

DANMARKS TEKNISKE UNIVERSITET



Authentication Service

34359 SDN: SOFTWARE-DEFINED NETWORK

Gea Staropoli (s243967)

Edoardo Santucci (s243100)

The project was completed with a fair
division of work between both
members.

January 9, 2025

Contents

1	Problem Statement	1
2	Project Design	1
2.1	Project Description	1
2.2	Involved Entities and Network Relations	1
2.2.1	Entities	1
2.2.2	Network Relations	3
2.3	Service Behavior	4
3	Project Implementation	6
3.1	Firewall	6
3.2	Webapp	7
3.2.1	Creation and Configuration	7
3.2.2	Web Interface: Index	7
3.2.3	Database and Javascript Server: Lancia	8
3.2.4	Flowrule Implementation: Push Rule	9
4	Project Testing Strategy and Results Analysis	11
4.1	Starting ONOS and Network Topology	11
4.2	Running the Firewall Service	13
4.3	Running the WebApp and Authentication	14
5	Critical Conclusions	17
	References	18

1 Problem Statement

The objective of this project was to develop an authentication service for a flexible network. Essentially, our system ensures that each host logs in with a fixed password associated with that specific host before being permitted to communicate with others in the network. Initially, all hosts in the network should be disabled. To authenticate, a host must access a web interface and submit its credentials. The web interface will forward the credentials to a web application that includes a server for their verification. If the credentials are invalid, the server will display an error message. If the credentials are correct, the server will send a REST call to the controller, instructing it to enable network access for the host. Once the process is complete, the web application will send a message through the web interface with the operation's results.

2 Project Design

2.1 Project Description

First, we assume that a random network topology is run. The implementation begins by initializing a firewall that blocks all communication between hosts by default. A web application then manages user login requests, verifying credentials in a database of valid pairs via a server. If the credentials are incorrect, an error message is displayed on the web interface. If the authentication succeeds, the application ensures that the blocking firewall rules for the authenticated host are removed. This grants the host access to the network, enabling communication with other devices.

2.2 Involved Entities and Network Relations

2.2.1 Entities

ONOS and REST API The Open Network Operating System (ONOS) [1] is an open-source SDN controller designed to implement next-generation SDN (Software-Defined Networking) solutions. Its primary objective is to enable real-time network configuration and control, eliminating the need to run traditional routing and switching protocols within the network infrastructure itself. In this project, ONOS was used as the controller to dynamically manage and configure network devices using OpenFlow 13 as a Southbound protocol. Furthermore, ONOS provided a Graphical User Interface (GUI) which offered a real-time visualization of the network topology.

This interface is accessible at <http://localhost:8181/onos/ui/login.html>, where a login page as in Figure 1 is presented and one can log in using the default credentials:

username: **onos**
password: **rocks**



Figure 1: ONOS Login Webpage

A core feature of ONOS is its Northbound API, the REST API, which enables applications, such as our Authentication Service, to interact with the network using high-level constructs. Specifically, it enabled interaction with the network using standard HTTP operations like GET, POST, and DELETE. For example, the GET `/devices` command retrieved information about all switches on the network, while GET `/hosts` fetched data on connected hosts [2]. In the same way it also enabled the dynamic update of flow rules for every switch in the network. The REST API is accessible at <http://localhost:8181/onos/v1/docs/>

Hosts and Switches This project utilized simulated network devices, specifically hosts and switches, within a custom network topology created using Python and deployed via Mininet. The switches are OpenFlow-enabled and communicate with the ONOS controller using the OpenFlow protocol (version 13) through the southbound interface.

Firewall The Firewall Service was designed as a standalone application with the primary purpose of automatically blocking all connections between hosts in the network. Upon execution, the service enforces a state where no host is allowed to send or receive data packets. This effectively creates a starting point in which all hosts are disconnected from the network.

Web App The web application works as a bridge between the user and the SDN controller by providing a user friendly interface. Technically, it is a software application that runs on a web server and is accessed via a web browser. Acting as a client platform, the web application communicates with the ONOS REST API to manage and control network functionalities. The web application is built using technologies such as HTTP Servlets, JSP (JavaServer Pages), and HTML pages:

- HTTP Servlets are Java classes that run on the web server to process HTTP requests (e.g., GET, POST). They serve as a middle layer between the user's browser and backend logic. When a user makes a request, the appropriate method in the servlet (e.g., doGet() or doPost()) is automatically executed to handle the request.
- JSP (JavaServer Pages) enable the creation of dynamic web pages by combining HTML tags, Java code, and other scripting elements. When the web app starts, the server converts the JSP file into a servlet, which then generates HTML content to send back to the browser for display.
- HTML Pages are static documents written in HyperText Markup Language (HTML). They define the structure and design of web pages using a series of tags and elements that instruct the browser on how to display the content.

Moreover, in the web app, we used Node.js [3], a cross-platform JavaScript runtime environment that allows JavaScript code to run outside of a web browser. In our case, it is used to create a server written in JavaScript, which queries the database to verify the correspondence between the provided user and password pair and the records stored in the database.

2.2.2 Network Relations

Functional Description The web application, through the web server, interacts with the ONOS REST API by sending HTTP requests (e.g., GET, POST). Firstly the user interacts with the web application via a browser. Then, the web server, sends REST API requests to ONOS, which next processes these requests and returns data or action confirmations. And finally, the web application displays the results to the user through the browser. Those steps just described are shown in Figure 2. Both ONOS and the web server are running on the same machine. The web server handles user interactions, while ONOS manages the network through its REST API [4]. Additionally, another local server has been created just to access the database and retrieve the needed informations.



Figure 2: Functional description of the web app

Topological Description We basically created 2 apps:

1. The firewall app (*Firewall_final*): blocks all connections in the network.

2. The web app (*SDNAuth_final*): provides a web interface, verifies the compatibility of the inserted host password with the ones in the database, then issues commands to ONOS to disable specific firewall rules, and displays the final message to the user.

Both the applications behave as a REST client and communicate with ONOS through its REST API, therefore those doesn't have any oar file in it, because they don't need to be directly installed in ONOS.

Additionally, we also created a Python script (*Topology.py*) to generate the network topology, which was used to demonstrate the functionality of the application, but as said the mechanism will work with every kind of topology, given that the host and its own password are present in the database.

2.3 Service Behavior

Use Cases The system is designed to handle the following scenarios:

1. **Initial State:** Initially, all hosts are disconnected from the network. This means that if any two hosts attempt to communicate, the firewall will block the communication. This behavior is enforced by flow rules installed by the Firewall module on all switches connected to the network's hosts.
2. **Invalid Credentials:** When a host attempts to authenticate with incorrect credentials, access to the network is denied. The login page will display an "Invalid Credentials" message, and the host will remain blocked from sending or receiving packets.
3. **Successful Authentication:** Finally, when a host successfully authenticates using valid credentials, access to the network is granted for that host. The host can then send and receive packets on the network. However, if this host is the only one connected to the network, communication will still not occur as no other host is available to exchange packets. In this specific case, while the host is technically authorized to use the network, it will not experience meaningful connectivity due to the lack of other active endpoints.

Chronogram of Relevant Message Exchanges The sequence of interactions in the system is described as follows:

1. **Firewall Initialization:**

The Firewall service retrieves the complete topology information (e.g., devices, hosts, and links) from the ONOS controller using **GET** requests to the REST API. Using this information, it installs flow rules on all relevant switches to block communication between every pair of hosts in the network.

2. **Initial Packet Transmission:**

A host attempts to send a packet on the network. Since all communication is blocked by default, the Firewall drops the packet, preventing it from being delivered.

3. Host Authentication Attempt:

The host accesses the login page provided by the WebApp and submits its credentials (username and password) for authentication.

4. Credential Validation:

The WebApp validates the submitted credentials by comparing them with the entries in a preconfigured database. If the credentials are correct, the WebApp receives the host username from the login page and sends a request to the ONOS controller via the REST API to remove the flow rule blocking communication for the authenticated host.

5. Successful Packet Transmission:

The authenticated host sends another packet, which is now successfully forwarded through the network. This is possible only if the destination host has also logged into the network and had its blocking flow rule removed.

Expected Impact The implementation of this authentication service as an application for an SDN-based network introduces several significant impacts:

- **Security Enhancement:**

By default, the network enforces a zero-trust policy where all host-to-host communication is blocked until authentication occurs. This reduces the risk of unauthorized access and ensures that only verified hosts can interact on the network.

- **Dynamic Network Behavior:**

The integration with the ONOS controller and its REST API enables real-time adjustments to the network configuration. The system dynamically removes or enforces blocking rules based on the authentication status of hosts, allowing a flexible and responsive network environment tailored to user behavior.

- **Scalability and Flexibility:**

The architecture, leveraging ONOS, REST API, and flow rule management, is topology-agnostic. This means the solution is scalable to larger networks and adaptable to diverse topologies without requiring changes to the underlying mechanism.

3 Project Implementation

3.1 Firewall

The Firewall Service is implemented as a standalone Java application (*Firewall_final*) that leverages the ONOS REST API to block all communication between hosts in a network. The relevant part of the code is found in the main class, called *MC2*, contained in the *src* folder. In the first part, a helper class *Authenticator* is used to initialize a REST client (using *onos*, *rocks* as username and password) to interact with the ONOS controller. It retrieves network information using **GET** operations to various ONOS REST API endpoints. The JSON responses are properly stored thanks to the POJOs created for each endpoint, and then displayed. Key endpoints include:

- `/onos/v1/topology` provides an overview of the network, including the number of clusters, devices, and links.
- `/onos/v1/devices` retrieves detailed information about all network devices. Among these, only devices of type "SWITCH" are displayed, and their IDs are stored in a list.
- `/onos/v1/hosts` gathers data about the connected hosts. The IP address of each host is then collected and mapped to the switch they are connected to, which is found by exploring the `elementID` stored in the `location` field included in the JSON response. This mapping is essential to ensure that flow rules are applied to the correct switch for each host.
- `/onos/v1/links` gather link details which, while not directly used in rule installation, contribute to a comprehensive understanding of the network's structure.

After collecting the necessary data, the application dynamically installs flow rules in every switch connected to a host in order to block all host-to-host communication. For every pair of source and destination hosts, a flow rule is constructed that matches packets based on their source and destination IP addresses. A **Selector** object specifies the match criteria, such as the Ethernet type (0x800 for IPv4 packets), the source IP and the destination IP. A **Treatment** object specifies the actions to be taken on packets matching the Selector. In this implementation, the default treatment is a "drop" action (empty field). Finally, a **FlowObjective** object encapsulates the rule details, including priority, timeout, device ID of the target switch (which change to iterate every switch in the network), and the operation type. In our specific case, a timeout of 300 seconds is used, and the operation type is **ADD**, since we are generating new rules. Furthermore, the device ID had to be processed to match the format required by the ONOS REST API (the colon in the switch ID must be replaced with %3A). These rules are then sent to the ONOS controller using **POST** operations to the `/flowobjectives/{switchId}/forward` endpoint. The application iterates through all possible host pairs (using a for loop), installing flow rules that block communication between any possible pair of hosts.

3.2 Webapp

3.2.1 Creation and Configuration

To create the web application, we started by creating a new project in IntelliJ, specifying Java as the programming language and including Servlet in the project settings. Then the project was automatically generated with this structure:

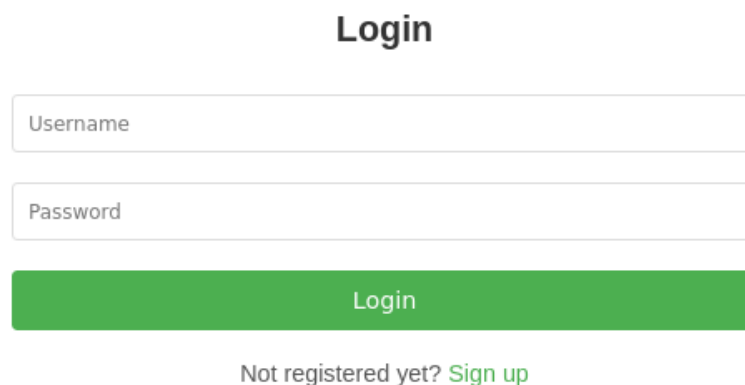
- A *src* folder for Java classes, such as HTTP servlets.
- A *web* folder to house the web app components, including JSP and HTML pages.
- A *web.xml* file for configuring the application.

In the web folder, we added an *index.jsp* file containing the HTML code for the main page. We then created a servlet named *pushRule* and registered it in *web.xml* to handle specific backend logic.

After that, we configured the hosting server, using GlassFish V4 for deployment. Finally, we deployed the application as WAR exploded, making it ready for testing and use. It is important to underline that ONOS REST API and GlassFish are serving both on purpose on different ports, the first one on 8090 and the second one on 8181.

3.2.2 Web Interface: Index

The *index.jsp* file contains CSS to enhance the look of the web page, along with the HTML needed to create the user interface shown in Figure 3. The interface includes a form with two input fields: one for the username, corresponding to the host name (e.g., h1), and another for the password.



Login

Username

Password

Login

Not registered yet? [Sign up](#)

Figure 3: Login page

When the user fills in these fields and clicks the submit button, a JavaScript code prevents the default form submission behavior. Instead, it captures the input values and sends them to the local server (*Lancia.js*) for verification. If the credentials are incorrect, the server returns an error message indicating an invalid username or password, and the page reloads with the input fields reset for the user to try again. In contrast, if the credentials are correct, a "Login successful" message is displayed, and the username is sent to the servlet *PushRule*. After executing *PushRule*, the web interface reloads, presenting the login page again.

3.2.3 Database and Javascript Server: Lancia

For the database containing the information about the password and the username we created a database using PostgreSQL 16 [5], an open source object-relational database. We created the database with the following command:

```
1 CREATE DATABASE sql_proj;
```

To create the specific table, we executed:

```
1 CREATE TABLE users (  
2 us VARCHAR(256) PRIMARY KEY,  
3 psw VARCHAR(256),  
4 conn BOOLEAN  
5 );
```

Where varchar (256) specifies that the field can contain a variable number of characters, up to 256, PRIMARY KEY ensures that each value in this field is unique. and BOOLEAN Indicates that the field can only store a boolean value (true or false). Finally, to insert our desired values into the table, we used the following code:

```
1 INSERT INTO users (us, psw, conn)  
2 VALUES ('h1', 'h1', false);
```

In the example above, we created an instance of the users table with the values us set to 'h1', psw set to 'h1', and conn set to false. The us field represents the username, which corresponds to the name of the host accessing the system. The psw field contains the password associated with that username. Finally, the conn field is a boolean value indicating the user's connection status, where true means the user is connected, and false means he is not.

To access the database, we enabled the Database Tools and SQL plugin [6] already available in IntelliJ IDEA. This allowed us to maintain a persistent connection to the database without having to reestablish access each time. We then created a server that listens on port 3000. This server processes POST requests sent from the index.jsp file, which includes the username and password entered by the user. The server verifies the credentials by executing the following query:

```
1 SELECT us FROM users WHERE us = $1 AND psw = $2
```

In this query, \$1 and \$2 represent the username and password provided in the POST request. If a match is found, the server updates the conn field for the corresponding user from false to true using the following query:

```
1 UPDATE users SET conn = true WHERE us = $1
```

Based on the result, the server responds with a message indicating the outcome. If the credentials are valid, it returns "Login successful." If the username or password doesn't match any record in the database, the response is "Invalid username or password." In the case of an unexpected issue, the server returns "Server error."

The mechanism of the login is explained in Figure 4.

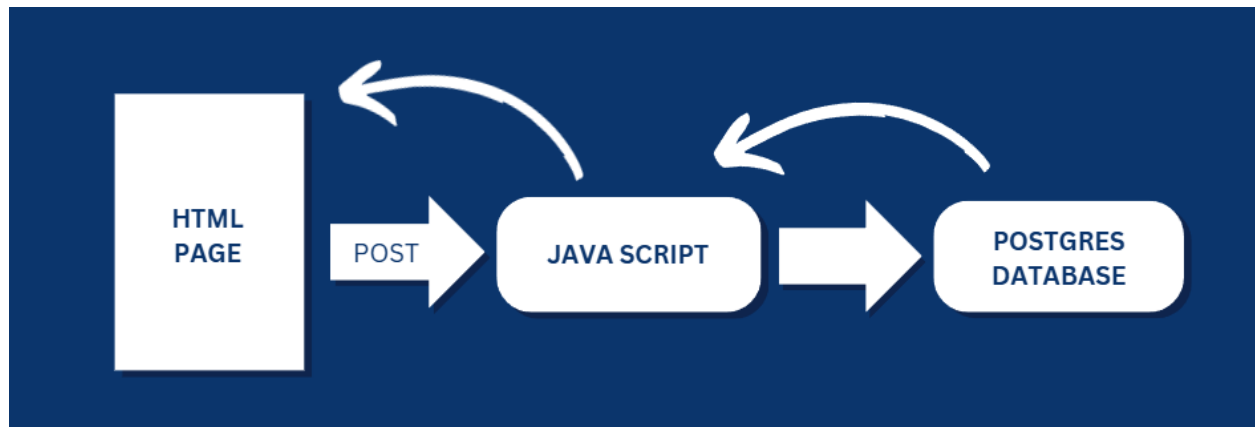


Figure 4: Functional description of the login

3.2.4 Flowrule Implementation: Push Rule

The "brain" of the web application is an HTTP servlet called *PushRule*, located in the core package of the `src` folder. This servlet is responsible for enabling network connectivity for authenticated hosts. It is triggered by a `GET` request containing a `username` parameter, which corresponds to the identifier of the authenticated host (e.g., `h1`, `h2`, etc.). The primary objective is to determine the host's network location (i.e., the switch to which the host is connected) and reconfigure the network by removing firewall rules from that switch. Similar to the *Firewall* implementation, this servlet interacts with ONOS to retrieve network information. Specifically, it sends a `GET` request to the ONOS endpoint `/onos/v1/hosts` to obtain a list of all hosts in the network, along with their respective switches and IP addresses. The servlet then processes the retrieved data to construct critical mappings:

1. **Host-to-Switch Mapping** (`hostIpToSwitchMap`): Each host's IP address is mapped to its `elementId`, which represents the `deviceId` of the switch is connected to.
2. **Username-to-IP Mapping** (`hostNameToIpMap`): A User-friendly host name is created for every host in the network and mapped to respective IP address.

For example:

A host with IP address `10.0.0.1` connected to switch with ID of `:00000000000000002` will have:

- `hostIpToSwitchMap.put("10.0.0.1", "of:00000000000000002")`
- `hostNameToIpMap.put("h1", "10.0.0.1")`

Using the username provided in the GET request, which eventually corresponds to a host name, the servlet identifies the corresponding host IP address from the `hostNameToIpMap` and retrieves the associated switch ID (`elementId`) using the `HostIpToSwitchMap`. If the username does not match any known host, the servlet logs an error and terminates its execution. Once identified, the switch ID is formatted by replacing `:` with `%3A`, consistent with the ONOS requirements previously mentioned. After identifying the authenticated host's IP address (stored in `inputHost`) and its corresponding switch (stored in `targetSwitch`), the servlet reactivates the connection for the authenticated host by modifying the configuration of the switch to which the host is connected. This process follows a logic similar to the *Firewall Service*, but instead of creating new flow rules, the program removes the existing rules that block connectivity between the authenticated host and other hosts in the network. The rules for other hosts remain unchanged. The servlet iterates through the network, targeting the connection between the authenticated host (`inputHost`) and each other host. A POST request is sent to the ONOS endpoint `/flowobjectives/switchId/forward`, specifically addressing the switch the authenticated host is connected to. To remove the rules, all the details of the `FlowObjectives` must match the previously installed rules, except the operation type, which is now set to `REMOVE` to delete the existing rules.

Once all the relevant flow rules are successfully removed from the target switch, the servlet forwards the user to a new login page, allowing them to input another username and password for authentication.

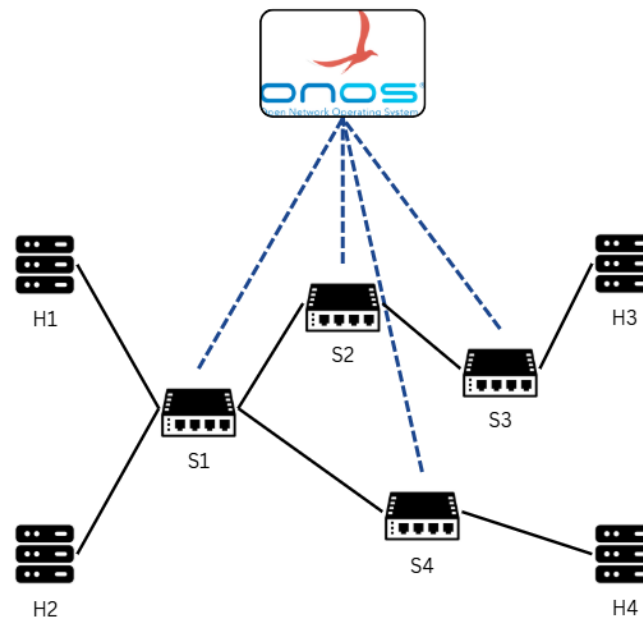


Figure 5: Topology used for testing

4 Project Testing Strategy and Results Analysis

4.1 Starting ONOS and Network Topology

Firstly, we started by activating ONOS. This involves opening a terminal inside the repository where ONOS is installed and executing the following command:

```
bazel run onos-local -- clean
```

Subsequently, in another terminal, we run:

```
onos localhost
```

Once ONOS is successfully started, we move on to setting up the topology. The topology used for this project was designed arbitrarily to demonstrate the functionality of the system and is shown in Figure 5. However, the service is compatible with any topology. The specific topology was implemented in a Python file named `Topology.py` using the Mininet network emulator. To initiate the topology, we navigated to the directory containing the Python file and executed the following command in the terminal:

```
sudo python Topology.py
```

After entering the password, if everything proceeds correctly, the output of the command will be as follows:

```
*** Creating network
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s1 s2 s3 s4
*** Adding links:
(100.00Mbit) (100.00Mbit) (s1, h1) (100.00Mbit) (100.00Mbit) (s1, h2)
(100.00Mbit) (100.00Mbit) (s1, s2) (100.00Mbit) (100.00Mbit) (s1, s4)
(100.00Mbit) (100.00Mbit) (s2, s3) (100.00Mbit) (100.00Mbit) (s3, h3)
(100.00Mbit) (100.00Mbit) (s4, h4)
*** Configuring hosts
h1 h2 h3 h4
Unable to contact the remote controller at 127.0.0.1:6633
*** Starting controller
ONOSController
*** Starting 4 switches
s1 s2 s3 s4 ... (100.00Mbit) (100.00Mbit) (100.00Mbit) (100.00Mbit)
Dumping host connections
h1 h1-eth0:s1-eth3
h2 h2-eth0:s1-eth4
h3 h3-eth0:s3-eth2
h4 h4-eth0:s4-eth2
```

From this output, we can see that the hosts and switches are initialized, and the links between them are automatically added. Finally, the command launches the Mininet CLI, within which we can test the connectivity between hosts to verify that the network was created correctly. This can be done using the command:

```
pingall
```

The expected result is as follows:

```
*** Ping: testing ping reachability
h1 -> h2 h3 h4
h2 -> h1 h3 h4
h3 -> h1 h2 h4
h4 -> h1 h2 h3
*** Results: 0% dropped (12/12 received)
```

Once the topology is created and the pinging is done, it becomes visible in the ONOS GUI, accessible through the browser.

4.2 Running the Firewall Service

Now the Firewall service is executed. This step initializes the network monitoring and flow rule installation process. Upon execution, the firewall retrieves and displays detailed information about the network topology. As previously described, this includes details about each switch, host and link in the network. For example, information regarding *Switch 1* is displayed as follows:

```
Device Id: of:000000000000000001
Device Type: SWITCH
Device Availability: true
Device Role: MASTER
Device MFR: SWITCH
Device SW: 2.13.8
Device HW: Open vSwitch
Device Serial: None
Chassis ID: 1
Device Annotation Management Address: 127.0.0.1
Device Annotation Protocol: OF_13
Device Annotation Channel Id: 127.0.0.1:59972
```

Similarly, information regarding *Host 1* is displayed as:

```
Host ID: 00:00:00:00:00:01/None
Host MAC: 00:00:00:00:00:01
Host VLAN: None
Host IPs: [10.0.0.1]
Locations list size: 1
Element ID: of:000000000000000001
Port: 3
```

Once the firewall completes its initialization and installs the necessary flow rules, the network is expected to block all host-to-host communication. This can be verified by running the `pingall` command in Mininet, which produces the expected following results:

```
*** Ping: testing ping reachability
h1 -> X X X
h2 -> X X X
h3 -> X X X
h4 -> X X X
*** Results: 100% dropped (0/12 received)
```

Furthermore, the installed flow rules can be directly inspected using the command:

```
dpctl dump-flows -O OpenFlow13
```

For example, on *Switch 1*, the following flow rules are observed, confirming that the connectivity for Host 1 and Host 2 (connected to Switch 1) is blocked:

```
*** s1 -----
cookie=0xc00000688ee7f6, duration=21.361s, table=0, n_packets=0, n_bytes=0,
send_flow_rem priority=60606,ip,nw_src=10.0.0.2,nw_dst=10.0.0.1 actions=drop
cookie=0xc00000842760e7, duration=21.348s, table=0, n_packets=0, n_bytes=0,
send_flow_rem priority=60606,ip,nw_src=10.0.0.2,nw_dst=10.0.0.4 actions=drop
cookie=0xc00000a8843d72, duration=21.333s, table=0, n_packets=0, n_bytes=0,
send_flow_rem priority=60606,ip,nw_src=10.0.0.2,nw_dst=10.0.0.3 actions=drop
cookie=0xc00000d110af0f, duration=21.322s, table=0, n_packets=0, n_bytes=0,
send_flow_rem priority=60606,ip,nw_src=10.0.0.1,nw_dst=10.0.0.2 actions=drop
cookie=0xc00000e7f51cbe, duration=21.309s, table=0, n_packets=0, n_bytes=0,
send_flow_rem priority=60606,ip,nw_src=10.0.0.1,nw_dst=10.0.0.4 actions=drop
cookie=0xc00000a7838afa, duration=21.291s, table=0, n_packets=0, n_bytes=0,
send_flow_rem priority=60606,ip,nw_src=10.0.0.1,nw_dst=10.0.0.3 actions=drop
```

Finally, the database table, shown in Figure 6, can also be explored, revealing all false value for every host in the network (no host is authenticated).

	us [PK] character varying (50)	psw character varying (50)	conn boolean
1	h1	h1	false
2	h2	h2	false
3	h3	h3	false
4	h4	h4	false

Figure 6: Database table before the authentication

4.3 Running the WebApp and Authentication

The Web App testing process begins by starting the *Lancia.js* server. Upon successful initialization, a confirmation message is displayed:

Server running on <http://localhost:3000>

After starting the server, the *SDNAuth_final* application is executed. This triggers the execution of *index.jsp*, which automatically opens the login page in a web browser, marking the application ready for user interaction.

To test the application's error-handling mechanisms, an attempt is made to log in with invalid credentials. If the provided username or password is incorrect, a pop-up message as the one in Figure 7 appears, stating: *"Invalid username or password."* In this case, no changes are made either to the database or to the network's configuration. The application's functionality ensures that no unauthorized host gains network access, maintaining the security of the system.

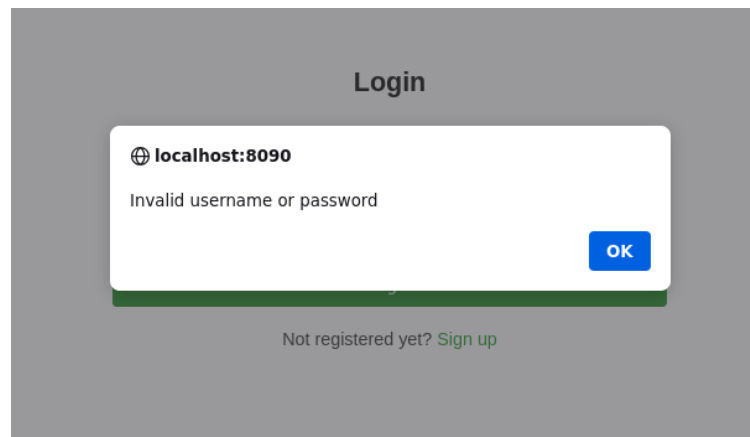


Figure 7: Login with invalid credentials

Next, valid credentials are provided for *Host 1*, with `username = h1` and `password = h1`. Upon clicking the login button, the application displays the message: "*Login successful*", as the one in Figure 8. It is **important to note** that for the program to function correctly and for the rules to be applied, the user must press the OK button below the "Login successful" pop-up message.

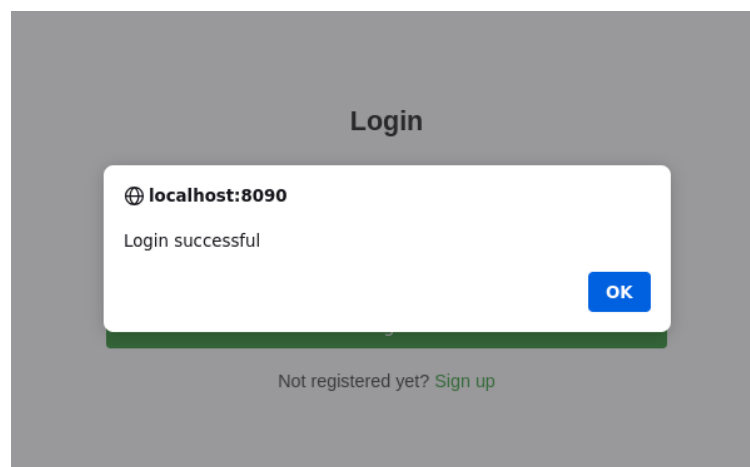


Figure 8: Login with valid credentials

At this point, *Host 1* (with IP address 10.0.0.1) is authenticated and should have network connectivity restored. To verify this, the flow rules installed on *Switch 1* are inspected using the `dpctl dump-flows -O OpenFlow13` command. The results show that the firewall rules blocking *Host 1* have been removed, confirming successful network reactivation for the authenticated host:

```
*** s1 -----
cookie=0xc00000688ee7f6, duration=96.005s, table=0, n_packets=0, n_bytes=0,
send_flow_rem priority=60606,ip,nw_src=10.0.0.2,nw_dst=10.0.0.1 actions=drop
cookie=0xc00000842760e7, duration=95.988s, table=0, n_packets=0, n_bytes=0,
send_flow_rem priority=60606,ip,nw_src=10.0.0.2,nw_dst=10.0.0.4 actions=drop
cookie=0xc00000a8843d72, duration=95.973s, table=0, n_packets=0, n_bytes=0,
send_flow_rem priority=60606,ip,nw_src=10.0.0.2,nw_dst=10.0.0.3 actions=drop
```

While *Host 1* is now authenticated and connected to the network, it remains unable to communicate with other hosts, as it is the only host currently authenticated. To further test the system, valid credentials for *Host 2* (username = h2, password = h2) are entered. After a successful login, both *Host 1* and *Host 2* are now authenticated and connected to the network. This allows them to exchange packets. A simple ping test (h1 ping h2) confirms successful communication:

```
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=6.46 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.890 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.096 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.193 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.117 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=0.093 ms
```

The results indicate that packets are successfully transmitted between the two authenticated hosts, with no packet loss observed. To confirm the integrity of the system, the flow rules on *Switch 1* can be checked once again. The rules blocking *Host 1* and *Host 2* are no longer present. For all other switches, the firewall rules remain unchanged, ensuring that unauthenticated hosts are still denied access to the network.

Finally, the database is examined to verify the authentication status of the hosts. The boolean values for *Host 1* and *Host 2* are updated to **true**, confirming their successful authentication. A visual representation of the database state is provided in Figure 9.




	us [PK] character varying (50) 	psw character varying (50) 	conn boolean 
1	h3	h3	false
2	h4	h4	false
3	h1	h1	true
4	h2	h2	true

Figure 9: Database table after the authentication of h1 and h2

5 Critical Conclusions

This project developed an authentication service for a network using a Software-Defined Networking (SDN) approach. The system was tested in a simulated environment with a Mininet network topology and an ONOS controller, demonstrating how SDN can effectively address security and network management needs. The goal was to create a flexible and dynamic authentication service operating at the application layer of the SDN stack. By using SDN's centralized control and programmability, the service adapts easily to changes in network topology, offering a significant improvement over traditional static network services. This adaptability makes the service suitable for various network setups and changes and highlights its potential as a flexible and reliable solution for modern network environments. Although tested in a simulation, we are confident that this system could be applied to real-world scenarios, such as enterprise networks or IoT environments, where scalability, adaptability, and strong security are essential.

References

- [1] ONOS, “Open networking operating system.” url: <https://opennetworking.org/onos/>.
- [2] ONOS, “Appendix b: Rest api.” url:<https://wiki.onosproject.org/display/ONOS/Appendix+B%3A+REST+API>.
- [3] Node.js, “Node.js.” url: <https://nodejs.org/en>.
- [4] E. O. Zaballa and R. Singh, “34359 software defined networking course slides,” 2025.
- [5] PostgreSQL, “Postgresql.” url: <https://www.postgresql.org/>.
- [6] JetBrains, “Enable the database tools and sql plugin in intellij.” url: <https://www.jetbrains.com/help/idea/postgresql.html>.