks" (6). Below this is a "Deployed" section showing "Process Definitions" (2) and "Decision Definitions" (2).

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



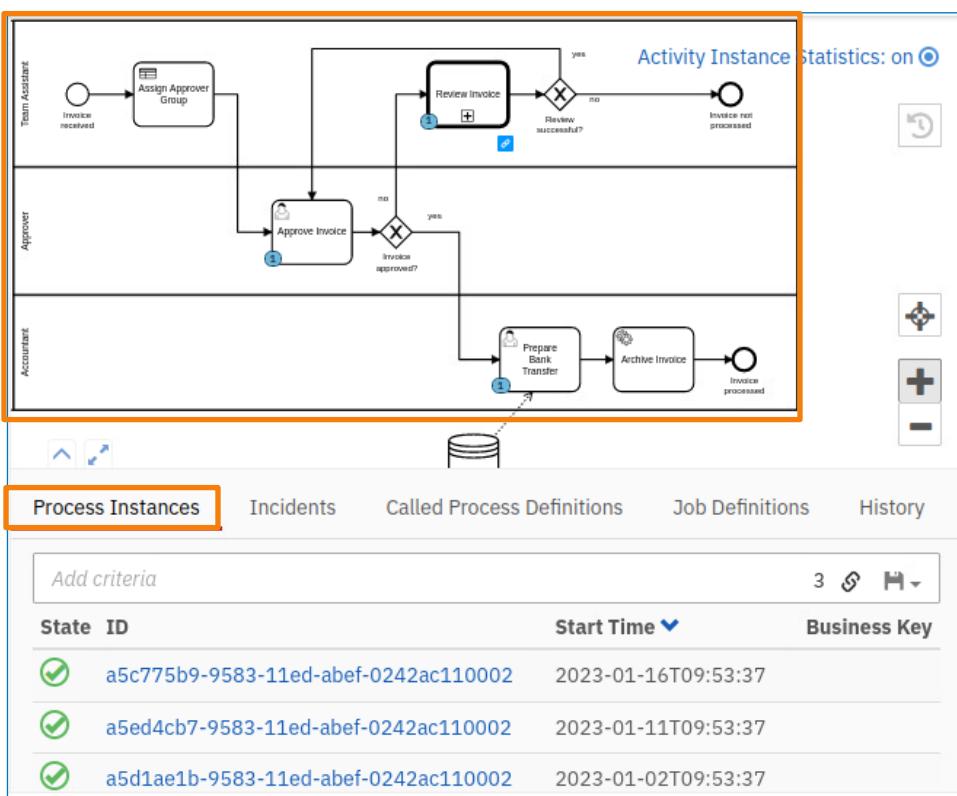
The screenshot shows the Camunda Cockpit menu bar. The tabs are "Processes" (highlighted with an orange border), "Decisions", "Human Tasks", and "More".

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.



The screenshot shows the Camunda BPMN process definition for 'Invoice Receipt'. The process starts with 'Team Assistant' receiving an 'Invoice received' event, which triggers the 'Assign Approver Group' task. This leads to the 'Approver' stage, where the 'Review invoice' task is performed. A decision diamond follows, with 'yes' leading to 'Invoice not processed' and 'no' leading back to the 'Review invoice' task. From the 'no' path, it goes to the 'Accountant' stage, where 'Approve Invoice' is checked. Another decision diamond follows, with 'yes' leading to 'Prepare Bank Transfer' and 'no' leading back to 'Approve Invoice'. Finally, 'Archive Invoice' is completed, marking the process as 'Invoice processed'.

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.

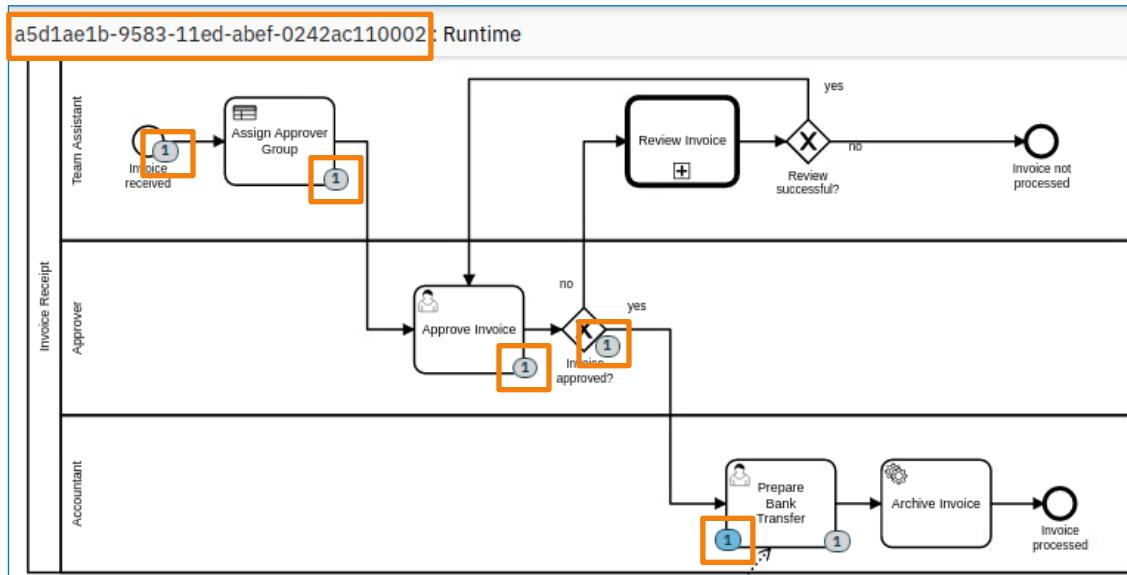


The screenshot shows the Camunda process instance list for the 'Invoice Receipt' process. It displays three active instances:

State	ID	Start Time	Business Key
	a5c775b9-9583-11ed-abef-0242ac110002	2023-01-16T09:53:37	
	a5ed4cb7-9583-11ed-abef-0242ac110002	2023-01-11T09:53:37	
	a5d1ae1b-9583-11ed-abef-0242ac110002	2023-01-02T09:53:37	

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 have already been performed, which decisions have been made, and what the next step of the process is.

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

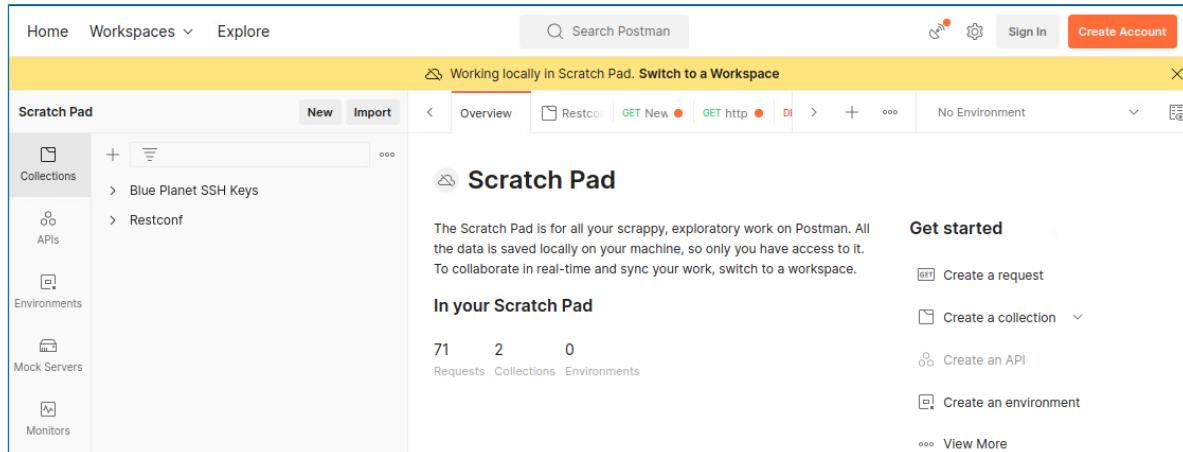


8. By opening the variables menu you can also check which variables were used or calculated in the process. For example, the value of the **approved** variable is **true** which is why the flow continued from the **invoice approved?** step to the **Prepare Bank Transfer** step. You can also change the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

Variables	Incidents	Called Process Instances	User Tasks	Jobs	Audit Log	External Tasks
<i>Add criteria</i>						
Name ▾	Type ▾					Value
amount	Double					900
approved	Boolean					true

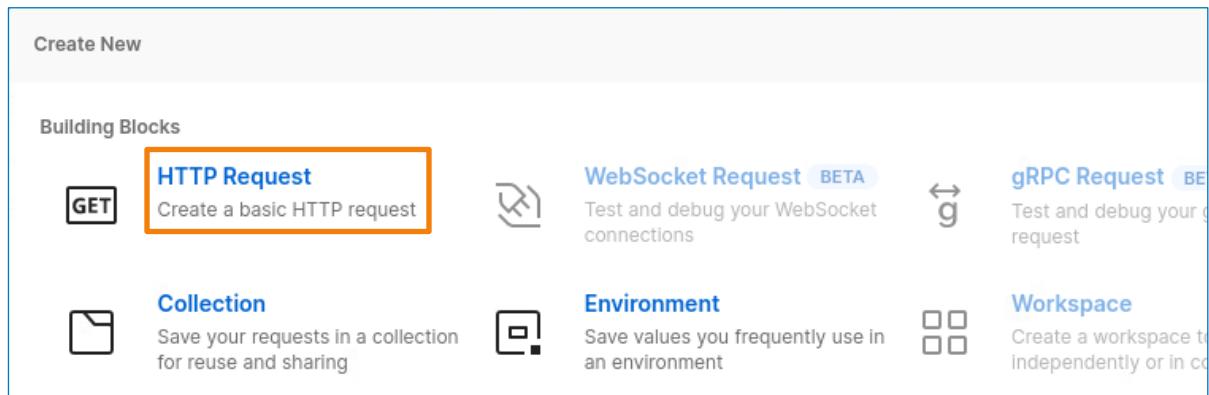
- As mentioned, the Cockpit application provides features to monitor and manage processes that are deployed on the Camunda engine. The application communicates with the engine through a REST API and all the functions that can be performed through the Cockpit web UI can also be performed through the API 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.



The screenshot shows the Postman application interface. At the top, there are tabs for Home, Workspaces, and Explore, along with a search bar and user authentication options. Below the header, a yellow banner says "Working locally in Scratch Pad. Switch to a Workspace". The main area is titled "Scratch Pad" and contains sections for Collections, APIs, Environments, Mock Servers, and Monitors. On the right side, there's a sidebar with "Get started" links for Create a request, Create a collection, Create an API, Create an environment, and View More.

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



The screenshot shows the "Create New" screen in Postman. Under the heading "Building Blocks", there are several options: "HTTP Request" (highlighted with an orange box), "WebSocket Request" (BETA), "gRPC Request" (BETA), "Collection", "Environment", and "Workspace". Each option has a brief description and a corresponding icon.

- 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>.



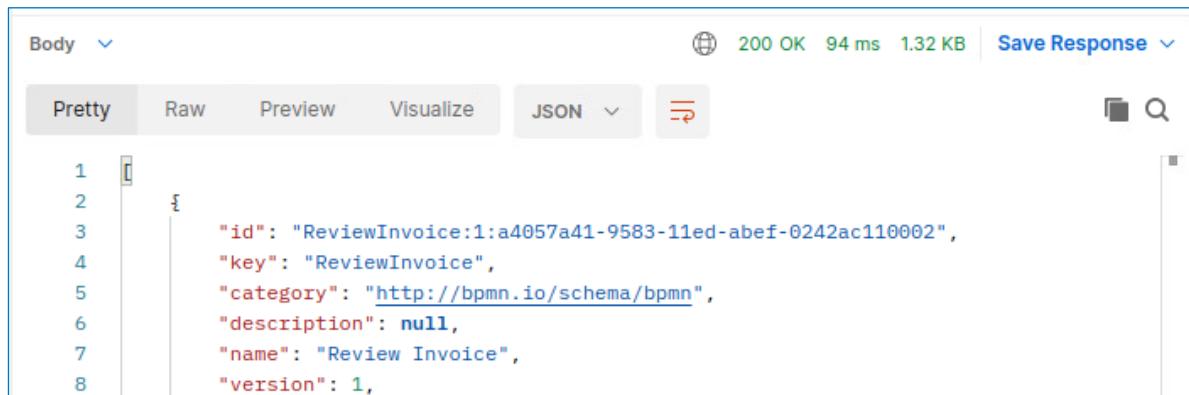
The screenshot shows the Postman request configuration screen. The method dropdown is set to "GET" and the URL field contains "http://localhost:8080/engine-rest". A blue "Send" button is visible on the right.

- 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.



The screenshot shows the Postman request configuration screen. The method dropdown is set to "GET" and the URL field contains "http://localhost:8080/engine-rest/process-definition". A blue "Send" button is visible on the right.

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



The screenshot shows a REST client interface with the following details:

- Body** dropdown is selected.
- HTTP status: 200 OK
- Time: 94 ms
- Size: 1.32 KB
- Save Response button is available.
- Pretty tab is selected.
- Raw, Preview, Visualize, and JSON tabs are present.
- JSON dropdown is set to "Pretty".
- The JSON response is displayed:

```

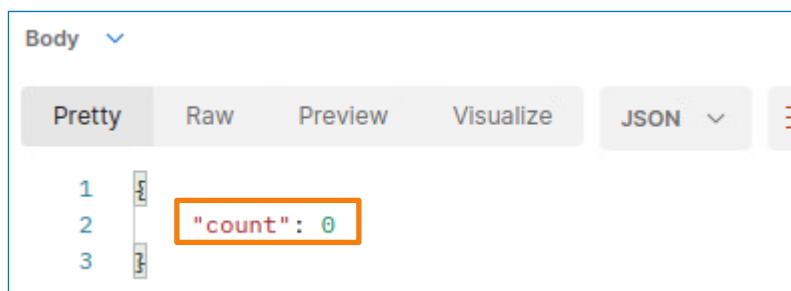
1
2   {
3     "id": "ReviewInvoice:1:a4057a41-9583-11ed-abef-0242ac110002",
4     "key": "ReviewInvoice",
5     "category": "http://bpmn.io/schema/bpmn",
6     "description": null,
7     "name": "Review Invoice",
8     "version": 1,

```

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.



The screenshot shows a REST client interface with the following details:

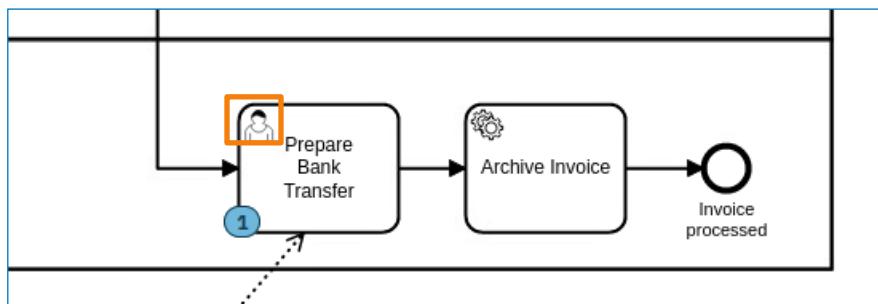
- Body dropdown is selected.
- Pretty tab is selected.
- Raw, Preview, Visualize, and JSON tabs are present.
- JSON dropdown is set to "Pretty".
- The JSON response is displayed:

```

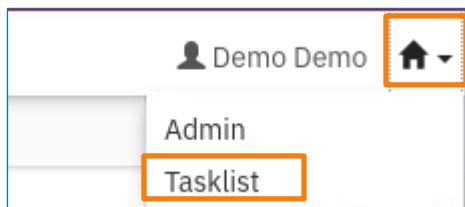
1   {
2     "count": 0
3   }

```

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.

Task Details	Assignee	Due Date
Assign Reviewer	Demo Demo	50
Review Invoice	Demo Demo	Created a day ago

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.

Task Details	Assignee	Due Date
Assign Reviewer	Demo Demo	50
Review Invoice	Demo Demo	Created a day ago

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.

Demo

John

Peter

Mary

Demo

Who should review this invoice?

Save Complete

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

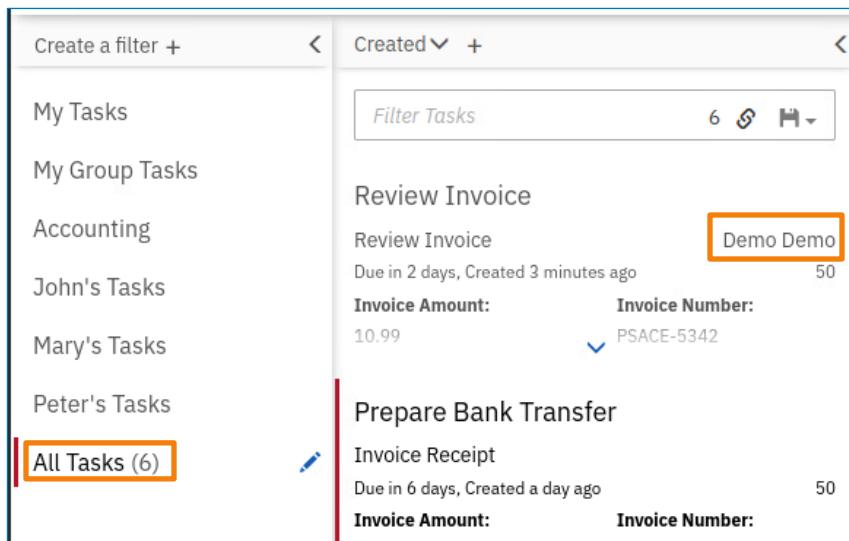
You are assigned to the following tasks in the same process : X

Review Invoice

Review Invoice Due in 2 days, Created 2 minutes ago Demo Demo

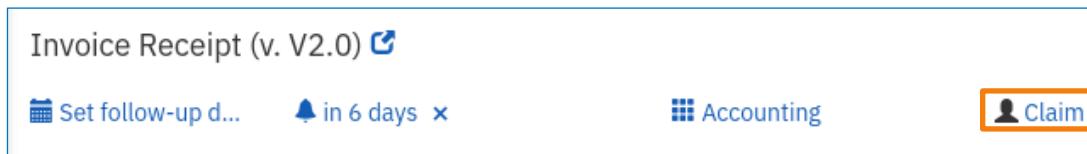
Invoice Amount: 10.99 Invoice Number: PSACE-5342

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.



Task Details	Assignee
Review Invoice Due in 2 days, Created 3 minutes ago Invoice Amount: 10.99 Invoice Number: PSACE-5342	Demo Demo
Prepare Bank Transfer Due in 6 days, Created a day ago Invoice Amount:	

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.



**Invoice Receipt (v. V2.0)** 

 Set follow-up d...     in 6 days 

 Accounting

 Claim

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]$
```

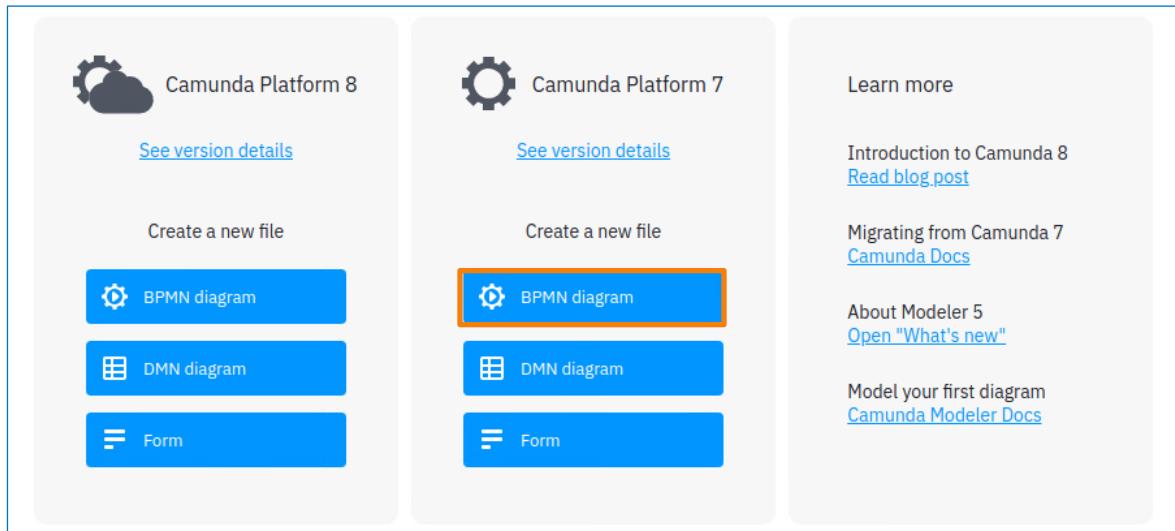
26. Start the Modeler application by executing the **./camunda-modeler** command.

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

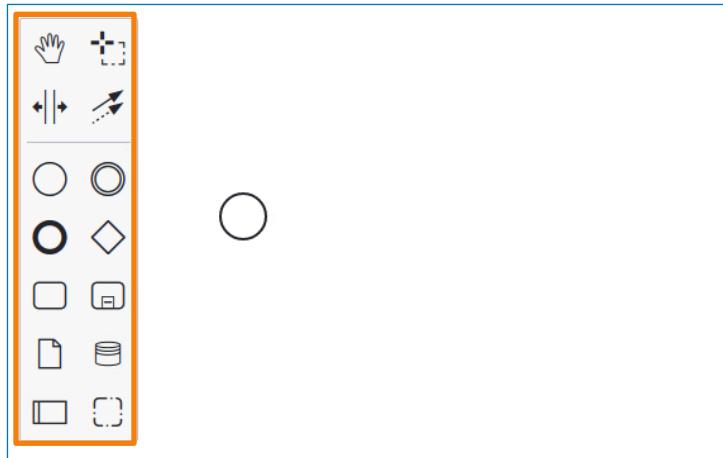
27. Camunda Modeler should open. The application can be used for designing BPMN and DMN diagrams. In this case, you will focus on the BPMN workflows. Click the **BPMN diagram** button and make sure that you choose the **Camunda Platform 7 version** since your lab environment runs this version of the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

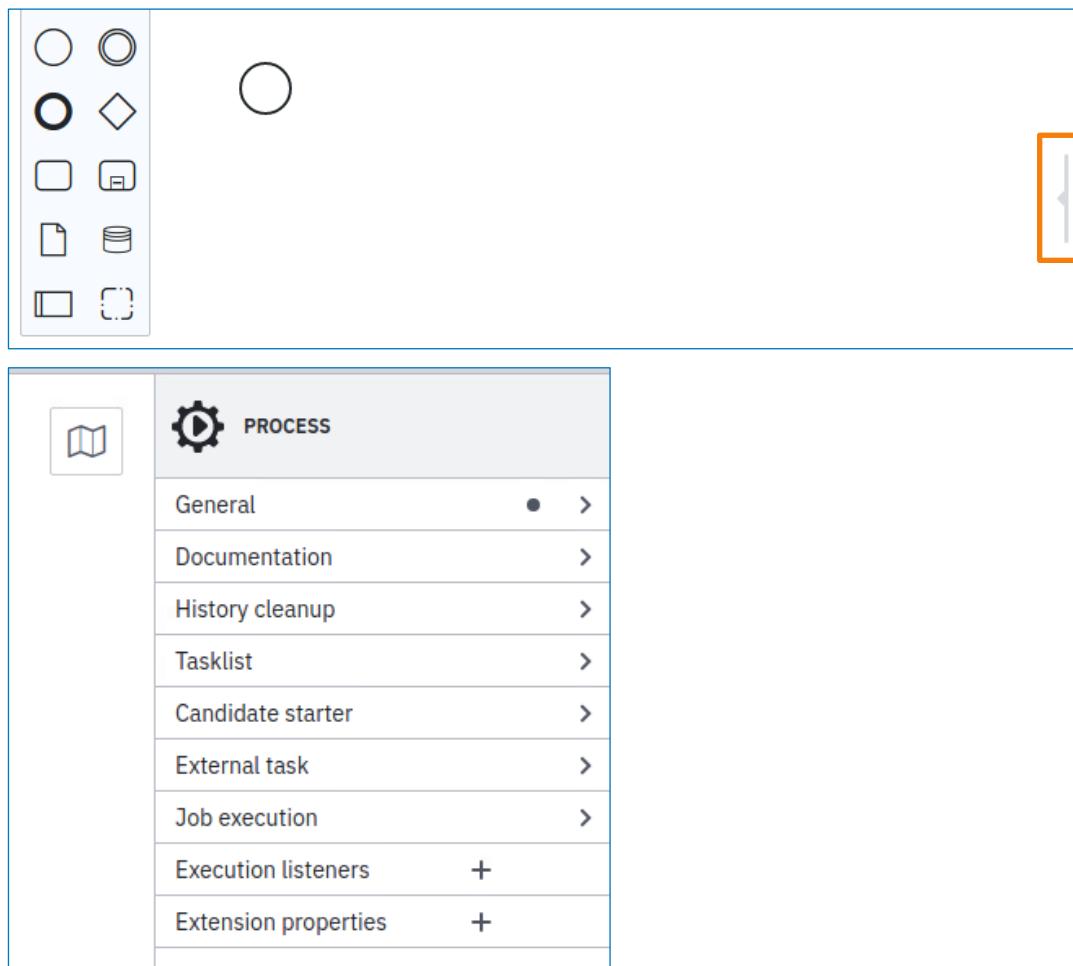


28. A graphical interface for designing BPMN diagrams should open. The menu on the left side of the screen includes icons that represent different actions. To construct the workflow you can simply drag and drop those icons on the screen and connect them.



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 right side of the editor.



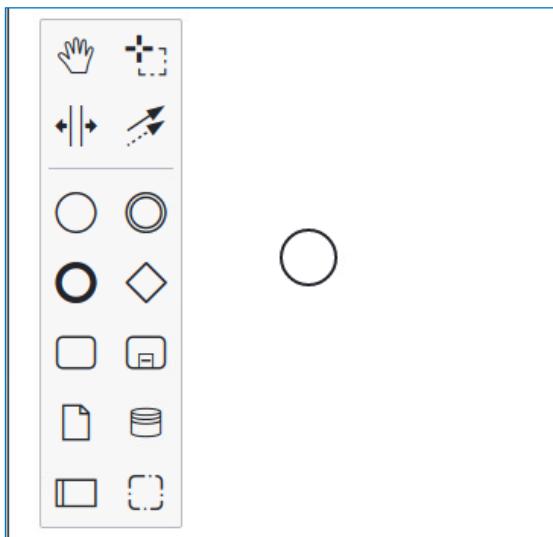
30. When the process is created you can deploy it to Camunda using the **Deploy** button in the Modeler app.



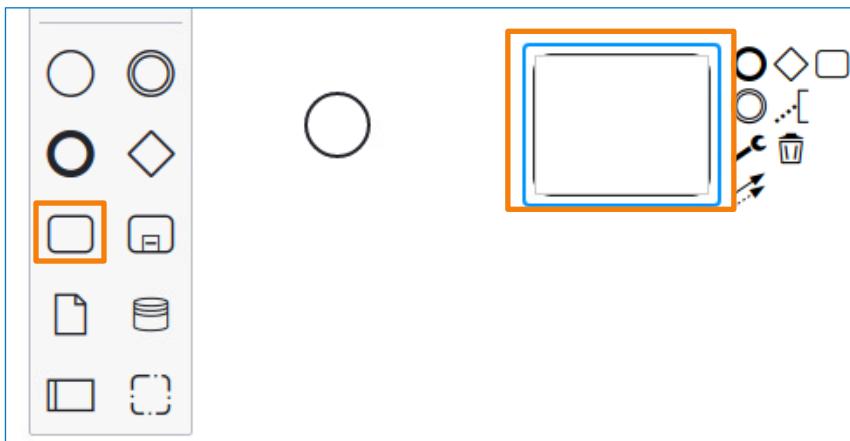
## Task 2: Creating and Deploying Custom Camunda BPMN Process

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

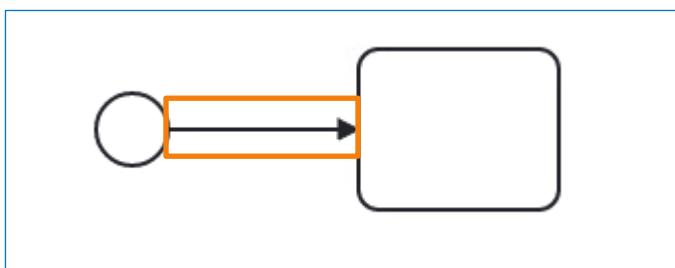
1. If you closed the Modeler open it again (follow the steps from the previous task to reopen it). You should see a blank process that only includes a start event (circle icon).



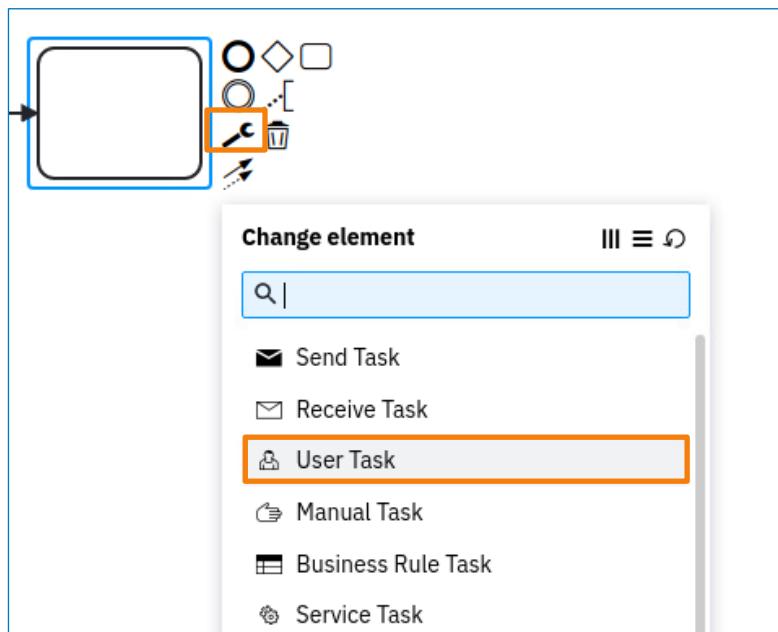
2. Add the first task of the process by dragging and dropping the **Create task** icon.



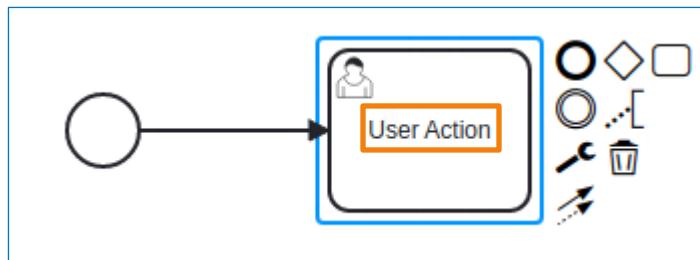
3. Connect the Start Event and the Task that you just added by clicking the **Start Event** and choosing the **arrows** icon from the side menu.



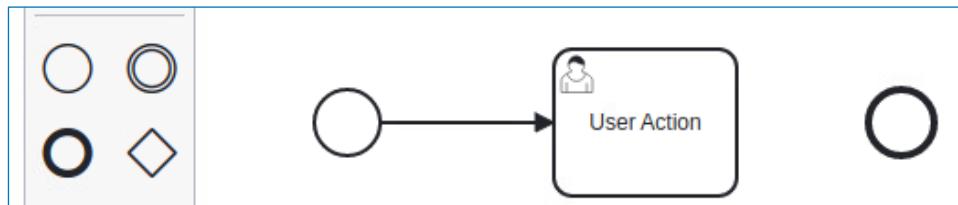
4. Different types of BPMN tasks can be used in Camunda. The most commonly used are User Tasks which require some kind of user action and Service Tasks which include some kind of logic and are executed automatically. In this example, set the type of task to **User Task**. Select the Create task icon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



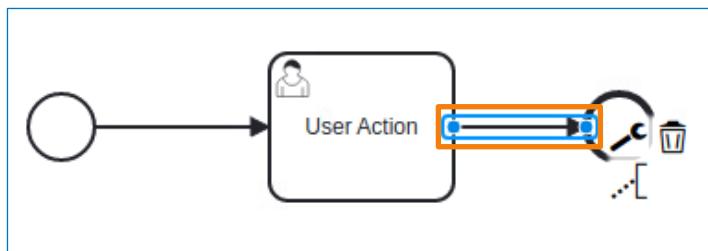
5. The task should be now marked as a user task (user symbol in the top-left corner). Name the task **User Action** by double-clicking the task and entering the name.



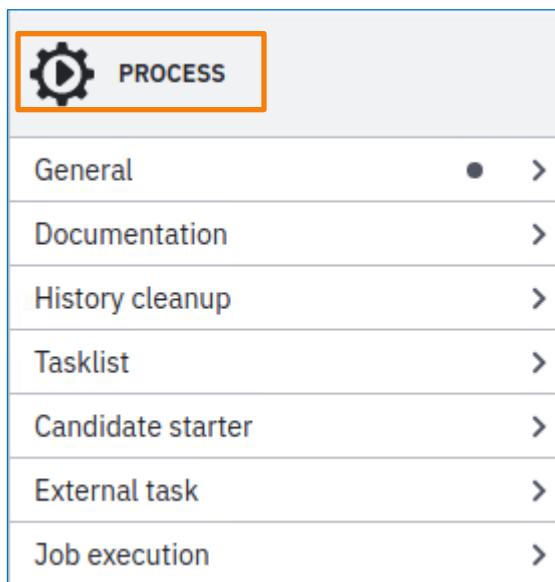
6. Since this is a simple task meant to demonstrate the development process, you can now add the **End Event** by dragging and dropping the **EndEvent** symbol (bold circle).



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.

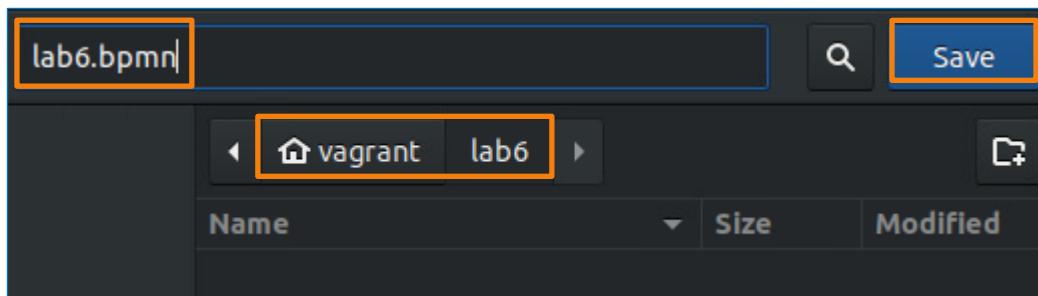


9. Expand the **General** section of properties and enter **Lab6** as the **name** and **ID** of the process.

The image shows the "General" properties dialog box with the following fields:

- Name:** Lab6 (highlighted with an orange border)
- ID:** Lab6 (highlighted with a blue border)
- Version tag:** (empty input field)
- Executable:**

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**.



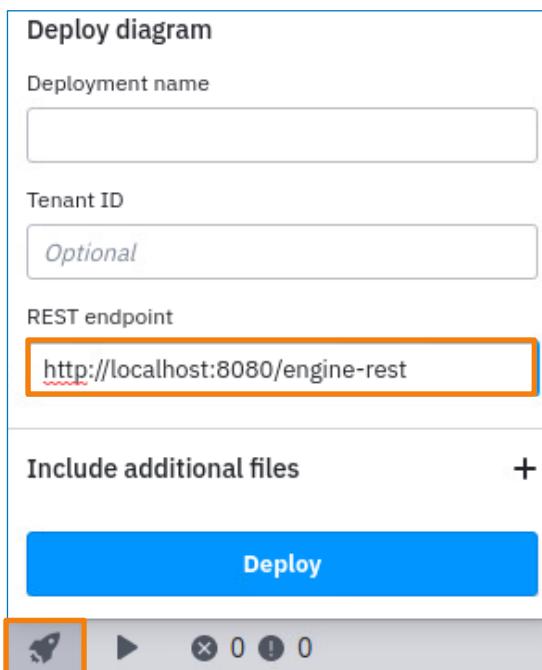
11. Before deploying the process to Camunda, explore what the .bpmn file that you created in the graphical editor looks like. Open a new terminal window and navigate to **lab6/** directory.

```
[~]$ [~]$ 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.



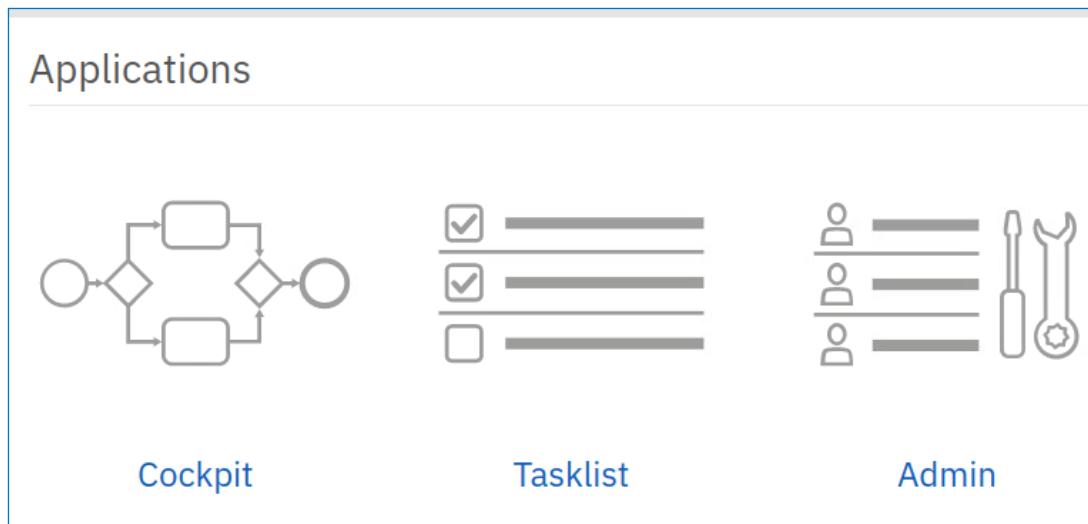
14. Name the deployment **Lab6** and click deploy. You should see a **Process definition deployed** pop-up message.

Process definition deployed

Process definition ID: Lab6:1:4928bd  
5a-9739-11ed-abef-0242ac110002

[Open in Camunda Cockpit](#)

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**.

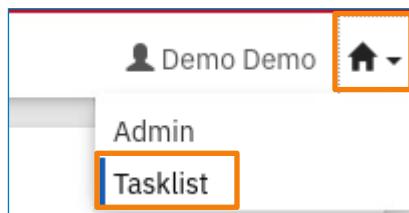


16. First open the cockpit application and click **Processes**. The list of deployed processes should now include your **Lab6** process.

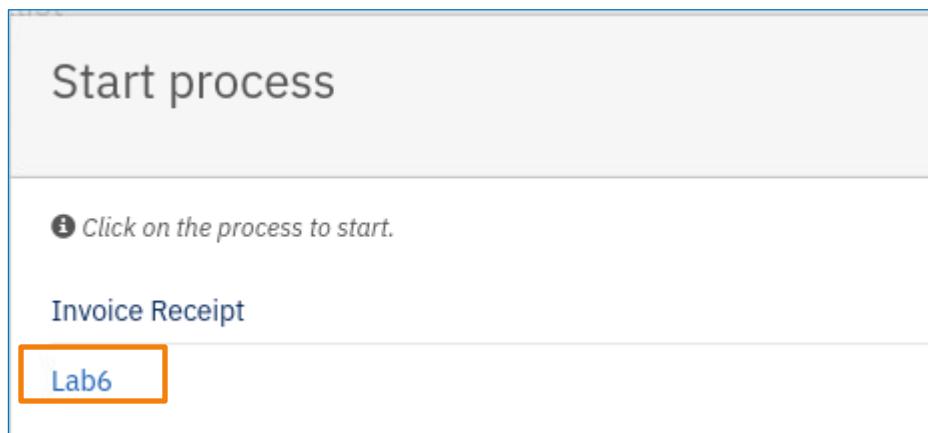
**NOTE:** If you currently have the Tasklist application open click on the house-shaped button in the top right corner to find the Cockpit.

3 process definitions deployed			
State	Incidents	Running Instances	Name
✓	0	6	Invoice Receipt
✓	0	0	Lab6
✓	0	2	Review Invoice

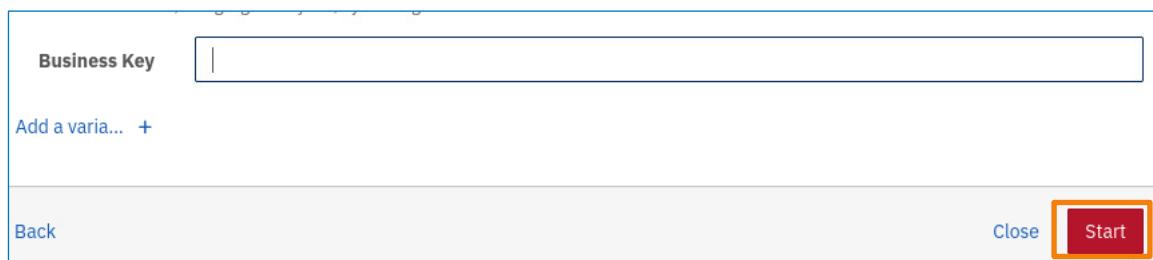
17. This process is not started yet. To do that, navigate to the **Tasklist** application.



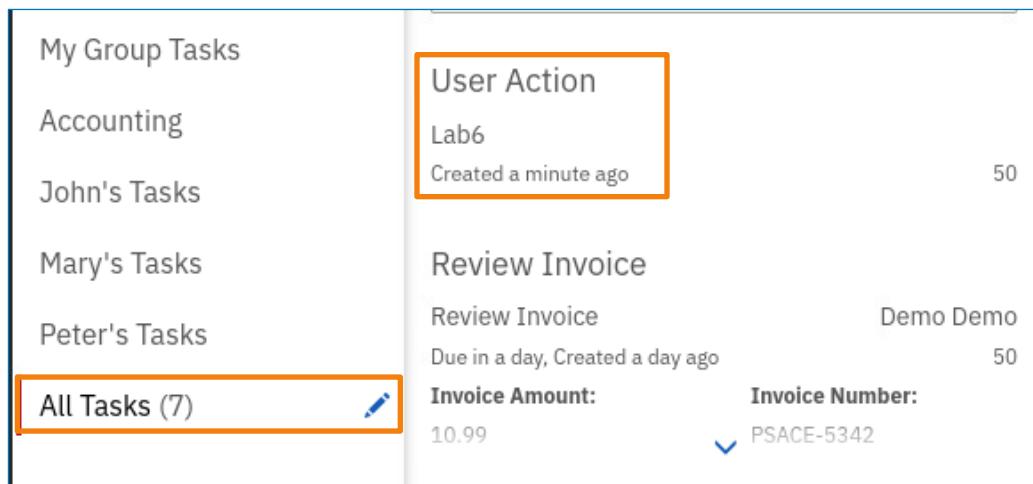
18. Click the **Start process** button from the top menu bar and select **Lab6** process.



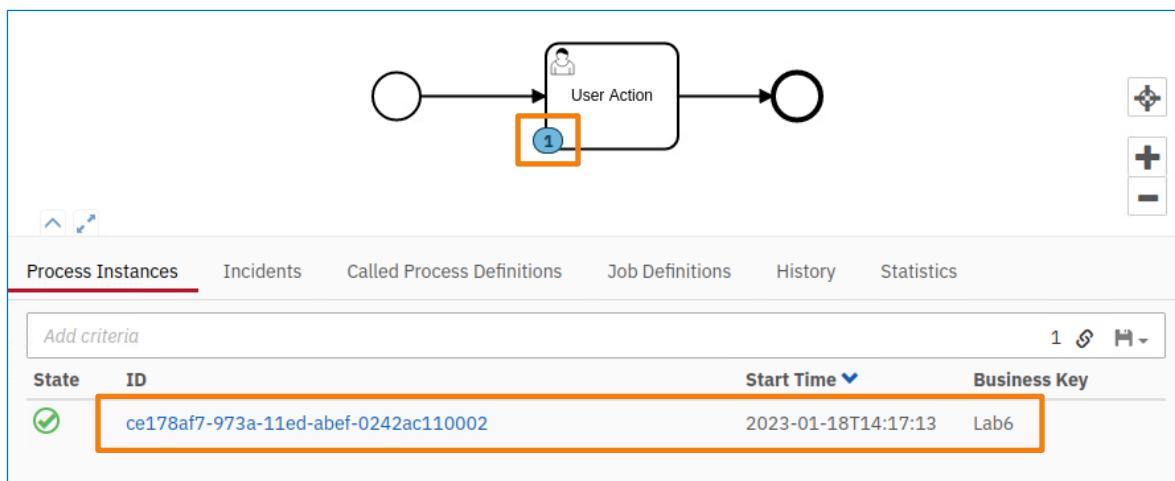
19. The Business key is not required so you can click the **Start** button.



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

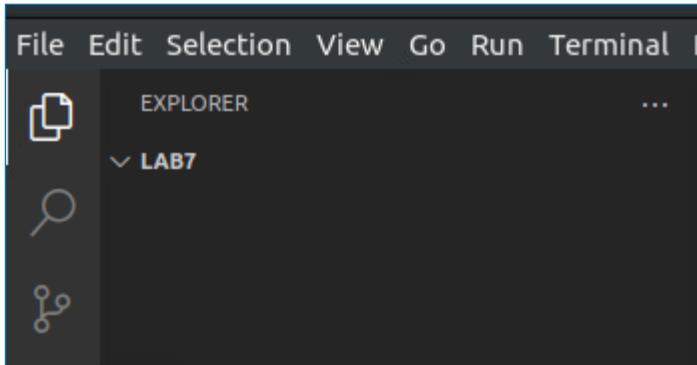
In this task, you prepare the working environment and create a Java project that will be used in a custom Camunda workflow.

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.

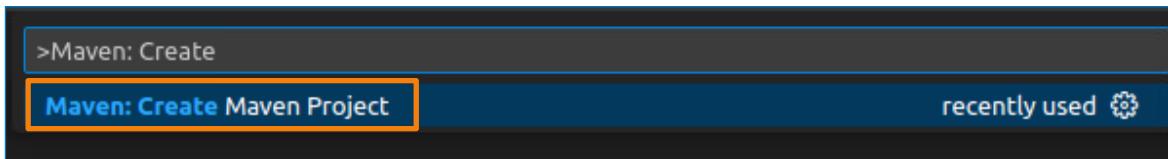
Create a new **lab7** directory and open it with the **code .** command.

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

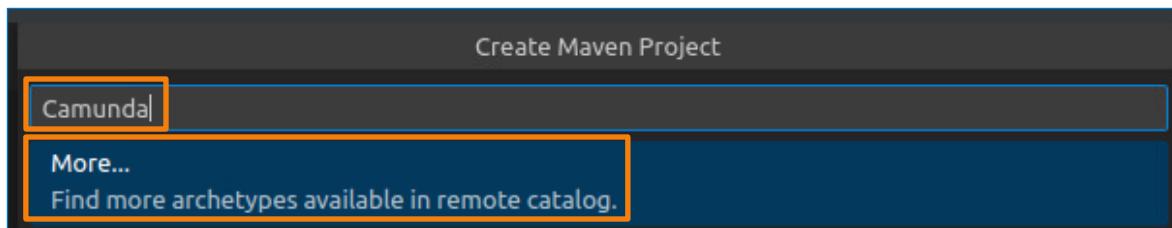
2. The empty lab7 directory should open in Visual Studio Code.



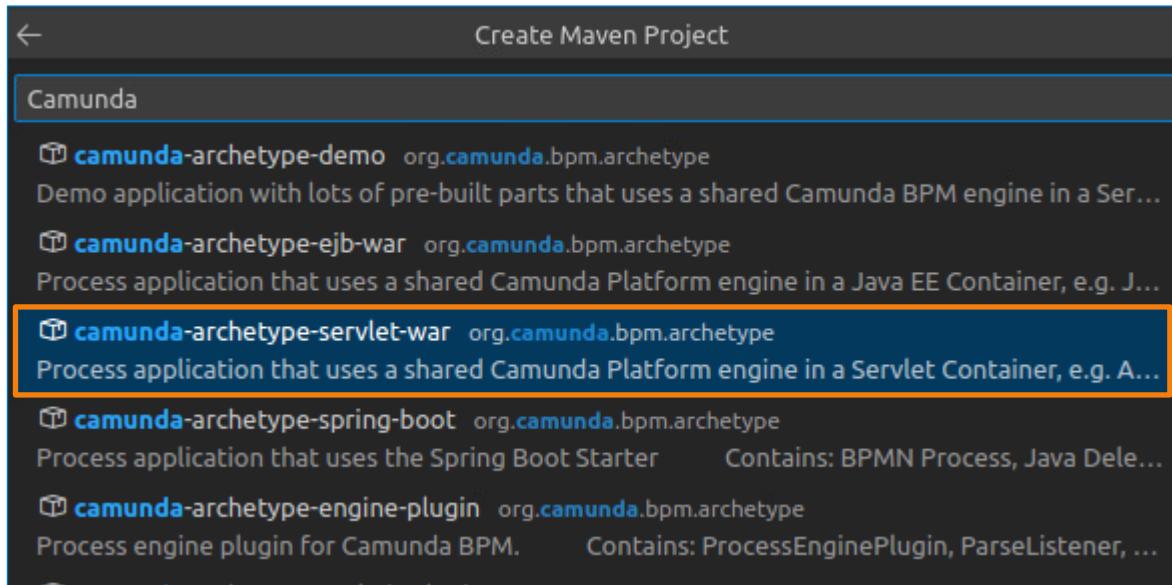
3. Create a new Maven project by pressing **Ctrl + Shift + P** to open the Command Palette and entering **Maven: Create Maven Project** in the search bar. Click the **Create Maven Project** button.



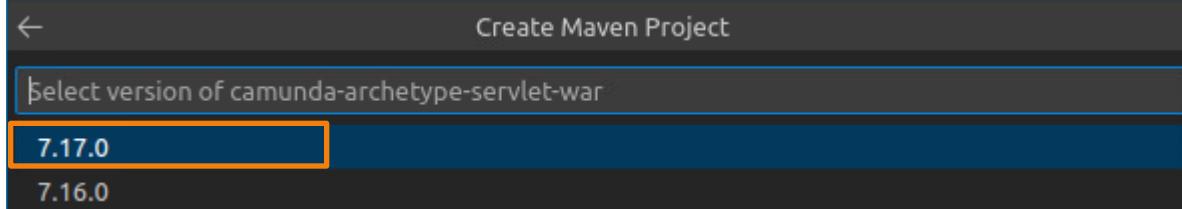
- To find the correct Archetype, search for **Camunda** in the **Select an archetype** menu and click the **More...** button.



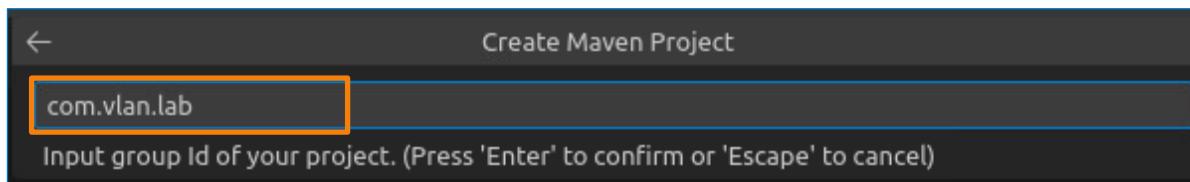
- Choose the **camunda-archetype-servlet-war** from the dropdown menu. This archetype works well for most of the basic Camunda projects.



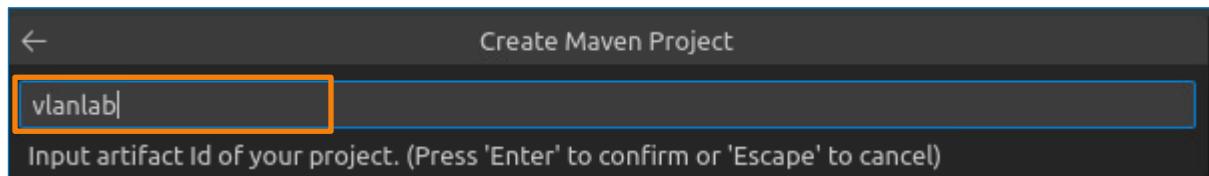
- From the dropdown menu choose version **7.17.0**.



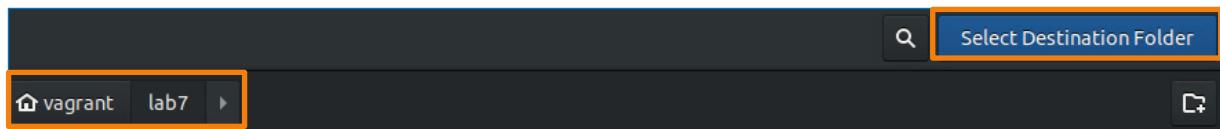
- For the **group Id** of the project, enter **com.vlan.lab** and press **Enter**.



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



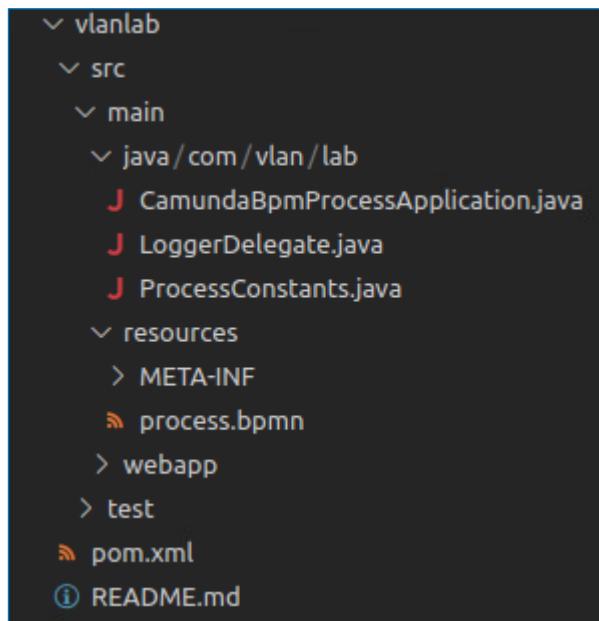
9. Choose **/home/vagrant/lab7** as the project location and click the **Select Destination Folder** button.



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

```
Define value for property 'version' 1.0-SNAPSHOT: :  
[INFO] Using property: package = com.vlan.lab  
[INFO] Using property: archetype-artifactId = camunda-archetype-servlet-war  
[INFO] Using property: archetype-groupId = org.camunda.bpm.archetype  
[INFO] Using property: archetype-version = 7.17.0  
[INFO] Using property: camunda-version = 7.17.0  
[INFO] Using property: project-description = A Process Application for [Camunda Platform](http://docs.camunda.org).  
[INFO] Using property: project-name = Camunda Platform Process Application  
Confirm properties configuration:
```

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

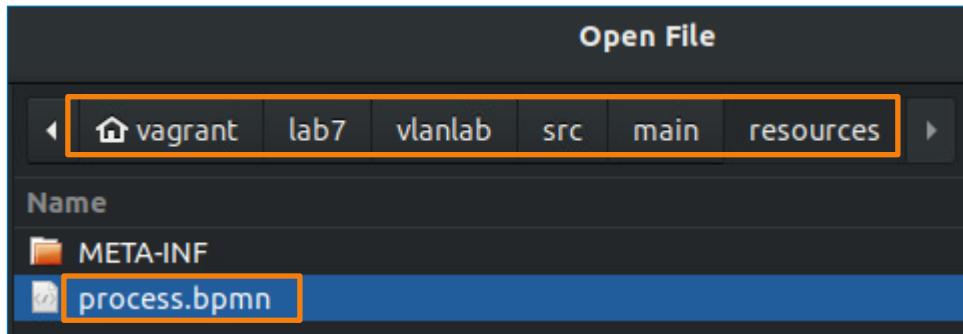
In this task, you create a custom BPMN workflow that reserves a VLAN ID if possible and gives feedback about the reservation status.

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.

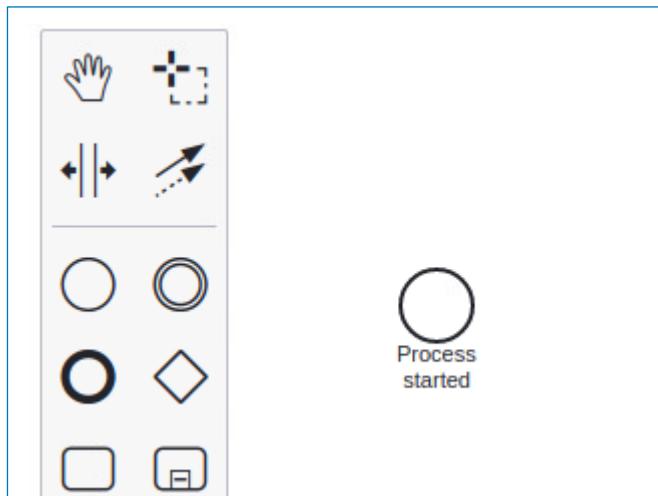
```
[~]$ cd camunda-modeler-5.6.0-linux-x64/  
[~/camunda-modeler-5.6.0-linux-x64]$ ./camunda-modeler
```

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.

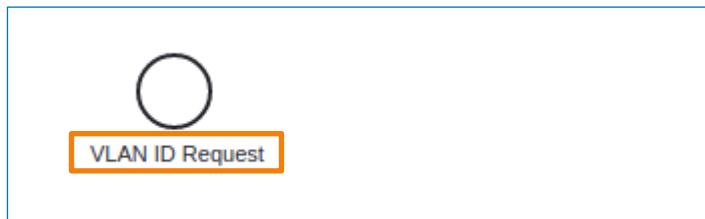
**NOTE:** If you did not close the .bpmn file from the previous lab, the modeler will open that file. You can close it by clicking the **x** button next to the file.



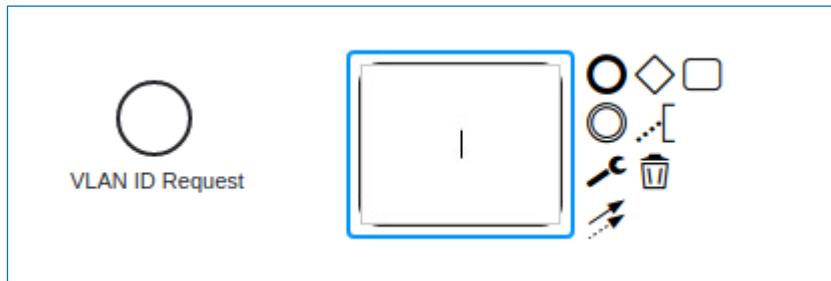
3. A process with only a start event and an end event should open. Delete the end event for now. Your process should now only include the start event.



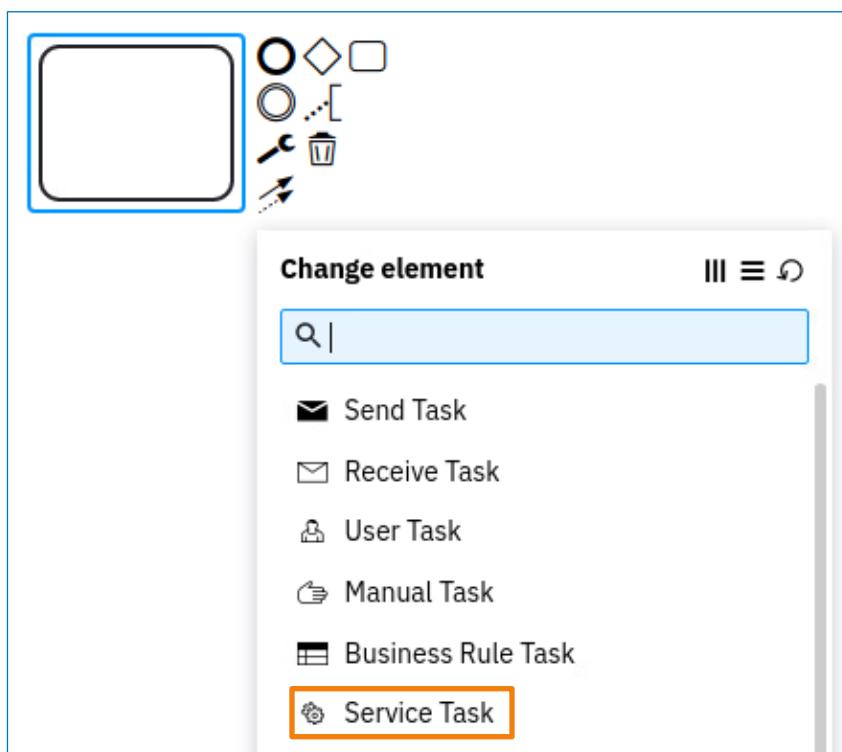
4. First name the start event **VLAN ID Request** as this process should be called when a request for a new VLAN ID is triggered. To change the name, double-click the start event icon and enter the name or change it under the general section of the properties menu.



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

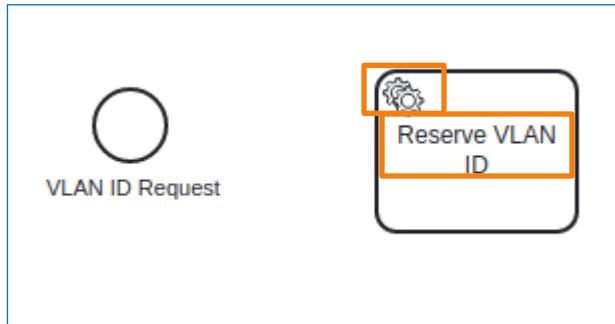


6. Set the type of task to **Service Task** by clicking the **wrench** button and choosing the type from the dropdown menu.

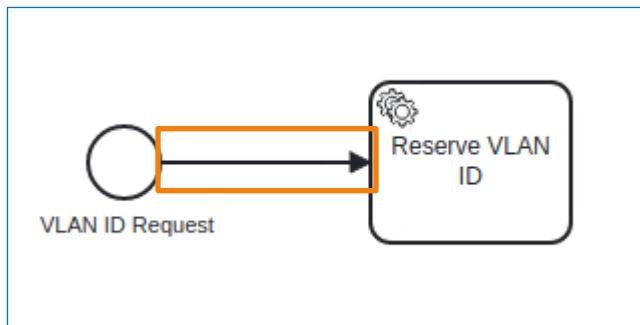


7. Name the task **Reserve VLAN ID** by double-clicking it and entering the name.

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

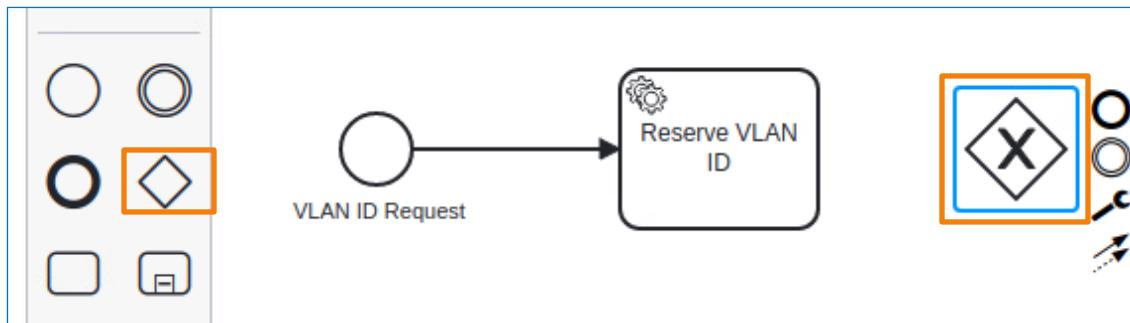


8. Connect the Start Event and the Task that you just added by clicking the **Start Event** and choosing the **arrows** icon from the side menu.

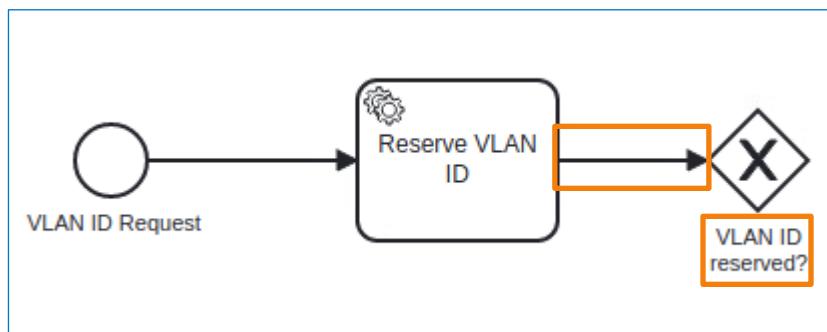


9. The goal of the **Reserve VLAN ID** task is to reserve one of the available IDs. Now you want to add the tasks that will give feedback about reservation status and notify the user whether the reservation was successful.

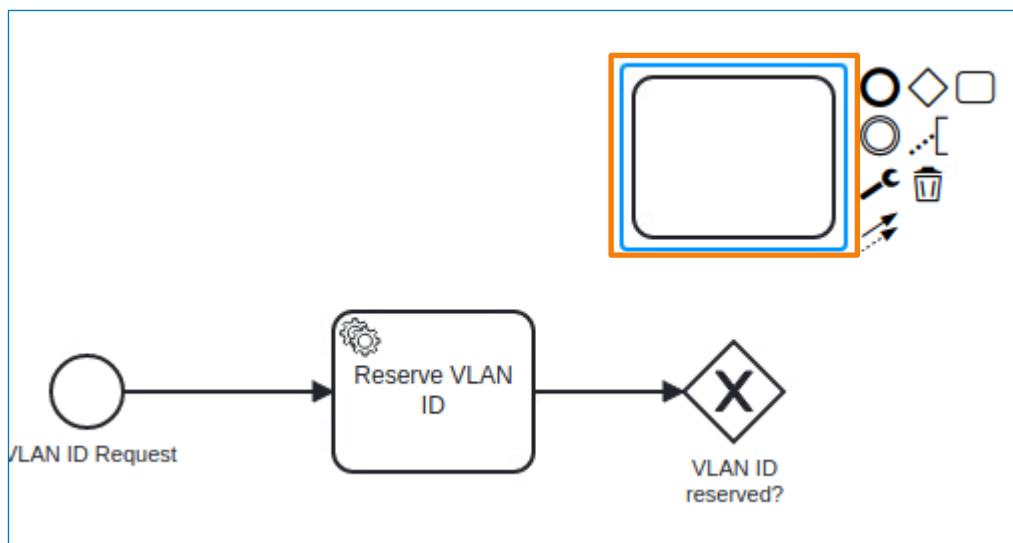
First, add an XOR gateway to the process by dragging and dropping the **Create Gateway** icon.



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



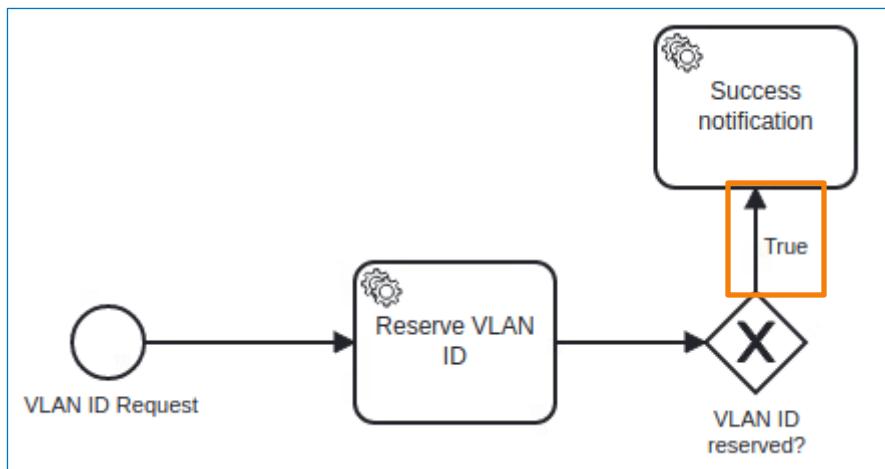
11. Now you will add two different paths to the workflow. First, add a task that will notify the user if the reservation was successful. Add another task to the process.



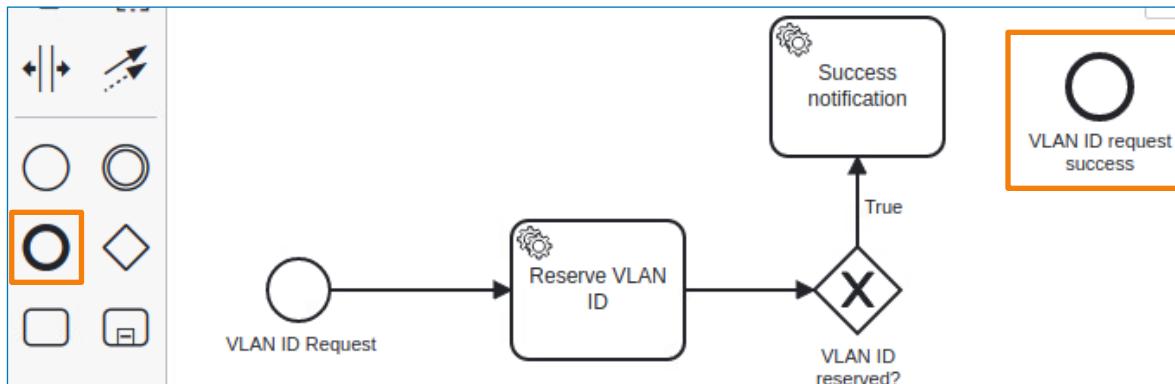
12. Change the type of the task to **service task** and name it **Success notification**. This task will notify the user when the VLAN ID is reserved successfully.



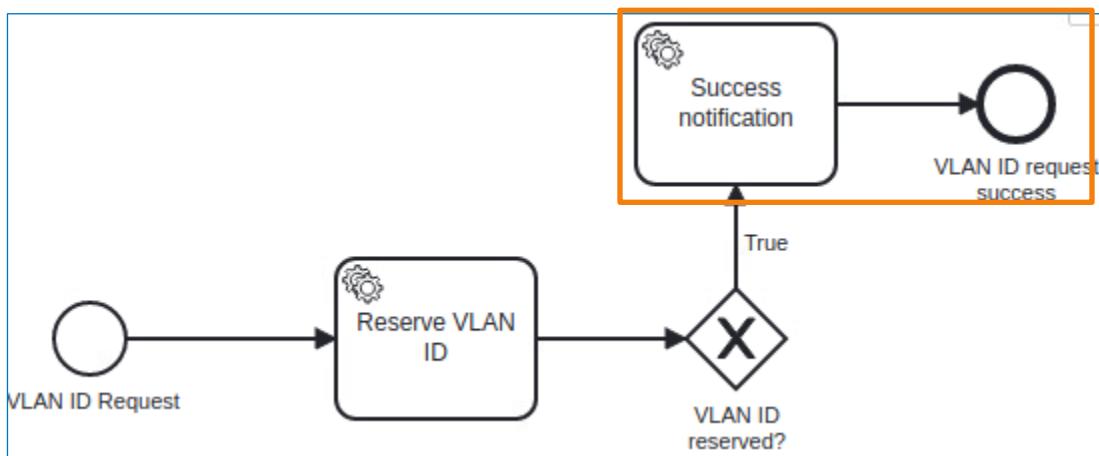
13. Connect the Gateway with the task that you created in the previous step and name the connection **True**, since this part of the workflow is only triggered when VLAN ID is reserved successfully.



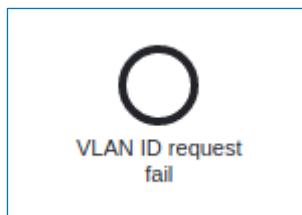
14. Add the **endEvent** to the workflow by dragging and dropping the **bold circle icon** to the screen. Name the event **VLAN ID request success**.



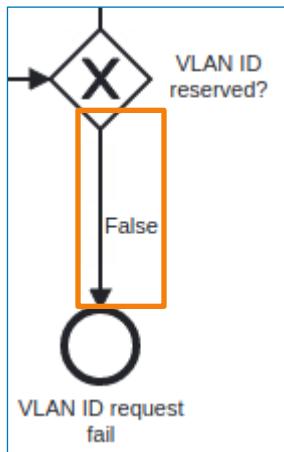
15. Connect the **Success notification** task with the **end event** as the process must end when the ID is reserved successfully.



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**.

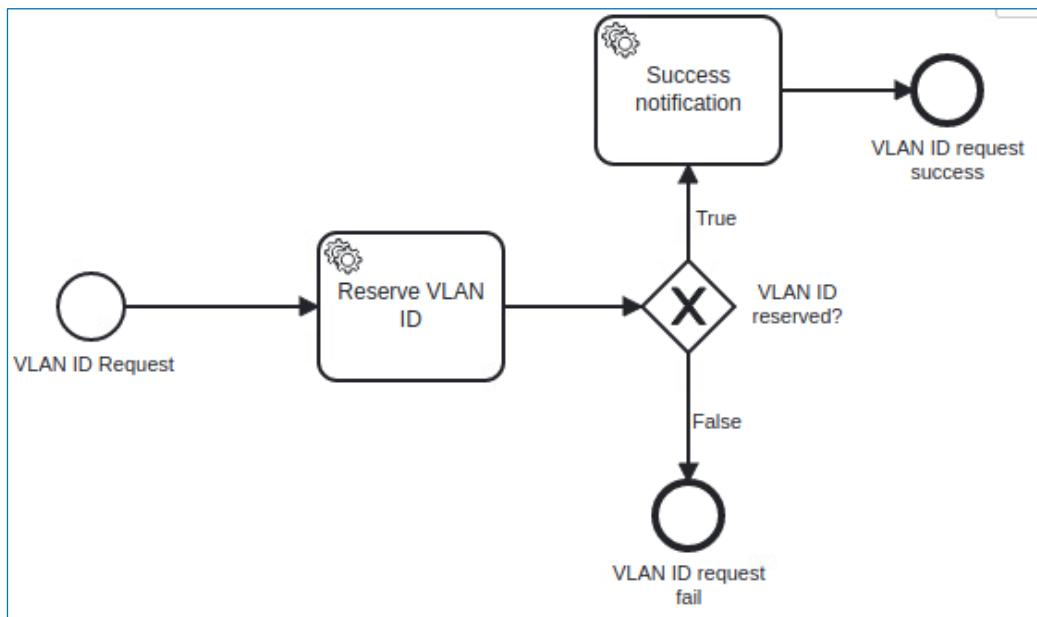


17. Add another connection to the gateway by connecting it to the **VLAN ID request fail** end event. Name the connection **False** as this path will be selected if the VLAN ID is not reserved.

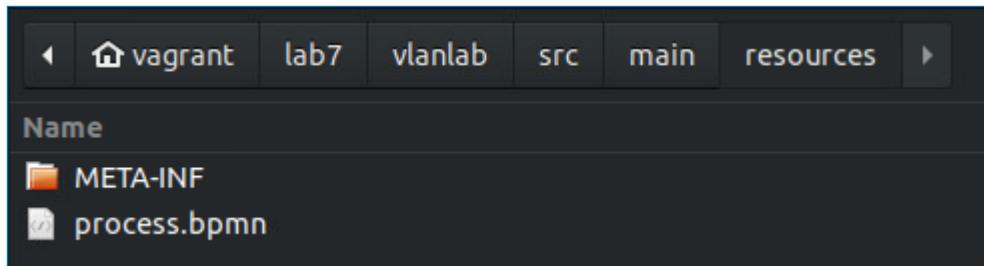


18. Your workflow should now look like the following figure:

**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.



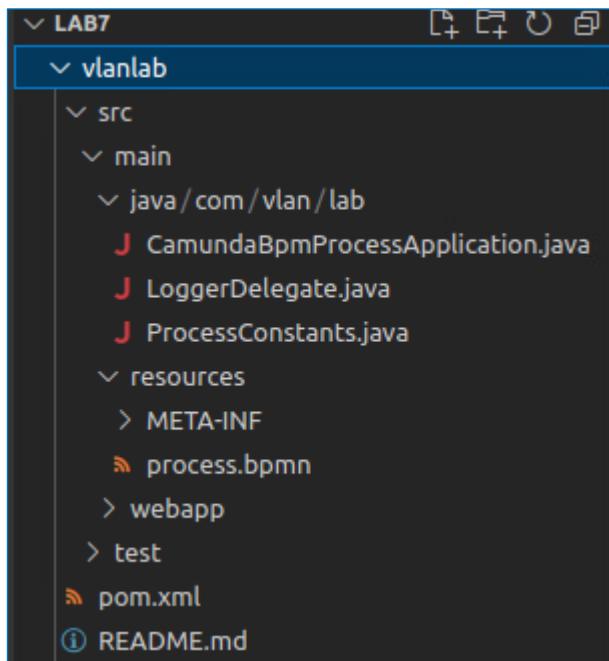
## Task 3: Implementing the Process Logic

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.

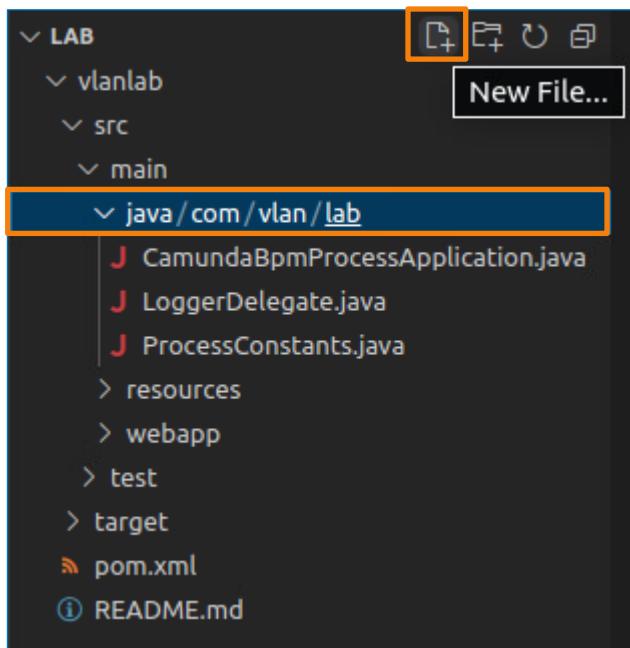
```
[~]$ [~]$ cd lab7/  
[~/lab7]$ code .
```

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

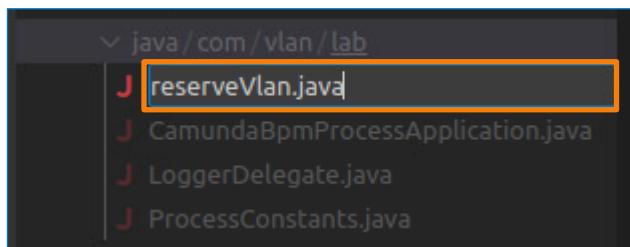


3. Since you used the Camunda archetype to create the project, most of the needed files are already prepared for you, and you only have to add the custom logic that you want to use in your process. In this example, you will add a new Java class that will reserve a VLAN ID if it is available.

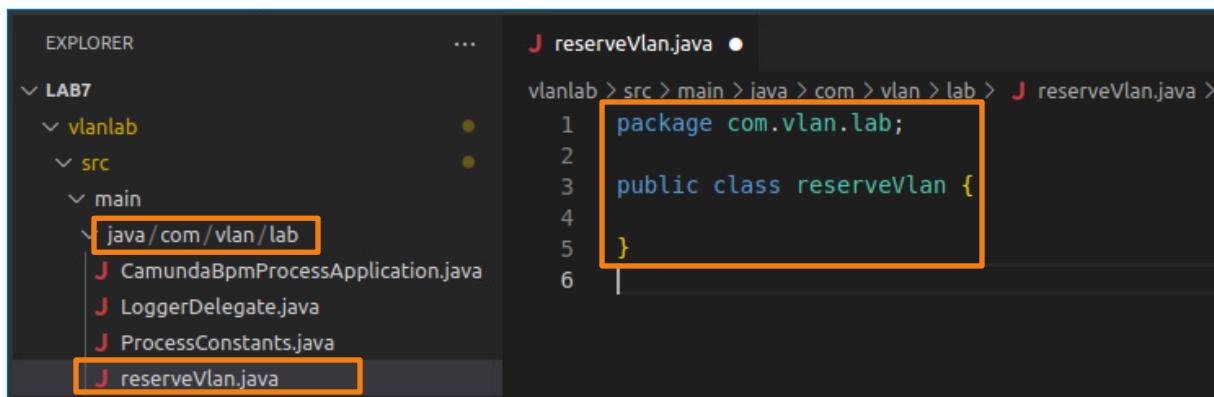
Select the **java/com/vlan/lab** directory and click the **New File...** button.



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 {

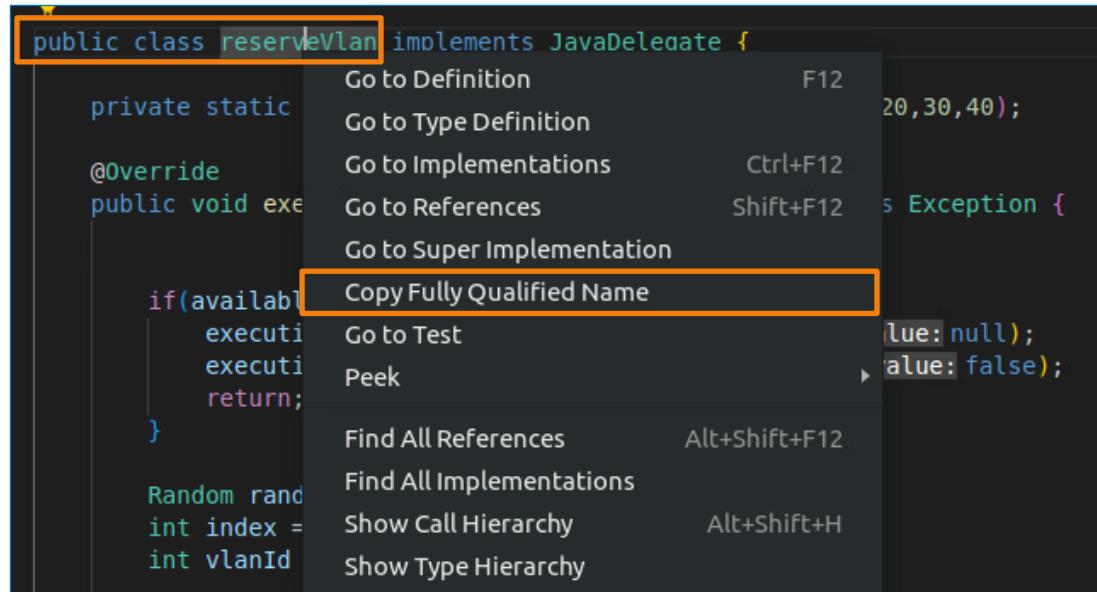
        if(availableIds.isEmpty()){
            execution.setVariable("vlanId", null);
            execution.setVariable("vlanSet", false);
            return;
        }

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

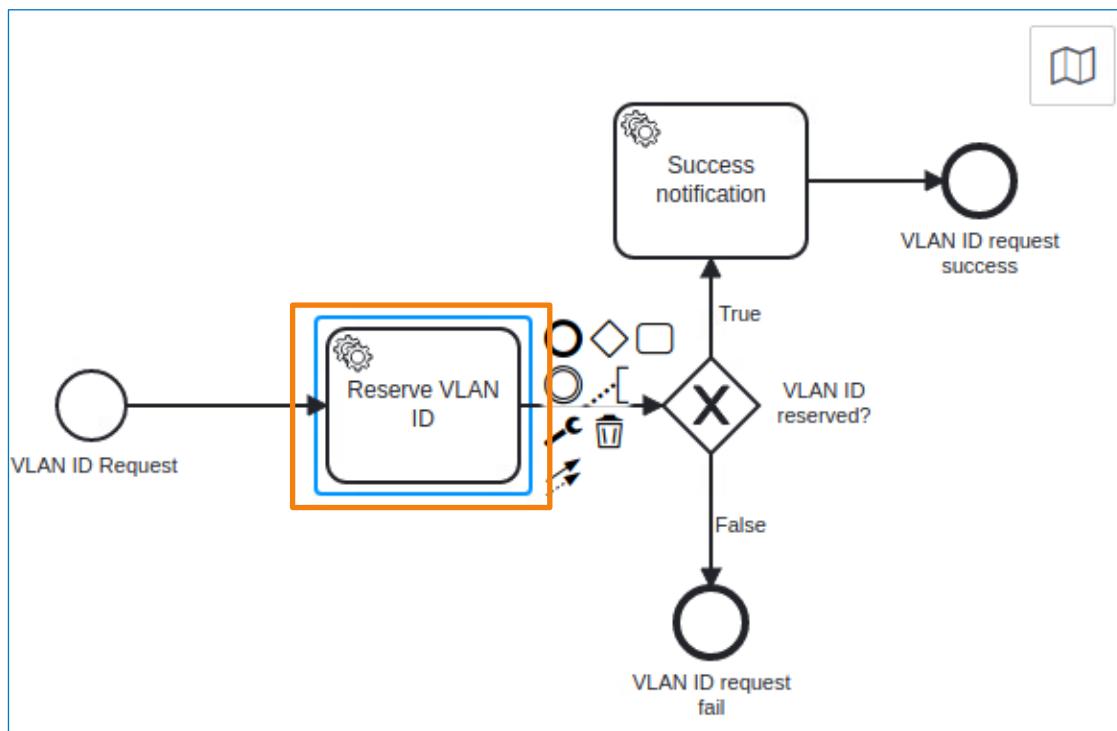
```
        execution.setVariable("name", "student");
        execution.setVariable("vlanId", vlanId);
        execution.setVariable("vlanSet", true);
    }
}
```

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.

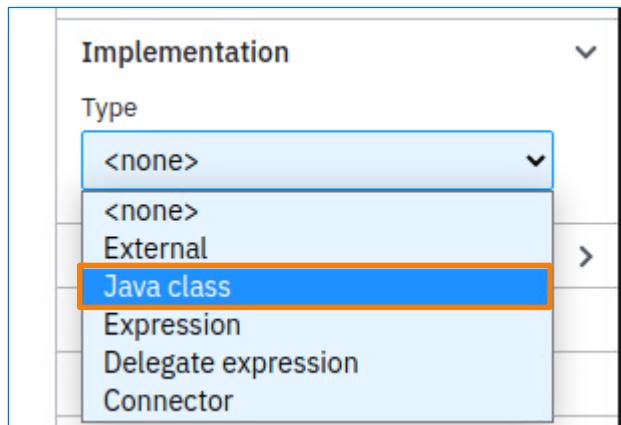
Save the file and right-click the **reserveVlan** class and copy the fully qualified name.



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

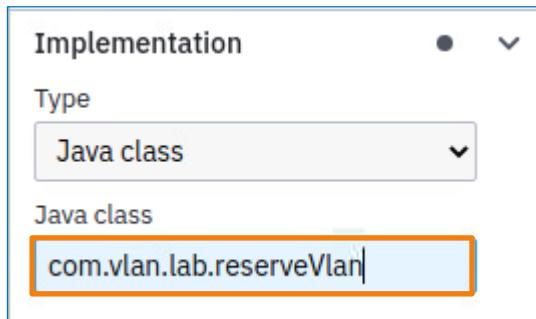


15. Expand the **Implementation** section of the properties menu and choose **Java class** as the implementation type.



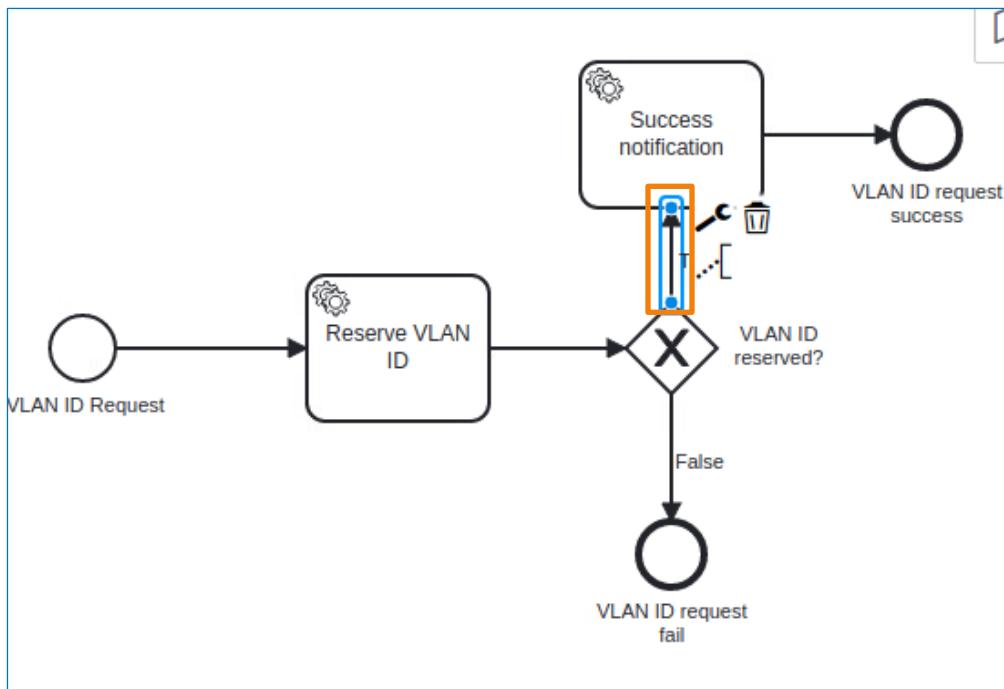
16. Paste the qualified name of your class to the **Java class** field.

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

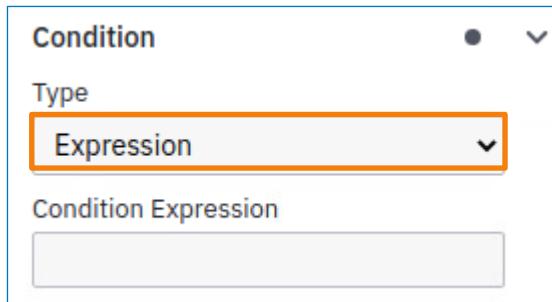


17. Now you covered the first task of your BPMN process. The next step that you must configure is the gateway that checks if the ID was reserved or not. Since you know that the first task will generate a variable named **vlanSet** with the value set to **true** or **false**, the gateway can simply check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

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



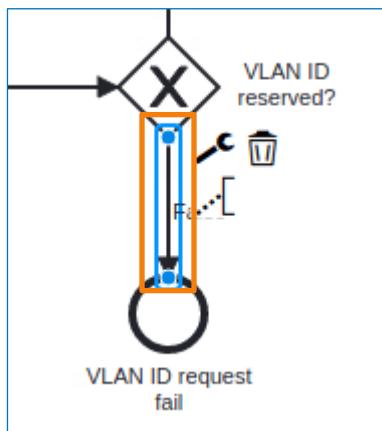
18. Expand the **Condition** section of the Properties menu. Here you can set under which condition the process will continue in this way. In this example, select **Expression** as the type of condition.



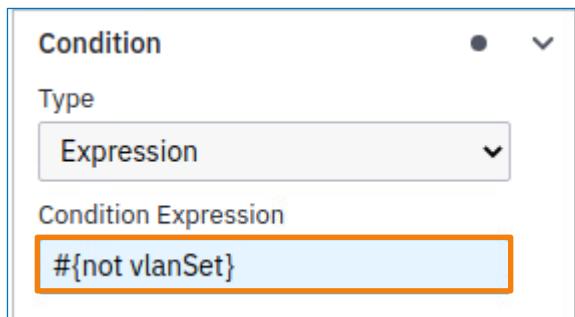
19. Enter **#{vlanSet}** in the **Condition Expression** field. This statement will check if the variable **vlanSet** is set to **true**.



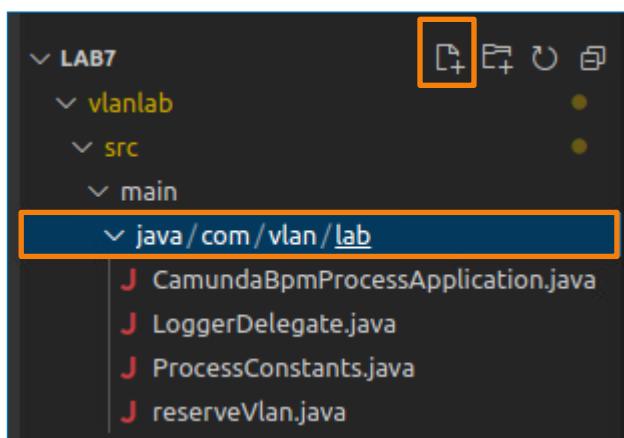
20. The same logic applies to the other possible path of the workflow. To define the other condition, click on the arrow named **False**.



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;
```

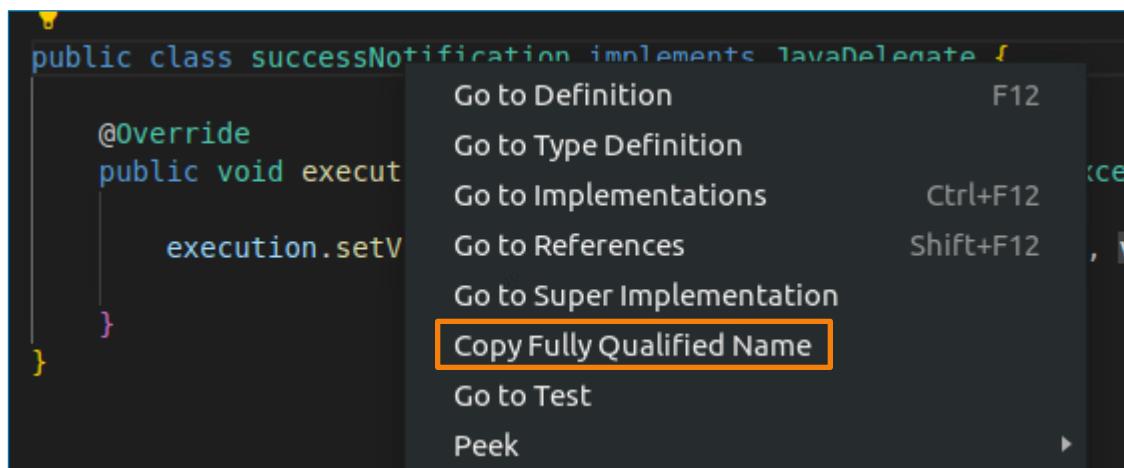
```
import org.camunda.bpm.engine.delegate.DelegateExecution;
import org.camunda.bpm.engine.delegate.JavaDelegate;
public class successNotification implements JavaDelegate {

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

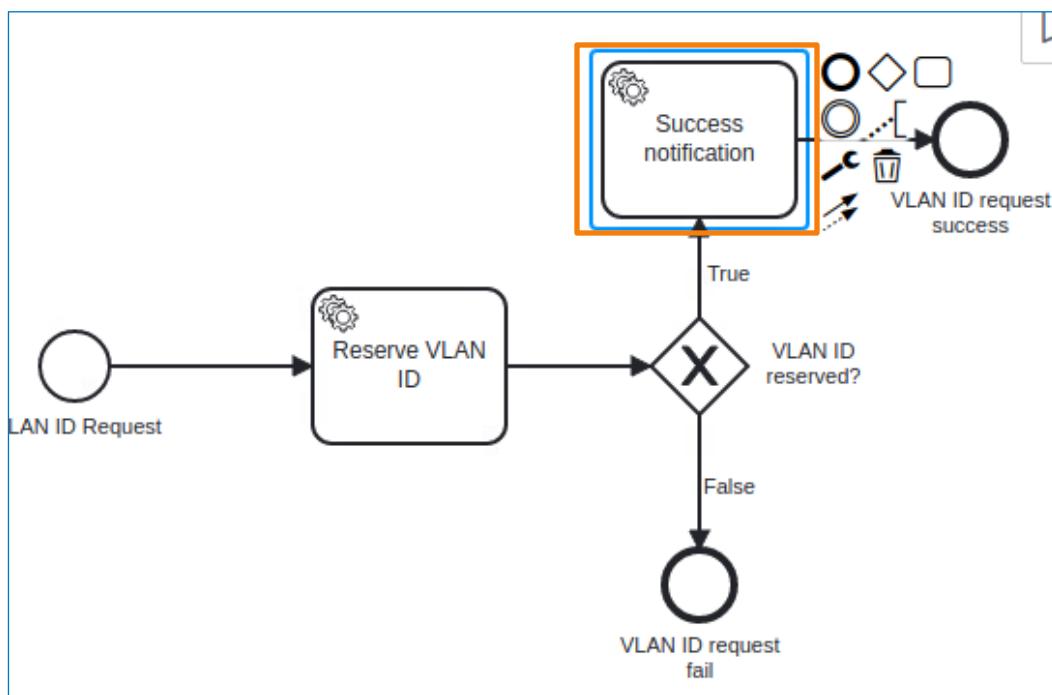
        execution.setVariable("reservationStatus", true);

    }
}
```

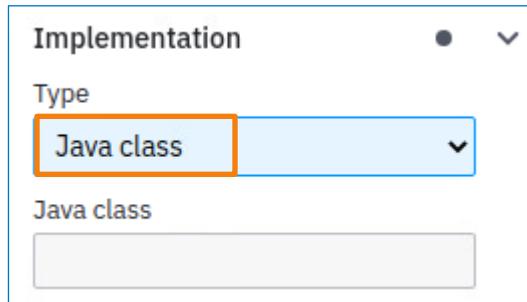
25. Save the file and right-click the class to copy the fully qualified name.



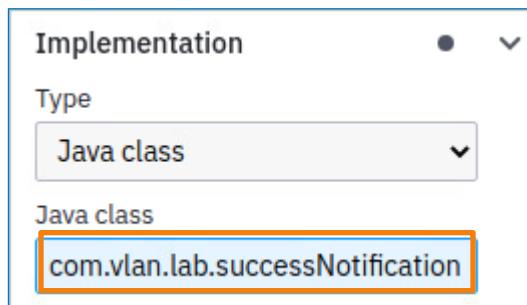
26. Navigate back to the modeler and click on the **Success notification** task.



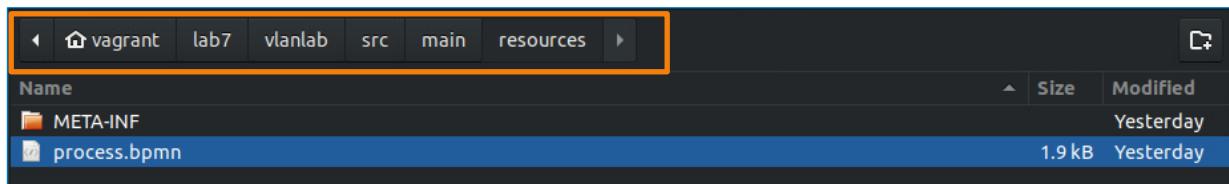
27. Open the implementation section of the properties menu and select **Java class** as the type.



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.

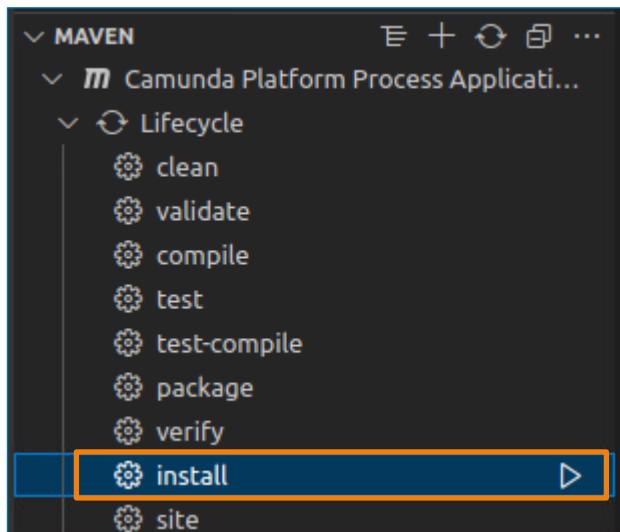


## Task 4: Building the Project and Deploying it to Camunda

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"`.

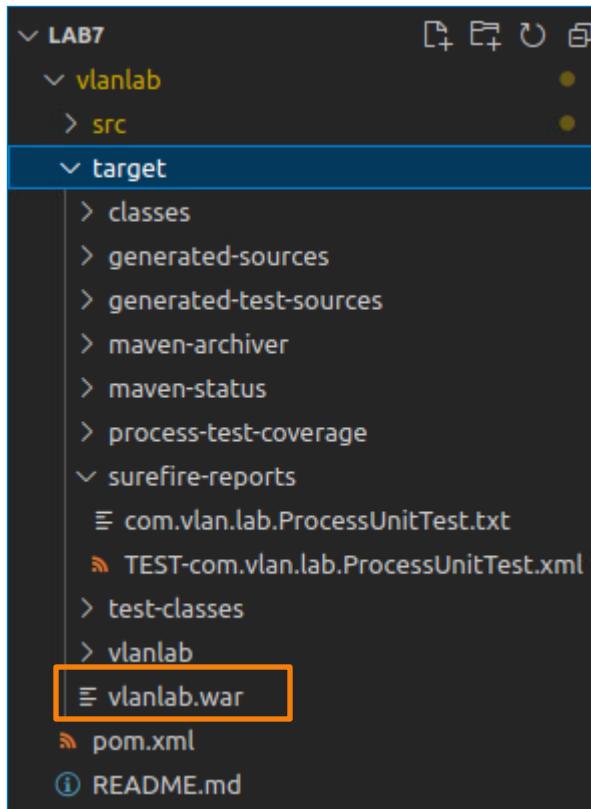


2. The project should be built successfully. If there are any errors, try to compare the code and the process shown in the lab guide with your code and BPMN process.

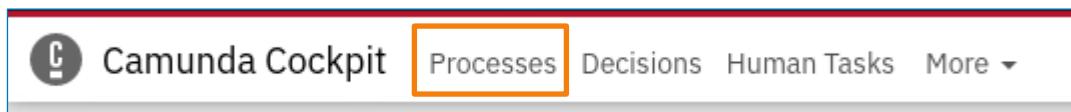
**NOTE:** If your process does not build successfully, you can also observe the logs in the `/target/surfice-reports` directory.

```
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time:  17.442 s
[INFO] Finished at: 2023-01-29T15:52:39Z
[INFO] -----  
vagrant@bp-devops-toolkit:~/Lab7$
```

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.

Running Instances	Name
6	Invoice Receipt
1	lab6
2	Review Invoice

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$
```

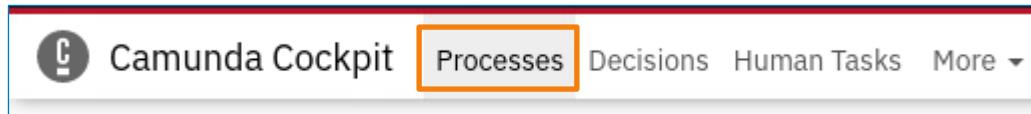
9. Navigate to the **webapps** directory of the container and list the contents of the directory.

```
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.

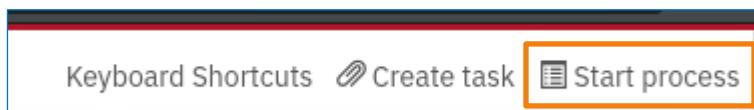
**NOTE:** If you left the Camunda Cockpit application opened you should refresh it to see the newly added process.



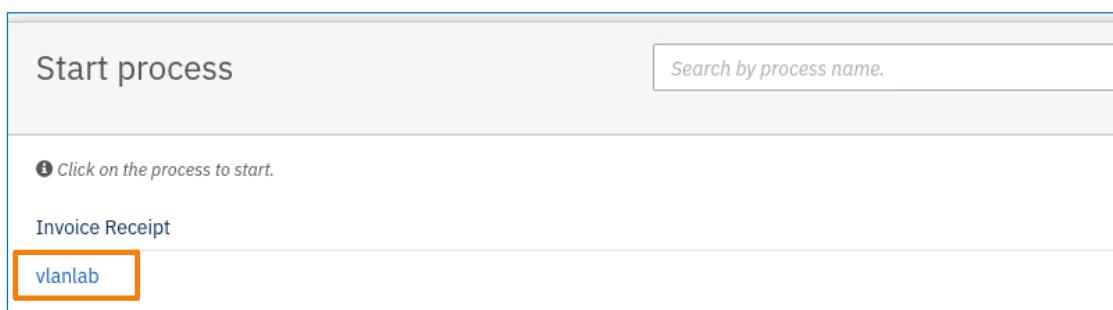
11. Make sure that the process is now available under the list of processes.

Running Instances	Name
6	Invoice Receipt
2	Review Invoice
0	vlanlab

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.



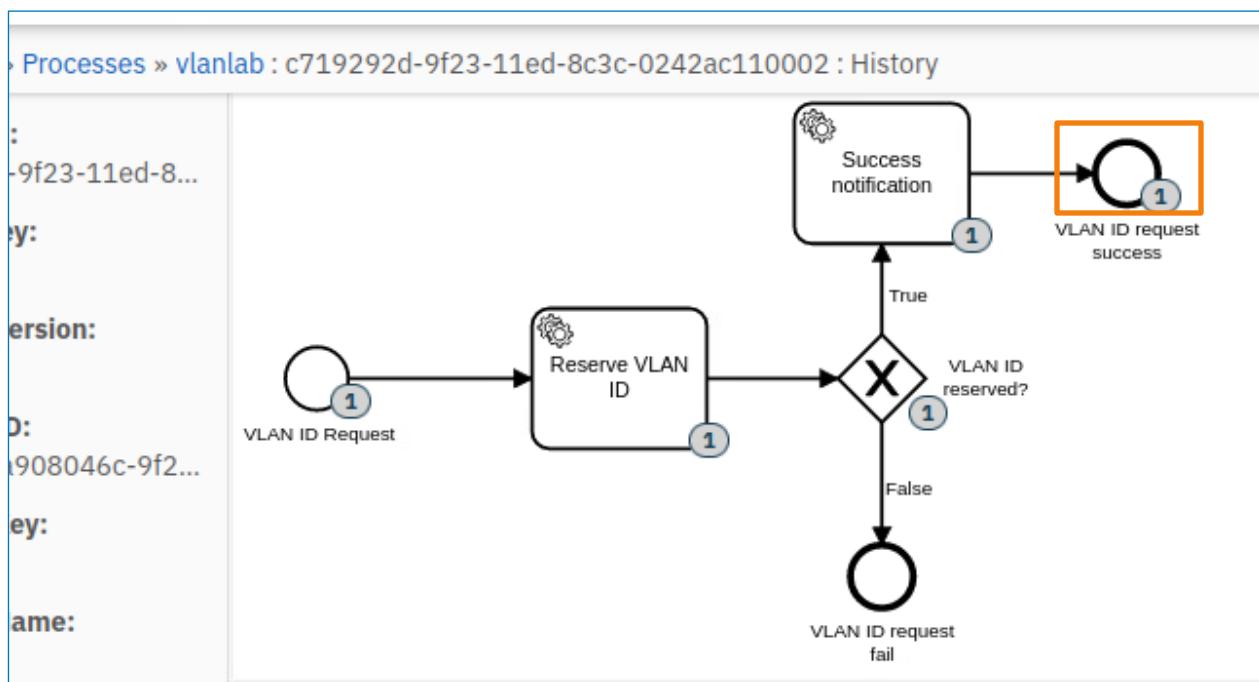
15. Select your **vlanlab** process from the list to find more information about the process.

Running Instances	Name
6	Invoice Receipt
2	Review Invoice
0	vlanlab

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

Process Instances	Incidents	Called Process Definitions	Job Definitions	History	Statistics
State —	Instance ID —	Start Time —			
COMPLETED	ef95b12c-a086-11ed-9864-0242ac110002			2023-01-30T10:14:52	

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).

Audit Log	Variables	
Name —	Type —	Value —
name	String	student
reservationStatus	Boolean	true
vlanId	Integer	40
vlanSet	Boolean	true

---

End of Lab