CS6530
APPLIED CRYPTOGRAPHY

ASSIGNMENT 4 REPORT

Name: **Rudra Pratap Singh**
Roll Number: **EE22B171**

Introduction

The objective of this assignment is to gain practical experience in building a complete post-quantum public key infrastructure (PKI) using quantum-safe signature algorithms. With the ongoing standardization of post-quantum cryptography by NIST, traditional RSA and ECC are no longer sufficient for long-term security. This assignment focuses on the use of the ML-DSA (formerly Dilithium) family of algorithms, which are based on lattice cryptography and selected by NIST for digital signatures in a post-quantum world.

In the first part of the assignment, we create a Post-Quantum Root Certificate Authority (Root CA) and a Subordinate Certificate Authority (Sub CA) using EJBCA. Both CAs use ML-DSA keys at different security levels (ML-DSA-87 for the Root CA and ML-DSA-65 for the Sub CA), demonstrating how a full PQC hierarchy can be constructed.
In the second part, we use SignServer to generate a Certificate Signing Request (CSR), issue a signing certificate from the PQC Sub CA, import it into SignServer, and finally perform post-quantum CMS signing and verification. A dedicated Keyfactor PQC Verifier tool is used to validate the ML-DSA signatures, since standard OpenSSL builds do not yet support these algorithms.

Overall, this assignment provides hands-on familiarity with post-quantum PKI components and demonstrates how PQC algorithms can be integrated into real-world certificate issuance, signing, and verification workflows.

Environment Setup

To begin the implementation, the environment was prepared using the EJBCA Community Edition Docker image (version 8.0.0). This setup ensured support for post-quantum algorithms such as Dilithium, which are required for constructing a PQC-enabled Root CA and Sub CA hierarchy. The environment setup involved the following major steps:

- **Deploying EJBCA using Docker:** The EJBCA CE 8.0.0 image was pulled and executed on Ubuntu. This container initializes the ManagementCA and exposes the Admin Web interface.
- **Reviewing initialization logs:** During startup, the container logs displayed the automatic creation of the default `ManagementCA`, along with the SuperAdmin enrollment URL and temporary password.
- **Performing SuperAdmin enrollment:** Using the enrollment URL, the SuperAdmin certificate was enrolled via the Web interface. A PKCS#12 or PEM file was downloaded for browser-based certificate authentication.
- **Importing the SuperAdmin certificate:** The downloaded credential was imported into the browser, enabling authenticated access to the EJBCA Admin Web.
- **Verifying successful initialization:** Final startup logs confirmed the initialization of the system and completion of the SuperAdmin setup process.

With the environment fully initialized and administrative access established, the system was ready for defining certificate profiles, creating PQC crypto tokens, and constructing the hierarchical PQC CA infrastructure in subsequent steps.

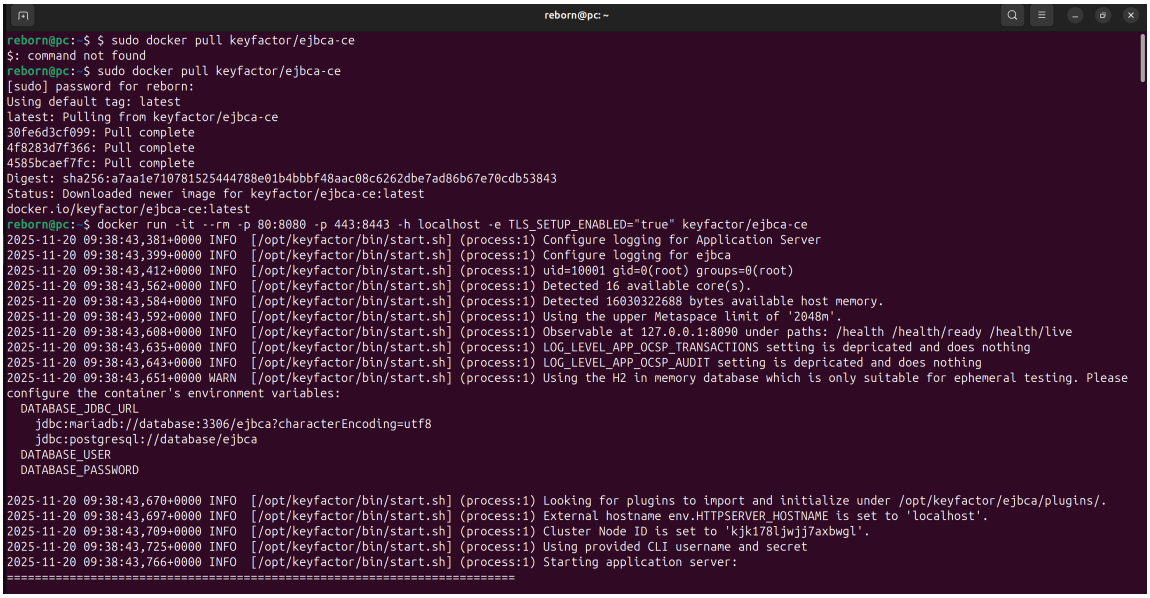

Figure 1: Pulling and running the EJBCA Community Docker image (version 8.0.0).

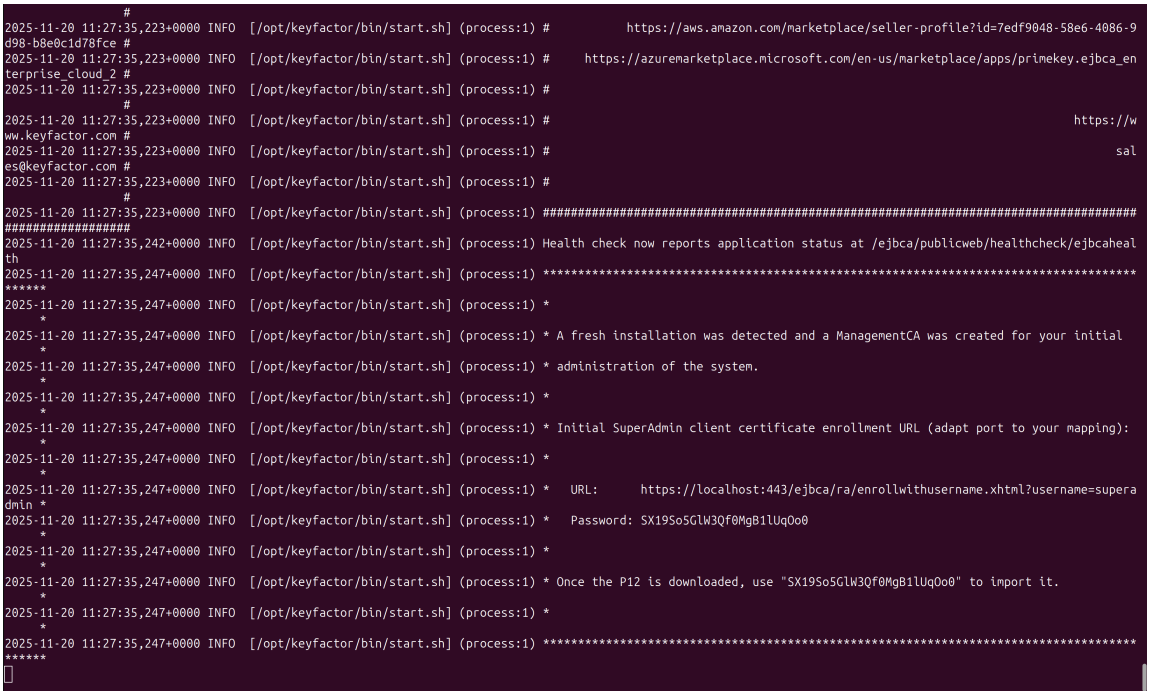

Figure 2: EJBCA container startup logs showing the generated SuperAdmin enrollment URL and temporary password.

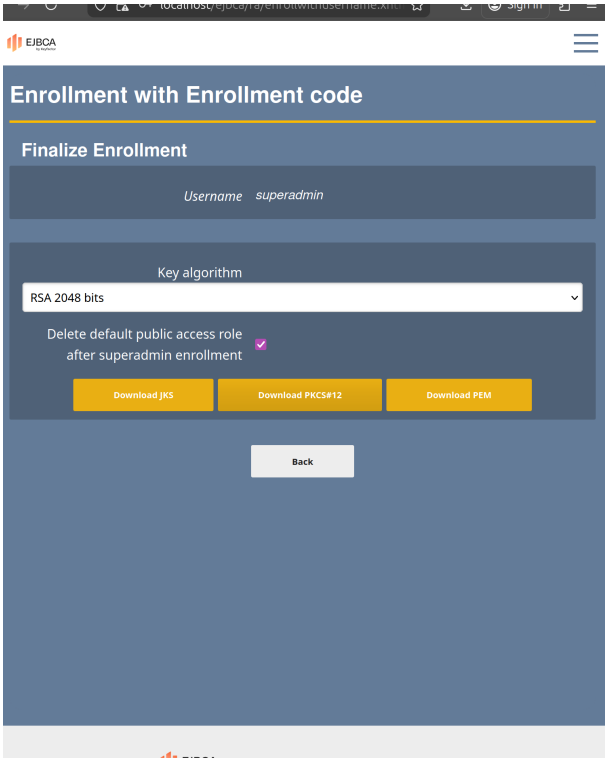

Figure 3: SuperAdmin enrollment performed using the Web interface, with certificate download options.

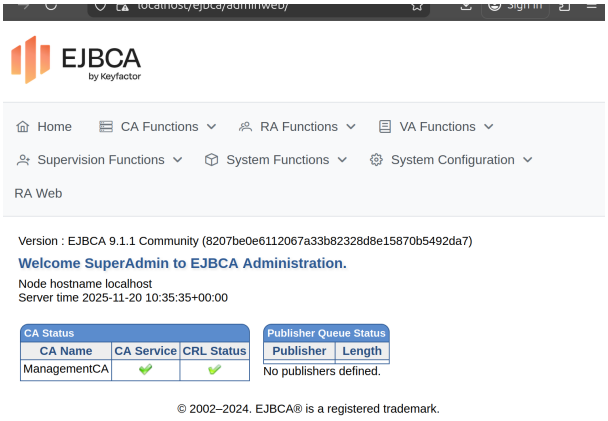

Figure 4: Successful login to the EJBCA Admin Web using the imported SuperAdmin certificate.

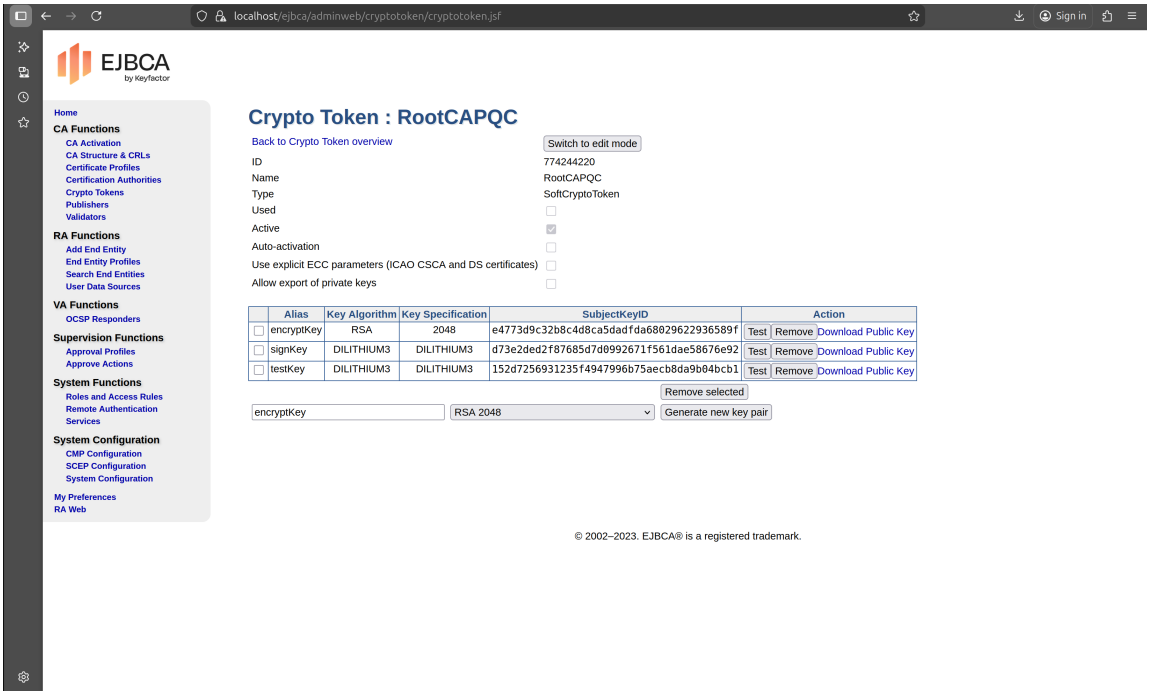

Figure 5: Final initialization logs confirming SuperAdmin setup and ManagementCA initialization.

Step 1 — Creating Certificate Profiles

In this step, certificate profiles were created for both the PQC Root CA and the PQC Sub CA. EJBCA Community Edition 8.0.0 includes support for post-quantum algorithms such as Dilithium2 and Dilithium3, which are required for building a quantum-safe CA hierarchy.

Root CA Certificate Profile

A new Root CA certificate profile (`MyPQCRootCAProfile`) was created by cloning the default `ROOTCA` profile. The profile was modified to:

- set the profile type to **Root CA**,
- enable the post-quantum key algorithm **Dilithium3**,
- inherit the signature algorithm from the issuing CA,
- set the certificate validity to **30 years**,
- clear LDAP DN order, as recommended for PQC usage.

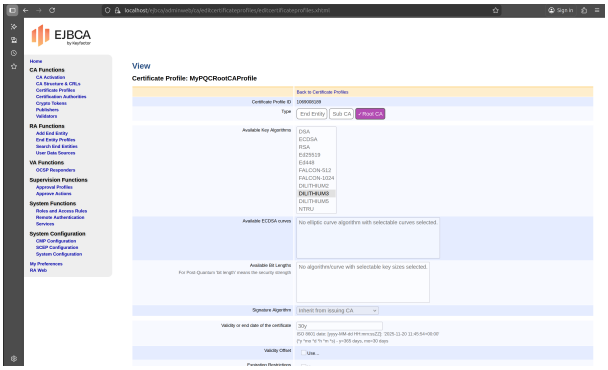

Figure 6: Root CA certificate profile configured to use Dilithium3.

Sub CA Certificate Profile

Similarly, a Sub CA certificate profile (`MyPQCSubCAProfile`) was created by cloning the default `SUBCA` profile. The following updates were applied:

- set the profile type to **Sub CA**,
- select the post-quantum key algorithm **Dilithium2**,
- keep the signature algorithm inherited from the issuing CA,
- set certificate validity to **15 years**,
- ensure LDAP DN order is cleared.

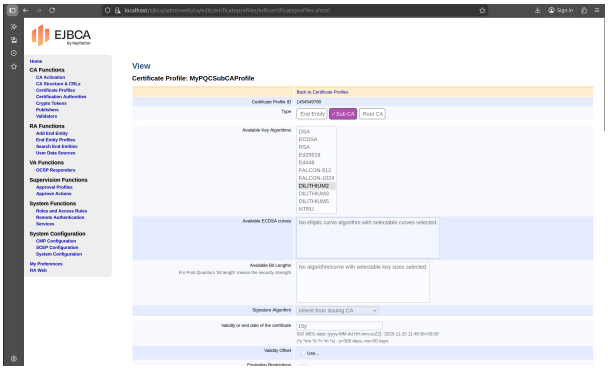

Figure 7: Sub CA certificate profile configured to use Dilithium2.

With the certificate profiles prepared for PQC-capable Root and Sub CAs, the next step is to generate the crypto tokens and key material required for each CA.

Step 2 — Creating Crypto Tokens for the Root CA and Sub CA

After configuring the certificate profiles, crypto tokens were created to hold the post-quantum key pairs required by the Root CA and the Sub CA. EJBCA 8.0.0 supports soft crypto tokens with PQC algorithms, enabling the use of Dilithium-based signing keys.

Root CA Crypto Token

A new crypto token named `RootCAPQC` was created for the Root CA. The token was configured with auto-activation and then populated with the following keys:

- **signKey** — Dilithium3 key for Root CA certificate signing.
- **testKey** — Dilithium3 test key (required by EJBCA for CA operations).
- **encryptKey** — RSA 2048 internal encryption key (required for CA keystore compatibility).

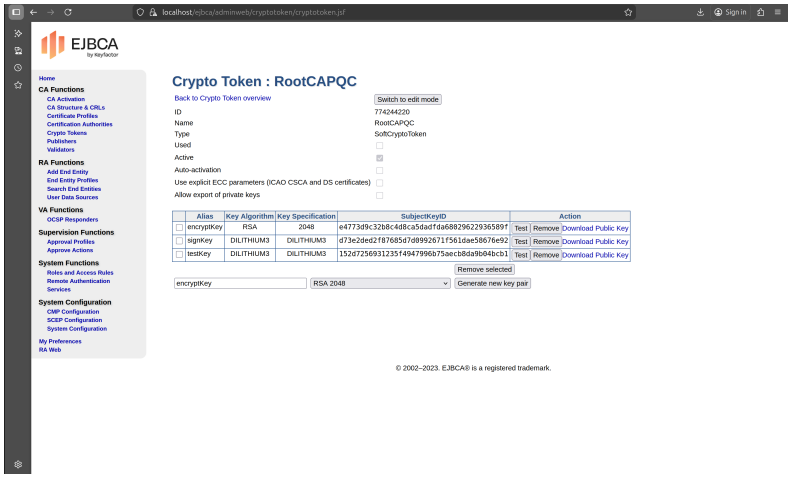

Figure 8: Root CA crypto token containing Dilithium3 signing and test keys.

Sub CA Crypto Token

Next, a crypto token named `SubCAPQC` was created for the Sub CA. This token was also auto-activated and initialized with:

- **signKey** — Dilithium2 key for Sub CA certificate signing.
- **testKey** — Dilithium2 test key.
- **encryptKey** — RSA 2048 internal encryption key.

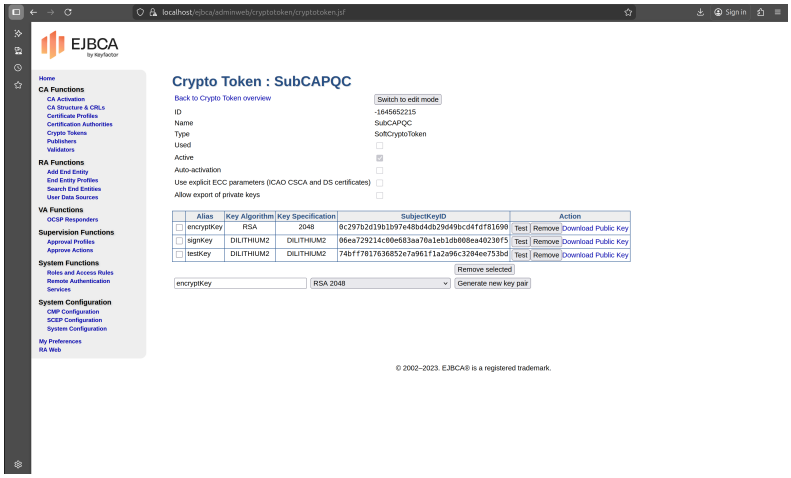

Figure 9: Sub CA crypto token containing Dilithium2 signing and test keys.

With both crypto tokens created and populated with the required PQC key material, the system was ready for generating the Root CA and Sub CA certificates in the next step.

Step 3 - Create Root CA

After configuring the crypto token for the Root CA, the next step was to create the `PQCRootCA`. Using the EJBCA Admin Web interface, a new CA was created and linked to the previously generated `RootCAPQC` crypto token. The signing algorithm for the Root CA was configured as `DILITHIUM3`, consistent with the certificate profile settings defined earlier.

All certificate data fields such as the Subject DN, certificate profile, validity period, and default key mappings were filled in according to the post-quantum hierarchy requirements. Figure 10 shows the Root CA creation page with the selected PQC parameters.

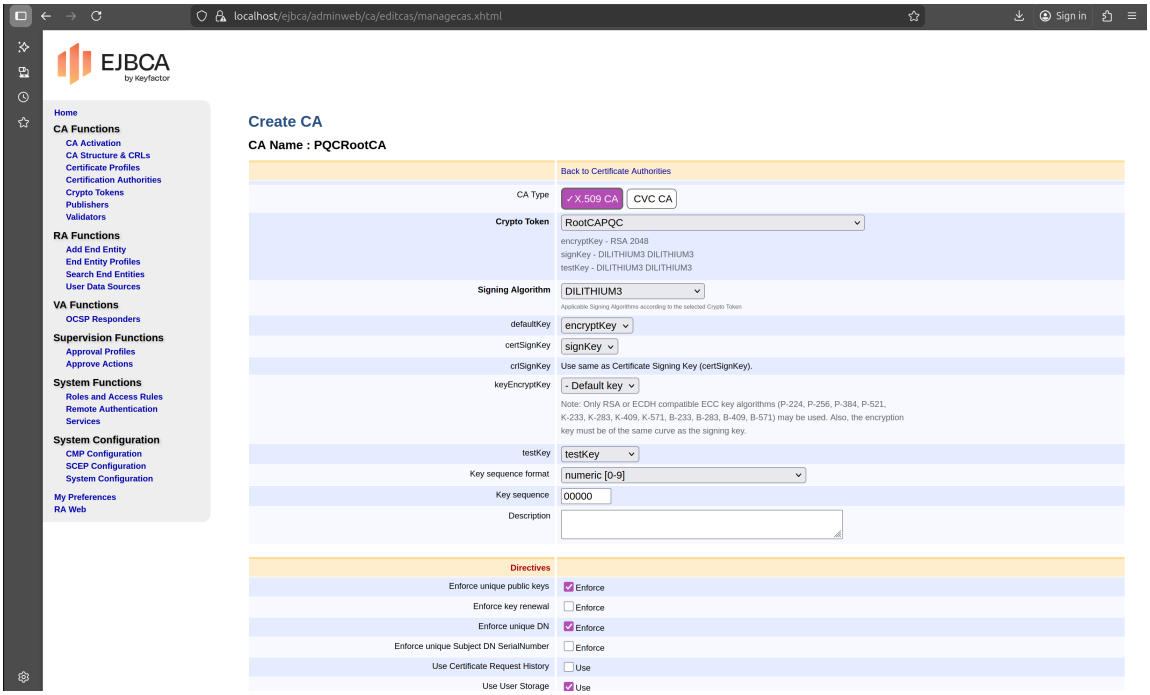

Figure 10: Configuration of the PQCRootCA using the Dilithium3 signing key and RootCAPQC crypto token.

Step 4 - Create Sub CA

After creating the Root CA, a subordinate CA named `PQCSubCA` was created to complete the two-tier post-quantum PKI hierarchy. This CA was linked to the `SubCAPQC` crypto token, and its signing algorithm was set to `DILITHIUM2`, following the Sub CA certificate profile.

The Sub CA was configured to be signed by the `PQCRootCA`, ensuring that the hierarchy is properly anchored at the PQC Root CA. Additional certificate parameters such as Subject DN, validity period, and certificate profile selection were set as required.

Figure 11 shows the Sub CA creation page with the appropriate Dilithium-based cryptographic settings.

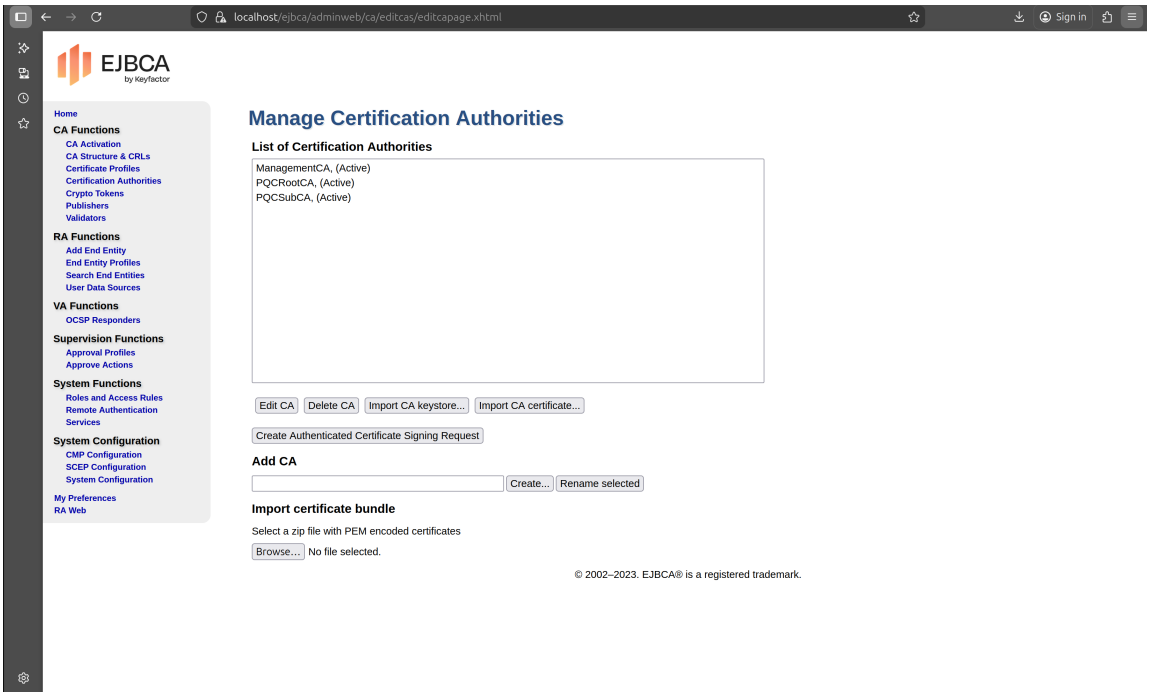

Figure 11: Configuration of the PQCSubCA signed by the PQCRootCA, using Dilithium2 keys.

Step 5 – View and Download CA Certificates

After the Root CA and Sub CA were successfully created, their certificates were inspected and exported using the `CA Structure & CRLs` section of the EJBCA Admin Web. This interface allows verification of each CA's metadata, validity period, public key algorithm, and issued certificate chain.

Figure 12 shows the Root CA certificate details. The certificate is self-signed and uses the Dilithium3 algorithm, consistent with the configuration selected during CA creation. Key usage fields confirm its role for certificate signing and CRL signing.

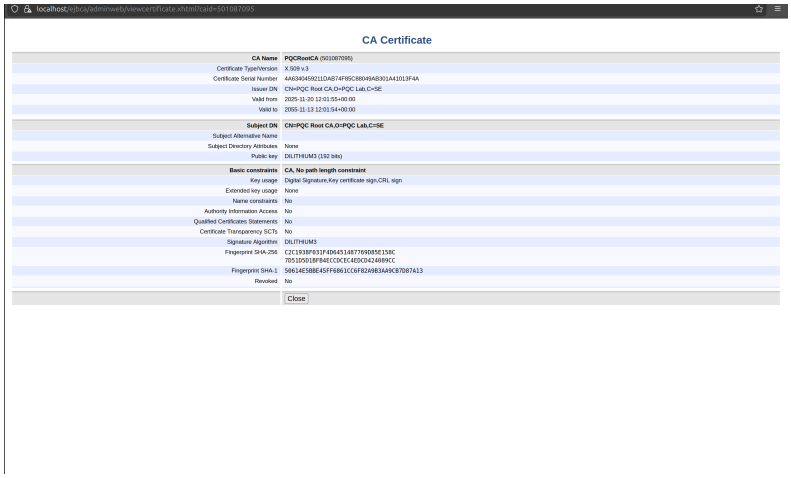

Figure 12: Root CA certificate details showing Dilithium3 public key and CA attributes.

Next, the Sub CA certificate was viewed. As shown in Figure 13, the Sub CA is correctly signed by the Root CA and uses the Dilithium2 algorithm. Its key usages match a subordinate CA, enabling certificate signing under the hierarchy.

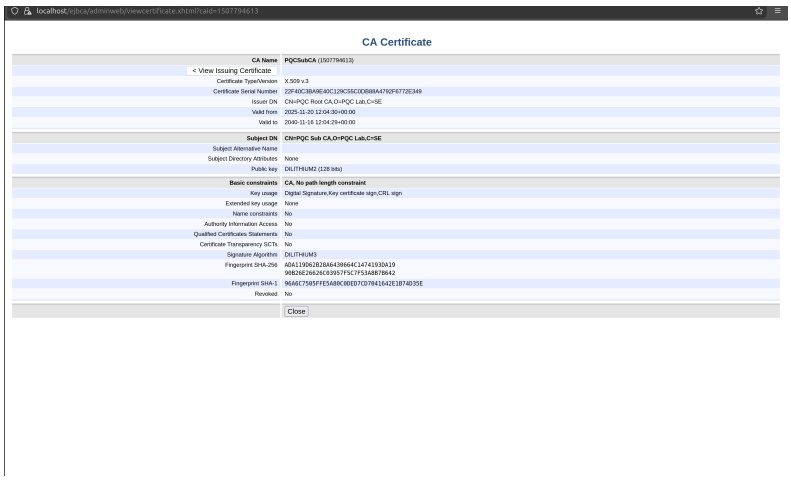

Figure 13: Sub CA certificate details signed by the Root CA, using Dilithium2.

Finally, the full CA hierarchy—ManagementCA, PQCRootCA, and PQCSubCA—was reviewed in the CA Structure page as shown in Figure 14. This page also provides direct download options for each certificate in PEM, JKS, or browser formats, which are required for configuring the SignServer workflows.

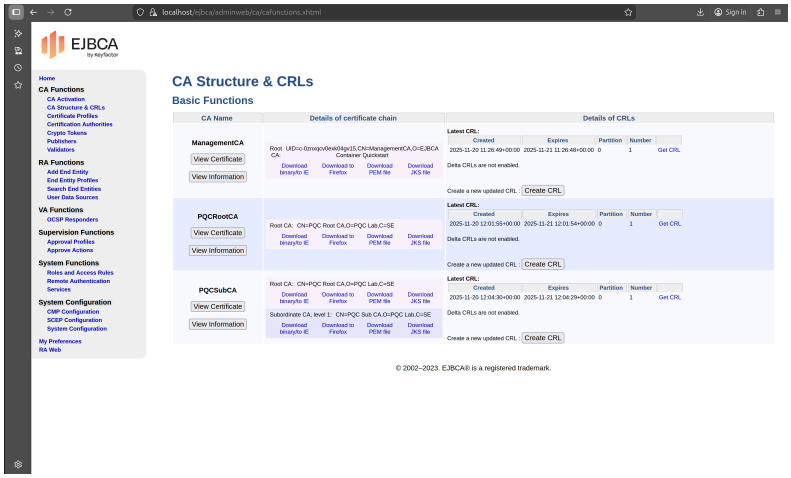

Figure 14: CA Structure overview showing the active Root and Sub CAs and providing download options for PEM and JKS files.

With the CA hierarchy verified and the PEM certificates downloaded, the environment was fully prepared for configuring post-quantum document signing workflows in SignServer.

SignServer Environment Setup

To perform post-quantum signing using ML-DSA, the SignServer Community container (version 7.1) was deployed and configured to trust the `ManagementCA` certificate. In the previous task, EJBCA 8.0.0 was used because it still included the Dilithium algorithms required for that part of the tutorial. However, EJBCA 8.0.0 does not support ML-DSA, and the SignServer PQ tutorial specifically requires ML-DSA. Therefore, for this task the **latest EJBCA version** was deployed, as the latest release has replaced Dilithium with the NIST-selected ML-DSA algorithms needed for SignServer.

The `ManagementCA.cacert.pem` file from the latest EJBCA deployment was mounted into the SignServer container to enable TLS client certificate authentication for the AdminWeb interface. The corresponding `SuperAdmin.p12` file was imported into Firefox, which was then used to authenticate securely to the SignServer AdminWeb.

The setup follows the official SignServer tutorial but adapted to match the updated toolchain:

- **Latest EJBCA release** for ML-DSA certificate issuance (this task),
- **SignServer Community 7.1** for ML-DSA signing operations,
- **ManagementCA client certificate imported into Firefox** for AdminWeb authentication.

After pulling the SignServer 7.1 image, the container was started with the ManagementCA certificate mounted. Once initialized, the SignServer Public Web, Client Web, and Admin Web interfaces became available. AdminWeb access required TLS client certificate authentication, which was completed using Firefox with the imported ManagementCA P12 file. With authentication successful, the system was ready for creating ML-DSA signer workers. .

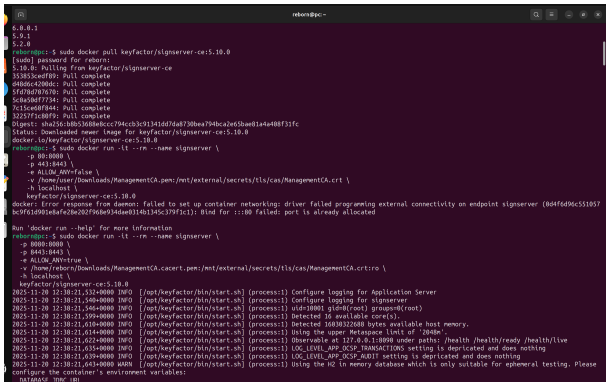

Figure 15: Pulling and running the SignServer Community container (version 5.10.0) with ManagementCA certificate mounted.

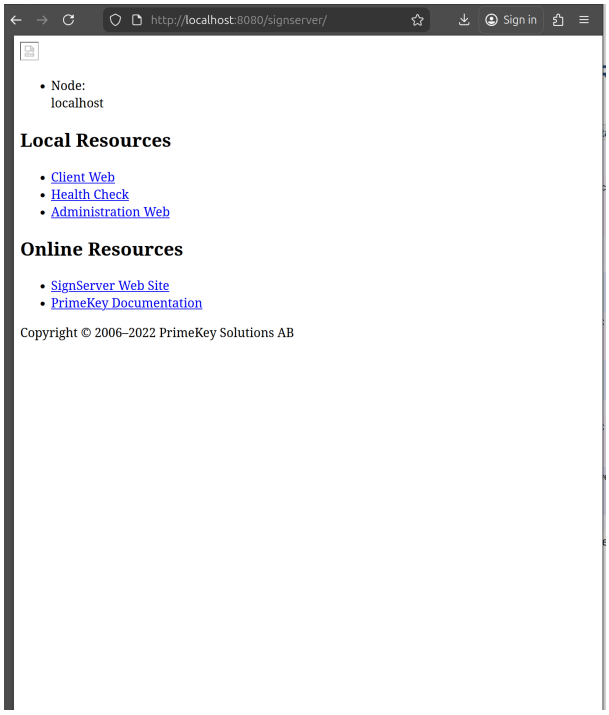

Figure 16: SignServer Public Web homepage confirming container startup.

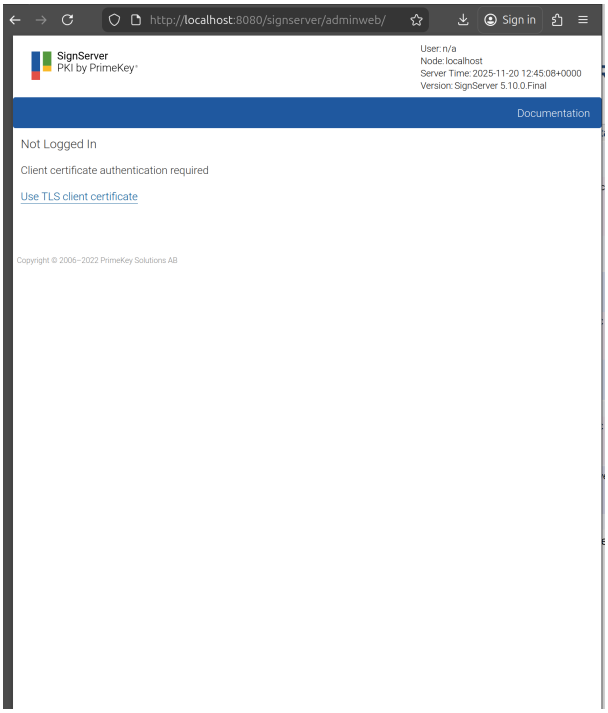

Figure 17: SignServer AdminWeb prompting for TLS client certificate authentication.

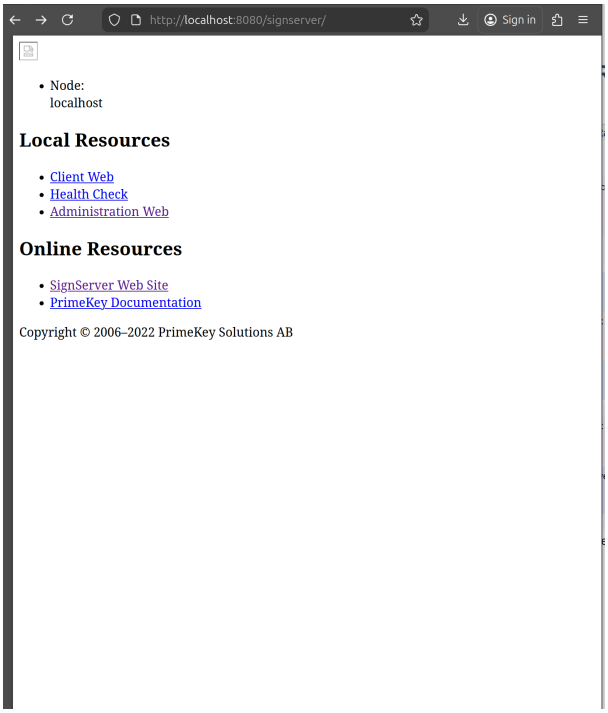

Figure 18: SignServer Public Web (post-load) showing Client, Health Check, and Administration links.

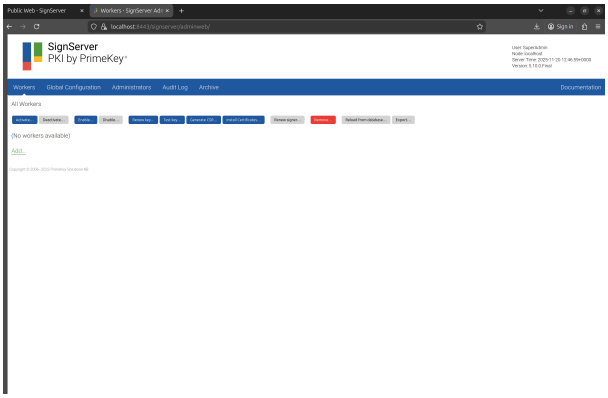

Figure 19: Successful SignServer AdminWeb login using the ManagementCA administrator certificate.

Step 1 — Create Signing Key and CSR in SignServer

This step describes the creation of a post-quantum ML-DSA signing key inside SignServer, followed by the generation of a Certificate Signing Request (CSR) that will later be issued by the PQC Sub CA in EJBCA. All actions were performed on **SignServer Community 7.1.1** using the **ManagementCA client certificate** imported into Firefox for TLS-authenticated administrator access.

1.1 Create Crypto Token

A new crypto token was created to store ML-DSA private keys inside SignServer's internal database. From the AdminWeb interface, a worker was added using the **keystore-crypto.properties** template. The configuration was updated as follows:

- `WORKERGENID1.KEYSTORETYPE = INTERNAL`
- The `KEYSTOREPATH` line was removed

After applying the configuration, the worker `CryptoTokenP12` appeared in OFFLINE state. It was then activated by setting a new keystore password.

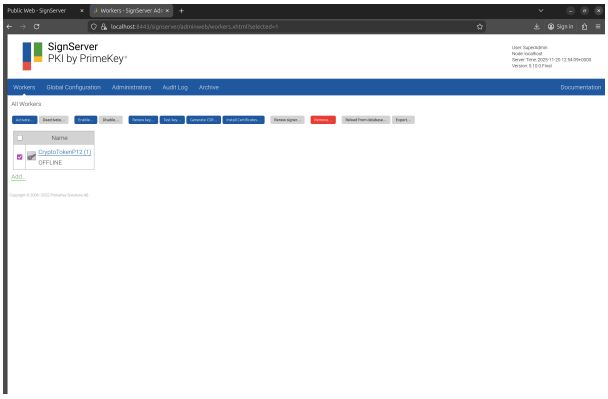

Figure 20: CryptoTokenP12 created from template and listed as OFFLINE before activation.

1.2 Generate Signing Key and Create Signing Worker

After activating the CryptoTokenP12, an ML-DSA signing key was created and immediately used to configure the CMS signing worker. Using the **Crypto Token** tab, a new key pair was generated with the following parameters:

- **Alias:** `CMSSigner`
- **Key Algorithm:** ML-DSA
- **Key Specification:** ML-DSA-44

Once the key was generated, the CMS signing worker was added using the `cmssigner.properties` template. The worker configuration was updated to reference the newly created key and enable ML-DSA signing:

- `WORKERGENID1.DEFAULTKEY = CMSSigner`
- `WORKERGENID1.DETACHEDSIGNATURE = TRUE`
- `WORKERGENID1.SIGNATUREALGORITHM = ML-DSA`

The worker appeared in **OFFLINE** state since it still required a certificate, which is generated in the next step.

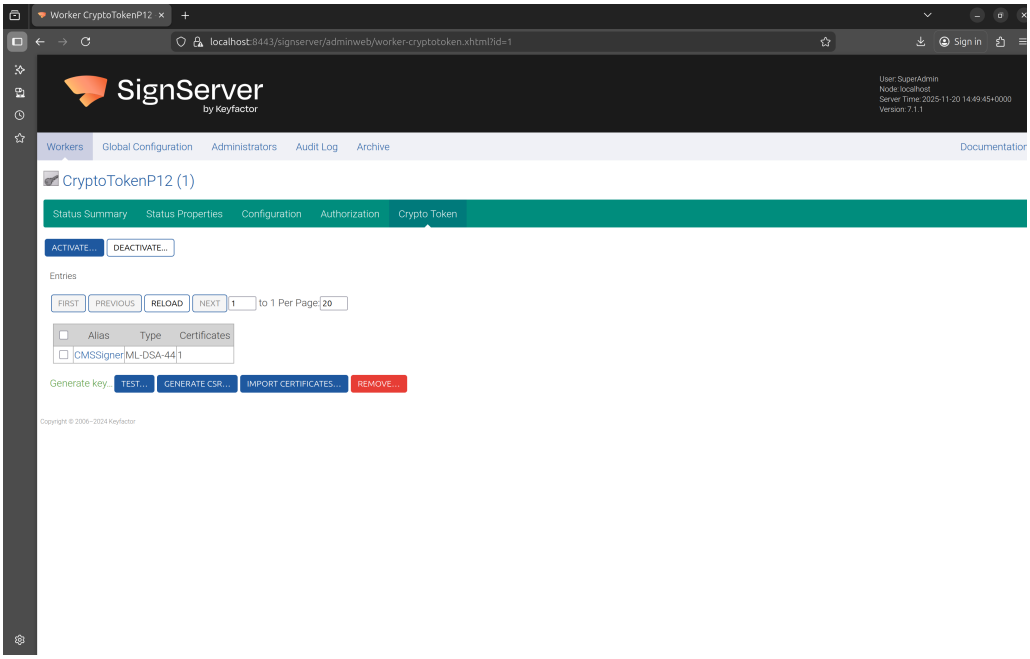

Figure 21: ML-DSA-44 key (`CMSSigner`) generated and associated with the CMS signing worker.

1.3 Generate CSR

A Certificate Signing Request (CSR) was generated for the new signing key:

- **Signature Algorithm:** ML-DSA
- **DN:** `CN=CMSSigner`

The CSR was downloaded as a PKCS#10 file `CMSSigner-CMSSigner.p10`, to be issued by the PQC Sub CA in EJBCA.

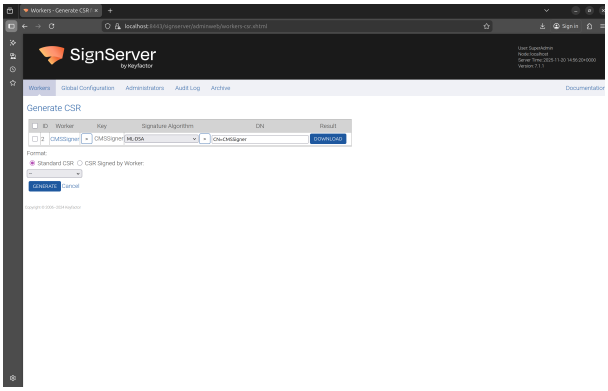

Figure 22: CSR generated for ML-DSA signing key using DN = CN=CMSSigner.

Step2 - Issue Signing Certificate

To issue the signing certificate, the CSR generated in SignServer was uploaded in the EJBCA RA Web using the CMSSigner certificate profile and the Provided by user key-pair option. EJBCA then issued the ML-DSA signing certificate, which was downloaded as a PEM full chain for installation back into SignServer.

2.1 Create Certificate Profile

To prepare the signing environment, a new certificate profile was created in EJBCA for issuing ML-DSA–based CMS signing certificates. In the EJBCA Admin Web, the Certificate Profiles section was opened, and a new profile named `CMSSigner` was added to the list. After creation, the profile was edited to align with the required constraints for post-quantum signing. The profile was configured to use the `ML-DSA-44` algorithm as the available key algorithm. The certificate validity period was set to one year. Under Key Usage, the options *Non-repudiation* and *Key Encipherment* were disabled so that only *Digital Signature* remained enabled. For Extended Key Usage, the setting was left at `Use` to avoid adding any extended key usage constraints. All remaining parameters were kept at their default values, and the profile was saved.

After saving, the `CMSSigner` profile appeared in the certificate profile list, confirming that the configuration was successfully completed and ready to be used for issuing the signing certificate in later steps.

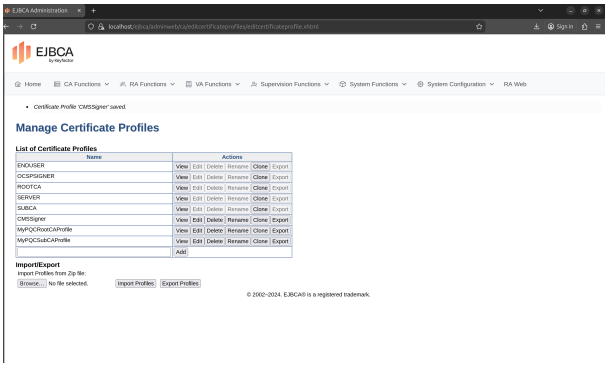

Figure 23: The `CMSSigner` certificate profile created and displayed in the EJBCA Certificate Profiles list.

2.2 Create End Entity Profile

To issue the ML-DSA certificate for the SignServer CMSSigner worker, an End Entity Profile was created in EJBCA. This profile defines how the signer's certificate request will be handled.

- In the EJBCA Admin Web, navigate to **RA Functions → End Entity Profiles**.
- Add a new end entity profile named `CMSSigner` and click **Add profile**.
- Select the newly created profile and click **Edit End Entity Profile**.

The following fields were configured to ensure that the Sub CA issues certificates using the correct ML-DSA signing profile:

- **End Entity E-Mail:** Cleared.
- **Default Certificate Profile:** `CMSSigner`.
- **Available Certificate Profiles:** `CMSSigner`.
- **Default CA:** `PQCSubCA`.
- **Available CAs:** `PQCSubCA`.
- **Available Tokens:** `User Generated`.

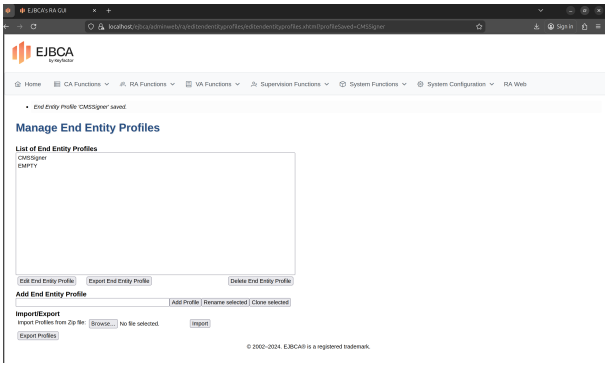

Figure 24: End Entity Profile configuration for the CMSSigner certificate.

Issue Certificate

The Certificate Signing Request (CSR) was submitted to the EJBCA RA Web to obtain the signing certificate. In the RA Web *Make Request* page the certificate type `CMSSigner` was selected and the previously generated CSR was uploaded. The uploaded CSR was recognized to use the `ML-DSA-44` key algorithm, and the subject Distinguished Name fields (including `CN = CMSSigner`) were populated accordingly. The key-pair generation method was set to *Provided by user*. After verifying the CSR details and algorithm, the request was submitted to the CA for issuance of the ML-DSA–based signing certificate.

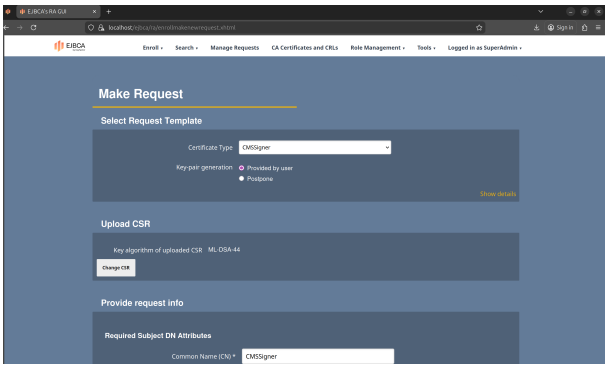

Figure 25: RA Web *Make Request* page showing the `CMSSigner` certificate type, uploaded CSR (ML-DSA-44), and populated subject DN fields prior to submitting the issuance request.

Step 3 — Download Signing CA Certificate Chain

To enable later verification of the signatures produced by the ML-DSA signer, the CA certificate chain must be downloaded from the EJBCA RA Web. Both the PQCRootCA and PQCSubCA certificates are required, as these form the trust chain for the CMSSigner certificate issued in the previous step.

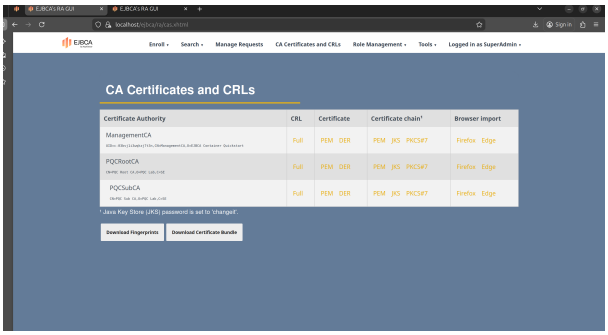

Figure 26: EJBCA RA Web interface listing the available CA certificates and their downloadable certificate chains.

Step 4 – Activate the Signing Worker in SignServer

After issuing the CMSSigner certificate in EJBCA, it was imported back into SignServer to activate the CMSSigner worker. The certificate chain was uploaded through the `Install`

`Certificates` option, after which the worker status changed to **ACTIVE**, indicating that the signing worker is now fully operational.

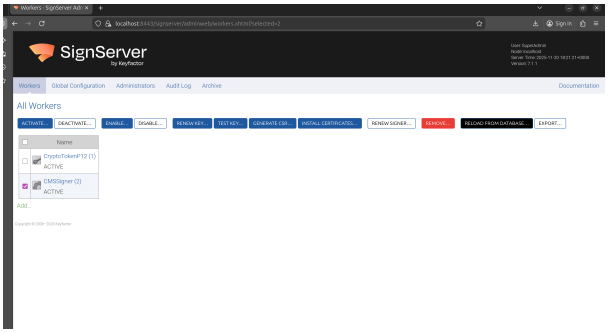

Figure 27: CMSSigner worker successfully activated after installing the issued certificate.

Step 5 - File Signing in SignServer

- Navigate to the SignServer Public Web (`https://localhost:8443/signserver/`) and open the **Client Web** interface.
- Under **Worker**, specify the signer name used earlier (`CMSSigner`).
- Upload the input file to be signed (in this example, the previously downloaded `CMSSigner.pem` file is used as sample data).
- Click **Submit** to generate and download the detached signature (.p7s file).

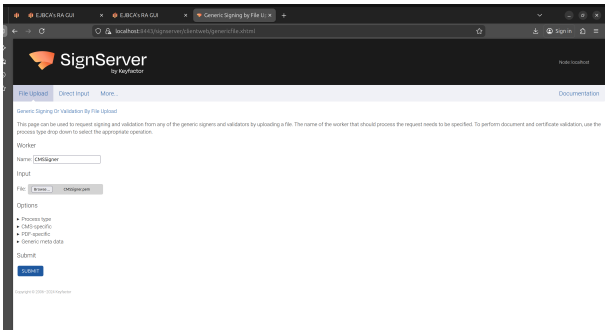

Figure 28: Submitting a file to the CMSSigner worker for ML–DSA signing in SignServer.

Figure 29: The Downloaded .p7 file.

Step 6 - Verify Signature

To verify the ML-DSA signature, the SignServer Post-Quantum Signature Verifier Tool was used. This method validates the .p7s signature using the signer certificate and the PQC trust chain (PQCSubCA + PQCRootCA), confirming successful verification.

Figure 30: Verification output from the SignServer Post-Quantum Verifier tool showing a successful ML-DSA signature verification.

Challenges Faced and How They Were Overcome

During the implementation of the PQC-enabled PKI setup using EJBCA and SignServer, I encountered several challenges related to software versions, algorithm support, and browser configuration. This section summarizes the obstacles I faced and the steps I took to resolve them.

Version Mismatch Between EJBCA and SignServer

The first major challenge arose when I followed the official Keyfactor tutorial, which demonstrated PQC support using the Dilithium and ML-DSA algorithms. Initially, I deployed the latest versions available on Docker Hub for both EJBCA Community Edition and SignServer. However, I soon discovered that algorithm support varied significantly across versions.

- The EJBCA version I used did **not** expose the ML-DSA algorithms in the certificate profile configuration (only classical algorithms such as RSA, ECDSA, etc., appeared).
- SignServer 5.x additionally showed runtime errors such as: java.security.NoSuchAlgorithmException: no such algorithm: ML-DSA for provider BC.

These issues indicated that either ML-DSA support was missing or the required cryptographic provider was not included in the versions I was running. After researching the official documentation and examining the Docker image tags, I realized that:

- Dilithium support existed in older EJBCA builds (8.0.0), but **SignServer Community Edition has never supported Dilithium**.
- ML-DSA support is available only in **EJBCA 8.x+** and **SignServer 7.1.x**.

I resolved this mismatch by switching to compatible versions:

- Deploying **EJBCA CE to latest**, which includes ML-DSA support.
- Deploying **SignServer CE 7.1.1**, which supports ML-DSA key generation and signing.

After aligning both systems to versions that support ML-DSA, the cryptographic options appeared correctly, and the errors disappeared.

TLS Client Certificate Issues in the Browser

Another recurring obstacle involved TLS client authentication when accessing the EJBCA admin interface. Initially, even after importing the SuperAdmin `.p12` certificate, the browser repeatedly displayed the message:

> *"No client certificate was presented."*

This prevented access to the AdminWeb entirely. The root cause turned out to be related to browser certificate storage:

- On Linux, Firefox stores client certificates in its own internal security module, not the OS trust store.
- I had imported the certificate incorrectly into the OS store, causing the browser to never present it during the TLS handshake.

I resolved this by explicitly importing the `SuperAdmin.p12` into Firefox's certificate manager ("Your Certificates" tab). After doing so, Firefox immediately prompted me to select the certificate when accessing `https://localhost/ejbca/adminweb`, and I could finally log in successfully.

Port Conflicts and Container Management

While deploying SignServer, I encountered a port allocation error:

```
Bind for :::80 failed:  port is already allocated
```

This was caused by running EJBCA and SignServer simultaneously on the same default ports (80 and 443). I fixed this issue by :

- mapping SignServer to alternate ports using Docker's `-p host:container` flags.

Once port conflicts were resolved, SignServer deployed cleanly and I could proceed with worker configuration.

Lessons Learned

In summary, the main challenges stemmed from cryptographic algorithm availability, version incompatibilities, and browser certificate handling. The experience reinforced several practical lessons:

- Always verify algorithm support against the documentation for the exact version in use.
- Align EJBCA and SignServer versions carefully when working with PQC features.
- Configure client certificates directly in the browser where TLS authentication is required.
- Being carefull of Docker port mappings when running multiple PKI-related services simultaneously.

Successfully overcoming these issues provided a deeper understanding of PQC toolchains, containerized PKI deployments, and certificate-based authentication workflows.