Master of Science in Internetworking

Capstone Project Report

On

SDN Controller GUI design for NETCONF

By **Najme Salehi**

Under the supervision of

Mr. Shahnawaz Mir

Winter 2023

Abstract

OpenDaylight (ODL) is an open-source software-defined networking (SDN) controller platform that provides a flexible and scalable network infrastructure. It enables network programmability and automation, allowing users to control and manage their networks through a centralized controller. ODL is widely used in various industries, including telecommunications, cloud computing, and data centers.

This project aims to develop a new Graphical User Interface (GUI) for the OpenDaylight SDN controller that is user-friendly, functional and aligned with the objectives of our university laboratory. The purpose of the GUI is to provide a visual representation of the network, automate many tasks, and enhance the overall efficiency and reliability of the network.

Due to the existing GUI's limitations, we have decided to design a custom GUI for ODL. This project report provides a comprehensive overview of the design and implementation of the OpenDaylight GUI, including the requirements, design considerations, technical challenges, testing, validation, and future directions of the project.

The new GUI is expected to improve the user experience, simplifying the platform's usage for users to complete their projects.

Table of Contents

1 Introduction

Software-Defined Networking (SDN) is a networking approach in which software-based controllers or application programming interfaces (APIs) are employed to interact with the underlying hardware infrastructure, enabling the control of network traffic. The centralization of network control and intelligence in SDN enables organizations to quickly manage and reconfigure network resource usage using automated provisioning. This makes SDN an important tool in modern networks, as it provides a more flexible and efficient way to manage network resources and functions, resulting in improved network performance and security. [1]

SDN separates the control plane from the data plane, centralizing network control in a single component called a Controller. This enables network administrators to define and manage network policies and configurations more efficiently, reducing the need for manual configuration of individual network devices. In traditional networks, network traffic is controlled by dedicated hardware devices such as switches and routers, which are configured and managed separately. This can be time-consuming and prone to human error, leading to network inefficiencies and security vulnerabilities. In contrast, SDN enables network administrators to program and manage the entire network through a single, centralized platform, resulting in more efficient and automated network operations. [1]

Another key advantage of SDN is its ability to quickly adapt to changing network requirements. In modern networks, network traffic patterns and resource usage can change quickly, requiring network administrators to respond quickly to reconfigure network resources. With SDN, network administrators can quickly and easily reconfigure network resource usage using automated provisioning, enabling the network to respond quickly to changing requirements. [2]

SDN also offers improved network security, as the centralization of network control in the Controller enables network administrators to monitor as well as control network traffic more. This can help prevent security breaches and other network security incidents and enable organizations to respond more effectively to security incidents when they occur. The ability of SDN to provide a more secure and efficient way to manage network traffic makes it an important tool in modern networks. [2]

The growing popularity of Software-Defined Networking (SDN) is due to its ability to provide a more agile and secure approach to managing network functions and resources. With its ability to centralize network control, improve network efficiency, and adapt to changing network requirements, SDN is becoming increasingly important in modern networks.

Adopting SDN enables companies to bring the benefits of cloud technology to their network deployment and management. By using network virtualization, they can utilize new tools and services like Software-as-a-Service (SaaS), Infrastructure-as-a-Service (IaaS), and other cloud computing options and integrate them with their software-defined network through APIs.

SDN also offers increased visibility and flexibility. In traditional network environments, switches or routers have limited information about the status of adjacent devices. With SDN, this information is centralized, giving companies the ability to monitor and control their entire

network and devices. Additionally, SDN enables the creation of virtual networks within a physical network or the connection of multiple physical networks into a single virtual network, providing great versatility. The companies adopt SDN to effectively control and scale their network traffic as needed, leveraging its benefits of efficiency, security, and versatility. [2]

2 SDN Operational Mechanism

It is useful to identify the fundamental components that make up the network ecosystem to gain a better understanding of how SDN operates. These components may be situated in the same physical location or distributed across multiple locations and include:

- **Applications**

Applications are responsible for providing information about the network or requesting resources from the network. Applications communicate with the SDN controller to indicate the intended behavior of the network, and the Controller then sends instructions to the networking devices on how to behave. Applications can request network services, and the SDN controller can act accordingly to make sure the requested service is provided.

For example, an application may request the allocation of a specific amount of bandwidth to meet the demands of a critical service. The SDN controller can then communicate with the networking devices to allocate the requested bandwidth. [3]

- **SDN Controllers**

SDN controller's primary role is to handle communication between applications and network devices in the data plane. The controllers act as the brain of the SDN network, and they are responsible for determining the destination of data packets based on information provided by the applications.

SDN controllers are also responsible for load balancing within the network, ensuring that traffic is distributed evenly across the available network resources. They receive instructions from the applications and translate them into low-level network configurations that are sent to the data plane. This allows the controllers to control and manage network traffic efficiently, resulting in better performance and reduced latency. [3]

- **Networking Devices**

The networking devices in an SDN ecosystem receive instructions from the SDN controllers, which determine how data packets should be routed within the network. These devices may include routers, switches, and other types of network hardware. Unlike traditional networks, where these devices also handle the control plane functions, in an SDN architecture, they are stripped down to only their data plane functions, meaning they are responsible only for forwarding data. By separating the control and data planes, it becomes possible to implement a centralized controller that is responsible for managing and directing the behavior of these network devices. This

approach makes it easier to manage complex networks and implement changes across the entire network in a more streamlined way. [3]

By combining these components, organizations can manage networks more easily and in a centralized manner. SDN involves a separation of the routing and packet forwarding functions, also known as the control plane, from the underlying infrastructure or data plane. SDN then deploys controllers, which are considered the brain of the SDN network, above the network hardware, either in the cloud or on-premises, allowing for policy-based management and automation of network control.

In SDN, the switches are instructed by controllers on where to forward packets. In some cases, virtual switches embedded in software or hardware may replace physical switches, consolidating their functions into a single intelligent switch capable of verifying data packets and their virtual machine destinations to ensure smooth packet movement.

3 Types of SDN

Two main types of SDN architectures can be deployed, each with its advantages and limitations: centralized and distributed.

3.1 Centralized SDN Architecture

The centralized architecture has a single controller that acts as the brain of the network. It collects data from all the network devices and manages the network through a central point of control. This architecture provides a simple way to manage network configurations from a single point, making it easy to deploy, manage, and monitor network devices. The primary benefit of a centralized architecture is that it provides greater visibility and control over the network. Since the Controller is responsible for managing all the network devices, the Controller can analyze network traffic and make real-time decisions to optimize network performance.

The OpenFlow protocol has become the official protocol for making high-level routing decisions in a centralized SDN model. The good news is that most switch/router vendors have announced OpenFlow support, but the bad news is that it is not yet a complete standard and lacks specific features for managing devices and communicating network status. The lack of a mechanism for fully managing devices and controlling port/trunk interfaces and queuing is a concern for the centralized model. The capabilities of central control software are crucial for the success of the centralized SDN model. However, the way OpenFlow forwarding is supported in network devices is also a concern, as not all implementations take full advantage of fast-path technology, resulting in slower performance. The use of native OpenFlow switches may be a solution, but they are limited in number. [4]

3.2 Distributed SDN Architecture:

The distributed architecture uses multiple controllers that are distributed throughout the network. These controllers work together to manage the network and make decisions about traffic flow. Unlike the centralized architecture, the distributed architecture distributes the workload across multiple controllers, making it more scalable and fault-tolerant. Since the controllers are distributed throughout the network, they can make faster decisions about traffic flow, resulting in better network performance.

In the distributed SDN model, the focus is on the control software, which is responsible for managing the network. The goal is to expose the network's traffic and connectivity management capabilities to a higher software layer, which would then frame these capabilities as "virtual network services" for the cloud or applications. Northbound APIs or interfaces allow software, including cloud stack software, to control network services.

The distributed SDN model has a different set of requirements for connecting technology than the centralized model. The distributed model does not centralize routing decisions like its competing model, so it does not need OpenFlow. However, it needs a practical way of gathering status and performance information from the network, which means gathering data across all protocol layers, device types, and vendors involved. Monitoring technology is crucial for the success of the distributed SDN model. [4]

Virtual networking software segments physical networks into multi-tenant or multi-application networks, and it could play a role in building the top software-control layer that all SDNs need. Standards for northbound APIs from either the central or distributed SDN models are not defined, making it difficult to marry virtual networks to SDNs except by customization.

The basic elements of SDN are common to all models, but the implementation of any of these elements may not be compatible with both models. Those who want to test or deploy SDNs will initially likely have to choose which model to adopt and ensure their infrastructure matches the model's requirements.

4 SDN Models

While SDN is founded on the concept of centralized software that controls the flow of data in switches and routers, there are four primary models or approaches to implementing SDN:

4.1 Open SDN

Open SDN, also known as OpenFlow-based SDN, is an SDN model that uses open protocols, such as OpenFlow, to manage the routing of data packets across the network. With Open SDN, network administrators can configure and manage network devices through a centralized controller, which provides more control and flexibility over the network.

Open SDN provides organizations with the ability to program and customize network behavior to meet specific business needs. The centralized Controller can make real-time decisions based on network traffic and can optimize network performance accordingly. This approach allows network administrators to adjust the network configuration and policies promptly, which is especially important for environments with rapidly changing network requirements.

One of the key benefits of Open SDN is its openness. The use of open protocols ensures that the network infrastructure is not tied to specific vendors or proprietary technologies, which can reduce costs and increase flexibility. OpenFlow, which is the most commonly used protocol in Open SDN, has a wide range of vendors that support it, making it easier for organizations to adopt this model. [1]

4.2 API SDN

API SDN is a software-defined networking model that utilizes programming interfaces, known as southbound APIs, to control the flow of data between devices. This model allows organizations to develop custom applications and management tools that can better cater to their specific needs. By using these interfaces, organizations can manage network devices from a central location, making it easier to configure and customize network policies. This approach provides more flexibility and control, allowing network administrators to optimize network performance based on real-time traffic patterns. [1]

4.3 Overlay Model SDN

This approach is used to create a virtual network above the current hardware, which includes tunnels with channels to data centers. The model proceeds to assign bandwidth in each channel and allocate devices to their respective channels. This approach enables organizations to create custom virtual networks on top of their physical network without the need to replace their existing infrastructure. [1]

4.4 Hybrid Model SDN

This model combines SDN and traditional networking, allowing organizations to use the optimal protocol for each type of traffic. Hybrid SDN is frequently used as an incremental approach to SDN, allowing organizations to gradually transition to more advanced SDN architectures while maintaining some aspects of their existing network infrastructure. [1]

By choosing the right model, organizations can create more efficient, flexible, and agile networks that can meet the unique needs of their business.

5 SDN Controller Architecture

The SDN Controller is a vital component of a software-defined network, offering a centralized platform for network traffic management and resource control. The SDN controller comprises multiple critical components, such as the management interface, southbound API, northbound API, control plane, and data plane. These components work together to provide network administrators with a secure, efficient, and adaptable approach to managing network resources and operations. The SDN Controller architecture can be categorized into two primary components: the control plane and the data plane. The control plane makes decisions about network traffic, while the data plane forwards network traffic.

The Data Plane includes a range of network devices, both physical and virtual, and its primary function is to forward data. In traditional networks, both the control and data plane exist within the same device, but with SDN, network devices only have a data plane, meaning that their primary role is solely to forward data. This results in a highly effective forwarding mechanism that improves efficiency.

The control plane refers to the component responsible for managing the forwarding behavior of the network. Specifically, the control plane defines how network traffic should be routed and processed by the network devices in the data plane. The control plane in an SDN controller typically receives instructions from higher-level applications and translates them into low-level network configurations that are sent to the data plane.

APIs enable communication between the SDN Controller and the underlying hardware, facilitating interaction between the control plane and the data plane.

The management interface serves as the primary interface between the SDN Controller and network administrators, offering a GUI or API that facilitates the configuration and management of the SDN network. The management interface is typically accessible through a web browser or a command line interface.

The southbound API is the interface between the SDN Controller and the underlying hardware, such as switches and routers. It sends configuration information and commands to these devices and receives status information from them, allowing the SDN Controller to interact with the underlying hardware and direct network traffic based on policies and configurations.

The northbound API is the interface between the SDN Controller and higher-level network management tools, such as network monitoring and management systems. It communicates network status and other information to these tools, enabling them to monitor the SDN network and provide administrators with detailed information about network performance and security. [5]

6 SDN Advantages

SDN architecture brings numerous advantages by centralizing network control and management.

- **Simplified Network Management**

One of the primary advantages is simplified network management, as SDN separates the packet-forwarding functions from the data plane. This allows for direct programming and simpler network control, such as configuring network services in real-time or allocating virtual network resources quickly to change the network infrastructure through one centralized location. With the centralized control of SDN architecture, network administrators can streamline their operations and easily manage their network infrastructure.

SDN architecture allows for ease of network control and simplified network management, making it an attractive option for organizations looking to streamline their operations. [6]

- **Agility**

Another benefit is agility, as SDN enables dynamic load balancing to manage traffic flow as needed, reducing latency and increasing network efficiency.

In a traditional network architecture, network devices are configured to perform specific functions, and their behavior is static. This means that any changes to the network require significant manual configuration and testing, which can be time-consuming and error-prone such as adding new devices or adjusting routing policies.

In contrast, SDN architecture enables dynamic load balancing, which means that the network can adapt to changing traffic patterns and usage. The central Controller can monitor network traffic in real-time and allocate resources to optimize performance, ensuring that resources are available where and when they are needed. This can result in reduced latency, improved response times, and increased overall efficiency of the network.

For example, if a particular application is experiencing heavy traffic, the SDN controller can dynamically allocate additional network resources to that application to ensure that it continues to function smoothly. Conversely, if a particular application is not being used, the SDN controller can reallocate resources to other applications to maximize network utilization.

Overall, the agility provided by SDN architecture enables network operators to respond quickly and effectively to changing network conditions without the need for manual configuration and testing. This can help organizations in lowering costs, improve network performance, and better serve their customers. [6]

- **Flexibility**

A software-based control layer provides more flexibility for network operators to control the network, change configuration settings, provision resources, and increase network capacity. In traditional network architectures, network control and management are tightly coupled with

hardware. This can make it difficult to make changes to the network, as each device must be configured separately, and changes can be time-consuming and error-prone. SDN provides more flexibility by separating the control plane from the data plane, allowing the control plane to be implemented in software and run on commodity hardware. This means that network operators can more easily control the network, change configuration settings, provision resources, and increase network capacity through a centralized software-based control layer. For example, if a company needs to add more capacity to its network, it can simply provision more virtual network resources through the centralized control layer rather than having to physically add more hardware to the network. This can result in a more agile and scalable network architecture. [6]

- **Enhanced Network Security**

SDN provides greater control over network security through centralized policy enforcement, which allows network administrators to set policies from a single location. This means that security policies can be consistently enforced across the network regardless of the underlying hardware or topology. By using micro-segmentation, administrators can segment the network into smaller, more manageable pieces and apply security policies tailored to specific workloads or network segments. This reduces complexity and provides a more granular approach to security that can be applied to various network architectures, including public, private, hybrid, or multi-cloud. SDN's centralized approach to security also simplifies policy management and reduces the risk of human error in configuration. [6]

- **Simplified Network Design and Operation**

It is another benefit, as administrators can use a single protocol to communicate with a wide range of hardware devices through a central controller. This simplifies the process of network management and reduces the need for complex configuration settings. Additionally, organizations can choose networking equipment that meets their specific needs, as SDN offers more flexibility in choosing open controllers instead of relying on vendor-specific devices and protocols. By simplifying the network design and operation, SDN enables organizations to more efficiently manage their network resources and adapt to changing business needs. [6]

- **Bringing Telecommunications up-to-date**

By leveraging SDN technology along with virtual machines and network virtualization, service providers can offer separate network separation and control to their customers, which improves scalability and enables bandwidth on demand for customers with variable bandwidth usage. This can lead to more efficient and cost-effective telecommunications services, helping to bring them up-to-date with the latest technologies. [6]

7 OpenDaylight

OpenDaylight is an open-source platform for developing, deploying, and managing Software-Defined Networking solutions. It is a modular, scalable, and extensible platform that provides centralized control over the network and allows network administrators to program the network according to their needs. OpenDaylight is designed to work with various networking protocols and provide APIs for application developers to build new applications on top of it. This allows organizations to automate their networking processes, making their networks more flexible and scalable. It also supports both Northbound and Southbound APIs, making it an ideal choice for organizations looking for a flexible, open-source SDN solution.

8 OpenDaylight Architecture

The OpenDaylight platform employs the Model-Driven Service Abstraction Layer (MD-SAL) to facilitate communication between network devices and applications. The MD-SAL models these devices and applications as objects and utilizes YANG models to describe their capabilities in a generalized manner without requiring knowledge of specific implementation details. The MD-SAL serves as an intermediary between these YANG models, enabling data exchange and adaptation. The MD-SAL identifies models by their role in a given interaction, either as a "producer" that implements an API and provides data or as a "consumer" that uses the API and consumes data. These roles are more accurate descriptors of MD-SAL interactions than "northbound" and "southbound." A protocol plugin and its model can operate as either a producer or consumer, for example. [7]

The MD-SAL connects producers and consumers by searching its data stores and exchanging information between them. Consumers can locate a suitable provider, while producers can generate notifications. Consumers can receive notifications and request data from providers using Remote Procedure Calls (RPCs). Producers can store data in the MD-SAL's storage, while consumers can access stored data. Producers are responsible for implementing an API and providing its data, while consumers utilize the API and consume its data. [7]

OpenDaylight includes several applications and plugins that provide functionality for various networking purposes. Some of the applications and plugins that are shipped with OpenDaylight include:

• OpenFlow Plugin: This plugin implements the OpenFlow protocol and provides support for Software Defined Networking (SDN) deployments.

• BGP-LS: This plugin implements the Border Gateway Protocol - Link State (BGP-LS) for link-state routing.

• BGPCEP: This is an application that implements the Border Gateway Protocol for use in path-vector routing.

• NETCONF/YANG: This plugin implements the NETCONF protocol and provides support for the YANG data modeling language.

• L2 Switch: This is an application that provides basic Layer 2 (L2) switching functionality for use in simple networks.

The OpenDaylight Controller is designed with a modular architecture that allows for the integration of plugins and services to provide additional functionality. At the core of the OpenDaylight Controller is the Model-Driven Service Abstraction Layer (MD-SAL), which provides a consistent interface for network devices and applications to interact with each other. The MD-SAL uses YANG models to describe the capabilities of network devices and applications and facilitates communication between them through the exchange of information.

In addition to the MD-SAL, the OpenDaylight Controller has a variety of plugins and services that can be added to provide additional functionality. These plugins and services can be developed by the OpenDaylight community or by third-party vendors and can be integrated into the Controller to provide features such as network virtualization, security, and analytics. [8]

One of the key benefits of the OpenDaylight Controller's modular design is its flexibility. By allowing users to add and remove plugins and services as needed, the Controller can be customized to meet specific network requirements. This also allows for rapid development and deployment of new features and functionality.

9 Key Features of OpenDaylight

OpenDaylight's microservices architecture is one of the key distinguishing features of the platform. Instead of having a monolithic controller that provides all services, OpenDaylight's architecture enables users to select and enable specific services based on their requirements. This approach provides increased adaptability and scalability since users only need to install and enable the services they need.

Another important distinction of OpenDaylight is its support for a wide range of network protocols. OpenDaylight's modular design allows it to support various protocols, making it a more versatile and flexible SDN option. The platform supports popular protocols like OpenFlow and NETCONF, as well as emerging protocols like P4 BGP and LISP. This flexibility makes it easier for network administrators to work with different types of network devices and protocols.

OpenDaylight also leverages a Model-Driven Service Abstraction Layer (MD-SAL) to manage network services. This layer provides a uniform interface for network services and abstracts the underlying complexity of network protocols. It uses YANG models to create datastore schemata, generate application REST API, and automate code generation. This approach simplifies the

development of network applications and allows network administrators to manage network services more efficiently.

Overall, OpenDaylight's microservices architecture, support for a wide range of network protocols, and Model Driven Service Abstraction Layer make it a versatile and flexible SDN platform that can be customized to meet a wide range of network management requirements.

10 OpenDaylight Controller

The OpenDaylight Controller is a Java-based software-defined networking controller that is designed to be flexible and modular, allowing it to support a variety of use cases. It uses YANG as its modeling language to represent various aspects of the system and applications, making it easier to develop and manage network resources. The OpenDaylight Controller is built on top of several key technologies.

- **OSGI**

The Open Services Gateway Initiative (OSGI) framework is a powerful tool for developing modular applications. In the context of the OpenDaylight Controller, OSGI allows developers to build and deploy software modules, called bundles, which can be dynamically loaded and unloaded at runtime. Bundles can be thought of as self-contained units of functionality that can be easily combined to create complex applications.

OSGI provides a set of APIs that allow bundles to interact with each other at runtime, making it easy to build modular applications that can be customized and extended with new features. Bundles can depend on other bundles, and the OSGI framework ensures that all dependencies are satisfied at runtime. This allows developers to easily mix and match components to create new applications without having to worry about dependencies and conflicts. [9]

The use of OSGI in the OpenDaylight Controller makes it highly modular and extensible. Developers can easily create new bundles that add functionality to the system, and existing bundles can be updated or replaced without having to restart the entire system. This makes it easier to add new features to the OpenDaylight Controller and to customize it for specific use cases.

- **Karaf**

The Karaf application container is a key component of the OpenDaylight Controller, providing a convenient and powerful way to manage and deploy applications within the system. Built on top of the OSGI framework, Karaf simplifies the operational aspects of packaging and installing applications, making it easier to manage and deploy software modules.

One of the key benefits of Karaf is its support for a wide range of packaging formats, including JAR, WAR, and OSGI bundles. This means that developers can package their applications in various ways, depending on their specific needs and requirements. Additionally, Karaf provides

several tools for managing and deploying applications, including a command-line interface (CLI) and a web-based console.

Another important feature of Karaf is its support for the hot deployment of applications. This means that new applications or updates to existing applications can be installed and activated without having to restart the entire system. This is a powerful capability that allows developers to iterate quickly and make changes to the system without disrupting other applications. [9]

- **YANG**

It is a data modeling language used to define data structures, such as configuration and state data, hierarchically. The use of YANG in the OpenDaylight Controller enables the representation of network resources in a standardized and consistent way. This makes it easier to develop and manage network resources, as it provides a common language for describing and manipulating data.

By using YANG, the OpenDaylight Controller can define the structure and content of data for different network resources, such as switches, routers, and network interfaces. YANG models define the properties, relationships, and constraints of these resources, providing a standardized way to represent them.

One of the key benefits of using YANG is that it provides a clear separation between data modeling and application logic. This makes it easier to develop and maintain applications, as developers can focus on implementing application logic without having to worry about the underlying data structures.

The use of YANG also enables the OpenDaylight Controller to expose YANG-modeled RPCs and notifications. RPCs are remote procedure calls that allow clients to invoke specific operations on network resources, while notifications enable the Controller to inform clients about changes in the state of these resources. This makes it easier to automate network management tasks and enables the development of applications that can interact with the Controller using standard interfaces. [9]

11 NETCONF and RESTCONF

The OpenDaylight Controller provides access to applications and data using model-driven protocols such as NETCONF and RESTCONF.

NETCONF

NETCONF (Network Configuration Protocol) is a network management protocol that provides a standardized mechanism for configuring network devices, including routers, switches, and firewalls. The protocol is based on XML and uses a client-server model, with the client sending requests to the server to retrieve or update configuration information. The NETCONF protocol is supported in OpenDaylight in the form of a southbound plugin, which allows the Controller to

communicate with network devices that support the NETCONF protocol. The NETCONF plugin in OpenDaylight provides a standardized interface for configuring and managing network devices, enabling administrators to use the OpenDaylight Controller to control the behavior of the network as a whole. The plugin supports transactional updates, selective queries, and error reporting, providing a flexible and scalable approach to network management.

In addition to supporting the NETCONF protocol, OpenDaylight also includes a set of test tools for simulating NETCONF devices and clients. These tools allow developers and administrators to test the functionality and interoperability of the NETCONF plugin and other network management tools in a controlled environment, helping to ensure the reliability and stability of the network. [10]

RESTCONF

RESTCONF (REpresentational State Transfer Configuration Protocol) is an HTTP-based protocol that enables the retrieval and manipulation of network configuration data using standard HTTP methods such as GET, POST, PUT, and DELETE. It is based on the REST architecture and is used as the primary northbound API in the OpenDaylight SDN controller, providing a standard interface for accessing and managing network resources. RESTCONF is a modern alternative to the legacy SNMP protocol and uses the YANG data modeling language to represent network configuration and state information. The OpenDaylight Controller includes a RESTCONF plugin that offers a RESTful API for device configuration and management using the RESTCONF protocol, providing a standard and flexible way for network administrators to interact with network devices. This plugin adheres to the protocol standard and provides a comprehensive set of REST APIs for configuring and managing network resources. [10]

12 OpenDaylight Installation

The OpenDaylight Controller is a software program that operates in a Java Virtual Machine (JVM). Since it is a Java-based application, it has the potential to operate on any operating system that can support java. However, for the best result, it is suggested to use a modern Linux distribution and a Java Virtual Machine version 1.7.

OpenDaylight (ODL) provides both a command line interface (CLI) and a web graphical user interface (GUI).

The CLI, also known as the Karaf shell, can be accessed through a terminal after starting ODL. The CLI provides a rich set of commands for managing ODL and its components.

The web GUI, known as the OpenDaylight Controller, provides a visual representation of the network and allows administrators to manage network elements and services through a web browser. The web GUI is accessible at http://localhost:8181/index.html by default after starting ODL.

The latest version of OpenDaylight is Sulfur. However, this version no longer includes the maintained version of the web interface and the DLUX interface, which is typically part of OpenDaylight. The web UI (known as DLUX in Oxygen and earlier versions) is no longer supported after the release of Oxygen.

Here are the general steps to install OpenDaylight (ODL) on Ubuntu 22.04 machine:

1. Prepare the operating system

Initially, we set up a virtual machine with the Ubuntu 22.04 operating system and used the Ubuntu apt package manager to update the operating system and applications.

- $ sudo apt-get –y update

2. Install the Java JRE

The following command installs the JAVA 11 JDK:

- $ sudo apt-get –y install openjdk-11-jre

To locate JAVA 11 on your server, run the "update-alternatives" command. This command enables you to choose the default Java version if multiple installations of java exist on your server. If "update-alternatives" presents a list of Java versions on your server, select JAVA 11 from the options.

- $ sudo update-alternatives –config java

3. Set JAVA_HOME

For OpenDaylight to run, JAVA_HOME must be set to the location of the entire Java toolkit.

To find the complete path to your Java executable, you can use this command:

- $ ls –l /etc/alternatives/java

Remove bin/java from the path. This sets JAVA_HOME to the location of the JDK and not the binary. Run this command to edit your BASH resource file to set and persist the JAVA_HOME value:

- $ echo ‘export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64’ >> ~/.bashrc

To set JAVA_HOME for the first time, you can source the resource file:

- $ source ~/.bashrc

Once you source the file, $JAVA_HOME should end with /java-11-openjdk-amd64. You can check with this command:

- $ echo $JAVA_HOME

4. Download the ODL distribution.

You can download the latest OpenDaylight distribution from this link:

https://docs.opendaylight.org/en/latest/downloads.html

- $ wget https://nexus.opendaylight.org/content/repositories/opendaylight.release/org/opendaylight/integration/opendaylight/16.3.0/opendaylight-16.3.0.zip

5. Unzip the distribution zip file:

- $ unzip opendaylight-16.3.0.zip

Navigate to the distribution directory:

- $ cd opendaylight-16.3.0

6. Start the OpenDaylight:

- $ ./bin/karaf

In this console, there are commands like the feature:

- Opendaylight-user@root>

Features are added to the system using the Apache Karaf container, which provides a flexible way to manage and configure the different components and features of the system. [11]

13 GUI architecture and implementation

This project aims to develop a Graphical User Interface for the OpenDaylight platform that is intuitive and user-friendly. To achieve this goal, we have chosen to use the Vue.js framework, Vite, and FastAPI as our tools for development.

13.1 Vue.js

It is a JavaScript framework designed specifically for creating user interfaces using standard HTML, CSS, and JavaScript. It employs a component-based and declarative approach, making it suitable for creating basic and complex user interfaces. Vue.js framework offers a range of advantages, such as an easy-to-use API, efficient state management, and a thriving developer community. Additionally, Vue.js's modular architecture ensures a sustainable codebase, making it simpler to maintain and upgrade the GUI in the future.

We have chosen to employ the latest version of Vue.js, Vue 3, for this project, as it provides superior performance, smaller bundle sizes, and enhanced compatibility with modern JavaScript syntax and features. Vue 3's improved tooling, including the Composition API, allows for more flexible and modular management of component state and logic. Furthermore, Vue 3 is highly modular and customizable, which provides developers with the ability to select only the necessary features and components for their projects. [12]

13.2 Vite

We decided to use Vite as our build tool for the GUI based on Vue.js. Vite is a powerful and efficient build tool that simplifies the development and bundling of JavaScript projects. It offers various benefits, such as hot module replacement, which allows for quick updates to the GUI without refreshing the entire page. This feature speeds up the development process and enhances productivity. Moreover, Vite uses modern features like ES modules and dynamic imports resulting in smaller bundle sizes and improved performance, thereby providing a streamlined development experience.

Vite's plugin system provides several tools and features that can be seamlessly integrated into our project. These include support for popular CSS pre-processors and other tools that can help streamline our development workflow. Vite is a well-known build tool and development server that works well with modern web technologies like Vue.js, React, and other frameworks and libraries. It is commonly used as a bundler for these technologies. [13]

13.3 FastAPI

For calling APIs and interacting with the OpenDaylight platform, our project utilizes the fastAPI framework as a middle layer. FastAPI is a Python-based web framework that is known for its speed and modern features. By using fastAPI, we can simplify the process of calling APIs and interacting with OpenDaylight. The framework acts as a bridge between the Vue.js-based front-end and the OpenDaylight API, eliminating the need for developers to handle the low-level details of the API.

FastAPI’s simplicity is one of its major advantages. It provides a straightforward syntax for defining API routes and handling requests and responses, which makes it a popular choice for

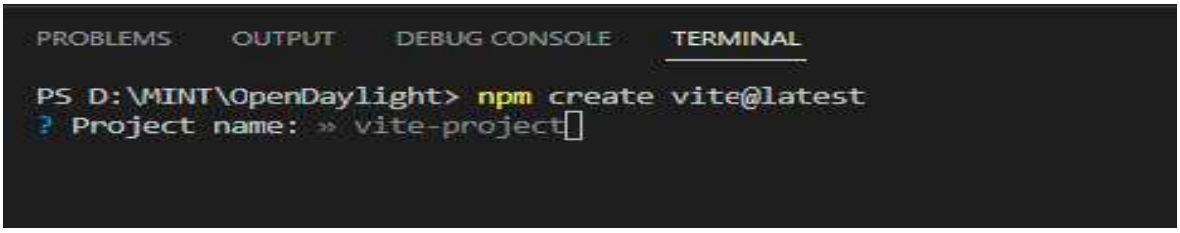
developing web applications and services. This allows developers to easily create complex and scalable web applications. FastAPI is also highly efficient. It is built using the latest Python features, such as asynchronous programming and type annotations. As a result, it can quickly and efficiently handle large numbers of requests without sacrificing performance. This is crucial for our project, where we need to rapidly call APIs and interact with the OpenDaylight platform. [14]

14 UI Setup Instruction

To develop this project, Visual Studio Code is utilized as an Integrated Development Environment (IDE) that runs on a desktop. You can follow these steps:

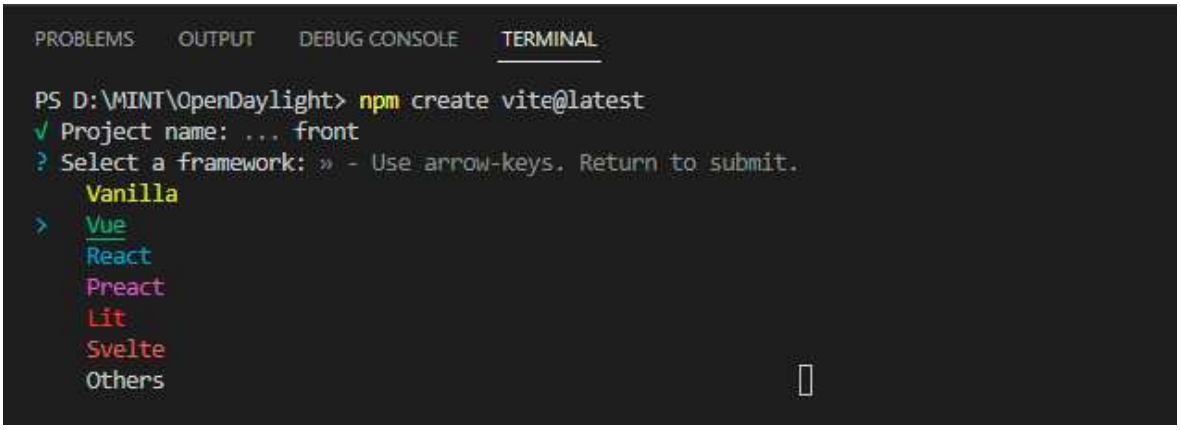
1. Firstly, you should create a new folder in your system to contain your project files and open this folder in Visual Studio Code. (A new folder named “OpenDaylight” has been created)

2. Open a new terminal in this folder with Ctrl + Shift + (backtick) on Windows and run the command "npm create vite@latest."

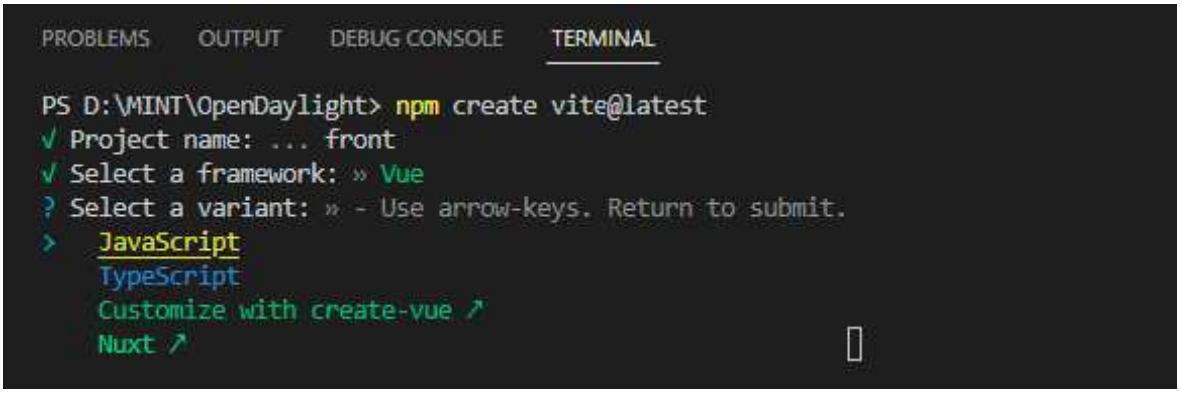


The npm command is used to interact with the Node Package Manager (NPM), which is a package manager for Node.js. The create command is a built-in feature of NPM that allows you to quickly create new projects from pre-built templates. In this case, the vite@latest argument specifies the Vite package and its latest version to be used as the template for the new project. This command will create a new directory with the name of the project and will contain all the necessary files and dependencies to start developing a Vite-based project. After running this command in the terminal, you will be prompted to enter a project name.

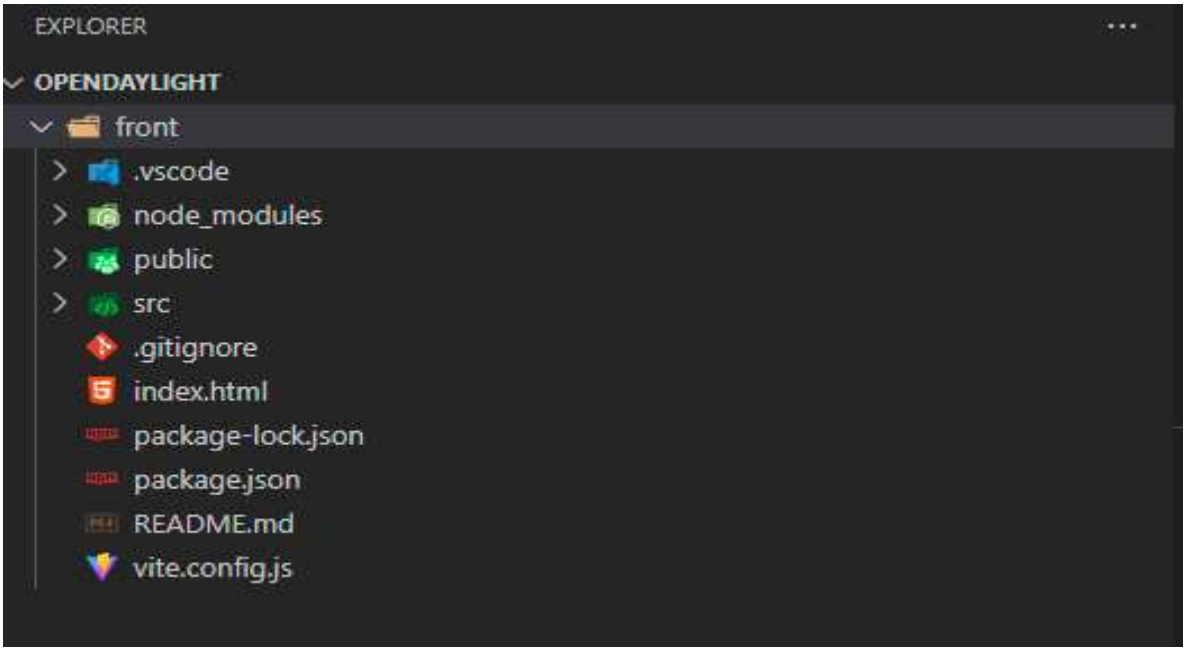
3. After entering the project name in the terminal (the “front” name is selected), you are prompted to select a framework. Choose Vue.js as the framework for this project.



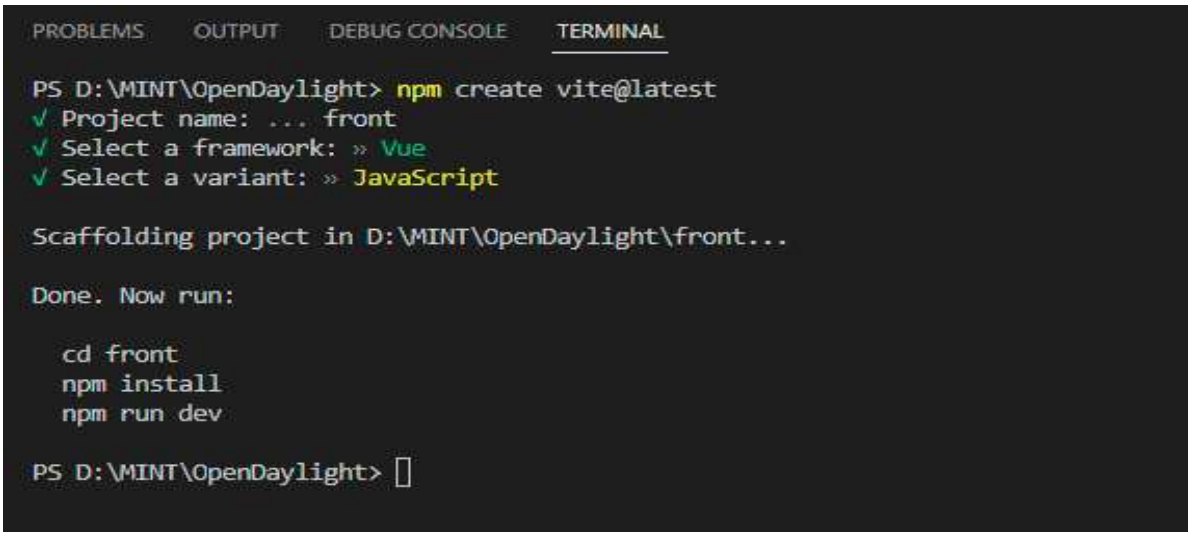
4. In the next step, you should select the JavaScript variant.



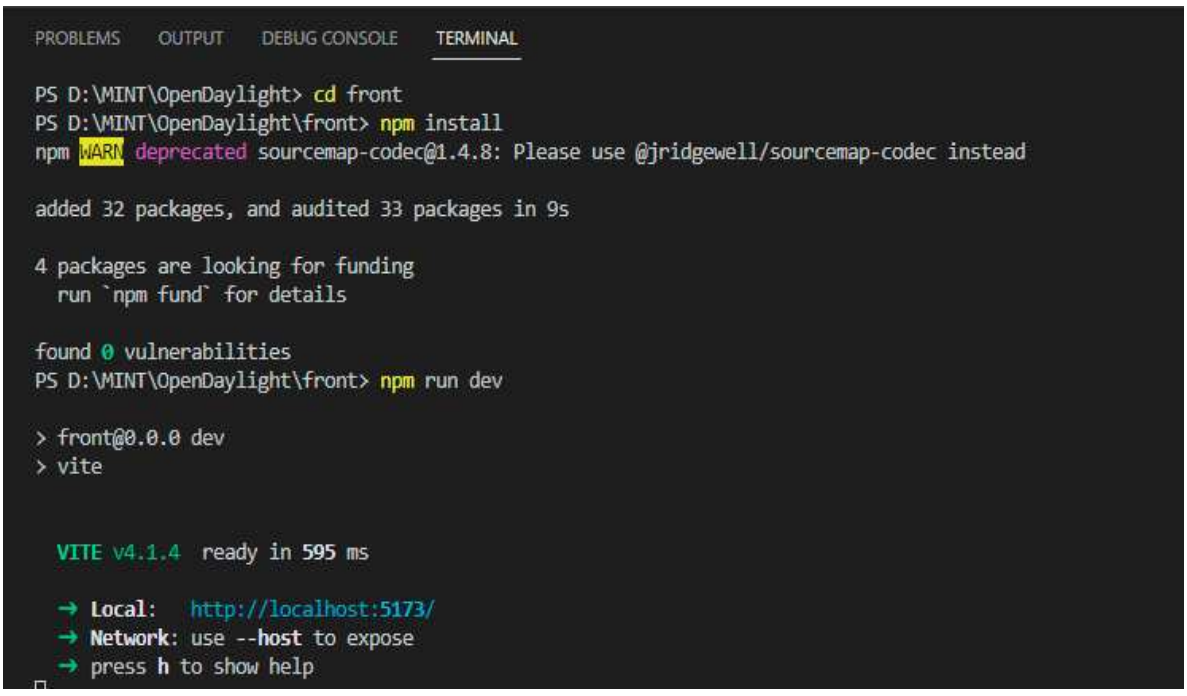
5. In this step, some new files and folders will be created under the “front” directory.



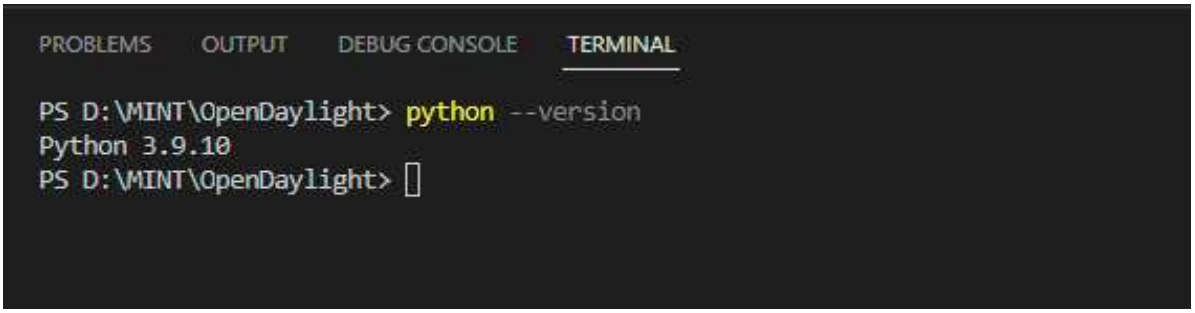
6. Once you are in the project directory, navigate to the “front” directory using the cd command.
7. Run the “npm install” command to install all the required dependencies for the project.
8. Finally, run the “npm run dev” command to start the development server.



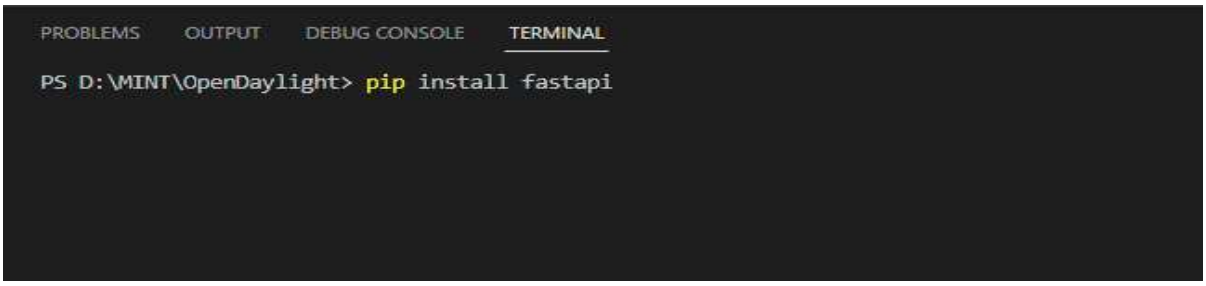
9. After running these commands, you should be able to access your application by navigating to http://localhost:5173 in your web browser.



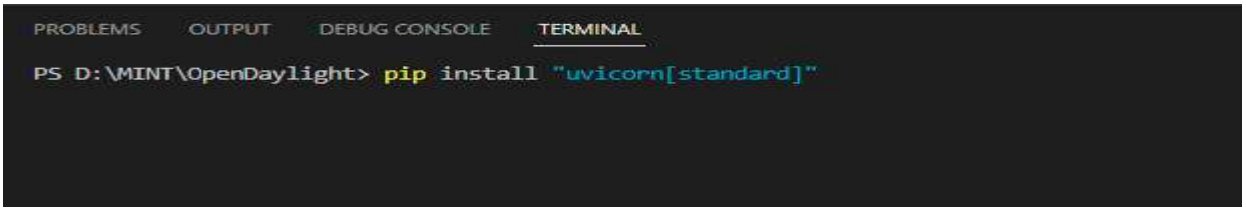
10. To install FastAPI, make sure you have Python 3.7 or above installed on your system. You can check the version by running the following command in your terminal or command prompt:



11. Open your terminal and type the following commands for installing FastAPI and its dependencies:



12. In addition to FastAPI, you will also need an ASGI server like Uvicorn or Hypercorn to run your application in production. If you want to install Uvicorn with additional optional dependencies, you can use the following command:



13. Navigate to the “OpenDaylight” folder, create a new folder (named “api”), and create a file named main.py inside this folder with these content:

main.py

main.py > read_item

```python
from typing import Union

from fastapi import FastAPI

app = FastAPI()


@app.get("/")
def read_root():
    return {"Hello": "World"}


@app.get("/items/{item_id}")
def read_item(item_id: int, q: Union[str, None] = None):
    return {"item_id": item_id, "q": q}
```

The first endpoint, defined with @app.get ("/"), returns a simple JSON object with the message "Hello, World."

The second endpoint, defined with @app.get ("/items/{item_id}"), takes an integer path parameter item_id and an optional query parameter q that can be a string or None. The endpoint returns a JSON object that includes both the item_id and the q value.

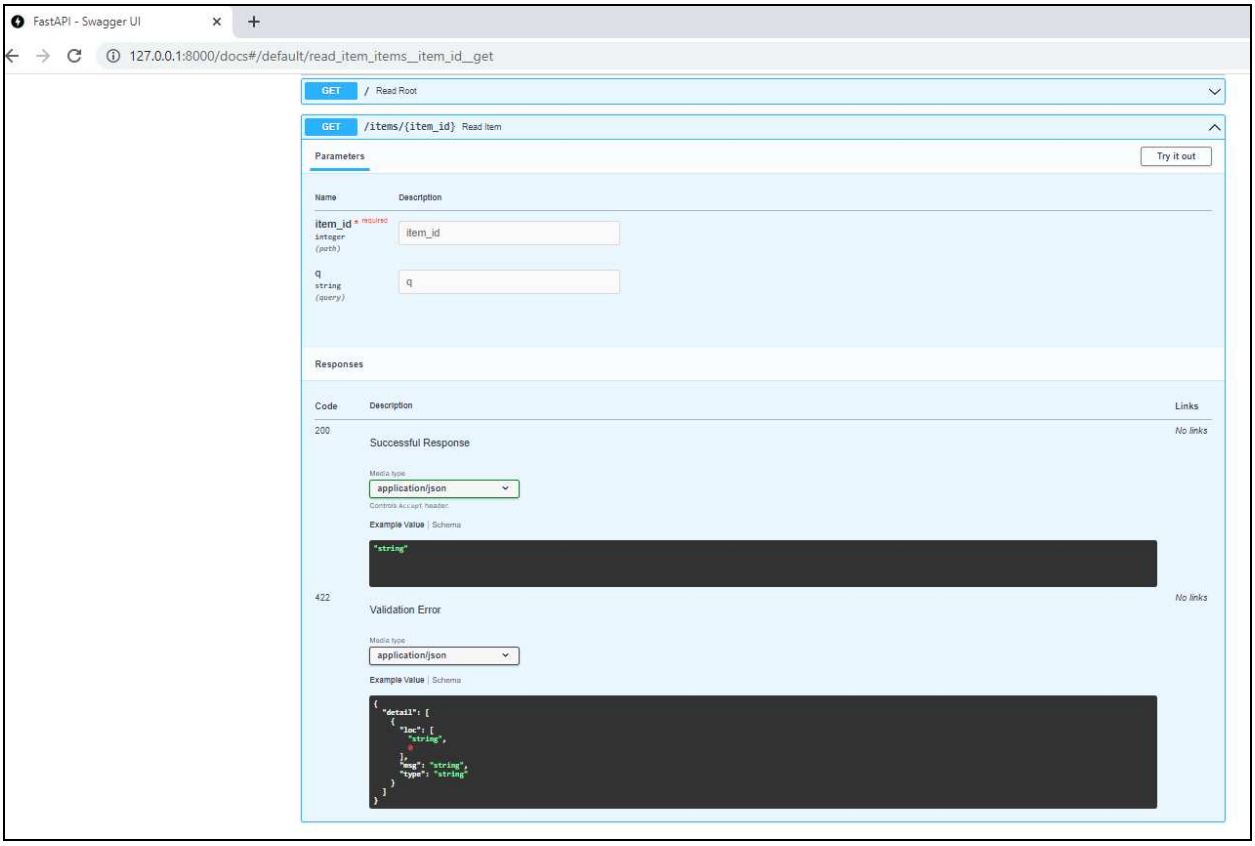
14. You can run this file using the following command:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS D:\MINT\OpenDaylight> cd api
PS D:\MINT\OpenDaylight\api> uvicorn main:app --reload
INFO:     Will watch for changes in these directories: ['D:\\MINT\\OpenDaylight\\api']
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [17672] using WatchFiles
INFO:     Started server process [13528]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

This command tells Uvicorn to run your application using the main module and the app instance as specified in your code. The --reload flag enables automatic reload of the server whenever you make changes to your code.

15. Now the application will be accessible at http://localhost:8000/ or http://127.0.0.1:8000/.

16. You can navigate to http://127.0.0.1:8000/docs in your web browser and see the automatic interactive API documentation provided by Swagger UI.



The documentation page will display all the available endpoints and their respective HTTP methods (e.g., GET, POST, PUT, DELETE), as well as any input parameters or request bodies required by each endpoint. You can use this page to test your API endpoints by submitting requests and viewing the responses.

15 GUI Components and Functionality

The GUI components that were designed for this application are aimed at providing a user-friendly interface that allows users to efficiently manage network devices. The sidebar serves as a quick and easy navigation menu, while the buttons on the sidebar enable users to access all the main features of the application. The icons used for Cisco and Juniper routers aid users in quickly identifying the type of device they are working with.

The form component allows users to input device information and add new devices to the network. The messages component displays a message indicating whether a device was successfully added or if there was an error. This helps users quickly identify and resolve issues with device addition.

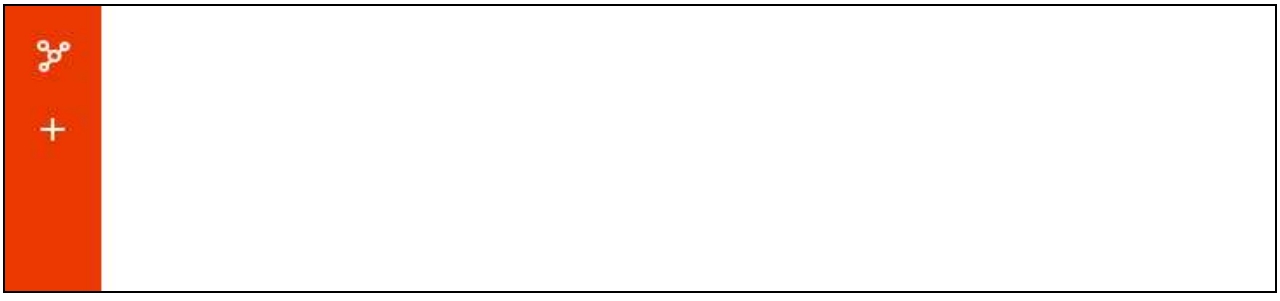
The context menu component provides users with the ability to perform different actions on each device. This feature improves the efficiency of the user by eliminating the need to navigate to a different screen to act. The five main tasks available in the context menu include getting configuration information, changing the hostname of the device, adding a static route, adding an interface, and deleting a node.

The configuration pages component provides users with detailed device information and enables them to perform specific tasks related to the device. Overall, these GUI components improve the user experience by providing easy navigation, efficient device management, and detailed device information.

A detailed analysis of the GUI components included in our project can be found below.

Sidebar: A sidebar is designed on the left side of the page, which contains two buttons. This sidebar serves as a quick navigation menu for the different features and functions of the application.

Buttons: The sidebar includes two buttons. The first button is used to display all of the devices that are currently defined in OpenDaylight. The second button is used to add a new device to the network.



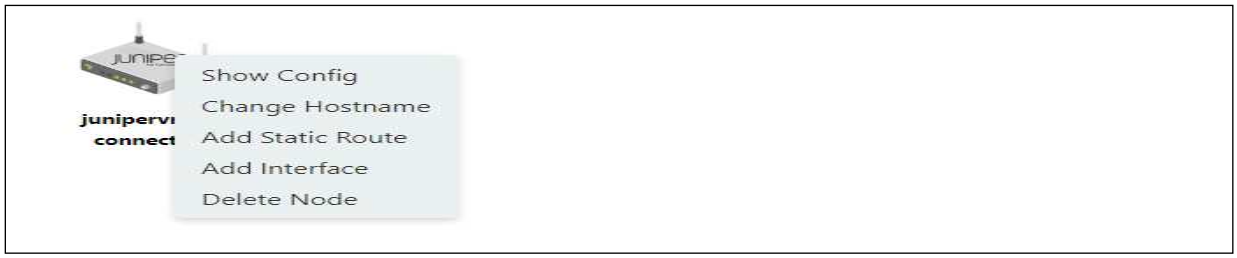
Icons: Cisco and Juniper routers are distinguished by using different icons. This allows users to quickly identify the type of device that they are working with.

Forms: When users click on the "add new device" button in a sidebar or select options "change hostname," "add Interface," or "add static route" in the context menu, a new page related to that

task is opened with a form. This form allows users to enter the required information, such as the device's name, IP address, username, and password.

Messages: After submitting the form, the application displays a message indicating whether the device was successfully added or if there was an error. If a user tries to add a device with the same name as an existing device, the application displays a message indicating that the device name already exists.

Context Menus: To perform different actions on each device, you have enabled context menus. Users can access these menus by right-clicking or left-clicking on each device. Five main tasks are provided: getting configuration information, changing the hostname of the device, adding a static route, adding an interface, and deleting a node.



Configuration Pages: When a user selects the "show config" option from the context menu, a new page is opened with the configuration of the selected device. Similarly, when a user selects the "change hostname" option, a new page is opened with a field for entering a new hostname. After submitting the new hostname, a message is displayed indicating whether the hostname was successfully changed or not.

16 GUI Components Overview and API Explanation

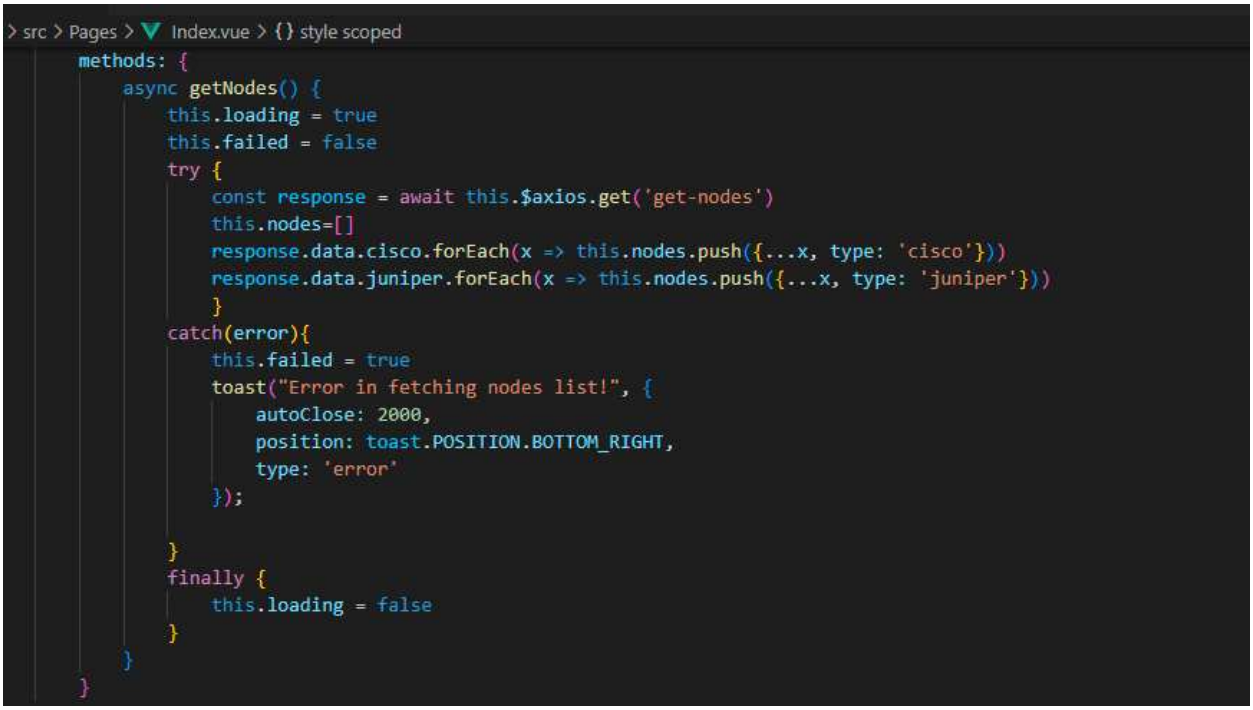
In this section, you can find detailed explanations of each component used in the application, as well as the corresponding files that contain the code for those components. Additionally, the APIs that are used in the application will be explained, including their endpoints, methods, and parameters. To note, we utilize Axios as an HTTP client in Vue.js for calling REST APIs. Axios is a popular JavaScript library that can be used to make HTTP requests from a web application. In Vue.js, Axios can be used to make HTTP requests to REST APIs by sending GET, POST, PUT, DELETE, and other HTTP methods to retrieve, create, update, or delete resources on a server. By using Axios in Vue.js, developers can easily interact with REST APIs and fetch data asynchronously from a server without having to manually handle low-level AJAX requests and responses.

16.1 Show Nodes

The "index. vue" component is used to create a "Show Nodes" button in the user interface. When the user clicks on this button, the getNodes method is executed.

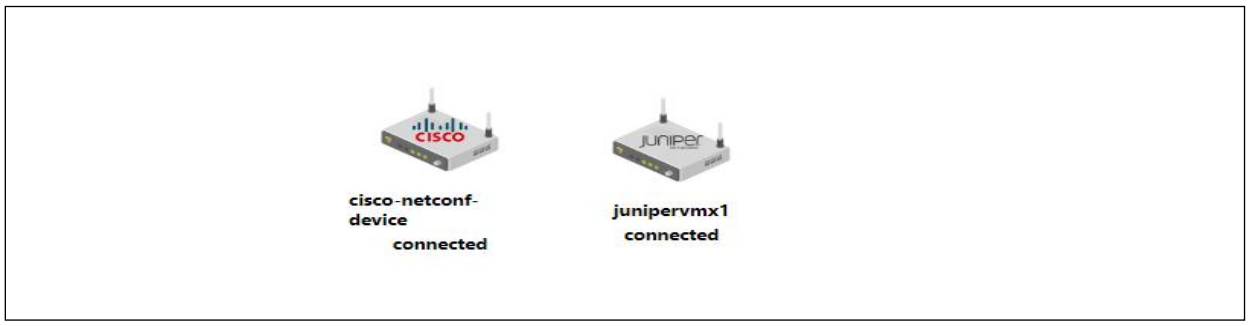
The getNodes method sets the loading flag to true, indicating that the data is being loaded. It also sets the failed flag to false, indicating that the data has not yet failed to load.

The method then makes an HTTP GET request to an API endpoint called get-nodes using the $axios plugin. If the request is successful, the nodes data returned in the response is assigned to the nodes array in the component's data. The loading flag is set to false, indicating that the data has finished loading. If the request fails, the failed flag is set to true, and an error message is displayed using the toast function.



On the backend, the FastAPI endpoint /get-nodes is defined using the @app.get decorator for the HTTP GET request method. When this endpoint is accessed, it runs the root () function, which retrieves node information from Juniper and Cisco RESTful APIs using a function called getNodesFrom(). The getNodesFrom() function is passed the URLs of the APIs using global variables. The node information from both APIs is combined into a single list called all_nodes. The root () function returns a JSON response containing the node data, with separate keys for the Juniper and Cisco nodes.

```python
main.py > root
@app.get("/get-nodes")
async def root():
    global juniper_api_server
    global cisco_api_server
    juniper_nodes = getNodesFrom(f'{juniper_api_server}rests/data/network-topology:network-topology?content=nonconfig')
    cisco_nodes  = getNodesFrom(f'{cisco_api_server}rests/data/network-topology:network-topology?content=nonconfig')
    all_nodes = cisco_nodes + juniper_nodes
    return { "cisco": cisco_nodes, "juniper": juniper_nodes } #{"cisco":cisco_nodes, "juniper":juniper_nodes}}
```



16.2 Add New Device

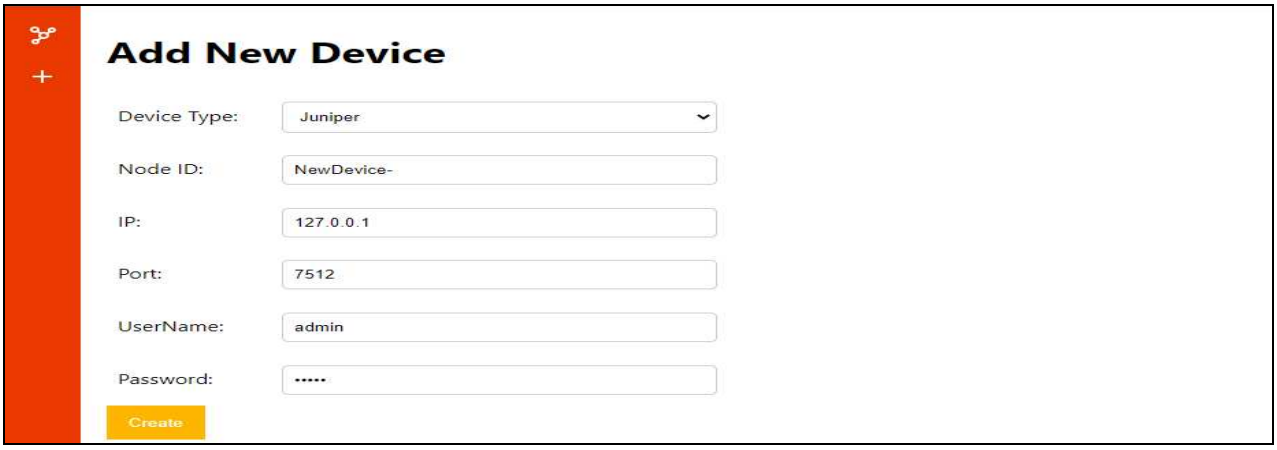
In the Vue.js project, when the user fills out the form for adding a new device and submits it, the "addDevice" method is called. The addDevice method is defined in the AddDevice.vue component. This method sets the loading state to true and clears any previous error message. It then sends an HTTP PUT request to the "/create-device" API endpoint using the Axios library with the device object as the request body. The response from the server is then checked to determine whether the device was successfully added or not. If the server returns a status code of 201, a success notification is displayed, and the user is redirected to the home page. If the status code is 204, it means that the device already exists, and an error message is displayed to the user.

```vue
src > Pages > AddDevice.vue > {} script > default > methods > addDevice
    methods: {
        async addDevice() {
            this.loading = true
            this.errorMessage = ''
            const response = await this.$axios.put(`/create-device`, this.device)
            if (response.data === 201) {
                toast(`New Device created: ${this.device.name}`, {
                    autoClose: 2000,
                    position: toast.POSITION.BOTTOM_RIGHT,
                    type: 'success'
                });
                this.$router.push({ path: "/" })
            }

            if (response.data === 204) {
                this.errorMessage = 'Device already exists'
            }
        }
    }
```

On the Python side, the "/create-device" API endpoint is defined using the "@app.put ('/create-device')" annotation, which specifies that this endpoint will use the HTTP PUT method. The method takes a "device" object as input, which contains the information provided by the user in

the form. This method then constructs the necessary payload for the NETCONF configuration of the new device and sends an HTTP PUT request to the main API server with the constructed URL. The response from the server is then returned as the HTTP status code to the front end.

```python
main.py > ...
@app.put("/create-device")
async def create_device(device: Device):
    global juniper_api_server
    global cisco_api_server

    global headers
    payload = json.dumps({
            "node": [
                {
                "node-id": device.name,
                "netconf-node-topology:port": device.port,
                "netconf-node-topology:reconnect-on-changed-schema": False,
                "netconf-node-topology:connection-timeout-millis": 20000,
                "netconf-node-topology:tcp-only": False,
                "netconf-node-topology:max-connection-attempts": 0,
                "netconf-node-topology:username": device.username,
                "netconf-node-topology:password": device.password,
                "netconf-node-topology:sleep-factor": 1.5,
                "netconf-node-topology:host": device.ip,
                "netconf-node-topology:between-attempts-timeout-millis": 2000,
                "netconf-node-topology:keepalive-delay": 120
                }
            ]
            })
    if device.deviceType=='juniper' :
        url = f"{juniper_api_server}rests/data/network-topology:network-topology/topology=topology-netconf/node={device.name}"
    else:
        url = f"{cisco_api_server}rests/data/network-topology:network-topology/topology=topology-netconf/node={device.name}"
    response = requests.request("PUT", url, headers=headers, data=payload)
    print(response.text)
    return response.status_code
```

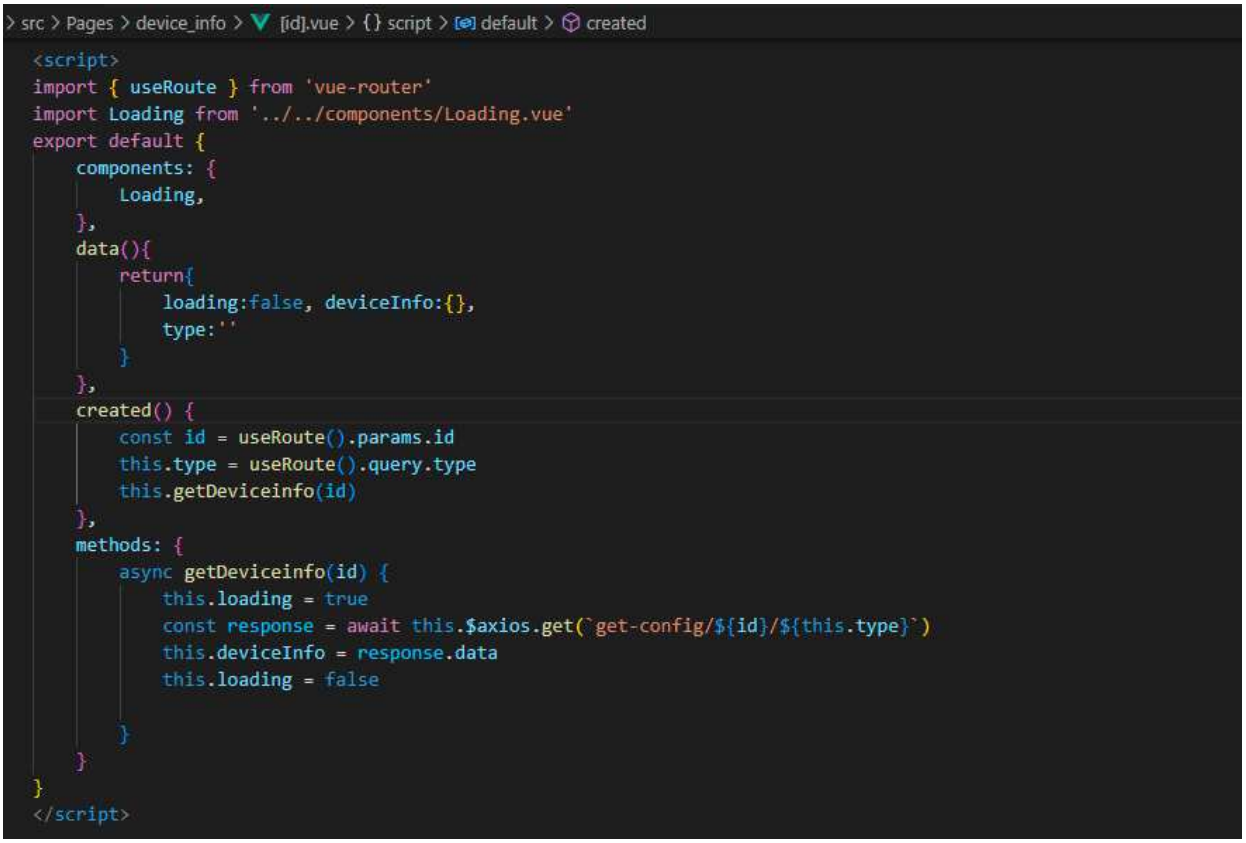


16.3 Show Config

The showConfig function is located in the Node.vue component is triggered when a user clicks on a device and selects the “Show Config” option from the context menu. It navigates the user to the device_info page with the node-id of the selected device and the type of the device's configuration specified in the URL parameters.

The getDeviceinfo function, which is defined in the device_info component, is responsible for making an HTTP request to the FastAPI backend to retrieve the configuration information for the specified device.

In this component, the useRoute function from the vue-router package is imported, along with a custom Loading component. The component's data function sets the initial state of loading and deviceInfo, and an empty type. The created function is called when the component is created and uses useRoute to extract the id and type parameters from the current route. The getDeviceInfo method is called with the id parameter and sends an HTTP GET request to the get-config endpoint in the FastAPI implementation, passing along the id and type as parameters. When the response is received, the loading state is set to false, and the deviceInfo state is updated with the data from the response.



In the FastAPI implementation, the get-config function is defined with two parameters, id and type. The function first declares two global variables, juniper_api_server, and cisco_api_server, which are presumably the URLs for the Juniper and Cisco devices, respectively. If the type parameter is 'juniper,' the getDeviceConfig function is called with the appropriate URL to retrieve the device configuration. Otherwise, the getDeviceConfig function is called with the appropriate Cisco URL. Finally, the function returns the device configuration obtained by the getDeviceConfig function.

```python
main.py > getDeviceConfig
@app.get("/get-config/{id}/{type}")
async def get_config(id:str, type:str):
    global juniper_api_server
    global cisco_api_server
    if type == 'juniper':
        config = getDeviceConfig(f'{juniper_api_server}restconf/operational/network-topology:network-topology/topology/topology-netconf/node/{id}/yang-ext:mount/')
    else:
        config = getDeviceConfig(f'{cisco_api_server}restconf/config/network-topology:network-topology/topology/topology-netconf/node/{id}/yang-ext:mount/Cisco-IOS-XE-native:native/')

    return config
```

The getDeviceConfig function takes a URL as a parameter, sends an HTTP GET request to the URL with authentication credentials, and returns the JSON data from the response.

```python
main.py > ...
def getDeviceConfig(url):
    response = requests.get(url, auth=('admin', 'admin'))
    response_json = response.json()
    return response_json
```

16.4 Change Hostname

By clicking on a device and selecting the "change hostname" option, the user can modify the hostname of that particular device. When the user selects this option, the Vue.js code calls the changeHostName function, which is located in the Node.vue component and pushes a new route to the Vue.js router. This new route includes the node-id of the device and the type of device.

```
src > components > Node.vue > {} script > default > methods
        changeHostName(item) {
            this.$router.push(`/host_name/${item['node-id']}?type=${this.data.type}`)
        },
```

This route leads to a page named "host_name" that includes a component named "[id].vue."

The component represents a selected device and includes an asynchronous function called updateHostName ().

The function first calls getHostName () to retrieve the current hostname from the device's configuration based on the device type. If the device type is "Juniper," the hostname is retrieved from the "junos-conf-root: configuration" path, and if not, it is retrieved from the "native" path.

UpdateHostName () then sends a PUT request to the "/update-config/{id}" endpoint using the $axios library with the device's updated configuration. If the response status code is between 200 and 300, indicating that the request was successful, a success message is displayed using the toast library. If the response status code falls outside this range, an error message is displayed.

```
> src > Pages > host_name > [id].vue > { } script setup > updateHostName

function getHostName() {
    if(type === 'juniper')
        return deviceInfo.value['junos-conf-root:configuration']['junos-conf-system:system']['host-name']
    else
        return deviceInfo.value.native.hostname
}

async function updateHostName() {
    const newName = getHostName()
    const response = await $axios.put(`/update-config/${id}/${type}`, deviceInfo.value)
    if(response.data >= 200 && response.data <= 300){

        toast(`Hostname changed to: ${newName}`, {
            autoClose: 2000,
            position: toast.POSITION.BOTTOM_RIGHT,
            type: 'success'
        });
    }
```

On the FastAPI backend, the "@app.put" decorator is used to define the "/update-config/{id}/{type}" endpoint. This endpoint is responsible for updating the device's configuration with the new hostname and sends a PUT request to the RESTCONF API endpoint using the "requests" library. The endpoint includes a function named "update_config ()," which accepts three arguments: "id," "type," and "device."

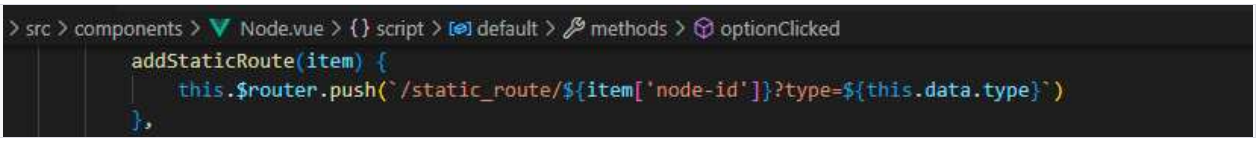
The function sends a PUT request to the RESTCONF API endpoint based on the device type and updates the device's configuration with the new hostname. If the device type is Juniper, the function builds a URL for the "junos-conf-root: configuration" path. Otherwise, the function builds a URL for the "native" path. The function then sends a PUT request to the RESTCONF API endpoint using the built URL, the headers, and the device configuration in JSON format.

Finally, the function returns the response status code to indicate whether or not the operation was successful.

```
main.py > ...

@app.put("/update-config/{id}/{type}")
async def update_config(id:str, type:str, device: Dict[Any, Any]):
    global juniper_api_server
    global cisco_api_server
    global headers
    if type == 'juniper':
        url = f'{juniper_api_server}restconf/config/network-topology:network-topology/topology/topology-netconf/node/{id}/yang-ext:mount/junos-conf-root:configuration/'
    else:
        url = f'{cisco_api_server}restconf/config/network-topology:network-topology/topology/topology-netconf/node/{id}/yang-ext:mount/Cisco-IOS-XE-native:native/'

    response = requests.request("PUT", url, headers=headers, json=device)
    print (response.text)
    return response.status_code
```

16.5 Add Static Route

To add a static route function addStaticRoute (item) is implemented. It is called when a user clicks on a button to add a static route. It uses Vue Router to navigate to the static route creation page with the node ID and route type as query parameters. This function is located in the Node.vue component.



The next function is addStaticRoute (), which is called when a user submits the form to create a static route. This function first checks the route type and creates a route object accordingly. It then updates the device information with the new route and sends a PUT request to the FastAPI backend to update the device configuration. If the PUT request is successful, a success message is displayed using the toast library. Otherwise, an error message is displayed.

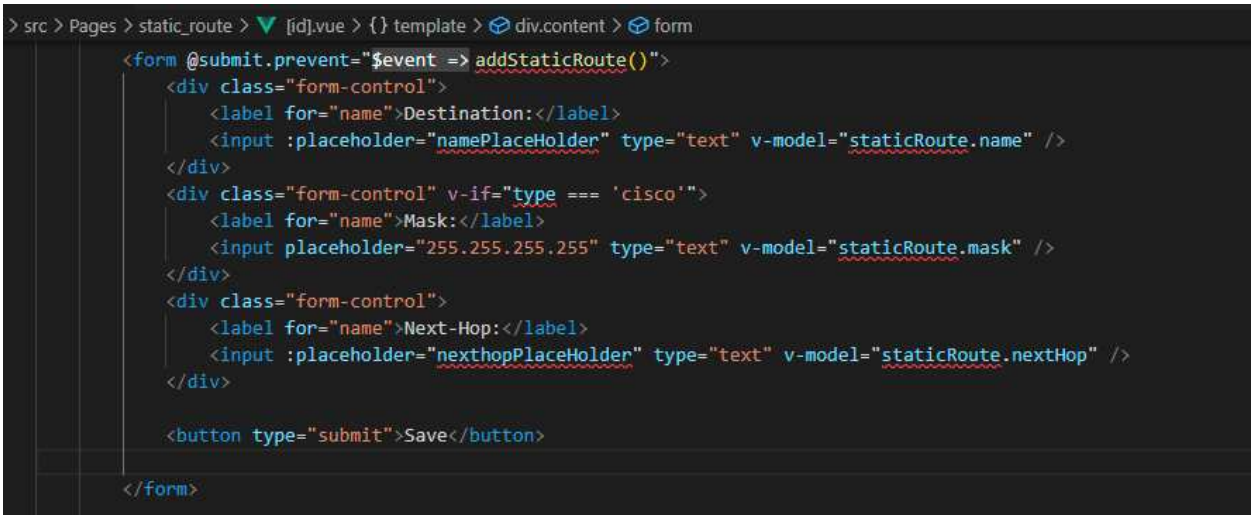
```
src > Pages > static_route > [id].vue > {} script setup > addStaticRoute > type
async function addStaticRoute() {
    if (type === 'juniper') {
        const route: Route = {
            name: staticRoute.value.name,
            "next-hop": [staticRoute.value.nextHop]
        }
        deviceInfo.value['junos-conf-root:configuration']['junos-conf-routing-options:routing-options'].static.route.push(route)
    }
    else {

        const route = {
            "prefix": staticRoute.value.name,
            "mask": staticRoute.value.mask,
            "fwd-list": [
                {
                    "fwd": staticRoute.value.nextHop
                }
            ]
        }

        deviceInfo.value['native']['ip']['route']['ip-route-interface-forwarding-list'].push(route)
    }

    const response = await $axios.put(`/update-config/${id}/${type}`, deviceInfo.value)
    if (response.data >= 200 && response.data <= 300) {
        toast(`new static route was added successfully`, {
            autoClose: 2000,
            position: toast.POSITION.BOTTOM_RIGHT,
            type: 'success'
        });

        return
    }
    toast(`Error in add static route`, {
        autoClose: 2000,
        position: toast.POSITION.BOTTOM_RIGHT,
        type: 'error'
    });
}
```

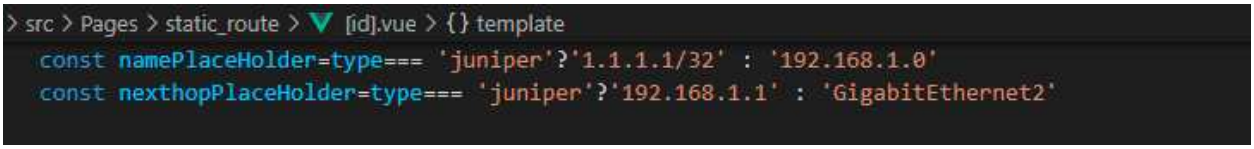
The Python FastAPI section starts with the decorator @app.put ("/update-config/ {id}/ {type}"). This decorator maps the HTTP PUT method to the /update-config endpoint with two parameters: id and type. The update_config () function is called when this endpoint is hit. It receives the device configuration as a dictionary object in the device parameter. Depending on the device type, it constructs the URL to update the configuration and sends a PUT request to the device's API server. The API server's response is returned to the client.

Note: The following HTML code, which is located in the static_route component, creates a form that allows users to add a static route to a network device. When the user submits the form, the @submit.prevent="addStaticRoute ()" directive invokes the addStaticRoute () function to handle the form submission.

The form contains three input fields: Destination, Mask (only displayed if the type is "cisco"), and Next-Hop. The placeholder attribute provides the default text for each input field.

The namePlaceHolder and nexthopPlaceHolder variables determine the default values of the input fields, depending on the value of type. If the type is "juniper," namePlaceHolder is set to "1.1.1.1/32," and nexthopPlaceHolder is set to "192.168.1.1". If the type is "cisco," namePlaceHolder is set to "192.168.1.0," and nexthopPlaceHolder is set to "GigabitEthernet2". These values are used as the default text in the input fields.





16.6 Add Interface

For adding an interface to a device, different vue components are deployed.

This section of code defines the add_interface component, which is responsible for displaying the appropriate interface configuration form based on the type of device being configured. The component makes an HTTP GET request to retrieve the device information and the getDeviceInfo () function.

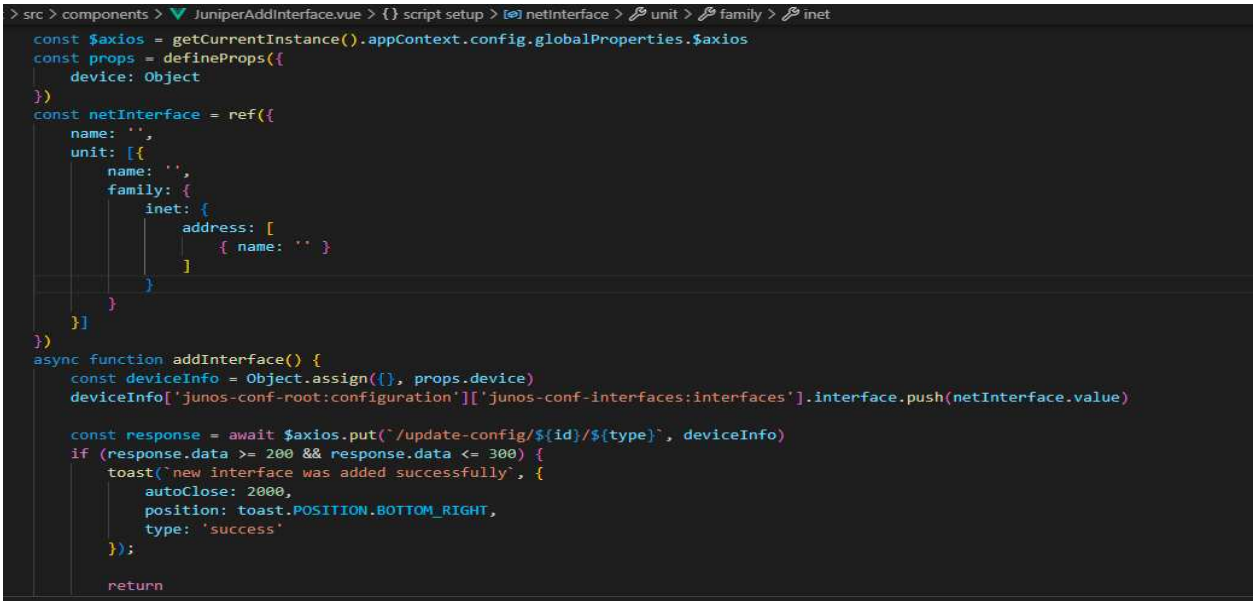
The component uses an isLoading reactive variable to display a loading spinner while the data is being retrieved and sets deviceInfo to the retrieved data once the request completes.

The template of the add_interface component includes the Loading component to display the spinner and conditionally renders either the JuniperAddInterface or CiscoAddInterface components based on the value of type. deviceInfo is passed as a prop to the appropriate interface configuration component.

The juniperAddInterface component is responsible for adding a new interface to a Juniper device. It uses the axios library to make an HTTP PUT request to update the configuration of the device. The component receives a device object as a prop, and the netInterface object holds the data entered by the user in the form.

The addInterface function is called when the user submits the form, and it creates a copy of the device object to avoid modifying the prop directly. It then adds the new network interface to the copy of the device object and sends a PUT request to update the configuration. If the request is successful, a success message is displayed using the toast library. If an error occurs, an error message is displayed.

The template of the juniperAddInterface component displays a form that allows the user to enter the details of the new interface, such as name, unit, and IP address. When the user submits the form, the addInterface function is called to add the interface to the device configuration.



The CiscoAddInterface component is responsible for adding a new interface to a Cisco device. It imports the Axios library for making HTTP requests and defines a props object that expects a device object to be passed in as a prop. The component also defines a netInterface object using the ref function, which holds the values of the new interface to be added. "Refs" is part of vue.js to create reactive objects and enable two-way binding functionality between UI and JavaScript code.

The addInterface function is called when the form is submitted, which creates a copy of the device object passed in as a prop and adds the new interface to the native.interface.GigabitEthernet array using the push method. Then, it sends a PUT request to the server with the updated deviceInfo object.

If the response status code is between 200 and 300, a success message is displayed using the toast library. Otherwise, an error message is displayed. The form allows the user to input the GigabitEthernet number, IP address, and subnet mask of the new interface.

src > components > CiscoAddInterface.vue > {} script setup > netInterface > ip

```
async function addInterface() {
    const deviceInfo = Object.assign({}, props.device)
    deviceInfo.native.interface.GigabitEthernet.push(netInterface.value)

    const response = await $axios.put(`/update-config/${id}/${type}`, deviceInfo)
    if (response.data >= 200 && response.data <= 300) {
        toast(`new interface was added successfully`, {
            autoClose: 2000,
            position: toast.POSITION.BOTTOM_RIGHT,
            type: 'success'
        });

        return
    }
    toast(`Error in add network interface`, {
        autoClose: 2000,
        position: toast.POSITION.BOTTOM_RIGHT,
        type: 'error'
    });
}
```

In the FastAPI endpoint that handles PUT requests to update the configuration of a device, the function takes in three parameters: the device ID, the type of device (either "juniper" or "cisco"), and the new configuration for the device as a dictionary.

The function starts by declaring global variables for the URLs of the Juniper and Cisco API servers, as well as the HTTP headers to use in the request.

It then checks the type of device and constructs the appropriate URL for the API call. If the device is a Juniper device, it uses the Juniper API server and constructs a URL to access the configuration data for the specified device ID. If the device is a Cisco device, it uses the Cisco API server and constructs a URL to access the "native" configuration data for the specified device ID. Finally, the function makes a PUT request to the specified URL using the provided device configuration data and HTTP headers. It prints the response text to the console and returns the status code of the response.

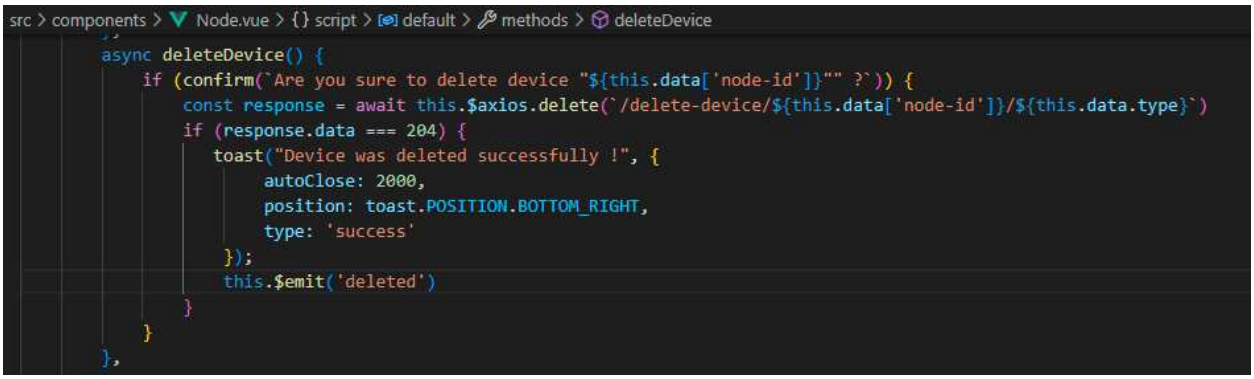
main.py > delete_device

```
@app.put("/update-config/{id}/{type}")
async def update_config(id:str, type:str, device: Dict[Any, Any]):
    global juniper_api_server
    global cisco_api_server
    global headers
    if type == 'juniper':
        url = f'{juniper_api_server}restconf/config/network-topology:network-topology/topology/topology-netconf/node/{id}/yang-ext:mount/junos-conf-root:configuration/'
    else:
        url = f'{cisco_api_server}restconf/config/network-topology:network-topology/topology/topology-netconf/node/{id}/yang-ext:mount/Cisco-IOS-XE-native:native/'

    response = requests.request("PUT", url, headers=headers, json=device)
    print (response.text)
    return response.status_code
```

16.7 Delete Node

When a user wants to delete a device, the deleteDevice () function sends a DELETE request to the FastAPI endpoint /delete-device/{id}, where the id parameter is the unique identifier of the device. If the response code from the endpoint is 204, the function uses the Vue.js library Toast to display

a success message to the user for 2 seconds at the bottom right corner of the screen. The deleteDevice () function is implemented within the Node.vue component.



On the FastAPI side, there is an endpoint with a DELETE HTTP method called "delete_device ."This endpoint can be accessed at the route "/delete-device/ {id}/{type}," where the "id" and "type" parameters are extracted from the URL.

The function first imports the global variables juniper_api_server, cisco_api_server, and headers. The "id" parameter represents the unique identifier of the device to be deleted, and the "type" parameter represents the type of device, either Juniper or Cisco.

The function uses an f-string to build the URL to the corresponding REST API endpoint to delete the device based on the "type" parameter. Then, the requests library is used to send a DELETE request to the built URL with the headers specified in the "headers" variable. Finally, the function returns the response status code from the REST API call as the result of the FastAPI endpoint.

17 Deploy a FastAPI App on Ubuntu

The following general steps can be followed to deploy a FastAPI application on Ubuntu 22.04 as a service:

1. Install dependencies: Make sure that you have Python 3.7 or higher and pip installed. You can install them using the following commands:

- sudo apt-get update
- sudo apt-get install python3 python3-pip –y

2. To verify the installation and Python 3 version, perform the following:

- python3 –version

3. Install FastAPI: You can install FastAPI and its dependencies using the following command:

- pip3 install fastapi Uvicorn

4. Navigate to your App directory and make a virtual environment for the app:

- cd /home/fastapi/api
- python3.10 -m venv .

5. Activate your FastAPI app:

- source ./bin/activate

6. Create a systemd service file: You need to create a systemd service file to run the FastAPI app as a service. You can create a file named "fastapi.service" in the /etc/systemd/system directory with the following content:

```
[Unit]
Description=FastAPI Service
After=network.target

[Service]
User=fastapi
WorkingDirectory=/home/fastapi/api
Environment=/home/fastapi/api/env/bin
ExecStart=/usr/bin/uvicorn main:app --host=10.3.31.14 --port=8000
Restart=always

[Install]
WantedBy=multi-user.target
```

7. Reload systemd and start the service: You need to reload systemd to pick up the new service file and then start the service using the following commands:

- Sudo systemctl daemon-reload
- Sudo systemctl start fastAPI

8. You can check the status of the service using the following command:

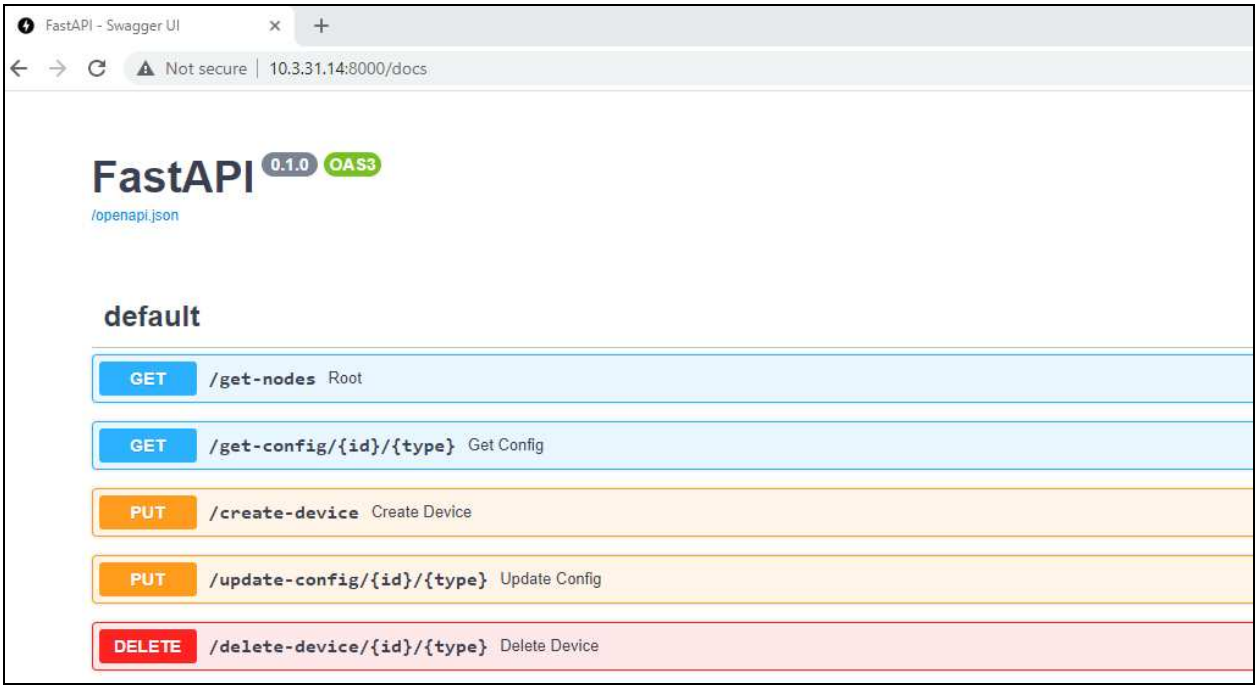
- Sudo systemctl status fastAPI

```
root@Opendaylight-GUI:/home/fastapi# systemctl status fastapi
● fastapi.service - FastAPI Service
     Loaded: loaded (/etc/systemd/system/fastapi.service; enabled; vendor preset: enabled)
     Active: active (running) since Mon 2023-03-06 23:38:13 UTC; 44s ago
   Main PID: 941 (uvicorn)
      Tasks: 5 (limit: 2193)
     Memory: 39.7M
        CPU: 436ms
     CGroup: /system.slice/fastapi.service
             └─941 /usr/bin/python3 /usr/bin/uvicorn main:app --host=10.3.31.14 --port=8000

Mar 06 23:38:13 Opendaylight-GUI systemd[1]: Started FastAPI Service.
Mar 06 23:38:14 Opendaylight-GUI uvicorn[941]: INFO:     Started server process [941]
Mar 06 23:38:14 Opendaylight-GUI uvicorn[941]: INFO:     Waiting for application startup.
Mar 06 23:38:14 Opendaylight-GUI uvicorn[941]: INFO:     Application startup complete.
Mar 06 23:38:14 Opendaylight-GUI uvicorn[941]: INFO:     Uvicorn running on http://10.3.31.14:8000 (Press CTRL+C to quit)
```

9. Enable the service: You can enable the service to start automatically at boot time using the following command:

- Sudo systemctl enable fastAPI

The FastAPI app should now be running as a service on Ubuntu 22.04. You can access it by visiting the server's IP address and port 8000 in a web browser.



18 Deploying a Vue.js app in Nginx

Here is a detailed step-by-step guide on how to do it:

1. Build your Vue.js application. First, you need to build your Vue.js application. Run the following command in your terminal:

Note: Before executing the build command, it is necessary to modify the IP address specified in the .env.production file to the server's IP address.

- npm run build

This will create a dist directory in your project root folder containing all the necessary files for deployment.

2. Configure Nginx

You need to create an Nginx server block for your Vue.js application. Open your Nginx configuration file (/etc/nginx/sites-available/default) and add the following server block:

```
server {
        listen 80;
#       listen [::]:80;
#
        server_name 10.3.31.14;
#
        root /var/www/opendaylight/dist;
        index index.html;
#
        location / {
                try_files $uri $uri/ =404;
        }
}
```

3. Restart Nginx

After modifying the Nginx configuration file, you need to restart Nginx to apply the changes:

- Sudo systemctl restart nginx

4. Verify the deployment

If you open a browser and go to the address http://10.3.31.14, you should see the Vue.js application up and running!


Daylight UI
Not secure
10.3.31.14

19 References

[1] https://www.vmware.com/topics/glossary/content/software-defined-networking.html

[2] https://searchsdn.techtarget.com/definition/software-defined-networking-SDN

[3] https://www.strongdm.com/blog/software-defined-networking

[4] https://www.techtarget.com/searchnetworking/tip/Centralized-vs-decentralized-SDN-architecture-Which-works-for-you

[5] https://ipcisco.com/lesson/sdn-architecture-components/

[6] https://www.ibm.com/topics/sdn?mhsrc=ibmsearch_a&mhq=sdn

[7] https://www.opendaylight.org/about/platform-overview#:~:text=OpenDaylight%20(ODL)%20is%20a%20modular,clear%20focus%20on%20network%20programmability.

[8] https://opendaylight-documentation.readthedocs.io/en/stable/getting-started-guide/karaf_features.html

[9] https://docs.opendaylight.org/projects/controller/en/latest/dev-guide.html

[10] https://docs.opendaylight.org/en/stable-oxygen/user-guide/netconf-user-guide.html

[11] https://john.soban.ski/install-opendaylight-ubuntu-lts-22-04.html

[12] https://v3.vuejs.org/guide/introduction.html

[13] https://vitejs.dev/

[14] https://fastapi.tiangolo.com/