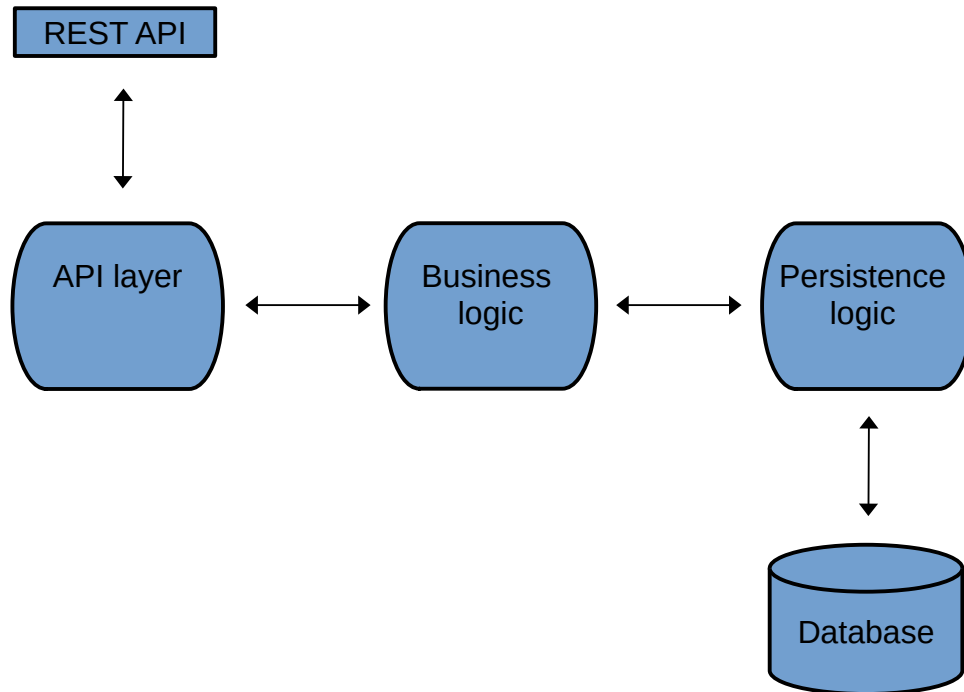


Naming – architecture & code

understand, compile, test, run & more

by lars.johansson@ess.eu, ICS Software



Introduction

Naming

The purpose of Naming is to handle Naming of ESS wide physical and logical devices according to ESS Naming Convention.

Naming backend is a web application implemented as a REST style web service backed by a relational database. The web service is implemented as a Spring Boot application and the database is available as PostgreSQL.

Document

The purpose of this document is to help development and maintenance of, and is link between design and code for, Naming backend, which is implementation of ESS Naming Convention.

This document will give introduction to architecture and code in Naming backend. The goal is to be able to understand, compile, test, run application & more. More includes working with Naming and database, checkout new branch, do something and/or perform a task, test, commit, push, deploy.

Table of Contents

Naming – architecture & code.....	1
Introduction.....	1
Getting started.....	3
Understand.....	5
Big picture.....	5
Names & Structures.....	5
Rules.....	6
Business rules.....	7
Domain model & Example.....	9
Database.....	9
Get up & running.....	11
Database.....	11
Application.....	12
Compile.....	12
Test.....	12
Run.....	13
Architecture & code.....	13
Architecture mapped to code.....	14
A request mapped to code.....	15
Implementation.....	16
Configuration.....	16
Database.....	16
ESS Naming Convention.....	16
Beans.....	17
REST API.....	18
Excel.....	20
Open API & Swagger UI.....	21
Business logic.....	21
Persistence logic.....	21
Tests.....	21
Integration tests with Docker containers.....	22
How to go about tasks.....	24
Repository.....	24
Branch handling.....	24
Deploy.....	24
Reference.....	24

Getting started

In various places are framed boxes with information that is noteworthy and may help understanding

Note!

Text that is noteworthy, summary, help & more

First set of sections of this document is aimed at understanding application from various perspectives.

- Understand – big picture – focus on high-level understanding without going into details. Having a mental picture of where things belong will help going through rest of document.
- Understand – get up & running – is aimed at being able to compile, test, run application and database.
- Understand – architecture & code – will go into detail about architecture of application, how a client call to Naming REST API is mapped into code and how the call flows through the code. Different layers and parts of code are covered with purposes and explanations.

Next set of sections is on implementation.

- ESS Naming Convention – details rules and implementation of rules for names & structures. This is core part of Naming.
- REST API – about requests to application that can be made and is how a client interacts with application.
- Persistence model & mapping – how application maps information to persistence model which is used for storage.
- Test – about unit, integration and manual tests.

Additional sections give help to how tasks may be approached together, usage of Gitlab together with references. This document mentions, but is not about, authentication and authorization.

It is assumed that

- *reader has access to Git repository for Naming backend*
 - <https://gitlab.esss.lu.se/ics-software/naming-backend>
 - */folder/subfolder/file* *path in repository*
- *reader is familiar with* *or willing to learn!*
 - *Git & Gitlab, Java, SQL, Maven, Docker*
 - *documents*
 - *brief introduction, cheat sheet, excel guide* */docs/about/*

To set up development environment

/README.md

To learn about refactoring (optionally)

/docs/developer/naming-convention-tool/

/docs/developer/refactoring/

A package containing “old” in its name, relates to refactoring of Naming and and implementation of previous functionality. This may be removed at any time.

Docker and docker-compose are used extensively throughout application for various purposes such as test and run. However, mentioned tools are not required.

At this point, it is assumed that repository is available and, if intention is to compile and run code, that development environment is set up with required tools available.

Understand

Big picture

Names & Structures, Rules – follow from ESS Naming Convention

Names & Structures

<i>ESS Name</i>	<i>System structure</i>	<i>Device structure</i>
	<i>Which part of the facility does the device provide service to?</i>	<i>What kind of service does the device provide?</i>
<i>Must refer to System structure</i>	<i>1 System Group</i>	<i>1 Discipline</i>
<i>May refer to Device structure</i>	<i>2 System</i>	<i>2 Device Group</i>
<i>May have index for instance</i>	<i>3 Subsystem</i>	<i>3 Device Type</i>

<i>ESS Name</i>	=	<i>System Group</i>				
		<i>System Group</i>	+	<i>Device Type</i>	+	<i>Index</i>
		<i>System</i>				
		<i>System</i>	+	<i>Device Type</i>	+	<i>Index</i>
		<i>Subsystem</i>				
		<i>Subsystem</i>	+	<i>Device Type</i>	+	<i>Index</i>

Rules

Rules for structures

Structures

System, Subsystem, Discipline, Device Type must have mnemonic

System Group may have mnemonic

Device Group must not have mnemonic

A mnemonic is a string of characters and numbers that must be unique in its namespace (rules apply)

Rules for names

Names

System structure

System structure + Device structure + Index

A name

system structure mnemonic path

system structure mnemonic path : device structure mnemonic path – index

System structure mnemonic path

if System Group then System Group mnemonic

if System then System mnemonic

if Subsystem then System mnemonic – Subsystem mnemonic

Device structure mnemonic path

if Device Type then Discipline mnemonic – Device Type mnemonic

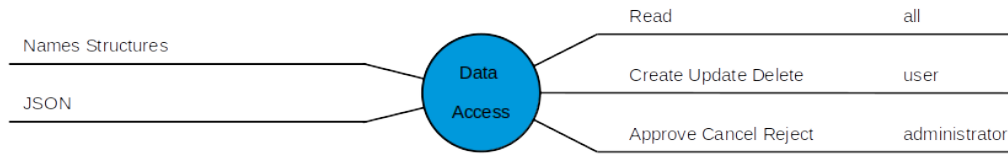
Index

a string of characters and numbers (rules apply)

Furthermore, there are rules for mnemonics and index.

Business rules

REST API follow business rules in order to handle names and structures.



This is expressed in more detail in table.

Operation		Names	Structures		Authority		
					All	User	Administrator
Read		x	x		x	x	x
Create		x	x			x	x
Update		x	x			x	x
Delete		x	x			x	x
Approve			x				x
Cancel			x				x
Reject			x				x

In order to do non-read operations for a name entry, a user has to have user authority. Work for a name entry is a one step process. An entry is created, updated or deleted.

In order to do non-read operations for a structure entry, a user has to have user or administrator authority. Work for a structure entry is a two step process. The first step is making a proposal and the second step is handling the proposal. In the first step, a proposal is made for an entry to be created, updated or deleted. In the second step, the proposal is handled as it is either approved, cancelled or rejected.

Name and structure entries are to relate to active entries at time of create or update in order to be handled. An active entry is approved, latest in its line of uuid and not deleted. Once an item is approved, latest and deleted, it has reached its end of line and can not be further processed. An entry prior to approved and latest is considered obsolete and will not be shown unless history is requested.

Before entries or proposal for entries are processed, content in entries are validated.

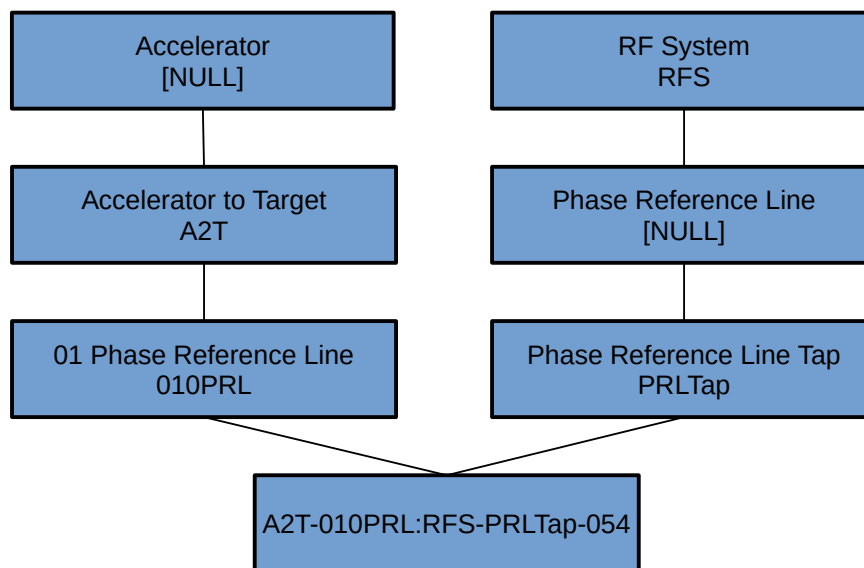
Notifications about changes for names and structures are sent to users and administrators by email.

Particular rules apply.

- names & structures
 - validation
 - input is checked such that it has expected content for each operation and that data is accepted, by itself and in relation to other data
 - ESS Naming convention governs rules
 - when a system structure entry is created – a proposal to create that is approved
 - a name referring to system structure entry is created unless it already exists – mnemonic, mnemonic path
 - when a structure entry is updated – a proposal to update that is approved
 - all related names are updated if mnemonic for structure entry is changed
 - when a structure entry is deleted – a proposal to delete that is approved
 - all sub entries are deleted – for any depth
 - all related names become legacy names, i.e. can not be further processed except deleted
- notification
 - names
 - sent as daily email to administrators for changes in previous day
 - created, updated, deleted
 - structures
 - sent as email to administrators for changes after operation
 - created, updated, deleted
 - approved, cancelled, rejected – copy to users for proposals

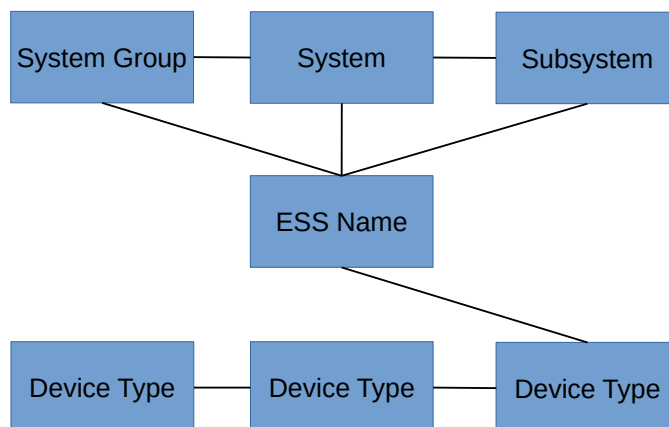
Domain model & Example

System structure	System Group	Accelerator		Level 1
	System	Accelerator to Target	A2T	Level 2
	Subsystem	01 Phase Reference Line	010PRL	Level 3
Device structure	Discipline	RF System	RFS	Level 1
	Device Group	Phase Reference Line		Level 2
	Device Type	Phase Reference Line Tap	PRLTap	Level 3
ESS name	Index		054	
ESS name	A2T-010PRL:RFS-PRLTap-054			



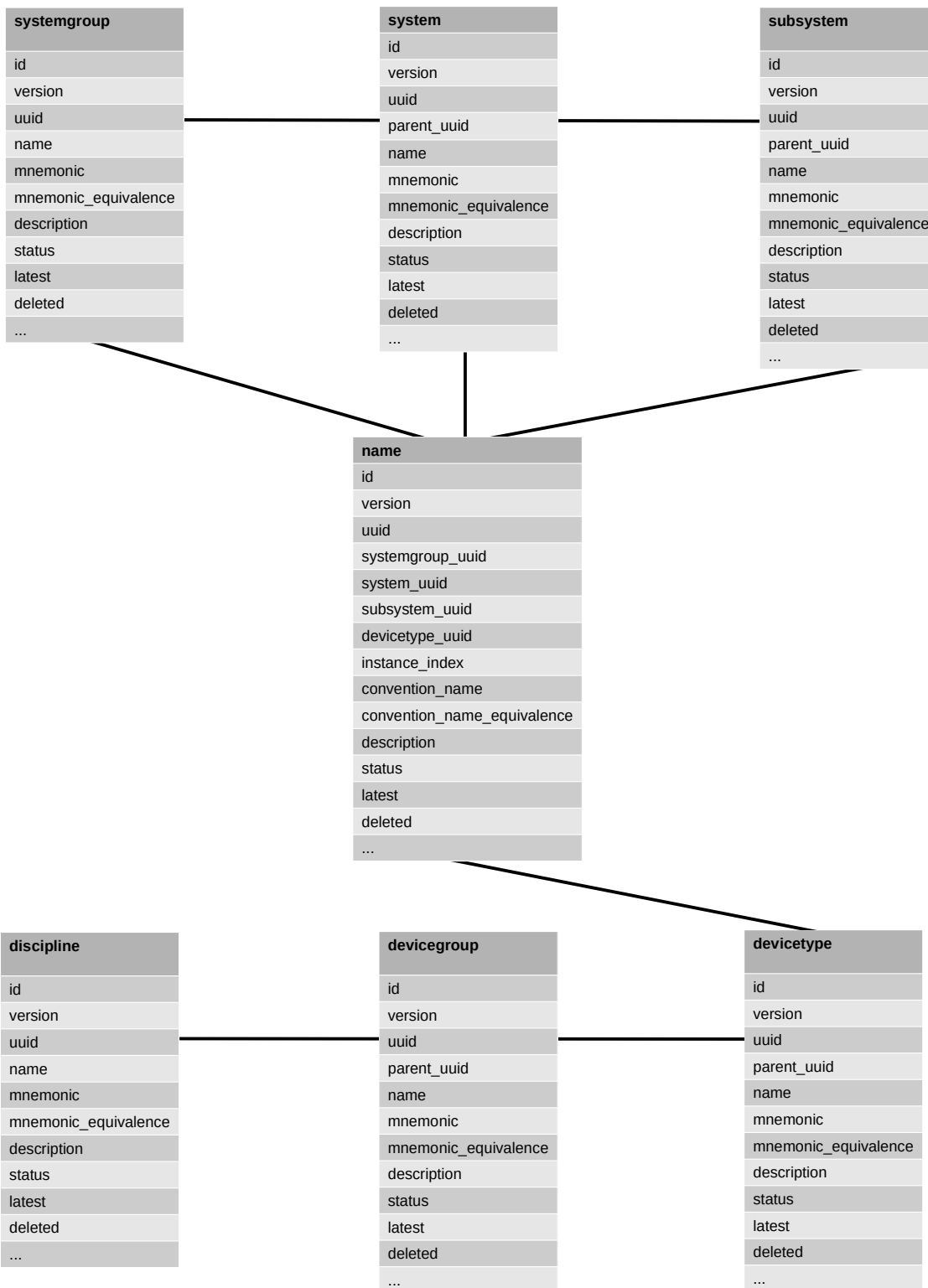
Database

Database is designed to correspond to domain model.



A name must refer to exactly one System structure, either level 1 or level 2 or level 3.

A name may refer to one Device structure, level 3.



Core attributes are uuid, mnemonic, instance index together with status, latest, deleted. Each table keeps history for its entries. In addition, an entry also has username, date and comment associated with each change.

See other folders and files for information about tables and columns (also migration).

- [docs/developer/refactoring/background_thoughts_database](#)

Get up & running

It is recommended to have local docker-compose files for various purposes, e.g.

- docker-compose-local.yml
- docker-compose-local-database.yml

.gitignore contains *docker-compose-local** → allows for local docker-compose files that are not committed

See docker-compose.yml, docker-compose-integrationtest.yml for examples.

Option to have local docker-compose file to

- set database to desired state (version) and content
- set environment variables for database and application

Make sure passwords and other sensitive information are not committed.

Database

To consider

- database version & tables, columns
 - in particular if/when Flyway is not enabled. Then it's necessary to ensure proper database version with proper scripts are run.
- content when running and/or testing application.
 - empty vs content
 - backup vs prepopulated
 - example content or fill yourself through REST API (json or Excel)
- see
 - */src/test/resources/db/data/*
 - */src/test/resources/db/migration/schema_migration/*

Database needs to be up-to-date when testing and running application. In addition, it needs to have content in order to be useful. This can be ensured in various ways.

- backup
- scripts without data
- scripts with data
- Excel
- integration tests
- do it yourself

It is possible to use docker-compose files for this purpose, e.g.

- (1) – computer path `./src/test/resources/db/schema_migration`
- (2) – docker container path `/docker-entrypoint-initdb.d`

```
postgres:
  volumes:
    - (1)/V1__Initial.sql:(2)/V1__Initial.sql
    - (1)/V2__Commit_Msg_to_Device.sql:(2)/V2__Commit_Msg_to_Device.sql
    - (1)/V3__Notification_CC_List.sql:(2)/V3__Notification_CC_List.sql
    - (1)/V4__Schema_data_migration.sql:(2)/V4__Schema_data_migration.sql
```

- (1) – computer path `./src/test/resources/db/data`
- (2) – docker container path `/docker-entrypoint-initdb.d`

```
postgres:
  volumes:
    - (1)/dump-discs_names_namesit.sql:(2)/dump-discs_names_namesit.sql
```

It is possible to use Excel for this purpose, i.e.

- `/src/main/resources/static/templates/StructureElementCommand.xlsx`
- `/src/main/resources/static/templates/NameElementCommand.xlsx`

It is possible to use (Docker) integration tests for this purpose. By debugging an integration test, database content can be exported or backup taken at a debug point.

Application

Compile

/README.md

Test

/README.md

Test may be done in various ways.

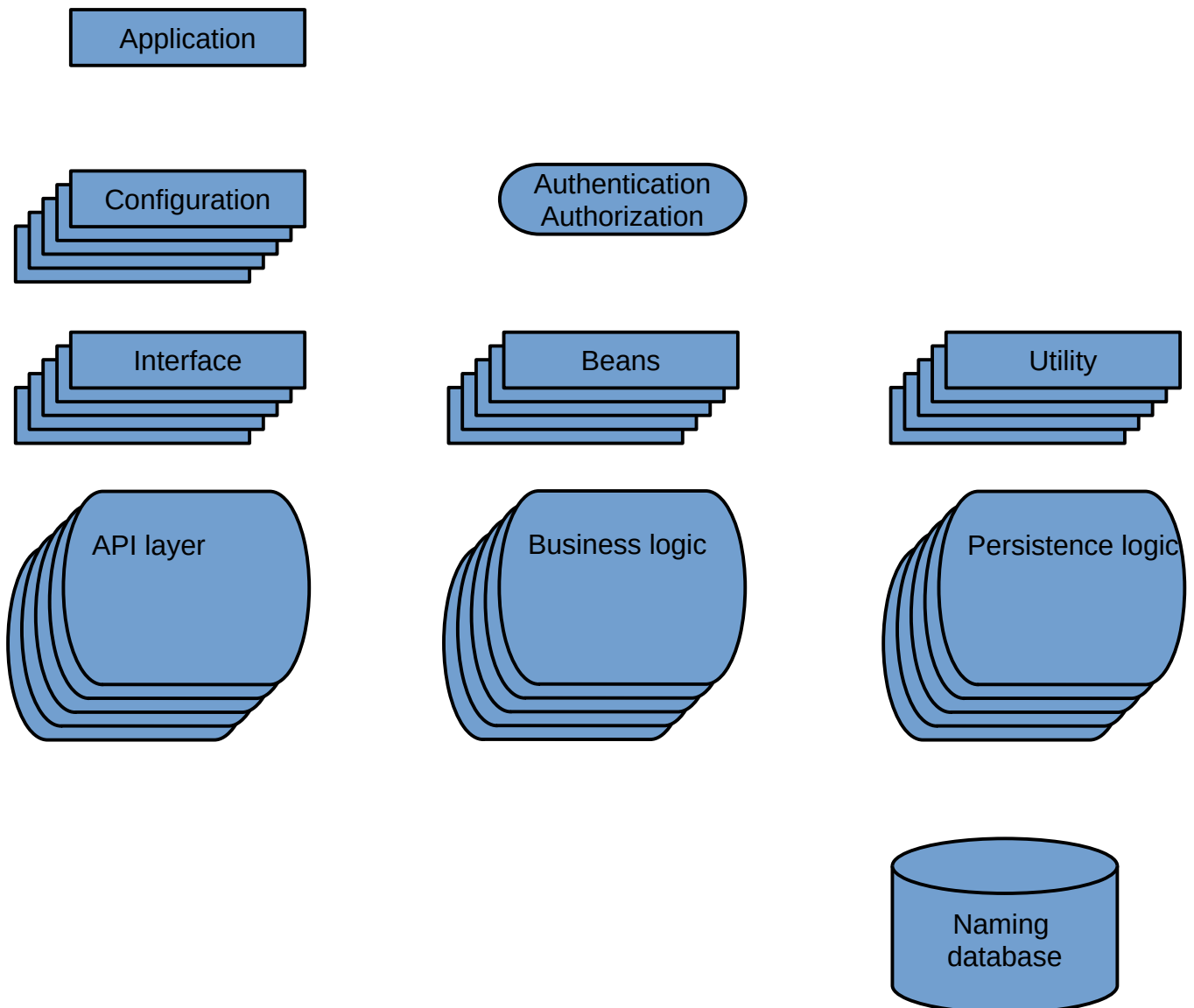
- Unit tests
- Integration tests with Docker containers
 - Tests may be debugged. Docker containers, application and database may be inspected for logs and content.
- Run application and database, and perform operations to Create Read Update Delete + Approve Cancel Reject.
- In local computer or at test server

Run

/README.md

Architecture & code

Naming backend is a web application implemented as a REST style web service backed by a relational database. The web service is implemented as a Spring Boot application and the database is available as PostgreSQL.



At start of application, various tasks such as setting up configuration and authentication / authorization is done. Depending on how application is started, database may also be started.

Application performs tasks for names and structures. There are Create Read Update Delete + Approve Cancel Reject for names and structures. A task is usually initiated by a client which can be both user and non-user.

In general, a request from a client to Naming backend is defined in an interface and implemented in API layer. Such separation is useful for version handling of request URLs. There are also URLs that are not versioned, e.g. healthcheck. Some URLs are protected in which case processing passes through authentication and authorization.

In flow for handling a request, the API layer implementation splits request into parts and handle each in sequential order. In a request to store information, data is read, validated, stored and finally result of request is returned. Validation and storing are implemented as Business logic. Storage itself, including having code that matches database and content, is available as Persistence logic. There are utilities to support mentioned layers, e.g. handling of input and output, error.

Validation is split into parts that handle input itself and data being correct.

Architecture mapped to code

Examples of mapping for parts of application, from architecture to code.

<i>Piece of architecture</i>	<i>What</i>	<i>Packages / Folders / Files</i>
Application		org.openepics.names
Configuration		/src/main/resources/ application.properties
Database		/src/main/resources/db/migration
ESS Naming Convention	Rules Validation	org.openepics.names.utility ESSNamingConvention NamingConvention ValidateNameElementUtil ValidateStructureElementUtil ValidateUtil
Beans	REST API Definition Data	org.openepics.names.rest.beans
Interface	REST API Definition Requests	org.openepics.names.rest.api.v1 INames IStructures
API layer	REST API Implementation Requests	org.openepics.names.rest.controller NamesController StructuresController
Business logic	Service Coordination	org.openepics.names.service NamesService StructuresService
Persistence logic	Repository Storage Model & mapping	org.openepics.names.repository org.openepics.names.repository.model
Unit test	Test of utilities	org.openepics.names.util
Integration test	Test of application & database	org.openepics.names.docker org.openepics.names.docker.complex

Core part of application is handling ESS Naming Convention

Rules

```
org.openepics.names.utility
    ESSNamingConvention
    NamingConvention
```

Validation

```
org.openepics.names.utility
    ValidateNameElementUtil
    ValidateStructureElementUtil
    ValidateUtil
```

use of rules

Integration tests with Docker containers are core to test of implementation

Parts of application, including above mentioned, are explored in more detail in sections that follow.

Business logic contains necessary means to handle transactions that are necessary. This is done in close cooperation with Persistence logic that handles storage.

A request mapped to code

E.g. a request to create new names by uploading Excel sheet

(simplified)

INames

```
createNames(MultipartFile)
```

Interface

NamesController

```
createNames(MultipartFile)
    ExcelUtil.hasExcelFormat
    ExcelUtil.excelToNameElementCommands
    NamesService.validateNamesCreate
    NamesService.createNames
    return ExcelUtil.nameElementsToExcel
```

API layer

NamesService

```
validateNamesCreate(List<NameElementCommand>)
    for each element
        validateNamesCreate(NameElementCommand)
            ValidateNameElementUtil.validateNameElementInputCreate
            ValidateNameElementUtil.validateNameElementDataCreate

createNames(List<NameElementCommand>)
    for each element
        createName(NameElementCommand)
            name = ...
            NameRepository.createName
    return createdName
```

Business logic

NameRepository

```
createName(Name)
    EntityManager.persist(name)
```

Persistence logic

ExcelUtil

```
hasExcelFormat(MultipartFile)
excelToNameElementCommands(InputStream)
```

Utility

ValidateNameElementUtil

```
validateNameElementInputCreate(NameElementCommand)
validateNameElementDataCreate(NameElementCommand)
```

Utility

Validation of data is tightly matched to ESS Naming Convention.

Implementation

Configuration

Spring Boot configuration is set through default values that may be overridden. This may be done with values in files or in terminal, at compile time or runtime, for test or deploy, by user or tool.

Spring Boot externalized configuration

<https://docs.spring.io/spring-boot/docs/current/reference/html/features.html#features.external-config>

Environment variables are set to default values in

`/src/main/resources/`

`application.properties`

Application and database may be started separately, with or without Docker, and configuration, set by environment variables, docker compose files and `application.properties` must reflect this. An example of this is data source which has configuration for name, username, password and hostname, ip address, port. Configuration must not be in conflict with environment in which application and database run.

If Docker containers are used, they may communicate on a Docker-internal network and configuration for containers is to reflect this. E.g. it is important that proper names and ports for Docker containers are used in order for containers to be able to communicate.

In case of Docker containers being used, configuration may be used to expose or hides parts of containers to /from access from outside of Docker internal network. An example of this is database, which may be exposed or hidden through configuration of ports of database container.

Database

Database scripts are available

`/src/main/resources/db/migration/`

The database scripts ensure that database is set and migrated to expected state. They may be used not only for running the application but also in tests.

ESS Naming Convention

The implementation of ESS Name Convention is core of Naming application. This ensures that names and structures follow rules which are interpretations of ESS Naming Convention into code.

`/src/main/java/`

`org.openepics.names.util.EssNamingConvention`
`org.openepics.names.util.NamingConvention`

implementation
definition

Rules & methods

<u>equivalenceClassRepresentative</u>	Return equivalence class representative for given name. This is used to ensure uniqueness of names when treating similar looking names, e.g. 0 vs. O, 1 vs. l treated as equal.
isInstanceIndexValid	Return if the convention name's instance index is valid according to convention rules, in the context of system structure and device structure.
isMnemonicPathValid	Return if mnemonic path is valid within the application according to the convention rules.
<u>isMnemonicRequired</u>	Return if the mnemonic is required or if it can be null, i.e. mnemonic not part of the name.
validateMnemonic	Return validation for given type and mnemonic according to convention rules.

The rules are applied at time of Create Update + Approve Cancel Reject and are called from Business logic. The actual implementation is Business logic → validation → rules.

There are different kinds of validation that each serves a purposes.

- input, by itself. E.g. value present, proper format for uuid
- data, according to rules, by itself. E.g. mnemonic (System structure, Device structure), instance index (Name)
- data, according to rules, in relation to other data. E.g. mnemonic not present, no duplicates in mnemonic path, instance index not present

Validation

/src/main/java/

org.openepics.names.util.ValidateNameElementUtil	<i>names</i>
org.openepics.names.util.ValidateStructureElementUtil	<i>structures</i>
org.openepics.names.util.ValidateUtil	<i>common</i>

Beans

Beans encapsulate content received from and sent to Naming client.

/src/main/java/

```
org.openepics.names.beans
org.openepics.names.beans.element
org.openepics.names.response
```

REST API

REST API is definition and implementation of what can be done from client perspective. It is split into parts, Interface and API layer. This code handles request and reply for Naming.

First and foremost are names and structures. In addition, there are options to handle healthcheck, report and verification of data. Some of those options need not be versioned while names and structures need to, and are, being versioned.

Operations for names and structures are Create Read Update Delete + Approve Cancel Reject.

Introduction

</docs/about/>

[naming_rest_api_brief_introduction](#)

[naming_rest_api_cheat_sheet](#)

[naming_rest_api_brief_introduction](#) contains list of REST API endpoints with paths, description, authorization (if required).

In addition to REST API endpoints listed in above document, there are also endpoints available for validation for names and structures. Such endpoints are public and available although not visible in Swagger UI. Reason for validation endpoints not being visible is that they may cause confusion for end users. Mentioned endpoints are however used in implementation of POST, PUT, PATCH, DELETE methods behind the scenes and for Docker integration tests.

Validation endpoints

Names

Path </api/v1/names>

HTTP method	Path & Query string	Description
GET	/validatecreate	Return if names are valid to create by list of name element commands. If names are valid to create, successful create of names can be expected.
GET	/validateupdate	Return if names are valid to update by list of name elements. If names are valid to update, successful update of names can be expected.
GET	/validateupdate	Return if names are valid to delete by list of name element commands. If names are valid to delete, successful delete of names can be expected.

Structures

Path /api/v1/structures

HTTP method	Path & Query string	Description
GET	/validatecreate	Return if structures are valid to create (propose) by list of structure element commands. If structures are valid to create, successful create of structures can be expected.
GET	/validateupdate	Return if structures are valid to update (propose) by list of structure element commands. If structures are valid to update, successful update of structures can be expected.
GET	/validatedelete	Return if structures are valid to delete (propose) by list of structure element commands. If structures are valid to delete, successful delete of structures can be expected.
GET	/validateapprove	Return if structures are valid to approve by list of structure element commands. If structures are valid to approve, successful approve of structures can be expected.
GET	/validatecancel	Return if structures are valid to cancel by list of structure element commands. If structures are valid to cancel, successful cancel of structures can be expected.
GET	/validatereject	Return if structures are valid to reject by list of structure element commands. If structures are valid to reject, successful reject of structures can be expected.

Definition

/src/main/java/

org.openepics.names.rest.api.v1

Implementation

/src/main/java/

org.openepics.names.rest.api.controller

Upon request, information is received from path and query string in URL in combination with json or Excel. Upon completion of request, information in reply is sent with json or Excel.

For Naming to act upon information – create, update, delete + approve, cancel, reject – not all of available information is required. It is enough for client to send subset of information. E.g. to create a name, a client may send uuid for references to System structure and Device structure, instance index and comment. Naming will then construct name based on received (command) information.

When sending reply to client from Naming, more information such as generated name, timestamp and other values is included in reply beside information in request.

The same applies for all requests and replies, whether information is sent with json or Excel.

Objects in request that contain information to act upon

/src/main/java/

org.openepics.names.rest.beans.element.NameElementCommand

org.openepics.names.rest.beans.element.StructureElementCommand

Objects in reply that contain information

/src/main/java/

org.openepics.names.rest.beans.element.NameElement

org.openepics.names.rest.beans.element.StructureElement

Other responsibilities handled in REST API

- authentication & authorization
- error handling

Excel

Guide to use of Excel to upload names and structures to Naming together with download of names and structures from Naming is available together with Excel templates. Columns in templates correspond to command objects.

Guide

/docs/about/

naming_rest_api_excel_guide

Templates

/src/main/resources/static/templates/

NameElementCommand.xlsx

StructureElementCommand.xlsx

For upload requests that contain Excel, reply contain Excel corresponding to above. Such Excel template is not available but is generated. For download requests that contain Excel, such templates are also not available but is generated.

API layer calls Excel utility methods to read names and structures from Excel or write names and structures to Excel.

API layer

/src/main/java/

org.openepics.names.rest.controller.NamesController

org.openepics.names.rest.controller.StructuresController

Excel utility

```
/src/main/java/
```

```
org.openepics.names.rest.util.ExcelUtil
```

Open API & Swagger UI

REST API documentation (part of) is available through Swagger UI. This is set up in definition of REST API endpoints. Such documentation include path, content types, summary, description, response codes and content.

Business logic

This is service and coordination layer for what is required to retrieve, package and store information.

```
/src/main/java/
```

```
org.openepics.names.rest.api.service
```

For request to retrieve information, it retrieves data from persistence logic and then transform data to such format that is suitable for reply to client.

For request to store information, it translates input such that it may be stored by persistence logic.

Persistence logic

This is repository, model and mapping, and storage layer that handles access to database

```
/src/main/java/
```

```
org.openepics.names.rest.api.respository
```

```
org.openepics.names.rest.api.respository.model
```

```
/src/main/resources/
```

```
application.properties
```

Database tables and columns are mapped to persistence model objects which then are used in application for easier handling. In practice, Hibernate is used, although in configuration only.

Tests

There are tests for various purposes, unit tests and integration tests. This includes tests for ESS Naming Convention in part and as a whole. Such tests contain verification of, and are as important as implementation of, ESS Naming Convention.

It is also possible, and recommended, to test application locally and at test server.

```
/src/test/java/
```

```
org.openepics.names
```

1

```
org.openepics.names.docker
```

2

<code>org.openepics.names.docker.complex</code>	3
<code>org.openepics.names.util</code>	4

1 contains `@SpringBootTest` which loads Spring application context. This is auto-generated and may be used for integration tests but is currently not used in that way. By loading application context, it is ensured that various definitions are as expected, e.g. that access methods to repository layer, that are defined in interfaces, go together with repository model.

4 contains unit tests for `org.openepics.names.util` package. This includes tests for `org.openepics.names.util.EssNamingConvention` together with a range of tests for import utility methods.

2, 3 contain integration tests with Docker containers.

Recommendation to learn how tests work. It will be valuable in subsequent work.

Integration tests with Docker containers

These are JUnit tests that start a docker container for the application (Naming backend) and another docker container for the database (Postgresql) through `docker-compose-integrationtest.yml`.

```
@Container
public static final DockerComposeContainer<?> ENVIRONMENT =
    new DockerComposeContainer<>(new File("docker-compose-integrationtest.yml"))
        .waitingFor(ITUtil.NAMING, Wait.forLogMessage(".*Started NamingApplication.*", 1));
```

Thereafter a number of Http requests (GET) and curl commands (POST, PUT, PATCH, DELETE) are run towards the application to test REST API (CRUD - create, read, update, delete) and replies are received and checked if content is as expected.

Before integration tests may be run, application must be compiled.

At start of each test class (with suffix IT), before any test is run, environment according to docker compose file is started. File specifies a container for application and another container for database. Application container awaits database container being ready before test environment is ready.

Integration tests in test class are then run in no particular order.

E.g. outline of integration test with Docker to create and approve discipline (simplified)

```
StructureElementCommand structureElement = null;
StructureElement createdStructureElement = null;

structureElement = new StructureElement(
    null, Type.Discipline, null,
    "name", "Ca",
    "description", "name"
);

ITUtilStructureElement.assertExists (Type.DISCIPLINE, "Ca", Boolean.FALSE);
ITUtilStructureElement.isValidToCreate(Type.DISCIPLINE, "Ca", Boolean.TRUE);

ITUtilStructureElement.assertValidate(structureElement, StructureChoice.CREATE, Boolean.TRUE);
ITUtilStructureElement.assertValidate(structureElement, StructureChoice.APPROVE, Boolean.FALSE);

// create
createdStructureElement = ITUtilStructureElement.assertCreate(structureElement);
structureElement.setUuid(createdStructureElement.getUuid());

ITUtilStructureElement.assertExists (Type.DISCIPLINE, "Ca", Boolean.TRUE);
ITUtilStructureElement.isValidToCreate(Type.DISCIPLINE, "Ca", Boolean.FALSE);

ITUtilStructureElement.assertValidate(structureElement, StructureChoice.CREATE, Boolean.FALSE);
ITUtilStructureElement.assertValidate(structureElement, StructureChoice.APPROVE, Boolean.TRUE);

// approve
ITUtilStructureElement.assertApprove(structureElement);

ITUtilStructureElement.assertExists (Type.DISCIPLINE, "Ca", Boolean.TRUE);
ITUtilStructureElement.isValidToCreate(Type.DISCIPLINE, "Ca", Boolean.FALSE);

ITUtilStructureElement.assertValidate(structureElement, StructureChoice.CREATE, Boolean.FALSE);
ITUtilStructureElement.assertValidate(structureElement, StructureChoice.APPROVE, Boolean.FALSE);
```

In above example, variables are defined and example data is created. A number of Http requests (GET) are run towards the application to check if data exists and if data is valid to create. Example data is converted to Json and sent to server to be validated. It is expected that discipline does not exists, that it is valid to be created and that data is valid.

Server is then invoked to create discipline. It is then expected that discipline exists but is not approved, and that it is not valid to be created another time.

Server is then invoked to approve discipline. It is then expected that discipline exists and is approved.

There are integration tests designed to test Create Read Update Delete + Approve Cancel Reject for names and structures.

Purpose of tests is to ensure REST API works as expected for parts and as a whole.

As part of effort to design and write integration tests, various test utilities have been developed and are available in same packages as integration tests. The purpose of test utilities is to simplify tests and make them more clear and easier to understand.

How to go about tasks

Recommendation to checkout separate branch for all work except only run application.

Ensure branch point is from proper commit.

Have well crafted branch names that match work at hand.

Test before commit & push. This includes unit tests and integration tests.

All integration tests may take 10 minutes to run and, unless all are required, subset of integration tests may be used.

Repository

Branch handling

If one or very few developers are active, main branch may be used for work, including merges, before content is pushed to repository. Otherwise recommendation is to do merge and similar operations in Gitlab.

Deploy

Deploy to servers are done in Gitlab through CI/CD section.

Reference

Naming convention

- <https://chess.esss.lu.se/enovia/link/ESS-0000757/21308.51166.45568.45993/valid>

Confluence

- <https://confluence.esss.lu.se/display/SW/Naming+Tool%2C+Cable+DB%2C+CCDB%2C+IOC+Factory%2C+RBAC>

Repository – Gitlab

- <https://gitlab.esss.lu.se/ics-software/naming-backend>
 - */docs/about/*
 - */docs/developer/refactoring/*

Test server

- <https://naming-test-02.cslab.esss.lu.se/swagger-ui/index.html>