FOSSEE Autumn Internship Report

on

System Administration

Submitted By:

Kolli Jyothi Swaruph

VIT Bhopal University

Under the Guidance of:

Lee Thomas Stephen

and

Raghavjit Rana

January 2026

Acknowledgement

I want to take this opportunity to express my heartfelt gratitude to everyone who played a crucial role in making my autumn internship with FOSSEE - System Administration a valuable and rewarding experience. First and foremost, I extend my deepest thanks to Mr. Lee Thomas Stephen and Mr. Raghavjit Rana for believing in my abilities and selecting me for this project. Their unwavering support and encouragement have been instrumental in my progress.

I am deeply grateful to my mentors, whose guidance and wisdom have been indispensable throughout this journey. Their expert advice and timely assistance helped me complete my tasks and enhanced my technical and non-technical skills. Additionally,

I want to express my appreciation to my colleagues who are working on various projects. Their constructive feedback and insightful discussions have significantly contributed to my learning and growth. I extend my sincerest gratitude to everyone who supported me during this internship for their contributions to this unforgettable experience.

Declaration

I hereby declare that this written submission represents my ideas in my own words. Whenever the ideas or words of others have been included,

I have appropriately cited and referenced the sources. I have accurately acknowledged all sources used in producing this report. Furthermore, I confirm that I have adhered to all academic honesty and integrity principles. I have not misrepresented, fabricated, or falsified any idea, data, fact, or source in this submission.

I understand that any violation of the above principles will lead to disciplinary action by the Institute and may also result in legal consequences from sources that have not been appropriately cited or from those from whom proper permission was not obtained where required

Kolli Jyothi Swaruph

Contents

Introduction

During my internship, I worked on a diverse range of system administration tasks, including identity management, messaging platforms, and cloud migration. Guided by real-world operational needs, I focused on migrating legacy services, containerizing applications, and automating SSO integrations to build secure, reliable systems.

A major effort involved migrating mailing infrastructure from Mailman 2 on CentOS 7 to Mailman 3 on AlmaLinux 10. This modernized the stack while preserving historical data and list functionality. I also containerized and deployed several applications, such as Zulip, Discourse, and an OpenPLC Drupal site, using Docker and Podman. These deployments significantly improved service isolation, repeatability, and maintainability.

SSO integration using Keycloak and OIDC was a key theme throughout. I integrated Keycloak with various applications and authored automation scripts to ensure reproducible configurations across different environments. Additionally, I handled cloud migration by exporting VirtualBox VMs to AWS as AMIs, orchestrating EC2 instances with Ansible, and configuring Application Load Balancers for scalability.

Collectively, these tasks deepened my expertise in Linux administration, containerization (Docker / Podman), and authentication protocols. The experience strengthened my ability to translate technical requirements into automated, production-oriented workflows while contributing tangible improvements to real-world projects.

Chapter 1

Drupal Site Deployment and Keycloak SSO Integration

This chapter details the deployment of the OpenPLC Drupal site and the subsequent integration of Keycloak for Single Sign-On (SSO). The work began with a local deployment to understand the application architecture, followed by a deployment on a virtualized Rocky Linux environment where both Drupal and Keycloak were hosted on a single machine. Finally, the setup was evolved into a distributed topology with Drupal and Keycloak running on separate Virtual Machines (VMs), utilizing automated scripts for configuration [10].

1.1 OpenPLC Local Deployment

The initial phase involved setting up the OpenPLC Drupal site locally using Podman to ensure the application logic and building process were functioning correctly. Podman was chosen over Docker because it operates in rootless mode by default, containers run as the unprivileged user rather than as root, reducing the attack surface for a web-facing application.

1.1.1 Directory and Environment Setup

A dedicated directory was created for the OpenPLC project. A `sites.json` file was defined to configure the repository and port settings for the site:

Listing 1.1: Configuration in sites.json

```
[
  {
    "SITE_NAME": "openplc",
    "PORT": 8081,
    "REPO": "https://github.com/FOSSEE/openplc_docker_image"
  }
]
```

The necessary system dependencies, including `mysql`, `podman`, `slirp4netns`, and `jq`, were installed to support the containerized environment.

1.1.2 Initialization and Execution

The `init_sites` [8] script was executed to clone the Drupal repository (main branch) and initialize the database. During configuration, the site name was set to `openplc`, and the

database `openplcdb` was linked to the user `openplcuser`.

To finalize the setup, we accessed the running container to create an administrator account using Drush:

Listing 1.2: Creating an administrator user using Drush

```
podman exec -it openplc_container bash
vendor/bin/drush user-create <username> --mail="<email>" --password="<password>"
vendor/bin/drush user-add-role "administrator" <username>
```

The site was successfully verified by accessing `http://localhost:8081`.

1.2 Drupal Site with Keycloak SSO on Single VM

The next task was to deploy the application on a Rocky Linux VM and integrate it with Keycloak for authentication.

1.2.1 VM and System Configuration

A Rocky Linux VM was provisioned using VirtualBox with a Bridged Adapter to ensure network accessibility from the host machine, which simplified initial testing. Essential firewall ports (8080 for Keycloak, 8081 for Drupal) were opened, and MySQL 8.4 was installed and secured.

1.2.2 Application Deployment

We reused the `init_sites` [8] and `sites.json` logic to deploy the OpenPLC container on the VM. A persistent volume `openplc_volume` was created to store site data.

1.2.3 Keycloak Installation

Keycloak 26.4.0 was installed to manage authentication. The setup involved:

- Installing Java 21 OpenJDK.
- Creating a dedicated `keycloak` system user and group.
- Configuring Keycloak to use the local MySQL database instead of the default H2 database. Keycloak ships with an embedded H2 database suitable only for development; H2 does not support concurrent access well and cannot be easily backed up, making MySQL the correct choice for any persistent deployment.
- Setting up a systemd service file `/etc/systemd/system/keycloak.service` to manage the Keycloak daemon.

1.2.4 SSO Configuration

Integration was achieved using the OpenID Connect protocol.

1. **Drupal Client**: The `openid_connect` and `keycloak` modules were installed via Composer and enabled using Drush.
2. **Keycloak Server**: A new client `openplc` was created in the Keycloak admin console with the valid redirect URI pointing to the Drupal site.

After exchanging the Client ID and Secret, the login flow was tested. Users attempting to access the Drupal login page were successfully redirected to Keycloak and authenticated.

1.2.5 Persistence Strategy

To ensure configuration changes like installed modules persisted across container restarts, the running container was committed to a new image, and the service was updated to use this custom image. Drupal module installations via Composer modify the container's filesystem; since the container is ephemeral, these changes would be lost on restart. Committing via `podman commit` was chosen over volume-mounting the entire Drupal codebase, as it is simpler and keeps the deployment self-contained.

1.3 Drupal and Keycloak Integration on Separate VMs

To simulate a robust production environment, Drupal and Keycloak were moved to separate VMs, communicating over a private network.

1.3.1 Network Configuration

Two VMs were created: `drupal-vm` and `keycloak-vm`. A Host-Only network was configured in VirtualBox to allow isolated communication between them. Unlike the Bridged Adapter used in the single-VM setup, a Host-Only network creates a private network that is not reachable from outside the host machine — the correct model for a backend authentication service like Keycloak that should not be directly internet-accessible. Static IP addresses were assigned:

- **Drupal VM**: 192.168.56.10
- **Keycloak VM**: 192.168.56.11
- Connectivity was verified using `ping`.

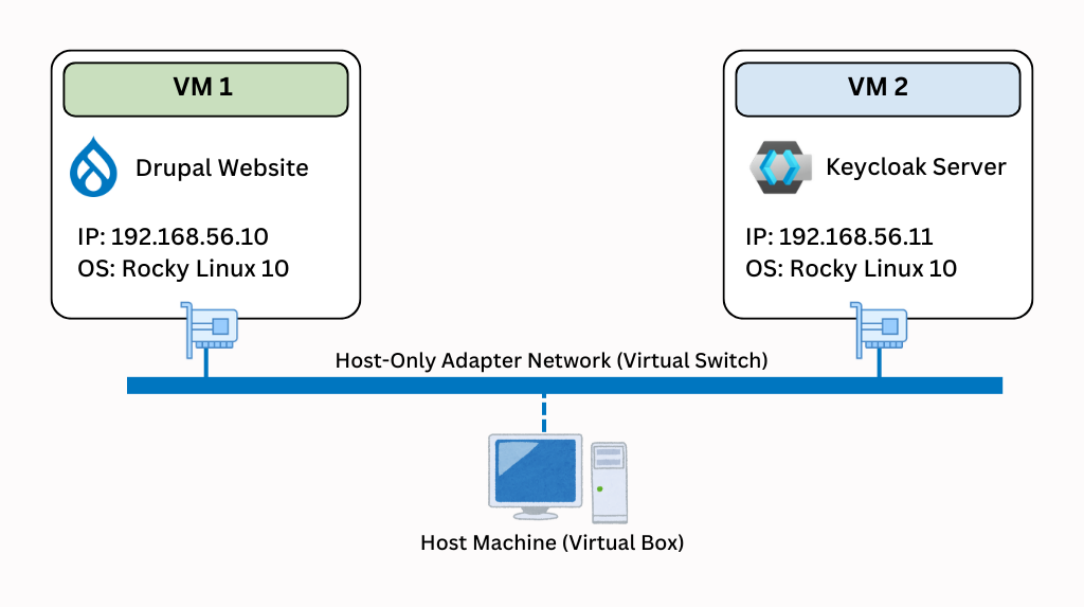

Figure 1.1: Network Configuration of Drupal and Keycloak

1.3.2 Automated Client Configuration

Instead of manual configuration, shell scripts were developed to automate the setup.

Keycloak Script

A `keycloak_client_setup` script [11] was written to interact with the Keycloak server locally. By reading a `config.json`, it automatically creates the `openplc` client with the correct redirect URIs for the remote Drupal VM.

Drupal Script

A `drupal_keycloak_sso` script [7] utilizing Drush commands was used on the Drupal VM to configure the OpenID Connect settings. Key configurations included:

Listing 1.3: Drush commands for OIDC; Path: /var/www/html inside drupal container)

```
vendor/bin/drush config:set openid_connect.settings.keycloak enabled true
vendor/bin/drush config:set openid_connect.settings.keycloak settings.client_id \
'$CLIENT_ID'
vendor/bin/drush config:set openid_connect.settings.keycloak settings.keycloak_base \
'$KEYCLOAK_BASE'
vendor/bin/drush config:set openid_connect.settings.keycloak settings.keycloak_sso
    ↪ true
```

1.3.3 Verification

The setup was validated by accessing `http://192.168.56.10:8081/user/login`. The browser was correctly redirected to the Keycloak server at `192.168.56.11:8080`, confirming the SSO integration between the isolated environments.

1.4 Challenges and Troubleshooting

1.4.1 Keycloak Redirect URI Mismatch

The most common issue encountered during SSO configuration was a `redirect_uri mismatch` error from Keycloak. This occurred when the "Valid Redirect URIs" configured in the Keycloak client did not exactly match the callback URL that Drupal sent during the OIDC flow. The fix required ensuring the Drupal site's base URL was correctly set and that the Keycloak client's redirect URI included the trailing wildcard (`http://192.168.56.10:8081/*`).

1.4.2 Firewall Blocking Inter-VM Communication

After configuring static IPs on the Host-Only network, the VMs could not communicate despite the network being correctly set up. The cause was `firewalld` blocking traffic on the host-only interface. The fix was to open the required ports (8080 for Keycloak, 8081 for Drupal) explicitly:

Listing 1.4: Opening required firewall ports

```
sudo firewall-cmd --permanent --add-port=8080/tcp
sudo firewall-cmd --permanent --add-port=8081/tcp
sudo firewall-cmd --reload
```

Chapter 2

Automation of Keycloak and Drupal SSO Configuration

This chapter describes the development of automation scripts to streamline the integration of Keycloak Single Sign-On (SSO) with Drupal. The manual configuration process, involving multiple steps on both the Keycloak Identity Provider and the Drupal Service Provider, was encapsulated into shell scripts to ensure reproducibility and reduce configuration drift [10].

2.1 Overview of Automation Components

The automation suite consists of two primary scripts:

1. `keycloak_client_setup` [11]: A script executed on the Keycloak server to automatically create and configure the OpenID Connect client.
2. `drupal_keycloak_sso` [7]: A script executed on the Drupal server to install required modules and configure the OpenID Connect provider settings.

These scripts are designed to work in tandem to establish a secure authentication flow between the two systems.

2.2 Keycloak Client Configuration Script

The `keycloak_client_setup` script [11] leverages the Keycloak Admin CLI (`kcadm.sh`) to manipulate the Keycloak configuration programmatically. `kcadm.sh` was chosen over direct REST API calls (e.g., with `curl`) because it handles token management automatically — a direct REST approach requires obtaining a bearer token, passing it with every request, and handling expiry, all of which the Admin CLI abstracts away.

2.2.1 Environment Preparation

A dedicated directory `~/keycloak-automation` is created to house the scripts and configuration files. Two JSON files are required to run the script:

- `config.json`: specific connection details and credentials for the Keycloak realm and the target Drupal client. Driving the script from a JSON file rather than command-line arguments separates configuration from execution, making it easier to version-control and re-run without re-entering credentials.

- `client.json`: generated by the script to store the output configuration, including the client secret. Writing the secret to a file creates a clean handoff mechanism to the Drupal script, avoiding the need to manually copy secrets from terminal output.

2.2.2 Configuration Structure

The `config.json` file requires the following structure:

Listing 2.1: Structure of config.json

```
[
  {
    "keycloak_realm": "fossee",
    "keycloak_base": "http://192.168.56.11:8080",
    "keycloak_user": "admin",
    "keycloak_password": "adminpassword",
    "drupal_base": "http://192.168.56.10:8081",
    "keycloak_client_id": "drupal-client"
  }
]
```

2.2.3 Execution

The script is downloaded and executed on the Keycloak server. It authenticated with the local Keycloak instance, creates a new client definition matching the `keycloak_client_id` [11], and configures the valid redirect URIs based on the `drupal_base` URL.

Listing 2.2: Execution of keycloak_client_setup

```
wget https://raw.githubusercontent.com/kjswaruph/fossee-daily-progress/refs/heads/main
    ↪ /sso-automation/keycloak_client_setup
chmod +x keycloak_client_setup
./keycloak_client_setup
```

2.3 Drupal SSO Configuration Script

The `drupal_keycloak_sso` script [7] facilitates the configuration on the Drupal side, utilizing `Composer` for dependency management and `Drush` for configuration updates.

2.3.1 Functionality

The script performs the following actions:

- Installs the `drupal/openid_connect` and `drupal/keycloak` modules.
- Creates a Drupal user with administrator privileges to manage the configuration.
- Sets the OpenID Connect settings to point to the Keycloak instance.

2.3.2 Usage

The script accepts command-line arguments to specify the connection parameters. It is executed within the `/var/www/html` directory inside the Drupal container. This working directory requirement is important as Drush resolves configuration paths relative to the current directory, so running the script from any other location causes it to fail.

Listing 2.3: Running drupal_keycloak_sso script; Path: /var/www/html inside drupal container

```
./drupal_keycloak_sso \
  --client-id <your-client-id> \
  --client-secret <your-client-secret> \
  --base <keycloak-base-url> \
  --realm <keycloak-realm> \
  --email <user-email> \
  --drupal-user <drupal-username> \
  --drupal-password <drupal-password> \
  --user <system-user>
```

Table 2.1 describes each argument and where to obtain its value.

Table 2.1: Arguments for `drupal_keycloak_sso`

Argument	**Description and source**
`--client-id`	The Client ID of the Keycloak client created for Drupal (e.g., `drupal-client`). Found in Keycloak Admin Console → Clients.
`--client-secret`	The client secret generated by Keycloak. Found under Clients → *drupal-client* → Credentials tab.
`--base`	The base URL of the Keycloak server, including port (e.g., `http://192.168.56.11:8080`).
`--realm`	The Keycloak realm name under which the client was created (e.g., `fossee`).
`--email`	The email address to assign to the Drupal administrator account that the script creates.
`--drupal-user`	The username for the Drupal administrator account (e.g., `admin`).
`--drupal-password`	The password for the Drupal administrator account.
`--user`	The Linux system user who owns the Drupal installation directory (e.g., `www-data`).

2.3.3 Verification of Configuration

Post-execution, the configuration can be verified using Drush to ensure the Keycloak settings are correctly applied.

Listing 2.4: Verifying Keycloak settings with Drush (Run inside drupal container)

```
vendor/bin/drush config:get openid_connect.settings.keycloak
```

This command should output a YAML configuration confirming that the `enabled` flag is set to `true` and the client ID, secret, and Keycloak URLs match the provided arguments.

2.4 Validation of the Integrated Workflow

The complete workflow involves running the Keycloak script first to generate the client credentials, followed by the Drupal script to consume them. Validation involves accessing the Drupal login page `/user/login`, which should redirect the user to the Keycloak authentication page. Upon successful login, the user is redirected back to Drupal with an active session, confirming the end-to-end functionality of the automated setup.

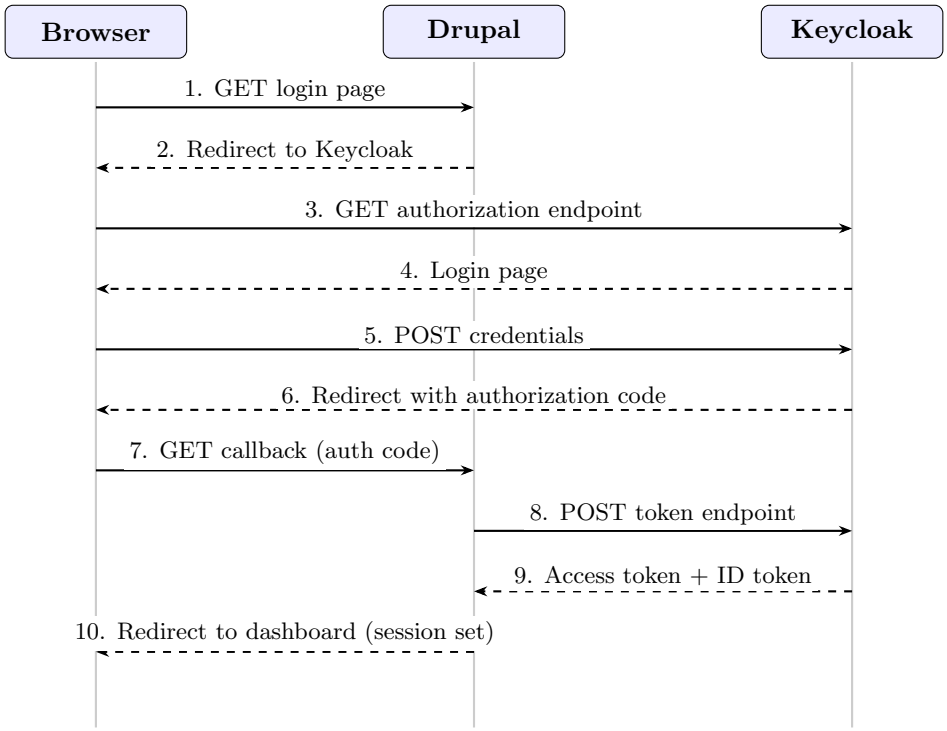

Figure 2.1: OIDC authorization code flow between Browser, Drupal, and Keycloak

2.5 Challenges and Troubleshooting

2.5.1 Drush Configuration Requiring Correct Working Directory

The `drupal_keycloak_sso` [7] script must be executed from within the Drupal site root `/var/www/html`, as Drush resolves configuration paths relative to the current directory. Running the script from any other location caused Drush to fail. This was addressed by having the script accept a `--user` argument specifying the system user who owns the Drupal directory, and internally changing to the correct directory before invoking Drush.

2.5.2 Keycloak Admin CLI Authentication Failure

The `kcadm.sh` tool requires an initial authentication step before any administrative commands can be run. If the Keycloak server URL or admin credentials in `config.json` were incorrect, the script would fail silently on subsequent commands rather than reporting the authentication error clearly. This was debugged by running the `kcadm.sh config credentials` command manually to confirm the connection before executing the full script.

Chapter 3

Keycloak SSO Integration for Online Test Application

This chapter documents the integration of the FOSSEE Online Test application with Keycloak for Single Sign-On (SSO) [14]. The application, built with Django, is containerized using Docker. The integration utilizes the `mozilla-django-oidc` [13] middleware to delegate authentication to the Keycloak Identity Provider working on Rocky Linux 10 [10].

3.1 Environment Setup

The deployment environment requires Docker and Docker Compose to manage the application containers.

3.1.1 Docker Installation

Docker was used here rather than Podman because the upstream `online_test` repository ships with a `docker-compose.yml` designed for Docker; adapting it to Podman Compose would have risked diverging from the upstream project's tested configuration. [9].

Listing 3.1: Installing Docker on Rocky Linux

```
sudo dnf -y install dnf-plugins-core
sudo dnf config-manager --add-repo https://download.docker.com/linux/rhel/docker-ce.
    ↪ repo
# Install packages
sudo dnf install docker-ce docker-ce-cli containerd.io docker-buildx-plugin \
docker-compose-plugin -y
```

3.2 Application Deployment

The Online Test application source code was cloned, and the container environment was prepared.

3.2.1 Configuration

The environment configuration file `.env` was created from the sample template. Additionally, dependencies were updated to include the OIDC library.

Listing 3.2: Adding OIDC dependency

```
echo "mozilla_django_oidc" >> docker/Files/requirements-common.txt
```

The Dockerfiles for the Django application and the code-server were updated to use `ubuntu:20.04` as the base image, as the original `ubuntu:18.04` had reached End of Life. Critically, the `ENV DEBIAN_FRONTEND=noninteractive` variable was added to prevent `apt-get` from hanging on interactive prompts (such as timezone configuration) during the build, which would cause it to stall indefinitely in a non-interactive environment.

3.3 Keycloak Client Configuration

A new OpenID Connect client was created in the Keycloak administration console to handle the authentication requests from the Django application.

- **Client ID**: `online-test`
- **Client Authentication**: Enabled
- **Valid Redirect URIs**: `https://<django_domain>/oidc/callback/`

The Client Secret was generated and securely stored for use in the Django settings.

3.4 Django OIDC Integration

The core integration involved configuring the `mozilla_django_oidc` [13] library within the Django project. This library was chosen over implementing the OIDC flow from scratch because it provides a complete Django authentication backend that handles the full authorization code flow, including token validation and automatic user creation on first login.

3.4.1 Settings Configuration

The `settings.py` file was modified to include the OIDC app, authentication backend, and connection parameters.

Listing 3.3: Django OIDC Settings

```
INSTALLED_APPS = [
    # ...
    'mozilla_django_oidc',
]

AUTHENTICATION_BACKENDS = (
    'mozilla_django_oidc.auth.OIDCAuthenticationBackend',
    'django.contrib.auth.backends.ModelBackend',
)

# Keycloak Configuration
OIDC_RP_CLIENT_ID = "online-test"
OIDC_RP_CLIENT_SECRET = "your_client_secret"
OIDC_OP_AUTHORIZATION_ENDPOINT = "http://<keycloak-url>/realms/<realm>/protocol/openid
    ↪ -connect/auth"
OIDC_OP_TOKEN_ENDPOINT = "http://<keycloak-url>/realms/<realm>/protocol/openid-connect
    ↪ /token"
OIDC_OP_USER_ENDPOINT = "http://<keycloak-url>/realms/<realm>/protocol/openid-connect/
    ↪ userinfo"
OIDC_RP_SIGN_ALGO = "RS256"
```

The `OIDC_RP_CLIENT_SECRET` is obtained from the Keycloak administration console: navigate to **Clients** → **online-test** → **Credentials** tab, then copy the value from the **Client Secret** field. Replace `<keycloak-url>` with the Keycloak server's hostname or IP (e.g., `192.168.56.11:8080`) and `<realm>` with the realm name (e.g., `fossee`).

3.4.2 URL Routing and UI

The OIDC URLs were included in `urls.py`. A "Login with Keycloak" button was added to the login template, linking to the `oidc_authentication_init` URL.

3.5 Build and Verification

The Docker containers were built and started. Database migrations, including those for the notifications plugin, were applied, and initial data fixtures were loaded.

Listing 3.4: Applying Migrations

```
docker compose exec yaksh-django python3 manage.py makemigrations notifications_plugin
docker compose exec yaksh-django python3 manage.py migrate
docker compose exec yaksh-django python3 manage.py loaddata demo_fixtures.json
```

3.5.1 SSO Testing

The integration was verified by navigating to the application's login page and authenticating via Keycloak. Upon success, the user was redirected back to the application and automatically logged in, with the Django user account being created if it did not exist.

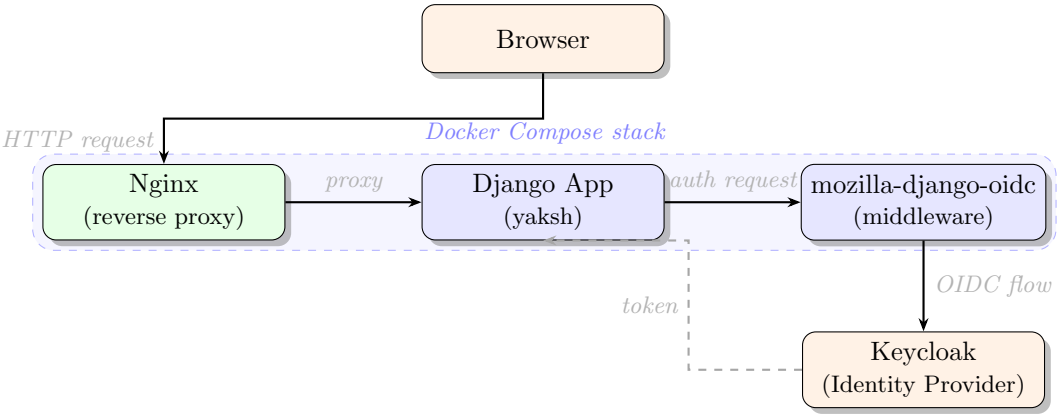

Figure 3.1: Django Online Test SSO architecture: Nginx, Django, and Keycloak via OIDC middleware

3.6 Challenges and Troubleshooting

3.6.1 Docker Build Hanging on `apt-get`

The Docker image build would hang indefinitely at the `apt-get install` step. This was caused by a package (specifically `tzdata`) prompting for timezone selection during installation. Since the Docker build runs non-interactively, there is no terminal to respond to the prompt. The fix was adding `ENV DEBIAN_FRONTEND=noninteractive` to the Dockerfile, which instructs `apt-get` to use default values for all prompts without user interaction.

3.6.2 OIDC Callback URL Not Registered

After configuring the Django OIDC settings, the login redirect to Keycloak succeeded, but Keycloak returned an "Invalid redirect URI" error. This was because the Keycloak client's "Valid Redirect URIs" did not include the specific OIDC callback path that `mozilla-django-oidc` uses (`/oidc/callback/`). The fix was to add `http://<host>/oidc/callback/` to the list of valid redirect URIs in the Keycloak client configuration.

Chapter 4

Migrating Mailman Deployment: CentOS 7 to AlmaLinux 10

This chapter documents the process of migrating a legacy GNU Mailman 2 mailing list server hosted on CentOS 7 to a modern containerized GNU Mailman 3 deployment on AlmaLinux 10. This migration is essential to ensure security updates, utilize modern Python 3 infrastructure, and benefit from the enhanced features of the Mailman 3 suite, including the HyperKitty archiver. [12].

4.1 Infrastructure Setup and Networking

The migration environment consisted of two Virtual Machines (VMs) running on VirtualBox: a source machine running CentOS 7 and a destination machine running AlmaLinux 10.

4.1.1 Virtual Machine Configuration

Both VMs were configured with dual network adapters:

- **Adapter 1 (NAT)**: To provide internet access for downloading packages.
- **Adapter 2 (Host-Only)**: To allow direct communication between the VMs for data transfer.

Network interfaces were configured using `nmcli` to assign static IP addresses on the Host-Only network (192.168.56.x/24), ensuring reliable connectivity between the source and destination.

4.2 Legacy Environment (CentOS 7)

The source environment was set up to simulate an active Mailman 2 server. Since CentOS 7 reached End of Life, its official mirrors were decommissioned; the package repositories had to be reconfigured to point to the CentOS Vault (`vault.centos.org`), which archives all historical CentOS releases, before any packages could be installed.

4.2.1 Mailman 2 Configuration

Mailman 2 was installed along with the Postfix MTA and Dovecot for IMAP/POP3 services. The `mm_cfg.py` and `main.cf` configuration files were adjusted to verify a working mailing list environment.

Listing 4.1: Mailman 2 Default URL Host Configuration

```
# Change to your hostname with FQDN
DEFAULT_URL_HOST = 'your_domain'
DEFAULT_EMAIL_HOST = 'your_domain_email'
```

4.3 Modern Environment (AlmaLinux 10)

The destination environment was prepared with AlmaLinux 10, leveraging Docker and Docker Compose to host the Mailman 3 stack.

4.3.1 Prerequisites and Docker Installation

Essential packages including git and postfix were installed. The Docker engine was set up by adding the official Docker CE repository [9].

Listing 4.2: Installing Docker on AlmaLinux 10

```
sudo dnf -y install dnf-plugins-core
sudo dnf config-manager --add-repo https://download.docker.com/linux/rhel/docker-ce.
    ↪ repo
sudo dnf install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-
    ↪ compose-plugin -y
sudo systemctl enable --now docker
```

4.3.2 Mailman 3 Deployment

The official `docker-mailman` repository was cloned to `/opt`. Docker Compose was chosen for the Mailman 3 deployment rather than a native installation because Mailman 3 consists of multiple tightly coupled components (Core REST API, Web interface, and database) that the `docker-mailman` project pre-configures into a tested, compatible stack, dramatically reducing setup complexity. The directory structure for persistent data (core, web, database) was created manually to ensure correct permissions.

Unique secrets for the Django application and the HyperKitty API were generated and added to the `docker-compose.yaml` file.

Listing 4.3: Generating Secrets for Mailman 3

```
openssl rand -base64 50 # run twice for two unique keys
```

The Postfix MTA was deliberately kept on the AlmaLinux host rather than running it inside a container. This is because Postfix needs to bind to port 25 (SMTP), which requires elevated privileges that a container may not have. More importantly, the host Postfix acts as the bridge between the external mail world and the Mailman containers: it receives inbound mail and uses transport maps generated by Mailman Core to route messages to the correct container via LMTP. The Postfix configuration on the host was updated to relay emails to and from the containers using these transport maps.

4.4 Migration Process

The migration involved exporting data from the legacy system, transferring it to the new server, and importing it into the Mailman 3 components (Core and HyperKitty).

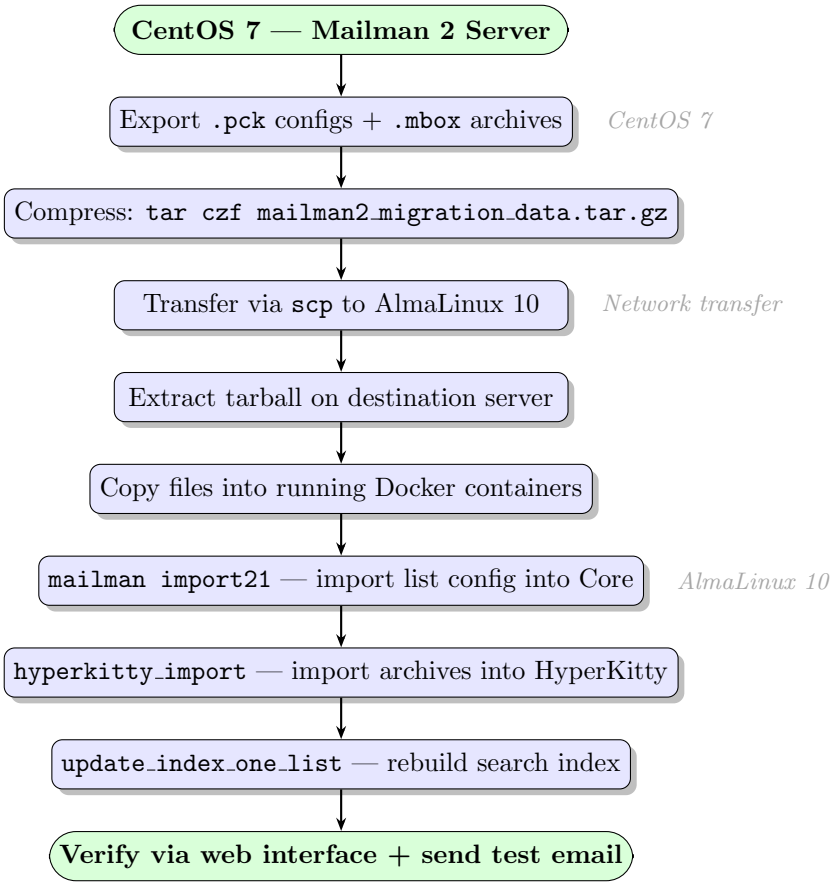

Figure 4.1: Mailman 2 to Mailman 3 migration pipeline

4.4.1 Data Export Strategy

On the CentOS 7 server, we identified and exported two critical data sets:

1. **List Configurations**: The pickle formatted (`.pck`) files containing list settings, located in `/var/lib/mailman/lists/`.
2. **Archives**: The mbox files containing message history, located in `/var/lib/mailman/archives/private/`.

These were compressed into a single tarball `mailman2_migration_data.tar.gz` and transferred via `scp` to the AlmaLinux server.

4.4.2 Data Import into Mailman 3

After extracting the data on the destination server, the files were copied into the running Docker containers.

Importing List Configuration

The list settings were imported into the Mailman Core container using the `import21` command, which converts Mailman 2.1 data structures to Mailman 3.

Listing 4.4: Importing List Configuration to Mailman Core

```
docker exec -it mailman-core bash
mailman import21 mylist /tmp/mm2_data/config.pck
```

Importing Archives

The message history was imported into HyperKitty (the web archiver) running in the `mailman-web` container. This process also involved rebuilding the search index to ensure old messages were searchable.

Listing 4.5: Importing Archives to HyperKitty

```
docker exec -it mailman-web-1 bash
python3 manage.py hyperkitty_import -l mylist@example.com /tmp/mm2_data/mylist.mbox
python3 manage.py update_index_one_list mylist@example.com
```

4.5 Verification

The migration was validated by accessing the Mailman 3 web interface at the server's IP address. We verified that the migrated mailing lists were present and that the historical archives were accessible via HyperKitty. Finally, a test email was sent to the list to confirm that the new Postfix relay configuration correctly delivered messages to the subscribers.

4.6 Challenges and Troubleshooting

4.6.1 Postfix Not Relaying Mail from Containers

After starting the Mailman 3 containers, outbound email from the Mailman web interface was not being delivered. The root cause was that Postfix's `mynetworks` parameter only trusted `127.0.0.0/8` by default. The Docker containers have their own internal IP range (typically `172.x.x.x`), which Postfix treated as untrusted and rejected with a "Relay access denied" error. The fix was to add the Docker bridge network subnet to `mynetworks` in `/etc/postfix/main.cf`.

Chapter 5

Cloud Migration: From VirtualBox OVA to AWS AMI

This chapter details the workflow for migrating a local VirtualBox Virtual Machine (VM) hosting the `online-test` application to Amazon Web Services (AWS) as an Amazon Machine Image (AMI). Furthermore, it covers the orchestration of a scalable fleet of EC2 instances using Ansible to deploy the application behind an Application Load Balancer (ALB).

5.1 Local VM Preparation

The migration source was a Rocky Linux 10 VM configured with the `online-test` application stack, which includes Docker, Django, and Code-Server.

5.1.1 Application Setup

The VM was provisioned with Docker and the application dependencies. A systemd service was created to ensure the Docker Compose stack starts automatically on boot.

Listing 5.1: Systemd service for online-test

```
[Unit]
Description=yaksh service
Requires=docker.service
After=docker.service

[Service]
Type=oneshot
RemainAfterExit=yes
WorkingDirectory=/home/online-test/online_test/docker/
ExecStart=docker compose up -d
ExecStop=docker compose down

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

5.1.2 System Cleanup for Export

To ensure the exported image is cloud-agnostic and secure, we performed a thorough cleanup. SSH host keys must be regenerated uniquely for each instance — if they are baked into the

image, every EC2 instance launched from the AMI would have identical host keys, a serious security vulnerability. `cloud-init` handles first-boot provisioning on AWS (injecting the SSH keypair, setting the hostname), so resetting it ensures it runs correctly on the first boot of each new instance. This involved removing SSH host keys, clearing shell history, and resetting `cloud-init`.

Listing 5.2: Cleanup commands before export

```
sudo rm -f /etc/ssh/ssh_host_*
sudo rm -f /etc/NetworkManager/system-connections/*
sudo cloud-init clean --logs
history -c
```

Finally, the VM was exported from VirtualBox as an Open Virtualization Archive (OVA) file named `rocky10-image.ova`. OVA was chosen as the migration format because it is VirtualBox's standard export format and is directly supported by the AWS VM Import/Export service, making it the most straightforward path from VirtualBox to AWS.

5.2 Importing to AWS

The migration to AWS involves storing the OVA in an S3 bucket and using the AWS System Manager's VM Import/Export service. [15].

5.2.1 Infrastructure Prerequisites

An S3 bucket was created in the target region to store the large OVA file. Additionally, an IAM role named `vmimport` was required to grant the VM Import service permissions to read from S3 and write snapshots to EC2.

Listing 5.3: IAM Role Trust Policy

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": { "Service": "vmie.amazonaws.com" },
      "Action": "sts:AssumeRole",
      "Condition": {
        "StringEquals": {
          "sts:Externalid": "vmimport"
        }
      }
    }
  ]
}
```

5.2.2 Executing the Import

After uploading the OVA to S3, the import task was initiated triggers the conversion of the uploaded disk image into an EBS snapshot and subsequently an AMI.

Listing 5.4: AWS CLI Import Command

```
aws ec2 import-image \
```

```
    --description "Online Test Server VM" \
    --disk-containers '[{
    "Format": "OVA",
    "UserBucket": {
      "S3Bucket": "your_bucket_name",
      "S3Key": "rocky10-image.ova"
    }
  }]'
```

5.3 Orchestration with Ansible

Once the AMI was available, we automated the deployment of the application infrastructure using Ansible. The architecture consists of multiple EC2 instances running behind an Application Load Balancer.

5.3.1 Ansible Configuration

A global configuration file `aws.yml` was used to define project variables such as the region, instance count, and the specific AMI ID derived from the import process.

Listing 5.5: Ansible global variables (aws.yml)

```
---
project_name: online-test
aws_region: ap-south-1
instance_count: 5
instance_type: your_instance_type
ami_id: your_ami_id
keypair_name: online-test

vpc_id: "your_vpc_id"
subnet_ids:
  - "your_subnet_id_1"
  - "your_subnet_id_2"
```

5.3.2 Playbook Workflow

The deployment is split into modular playbooks. This modular structure allows individual components (keypairs, security groups, EC2 instances, ALB) to be updated or re-deployed independently without affecting the others:

1. **keypair.create.yml** [4]: Imports the local SSH public key into AWS to allow access to the instances.
2. **sg.create.yml** [3]: Creates Security Groups. One for the ALB (allowing HTTP traffic from the world) and one for the EC2 instances (allowing traffic only from the ALB and SSH).
3. **ec2.create.yml** [2]: Launches the fleet of EC2 instances using the imported AMI and attaches them to the created security group.
4. **alb.create.yml** [1]: Provisions the Application Load Balancer and Target Group, registering every running instance as a target for load balancing.

5.3.3 Deployment and Verification

The playbooks were executed sequentially. Upon completion, the application was accessible via the ALB's DNS name, effectively distributing traffic across the five instances created from the original VirtualBox VM.

Listing 5.6: Deploying infrastructure with Ansible

```
ansible-playbook keypair.create.yml
ansible-playbook sg.create.yml
ansible-playbook ec2.create.yml
ansible-playbook alb.create.yml
```

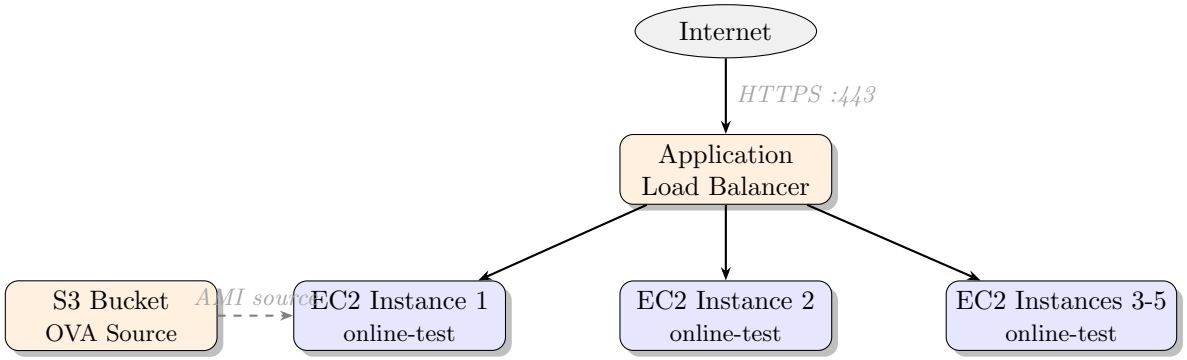

Figure 5.1: AWS deployment architecture: EC2 fleet behind an ALB

5.4 Challenges and Troubleshooting

5.4.1 IAM Role Configuration for VM Import

The AWS VM Import service requires a specific IAM role named exactly `vmimport` with a trust policy allowing the `vmie.amazonaws.com` service to assume it. The import task would fail with an "Access Denied" error if the role name was incorrect or if the trust policy's `sts:Externalid` condition value did not match `vmimport`. This was resolved by carefully following the AWS documentation for the exact trust and permission policy JSON structures.

5.4.2 Import Task Taking Extended Time

The `aws ec2 import-image` task is asynchronous and can take 30–60 minutes for a large OVA file. The task status was monitored using the `aws ec2 describe-import-image-tasks` command. An initial failure occurred because the S3 bucket was in a different region than the target EC2 region. The fix was to ensure the S3 bucket and the import task were both initiated in the same AWS region.

5.4.3 cloud-init Not Running on First Boot

After launching an EC2 instance from the imported AMI, the SSH keypair injected by AWS was not being applied, making the instance inaccessible. This was because `cloud-init` had not been properly reset before the OVA export, so it detected it had already run and skipped first-boot provisioning. The fix was to re-export the OVA after running `sudo cloud-init clean --logs` on the source VM.

Chapter 6

Deploying Zulip Chat Server with Podman

This chapter describes the deployment of the Zulip team collaboration chat server using Podman containers. The setup includes configuring Nginx as a reverse proxy with SSL termination and establishing a Postfix mail relay to facilitate email notifications via an external SMTP provider (Gmail).

6.1 System Preparation

The host system was updated, and essential repositories (EPEL, Remi) were enabled. Dependencies including `podman`, `podman-compose`, `git`, `nginx`, and `postfix` were installed.

6.1.1 System Limits Configuration

Zulip requires increased file descriptor limits. We modified `/etc/security/limits.conf` to accommodate high-performance requirements.

Listing 6.1: Increasing file descriptor limits

```
youruser soft nofile 1048576
youruser hard nofile 1048576
youruser soft nproc  65535
youruser hard nproc  65535
```

6.2 Containerized Zulip Deployment

The official Zulip Docker repository was cloned, and the `docker-compose.yml` file was customized.

6.2.1 Environment and Network Configuration

Environment variables were updated with secure passwords generated using `openssl`. Port mappings were adjusted because ports below 1024 are privileged and require root access to bind — since Podman runs rootlessly, it cannot bind to ports 25, 80, or 443 directly. The remapping allows the rootless containers to use unprivileged ports, with Nginx on the host handling the public-facing ports:

- SMTP: 1025 (Host) → 25 (Container)
- HTTP: 8080 (Host) → 80 (Container)
- HTTPS: 8443 (Host) → 443 (Container)

6.2.2 Service Startup

The containers were orchestrated using Podman Compose.

Listing 6.2: Starting Zulip containers with Podman

```
podman compose up -d
podman ps
```

6.3 Reverse Proxy and SSL

Nginx was configured as a reverse proxy rather than exposing the Zulip container's HTTPS port directly. This provides a clean separation of concerns: Nginx handles SSL certificate management (via Certbot/Let's Encrypt) and public traffic, while Zulip handles application logic. It also makes it straightforward to add other services behind the same Nginx instance in the future.

6.3.1 SSL Certificate

We utilized `Certbot` to obtain a free SSL certificate from Let's Encrypt for the domain.

Listing 6.3: Obtaining SSL certificate

```
sudo certbot --nginx -d your.domain.com
```

6.3.2 Nginx Configuration

The Nginx server block was updated to proxy requests to the Zulip container.

Listing 6.4: Nginx proxy configuration

```
location / {
    proxy_pass https://127.0.0.1:8443;
    proxy_set_header Host $host;
    proxy_set_header X-Forwarded-Proto https;
}
```

SELinux booleans were updated to allow the web server to make network connections (`httpd_can_network_connect`).

6.4 Email Relay Configuration

Zulip relies heavily on email for user invitations and notifications. A local Postfix instance was configured to relay mail through Gmail's SMTP servers.

6.4.1 Postfix Relay Setup

We configured `main.cf` to relay mail to `[smtp.gmail.com]:587` using SASL authentication. A `sasl_passwd` file was created with the Gmail App Password.

6.4.2 Allowing Container Relay

Since Zulip runs in a rootless Podman container, its traffic originates from a specific subnet. Postfix was configured to trust this traffic by adding the Podman network and the host IP to `mynetworks`.

Listing 6.5: Trusting Podman networks in Postfix

```
sudo postconf -e "mynetworks = 127.0.0.0/8, [::1]/128, 10.89.0.0/24, 192.168.56.17/32"
sudo systemctl restart postfix
```

6.5 Validation

The setup was validated by:

1. Generating the realm creation link via the CLI:

Listing 6.6: Generating Realm Creation Link

```
podman exec -it docker-zulip_zulip_1 sudo -u zulip \
  /home/zulip/deployments/current/manage.py generate_realm_creation_link
```

2. Verifying email delivery by sending a test email from within the container and confirming its arrival in the external inbox.

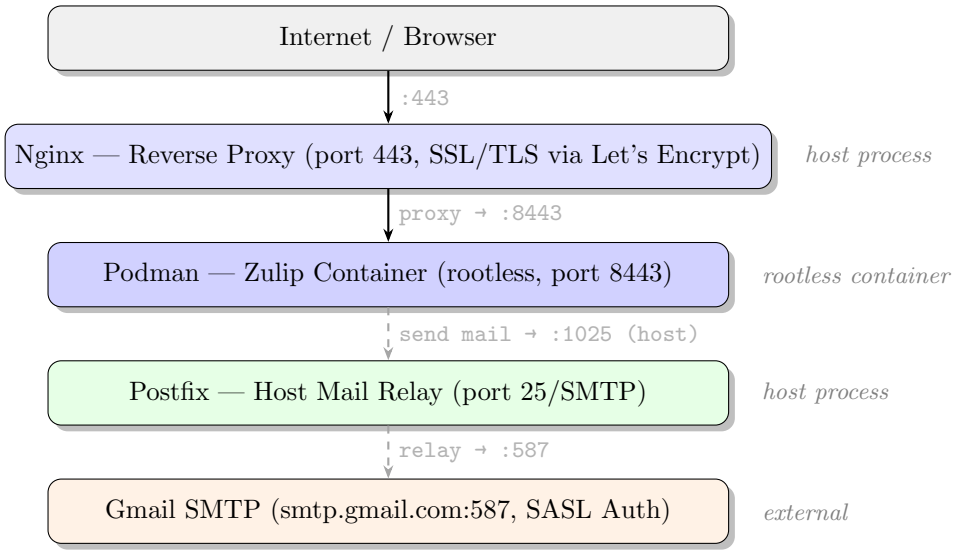

Figure 6.1: Zulip deployment stack: Nginx reverse proxy, rootless Podman container, and Postfix mail relay

6.6 Challenges and Troubleshooting

6.6.1 SELinux Blocking Nginx Proxy Connections

After configuring Nginx as a reverse proxy, requests to the domain returned a `502 Bad Gateway` error. The Nginx error log showed "Permission denied" when attempting to connect to `127.0.0.1:8443`. The cause was SELinux: by default, SELinux prevents the `httpd` process

(which Nginx runs as) from making outbound network connections. The fix was to enable the `httpd_can_network_connect` SELinux boolean:

Listing 6.7: Enabling SELinux boolean for Nginx proxy

```
sudo setsebool -P httpd_can_network_connect 1
```

6.6.2 Postfix Rejecting Mail from Podman Containers

Zulip's email sending failed with a "Relay access denied" error in the Postfix logs. The root cause is that rootless Podman assigns containers IPs from a private subnet (e.g., `10.89.0.0/24`), and traffic from the container appears on the host from this subnet. Postfix's default `mynetworks` only trusts `127.0.0.0/8`, so it rejected the relay request. The fix was to explicitly add the Podman network subnet and the host's own IP to `mynetworks`:

Listing 6.8: Trusting Podman network in Postfix

```
sudo postconf -e "mynetworks = 127.0.0.0/8, [::1]/128, 10.89.0.0/24, 192.168.56.17/32"
sudo systemctl restart postfix
```

6.6.3 File Descriptor Limits Causing Container Startup Failure

The Zulip container failed to start with an error indicating it could not open enough file descriptors. Zulip is a high-concurrency application and requires a very high `nofile` limit (up to 1,048,576). The default system limits were far too low. The fix was to add the required limits to `/etc/security/limits.conf` and log out and back in for the changes to take effect before restarting the containers.

Chapter 7

Discourse Setup and Anti-Spam Hardening

This chapter details the deployment of the Discourse discussion platform using Docker and the implementation of robust anti-spam and security measures to protect the community. The setup focuses on hardening the instance against unauthorized access and automated bots.

7.1 Installation and Initial Configuration

The deployment logic relies on the official `discourse_docker` repository, which manages the application and its dependencies (PostgreSQL, Redis) within containers [5]. Docker was used here rather than Podman because the `discourse_docker` launcher script manages the entire application lifecycle — building the image, starting/stopping the container, and running upgrades — and it explicitly requires Docker. Using Podman would have required maintaining a fork of the launcher script, which was not justified.

7.1.1 Prerequisites

The host system (Red Hat based) was prepared by installing Docker Engine and essential tools [9].

Listing 7.1: Installing Docker Engine

```
sudo dnf config-manager --add-repo https://download.docker.com/linux/rhel/docker-ce.
    ↪ repo
sudo dnf install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-
    ↪ compose-plugin -y
sudo systemctl enable --now docker
```

7.1.2 Discourse Deployment

The installation process was automated using the `discourse-setup` [6] script, which creates the `containers/app.yml` configuration file based on user input.

Listing 7.2: Launching Discourse setup

```
git clone https://github.com/discourse/discourse_docker.git /var/discourse
cd /var/discourse
./discourse-setup
```

Critical parameters such as the hostname, SMTP server credentials, and administrator email were configured during this step.

7.2 Security and Anti-Spam Measures

To ensure the integrity of the discussion platform, several security layers were implemented via the administration panel.

7.2.1 Trust Levels and Access Control

We configured strict default trust levels to limit the privileges of new users immediately after registration. A layered anti-spam strategy was chosen because no single measure is sufficient: hCaptcha stops automated bots, admin approval stops determined human spammers who pass the CAPTCHA, and 2FA protects legitimate accounts from being compromised.

- **Default Trust Level**: Set to `0: new user` to restrict capabilities (e.g., posting links/images) until trust is earned. This is a significant deterrent for spammers, whose primary goal is to post links.
- **Approval Policy**: enabled `Must approve users` to require administrative review for every new account activation.

7.2.2 Two-Factor Authentication (2FA)

Two-factor authentication was enforced globally to secure user accounts.

- **Enforcement**: `Enforce second factor` was set to `all`, mandating 2FA for administrators, moderators, and standard users.
- **External Auth**: `Enforce second factor on external auth` was enabled to ensure security even when logging in via SSO providers.

7.2.3 Bot Protection with hCaptcha

To prevent automated spam registrations, hCaptcha was integrated into the sign-up flow.

1. An account was created on `hcaptcha.com` to generate a Site Key and Secret Key.
2. The keys were configured in the Discourse Hcaptcha plugin settings.

This ensures that every new registration request is verified as human.

7.3 Backup and Recovery Strategy

Reliable backup and restore procedures were established to safeguard community data.

7.3.1 Automated Backups

Discourse was configured to perform daily automated backups to local storage. These backups include the database dump and uploaded files, stored in `/var/discourse/shared/standalone/backups/default`. Local storage provides a fast recovery path for the most common failure scenario (data corruption or accidental deletion), while the `launcher` tool's manual backup command provides on-demand snapshots for pre-migration use.

7.3.2 Manual Backup and Restore

Administrative commands via the `launcher` tool allow for manual disaster recovery operations.

Listing 7.3: Creating a manual backup

```
cd /var/discourse
./launcher enter app
discourse backup
```

To migrate the forum to a new server, the backup file can be transferred and restored into a fresh instance.

Listing 7.4: Restoring from backup

```
discourse enable_restore
discourse restore <backup_filename>.tar.gz
```

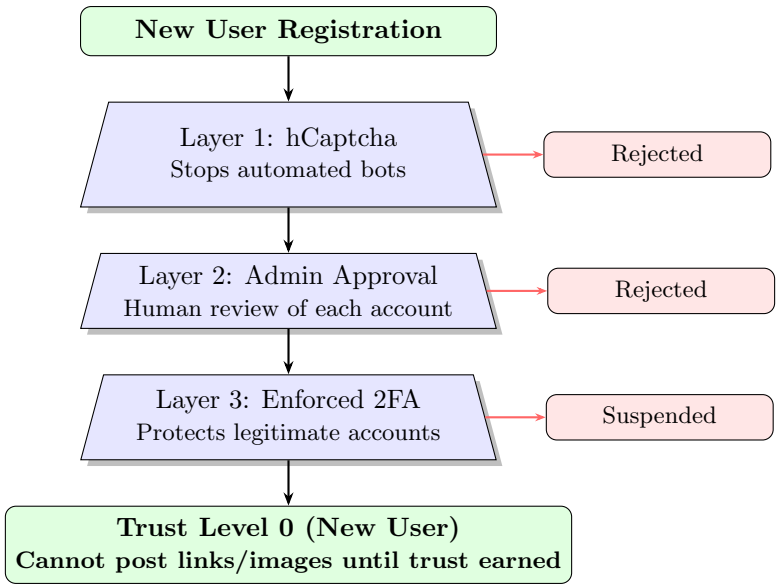

Figure 7.1: Discourse anti-spam registration funnel: three layered gates before a user gains access

7.4 Challenges and Troubleshooting

7.4.1 SMTP Configuration Failure During Setup

The `discourse-setup` script requires valid SMTP credentials to send the initial administrator confirmation email. If the SMTP settings are incorrect, the setup script completes but the admin account cannot be activated. This was resolved by first verifying the SMTP credentials independently (using `swaks` or a similar tool) before running the setup script, and then using the `discourse-setup` script's ability to re-run and update the `app.yml` configuration.

7.4.2 Container Rebuild Required After Configuration Changes

Changes to the Discourse configuration file `containers/app.yml` do not take effect by simply restarting the container. Discourse requires a full container rebuild (`./launcher rebuild app`) to apply any configuration changes. This is because the configuration is baked into the Docker

image at build time, not read at runtime. Failing to rebuild after a configuration change is a common source of confusion and results in the old configuration remaining active.

7.4.3 hCaptcha Plugin Not Available in Default Installation

The hCaptcha integration is provided by a Discourse plugin that must be added to `app.yml` before the container is built. Attempting to configure hCaptcha through the admin panel without the plugin installed results in the settings being absent from the interface. The fix was to add the plugin's GitHub URL to the `hooks.after_code.exec` section of `app.yml` and rebuild the container.

Bibliography

[1] *Ansible playbook for creating AWS ALB*. URL: https://github.com/kjswaruph/fossee-daily-progress/blob/main/ova-to-ami/alb.create.yml.

[2] *Ansible Playbook for creating AWS EC2*. URL: https://github.com/kjswaruph/fossee-daily-progress/blob/main/ova-to-ami/ec2.create.yml.

[3] *Ansible playbook for creating AWS Security Group*. URL: https://github.com/kjswaruph/fossee-daily-progress/blob/main/ova-to-ami/sg.create.yml.

[4] *Ansible Playbook for creating keypair*. URL: https://github.com/kjswaruph/fossee-daily-progress/blob/main/ova-to-ami/keypair.create.yml.

[5] *Discourse Cloud Install Guide*. URL: https://github.com/discourse/discourse/blob/main/docs/INSTALL-cloud.md.

[6] *Discourse Setup Script*. URL: https://github.com/discourse/discourse_docker/blob/main/discourse-setup.

[7] *drupal_keycloak_sso Script*. Automation script for Drupal SSO configuration. URL: https://github.com/kjswaruph/fossee-daily-progress/blob/main/sso-automation/drupal_keycloak_sso.

[8] *init_sites script*. Script for initializing Drupal sites with Podman. URL: https://github.com/kjswaruph/fossee-daily-progress/blob/main/openplc/init_sites.

[9] *Install Docker Engine on RHEL*. URL: https://docs.docker.com/engine/install/rhel/.

[10] *Keycloak Server Administration Guide*. URL: https://www.keycloak.org/docs/latest/server_admin/index.html.

[11] *keycloak_client_setup Script*. Automation script for Keycloak OIDC client configuration. URL: https://github.com/kjswaruph/fossee-daily-progress/blob/main/sso-automation/keycloak_client_setup.

[12] *Migrating from Mailman 2.1 to Mailman 3*. URL: https://docs.mailman3.org/en/latest/migration.html.

[13] *Mozilla Django OIDC Guide*. URL: https://mozilla-django-oidc.readthedocs.io/en/stable/installation.html#quick-start.

[14] *Online Test Production Setup*. URL: https://github.com/FOSSEE/online_test/blob/master/README_production.rst.

[15] *VM Import/Export Processes*. URL: https://docs.aws.amazon.com/vm-import/latest/userguide/import-export-processes.html.