Niord System Development Guide

Table of Contents

1. Introduction 1
2. Java Coding Conventions 3
3. Webapp Coding Conventions 7
4. Niord Development Set-Up 9
5. Niord-Proxy Development Set-Up 11
6. Building Niord App as Docker Container 12

Chapter 1. Introduction

- This version of the development guideline is for Niord 3.0. Niord 3.0 is based on the Quarkus framework and replaces Niord 2.0 which was based on Wildfly.

Niord (Nautical Information Directory) is a system for producing and publishing Navigational Warnings (*NW*) and Notices to Mariners T&P (*NM*).

It was originally developed as part of the EfficienSea2 EU project and subsequently implemented as a production system for the Danish Maritime Authority.

This guide should be read by developers of Niord. It will use the Danish Niord system as a use case. The guide will cover topics, such as coding conventions, how to set up a development environment, etc.

1.1. Developer Profile

For the development process to be effective, the developer should preferably have a thorough knowledge of the following technologies:

- Java 21
- Jakarta EE: Niord uses the entire Jakarta EE stack, specially JPA, JAX-RS, Servlets, Batch API, CDI, etc.
- AngularJS + Bootstrap
- MySQL
- Docker

It would also be advantageous, if the developer has some knowledge of the following technologies:

- OpenLayers
- ActiveMQ Artemis
- Quarkus
- Keycloak
- Freemarker
- Maven
- Git(hub)

Furthermore, the actual Niord source code also has a certain learning curve.

1.2. Resources

The main GitHub repository for Niord is found at https://github.com/niordorg. For the developer, the main projects of the repository are.

Projects	Description
niord	Main project for the Niord production system.
niord-dk	Extensions for the Danish Niord production system.
niord-proxy	Simple end-user facing website that displays the active messages from the Niord production system.
niord-gh	Extensions for the Ghanaian Niord production system (NB: separate repo).

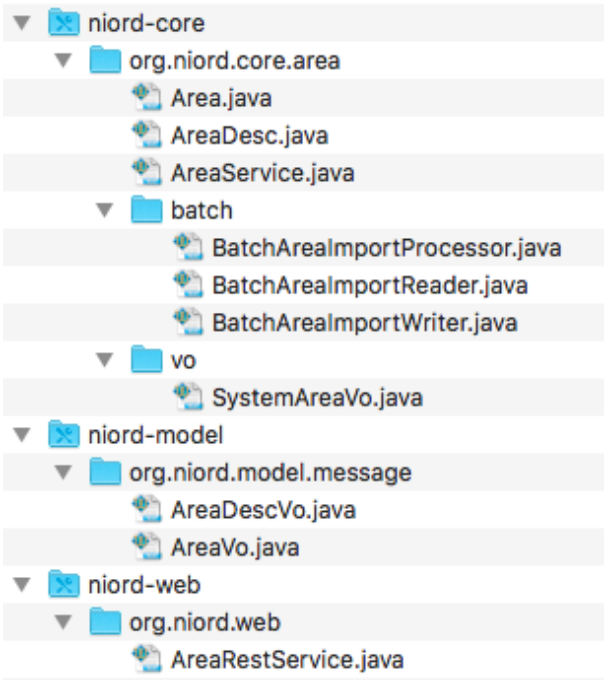
Chapter 2. Java Coding Conventions

Niord is fairly consistent about using naming and coding conventions, and understanding these conventions makes it easier to read the code.

Each *logical entity* of Niord, such as an *area*, a *chart*, a *message* or a *publication*, is implemented using JPA entities, JSON-serializable value objects, business logic, REST interfaces, and batch import classes.

2.1. Area Example

As an example, consider the *Area* entity, which represents a specific area in the Areas tree. The classes used to implement areas, can be seen below:



The classes are spread across three niord sub-projects:

Projects	Description
niord-model	Contains the Niord Message Model. The message model is made up of value objects. Third-party application can define a dependency on this project if they integrate with Niord via the Public REST API.
niord-core	Contains the actual entity classes and business logic (session beans). May also define system model value objects, in as much as these may extend the niord-model classes.
niord-web	Contains the JAX-RS REST interface used by clients (such as the Niord web application) to access the entities and execute business logic.

2.1.1. Classes and Naming Conventions

Classes	Description
`AreaVo`	The Niord Message Model representation of an area. The `Vo` suffix is used for all *value objects*, which are essentially JSON-serializable versions of the real entities.
`AreaDescVo`	Contains a language code and all localizable attributes of `AreaVo`. These associated entities have a `Desc` suffix, as in `AreaDescVo`.
`Area`	The JPA entity definition of an area. Whereas there will be a strong correlation between the attributes of the `Area` entity and the `AreaVo` value object, the entity class may define additional attributes, which are not part of the public niord-model class. The entity class thus constitutes a *system model*.
`AreaDesc`	The JPA entity definition of the area description entity, i.e. the localizable attributes of an area.
`SystemAreaVo`	When a *system model* entity, such as `Area`, contains attributes not included in the value objects of the Niord Message Model, then a value object with a `System` prefix is introduced to capture the additional attributes. As such, `SystemAreaVo` will extend `AreaVo` and includes the additional attributes.
`AreaService`	Each entity will have a companion stateless session bean or singleton EJB, with a `Service` suffix. This class defines the business logic and life-cycle management functions of the entity.

`BatchAreaImportReader`
`BatchAreaImportProcessor`
`BatchAreaImportWriter`
	Many of the Niord model entities have an associated batch job for importing the entities.
	The batch job is typically implemented using the Java EE batch API, and the three phases will be implemented by classes that have a `Batch` prefix and the `Reader`, `Processor` and `Writer` suffixes respectively.

`AreaRestService`	Each Niord entity will also be associated with a JAX-RS REST interface. This interface is typically a thin wrapper on top of the service interface, and will have the `RestService` suffix. The REST interface also performs all the security and permission checks used to protect the Niord system.

2.2. Localization

Almost all entities in Niord are localizable to any number of languages.
As can be seen from the Niord Message Model, this is implemented by associating an entity with a list of classes that contain a language code and all localizable attributes.

Area contains one localizable attributes; *name*:

```
public class AreaVo implements ILocalizable<AreaDescVo>, IJsonSerializable {
    List<AreaDescVo> descs;
}

public class AreaDescVo implements ILocalizedDesc, IJsonSerializable {
    String lang;
    String name;
}
```

2.3. Serialization and De-serialization

All Niord entities have methods for converting to and fro their *value object* representation.

In the simplest form, the JPA entity model will define a constructor that takes the *value object* representation as a parameter, and it will have a `toVo()` function that returns the *value object* representation of the entity. Example:

```
@Entity
public class Domain extends BaseEntity<Integer> {
    public Domain() {}
    public Domain(DomainVo domain) {
        // Instantiate entity from value object
    }
    public DomainVo toVo() {
        // instantiate and return a value object from entity
    }
}
```

Sometimes, however, things are a little more complex, as is the case for `Area`.

`Area` has two *value object* representations, `AreaVo` and `SystemAreaVo`. When, say, a public REST API call returns a message with an associated area, then the `AreaVo` should be returned. If, however, a system administrator edits an area via the Niord webapp, then the `SystemAreaVo` representation should be used.
So, `Area.toVo()` actually takes the target value object class as a parameter, and leave it to the REST service to decide which representation to use.

Another complexity in serializing an entity to its *value object*, is that often you wish to exact control over which fields to return.
One example is language control. Most of the REST API calls will only return the localizable entities (e.g. `AreaDescVo`) for the requested language. This preserves bandwidth and makes client code simpler.
Another good example is control over the hierarchical relationship of Areas. When a message with an associated area (say, "Kattegat") is returned from a REST call, then you want the *parent* relationship of areas to be included ("Kattegat" should include a parent-reference to "Denmark). Alternatively, when editing the area tree on the *Areas* admin page, then you want the REST call to return root areas with their *children* relationship.

To facilitate this type of serialization control, Niord use a `DataFilter` helper class, which defines the fields and language to include. To control the serialization of an entire tree of related entities, the fields can be prefixed with the entity name, as seen in the example below:

```
DataFilter filter = DataFilter.get()
    .fields("Message.details", "Message.geometry", "Area.parent", "Category.parent")
    .lang("en");
```

Hence, the resulting serialization code for `Area` will thus be:

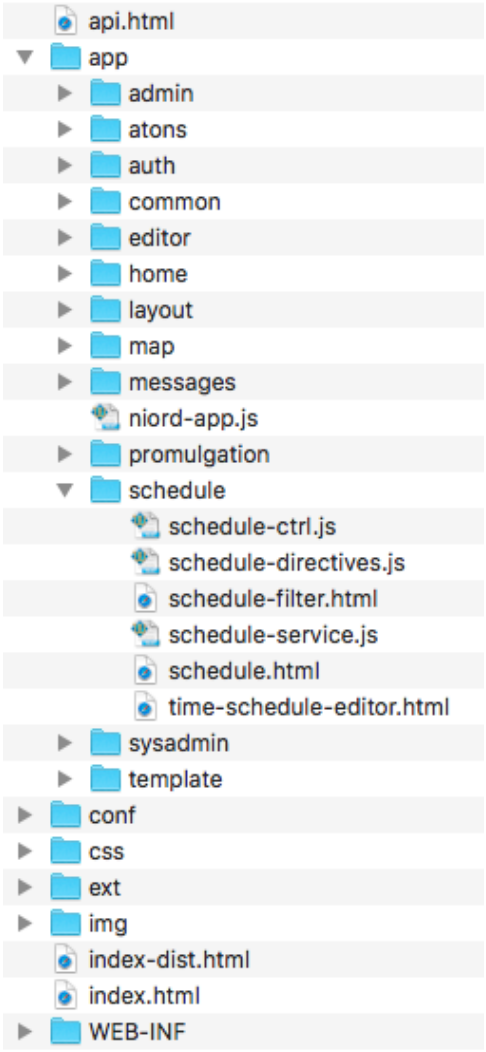
```
@Entity
public class Area extends TreeBaseEntity<Area> implements ILocalizable<AreaDesc> {
    public Area() {}
    public Area(AreaVo area, DataFilter filter) {
        // Instantiate entity from value object
    }
    public <A extends AreaVo> A toVo(Class<A> clz, DataFilter filter) {
        // instantiate and return a value object from entity
    }
}
```

Chapter 3. Webapp Coding Conventions

The other big chunk of code in Niord is the AngularJS and JavaScript-based web application.

3.1. Organization of Angular Sources

The web application is organized as follows:



All source files developed as part of Niord, is placed in the *app* folder. All external dependencies, such as third-party AngularJS directives, are placed in the *ext* folder.

Under *app*, the angular sources, such as directives, controllers, services and partials (html), are primarily organized by *main page*. So, all sources for the *Messages* page are in the *messages* folder, etc.

3.2. Application Cache

The Niord web application use various HTML 5 features, such as *Local Storage* and *Application Cache*. The *Application Cache* in particular, makes day-to-day use of Niord substantially faster for end users.

Application Cache has supposedly been deprecated, and is to be replaced with a *Service Workers* mechanism. However, Service Workers are not yet supported by Safari (read: iOS).

The niord-web project can be built using the "dist" profile:

```
cd niord-web
mvn -P dist clean install
```

This will perform the following modifications to the resulting war file:

- All Niord CSS and JavaScript files will be merged into single files.
- An HTML5 Application Cache manifest file is generated to facilitate caching.

3.3. Overlay Wars

Another mightily important feature used by Niord, is the web-application overlay mechanism.

In reality, the Niord project would not be used in production by itself. Rather, developers would create a country-specific version (such as niord-dk for Denmark, or niord-gh for Ghana) with all the customizations and legacy integration needed for that particular country.

The main trick is to create a web application that functions as an *overlay* of the Niord web application. This allows the developer to selectively replace individual files, such as AngularJS files, CSS files, resource bundles, etc.

The pom.xml file of the new web application, should define the overlay as:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-war-plugin</artifactId>
    <version>2.4</version>
    <configuration>
        <workDirectory>target/overlay</workDirectory>
        <overlays>
            <overlay>
                <groupId>org.niord</groupId>
                <artifactId>niord-web</artifactId>
            </overlay>
        </overlays>
    </configuration>
</plugin>
```

Chapter 4. Niord Development Set-Up

As described in the [Overlay Wars] section, a concrete implementation of the Niord system will almost always involve a country-specific customized project, such as the niord-dk project for Denmark, or the niord-gh project for Ghana (This project is still running on (Wildfly) Niord 2.0).

The development setup described in this section will be based on *niord-dk*.

4.1. Prerequisites

- Java 21
- Maven
- Docker

The set-up described in this section assumes that you are using Linux / MacOS X. If you are using Windows, you will probably need to adjust the various commands and scripts accordingly. Consider using Git Bash for the easiest migration.

4.2. Check out Niord Projects

As mentioned, the development set-up used in this document is based on the Danish Niord project.

Either use your favorite IDE (assume IntelliJ) to check out the niord and niord-dk projects, or check them out from the command line:

```
git clone https://github.com/NiordOrg/niord.git
cd niord
mvn clean install
cd ..
git clone https://github.com/NiordOrg/niord-dk.git
cd niord-dk
mvn clean install
```

The rest of the section will assume that you are working in the *niord-dk* directory.

Import the niord-dk project in IntelliJ via its pom.xml. Under the "Maven Projects" also import the parent niord project. This will allow you to work and debug both code-bases from within IntelliJ.

The first time around, IntelliJ may have created a new unversioned directory, *niord-dk-web/overlays*. Just delete it and build again.

4.3. Using Docker to start needed services for development purposes

Make sure your working directory is *niord-dk* and run

```
docker compose up
```

Which will start the following services:

- niord-db : Database for Niord (MySQL)
- niord-mq : Message Queue used for promulgation of messages (ActiveMQ)
- niord-smtp: : Used for testing receiving mails (mailhog)
- niord-jeager : A Jaeger services for collection requests traces
- niord-keycloak : A Keycloak instance for testing Niord
- niord-keycloak-db : A Database (MySQL) for use by Keycloak

4.4. Running the Niord App

Next (make sure you are in the niord-dk repository), run the Quarkus development script:

```
./q-dev.sh
```

This will startup the application and create a Niord home directory (niord-dk/home-niord-dk) that Niord uses to store various files. The script will also bootstrap the database with data after 10-20 seconds.

The Keycloak docker image creates an initial domain, "Master", and a Niord user, sysadmin/sysadmin, that should be used for the initial configuration of the system, whereupon they should be deleted.

Enter http://localhost:8080 and check that you can log in using the Niord sysadmin/sysadmin user.

4.5. Starting over

If you end up with a system that doesn't work try starting over by deleting the container group (niord-dk) in Docker. And the niord-dk/home-niord-dk directory. Then run docker compose and q-dev.sh again.

Chapter 5. Niord-Proxy Development Set-Up

The Niord-Proxy is a simple client-facing website that retrieves and renders messages from a Niord back-end server.

5.1. Prerequisites

- Java 8
- Maven 3.5.4 (Project does not supper Maven 3.6.0 or greater)

When setting up the development environment for the Niord-Proxy, you need to point it to an existing Niord service, from where it will fetch data, i.e. the active messages and publications.

The set-up in this document will assume that you are running a development version of the Niord service, as described in the [Niord Development Set-Up] section.

5.2. Check out Niord-Proxy Project

Either use your favorite IDE (assume IntelliJ) to check out the niord-proxy project, or check it out from the command line:

```
git clone https://github.com/NiordOrg/niord-proxy.git
cd niord-proxy
```

Import the niord-proxy project in IntelliJ via its pom.xml.

5.3. Starting Niord-Proxy

The Niord Proxy can be run as an executable jar:

```
mvn clean install
java -Dswarm.http.port=9000 \
    -Dniord-proxy.executionMode=DEVELOPMENT \
    -Dniord-proxy.server=http://localhost:8080 \
    -Dniord-proxy.repoType=SHARED \
    -Dniord-proxy.repoRootPath=/Users/carolus/.niord-dk/repo \
    -Dniord-proxy.timeZone=Europe/Copenhagen \
    -Dniord-proxy.areas="urn:mrn:iho:country:dk|56|11|6,urn:mrn:iho:country:gl|70|
-40|4,urn:mrn:iho:country:fo|62|-7|8" \
    -Dniord-proxy.analyticsTrackingId= \
    -jar target/niord-proxy-swarm.jar
```

An easier alternative is to run the `org.niord.proxy.NiordProxyMain` main class directly from your IDE (e.g. IntelliJ). Use the same *VM Options* as for the executable jar above.

Chapter 6. Building Niord App as Docker Container

To build the Niord App as a docker container you can use

```
# Standing in niord-dk, create the jars needed for the Docker image
mvn clean package

# build the docker image. Change the target container name to your liking
docker build -f niord-dk-web/src/main/docker/Dockerfile.jvm -t dma/niord-dk-app:1.0.0
niord-dk-web
```