Setting up the MCP Identity Registry

Table of Contents

Prerequisites

- Java JDK 17 (A distribution of OpenJDK is recommended due to Oracle Licensing, but not a requirement)
- Maven 3.8.5+
- A running PostgreSQL database instance with version ≥ 10
  - Alternatively a running MySQL/MariaDB instance can also be used (MySQL 8.0+ and MariaDB 10.4 have been verified to work, but newer versions should also work)
- Keycloak (Only needed if not using the Docker image)
- NGINX
- Docker (if you want to run the identity registry using Docker containers)
- A SMTP server

Introduction

The MCP Identity Registry (MIR) is an identity solution targeted against actors of the maritime domain. It consists of three parts – an identity broker for logging in and federating users from other identity providers, a public key infrastructure for issuing and managing digital certificates and an API for management and access to the two first parts.

For the identity broker Keycloak has been chosen as it is open source and because it is based on OpenID Connect which is a widely used protocol for token based authentication.

This document will provide a generic guideline of how to setup and deploy the MIR on a local system.

The MCP is formerly known as the Maritime Cloud and therefore there will still be references to that in this document.

As the different components of the MIR rely on each other the configuration of one component may rely on the configuration of another component, but this will be documented as thoroughly as possible. https://management.maritimeconnectivity.net/

Complementary documentation of setup can be found at https://github.com/maritimeconnectivity/IdentityRegistry, https://github.com/maritimeconnectivity/MCP-PKI and https://github.com/maritimeconnectivity/MCPKeycloakSpi

For easier management of the MIR API it is advised to also setup the MCP management portal. The code for this can be found at https://github.com/maritimeconnectivity/MCP-Portal. A running instance of this can be found at https://management.maritimeconnectivity.net/.

Setting up the PKI

A part of the MIR is based on public key certificates. For this a custom PKI has been developed to issue and manage certificates for stakeholders.

To set up the PKI you must get the source code for it from https://github.com/maritimeconnectivity/MCP-PKI/archive/master.zip or 'git clone https://github.com/maritimeconnectivity/MCP-PKI.git'.

Alternatively you can also get the needed executable from https://github.com/maritimeconnectivity/MCP-PKI/releases/download/v1.1.0/mcp-pki-cli-1.1.0-jar-with-dependencies.jar. Please note that this executable may not be the same as the current HEAD of the master Git branch, and therefore some of the commands that are described below might not work.

Please note that the name of the executable differs between the different versions.

After you have done this you can build the project from the directory storing the code using the following command:

mvn clean install

After this has finished there should now be a new folder called **target** under **mcp-pki-cli**. This folder contains the resulting artifacts of the build process, including the jar file **mcp-pki-cli-1.1.0-**

SNAPSHOT-jar-with-dependencies.jar which is the artifact that is going to be used to generate the root certificate authority (CA) and sub-CAs for the PKI.

Generate root and intermediate CA

The next step is to generate the root CA. This can be done using the following command where you should of course replace the different fields with information relevant to your setup. For example here `root` will be the alias for the root CA, where you might want it to be called something else in your setup:

```
java -jar mcp-pki-cli-1.1.0-SNAPSHOT-jar-with-dependencies.jar \
    --init \
    --root-ca-alias root \
    --truststore-path mcp-truststore.jks \
    --truststore-password changeit \
    --root-keystore-path root-ca-keystore.jks \
    --root-keystore-password changeit \
    --root-key-password changeit \
    --x500-name "C=DK, ST=Denmark, L=Copenhagen, O=MCP Test, OU=MCP Test, CN=MCP
Test Root Certificate, E=info@maritimeconnectivity.net" \
    --crl-endpoint "http://api.example.com/x509/api/certificates/crl/root"
```

Please note that the last sub-path in the crl-endpoint parameter should match the value of the root-ca-alias parameter, so in this case the string **root**. Also note that if you use any special characters in the root-ca-alias they will need to be URL encoded in the crl-endpoint. Finally it is highly recommended to NOT use HTTPS and only normal HTTP for the CRL endpoint due to compatibility reasons.

You should now have two different keystore files called mcp-truststore.jks, which contains the root CA certificate, and root-ca-keystore.jks, which contains both the root CA certificate and the root CA private key. The password for both of these is ‘changeit’, but can be set to something else by changing the values of truststore-password and root-keystore-password.

Now you will need to create a certificate revocation list (CRL) for the root CA. This can be done using the following command:

```
java -jar mcp-pki-cli-1.1.0-SNAPSHOT-jar-with-dependencies.jar \
    --generate-root-crl \
    --root-ca-alias root \
    --root-keystore-path root-ca-keystore.jks \
    --root-keystore-password changeit \
    --root-key-password changeit \
    --revoked-subca-file revoked-subca.csv \
    --root-crl-path root-ca.crl
```

The file **revoked-subca.csv** should either be empty or contain the sub CAs that have been revoked separated by commas and should have a format like this

```
<serial-number>;<revocation-reason>;<revocation-date>
```

The resulting file root-ca.crl contains the list of sub CAs that have been revoked. Please note that the signature on this file is only valid for **one** year which means that a new root CRL must be generated and replace the current root CRL before this expires. To do this you only need to repeat the above command.

The next step is then to generate the sub CA(s) that is going to be used for issuing client certificates. The command below shows an example of how to do this:

```
java -jar mcp-pki-cli-1.1.0-SNAPSHOT-jar-with-dependencies.jar \
    --create-subca \
    --root-ca-alias root \
    --root-keystore-path root-ca-keystore.jks \
    --root-keystore-password changeit \
    --root-key-password changeit \
    --truststore-path mcp-truststore.jks \
    --truststore-password changeit \
    --subca-keystore subca-keystore.jks \
    --subca-keystore-password changeit \
    --subca-key-password changeit \
    --x500-name "UID=mcp-idreg, C=DK, ST=Denmark, L=Copenhagen, O=MCP Test,
OU=MCP Test, CN=MCP Test Identity Registry, E=info@maritimeconnectivity.net"
```

This creates a sub CA that contains the information given in x500-name and signed by the root CA. The sub CA certificate is added to mcp-truststore.jks and the keys of the sub CA are stored in subca-keystore.jks.

All the files generated in this section should be stored for later usage.

Using PKCS#11 to generate root and intermediate CA

Storing and handling of root and intermediate CAs and their private keys on hardware security modules (HSM) instead of keystores is also supported by the MCP PKI library using the PKCS#11 standard.

If you are going to use an HSM please make sure that you have the drivers installed for it, and that a PKCS#11 connector shared library is also available for it – this usually comes in the form of a DLL for Windows or a .so file for Unix-like operating systems (Linux, MacOS, etc.). If there are any issues regarding this, please refer to the manufacturer of the HSM. Note that even though PKCS#11 is the one of PKI standards for interfacing to different types of HSM, the implementation of it may differ by manufacturers which can impact the compatibility to the MCP PKI PKCS#11 support. The policy of an HSM may impact it as well.

Before you move on you will need to prepare a token slot in the HSM – how this can be done will usually be described in the documentation of the HSM. In an ideal setting you will also need to have two different HSMs – one for the root CA and one for the intermediate CA. If that is not possible, then the root and intermediate CA should at least be stored in different slots with different passwords in the HSM.

Using MCP PKI

Generation of root and intermediate CA's using the MCP PKI is possible as long as the HSM supports modifying objects over PKCS#11. An example of an HSM that does not support this is the YubiHSM 2 (see https://developers.yubico.com/YubiHSM2/Component_Reference/PKCS_11/).

For Java, to be to able to use the PKCS#11 connector you will need to make a file called pkcs11.cfg. The structure of it and what should be included in this file is described at https://docs.oracle.com/en/java/javase/17/security/pkcs11-reference-guide1.html#GUID-C4ABFACB-B2C9-4E71-A313-79F881488BB9.

An example of how this file could look like when using SoftHSMv2 is shown below:

```
name = RootCA
library = /usr/lib/softhsm/libsofthsm2.so
showInfo = true
slot = 1418149817
```

This tells the Java to make a PKCS#11 provider called ‘RootCA’, that the shared library for the PKCS#11 connector is located at the absolute path ‘/usr/lib/softhsm/libsofthsm2.so’, that info about the HSM should be written to the log on initialization and that slot 1418149817 should be used in the HSM.

Note that sometimes it might be necessary to add additional attributes in the file to be able to use certain features. The snippet below can be added to the file to allow any private key to be able to sign data:

```
attributes(*,CKO_PRIVATE_KEY,*) = {
   CKA_SIGN = true
}
```

A bit of experimentation might be necessary to find the correct configuration for your HSM before going into production.

Generation of the root CA can then be done using the following command:

```
java -jar mcp-pki-cli-1.1.0-SNAPSHOT-jar-with-dependencies.jar \
    --init \
    --pkcs11 \
    --pkcs11-conf pkcs11.cfg \
    --pkcs11-pin 2468 \
    --truststore-path mcp-truststore.jks \
    --truststore-password changeit \
    --root-ca-alias root \
    --x500-name "C=DK, ST=Denmark, L=Copenhagen, O=MCP Test, OU=MCP Test, CN=MCP
Test Root Certificate, E=info@maritimeconnectivity.net" \
    --crl-endpoint "http://api.example.com/x509/api/certificates/crl/root"
```

This will use the configuration that you specified in pkcs11.cfg to interact with the HSM and use the pin “2468” to use slot that is defined. Note that this also assumes that pkcs11.cfg is located in your current working directory, so if it is not you will need to set the value of the pkcs11-conf parameter to the absolute path of the file. If you have used another pin when setting up the slot in your HSM (which you definitely should do!) you should change the value of the pkcs11-pin parameter to the pin that you chose.

Given that this succeeded you will now have a file called ‘mcp-truststore.jks’ in your current working directory. If it failed you might either have to change things in your configuration, or in some cases you might need to use procedure using OpenSSL that is described in the next section.

The next step is to generate the root CA certificate revocation list (CRL). For this you will need a file called **revoked-subca.csv** which should contain the list of revoked intermediate CA’s. When you are initially setting up everything this will of course be empty, so therefore the file should also be empty in that case. If you have to revoke an intermediate CA you will need to include that and any other revoked intermediate CAs separated by commas in the format

```
<serial-number>;<revocation-reason>;<revocation-date>
```

Then to generate the root CA CRL the following command can be used:

```
java -jar mcp-pki-cli-1.1.0-SNAPSHOT-jar-with-dependencies.jar \
    --generate-root-crl \
    --root-ca-alias root \
    --pkcs11 \
    --pkcs11-conf pkcs11.cfg \
    --pkcs11-pin 2468 \
    --revoked-subca-file revoked-subca.csv \
    --root-crl-path root-ca.crl
```

Given that this succeeded you will now have a file called ‘root-ca.crl’ which is the actual root CA CRL which is signed using the private key of the root CA.

The next step is now to generate the intermediate CA that is going to be used to sign end user certificates. Before this can be done it is assumed that the root CA and the intermediate CA will be stored in either two different HSM’s or two different slots in the same HSM. This also means that there needs to be different PKCS#11 configurations for the two – for the root CA the pkcs11.cfg can be reused from previously, but for the intermediate CA you will need to make a new file ‘pkcs11_sub.cfg’ that should contain the configuration for it. To then generate the intermediate CA the following command can be used:

```
java -jar mcp-pki-cli-1.1.0-SNAPSHOT-jar-with-dependencies.jar \
    --create-subca \
    --root-ca-alias root \
    --pkcs11 \
    --pkcs11-root-conf pkcs11.cfg \
    --pkcs11-root-pin 2468 \
    --pkcs11-sub-conf pkcs11_sub.cfg \
    --pkcs11-sub-pin 2468 \
    --truststore-path mcp-truststore.jks \
    --truststore-password changeit \
    --x500-name "UID=mcp-idreg, C=DK, ST=Denmark, L=Copenhagen, O=MCP Test,
OU=MCP Test, CN=MCP Test Identity Registry, E=info@maritimeconnectivity.net"
```

Given that this succeeded, the intermediate CA will now have been generated and stored in the HSM and the certificate of it will have been added to the ‘mcp-truststore.jks’ file. This file along with the .cfg file for the intermediate CA and the root CA CRL should all be kept for later usage.

Using OpenSSL

To use OpenSSL to generate the root and intermediate CA’s you will need to first install a PKCS#11 provider. An example of one such provider is the one that is included in libp11. On Ubuntu this can be installed using the command

```
sudo apt install libengine-pkcs11-openssl
```

When that has been installed you will need to generate the private key that you will be using for your root CA. The following command can be used to generate an elliptic curve key pair using the secp384r1 curve:

```
pkcs11-tool --module /usr/lib/softhsm/libsofthsm2.so --login --pin <password> \
--keypairgen --slot <slot id> --key-type EC:secp384r1 --label root --usage-sign
```

This command uses the path for the SoftHSMv2 connector, so if you are using another HSM you will of course need to change the path given to the ‘module’ parameter to its PKCS#11 connector. Please verify yourself with the documentation of both pkcs11-tool and your specific HSM whether you need to specify additional parameters. After execution of the command please take note of the ID that is given to the private key object in the output of the command as it will be needed in the later stages.

The next step now is to make a configuration file that specifies how OpenSSL should be able to use PKCS#11. The easiest way to do this is to first make a copy of OpenSSL’s default configuration file to a suitable working directory and call it engine.conf. The default configuration file is usually located in /usr/lib/ssl/openssl.cnf or /etc/ssl/openssl.cnf in Linux based systems. Next you will need to append some additional configuration to the file. At the top of the file you should add the line:

```
 openssl_conf = openssl_init
```

At the bottom of the file you should add:

```
[ v3_intermediate_ca ]
# Extensions for a typical intermediate CA (`man x509v3_config`).
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid:always,issuer
basicConstraints = critical, CA:true, pathlen:0
keyUsage = critical, digitalSignature, cRLSign, keyCertSign

[openssl_init]
engines = engine_section

[engine_section]
pkcs11 = pkcs11_section

[pkcs11_section]
engine_id = pkcs11
#dynamic_path is not required if you have installed
#the appropriate pkcs11 engines to your openssl directory
#dynamic_path = /path/to/engine_pkcs11.{so|dylib}
MODULE_PATH = /path/to/pkcs11_driver_library.so
#INIT_ARGS = <any needed init args>
init = 0
```

Next you will need to generate your self signed root CA using the key pair that you generated earlier. This can be done using the following command:

```
openssl req -new -x509 -days 3650 -sha384 -config engine.conf -engine pkcs11 \
-subj '/C=DK/ST=Denmark/O=MCP/OU=MCP/CN=MCP Root \

/E=info@maritimeconnectivity.net' \

-keyform engine -key <pkcs11-uri> -out root.pem
```

The value of `<pkcs11-uri>` can be formatted like `<slot-id>:<key-id>` or `"pkcs11:object=root-ca;type=private;pin-value=changeit"` if the HSM follows RFC7512 PKCS #11 URI scheme. Remember to change the content of the subject string to what you need and use the slot id and key id of the key pair that you generated earlier. If this succeeded you will now have a file called root.pem containing your root CA certificate. This file you will need to import into your HSM and store it under the same ID as the corresponding private key. How this can be done depends on the HSM itself and the interface to it. An example on how to do this with the YubiHSM can be found at https://developers.yubico.com/YubiHSM2/Commands/Put_Opaque.html. Then it is time to generate the root CA CRL:

```
openssl ca -config engine.conf -engine pkcs11 -keyform engine -keyfile <pkcs11-
uri> -gencrl -out root-ca.crl
```

The next step is then to generate the intermediate CA. To do that you must first generate a key pair for it. This can be in the same way as described for the root CA above. Then you must generate a certificate signing request (CSR) for the intermediate CA. This can be done using the command:

```
openssl req -new -days 1825 -sha384 -config engine.conf -engine pkcs11 \

-subj '/C=DK/ST=Denmark/O=MCP/OU=MCP/CN=MCP/E=info@maritimeconnectivity.net' \

-keyform engine -key <pkcs11-uri> -out intermediate.csr
```

Again, remember to change subject to what you need, and use the slot id and key id for the

Next you will need to sign the CSR with the private key of the root CA. This can be done using the following command:

```
openssl ca -config engine.conf -extensions v3_intermediate_ca -notext \

-md sha384 -in intermediate.csr -out intermediate.pem -engine pkcs11 \

-keyfile <pkcs11-uri> -keyform engine
```

The resulting intermediate CA certificate will then be output to the file intermediate.pem. Again you will need to remember to import the file into the HSM. The certificate chain can be created as following:

```
cat intermediate.pem root.pem > mcp-ca-chain.pem
```

The truststore should be created manually for the MIR integration using the following command:

```
keytool -import -keystore truststore.jks -file root.pem -alias <root-ca-alias>
```

For the registration of the intermediate CA to the truststore:

```
keytool -import -keystore truststore.jks -file intermediate.pem -alias
<intermediate-ca-alias>
```

Remember the values `<root-ca-alias> and <intermediate-ca-alias>` need to be matched with the corresponding value in the application.yaml of MIR.

Setting up NGINX

The next step is to set up NGINX. This is not required, but it is highly recommended to use NGINX as a reverse proxy because it makes handling of TLS and client certificate authentication a lot easier.
In the supplied file nginx.conf is an example of how NGINX can be used to proxy incoming requests to the Identity Registry API and Keycloak assuming that they are running on the same machine.
There are some requirements that need to be met to use the configuration.
The first one is that you need to have a domain registered with the sub-domains **api, api-x509, maritimeid, maritimeid-x509**. These can of course be changed to something else in the configuration is necessary.
The second requirement is that you need to have a TLS certificate for your domain. The path to the certificate should then be defined in the variable **ssl_certificate** and then the path to the corresponding private key should be defined in the variable **ssl_certificate_key**.
The third requirement is that the variable **ssl_crl** should point to an up-to-date file that contains the CRLs of the root CA and the sub CAs concatenated together. How to set this up will be described later in this document.
The fourth requirement is that the variable **ssl_client_certificate** must point to a file that contains the certificates of the root CA and the sub CAs in PEM format. These can be extracted from the mc-truststore.jks generated in the previous step using a tool like KeyStore Explorer.
NGINX can either be installed and run directly on the machine or in a Docker container. Note that if you choose to run it in a Docker container it is recommended to run it with the option --net=host so it binds directly to the network interface of the host instead of using the default bridge driver.

Setting up database

Prerequisite of installing both Keycloak and identity registry API is an installation of MySQL or MariaDB. For Keycloak, it is important to set exactly same environment defined in the dockerfile of MCPKeycloakSPI (https://github.com/maritimeconnectivity/MCPKeycloakSpi/blob/master/docker/Dockerfile). For example, if it contains ‘ENV DB_DATABASE keycloak’, you have to create a database ‘keycloak’ through *CREATE DATABASE keycloak*;. It is also required to set a dedicated user id in mysql.user and a password that perfectly corresponds to the values in the dockerfile. For the case using the prebuilt Docker container will be described later the database environment setup should be matched to the dockerfile of the repository of it.

Identity registry API provides bash files to setup the initial database, which are available on the repository (https://github.com/maritimeconnectivity/IdentityRegistry/tree/master/setup). ‘setup-db.sh’ that should be executed first includes execution of a series of SQL statements in ‘create-database-and-user.sql’. After that flywaydb will handle creation and migration of tables when the program starts, and will fail if the tables already exists. Importing the content from ‘create-mc-org.sql’ should only be done after the first execution as described in the section **Putting everything together**. For the case to remove the database and the user you can just execute ‘drop-db.sh’ or just ‘drop-db-and-user.sql’.

Setting up Keycloak

The next step is to setup Keycloak. This can be done using either a standalone installation of Keycloak or using the prebuilt Docker container. Using the Docker container is recommended, but if a standalone installation is required a guide on how to use this can be found at https://github.com/maritimeconnectivity/MCPKeycloakSpi.

Using a Docker container

A prebuilt Docker container for Keycloak with MCP specific functionality can be found at https://cloud.docker.com/u/dmadk/repository/docker/dmadk/keycloak-mysql-mc-ha. At the time of writing the latest tag for this is **1.1.0**. The **latest** tag is built from the latest code from the main git branch and is therefore not guaranteed to be stable.

For creating the Docker container using the **1.1.0** tag you can use the following command:

```
docker create --name=mir-keycloak --restart=unless-stopped -p 8080:8080 \
-v <directory for configurations>:/mc-eventprovider-conf \
-e MC_IDREG_SERVER_ROOT=https://api-x509.example.com \
-e JGROUPS_DISCOVERY_EXTERNAL_IP=<valid IP address>
dmadk/keycloak-mysql-mc-ha:1.1.0
```

The directory that is mounted to '/mc-eventprovider-conf' in the container must contain the following file:

- idbroker-updater.jks

This file is used be Keycloak to get specific attributes from the Identity Registry API for users when they login. How this file is generated is described later in this document and does not need to be present to begin with.

The URL given as value for the variable MC_IDREG_SERVER_ROOT must be set to the same value as the one set in the NGINX configuration.

If running Keycloak in clustered mode the value of JGROUPS_DISCOVERY_EXTERNAL_IP must be set to an IP address that Keycloak is reachable on. If it is not going to be run in clustered mode, it can just be set to 127.0.0.1 or another random IP.

As default the database and the user that is to be used for it are set to be **keycloak** and the password for the database user is set to be **password**. If you want to use other values than these you can set them as variables in the above command as described at https://hub.docker.com/r/jboss/keycloak.

If you want to use PostgreSQL as the database instead of MySQL/MariaDB you will need to change the value the environment variable JGROUPS_DISCOVERY_PROPERTIES to the one that can be found at https://github.com/keycloak/keycloak-containers/blob/main/docker-compose-examples/keycloak-postgres-jdbc-ping.yml#L49.

To start Keycloak you can then use the command

```
docker start mir-keycloak
```

Setting up Realms in Keycloak

When you have gotten Keycloak up and running you will need to setup the needed Realms. To login to the admin interface of Keycloak go to http://localhost:8080 if you have Keycloak running on localhost or else use the domain set in your NGINX configuration. The first time you login you will need to setup an admin user and password.

After that you will need to import the three *-realm.json files from https://github.com/maritimeconnectivity/IdentityRegistry/tree/master/setup.

This can be done in the admin interface of Keycloak by hovering the mouse over the dropdown 'Select realm' and then click 'Add realm'. Here you can then import the *-realm.json files one by one.

Note that the URLs to OIDC clients, identity providers, etc. are set to localhost in the *-realm.json files so these will need to be updated to the correct URLs after the files have been imported. Please refer to Keycloak's own documentation at https://www.keycloak.org/documentation.html.

Setting up the Identity Registry API

The next step then is to set up the Identity Registry API (MIR API). There are two ways this can be done – either by setting it up directly on the system or by running it in a Docker container. Running in a Docker container is recommended, but both methods will be described in separate sections.

The source code for the Identity Registry API can be found at https://github.com/maritimeconnectivity/IdentityRegistry.

Using a Docker container

A prebuilt Docker image for the Identity Registry API can be found at https://cloud.docker.com/u/dmadk/repository/docker/dmadk/mc-identity-registry-api. At the time of writing the latest tag for this is **1.1.0**. The **latest** tag is built from the latest code from the main git branch and is therefore not guaranteed to be stable.

For creating the Docker container using the **1.1.0** tag you can use the following command:

```
docker create --name=mir-api --restart=unless-stopped -p 8443:8443 \
-v <directory for configurations>:/conf dmadk/mc-identity-registry-api:1.1.0
```

The folder that has been mounted to 'conf' in the container needs to contain the following files:

- An application.yaml file
- A keycloak.json file
- mc-truststore.jks
- root-ca.crl
- subca-keystore.jks

How to make the two first files will be described later on. The three other files were the ones that were generated in the first step.

Directly on the system

For the standalone method you need to first download the newest version of the Identity Registry API from https://github.com/maritimeconnectivity/IdentityRegistry. When you have done that you can compile the source code by going into the root directory of the project and issue the following command:

```
mvn clean install
```

After this has finished you will now have a new directory called ‘target’ that should contain a file called mcp-identityregistry-core-latest.war. This can then be executed using the following command:

```
java -jar mcp-identityregistry-core-latest.war \
--spring.config.location=<location for an application.yaml file> \
--keycloak.config.file=<location for a keycloak.json file>
```

Again how to make the two needed files will be described later.

Getting keycloak.json

For getting the keycloak.json file for MIR API you will need to login the admin interface of Keycloak. Here you should then choose the MCP realm and click on ‘Clients’. Here you then need to find and click on the client called ‘mcpidreg’. Here you should then click on the ‘Installation’ tab and in the dropdown that appears click ‘Keycloak OIDC JSON’ and then ‘Download’. This file should be put in the configuration directory of the MIR API.

Making an application.yaml file

An example of an application.yaml file can be found at https://github.com/maritimeconnectivity/IdentityRegistry/blob/master/src/main/resources/application.yaml.

It is important that this file is configured according to the correct URLs, database connections, name of the default sub CA, name of the root CA, mail sending settings, etc.

When all of this has been configured the resulting file should then be put in the configuration directory of the MIR API.

Putting everything together

You can now also start up the MIR API. If you have set it up using the Docker container you can start it using the command

```
docker start mir-api
```

If you are not running it in a Docker container you can start it as described earlier.

If everything has been configured correctly all tables for the database should be generated automatically while the program starts.

After the program has finished starting up, indicated in the log by the entry ‘Started McIdregApplication in X seconds (JVM running for Y)’, you now need to create an organization that is going to be used to bootstrap everything. To do this you need to insert the values from https://github.com/maritimeconnectivity/IdentityRegistry/blob/master/setup/create-mc-org.sql into the database that is used for MIR API. This will create an organization called ‘Bootstrap Org’ and a permission called MCPADMIN for the organization that maps to the role ROLE_SITE_ADMIN which is a kind of ‘superadmin’ role that is able to administrate all entities in the MIR.

When this has been done we can move on to creating the organization that is going to be managing the MIR instance. A description on how to do this is given at https://github.com/maritimeconnectivity/IdentityRegistry#insert-data. Note that this description assumes that both the MIR API and Keycloak are running on localhost so you might need to change the URL to fit your own setup. Also note that the organization that is going to be created is called ‘Danish Maritime Authority’ which you might also want to change. Finally the role mapping that is created from dma.json contains the role ROLE_ORG_ADMIN and the permission MCPADMIN. This gives users from the DMA organization with the permission MCPADMIN admin permissions for entities in the DMA organization. Like for the ‘Bootstrap Org’ organization you might also want to create another role with a different name and the role ROLE_SITE_ADMIN for the organization if you want it to be able to have ‘superadmin’ permissions. Note that this cannot be created using the API and needs to be created directly in the database as was done earlier when creating the ‘Bootstrap Org’ organization.

Finish setting up NGINX

Now that the MIR API is up and running it is time to finish the setup of NGINX for it to be able to validate client certificates. This is done by setting up a CRON job that executes a script that gets the CRL for the root CA and for the sub CA(s) and combines them in a single file. If you want the CRON job to be run every 5 minutes and redirect the console output to a log file the CRON configuration could look like this:

```
*/5 * * * * update-crl.sh &>> update-crl.log
```

An example of this script can be seen in the supplied update-crl.sh.

In the NGNIX configuration you should then change the value of **ssl_crl** to the location of the file that contains the CRLs. After that you should then restart NGINX for the change to take effect.

Setting up synchronization between MIR API and Keycloak

The next step now is to setup synchronization between the MIR API and Keycloak. This needs to be setup so that when a user authenticates in Keycloak, information about the user, like name, permissions and so on, can be fetched from the MIR API by Keycloak in order to make sure that the information that Keycloak has stored about the user is synchronized with the information that the MIR API has stored about the same user.

To do this you need to create a ‘device’ entity in the organization that you have setup to have the ‘superadmin’ permissions. This ‘device’ will then be used to facilitate the authentication of Keycloak against the MIR API using a client certificate.

If you create the device directly using the MIR API you can take inspiration from the given sync-device.json. The details of this device corresponds to the default configuration in the user-sync part of the MIR API application.yaml. You can also consider setting up the MCP management portal so you don’t need to use the APIs directly. A manual on how to use this can be found here https://manual.maritimeconnectivity.net/.

If you are using the MIR API directly you can reuse the authentication that you used previously to insert data. An example of the values for the identity of this device can be found at https://github.com/maritimeconnectivity/IdentityRegistry/blob/master/setup/guide/sample-conf/sync-device.json. If using this you will of course need to change the values to fit your use case. The device can then be created using the following command:

```
curl -k -H “Content-Type: application/json” -H “Authorization: Bearer $TOKEN” \
--data @sync-device.json
https://example.com/oidc/api/org/urn:mrn:mcp:org:idp1:dma/device
```

When you have then created this device you will need to issue a certificate for it. This can either be done using the management portal or again using the API directly. To do the latter the following command can be used if the MIR API is configured to allow server generated keys:

```
curl -k -H “Authorization: Bearer $TOKEN”
https://example.com/oidc/api/org/urn:mrn:mcp:org:idp1:dma/device/urn:mrn:mcp:dev
ice:idp1:dma:sync/issue-new
```

The format of the returned JSON object is described at https://github.com/maritimeconnectivity/IdentityRegistry#insert-data. The parts of the object that need to be used are the JKS keystore and the keystore password. The JKS keystore is BASE64 encoded so therefore it needs to be decoded and saved in a file called idbroker-updater.jks. This file should then be put in the directory that corresponds to the /conf directory for the Keycloak Docker container or in the directory that corresponds to the ‘keystore-path’ variable if using a standalone installation of Keycloak. In the default configuration of Keycloak the password for the keystore file is set to be ‘changeit’ so Keycloak to be able to open the keystore you might need to either change the password of it or set the ‘keystore-password’ variable to the correct password.

If the MIR API is NOT configured to allow server generated keys you will need to generate a key pair and a certificate signing request (CSR) for the device you just created.
To do that you can use OpenSSL as follows:

```
openssl ecparam -out private.key -name secp384r1 -genkey
```

```
openssl req -new -key private.key -out request.csr
```

What the first command did was that it created an elliptic curve private key and saved it in the file ‘private.key’, which is then used by the second command to generate a CSR that is signed using the private key and saved in the file ‘request.csr’.
The next step is then to send the CSR to the MIR API to get it signed. This can be using for example cURL in the following way:

```
curl -k -H "Authorization: Bearer $TOKEN" -H "Content-Type: text/plain" \
--data-binary @request.csr \
https://localhost/oidc/api/org/urn:mrn:mcp:org:idp1:dma/user/urn:mrn:mcp:device:
idp1:dma:sync/certificate/issue-new/csr
```

If this worked correctly the MIR should have returned a signed certificate chain which looks similar to this:

```
-----BEGIN CERTIFICATE-----
....
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
....
```

```
-----END CERTIFICATE-----
```

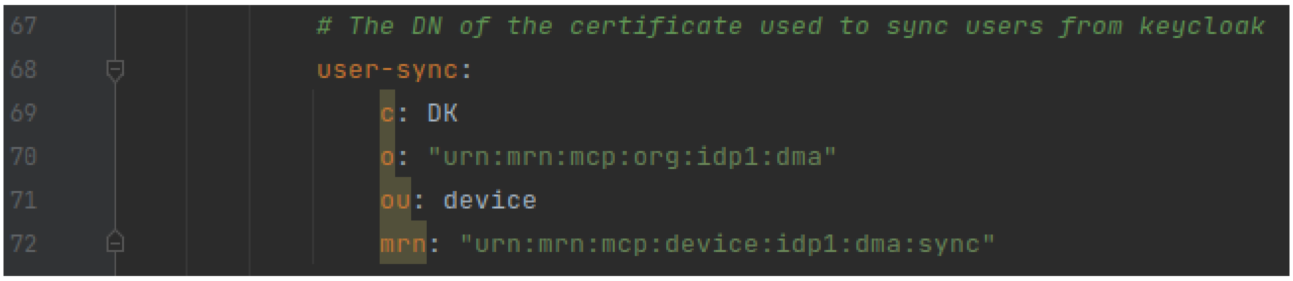
This should be saved in a file called certificate.pem.
The next step then is to first combine the certificate chain and private key into one file and then generate a keystore that enables Keycloak to use the generated certificate. This can be done in the following way:

```
openssl pkcs12 -export -out keystore.p12 -in certificate.pem -inkey private.key
```

```
keytool -importkeystore -deststorepass changeit -destkeystore \
idbroker-updater.jks -srckeystore keystore.p12 -srcstoretype PKCS12
```

The resulting file ‘idbroker-updater.jks’ should then be put in the directory that corresponds to the /conf directory for the Keycloak Docker container or in the directory that corresponds to the ‘keystore-path’ variable if using a standalone installation of Keycloak. In the default configuration of Keycloak the password for the keystore file is set to be ‘changeit’ so Keycloak to be able to open the keystore you might need to either change the password of it or set the ‘keystore-password’ variable to the correct password.

The final step you will need to do is to configure the MIR API to accept this newly created certificate. For this you will need to find the section that is shown in the image below in the application.yaml file of the MIR API.



```
        # The DN of the certificate used to sync users from keycloak
        user-sync:
            c: DK
            o: "urn:mrn:mcp:org:idp1:dma"
            ou: device
            mrn: "urn:mrn:mcp:device:idp1:dma:sync"
```

In this section you will need to change the value of the **c, o, ou** and **mrn** variables to match the values of the **C, O, OU** and **UID** fields of the certificate you just issued. If you are not sure about what these are you can use OpenSSL to print out the contents of the certificate:

```
openssl x509 -in certificate.pem -noout -text
```

In the output from this command you should look for the line that looks something like:

```
Subject: C = …
```

After you have changed the configuration you will need to restart the MIR API.

After the MIR API has been restarted everything should now be good to go and ready to be used.

Upgrading

WARNING

Even though upgrades in most cases should not cause any problems, there is always the possibility that something can go wrong. Therefore it is always a good idea to backup your databases and any external configuration before performing the upgrade.

For database backups you should make sure that you don’t include any database locks. An example of this can be done can be found at this StackExchange post.

Upgrading from an older version to a newer one of the MIR API and Keycloak can in most cases be done by just stopping the old version of the components and then starting the new ones with the same configurations and attributes.

For the upgrade of Keycloak there are unfortunately some MCP specific configurations that cannot be updated automatically. For this purpose a Bash script has been made. The script can be found here and requires that the user has the Keycloak Admin CLI and jq installed.