

---

# **ehrbase Documentation**

**Birger Haarbrandt**

**Oct 12, 2021**



---

## Contents:

---

<b>1</b>	<b>Release Notes</b>	<b>3</b>
<b>2</b>	<b>Getting Started</b>	<b>5</b>
2.1	openEHR Introduction . . . . .	5
2.2	Step 1: Data Models . . . . .	8
2.3	Step 2: Upload a Template . . . . .	15
2.4	Step 3: Create an EHR . . . . .	15
2.5	Step 4: Load Data . . . . .	16
<b>3</b>	<b>Development</b>	<b>21</b>
3.1	Developing . . . . .	21
3.2	Testing . . . . .	21
3.3	Deploying . . . . .	32
3.4	Docker Images . . . . .	32
3.5	Technical Documentation . . . . .	40
3.6	Security . . . . .	49
3.7	Admin API . . . . .	49
3.8	Status and Metrics . . . . .	57
3.9	Attribute-based Access Control . . . . .	64
<b>4</b>	<b>SDK</b>	<b>69</b>
4.1	Guides . . . . .	69
4.2	Reference . . . . .	70
<b>5</b>	<b>Load Testing</b>	<b>79</b>
5.1	Testehr . . . . .	79
5.2	Script execution . . . . .	80
<b>6</b>	<b>FHIR Bridge</b>	<b>81</b>
6.1	Overview . . . . .	81
6.2	Installation . . . . .	83
6.3	Database for Audit Logs in FHIR Bridge . . . . .	84
6.4	Do the mapping . . . . .	84
6.5	Flows . . . . .	88
<b>7</b>	<b>Terminology Validation</b>	<b>91</b>
7.1	Introduction . . . . .	91

7.2	Configuration . . . . .	92
<b>8</b>	<b>Indices and tables</b>	<b>95</b>



For now, there is only the Release Notes. We will add detailed documentation as soon as possible.



# CHAPTER 1

---

## Release Notes

---

The following pages show the release notes





**Warning:** WIP

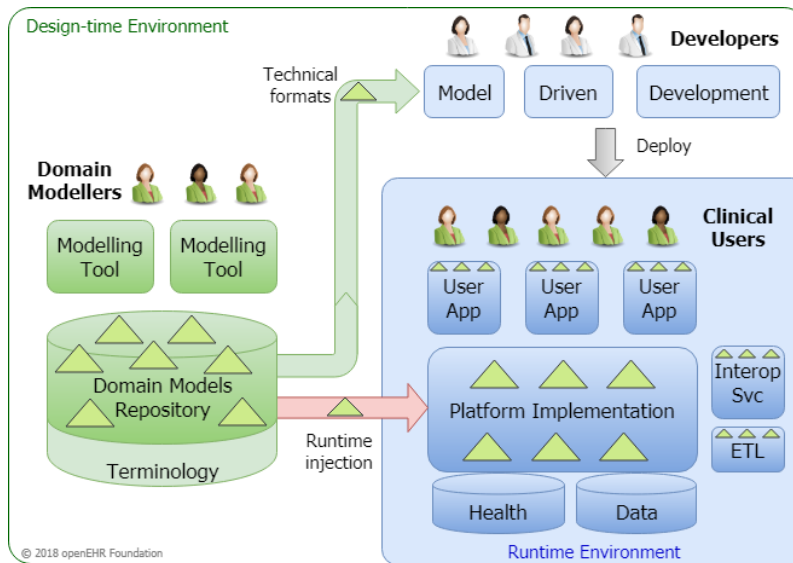
This chapter aims to give an overview an “hello world” example for software developers to build applications based on EHRbase and the openEHR specification. While we use EHRbase as the backend, the example will actually run against every other conformant openEHR implementation that implements the official openEHR REST API.

## 2.1 openEHR Introduction

**Warning:** WIP

openEHR is an open platform specification. From a practical perspective you can think about it as an electronic health record that consists of a database that is wrapped with a service layer. The database itself provides only a basic architecture and does not define the clinical content. This is done in a separate modelling layer. Hence, from a developer’s perspective, openEHR can be understood as a model-driven software development approach based on an adaptive database that can consume new data definitions at runtime. This allows to manage the high complexity of the medical domain.

As of now, openEHR defines the service access layer based on REST. However, there could be other protocols used in the future as the underlying openEHR datamodel is agnostic in terms of API definition. The following figure gives a high-level summary of the approach:



The model-driven openEHR technology ecosystem

The above figure shows the basic concept of separating the clinical definitions on the left side, from the technical implementation inside the platform (which is EHRbase in our case) on the right side.

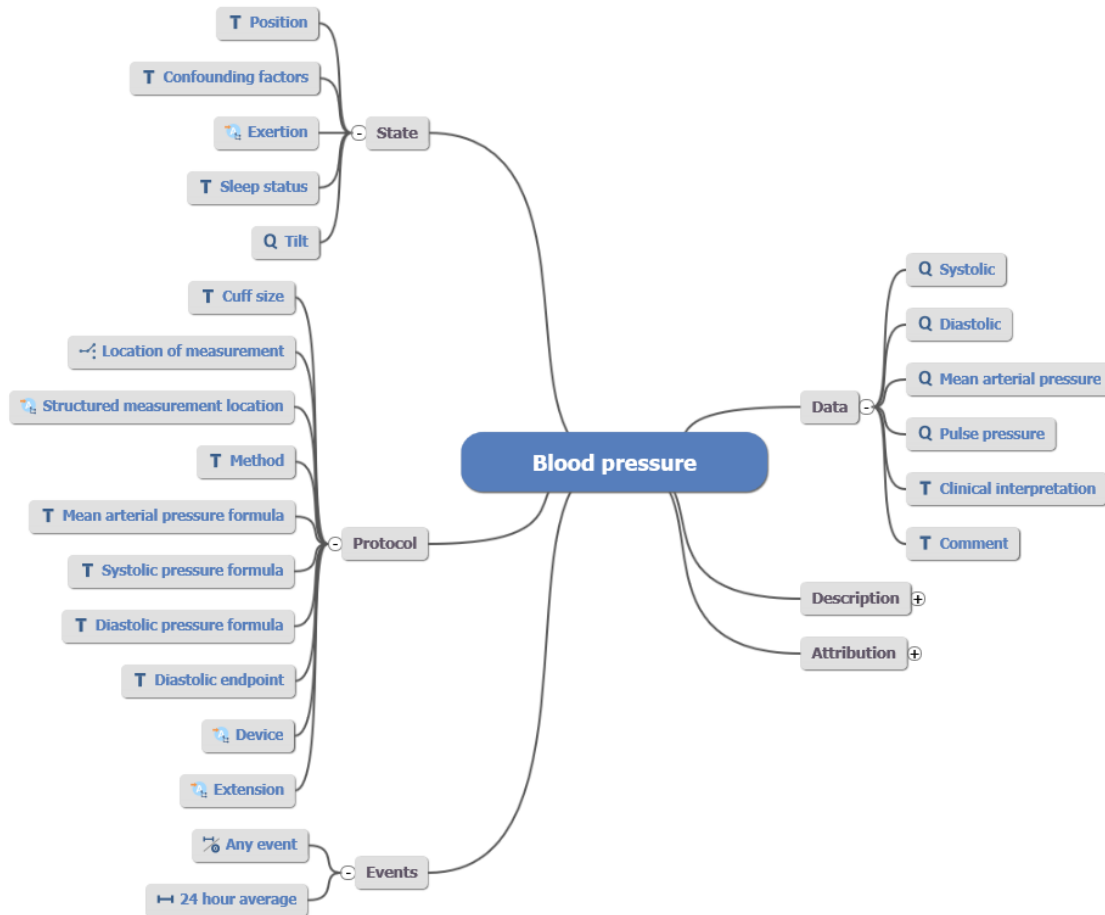
Domain experts define the clinical information models (called Archetypes), which are re-usable models of clinical concepts. Archetypes follow a formalism called Archetype Definition Language, that allows to flexibly model clinical concepts. There are several tools that can be used to create Archetypes in ADL 1.4:

- [Archetype Editor](#)
- [LinkEHR Editor](#)
- [ADL Designer](#) (web only)

The creation of Archetypes is a topic for itself and we will provide another tutorial. Note, that normally system developers should not be too much concerned with the definition and management of Archetypes, as this is the domain of medical information managers and medical professionals.

The goal of Archetypes is to provide standardized sets of data elements and their relations to achieve defined patterns in structured medical documentation. Hence, Archetypes need a strict government to fulfil their potential of enabling semantic interoperability. These models can already contain references to clinical terminologies (e.g. LOINC or SNOMED CT) and particular value sets.

The following image shows the mindmap representation of an Archetype to store data about a blood pressure measurement:



It is obvious that this model is very detailed. This is because Archetypes aim to capture the requirements of different medical specialities. This means, that use-cases from a simple measurement at the general practitioner as well as a detailed assessment through a cardiologist needs to be supported. Normally, the full richness of the model will be reduced before usage in a real-world application.

The governance of Archetypes happens inside a domain model repository. The most commonly used tool is the [Clinical Knowledge Manager \(CKM\)](#). For international standardization efforts, the CKM is the first address to go to. As local needs cannot be avoided, there are also national instances of the Clinical Knowledge Manager, for example in [Germany](#) or [Norway](#)

To represent actual clinical use-cases, elements from Archetypes need to be combined inside a Template. You can think of Templates as data sets that can be used to capture data in a form. To create a Template, there are currently two tools available:

- [Template Designer](#)
- [ADL Designer](#) (web only)

We will soon add another tutorial to give some more details about the creation of Templates. The Template Designer and the ADL Designer have an export format called Operational Template (OPT). This format is used to inject the use-case specific definitions (that are based on Archetypes) into the openEHR platform (like EHRbase).

This can be done using the [POST Template Endpoint](#) of the openEHR REST API.

Now we can take a look at the clinical applications that are based on the openEHR platform. Here, the approaches can differ. The challenge is that the openEHR Reference Model is quite technical a generic to provide optimal handling for computation like validation, storage and querying.

Hence, intermediate formats are often used to make life simpler for developers. In the case of EHRbase, we use the OPT files to enable data-driven development. In the [EHRbase Client Library](#) OPTs are used to automatically generate Java classes that can be used to easily build data instances. A data instance in openEHR is called a **composition**.

To allow easier handling, classes are automatically created from the OPT and are much easier for humans to handle. Once data is created, it is transformed to the canonical formats and sent to the openEHR server to a patient's electronic health record. The composition can either be sent alone or as part of a bigger transaction, called a **Contribution**, which can contain different operations on several objects inside the electronic health record, including compositions and folders.

On the server-side, it is checked that all elements inside the composition are valid according to the constraints that were defined in the respective Archetypes and the Template. Once the data has passed all tests, it is permanently stored within a patient's electronic health record. Normally, data can only be updated or logically deleted (in contrast to a physical delete) as electronic health records require a full audit trail about the patient data.

Once the data is stored, it can be retrieved through the openEHR REST API. The most common use-case is to fill user interfaces, for example to plot a list of the latest medications or lab values. This can be done using the Archetype Query Language, a model-based query formalism that only relies on definitions from the Archetypes.

## 2.2 Step 1: Data Models

**Warning:** WIP

Now, after we got an overview, it's time to put our hands on the tools. Though, if you want to skip this part of the tutorial to directly work with EHRbase, you can get the example files [here](#).

As a first step, we need to obtain the information models. As mentioned in the introduction, the Clinical Knowledge Managers are our first address. To download all Archetypes, go to the [International Clinical Knowledge Manager](#)

← → ↻ 🔒 openehr.org/ckm/

**openEHR**  
Clinical Knowledge Manager

Archetypes ▾ Templates ▾ Termsets ▾ Release Sets ▾ Projects ▾ Reports ▾ Help ▾

Dashboard 🔍 Find Resources

All Resources

Subdomain: All subdomains ▾

Project / Incubator: All projects ▾

Active Under review Published

Archetypes

- EHR Archetypes
  - Cluster
  - Composition
  - Element
  - Entry
    - Action
    - Evaluation
    - Observation
    - Instruction
    - Admin
    - Section
    - Structure
- Demographic Model Archetypes

Become a Part of Our Online Community

As a registered user, you can participate in the development of open and shared clinical content for eHealth projects as part of the CKM community collaboration.

FIND OUT MORE »

GET INVOLVED »

Register Today!

It only takes a minute to get started.

Register

What Do You Need to Know?

Archetypes Templates Termsets

Release sets Projects Incubators

FIND OUT MORE »

Quick Search

News

	Title
✓	Edmonton Symptom Assessment System Revised (ESAS-r) <i>Published archetype</i>
🔍	Sequencing assay <i>New archetype</i>
✗	Respirations <i>Deprecated archetype</i>
🔍	Service <i>Updated archetype</i>
🌐	Global Physical Activity

On the left side, you can find different categories of Archetypes, for example observations that contain data models like blood pressure, body temperature or Glasgow coma scale. For our tutorial, we want to get a copy of the archetypes from the Clinical Knowledge Manager.

Under Archetypes (marked in the image), you will find a function called *Bulk Export*.

[Dashboard](#) [Find Resources](#) **Bulk Export** ×











### Bulk Export Archetypes as ZIP



Project / incubator:

Reference model classes:


Created on or after:

Modified on or after:

<input type="checkbox"/>	States
<input checked="" type="checkbox"/>	 Initial / Predraft
<input checked="" type="checkbox"/>	 Draft
<input checked="" type="checkbox"/>	 Team review
<input checked="" type="checkbox"/>	 Review suspended
<input checked="" type="checkbox"/>	 Published
<input checked="" type="checkbox"/>	 Reassess (Draft)
<input checked="" type="checkbox"/>	 Reassess (Team review)
<input checked="" type="checkbox"/>	 Reassess (Review suspended)
<input type="checkbox"/>	 Rejected
<input type="checkbox"/>	 Deprecated

**REVISION**  
☐ Get latest trunk revision   
☒ Get latest published revision 

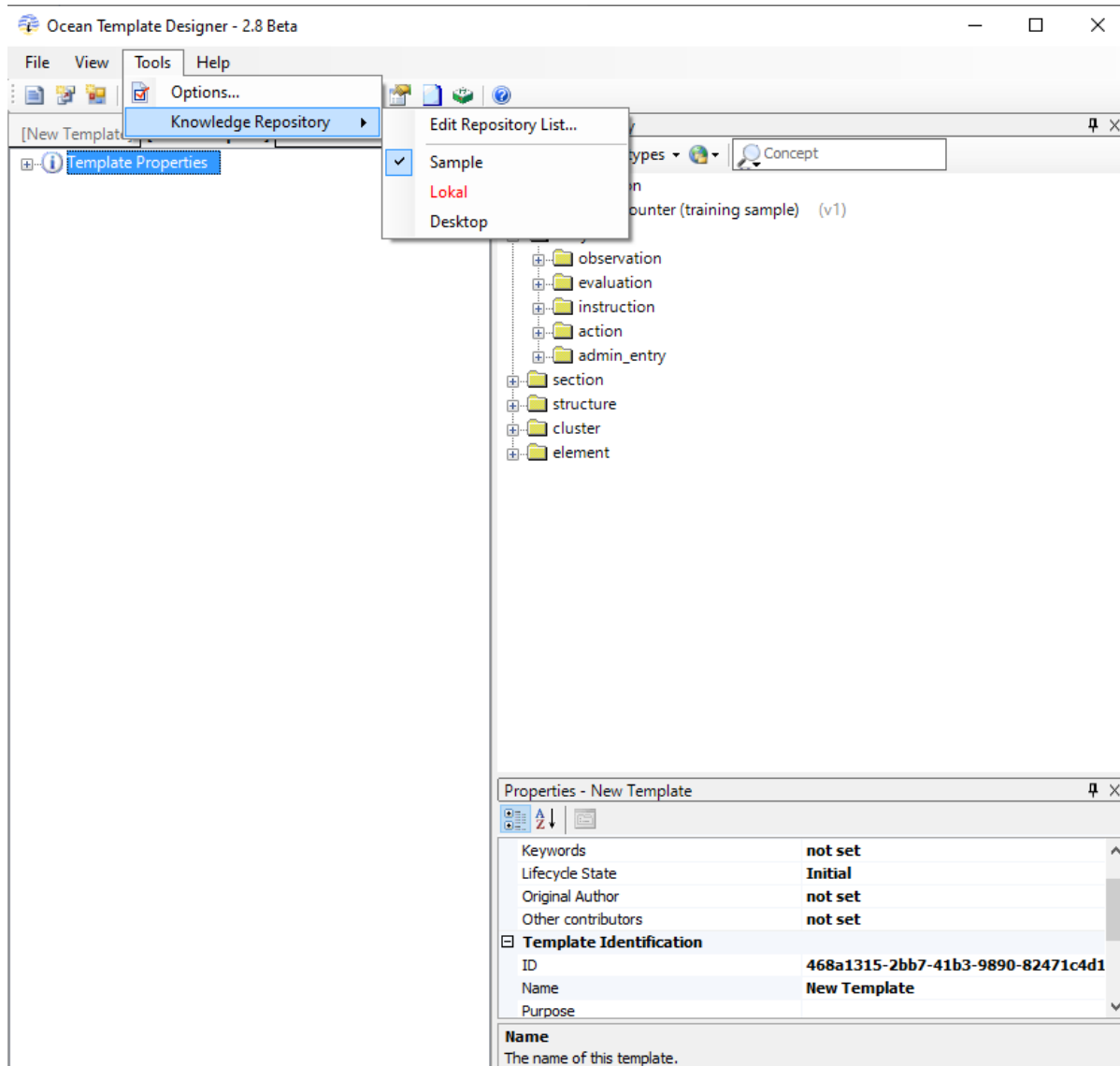
**FORMAT**  
☒ ADL  
☐ XML

 **Bulk Export**

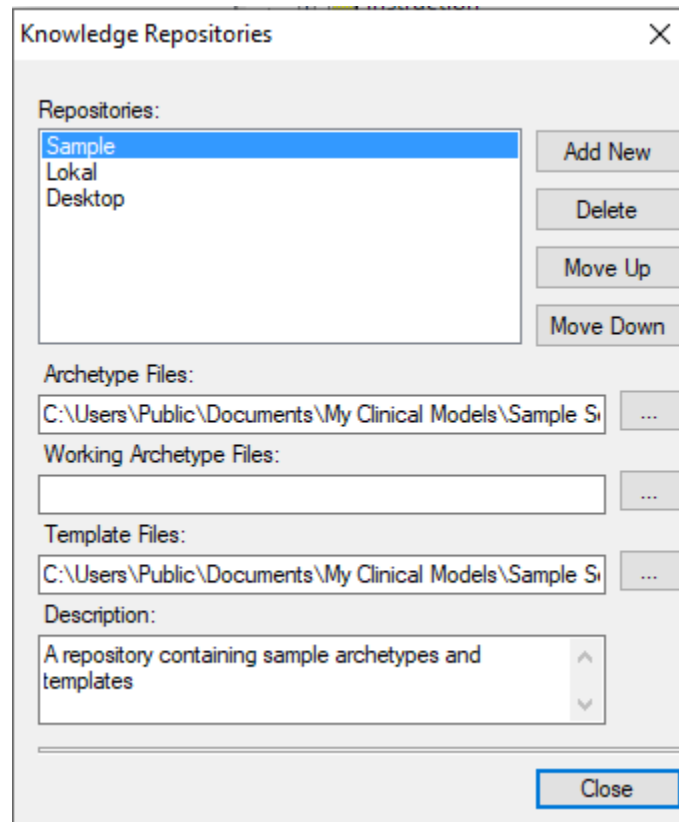
You can choose if the export should only contain Archetypes from a selected project or all and depending on its lifecycle status (published, draft etc.). Choose to get the latest published revision and use ADL (Archetype Definition Language) as export format. Clicking **Bulk Export** will then download a zip folder containing all Archetypes meeting the criteria.

Next, install the [Template Designer](#). The process should be straight forward (At least on Windows). Alternatively, the [ADL Designer](#) can also be used to create Templates by following [this guide](#).

Open the Template Designer. The first step is to configure a *Knowledge Repository*.

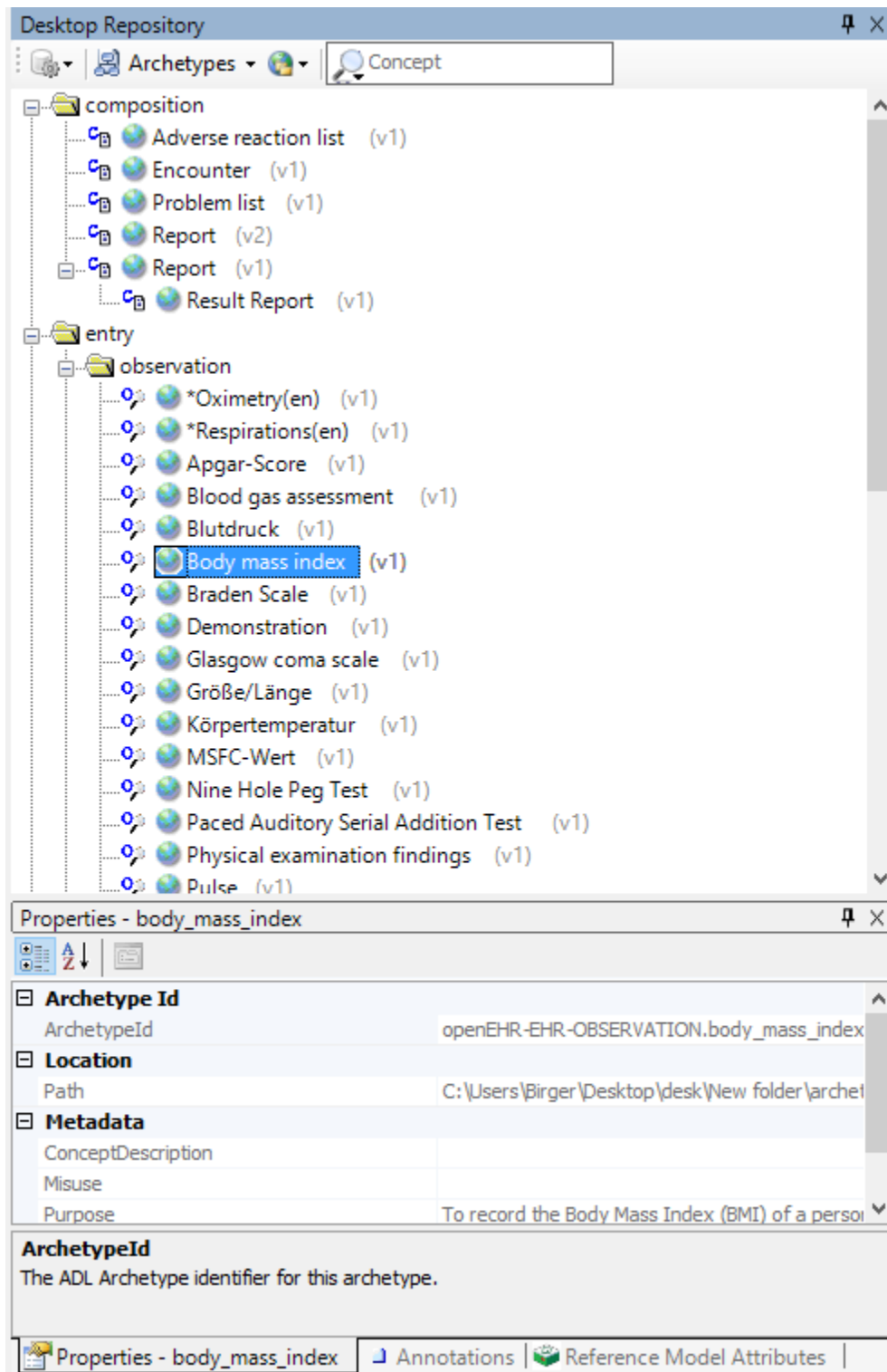


Click on *Edit Repository List*



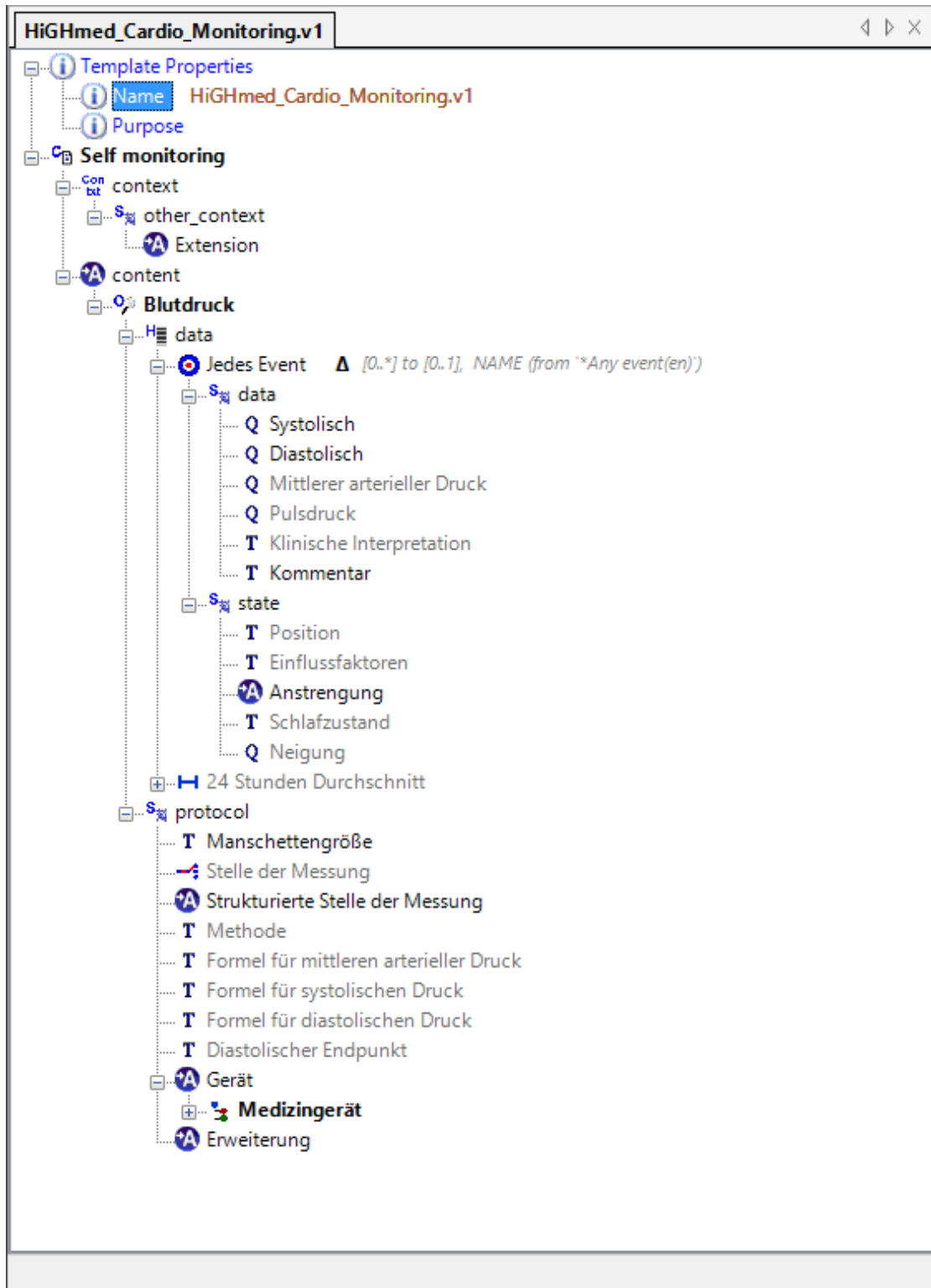
Set *Archetype Files* to the path where you unzipped the Archetypes you obtained through the Bulk Export. When you then select your new repository, the Archetypes should appear on the right window:





Now you can start to create your own Template. Typically, a Template needs an Archetype of type *Composition* as the root element. The *Composition* Archetypes provide the basic structure for the Template through *Slots* (which can be filled with Archetypes) and predefined metadata elements. In our example, we use the *Self Monitoring* Archetype. Just drag and drop the Archetype from the right panel to the left panel. Additionally, we add a blood pressure Archetype. Next, you can define further constraints on the particular elements, for example defining their cardinality, remove single elements, add terminology bindings etc.

We could also fill the slot within the blood pressure Archetype with a device Archetype to collect information about the device used for the measurement.



Finally, give your Template a name. Then you can Export the Template in the Operational Template (OPT) Format (File -> Export -> As Operational Template). This is all you need to upload your Template to EHRbase or any other openEHR server.

## 2.3 Step 2: Upload a Template

After we created the Template, it's time to put upload it to EHRbase. The data format that is uploaded to an openEHR server like EHRbase is called an Operational Template (OPT-File). You can find an example of an OPT here.

Please see <https://specifications.openehr.org/releases/ITS-REST/latest/definitions.html#definitions-adl-1.4-template-post> for the REST endpoint that you can use in EHRbase to POST a template. On a local instance of EHRbase, the URL should be like `localhost:8080/ehrbase/rest/openehr/v1/definition/template/adl1.4`

Copy the content of the OPT file into the body of the REST call. Make sure that the Content-Type attribute is set to `application/XML`. After the successful call, you should receive a 200 response code.

### 2.3.1 Client Library

As an alternative to directly using the REST API, the EHRbase Client Library provides functionality to provide a Template.

```
OPERATIONALTEMPLATE template = TemplateDocument.Factory.  
    ↳parse(OperationalTemplateTestData.BLOOD_PRESSURE_SIMPLE.getStream()).getTemplate();  
String templateId = "ehrbase_blood_pressure_simple.de.v" + RandomStringUtils.  
    ↳randomNumeric(10);  
template.getTemplateId().setValue(templateId);  
String actual = new DefaultRestTemplateEndpoint(cut).upload(template);
```

## 2.4 Step 3: Create an EHR

**Warning:** WIP

When you start EHRbase from scratch, you will find an empty electronic health record. OpenEHR has a patient-centric architecture. This means that all clinical information inside the database are associated with the EHR of a patient. Hence, the first thing to get started is the creation of an EHR for a patient.

Beware that demographic data of a patient (name, date of birth etc.) are not stored inside an openEHR system by design to ensure a clear separation from the clinical data. Hence, patients are not directly represented in openEHR but their electronic health record. In many cases, a separate demographics service (for example an IHE PIX/PDQ actor, a FHIR Server, an openEHR Demographics Repository or a custom solution) is used.

To create a new EHR, you can either directly use the openEHR REST API or a function within the EHRbase Client Library that encapsulates the REST call.

The REST API <https://specifications.openehr.org/releases/ITS-REST/latest/ehr.html#ehr-ehr-post>

### 2.4.1 REST

In this tutorial, we assume that we have a new patient coming to our organization. We simply make a REST call with an empty body.

```
{  
  "system_id": {  
    "value": "d60e2348-b083-48ce-93b9-916cef1d3a5a"  
  },  
}
```

(continues on next page)

(continued from previous page)

```

"ehr_id": {
  "value": "7d44b88c-4199-4bad-97dc-d78268e01398"
},
"ehr_status": {
  "id": {
    "_type": "OBJECT_VERSION_ID",
    "value": "8849182c-82ad-4088-a07f-48ead4180515::openEHRSys.example.com::1"
  },
  "namespace": "local",
  "type": "EHR_STATUS"
},
"ehr_access": {
  "id": {
    "_type": "OBJECT_VERSION_ID",
    "value": "59a8d0ac-140e-4feb-b2d6-af99f8e68af8::openEHRSys.example.com::1"
  },
  "namespace": "local",
  "type": "EHR_ACCESS"
},
"time_created": {
  "value": "2015-01-20T19:30:22.765+01:00"
}
}

```

In the result, you should find the EHR ID. This ID will be needed for further operations.

```

"ehr_id": {
  "value": "7d44b88c-4199-4bad-97dc-d78268e01398"
}

```

## 2.4.2 Client Library

In the EHRbase Client Library, creating a new EHR object is straight forward:

```

openEhrClient = DefaultRestClientTestHelper.setupDefaultRestClient();
EhrEndpoint ehrEndpoint = openEhrClient.ehrEndpoint();
UUID ehr = ehrEndpoint.createEhr();

```

## 2.5 Step 4: Load Data

**Warning:** WIP

Step number 4 brings us to the core functionality of openEHR: creating and storing clinical data! For this purpose, we will re-use the Template that we created in step 1. As data instances in openEHR are stored as instances of its Reference Model, it's rather difficult to read for humans. However, this level of abstraction is needed inside the backend to achieve high scalability.

For application developers, more accessible formats are needed. There are two options: The EHRBase Client Library and using Flat Forms.

## 2.5.1 EHRBase Client Library

The EHRbase Client Library allows to use a Template (as OPT) as input and automatically create java classes. These can then be used to create the data. We explain this processes step by step. We assume that you have successfully built the Client Library.

Firstly, you need to create the Java classes from the OPT. This could look like as follows:

```
java -jar client-library-0.2.0.jar -opt "C:\Users\MyUser\Desktop\HiGHmed_Cardio_
↳Monitoring_v1.opt" -out "C:\openEHR SDK\ehrbase_client_
↳library\src\test\java\org\ehrbase\client\classgenerator" -package ""
```

You should find a file named *HiGHmed\_Cardio\_Monitoring\_v1.java* inside your project structure that should look like this:

```
...
@Entity
@Archetype("openEHR-EHR-COMPOSITION.self_monitoring.v0")
@Template("HiGHmed_Cardio_Monitoring.v1")
public class HighmedCardioMonitoringV1 {
    @Path("/context/end_time|value")
    private TemporalAccessor endTimeValue;

    @Path("/language")
    private CodePhrase language;

    @Path("/context/health_care_facility")
    private PartyIdentified healthCareFacility;

    @Path("/composer|external_ref")
    private PartyRef composerExternalref;

    ...
}
```

Next, we can create a new test function like this:

```
public static HighmedCardioMonitoringV1 buildCardioExample() {

    //Create the composition instance and add metadata
    HighmedCardioMonitoringV1 cardioMonitoring = new HighmedCardioMonitoringV1();
    cardioMonitoring.setLanguage(new CodePhrase(new TerminologyId("ISO_639-1"), "de
↳"));
    cardioMonitoring.setTerritory(new CodePhrase(new TerminologyId("ISO_3166-1"), "DE
↳"));
    cardioMonitoring.setSettingDefiningcode(new CodePhrase(new TerminologyId("openehr
↳"), "229"));

    //Create a blood pressure object
    HighmedCardioMonitoringV1.Blutdruck bloodpressure = new HighmedCardioMonitoringV1.
↳Blutdruck();

    //Add data for systolic and diastolic blood pressure
    bloodpressure.setSystolischMagnitude(160d);
    bloodpressure.setSystolischUnits("mm[HG]");

    bloodpressure.setDiastolischMagnitude(120d);
    bloodpressure.setDiastolischUnits("mm[HG]");
}
```

(continues on next page)

(continued from previous page)

```
//Add data for a medical device
HighmedCardioMonitoringV1.Blutdruck.MedizingerT geraet = new
↳HighmedCardioMonitoringV1.Blutdruck.MedizingerT();
geraet.setBeschreibungValue("OMRON Sensor");

DvIdentifier identifier = new DvIdentifier();
identifier.setId("4567879799");
geraet.setEindeutigeIdentifikationsnummerId(identifier);

List<HighmedCardioMonitoringV1.Blutdruck> bpList = new ArrayList<>();
bpList.add(bloodpressure);

cardioMonitoring.setBlutdruck(bpList);

return cardioMonitoring;
}
```

Finally, the composition can be sent to the openEHR server:

```
CompositionEndpoint compositionEndpoint = openEhrClient.compositionEndpoint(ehr);
UUID compositionId = compositionEndpoint.
↳saveCompositionEntity(highmedCardioMonitoringV1);
```

## 2.5.2 Flat Format

Another alternative to using the Client Library is to use a [Simplified Data Template](#) also known as the “Flat format”. In particular, we’ll be looking at the simplified IM Simplified Data template (simSDT) which is based on the web template format created by Marand for the Better platform. The first thing you need is to get the Web Template version of the Template. The ADL Designer tool allows you to export templates as Web Templates. An example of a simple Body Temperature Web Template (borrowed from [EhrScape Examples](#)) would look like this:

```

▼ "webTemplate": {
  "templateId": "Vital Signs",
  "version": "2.1",
  "defaultLanguage": "en",
  "languages": [ ... ], // 2 items
  "tree": {
    "id": "vital_signs",
    "name": "Vital Signs",
    "localizedName": "Vital Signs",
    "rmType": "COMPOSITION",
    "nodeId": "openEHR-EHR-COMPOSITION.encounter.v1",
    "min": 1,
    "max": 1,
    "localizedNames": { ... }, // 2 items
    "aqlPath": "",
    "children": [
      { ... }, // 7 items
      {
        "id": "body_temperature",
        "name": "Body temperature",
        "localizedName": "Body temperature",
        "rmType": "OBSERVATION",
        "nodeId": "openEHR-EHR-OBSERVATION.body_temperature.v1",
        "min": 0,
        "max": -1,
        "localizedNames": { ... }, // 2 items
        "aqlPath": "/content[openEHR-EHR-OBSERVATION.body_temperature.v1]",
        "children": [
          {
            "id": "any_event",
            "name": "Any event",
            "localizedName": "Any event",
            "rmType": "EVENT",
            "nodeId": "at0003",
            "min": 0,
            "max": -1,
            "localizedNames": { ... }, // 2 items
            "aqlPath": "/content[openEHR-EHR-OBSERVATION.body_temperature.v1]/data[at0002]/events[at0003]",
            "children": [
              {
                "id": "temperature",
                "name": "Temperature",
                "localizedName": "Temperature",
                "rmType": "DV_QUANTITY",
                "nodeId": "at0004",
                "min": 1,
                "max": 1,
                "localizedNames": { ... }, // 2 items
                "aqlPath": "/content[openEHR-EHR-OBSERVATION.body_temperature.v1]/data[at0002]/events[at0003]/data[at0004]",
                "inputs": [
                  {
                    "suffix": "magnitude",
                    "type": "DECIMAL"
                  },
                  {
                    "suffix": "unit",
                    "type": "CODED_TEXT"
                  }
                ]
              }
            ]
          }
        ]
      }
    ]
  }
}

```

Next, we create the composition from the Web Template as a simple key-value pair with the keys being a path obtained by concatenating the `id` of each level delimited by a `/`. The last segment is the suffix and uses `|` as a delimiter.

For example, in the above image all the `id` to be concatenated are highlighted in red.

So the paths built from the above example would look like:

```

vital_signs/body_temperature/any_event/temperature|magnitude      vital_signs/
body_temperature/any_event/temperature|unit

```

The value of these above keys would be the actual data. Representing this in JSON would look like:

```

{
  "vital_signs/body_temperature/any_event/temperature|magnitude": 92,
  "vital_signs/body_temperature/any_event/temperature|unit": "°C"
}

```

However, since the cardinality of the `body_temperature` and `any_event` elements are -1 it means that the composition can have an infinite number of `body_temperature` and `body_temperature` recorded in the same composition. To resolve this, we have to index the path like so: `vital_signs/body_temperature:0/any_event:0/temperature|magnitude` `vital_signs/body_temperature:0/any_event:0/temperature|unit`

With these paths, and more context data, a composition with multiple recordings of body temperature will look like:

```
{
  "ctx/time": "2014-03-19T13:10:00.000Z",
  "ctx/language": "en",
  "ctx/territory": "CA",
  "vital_signs/body_temperature:0/any_event:0/time": "2014-03-19T13:10:00.000Z",
  "vital_signs/body_temperature:0/any_event:0/temperature|magnitude": 37.1,
  "vital_signs/body_temperature:0/any_event:0/temperature|unit": "°C",
  "vital_signs/body_temperature:0/any_event:1/time": "2014-03-19T16:33:00.000Z",
  "vital_signs/body_temperature:0/any_event:1/temperature|magnitude": 37.7,
  "vital_signs/body_temperature:0/any_event:1/temperature|unit": "°C"
}
```

The API endpoints for the Flat Format is different from the normal composition API. More details can be found in [this Postman Collection](#). To use the Flat Format, the latest version of EHRBase should be used. More information can be found [here](#).

Congratulations, you stored your first clinical data inside EHRbase! Next, we will take a look how we can retrieve the data using the Archetype Query Language.



This section gives information about the Development of EHRbase and documents the software in detail.

### 3.1 Developing

**Warning:** WIP

For the moment, please see the [EHRbase GitHub repository](#) for developing information, issue tracker and the source code.

### 3.2 Testing

#### 3.2.1 EHRbase Integration Tests with Robot Framework

- *Prerequisites*
- *Test Environment & SUT*
- *Test Execution (under Linux, Mac & Windows)*
  - *With Robot Command*
  - *With Shell Script*
  - *Example Content Of Shell Script (run\_local\_tests.sh)*
- *Local SUT / Manually Controlled SUT*
  - *Usage examples:*
- *Remote SUT / OR how to execute the tests against other systems*

- *Preconditions*
  - *Customize your configuration*
  - *Execute test against EHRSCAPE*
    - \* *Run all tests at one (\*\*not\*\* recommended)*
    - \* *Run single test suites, one by one (recommended)*
  - *Execution Control - Test Suites & Tags*
  - *CI/CD Pipeline (on CircleCI)*
  - *Errors And Warnings*
  - *Auto-Generated Test Report Summary And Detailed Log*
- 

### Prerequisites

- 1) Docker, Java 11 & Maven, Python 3.7+ & Pip are installed
- 2) Robot Framework & dependencies are installed (`pip install -r requirements.txt`)
- 3) Build artefacts created (`mvn package -> application/target/application-x.xx.x.jar`)
- 4) **No DB / no server running!**
- 5) ports 8080 and 5432 not used by any other application! (check it w/ `netstat -tulpn`)

### Test Environment & SUT

The test environment of this project consists of three main parts

1) EHRbase OpenEHR server (application-\*.jar) 2) PostgreSQL database 3) OS with Docker, Java runtime, Python runtime, Robot Framework (generic test automation framework)

Let's refer to the first two parts as the SUT (system under test). The tests are implemented in a way that by default Robot Framework (RF) will take control of the SUT. That means to execute the tests locally all you have to do is to ensure your host machine meets required *prerequisites*. RF will take care of properly starting up, restarting and shutting down SUT as it is required for test execution. There is an option to hand over control of SUT to you, though - described in section *Manually Controlled SUT*.

### Test Execution (under Linux, Mac & Windows)

In general tests are executed by 1) `cd` into tests/ folder and 2) call the **“robot“\*\* command with the folder which contains the test suites as argument. Alternatively you can use prepared shell script: \*\*run\_local\_tests.sh.**

### With Robot Command

The following examples will run all test-cases that are inside robot/ folder

```
# 1) from project's root
cd tests/

# 2) call robot command
robot robot/      # Linux
robot ./robot/    # Mac OS
robot .\robot     # Windows
```

Everything between robot command and the last argument are commandline option to fine control test execution and the processing of test results. Examples:

```
# QUICK COPY/PASTE EXAMPLES TO RUN ONLY A SPECIFIC TEST-SUITE

robot -i composition      -d results --noncritical not-ready -L TRACE robot/
↪COMPOSITION_TESTS/
robot -i contribution     -d results --noncritical not-ready -L TRACE robot/
↪CONTRIBUTION_TESTS/
robot -i directory        -d results --noncritical not-ready -L TRACE robot/DIRECTORY_
↪TESTS/
robot -i ehr_service      -d results --noncritical not-ready -L TRACE robot/EHR_
↪SERVICE_TESTS/
robot -i ehr_status       -d results --noncritical not-ready -L TRACE robot/EHR_STATUS_
↪TESTS/
robot -i knowledge        -d results --noncritical not-ready -L TRACE robot/KNOWLEDGE_
↪TESTS/
robot -i aqlANDEmpty_db   -d results --noncritical not-ready -L TRACE robot/QUERY_
↪SERVICE_TESTS/
robot -i aqlANDloaded_db  -d results --noncritical not-ready -L TRACE robot/QUERY_
↪SERVICE_TESTS/
```

## With Shell Script

Use [shell script](#) to run **all** available tests at once or use it as a reference to see which [command line options](#) are available to the robot command. Examples below demonstrate it's usage:

```
# Linux
. run_local_tests.sh

# Mac OS
./run_local_tests.hs

# Windows
robot -d results --noncritical not-ready -L TRACE robot/

(No script there yet. TODO: create a proper .bat file)
```

## Example Content Of Shell Script (run\_local\_tests.sh)

```
robot --include contribution \
      --exclude TODO -e future -e obsolete -e libtest \
      --loglevel TRACE \
      --noncritical not-ready \
      --flattenkeywords for \
```

(continues on next page)

(continued from previous page)

```
--flattenkeywords foritem \
--flattenkeywords name:_resources.* \
--outputdir results \
--name CONTRIBUTION \
robot/CONTRIBUTION_TESTS/
```

## Local SUT / Manually Controlled SUT

In case you don't want Robot to start up and shut down server and database for you - i.e. during local development iterations - there is a command line option (`-v nodocker`) to turn this off. **This option should be used with some precaution, though!**

Test Suite Setups and Teardowns will NOT be orchestrated by Robot any more. This can lead to issues when trying to run ALL tests at once (i.e. with `robot robot/`) - tests may impact each other and fail. Thus you will have to pass at least a test suite folder as argument or limit test selection by using tags to avoid this (see section below). Moreover

- you have to start the server with cache DISABLED (`--cache.enabled=false`)
- you have to ensure your server configuration applies to Robot's DEV configuration (see `tests/robot/_resources/suite_settings.robot`)
- you have to ensure your DB configuration applies to the one described in main README
- you have to restart server and rollback/reset database properly
- when in doubt about your results, compare them with results in CI pipeline
- YOU HAVE BEEN WARNED!

### Usage Examples:

```
robot --variable nodocker:true robot/TEST_SUITE_FOLDER

# short variant
robot -v nodocker robot/TEST_SUITE_FOLDER
robot -v nodocker -i get_ehr robot/EHR_SERVICE_TESTS
```

Robot will print proper warning in console if it can't connect to server or database:

```
[ WARN ] //////////////////////////////////////
[ WARN ] //                                     ///
[ WARN ] // YOU HAVE CHOSEN TO START YOUR OWN TEST ENVIRONMENT! ///
[ WARN ] // BUT IT IS NOT AVAILABLE OR IS NOT SET UP PROPERLY!  ///
[ WARN ] //                                     ///
[ WARN ] //////////////////////////////////////
[ WARN ]
[ WARN ] [ check "Manually Controlled SUT" in test README ]
[ WARN ] [ https://github.com/ehrbase/ehrbase/blob/develop/tests/README.md#manually-
↪sut ]
[ WARN ]
[ ERROR ] ABORTING EXECUTION DUE TO TEST ENVIRONMENT ISSUES:
[ ERROR ] Could not connect to server!
[ ERROR ] Could not connect to database!
```

## Remote SUT / OR how to execute the tests against other systems

All integration tests in this repository can be executed against other (possibly remotely accessible) OpenEHR conform systems (other than EHRbase). Here we will demonstrate how to run the test against your own remote system. We'll use EHRSCAPE as an example configuration. If you don't have access to EHRSCAPE you'll have to adjust related parts to your needs.

### Preconditions

- 1) the following environment variables have to be available:

```
BASIC_AUTH (basic auth string for EHRSCAPE, i.e.: export BASIC_AUTH="Basic abc...")
EHRSCAPE_USER
EHRSCAPE_PASSWORD
```

- 2) Python 3.7+ installed
- 3) Test dependencies installed

```
cd tests
pip install -r requirements.txt
```

### Customize your configuration

Open `tests/robot/_resources/suite_settings.robot` and adjust the following part to your needs if you don't have access to EHRSCAPE. If you do any changes here, remember to adjust your environment variables in step 1)

```
&{EHRSCAPE}          URL=https://rest.ehrscape.com/rest/openehr/v1
...                  HEARTBEAT=https://rest.ehrscape.com/
...                  CREDENTIALS=@{scapecreds}
...                  AUTH={"Authorization": "%{BASIC_AUTH}"}
...                  NODENAME=piri.ehrscape.com
...                  CONTROL=NONE
@{scapecreds}        %{EHRSCAPE_USER}      %{EHRSCAPE_PASSWORD}
```

### Execute test against EHRSCAPE

The only difference in contrast to normal execution is that you now want to specify that EHRSCAPE configuration from `suite_settings.robot` should be used. This is done by setting SUT variable to EHRSCAPE which you can achieve by passing `-v SUT:EHRSCAPE` when calling `robot`. Check examples below.

### Run all tests at one (not recommended)

This is not recommend because it may take from 30 to 60 minutes and makes it harder to analyse the results.

```
robot -v SUT:EHRSCAPE -e future -e circleci -e TODO -e obsolete -e libtest -d results_
↪ -L TRACE --noncritical not-ready robot/
```

### Run single test suites, one by one (recommended)

Execute the test suite that you are interested in by copy&pasting one of the lines below, then analyse the results of that test suite.

Best practice is also to reset your system under test to a clear state before executing the next test suite.

```

robot -v SUT:EHRSCAPE -d results/composition -e future -e circleci -e TODO -
↳e obsolete -e libtest -L TRACE --noncritical not-ready --name COMPO robot/
↳COMPOSITION_TESTS
robot -v SUT:EHRSCAPE -d results/contribution -e future -e circleci -e TODO -
↳e obsolete -e libtest -L TRACE --noncritical not-ready --name CONTRI robot/
↳CONTRIBUTION_TESTS
robot -v SUT:EHRSCAPE -d results/directory -e future -e circleci -e TODO -e
↳obsolete -e libtest -L TRACE --noncritical not-ready --name FOLDER robot/
↳DIRECTORY_TESTS
robot -v SUT:EHRSCAPE -d results/ehr_service -e future -e circleci -e TODO -
↳e obsolete -e libtest -L TRACE --noncritical not-ready --name EHRSERVICE
↳robot/EHR_SERVICE_TESTS
robot -v SUT:EHRSCAPE -d results/ehr_status -e future -e circleci -e TODO -e
↳obsolete -e libtest -L TRACE --noncritical not-ready --name EHRSTATUS
↳robot/EHR_STATUS_TESTS
robot -v SUT:EHRSCAPE -d results/knowledge -e future -e circleci -e TODO -e
↳obsolete -e libtest -L TRACE --noncritical not-ready --name KNOWLEDGE
↳robot/KNOWLEDGE_TESTS
robot -v SUT:EHRSCAPE -d results/aql_1 -e future -e circleci -e TODO -e
↳obsolete -e libtest -L TRACE --noncritical not-ready --name "QUERY empty_db
↳" -i empty_db robot/QUERY_SERVICE_TESTS
robot -v SUT:EHRSCAPE -d results/aql_2 -e future -e circleci -e TODO -e
↳obsolete -e libtest -L TRACE --noncritical not-ready --name "QUERY SMOKE"
↳-i SMOKE robot/QUERY_SERVICE_TESTS
robot -v SUT:EHRSCAPE -d results/aql_3 -e future -e circleci -e TODO -e
↳obsolete -e libtest -L TRACE --noncritical not-ready --name "QUERY loaded_
↳db" -i loaded_db robot/QUERY_SERVICE_TESTS

```

## Execution Control - Test Suites & Tags

Execution of **all** integration tests takes **about 30 minutes** (on a fast dev machine). To avoid waiting for all results you can specify exactly which test-suite or even which subset of it you want to execute. There are seven test-suites to choose from by passing proper TAG to robot command via the `--include` (or short `-i`) option:

TEST SUITE	SUPER TAG	SUB TAG(s)	EXAMPLE(s)
COMPOSITION_TESTS	composition	json, json1, json2, xml, xml1, xml2	robot --include composition robot -i composition robot -i compositionANDjson
CONTRIBUTION_TESTS	contribution	contribution_commit, contributions_list, contribution_has, contribution_get	robot -i contribution
DIRECTORY_TESTS	directory	directory_create, directory_update, directory_get, directory_delete, directory_get@time, ...	robot -i directory robot -i directory_createORDirectory_update
EHR_SERVICE_TESTS	ehr_service	create_ehr, update_ehr, has_ehr, get_ehr, ehr_status	robot -i ehr_service
EHR_STATUS_TESTS	ehr_status	ehr_status_get, ehr_status_set, ehr_status_set_queryable, ehr_status_clear_queryable, ehr_status_set_modifiable, ehr_status_clear_modifiable	robot -i ehr_status
KNOWLEDGE_TESTS	knowledge	opt14	robot -i knowledge
QUERY_SERVICE_TESTS	query	adhoc-query, aql_stored-query, aql_register-query, aql_list-query	robot -i adhoc-query

The **SUPER TAG** is meant to reference *all* tests from related test-suite. The **SUB TAGs** can be used (in combination with a SUPER TAG) to further narrow down which tests to include into execution. As you can see from the examples in the table above it is possible to combine TAGs with AND and OR operators. Tags themselves are case insensitive but **the operators have to be upper case**. In addition to `--include` or `-i` option there is also an `--exclude` / `-e` option. It is possible to combine `-i` and `-e` in one call. Example below would include all test from `EHR_SERVICE_TESTS` folder which have the `ehr_service` **and** `get_ehr` tags and would ignore all test which have the `future` tag.

```
robot -i ehr_serviceANDget_ehr -e future robot/EHR_SERVICE_TESTS/
```

Using TAGs to include/exclude tests from execution is very well documented in [Robot Framework's User Guide](#).

## CI/CD Pipeline (on CircleCI)

Check out `.circleci/config.yml` in project root for an CircleCI example pipeline which runs Robot test suites in parallel.

## Errors And Warnings

You will see `[WARN]` and `[ERROR]` in console output and in `log.html`

`[ERROR]` -> take a closer look, probably important

`[WARN]` -> minor issues like wrong status code or keyword deprecation warning.

NOTE: `[WARN]` Response body content is not JSON. Content-Type is: text/html

You will see this warning very often. IGNORE it! It's caused by a RF library.

## Auto-Generated Test Report Summary And Detailed Log

After each test run Robot creates a report.html (summary) and a log.html (details) in results folder. The files are overwritten after each run by default. If you want to keep history of your test runs you can [time-stamp](#) the output files.

### 3.2.2 RESTORE KEYCLOAK FROM PREVIOUSLY EXPORTED CONFIGURATION

(run all commands listed below from I\_OAuth2\_Keycloak folder)

#### 1) START KC W/ A MOUNTED VOLUME

```
docker run -d --name keycloak \
  -p 8081:8080 \
  -v $(pwd)/exported-keycloak-config:/restore-keycloak-config \
  -e KEYCLOAK_USER=admin \
  -e KEYCLOAK_PASSWORD=admin \
  jboss/keycloak:10.0.2
```

#### 2) RESTORE CONFIG FROM DIRECTORY

```
docker exec -it keycloak /opt/jboss/keycloak/bin/standalone.sh \
  -Djboss.socket.binding.port-offset=100 \
  -Dkeycloak.migration.action=import \
  -Dkeycloak.migration.provider=dir \
  -Dkeycloak.profile.feature.upload_scripts=enabled \
  -Dkeycloak.migration.dir=/restore-keycloak-config \
  -Dkeycloak.migration.strategy=OVERWRITE_EXISTING
```

When the import is complete use Ctrl-C to exit the session.

NOTE: This is a minimal setup using Keycloak's embedded H2 DB which is just enough for testing. It's probably a good idea not to use this in production :)

If you ever have to reconfigure Keycloak Docker setup manually and to recreate the export follow the steps below:

### 3.2.3 EXPORT COMPLETE KEYCLOAK CONFIGURATION

#### 1) START KC W/ A MOUNTED VOLUME

```
docker run -d --name keycloak \
  -p 8081:8080 \
  -v $(pwd)/exported-keycloak-config:/restore-keycloak-config \
  -e KEYCLOAK_USER=admin \
  -e KEYCLOAK_PASSWORD=admin \
  jboss/keycloak:10.0.2
```

#### 2) LOGIN AS ADMIN AND CONFIGURE KC TO YOUR NEEDS

- a) create realm: ehrbase
- b) create client: ehrbase-robot - IMPORTANT: make sure in client settings
  - 'Access Type' is set to **public**
  - 'Direct Access Grants Enabled' is set to **ON**
- c) create user: robot w/ password robot

#### 3) EXPORT CONFIGURATION INTO MULTIPLE FILES WITHIN A DIRECTORY



```
docker exec -it keycloak /opt/jboss/keycloak/bin/standalone.sh \
-Djboss.socket.binding.port-offset=100 \
-Dkeycloak.migration.action=export \
-Dkeycloak.migration.provider=dir \
-Dkeycloak.migration.dir=/opt/jboss/keycloak/export-dir \
-Dkeycloak.migration.usersPerFile=1000 \
-Dkeycloak.migration.strategy=OVERWRITE_EXISTING
```

When the export is complete use Ctrl-C to exit the session. The export is complete when you see something like

```
Keycloak 10.0.2 (WildFly Core 11.1.1.Final) started in 11390ms -
Started 591 of 889 services (606 services are lazy, passive or on-demand)
```

#### 4) COPY EXPORTED CONFIGURATION FROM CONTAINER TO YOUR HOST

```
docker cp keycloak:/opt/jboss/keycloak/export-dir ./exported-keycloak-config
```

optional before copying check the folder exists and contains exported config files:

```
docker exec -it keycloak bash
ls /opt/jboss/keycloak/export-dir
```

Alternatively (and in case above steps stop to work for what ever reason) it is possible to export complete KC configuration into a single JSON file:

#### 1) START KEYCLOAK W/ MOUNTED VOLUME

```
docker run -d --name keycloak \
-p 8081:8080 \
-v $(pwd):/workspace \
-e KEYCLOAK_USER=admin \
-e KEYCLOAK_PASSWORD=admin \
jboss/keycloak:10.0.2
```

#### 2) EXPORT (SINGLE FILE)

Then **export** your database into a single JSON file:

```
docker exec -it keycloak /opt/jboss/keycloak/bin/standalone.sh \
-Djboss.socket.binding.port-offset=100 \
-Dkeycloak.migration.action=export \
-Dkeycloak.migration.provider=singleFile \
-Dkeycloak.migration.file=/workspace/exported-kc-config-single-file/keycloak-
→export.json
-Dkeycloak.migration.strategy=OVERWRITE_EXISTING
```

#### 3) IMPORT FROM THE COMMAND LINE

Start with a blank canvas ...

```
docker container stop keycloak
docker container rm keycloak

docker run -d --name keycloak \
-p 8081:8080 \
```

(continues on next page)

(continued from previous page)

```
-v $(pwd) :/workspace \
-e KEYCLOAK_USER=admin \
-e KEYCLOAK_PASSWORD=admin \
jboss/keycloak:10.0.2
```

To import from a (previously exported) file into your database ...

```
docker exec -it keycloak /opt/jboss/keycloak/bin/standalone.sh \
-Djboss.socket.binding.port-offset=100 \
-Dkeycloak.migration.action=import \
-Dkeycloak.migration.provider=singleFile \
-Dkeycloak.migration.file=/workspace/exported-kc-config-single-file/keycloak-
→export.json
```

When the import is complete use Ctrl-C to exit the session.

---

```
** WARNING **
DO NOT TRY TO RESTORE KEYCLOAK W/ ` -e KEYCLOAK_IMPORT=/path-to/exported-config.json `
APPROACH AS DOCUMENTED ON KEYCLOAK'S DOCKER IMAGE DISCRIBTION ON DOCKER HUB.
////////////////////////////////////
////                               THAT DOES NOT WORK!                               ////
////////////////////////////////////
DON'T WASTE YOUR TIME! I'VE BEEN THERE, I'VE DONE THAT!
```

## 3.2.4 CI/CD

This part of the documentation describes how continuous integration helps us to have a fast feedback loop during development. This allows the project to keep up testing with the fast pace of iterations within the agile development environment (Scrum sprints).

### Continuous Integration

EHRbase uses [CircleCI](#) for continuous integration and deployment. The CI pipeline consists of the following workflows:

#### Pipeline workflow 1/3 - build-and-test

- trigger: commit to any branch (except - release/v\*, master, sync/, feature/sync/)
- jobs:
  - build artifacts
  - run unit tests
  - run sdk integraiton tests
  - run robot integration tests
  - perform sonarcloud analysis and OWASP dependency check

### Pipeline workflow 2/3 - release

- trigger: commit to release/v or master branch
- jobs:
  - build artifacts
  - run unit tests
  - run sdk integraiton tests
  - run robot integration tests
  - perform sonarcloud analysis and OWASP dependency check
  - TODO: deploy to Maven Central
  - TODO: deploy to Docker Hub

### Pipeline workflow 3/3 - synced-feature-check

This is a special workflow to catch errors that can occur when code changes introduced to EHRbase AND openEHR\_SDK repository are related in a way that they have to be tested together and otherwise can't be caught in workflow 1 or 2.

- trigger: commit to sync/\* branch
- jobs:
  - pull, build, and test SDK from sync/\* branch of openEHR\_SDK repo
  - build and test ehrbase (with SDK installed in previous step)
  - start ehrbase server (from .jar packaged in previous step)
  - run SDK's (java) integration tests
  - run EHRbase's (robot) integration tests

HOW TO USE WORKFLOW 3/3  
=====

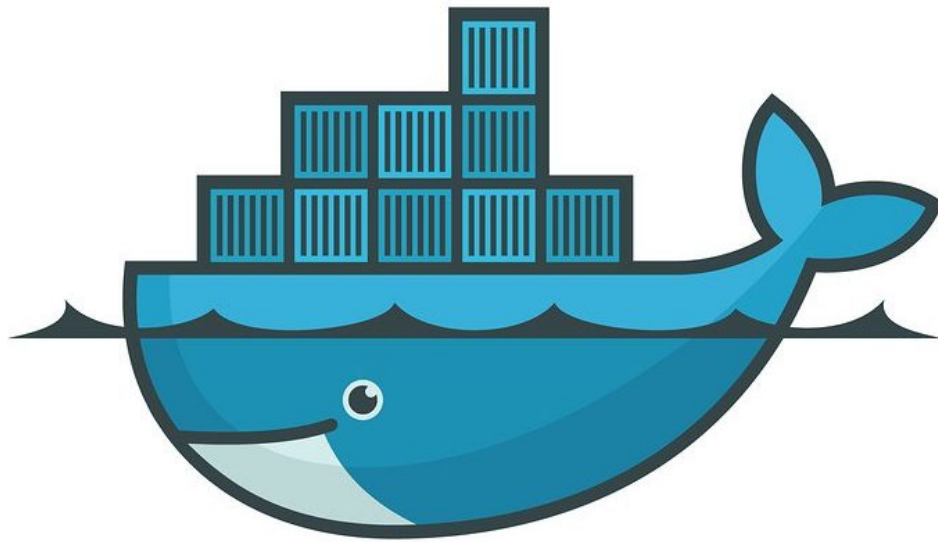
1. create TWO branches following the naming convention `sync/[issue-id]\_some-  
↪description`  
  - in** both repositories (EHRbase and openEHR\_SDK) with exact the same name:
  - ehrbase repo       --> i.e.     sync/123\_example-issue
  - openehr\_sdk repo   --> i.e.     sync/123\_example-issue
2. apply your code changes
3. push to openehr\_sdk repo (NO CI will be triggered)
4. push to ehrbase repo (CI will trigger this workflow)
5. create TWO PRs (one **in** EHRbase, one **in** openEHR\_SDK)
6. merge BOTH PRs considering below notes:
  - make sure both PRs are reviewed and ready to be merged at the same time!
  - make sure to sync both PRs with develop before merging!
  - MERGE BOTH PRs AT THE SAME TIME!

## 3.3 Deploying

**Warning:** WIP

For the moment, please see the [EHRbase GitHub repository](#) for developing information, issue tracker and the source code.

## 3.4 Docker Images



This part of the documentation explains how to use EHRbase from a Docker container and/or how to create your own EHRbase and PostgreSQL DB Docker image.

### 3.4.1 EHRbase Docker Image

This part of the documentation explains how to use EHRbase from a Docker container and/or how to create your own EHRbase Docker image.

#### Build EHRbase Image

This part of the documentation explains how to build a Docker Image of EHRbase Server Application locally from Dockerfile in the Git repository.

## Build Image From Dockerfile

EHRbase's Github repository contains a [Dockerfile](#) which you can use to build your custom Docker image from. Follow steps below to build your own Docker Image (with default EHRbase settings):

```
git clone https://github.com/ehrbase/ehrbase.git
cd ehrbase
docker build -t give-it-a-name .      # don't forget the `.` at the end of the command!!!
docker image ls                      # you should be able to see the image you just_
↳ created
```

## Why To Build Own Image?

EHRbase's Dockerfile defines some environment variables with default values which will be active when you run a container from created Docker image. For example when you run the following command

```
docker run ehrbase/ehrbase
```

the running Docker container will have environment variables with default values as shown in code snippet from related part of Dockerfile below:

```
...

ARG DB_URL=jdbc:postgresql://ehrdbs:5432/ehrbase
ARG DB_USER="ehrbase"
ARG DB_PASS="ehrbase"
ARG SERVER_NODENAME=docker.ehrbase.org

ENV EHRBASE_VERSION=${EHRBASE_VERSION}
ENV DB_USER=$DB_USER
ENV DB_PASS=$DB_PASS
ENV DB_URL=$DB_URL
ENV SERVER_NODENAME=$SERVER_NODENAME
ENV SECURITY_AUTHTYPE="NONE"
...
```

The values of all ARG(s) can be overwritten during image build time to adjust default (run time) behaviour of your custom Docker image. Use `--build-arg ARG_name=value` to override default values when building your image. See example below:

```
docker build --build-arg DB_URL=your-db-url \
--build-arg DB_USER=your-db-user \
--build-arg DB_PASS=your-db-pass \
--build-arg SERVER_NODENAME=your-system-name \
-t give-your-image-a-name:and-tag .
```

In addition to overriding default ENV values during build time it is also possible to override ENV values and even add new ENVs to a container's run time. Check next example (which assumes you pulled or created an image named `ehrbase/ehrbase`):

```
docker run -e DB_URL=jdbc:postgresql://ehrdbs:5432/ehrbase \
-e DB_USER=foouser \
-e DB_PASS=foopass \
-e SERVER_NODENAME=what.ever.org \
ehrbase/ehrbase
```

### Use EHRbase Image

This part of the documentation explains how to run EHRbase as a Docker Container created from the image in previous steps.

### Run EHRbase in Docker

---

**Note:** Remember: EHRbase requires a properly configured and running PostgreSQL DB to work. Make sure to set this up first before you try run EHRbase.

---

To run EHRbase in a Docker Container first pull the official Docker image from Docker Hub:

```
docker pull ehrbase/ehrbase
```

### OR

build your own image form Dockerfile:

```
git clone https://github.com/ehrbase/ehrbase.git
cd ehrbase
docker build -t myehrbase/ehrbase .
docker image ls
```

**THEN** use the *docker run* command adjusting parameters to your needs to change Container's default behaviour.

---

**Note:** Remember: Container's default behaviour is set during Docker image build time.

---

```
docker run -e DB_URL=jdbc:postgresql://ehbdb:5432/ehrbase \
-e DB_USER=foouser \
-e DB_PASS=foopass \
-e SERVER_NODENAME=what.ever.org \
-p 8080:8080 \
ehrbase/ehrbase
```

Parameter	Usage	Example
DB_URL	Database URL. Must point to the running database server.	<a href="jdbc:postgresql://ehrdp:5432/ehrbase">jdbc:postgresql://ehrdp:5432/ehrbase</a>
DB_USER	Database user configured for the ehr schema.	ehrbase
DB_PASS	DB user password	ehrbase
SERVER_NODENAME	Name of the server	local.ehrbase.org
SECURITY_AUTHTYPE	HTTP security method	BASIC / OAUTH
SECURITY_AUTHUSER	BASIC Auth username	myuser
SECURITY_AUTHPASSWORD	BASIC Auth password	myPassword432
SECURITY_AUTHADMINUSER	BASIC auth admin user	myadmin
SECURITY_AUTHADMINPASSWORD	BASIC auth admin password	mySuperAwesomePassword123
ADMINAPI_ACTIVE	Should admin endpoints be enabled	true / false
ADMINAPI_ALLOWDELETEALL	Allow admin to delete all resources - i.e. all EHRs	true / false
MANAGEMENT_ENDPOINT_ENV_ENABLED	Enable /management/env endpoint from actuator	true / false
MANAGEMENT_ENDPOINT_HEALTH_ENABLED	Enable /management/health endpoint from actuator	true / false
MANAGEMENT_ENDPOINT_INFO_ENABLED	Enable /management/info endpoint from actuator	true / false
MANAGEMENT_ENDPOINT_METRICS_ENABLED	Enable /management/metrics endpoint from actuator	true / false
MANAGEMENT_ENDPOINT_PROMETHEUS_ENABLED	Enable /management/prometheus endpoint from actuator	true / false
SERVER_DISABLESTRICTVALIDATION	Disable strict validation of openEHR input	true / false

**Note:** Do NOT set `SPRING_SECURITY_OAUTH2_RESOURCESERVER_JWT_ISSUERURI` in combination with `SECURITY_AUTHTYPE=BASIC`! This will crash EHRbase at start up.

Parameter	Usage
SPRING_SECURITY_OAUTH2_RESOURCESERVER_JWT_ISSUERURI	OAuth2 server issuer uri
example:	<a href="https://keycloak.example.com/auth/realms/ehrbase">https://keycloak.example.com/auth/realms/ehrbase</a>

## Run EHRbase + DB with Docker-Compose

**Note:** Prerequisite: docker-compose is installed on your machine

With [Docker-Compose](#) you can start EHRbase and the required DB from a configuration file written in YAML format. There is an example [docker-compose.yml](#) configuration file in our Git repository. Using it allows you to set up and start EHRbase along with the required database with a few simple steps:

```
# download the docker-compose.yml file to your local
wget https://github.com/ehrbase/ehrbase/raw/develop/docker-compose.yml
wget https://github.com/ehrbase/ehrbase/raw/develop/.env.ehrbase
docker-compose up

# OR: start both containers detached, without blocking the terminal
docker-compose up -d
```

---

**Note:** It is not necessary to have the whole Git repository on your machine, just copy the `docker-compose.yml` file to a local working directory and run `docker-compose up`.

---

---

**Note:** DB data is saved in `./pgdata` for easier access.

---

You can configure all environment variables via the file `.env.ehrbase` which is located at the same folder as the `docker-compose.yml` file. This is also required for setting boolean values due to Docker compose files do not allow setting boolean values directly inside `docker-compose.yml`.

### Docker environment examples

Here you can find some example settings for common use cases for the usage of EHRbase Docker containers. You can also use the environment variables with the normal `.jar` execution by setting the variables according to your operating system.

#### Use BASIC auth

Run the docker image with this setting:

```
docker run --network ehrbase-net --name ehrbase -e SECURITY_AUTHTYPE=BASIC \
-e SECURITY_AUTHUSER=myuser -e SECURITY_AUTHPASSWORD=ThePasswordForUser \
-e SECURITY_AUTHADMINUSER=myadmin -e SECURITY_AUTHADMINPASSWORD=SecretAdminPassword \
-d -p 8080:8080 ehrbase/ehrbase:latest
```

This will set the used authentication method to BASIC auth and all requests against the EHRbase must be provided with the Authorization header set to `Basic %username%:%password%` whereas the username and password must be encoded with base64.

---

**Note:** Ensure you use an encrypted connection over https otherwise the username and password can be described easily

---

#### Use OAuth2

Run the docker image with this setting.

```
docker run --network ehrbase-net --name ehrbase -e SECURITY_AUTHTYPE=OAUTH \
-e SPRING_SECURITY_OAUTH2_RESOURCESERVER_JWT_ISSUERURI=https://keycloak.example.com/
↪auth/realms/ehrbase \
-d -p 8080:8080 ehrbase/ehrbase:latest
```



You have to prepare the authentication server including a valid client at the target server to get this setup run.

## Use OAuth2 and Attribute-based Access Control

Run the docker image with this setting.

```
docker run --network ehrbase-net --name ehrbase
-e SECURITY_AUTHTYPE=OAUTH \
-e SPRING_SECURITY_OAUTH2_RESOURCESERVER_JWT_ISSUERURI=https://keycloak.example.com/
  →auth/realms/ehrbase \
-e ABAC_ENABLED=true
-e ABAC_SERVER=http://localhost:3001/rest/v1/policy/execute/name/
-d -p 8080:8080 ehrbase/ehrbase:latest
```

Additionally, add the configuration of the endpoints and policies either here with additional `-e` parameters or more user-friendly in a separate `docker-compose.yml` file.

## Publish EHRbase Image

This project uses [Docker Hub / Cloud](#) infrastructure to automatically build and publish Docker images on the public Docker Hub Registry whenever there is an update to the code - check recent [EHRbase Docker image tags](#).

## Docker Hub Autobuilds

EHRbase Docker image is created and published on [Docker Hub Registry](#) on every push/merge to *master*, *develop* and *release-\** branch. Created Docker images are tagged as shown in table below:

Branch	Docker Tag	Example
master	latest	<i>ehrbase/ehrbase:latest</i>
develop	next	<i>ehrbase/ehrbase:next</i>
release-*	semversion	<i>ehrbase/ehrbase:0.13.0</i>

## Docker Hub Configuration

**Note:** This part serves only as a reference and does not have to be repeated - it describes what was needed to do to configure automated Docker image builds on Docker Hub.

1. Create a Dockerfile in root of Github repository
2. Login to Docker Hub (docker.com) with the tech-user
3. Login to Github with the tech-user (he has owner access to the Organisation)
4. Connect tech-user to Docker Hub granting access to EHRbase Organisation to enable Autobuilds

ehrlbaseorg

Community Organization EHRbase https://secure.gravatar.com/avatar/0545d964c568d93c9189b2cc6fb2f4a3 Joined September 4, 2019

Members Teams Repositories **Settings** Billing

General **Linked Accounts** Default Privacy Notifications Deactivate Org

### Linked Accounts

These account links are used for Automated Builds, so that Docker Hub can access your project lists and help you configure your Automated Builds. **Please note: A Github/Bitbucket account can only be connected to one Docker Hub account at a time.**

**Service user (or machine/bot account) suggested**  
Attaching your personal Github or Bitbucket account to this Docker Hub organization will allow other organization owners to create builds from your private repositories. We suggest using a [service user](#) (also referred to as a machine user or bot account).

Provider	Account			
GitHub	No account linked	🔄	🚫	<a href="#">Connect</a>
Bitbucket	No account linked	🔄	🚫	<a href="#">Connect</a>

ehrlbase

User Joined September 4, 2019

General **Linked Accounts** Security Default Privacy Notifications Convert Account Deactivate Account

**Linked Accounts**

These account links are used for Automated Builds, so that Docker Hub can access your project lists and help you configure your Automated Builds. **Please note: A Github/Bitbucket account can only be connected to one Docker Hub account at a time.**

**Service user (or machine/bot account) suggested**  
Attaching your personal Github or Bitbucket account to this Docker Hub organization will allow other organization owners to create builds from your private repositories. We suggest using a [service user](#) (also referred to as a machine user or bot account).

Provider	Account			
GitHub	ehrlbase	🔄	🚫	
Bitbucket	No account linked	🔄	🚫	<a href="#">Connect</a>

## 5. Go to Builds / Configure Automated Builds

Build Activity

Overview of your build activity of the last 9 builds

18 min

Queue Success Failed Canceled

## 6. Set Up Build Rules

Build configurations

SOURCE REPOSITORY: ehrlbase

NOTE: Changing source repository may affect existing build rules.

BUILD LOCATION: Build on Docker Hub's Infrastructure

AUTOTEST: ☒ Off

REPOSITORY LINKS: ☒ Off

BUILD RULES +

The build rules below specify how to build your source into Docker images.

Source Type	Source	Docker Tag	Dockerfile location	Build Context	Autobuild	Build Caching
Branch	master	latest	Dockerfile	/	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Branch	develop	next	Dockerfile	/	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Branch	/^release/v([0-9.]+)\$/	{\1}	Dockerfile	/	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

View example build rules

Scenario	Source Type	Source	Docker Tag	Matches	Docker Tag Built
Exact match	Branch	master	latest	master	latest
Match versions	Tag	/^[0-9.]+\$/	release-{sourceref}	1.2.0	release-1.2.0
Trailing modifiers	Tag	/^[0-9.]+/	release-{sourceref}	1.2.0-rc	release-1.2.0-rc
Extract version number	Tag	/^v([0-9.]+)\$/	version-{v1}	v1.2.3	version-1.2.3

### 3.4.2 EHRbase DB Docker Image

This part of the documentation explains how to create and use PostgreSQL DB Docker image required by the EHRbase Server Application.

### Build DB Image

This part of the documentation explains how to locally build a Docker Image of PostgreSQL DB required by EHRbase Server Application.

### Build Image From Dockerfile

```
git clone https://github.com/ehrbase/docker.git
cd docker/dockerfiles
docker build -t ehrbase_db -f ehrbase-postgresql-full.dockerfile .
docker image ls
```

### Use DB Image

This part of the documentation explains how to run EHRbase DB in a Docker Container and how to change environment variables inside the Container if needed.

### Run DB with default parameters

```
docker pull ehrbase/ehrbase-postgres:11.10
docker run --name ehrdb -d -p 5432:5432 ehrbase/ehrbase-postgres:11.10
```

### Customization

If you want to set specific parameters use environment variables provided with the `-e` option to the docker run command. This will be used to set the specific parameters for root postgres user password and ehrbase user and password. If not provided the default values will be used.

The following parameters can be set via `-e` option:

Parameter	Usage	Default
POSTGRES_PASSWORD	Password for postgres	postgres
EHRBASE_USER	ehrbase db username	ehrbase
EHRBASE_PASSWORD	ehrbase db password	ehrbase

## 3.5 Technical Documentation

This part of the documentation lists and explains technical details of the implementation of EHRbase.

### 3.5.1 Overview

**Warning:** WIP

- openEHR

- REST
- AQL
- etc.

### 3.5.2 Service Layer

**Warning:** WIP

#### General

The service layer of EHRbase is composed of ...

#### openEHR Platform Abstract Service Model

Based on the [openEHR Platform Abstract Service Model](#) the following check list is build to give an overview and document the current state. Each service component has a table documenting the current state of

- implementation of the *method* itself, if applicable
- implementation and utilization of the *pre checks* of the method, if applicable
- implementation and utilization of the *post checks* of the method, if applicable

#### Services

- *EHR*
- *EHR\_STATUS*
- *DIRECTORY*
- *COMPOSITION*
- *CONTRIBUTION*

#### EHR

For more details see [I\\_EHR\\_SERVICE](#) in the official documentation.

Method	Implemented	Pre	Post
has_ehr	Yes	/	/
has_ehr_for_subject	I	/	/
create_ehr	C	/	No
create_ehr_with_id	C	No	No
create_ehr_for_subject	No	/	/
create_ehr_for_subject_with_id	No	No	/
get_ehr	No	No	/
get_ehrs_for_subject	No	/	/
i_ehr	No	/	/

Methods with **I** note are currently indirectly implemented. Their functionality is available, but the general signature might be different.

Methods with **C** note are currently combined in a more general *createEhr* method.

### EHR\_STATUS

For more details see [I\\_EHR\\_STATUS](#) the in official documentation.

Method	Implemented	Pre	Post
has_ehr_status_version	I	Yes	/
get_ehr_status	Yes	Yes	/
get_ehr_status_at_time	I	Yes	/
set_ehr_queryable	C	No	No
set_ehr_modifiable	C	No	No
clear_ehr_queryable	C	No	No
clear_ehr_modifiable	C	No	No
update_other_details	C	/	/
get_ehr_status_at_version	Yes	Yes	/
get_versioned_ehr_status	No	No	No

Methods with **I** note are currently indirectly implemented. Their functionality is available, but the general signature might be different.

Methods with **C** note are currently combined in a more general *updateStatus* method.

### DIRECTORY

For more details see [I\\_EHR\\_DIRECTORY](#) the in official documentation.

Method	Implemented	Pre	Post
has_directory			
has_path			
create_directory			
get_directory			
get_directory_at_time			
update_directory			
delete_directory			
has_directory_version			
get_directory_at_version			
get_versioned_directory			

### COMPOSITION

For more details see [I\\_EHR\\_COMPOSITION](#) the in official documentation.

Method	Implemented	Pre	Post
has_composition			
get_composition_latest			
get_composition_at_time			
get_composition_at_version			
get_versioned_composition			
create_composition			
update_composition			
delete_composition			

## CONTRIBUTION

For more details see [I\\_EHR\\_CONTRIBUTION](#) the in official documentation.

Method	Implemented	Pre	Post
has_contribution			
get_contribution			
commit_contribution			
list_contributions			
contribution_count			

### 3.5.3 New Contain Clause Resolution Strategy

C. Chevalley 3.7.20

### 3.5.4 Background

AQL specifies the important clause ‘CONTAINS’. This allows to specify a containment criteria on specified archetypes anywhere into composition projections. The specification is found in [openEHR AQL containment](#). As mentioned in the specification, ‘CONTAINS’ specifies an *hierarchical* relationship with the Tree based data architecture (hence not to be confused with a WHERE clause criteria). Hierarchical constraint is modeled using connected and *acyclic graph*; a node can be accessed from the root through a unique path.

#### Previous Approach

The previous strategy was based on maintaining a specific containment table based on a hierarchical data representation using PostgreSQL [ltree](#). The algorithm was based on identified AQL paths during the composition serialization: each path expression was then stored in a simplify way as to describe the hierarchy of archetypes within the composition, this for each composition. The table was then used to build the SQL expression corresponding to an AQL statement:

- identify the template(s) matching the contain clause
- retrieve the path for a given contain constraint for each identified template(s)

The resulting SQL expression is a combination (UNION) of SQL statement for each template.

An example of containment records is as follows:

```
CONTAINS COMPOSITION c CONTAINS OBSERVATION o [openEHR-EHR-OBSERVATION.pulse-oximetry.v1]
```

Is translated as

```
SELECT composition_id FROM ehr.contain WHERE label ~= '*.openEHR_EHR_OBSERVATION_  
↪pulse_oximetry_v1'
```

The template Id is then retrieve from the correlation between the composition entry (ehr.entry) and the template\_id attribute. The same logic is used to retrieve the path of a particular node relatively to a template.

Although this approach was initially satisfactory, it has been seen as impacting performance whenever the number of records increases. As shown in the above example, the number of entries for a single composition can be significant and, in the lack of proper indexing, the identification of a template may require costly sequential search. Further, the construction of an SQL expression corresponding to an AQL CONTAINS clause was problematic. Another issue was that item\_structure in /context/other\_context was not referenced in containment and then was not resolved for querying.

## New Approach

### Assumptions

This approach assumes that all stored compositions are bound to one known template (at the time of this writing, operational template v1.4). A template is known whenever it is defined in the platform, it is stored in the DB in table ehr.template\_store

### Objectives

The new logic consists in resolving an AQL CONTAINS clause by:

- identifying the template(s) matching the constraints
- resolving the paths for the nodes defined in the CONTAINS clause

Identified templates are used to build the resulting SQL expression, each identified template produces a SQL query. At the end of the process, SQL queries are chained by a UNION clause.

Resolved paths are used to construct the json path expression used to query JSONB structure in the DB.

## Technical Approach

### Operational Template Traversal

All resolution are now based on so-called [WebTemplates](#) (class `OptVisitor`) providing a tree construct detailing all constraints and attributes of an operational template. The tree structure is traversed using JsonPath expressions (see f.e. [Baeldung's guide](#) on this).

For instance, to check the existence of a node containment and return the corresponding AQL path, the following logic is illustrated as follows.

Assume we want to retrieve the template(s) where the following expression is satisfied:

```
contains COMPOSITION c[openEHR-EHR-COMPOSITION.report-result.v1] contains  
CLUSTER f [openEHR-EHR-CLUSTER.case_identification.v0]
```

The corresponding jsonpath expression to traverse the WebTemplate is:

```
$...[?(@.node_id == 'openEHR-EHR-COMPOSITION.report-result.v1')]..[?(@.node_id  
== 'openEHR-EHR-CLUSTER.case_identification.v0')]
```



When applied to template Virologischer Befund, the following structure is returned (these are the attributes for the retrieved node)

```
{
  "min" : "1",
  "aql_path" : "/context/other_context[at0001]/items[openEHR-EHR-CLUSTER.case_
↪identification.v0]",
  "max" : "1",
  "children" : " size = 2",
  "name" : "Fallidentifikation",
  "description" : "Zur Erfassung von Details zur Identifikation eines Falls im_
↪Gesundheitswesen.",
  "id" : "fallidentifikation",
  "type" : "CLUSTER",
  "category" : "DATA_STRUCTURE",
  "node_id" : "openEHR-EHR-CLUSTER.case_identification.v0",
}
```

The corresponding AQL path for node openEHR-EHR-CLUSTER.case\_identification.v0 in template Virologischer Befund is /context/other\_context[at0001]/items[openEHR-EHR-CLUSTER.case\_identification.v0]

The corresponding WebTemplate section for this particular node is represented as follows:

```
{
  "min": 1,
  "aql_path": "/context/other_context[at0001]/items[openEHR-EHR-CLUSTER.case_
↪identification.v0]",
  "max": 1,
  "children": [
    {
      "min": 1,
      "aql_path": "/context/other_context[at0001]/items[openEHR-EHR-CLUSTER.case_
↪identification.v0]/items[at0001]",
      "max": 1,
      "name": "Fall-Kennung",
      "description": "Der Bezeichner/die Kennung dieses Falls.",
      "id": "fall_kennung",
      "category": "ELEMENT",
      "type": "DV_TEXT",
      "constraints": [
        {
          "aql_path": "/context/other_context[at0001]/items[openEHR-EHR-CLUSTER.case_
↪identification.v0]/items[at0001]/value",
          "mandatory_attributes": [
            {
              "name": "Value",
              "attribute": "value",
              "id": "value",
              "type": "STRING"
            }
          ],
          "attribute_name": "value",
          "constraint": {
            "occurrence": {
              "min": 1,
              "max_op": "\u003c\u003d",
              "min_op": "\u003e\u003d",
            }
          }
        }
      ]
    }
  ]
}
```

(continues on next page)

(continued from previous page)

```

        "max": 1
      }
    },
    "type": "DV_TEXT"
  }
],
"node_id": "at0001"
},
{
  "aql_path": "/context/other_context[at0001]/items[openEHR-EHR-CLUSTER.case_
↪identification.v0]/items",
  "name": "Items",
  "attribute": "items",
  "id": "items",
  "occurrence": {
    "min": 1,
    "max_op": "\u003c\u003d",
    "min_op": "\u003e\u003d",
    "max": 1
  },
  "category": "ATTRIBUTE",
  "type": "ITEM"
}In other terms, t
],
"name": "Fallidentifikation",
"description": "Zur Erfassung von Details zur Identifikation eines Falls im_
↪Gesundheitswesen.",
"id": "fallidentifikation",
"type": "CLUSTER",
"category": "DATA_STRUCTURE",
"node_id": "openEHR-EHR-CLUSTER.case_identification.v0"
},

```

Whenever the node\_id is not specified, the jsonpath expression uses class names. For example the following AQL

```
SELECT location FROM EHR e CONTAINS COMPOSITION CONTAINS ADMIN_ENTRY CONTAINS
location [openEHR-EHR-CLUSTER.location.v1]
```

Is translated as:

```
$..[?(@.type == 'COMPOSITION')]..[?(@.type == 'ADMIN_ENTRY')]..[?(@.node_id ==
'openEHR-EHR-CLUSTER.location.v1')]
```

## AQL Clause Interpretation

Contains clause interpretation consists in parsing the AQL expression (ANTLR) and create a corresponding list of propositions to evaluate.

The logic is based on the recursive traversal of the tree expression (AST), from bottom left to the top of the tree, and create the template traversal query as well as the boolean validations if any if the expression contains logical operators (AND, OR, XOR ...).

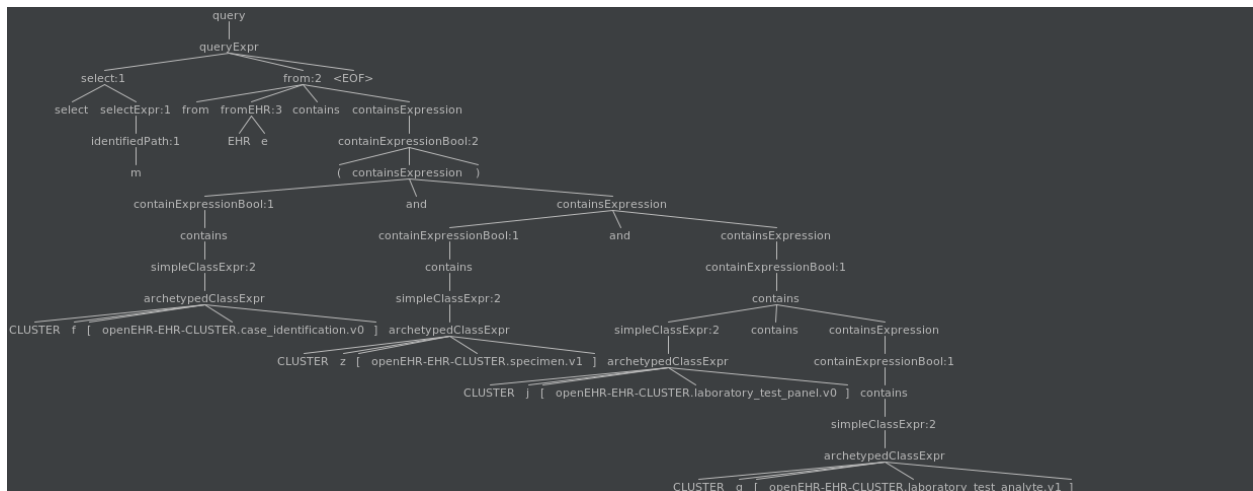
The evaluation does check first simple containment chains (CONTAINS...CONTAINS...CONTAINS...) using WebTemplate traversals described above, and then checks the logical propositions based on these.

## Example

AQL expression:

```
select
m
from EHR e
contains (
  CLUSTER f[openEHR-EHR-CLUSTER.case_identification.v0] and
  CLUSTER z[openEHR-EHR-CLUSTER.specimen.v1] and
  CLUSTER j[openEHR-EHR-CLUSTER.laboratory_test_panel.v0]
  contains CLUSTER g[openEHR-EHR-CLUSTER.laboratory_test_analyte.v1])
```

The containments are evaluated with the following tree



The containments are evaluated as follows:

1. “CLUSTERf[openEHR-EHR-CLUSTER.case\_identification.v0]” -
2. “CLUSTERz[openEHR-EHR-CLUSTER.specimen.v1]”
3. “CLUSTERg[openEHR-EHR-CLUSTER.laboratory\_test\_analyte.v1]” as in CLUSTER j[openEHR-EHR-CLUSTER.laboratory\_test\_panel.v0] contains CLUSTER g[openEHR-EHR-CLUSTER.laboratory\_test\_analyte.v1])
4. “CLUSTERz[openEHR-EHR-CLUSTER.specimen.v1] and CLUSTERj[openEHR-EHR-CLUSTER.laboratory\_test\_panel.v0]containsCLUSTERg[openEHR-EHR-CLUSTER.laboratory\_test\_analyte.v1]”: check the INTERSECTION of the results from 2 AND 3 above
5. “CLUSTERf[openEHR-EHR-CLUSTER.case\_identification.v0] and CLUSTERz[openEHR-EHR-CLUSTER.specimen.v1]andCLUSTERj[openEHR-EHR-CLUSTER.laboratory\_test\_panel.v0]containsCLUSTERg[openEHR-EHR-CLUSTER.laboratory\_test\_analyte.v1]”: check the INTERSECTION of the results from 1 & 4
6. “(CLUSTERf[openEHR-EHR-CLUSTER.case\_identification.v0]andCLUSTERz[openEHR-EHR-CLUSTER.specimen.v1]andCLUSTERj[openEHR-EHR-CLUSTER.laboratory\_test\_panel.v0]containsCLUSTERg[openEHR-EHR-CLUSTER.laboratory\_test\_analyte.v1])”: same as 5 since it is enclosed in parenthesis.

If another operator is used: OR or XOR, then we apply UNION or DISJUNCTION respectively.

## DB Changes

The two most significant changes are

1. Deprecation of table ehr.containment. This table is now removed, as well as all logic associated to its population.
2. New encoding of composition entry (item\_structure)

The composition entry encoding (jsonb) has now the composition name encoded outside the json structure as a dv\_coded\_text (UDT) in table ehr.entry and removed from the archetype node id in the composition path.

This change is required since now the identified path is a generic AQL path without composition dependent values.

```
{
  "/name": [
    {
      "value": "Bericht"
    }
  ],
  "$CLASS$": "Composition",
  "/composition[openEHR-EHR-COMPOSITION.report.v1 and name/value='Bericht']": {
    "/content[openEHR-EHR-OBSERVATION.blood_pressure.v2]": [
      {
        "/name": [
          {
            "value": "Blutdruck"
          }
        ],
        "$CLASS$": "Observation"
      }
    ]
  }
}
```

The name/value attribute in the node id is now passed as an external attribute 'name' and the composition item\_structure is encoded as

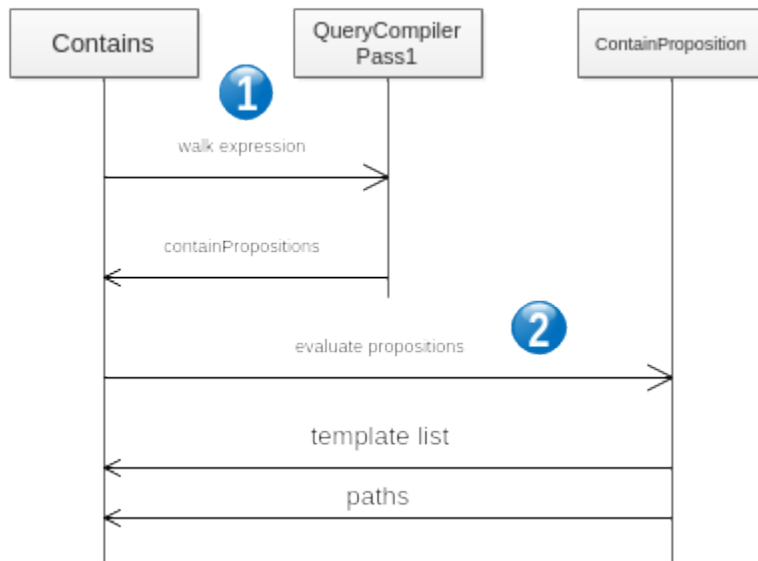
```
{
  "/name": [
    {
      "value": "Bericht"
    }
  ],
  "$CLASS$": "Composition",
  "/composition[openEHR-EHR-COMPOSITION.report.v1]": {
    "/content[openEHR-EHR-OBSERVATION.blood_pressure.v2]": [
      {
        "/name": [
          {
            "value": "Blutdruck"
          }
        ],
        "$CLASS$": "Observation"
      }
    ]
  }
}
```

While name is

(Bericht,,,,)

## Processing

The sequence of containment resolution is the following



1. Consists in parsing the AQL CONTAINS expression and build the propositions as described above.
2. The propositions are evaluated as
  1. Simple containment chains using cached WebTemplates
  2. Computed boolean expressions based on the simple containment chains

## Further Enhancements

1. At this stage, ehr\_status/other\_details is not part of the contains resolution. The main issue here is that it is generally not associated to a valid template.
2. There need to do more research for archetype\_slots in a ANY type.

## 3.6 Security

**Warning:** WIP

For the moment, please see the [EHRbase GitHub repository](#) for security information, issue tracker and the source code.

## 3.7 Admin API

**Warning:** WIP

**Warning:** Please be aware of potential security and consistency risks in production if API security configuration is not done properly.

---

**Important:** The Admin REST API is not part of the official openEHR standard. This is an additional feature provided by the EHRbase team to support development and system administrators.

---

This section covers the Admin API for EHRbase which can be used for administrative tasks or help in development.

To generally enable the Admin API set the `ADMINAPI_ACTIVE` environment variable to true (see [Spring Boot Externalized Configuration](#) for more details and options on how to set such configuration attributes).

The Admin API interface is available at the “/admin” resource which will be appended to the base URL of the openEHR REST interface. E.g. if the base URL is “<https://api.ehrbase.org/ehrbase>” the admin API and all sub resources are available at “<https://api.ehrbase.org/ehrbase/rest/admin>”.

### 3.7.1 Security

This security documentation describes how to configure the target system for using and securing the admin API resources.

#### General

The Admin API resources allow several operations on data that circumvent the versioning of changes in the openEHR system. For instance, after using a DELETE operation via the standard client API the entries will still be available inside the history tables and all changes to that resources can be seen in the audit of the data entry. The admin API allows the physical deletion of the entry and all of the history entries as well, thus the changes cannot be traced and the data is completely lost.

In health IT systems all operations on data must be stored in a manipulation safe way and all changes must be traceable. Our recommendation is therefore to not use the admin API in production.

During development of EHRbase and connected systems it is often necessary to replace or remove data from the system completely. This could also be done via common database tools, but this solution is often very cumbersome and not cleaning up all data related to the resource.

#### Role based access control

As a minimum security measurement the EHRbase Security configuration should use a role based access control. Independent from the selected security mechanism (Basic auth, OpenID-Connect, etc.) each resource request should be checked for the users role and then be permitted or rejected.

We are currently using this two roles in the system:

Role	Permissions
user	Can access all client resources specified by openEHR REST API specification and do not have access to all resources in the /admin and subsequent endpoints
ad-min	Can access all resources; i.e. all client resources and the /admin endpoints

## Security related response codes

Execution of requests to each endpoint can have two security related error codes that have to be handled by client applications:

Code	Reason	Possible Solution
401	Authorization information is incorrect / expired	Reauthenticate the user via a new login or by refreshing the authorization information (id_token etc.)
403	Authorization information is valid but permission to requested resource is denied	The users role is not allowed to access this resource. Thus this is okay or the users role must be configured in the related security system (OpenID-Connect server or config files etc.)

Depending on the security configuration the 401 error could be handled automatically, e.g. if using OAuth2 the client could get a new id\_token if the used one has expired and then will re-do the request with new information. Also it could be possible to show the user a login screen to reauthenticate against the security system.

All resources in the admin API can return this response code and will not be mentioned in the documentation pages itself.

### 3.7.2 /admin/ehr

Methods to administrate the EHR resource.

#### DELETE /admin/ehr/{:ehr\_id}

Remove an EHR and related data physically from the database.

#### Request format

The request should be formatted as follows:

##### Headers

There are no headers required.

##### Body

No body required

#### Response format

The response will be formatted as follows:

##### Headers

There are no extra headers returned

##### Status Codes

Depending on several request conditions or errors during the request handling there will be one of the following status codes returned:

- 204 (NO CONTENT) EHR has been deleted successfully.

- 404 (NOT FOUND) The EHR with provided id cannot be found. This can also occur if the id is not in valid HIER\_OBJECT\_ID format.

**Body**

No body returned

### 3.7.3 /admin/{:ehr\_id}/composition

Methods to administrate the Composition resource.

#### **DELETE /admin/{:ehr\_id}/composition/{:composition\_id}**

Delete the composition identified by “composition\_id” physically from database. This will also remove all history information on the related composition and their entries.

The target “composition\_id” must be formatted as versioned object uid (i.e. UUID). In contrast to the client API this method will physically remove the given composition and all linked metadata.

#### **Request format**

The request should be formatted as follows:

**Headers**

There are no headers required.

**Body**

No body required

#### **Response format**

The response will be formatted as follows:

**Headers**

There are no extra headers returned

**Status Codes**

Depending on several request conditions or errors during the request handling there will be one of the following status codes returned:

- 204 (NO CONTENT) Composition has been deleted successfully.
- 404 (NOT FOUND) The Composition with provided id cannot be found. This can also occur if the id is not in valid UID format.

**Body**

No body returned

### 3.7.4 /admin/{:ehr\_id}/contribution

Methods to administrate the Contribution resource.



**DELETE /admin/{:ehr\_id}/contribution/{:contribution\_id}**

Remove an Contribution and related data physically from the database.

**Request format**

The request should be formatted as follows:

**Headers**

There are no headers required.

**Body**

No body required

**Response format**

The response will be formatted as follows:

**Headers**

There are no extra headers returned

**Status Codes**

Depending on several request conditions or errors during the request handling there will be on of the following status codes returned:

- 204 (NO CONTENT) Contribution has been deleted successfully.
- 404 (NOT FOUND) The Contribution with provided id cannot be found. This can also occur if the id is not in valid UUID format.

**Body**

No body returned

**3.7.5 /admin/{:ehr\_id}/directory**

Methods to administrate the Folder resource.

**DELETE /admin/{:ehr\_id}/directory/{:folder\_id}**

Delete the Folder identified by “folder\_id” physically from database. This will also remove all history information on related folders and their hierarchies.

The target “folder\_id” must formatted as versioned object uid (i.e. UUID).

**Request format**

The request should be formatted as follows:

**Headers**

There are no headers required.

**Body**

No body required

**Response format**

The response will be formatted as follows:

**Headers**

There are no extra headers returned

**Status Codes**

Depending on several request conditions or errors during the request handling there will be one of the following status codes returned:

- 204 (NO CONTENT) Folder has been deleted successfully.
- 404 (NOT FOUND) The Folder with provided id cannot be found. This can also occur if the id is not in valid UID format.

**Body**

No body returned

**3.7.6 /admin/template**

Methods to administrate the Template resource.

**PUT /admin/template/:template\_id**

Replace an existing template with the new provided data.

**Request format**

The request should be formatted as follows:

**Headers**

Header-Name	Re-quired	Description	Accepted values
Accept		Desired response format after a successful update operation.	application/json; application/xml
Content-Type	Yes	Format of the content body	application/json; application/xml
Prefer		Tell the API if you want the updated template data to be returned or not.	return=representation   return=minimal

**Body**

The request body can contain the same data as for the client POST request in the desired format.

## Response format

The response will be formatted as follows:

### Headers

Header-Name	Description
Content-Type	Returned data type. Depends on data type sent with “Accept” header.

### Status Codes

Depending on several request conditions or errors during the request handling there will be one of the following status codes returned:

Code	Cause/Meaning
200 (OK)	Template has been replaced successfully and the body contains the new Template data after the update.
204 (NO CONTENT)	Template has been replaced successfully and the body contains no data since “Prefer” header was set with “return=minimal”
400 (BAD REQUEST)	The body contains invalid data to replace the existing content; e.g. missing mandatory fields or data structures that could not be serialized.
404 (NOT FOUND)	The Template with provided id cannot be found.

### Body

Whether the clients requested “Prefer” header setting the full new Template entry after the updated has been applied will be returned or it will be empty.

## DELETE /admin/template/:template\_id

Delete the template identified by “template\_id” physically from server. Depending on your implementation the entry has to be removed from file or database storage.

## Request format

The request should be formatted as follows:

### Headers

There are no headers required.

### Body

No body required

## Response format

The response will be formatted as follows:

### Headers

There are no extra headers returned

### Status Codes

Depending on several request conditions or errors during the request handling there will be one of the following status codes returned:

Code	Cause/Meaning
204 (NO CONTENT)	Template has been deleted successfully.
404 (NOT FOUND)	The Template with provided id cannot be found.
422 (UNPROCESSABLE ENTITY)	The Request was correct and template can be found but it is still used by compositions.

### Body

No body returned

## DELETE /admin/template/all

Delete all templates physically from server. Depending on your implementation the entries has to be removed from file or database storage.

---

**Note:** The EHRbase environment variable “ADMINAPI\_ALLOWDELETEALL” must be set to true. Otherwise the endpoint does not accept requests.

---

### Request format

The request should be formatted as follows:

#### Headers

There are no headers required.

#### Body

No body required

### Response format

The response will be formatted as follows:

#### Headers

There are no extra headers returned

#### Status Codes

Depending on several request conditions or errors during the request handling there will be one of the following status codes returned:

Code	Cause/Meaning
200 (OK)	Templates have been deleted successfully.
422 (UNPROCESSABLE ENTITY)	The Request was correct but there are templates which are still used by compositions.

### Body

For 200 (OK): The number of deleted templates is returned in the following schema:

```
{
  "deleted": integer
}
```

For 422 (UNPROCESSABLE ENTITY): Body contains message with list of Compositions that are referencing at least one Template.

## 3.8 Status and Metrics

**Note:** Added in EHRbase version 0.15.0

**Warning:** The status and metrics endpoints can contain critical and sensitive information on the running ehrbase instance that could serve possible attackers to identify vulnerabilities. Please ensure to use a solid security config on the EHRbase server or disable the metrics and status endpoints in production environments with sensitive data.

This section covers the status and metric endpoints provided by Spring-Boot-Actuator to provide additional information on the running EHRbase server including metrics on the usage of the API since last boot.

The Status and Metric endpoints provide additional information which is useful in environments that use a bunch of microservices that rely on each other as in cloud orchestration like Kubernetes. The status endpoint can support the management of these complex systems by providing *liveness* and *readiness* endpoints which allow orchestration tools to check whether the EHRbase service is running and if it capable of serving incoming requests.

Additional metrics allow to identify bottlenecks in connections between services as between EHRbase service and a connected database server if the connection metric shows very long durations for communications or if the connection pool limit is exhausted. Another possible use case could be detection of attacks against the EHRbase service due to a high occurrence of authorization client errors. As you see there are many things you can do with these endpoints.

The Status and Metrics API interface is available at the “/management” resource which will be appended to the base URL of the ehrbase instance. E.g. if EHRbase is running at “<https://api.ehrbase.org>” the status API and all sub endpoints are available at “<https://api.ehrbase.org/ehrbase/management>”.

### 3.8.1 Security

**Note:** Added in EHRbase version 0.15.0

Since the Status and Metric endpoints provide such valuable information they are secured by EHRbase integrated security solutions. If you are using at least the *Basic Auth* method to secure incoming requests the endpoints are only available for users with the *Admin-Role*.

If you want to access specific metric and health endpoint from a statistics and management service ensure not sending the passwords in clear text. For Basic Auth encode them into a Base64 string as described here: [EHRbase Security docs](#).

**Warning:** All auth methods can be attacked easily if you do not use HTTPS encrypted communication outside trusted networks as internal VPN or secured cloud system networks. Ensure to secure your environment properly.

For OAuth2 methods please configure the auth server appropriate and provide the *Admin*-Role to the user logging in. Also check your monitoring and management service manuals how to authorize against the OAuth2 service and how to obtain new access tokens on token expiration (refresh).

### 3.8.2 Usage

**Note:** Added in EHRbase version 0.15.0

By default all status and metric endpoints are disabled in EHRbase. To opt-in endpoints you should start the EHRbase with an environment variable per endpoint you want to enable.

#### Run from command line:

```
$~/ehrbase: export MANAGEMENT_ENDPOINT_ENV_ENABLE=true
$~/ehrbase: java -jar application/target/application-0.15.0.jar
```

#### Run from Docker container:

```
$~/: docker run -e MANAGEMENT_ENDPOINT_ENV_ENABLE=true --name ehrbase --network_
↪ehrbase-net -p 8080:8080 -d ehrbase/ehrbase:latest
```

The following table lists all available endpoints that can be enabled with environment variables:

Parameter	Usage	Example
MANAGEMENT_ENDPOINT_ENV_ENABLED	Enable /management/env endpoint from actuator	true / false
MANAGEMENT_ENDPOINT_HEALTH_ENABLED	Enable /management/health endpoint from actuator	true / false
MANAGEMENT_ENDPOINT_INFO_ENABLED	Enable /management/info endpoint from actuator	true / false
MANAGEMENT_ENDPOINT_METRICS_ENABLED	Enable /management/metrics endpoint from actuator	true / false
MANAGEMENT_ENDPOINT_PROMETHEUS_ENABLED	Enable /management/prometheus endpoint from actuator	true / false

Additionally you can configure the following actuator settings if required:

Parameter	Usage	Example		
MANAGEMENT_ENDPOINT_HEALTH_PROBES_ENABLED	Enable Kubernetes probe endpoints /management/health/readiness and /management/health explicit in non Kubernetes environments	true/false		
MANAGEMENT_ENDPOINTS_WEB_EXPOSURE	Expose enabled endpoint to clients. Only set if required	env	health	info
MANAGEMENT_ENDPOINTS_WEB_BASEPATH	Change base path for all endpoints	/management		

### 3.8.3 /management/env

---

**Note:** Added in EHRbase version 0.15.0

---

The `/env` endpoint serves information of the environment that is used to run the EHRbase currently. These information can include the running operating system type, architecture and version as well as information on the Java Runtime Executable.

An example response on a Windows system running EHRbase could look like this:

```
{
  "activeProfiles": [
    "local"
  ],
  "propertySources": [
    {
      "name": "server.ports",
      "properties": {
        "local.server.port": {
          "value": 8080
        }
      }
    },
    {
      "name": "servletContextInitParams",
      "properties": {}
    },
    {
      "name": "systemProperties",
      "properties": {
        "sun.desktop": {
          "value": "windows"
        },
        "awt.toolkit": {
          "value": "sun.awt.windows.WToolkit"
        },
        "java.specification.version": {
          "value": "11"
        }
      }
    }
  ]
}
```

### 3.8.4 /management/health

---

**Note:** Added in EHRbase version 0.15.0

---

The health endpoint provides information on the current status of the running EHRbase service process. Therefore it can provide information on *liveness* (service started) and *readiness* (service is available to serve requests) which is often used by cloud orchestration software like Kubernetes.

If Spring-Boot-Actuator detects a Kubernetes environment the probes for liveness and readiness are enabled automatically. If you like to test the endpoints outside a Kubernetes environment open file *application.yml* inside application module and set value *management->endpoint->health->probes-> enabled* to **true** and re-build and re-start EHRbase.

An example response for health on Kubernetes would look like this:

```
{
  "status": "UP",
  "components": {
    "db": {
      "status": "UP",
      "details": {
        "database": "PostgreSQL",
        "validationQuery": "isValid()"
      }
    },
    "diskSpace": {
      "status": "UP",
      "details": {
        "total": 1901064089600,
        "free": 1775335157760,
        "threshold": 10485760,
        "exists": true
      }
    },
    "livenessState": {
      "status": "UP"
    },
    "ping": {
      "status": "UP"
    },
    "readinessState": {
      "status": "UP"
    }
  }
}
```

This example also contains the output of the probes for liveness and readiness. If the status equals **UP** the probe reports success.

See the [official documentation](#) for more information.

### 3.8.5 /management/info

---

**Note:** Added in EHRbase version 0.15.0

---

The service /info exposes common information about the build artifact that contains the current running EHRbase instance. These information contain the current EHRbase version as well as versions of *Archie* and *openEHR\_SDK* that have been used to build this EHRbase artifact. You should include this version when creating a bug issue in any EHRbase related project.



This is an example response for the info service:

```
{
  "build": {
    "openEHR_SDK": {
      "version": "c44284e"
    },
    "version": "0.14.0",
    "artifact": "application",
    "archie": {
      "version": "738152e"
    },
    "name": "application",
    "time": "2020-12-14T15:59:15.097Z",
    "group": "org.ehrbase.openehr"
  }
}
```

### 3.8.6 /management/metrics

---

**Note:** Added in EHRbase version 0.15.0

---

The metrics service provides multiple useful data on the usage of EHRbase since startup of the service. These could be for example the number of requests that have been served to clients and also information on the response types and errors.

You can get an overview on all supported metrics that can be served with this endpoint via *GET* to the path */management/metrics*. This will return a list in JSON format with all possible values for more detailed metrics that have to be requested in a separate request.

Example output:

```
{
  "names": [
    "hikaricp.connections",
    "hikaricp.connections.acquire",
    "hikaricp.connections.active",
    "hikaricp.connections.creation",
    "hikaricp.connections.idle",
    "hikaricp.connections.max",
    "hikaricp.connections.min",
    "hikaricp.connections.pending",
    "hikaricp.connections.timeout",
    "hikaricp.connections.usage",
    "http.server.requests",
    "jdbc.connections.active",
    "jdbc.connections.idle",
    "jdbc.connections.max",
    "jdbc.connections.min",
    "jvm.buffer.count",
    "jvm.buffer.memory.used",
    "jvm.buffer.total.capacity",
    "jvm.classes.loaded",
    "jvm.classes.unloaded",
    "jvm.gc.live.data.size",
    "jvm.gc.max.data.size",
    "jvm.gc.memory.allocated",
    "jvm.gc.memory.promoted",
    "jvm.gc.pause",
    "jvm.memory.committed",
```

You can use the values from this list in a further request as the sub-path of metrics and you will get the detailed metric for that point.

As an example you can use the path **http.server.requests** to get information on all serves http requests as shown below:

```
{
  "name": "http.server.requests",
  "baseUnit": "seconds",
  "measurements": [
    {
      "statistic": "COUNT",
      "value": 18.0
    },
    {
      "statistic": "TOTAL_TIME",
      "value": 1.0883589
    },
    {
      "statistic": "MAX",
      "value": 0.0130664
    }
  ],
  "availableTags": [
    {
      "tag": "exception",
      "values": [
        "None"
      ]
    },
    {
      "tag": "method",
      "values": [
        "GET"
      ]
    }
  ]
}
```

This example shows also a list of available tags. If you see a list you can filter and aggregate the results by using query parameters on the same metric.

In our example we can also apply a filter for all unauthorized request by adding the tag like this: **/management/metrics/http.server.requests?tag=status:401** which could look like this:

```
{
  "name": "http.server.requests",
  "baseUnit": "seconds",
  "measurements": [
    {
      "statistic": "COUNT",
      "value": 6.0
    },
    {
      "statistic": "TOTAL_TIME",
      "value": 0.524429
    },
    {
      "statistic": "MAX",
      "value": 0.0
    }
  ],
  "availableTags": [
    {
      "tag": "exception",
      "values": [
        "None"
      ]
    }
  ]
}
```

You can find more information on using tags on metrics in the [official documentation](#).

### 3.8.7 /management/prometheus

---

**Note:** Added in EHRbase version 0.15.0

---

This endpoint provides metrics for the prometheus monitoring format and can be set as an target uri for external monitoring services to get the info from the running EHRbase instance.

You can find more information on this format in the [official Prometheus Github doc](#).

## 3.9 Attribute-based Access Control

This section covers EHRbase's Attribute-based Access Control (ABAC) system and how to configure it.

### 3.9.1 Concept

EHRbase offers a predefined set of rules, logics and configurations to enable and use ABAC checks for several operations.

ABAC checks are available for many endpoints. Endpoints are clustered and processed by resource/path. So all .../composition/... endpoints are defined the same and run through the same logic.

There are five of those resource groups:

- EHR
- EHR Status
- Composition
- Contribution
- Query

---

**Note:** Directory endpoints are only handling references. Definitions aren't patient (or most likely practitioner) facing. So both aren't available for ABAC checks.

---

Each can be configured to create an ABAC request with

- the given policy name and
- a set of parameters (rendered as simple JSON request body)

The set of parameters can be composed via configuration from the predefined set of attributes:

- Organization (taken from the JWT claim)
- Patient (taken from the JWT claim, with current scope's EHR subject as internal comparison)
- Template ID (taken from the scopes composition, where applicable)

### 3.9.2 Configuration

Enable with `ABAC_DISABLED=false` or matching `application.yml` attribute.

---

**Note:** The ABAC handling relies on OAuth2 tokens. Therefore, ABAC can practically only be used together with OAuth enabled.

---

The ABAC server can be set with `ABAC_SERVER`. The URL is expected to end with `/` as the ABAC requests will directly add the policy name like: `“.../policy/execute/name/”` will be modified to `“.../policy/execute/name/has_consent_template”` in a specific instance.

Real example to enable ABAC with OAuth: `SECURITY_AUTHTYPE=OAUTH;ABAC_DISABLED=false;ABAC_SERVER=http://localhost:3001/rest/v1/policy/execute/name/;SPRING_SECURITY_OAUTH2_RESOURCESERVER_JWT_ISSUERURI=https://keycloak.PROJECT.com/auth/realms/ctr`

Both the organization and patient claim of the expected JWT can be configured via `ABAC_ORGANIZATIONCLAIM` and `ABAC_PATIENTCLAIM` respectively. Their default is `organization_id` and `patient_id`.

For the configuration of the endpoints please refer to the `application.yml`. Here is one group's section as example, where the composition endpoints are configured to have the policy `has_consent_template` with the listed attributes in the request body:

```
policy:
  composition:
    name: 'has_consent_template'
    parameters: 'organization, patient, template'
```

**Warning:** If parameters like `organization` and `patient` are configured to be used by the logic, but the context's JWT doesn't carry the matching claim the ABAC check will fail.

### 3.9.3 Detailed endpoint overview

#### EHR

Enabled ABAC-parameters: `organization`, `patient`

Disabled: `template` - not valid in scope of EHR

Implemented and tested:

- `Get(s)`

Not included:

- `Post`, `Put`: Completely new EHR has no subject context and EHR has no template. No ABAC checks.

#### EHR Status

Enabled ABAC-parameters: `organization`, `patient`

Disabled: `template` - not valid in scope of `EHR_STATUS`

Implemented and tested:

- `Get(s)`
- `Put`
- `Get(s)` - Versioned sub-type - `EHR_STATUS` version

#### Composition

Enabled ABAC-parameters: `organization`, `patient`, `template`

Implemented and tested:

- `Get(s)`
- `Post`, `Put`
- `Delete`
- `Get(s)` - Versioned sub-type - `Composition` version

#### Contribution

Enabled ABAC-parameters: `organization`, `patient`, `template`

Implemented and tested:

- `Post`

Not included:

- `Get`: Only returns references, so no ABAC checks necessary.

## Query

Enabled ABAC-parameters: organization, patient, template

Note: Currently this handling relies on the AuditResultMap of the QueryService to provide the ABAC logic with patient and template context data. This entails, that patient and template ID(s - multiple per query possible) are only given, when the query directly sets them in the SELECT.

Implemented and tested:

- All four query endpoints





This section gives information about the openEHR Software Development Kit.

Please also see the [openEHR SDK Github repository](#).

## 4.1 Guides

**Warning:** WIP

### 4.1.1 SDK as dependency

**Warning:** WIP

To make use of the SDK's features it needs to be included as dependency.

For instance, to build a simple client, include the `client` module as dependency.

Depending on the project structure and used dependency management tools this might look like the following `pom.xml` snippet in a Maven example:

```
<properties>
  <!-- ... -->
  <ehrbase.sdk.version>$VERSION_TAG_OR_LATEST_COMMIT_HASH</ehrbase.sdk.version>
</properties>

<repositories>
  <!-- ... -->
  <!-- external -->
  <repository>
```

(continues on next page)

(continued from previous page)

```

        <id>jitpack.io</id>
        <url>https://jitpack.io</url>
    </repository>
</repositories>

<dependencyManagement>
    <dependencies>
        <!-- ... -->
        <dependency>
            <groupId>com.github.ehrbase.openEHR_SDK</groupId>
            <artifactId>client</artifactId>
            <version>${ehrbase.sdk.version}</version>
        </dependency>
    </dependencies>
</dependencyManagement>

<dependencies>
    <!-- ... -->
    <dependency>
        <groupId>com.github.ehrbase.openEHR_SDK</groupId>
        <artifactId>client</artifactId>
    </dependency>
</dependencies>

```

## 4.2 Reference

**Warning:** WIP

Reference documentation of the openEHR Software Development Kit from EHRbase.

### 4.2.1 Client module

**Warning:** WIP

Reference documentation of the client module.

**Note:** Please have a look at the integration tests for many working examples. (Tests named \*IT in `src/test/java/org/ehrbase/client/openehrclient/defaultrestclient/`)

#### Client

**Warning:** WIP

The openEHR Client is the foundation of all following functionalities. It needs to be created and set up before the endpoints can be used.

## Interface and implementation

All compatible clients need to implement the `OpenEhrClient` Interface.

Standard operation is possible with the included `DefaultRestClient` implementation.

## Client setup

To set up a default client it is necessary to create a new instance, which requires the following components as parameters:

- `OpenEhrClientConfig` containing the base URI of the openEHR REST API backend server
- And a `TemplateProvider` (see below)

Together a typical setup might look like:

```
DefaultRestClient client = new DefaultRestClient(
    new OpenEhrClientConfig(new URI("http://localhost:8080/ehrbase/rest/openehr/v1/"),
    templateProvider);
```

## Template provider

The `TemplateProvider` interface needs an implementation that gives the SDK access to the templates used.

Technically, it needs to provide a function `Optional<OPERATIONALTEMPLATE> find(String s)` which returns an object representation of a given template ID.

An example is the `TestDataTemplateProvider` from the SDK's own integration tests:

```
@Override
public Optional<OPERATIONALTEMPLATE> find(String templateId) {
    return Optional.ofNullable(OperationalTemplateTestData.
    findByTemplateId(templateId))
        .map(OperationalTemplateTestData::getStream)
        .map(s -> {
            try {
                return TemplateDocument.Factory.parse(s);
            } catch (XmlException | IOException e) {
                throw new RuntimeException(e.getMessage(), e);
            }
        })
        .map(TemplateDocument::getTemplate);
}
```

Please also see the used `OperationalTemplateTestData` enum to understand how actual .opt files can be made accessible.

### Additional steps

Now the client is set up and ready to be used. Before going on it might make sense to sync your backend with the templates used by your client.

The template provider could have a `listTemplateIds()` method for that purpose. So the following would use this method and the template endpoint (see *Template endpoint*) to make sure all templates of this clients scope are available in the backend.

```
templateProvider.listTemplateIds().forEach(  
    t -> client.templateEndpoint().ensureExistence(t)  
);
```

### EHR Endpoint

**Warning:** WIP

The `DefaultRestClient` includes a `DefaultRestEhrEndpoint` with the following functionalities.

#### Create an EHR

The creation of a new EHR is as simple as calling:

```
UUID ehr = openEhrClient.ehrEndpoint().createEhr();
```

---

**Note:** The option to add a custom EHR Status object at this step is already on the road map.

---

#### Get the EHR status

Retrieval of the EHR Status of a given EHR is done with a call like:

```
Optional<EhrStatus> ehrStatus = openEhrClient.ehrEndpoint().getEhrStatus(ehrId);
```

#### Update the EHR status

Updating works in the same way and might be used like in the following:

```
// Retrieval and modification of the Status  
...  
// Followed by updating it  
openEhrClient.ehrEndpoint().updateEhrStatus(ehrId, ehrStatus);
```

### Template Endpoint

The `DefaultRestClient` includes a `DefaultRestTemplateEndpoint` with the following functionalities.

**Warning:** WIP

## Ensure existence of a template

Uploading necessary operational templates is vital for many other client functions.

The `ensureExistence(String templateId)` method combines a check and the upload, if the template is not already available on the server.

The following example utilizes a `TemplateProvider` (see [Client reference](#)) to go through all templates of the client's scope and ensure their existence:

```
templateProvider.listTemplateIds().forEach(
    t -> client.templateEndpoint().ensureExistence(t)
);
```

## Composition Endpoint

**Warning:** WIP

The `DefaultRestClient` includes a `DefaultRestCompositionEndpoint` with the following functionalities.

## Commit composition

Committing a composition is done using the `mergeCompositionEntity` method. Its idea is that the client already works with an instance of the composition (of a generated template entity type, like described [here](#)).

For example, a new instance could be created and modified to contain two values entered via a user interface. This object now might only contain those two values, and a representation of the structure of the composition (as generated type/class definition).

**Note:** Despite that this simplified example only mentions two values, it is still necessary to provide data to all fields, which are required as per openEHR Reference Model. Otherwise the backend server might reject the payload as invalid.

Afterwards, calling the `mergeCompositionEntity` method would commit the composition to the server. Before returning data, this method also processes the server response to enrich the given composition object. Specifically, server-side created data values like the composition's Version Uid are written into the object.

Therefore, and after successful execution, the method returns a representation of the composition as it is now persisting on the backend server.

A very simplified example could look like:

```
// Initializing the object
EhrbaseBloodPressureSimpleDev0Composition composition = new
↳ EhrbaseBloodPressureSimpleDev0Composition();
```

(continues on next page)

(continued from previous page)

```
// Adding values
[...].setSystolicMagnitude(120.0);
[...].setSystolicUnits("mm[Hg]");
[...].setDiastolicMagnitude(120.0);
[...].setDiastolicUnits("mm[Hg]");

// Committing and altering the object with the response data
client.compositionEndpoint(ehrId).mergeCompositionEntity(composition);

// For instance, the specific Version UID can now be accessed
composition.getVersionUid();
```

## Find composition

To retrieve the latest version of a specific composition - or to get response that allows to understand that no such composition exists - the `find` method can be used.

The usage is illustrated in the following example:

```
UUID compositionId = $COMPOSITION_ID;
Optional<EhrbaseBloodPressureSimpleDev0Composition> compo = compositionEndpoint
    .find(compositionId, EhrbaseBloodPressureSimpleDev0Composition.class);
```

## AQL Endpoint

**Warning:** WIP

The `DefaultRestClient` includes a `DefaultRestAqlEndpoint` with the following functionalities.

## Query execution

The execution is the only function of this endpoint. It supports two types of query implementations, both based on a common definition.

## Abstract query

The basic querying concept is inspired by libraries and tools like `jOOQ` and therefore focuses on typesafe AQL, based on generated entity code and record handling.

The following examples illustrates that with a simple native AQL query. All types of `Query` and their detailed usage will be discussed below.

```
private List<UUID> queryAllEhrs() {
    Query<Record1<UUID>> query = Query.buildNativeQuery("SELECT e/ehr_id/value FROM_
    ↪EHR e", UUID.class);
    List<Record1<UUID>> result = client.aqlEndpoint().execute(query);

    List<UUID> ehRs = new ArrayList<>();
```

(continues on next page)

(continued from previous page)

```

    result.forEach(r -> ehRs.add(r.value1()));
    return ehRs;
}

```

In this example a query is created. Its return type is already embedding a `Record`, which is defined depending on the amount and type of the result values. Here, the query has one *select value*. The response will only have *one* column of type `UUID`, so together the response is defined as `Record1<UUID>`.

After execution the result is wrapped in a `List`, because it can have none or many result values (rows). Accessing the values is type safe, since the record was already defined with the specific type.

Records are supported up to 21 result types (`Record21`). There are no restrictions on which types can be used.

## Native query

A native query can be build with the input of a string representation of a native AQL query. For instance, `SELECT e/ehr_id/value FROM EHR e`.

Additionally, parameters can be used and added as Java variables. They need to be formatted like `$ehr_id` in the native query string. Setting its value is done using the `ParameterValue` class. See the following example:

```

Query<Record2<VersionUid, TemporalAccessor>> query = Query.buildNativeQuery(
    "select  a/uid/value, a/context/start_time/value from EHR e[ehr_id/value = $ehr_
    ↪id]
        contains COMPOSITION a [openEHR-EHR-COMPOSITION.sample_encounter.v1]",
    VersionUid.class, TemporalAccessor.class);

List<Record2<VersionUid, TemporalAccessor>> result = openEhrClient.aqlEndpoint().
    ↪execute(
        query, new ParameterValue("ehr_id", ehr));

```

## Entity query

In contrast, entity queries make use of the entities generator by the Generator (see [here](#)). Therefore, is it not necessary to manually design and implement a custom native AQL query.

The generator creates Java POJO classes of the composition, but also containment classes that can be used in a query. In the following example a simple exemplary blood pressure type is assumed to be generated already.

```

EhrbaseBloodPressureSimpleDeV0CompositionContainment containmentComposition =
    ↪EhrbaseBloodPressureSimpleDeV0CompositionContainment.getInstance();
BloodPressureTrainingSampleObservationContainment containmentObservation =
    ↪BloodPressureTrainingSampleObservationContainment.getInstance();

containmentComposition.setContains(containmentObservation);

EntityQuery<Record3<TemporalAccessor, BloodPressureTrainingSampleObservation,
    ↪CuffSizeDefiningcode>> entityQuery = Query.buildEntityQuery(
    containmentComposition,
    containmentComposition.START_TIME_VALUE,
    containmentObservation.BLOOD_PRESSURE_TRAINING_SAMPLE_OBSERVATION,
    containmentObservation.CUFF_SIZE_DEFININGCODE
);
// plus some conditions like WHERE, see below

```

This snippet shows three things:

First, the usage of the containment classes to define parts of the query (like select values). The internal query handling in `buildEntityQuery` will create a correct native AQL query from the input. This makes AQL querying much easier, because it removed the burden of being an expert on the query language.

Second, next to being able to retrieve simple types, this query also shows automatic parsing and conversation to complex result types like an `openEHR Observation`. Specifically, a `BloodPressureTrainingSampleObservation` type is directly available for further processing, after the query was executed.

Third, the `setContains` method simplifies the building of native AQL strings like `COMPOSITION c0[openEHR-EHR-COMPOSITION.report.v1]` contains `SECTION s1[openEHR-EHR-SECTION.adhoc.v1]` (different example).

Additionally, entity queries are supporting an included set of **conditions**:

- `and`
- `or`
- `not`
- `equal`
- `notEqual`
- `greaterOrEqual`
- `greaterThan`
- `lessOrEqual`
- `lessThan`
- `matches`
- `exists`

These methods can be used to build AQL conditions, like in the following example:

```
Condition condition1 = Condition.greaterOrEqual(containmentObservation.DIASTOLIC_
↳MAGNITUDE, 13d);
Condition condition2 = Condition.notEqual(containmentObservation.MEAN_ARTERIAL_
↳PRESSURE_UNITS, "mh");
Condition condition3 = Condition.lessThan(containmentObservation.TIME_VALUE,
↳OffsetDateTime.of(2019, 04, 03, 22, 00, 00, ZoneOffset.UTC));

Condition cut = condition1.and(condition2.or(condition3));

assertThat(cut.buildAql()).isEqualTo("(v/data[at0001]/events[at0002]/data[at0003]/
↳items[at0005]/value/magnitude >= 13.0 and " +
    "(v/data[at0001]/events[at0002]/data[at0003]/items[at1006]/value/units !=
↳'mh' or v/data[at0001]/events[at0002]/time/value < '2019-04-03T22:00:00Z') " +
    ")");
```

Finally, entity queries can use those *conditions* and other specific logical expressions to create

- **WHERE** (see *Observation* above)
- **ORDER BY** (`ascending`, `descending`, `andThenAscending`, `andThenDescending`)
- **TOP** (`forward`, `backward`)

clauses. See the following example:



```
EntityQuery<Record1<EhrbaseBloodPressureSimpleDeV0Composition>> entityQuery = Query.
↳buildEntityQuery(
    containmentComposition,
    containmentComposition.EHRBASE_BLOOD_PRESSURE_SIMPLE_DE_V0_COMPOSITION
);
Parameter<UUID> ehrIdParameter = entityQuery.buildParameter();

Condition where = Condition.equal(EhrFields.EHR_ID(), ehrIdParameter);
OrderByExpression orderBy = OrderByExpression.descending(containmentObservation.
↳SYSTOLIC_MAGNITUDE).andThenAscending(containmentObservation.DIASTOLIC_MAGNITUDE);
entityQuery.where(where).orderBy(orderBy);
```

## Directory Endpoint

### Warning: WIP

The `DefaultRestClient` includes a `DefaultRestDirectoryEndpoint`. But unlike the other endpoints the directory handling utilizes a different paradigm. The `FolderDAO` is used to keep an active record, i.e. a Java object constantly synced with the database entry.

## Folder

To start working with a folder the directory object and its root folder can be retrieved with:

```
UUID ehr = openEhrClient.ehrEndpoint().createEhr();
FolderDAO root = openEhrClient.folder(ehr, "");
```

With the `FolderDAO` at hand the following operations are available:

- Getting and setting the name
- Listing all sub folders
- Getting a sub folder, which will be created, if it not exists already
- Adding compositions to the folder, which includes committing the composition to the backend
- Finding of compositions in the folder structure, i.e. the matching EHR context

Together an example might be:

```
UUID ehr = openEhrClient.ehrEndpoint().createEhr();

FolderDAO root = openEhrClient.folder(ehr, "");

FolderDAO visit = root.getSubFolder("case1/visit1");

EhrbaseBloodPressureSimpleDeV0Composition bloodPressureSimpleDeV01 = TestData.
↳buildEhrbaseBloodPressureSimpleDeV0();
visit.addCompositionEntity(bloodPressureSimpleDeV01);

EhrbaseBloodPressureSimpleDeV0Composition bloodPressureSimpleDeV02 = TestData.
↳buildEhrbaseBloodPressureSimpleDeV0();
visit.addCompositionEntity(bloodPressureSimpleDeV02);
```

(continues on next page)

(continued from previous page)

```
List<EhrbaseBloodPressureSimpleDeV0Composition> actual = visit.  
↪find(EhrbaseBloodPressureSimpleDeV0Composition.class);  
assertThat(actual).size().isEqualTo(2);
```

## 4.2.2 Generator module

**Warning:** WIP

Reference documentation of the generator module.

### Usage

The generator can be used to create Java classes representing a given openEHR template.

After locally building the SDK with `mvn clean install` a generator .jar is available.

To generate and entity class from a template generally use:

```
java -jar generator-version.jar  
-h          show help  
-opt <arg>  path to opt file  
-out <arg>  path to output directory  
-package <arg> package name
```

In a custom use case the generation could look like:

```
java -jar generator/target/generator-$VERSION.jar -opt ../$PATH_TO_TEMPLATE/ehrbase_  
↪blood_pressure_simple.de.v0.opt -out ../$OUTPUT_PROJECT/src/main/java -package org.  
↪$OUTPUT_PACKAGES.opt
```

This section describes the load testing process used to test EHRBASE.

### 5.1 Testehr

Testehr is a Groovy script used to bomb any openEHR compliant server, following this flow:

- user provides parameters
  - ehRs: number of EHRs to be created
  - template: valid operational template
  - compositions: number of COMPOSITIONs to be committed
  - aql: valid AQL query, associated with the given template
  - scaleTemplates: optional, number of templates to use

Pseudo-code:

```
for (i in 1..scaleTemplates)
{
    ehRsCreated = []

    testTemplate = copyAndChangeId(template)

    if (!server.templateExists(testTemplate))
    {
        server.templateUpload(testTemplate)
    }

    for (j in 1..ehRs)
    {
        ehRsCreated << server.createEhr(...)
    }
}
```

(continues on next page)

(continued from previous page)

```

commitTime = getTime()
for (k in 1..compositions)
{
    compo = generateComposition(testTemplate)
    ehr = ehRsCreated.pick()
    server.commitComposition(ehr, compo)
}
commitTime = getTime() - commitTime // elapsed time

aqlTimes = []
for (n in 1.. repeatAql)
{
    aqlTime = getTime()
    server.runAql(aql)
    aqlTime = getTime() - aqlTime // elapsed time
    aqlTimes << aqlTime
}

[aqlTimeMax, aqlTimeMin, aqlTimeAvg] = calculateMaxMinAvg(aqlTimes)
}

```

For instance, if you provide these parameters:

- ehRs = 100
- compositions = 20
- scaleTemplates = 3
- repeatAql = 5

The script will do 3 loops over 3 different templates (it's the same template but has different ID), will create 100 EHRs, and commit 20 COMPOSITIONs, distributed between the 100 EHRs (the ehRsCreated.pick() is random). In general, you might want to provide compositions >> ehRs (much greater then). Then an AQL query will be executed 'repeatAql' times, and the max, min and avg execution times will be calculated.

## 5.2 Script execution

You need Gradle (<https://gradle.org/>) to be installed to build, package and run. This was tested with Gradle 6.4.1.

- \$ gradle clean // clean the build
- \$ gradle build // compiles
- \$ gradle fatJar // packages generating a .jar file with all the dependencies (standalone)
- \$ gradle run -args="-ehRs 100 -template src/main/resources/opts/LabResults1.opt -compositions 20 -aql src/main/resources/queries/LabResults1.json" // run the script

Note: gradle run will build the script.

If you built the fatJar, you can run it anywhere without gradle:

- \$ java -jar build/libs/load-testehr-all.jar -ehRs 10 -template src/main/resources/opts/LabResults1.opt -compositions 20 -aql src/main/resources/queries/LabResults1.json

This section gives information about the FHIR-Bridge

Please also see the [openEHR SDK Github repository](#).

## 6.1 Overview

The FHIR Bridge is a component designed as a broker between an HL7 FHIR client and an openEHR server. It contains several ad-hoc integrations for creating, searching and getting data using the FHIR formats in the front-end and the openEHR data structures in the backend. It helps to transform data in the FHIR format to openEHR compositions. It implements FHIR endpoints based on the HAPI FHIR implementation, validates incoming data, transforms data from FHIR to openEHR (based on manually created data mapping classes), stores the FHIR resources in an internal database (including a status information regarding the transformation) and also provides functions to get data back out from an openEHR data repository (GET and FHIR Search). For the COVID-19 platform, the focus lies on the integration of the GECCO data set, the German COVID-19 consensus data set.

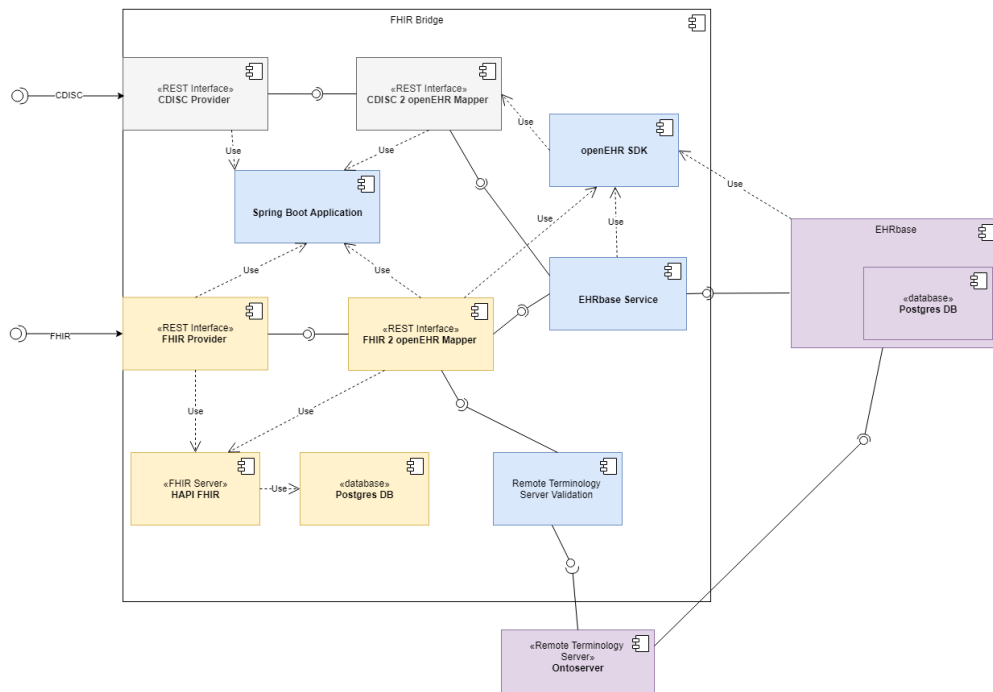
You can clone the project from <https://github.com/ehrbase/fhir-bridge>

### 6.1.1 Design decisions

1. Each integration is designed and developed ad-hoc, there is no generic solution to map FHIR into openEHR and viceversa.
2. On the FHIR side the type of resource is needed, and a correspondent profile. For some resources there might not be a profile available, which is not ideal since semantics for data mapping depend on specific FHIR profiles.
3. On the openEHR side, the Operational Templates are needed (OPT).
4. Data mappings are mainly done between a FHIR profile and an openEHR OPT.
5. To support the 'create' FHIR operation, a mapping should be designed and implemented to receive a FHIR resource, map its data to an openEHR COMPOSITION, and submit that COMPOSITION to the openEHR Server.

6. To support the 'search' FHIR operation, an openEHR query (AQL) should be designed to get the required data from the openEHR Server to map to FHIR resources to be retrieved. We chosen to return the COMPOSITION.uid as the FHIR resource id.
7. To support the 'read' FHIR operation, we use the COMPOSITION.uid retrieved as the FHIR resource id in the search, as the FHIR resource identifier to get the individual resource.

### 6.1.2 Architecture Overview



The FHIR bridge is mainly implemented over Spring Boot (application + configuration), using HAPI FHIR to process all FHIR related requests, expose API endpoints (via “resource providers”) and support all FHIR resource structures. Currently, the FHIR Bridge implements endpoints for HL7 FHIR (R4) in accordance with the German Corona Consensus Data Set (GECCO) and the profiles and resources of the Medical Informatics Initiative.

The controller forward the data to the mapping classes which mainly use the openEHR Software Development Kit (SDK) for conducting the mapping and transformation. The SDK has a code generator which allows to automatically generate Java classes from openEHR Templates (using the Operational Template Format (OPT)). These generated classes can be directly incorporated to build objects representing a human-friendly format for handling openEHR data. Moreover, the SDK encapsulates the REST calls of the official openEHR REST API and provides convenience functions that are implemented within the Mapper and the EHRbase Service components.

### 6.1.3 Extensibility

Above figure shows the ability of the FHIR Bridge to implement endpoints for further data formats. As the application is based on Spring Boot, it is possible to add new services using the same application and configuration. Such a mapping, for example for CDISC ODM or HL7 CDA. To achieve this, there is only the need to add a new Spring Rest Service with the expected parameters and a corresponding Mapper component. The other components can be re-used.

### 6.1.4 Testing

We prepared a set of HTTP requests to be able to test different services of the FHIR bridge. The requests work on Insomnia REST Client (<https://insomnia.rest/>).

Just install Insomnia, and import this file: [https://github.com/ehrbase/fhir-bridge/blob/develop/src/test/resources/Insomnia\\_2020-07-27.json](https://github.com/ehrbase/fhir-bridge/blob/develop/src/test/resources/Insomnia_2020-07-27.json)

Both, the openEHR server (EHRBASE) and FHIR bridge need to be running to be able to test.

## 6.2 Installation

- Install Java 11 (preferably <https://adoptopenjdk.net/>)

- Check for correct version:

```
java[c] -version
```

- for Linux: <https://adoptopenjdk.net/installation.html#linux-pkg>

```
sudo apt-get install adoptopenjdk-11-hotspot
sudo apt-get install adoptopenjdk-11-hotspot-jre
```

- Install Maven 3

- Instructions: <https://maven.apache.org/install.html>
- Check for correct version:

```
mvn --version
```

- for Linux:

```
# in (.bashrc)
# 2020-08-05 add maven to path
export PATH="/home/birgit/local/apache-maven-3.6.3/bin:$PATH"
$ source ~/.bashrc
```

- install git / git bash
- install docker

```
sudo apt-get install docker
```

- Install IntelliJ (or similar) if necessary

```
sudo snap install intellij-idea-community --classic
```

- Linux: don't forget to set path variables

```
# in (.bashrc)
# 2020-08-05 add maven to path
export PATH="/home/USERNAME/local/apache-maven-3.6.3/bin:$PATH"
export PATH="/home/USERNAME/Desktop/num/Installer/jdk-11.0.8+10/bin:$PATH"
export JAVA_HOME="/home/USERNAME/Desktop/num/Installer/jdk-11.0.8+10"
```

- Windows: set path (if not automatically done):

```
PATH="C:\Program Files\AdoptOpenJDK\jdk-11.0.7.10-hotspot\bin"
PATH="C:\Program Files\apache-maven-3.6.3-bin\apache-maven-3.6.3\bin"
JAVA_HOME="C:\Program Files\AdoptOpenJDK\jdk-11.0.7.10-hotspot\"
JAVA_TOOL_OPTIONS="-Dfile.encoding=UTF8"
```

## 6.3 Database for Audit Logs in FHIR Bridge

1. add config/application.yml to the root folder of fhir-bridge
2. with the following properties:

```
ehrbase:
  address:      localhost
  port:         8080
  path:         /ehrbase/rest/openehr/v1/
spring:
  datasource:
    url:         jdbc:postgresql://localhost:9999/fhir_bridge
    username:    fhir_bridge
    password:    fhir_bridge
```

3. run `docker run --name fhirdb -e POSTGRES_PASSWORD=fhir_bridge -e POSTGRES_USER=fhir_bridge -d -p 9999:5432 postgres`
4. run fhir-bridge

## 6.4 Do the mapping

If not further mentioned, the paths are relative to `fhir-bridge/src/main/java/org/ehrbase/fhirbridge` ## Prepare setup

### 6.4.1 Create new branch

Each change to the FHIR bridge should have a ticket created, explaining the change. Create a new feature branch with ticket number like: `feature/123_awesome_new_feature`, where 123 stands for the issue number

```
# optional: make a new checkout
git clone git@github.com:ehrbase/fhir-bridge.git
# default:
cd fhir-bridge
git checkout -b [BRANCH-NAME]
# At the first push:
git push -u origin [BRANCH-NAME]
# For later pushes:
git push
```

### 6.4.2 Start docker

Start docker, the `erhdb` and the `fhirdb` if they don't already run. For example (Birgit 2020-10-16)



```

docker rm ehrdb
docker rm ehrbase
docker run --name ehrdb --network ehrbase-net -e POSTGRES_PASSWORD=postgres -d -p 5432:5432 ehrbase/ehrbase-postgres:latest
docker run --name ehrbase --network ehrbase-net -d -p 8080:8080 -e DB_URL=jdbc:postgresql://ehrdbe:5432/ehrbase -e DB_USER=ehrbase -e DB_PASS=ehrbase -e AUTH_TYPE=none -e SYSTEM_ALLOW_TEMPLATE_OVERWRITE=true -e SERVER_NODENAME=local.ehrbase.org ehrbase/ehrbase:0.14.0
docker run -p 9999:5432 --name fhir-bridge -e POSTGRES_PASSWORD=fhir_bridge -e POSTGRES_USER=fhir_bridge -d postgres

```

### 6.4.3 Build

Build the current fhir-bridge

```
mvn clean install
```

### 6.4.4 IDE

Load project into development environment

- especially for eclipse: as a Maven project

#### Add external files

The following files must be copied into the respective target directories.

- FHIR data structure (XML format)
  - Target directory `fhir-bridge/src/main/resources/profiles`
  - Source <https://simplifier.net/ForschungsnetzCovid-19> (under Resources/Observation)
- FHIR observation sample file (JSON format)
  - Target directory `fhir-bridge/src/test/resources/Observation`
  - Source [https://simplifier.net/ForschungsnetzCovid-19/~resources?fhirVersion=R4&sortBy=RankScore\\_desc](https://simplifier.net/ForschungsnetzCovid-19/~resources?fhirVersion=R4&sortBy=RankScore_desc)
  - You can set a filter to the Examples
  - Caution:
    - \* the profile must be a profile from simplifier-Covid 19, fitting to the profile you just downloaded
    - \* the profile must have a working UUID (like `subject: { reference: Patient/07f602e0-579e-4fe3-95af-381728bf0d49 }`)
- Operational template (OPT format)
  - Target directory `fhir-bridge/src/main/resources/opt`
  - Source <http://88.198.146.13/ckm/projects/1246.152.26/resourcecentre> (GECCO Core)
  - Check the OPT in Pablos Tool: [toolkit.cabolabs.com](http://toolkit.cabolabs.com)
- Remember to check the downloaded files for content and syntax errors.
- Here you can check your syntax

- Check for xml: <https://xmllint.com/en>
- Check for json: <https://jsonlint.com/>

## Design the mapping

- Design the mapping by looking for 1..1 in the Fhir-profile and the fields in the OPT.
- Example <https://drive.google.com/file/d/1naGVhto6efWfF2sDoO86pYTnaUiiMq3J/view?usp=sharing>

## 6.4.5 Structure Definition (Enum)

- copy the Fhir-Url from `resources/profiles/[TEMPLATE].opt` to `fhir/Profile.java`

```
<StructureDefinition xmlns="http://hl7.org/fhir">
  <url value="https://www.netzwerk-universitaetsmedizin.de/fhir/StructureDefinition/
↪body-height" />
  <name value="BodyHeight" />
```

- if your FHIR-observation-sample-file contains an extension, add its Profile URL, too.
- Add an entry in `/config/util/OperationalTemplateData.java` with the name of the opt-file and the `template_id`-value

```
# Koerpergroesse.opt
[...]
  <template_id>
    <value>Körpergröße</value>
  </template_id>
```

```
HEART_RATE(" ", "Koerpergroesse.opt", "Körpergröße"),
```

## 6.4.6 Use the SDK generator to create new classes from the operational template

- (Windows example started from the path `:code:../../openEHR_SDK/generator/target`)

```
java
-jar generator-0.3.7.jar
-opt ../../fhir-bridge/src/main/resources/opt/Atemfrequenz.opt
-out ../../fhir-bridge/src/main/java
-package org.ehrbase.fhirbridge.opt
```

- Linux example (only resource-opt needs to be adapted)

```
java -jar ../../openEHR_SDK/generator/target/generator-0.3.7.jar -opt src/main/resources/
↪opt/KlinischeFrailty.opt -out src/main/java/ -package org.ehrbase.fhirbridge.opt
```

- Note: Ignore error message regarding missing language packages (temporary problem; TerminologyProvider).
- Refresh project in the development environment
- New classes and structures (example breathing rate)

```
$ opt/breathingfrequencycomposition
├── BreathrateComposition.java
├── BreathrateCompositionContainment.java
├── definition
│   ├── RespiratoryRateObservation.java
│   └── RespiratoryRateObservationContainment.java
└── Structure Definition (Enum)
```

- copy the Fhir-Url from resources/profiles/[TEMPLATE].opt to fhir/Profile.java

```
<StructureDefinition xmlns="http://hl7.org/fhir">
  <url value="https://www.netzwerk-universitaetsmedizin.de/fhir/StructureDefinition/
  ↳body-height" />
  <name value="BodyHeight" />
```

- if your FHIR-observation-sample-file contains an extension, add its Profile URL, too.
- Add an entry in /config/util/OperationalTemplateData.java with the name of the opt-file and the template\_id-value

```
# Koerpergroesse.opt
[...]
```

```
  <template_id>
    <value>Körpergröße</value>
  </template_id>
```

```
HEART_RATE("", "Koerpergroesse.opt", "Körpergröße"),
```

## 6.4.7 Use the SDK generator to create new classes from the operational template

- (Windows example started from the path :code:'. ./openEHR\_SDK/generator/target)

```
java
-jar generator-0.3.7.jar
-opt ../../fhir-bridge/src/main/resources/opt/Atemfrequenz.opt
-out ../../fhir-bridge/src/main/java
-package org.ehrbase.fhirbridge.opt
```

- Linux example (only resource-opt needs to be adapted)

```
java -jar ../openEHR_SDK/generator/target/generator-0.3.7.jar -opt src/main/resources/
↳opt/KlinischeFrailty.opt -out src/main/java/ -package org.ehrbase.fhirbridge.opt
```

- Note: Ignore error message regarding missing language packages (temporary problem; TerminologyProvider).
- Refresh project in the development environment
- New classes and structures (example breathing rate)

```
$ opt/breathingfrequencycomposition
├── BreathrateComposition.java
├── BreathrateCompositionContainment.java
├── definition
│   ├── RespiratoryRateObservation.java
│   └── RespiratoryRateObservationContainment.java
```

## 6.5 Flows

### 6.5.1 ‘Create’ operation internal flow

To process a ‘create resource’ request, the FHIR bridge receives the FHIR resource, already parsed to an object, on a ‘resource provider’ (like <https://github.com/ehrbase/fhir-bridge/blob/develop/src/main/java/org/ehrbase/fhirbridge/fhir/provider/ObservationResourceProvider.java>).

The method annotated with `@Create` will be the one that receives the resource object. For instance, in `ObservationResourceProvider`, the method will be:

```
@Create
@SuppressWarnings("unused")
public MethodOutcome createObservation(@ResourceParam Observation observation)
```

If the resource complies with a profile, a check for the profile is done: is the profile supported by the FHIR bridge?

Because of that step, each time a new mapping is added, the profile should also be added to FHIR bridge.

Then the resource is mapped to the correspondent COMPOSITION, depending on the FHIR profile, we determine which COMPOSITION should be created. Each type of COMPOSITION is determined by an OPT. This is why a profile is needed for each FHIR resource: if we don’t have a profile, a resource could really be mapped to different OPTs.

We created one mapping class for each FHIR profile/OPT pair, in which we implement the bidirectional mappings. For instance, for the ‘body temperature’ FHIR profile, we have a `FhirObservationTempOpenehrBodyTemperature` class that implements the FHIR -> openEHR mapping. So we can pass the observation object and get the correspondent openEHR COMPOSITION instance. That class implements the mappings designed at design time.

Once we have a COMPOSITION instance, the client library is used to commit the COMPOSITION to the openEHR Server. This operation returns the UID of the COMPOSITION created in the server.

Finally, if everything worked correctly, the FHIR client should get a successful response.

### 6.5.2 ‘Search’ operation internal flow

To process a ‘search resource’ request, the FHIR bridge receives the search request on a ‘resource provider’ (like <https://github.com/ehrbase/fhir-bridge/blob/develop/src/main/java/org/ehrbase/fhirbridge/fhir/provider/ObservationResourceProvider.java>).

The method annotated with `@Search` is the one doing the processing. For instance in `ObservationResourceProvider`, the method will be:

```
@Search
@SuppressWarnings("unused")
public List<Observation> getAllObservations(
    @OptionalParam(name="_profile") UriParam profile,
    @RequiredParam(name=Patient.SP_IDENTIFIER) TokenParam subjectId,
    @OptionalParam(name=Observation.SP_DATE) DateRangeParam dateRange,
    @OptionalParam(name=Observation.SP_VALUE_QUANTITY) QuantityParam qty,
    @OptionalParam(name="bodyTempOption") StringParam bodyTempOption
)
```

Note: the parameters for the search are defined arbitrarily, it depends on the resource type and the requirements of the client.

For observations, we allow to search by patient identifier, date range, value of the observation (only quantities for now). Because we have many profiles for the observation resource, we also need a profile parameter as a filter. The

‘bodyTempOption’ parameter is just a test to evaluate how different implementations of the search functionality work, it will be removed in the future.

More about search parameters: [https://hapifhir.io/hapi-fhir/docs/server\\_plain/rest\\_operations\\_search.html](https://hapifhir.io/hapi-fhir/docs/server_plain/rest_operations_search.html)

When the method receives the requests, it checks the profile parameter to choose the right handler for the search. For instance, if the profile is ‘<http://hl7.org/fhir/StructureDefinition/bodytemp>’, this method will be resolving the search:

```
List<Observation> processSearchBodyTemperature2(TokenParam subjectId, DateRangeParam
↳dateRange, QuantityParam qty)
```

What the search resolution does is:

1. creates an AQL query using the filters and search parameters, this is ad-hoc per FHIR profile and openEHR OPT.
2. executes the AQL query in EHRBASE to get the matching data (should be enough the data to fill the FHIR resources to be retrieved)
3. the openEHR query results are processed, mapping the openEHR data to the FHIR resource structure
4. each FHIR resource is stored in a list to be retrieved
5. the ‘resource provider’ receives the list and returns it
6. HAPI FHIR does the work of serializing that list to JSON, and that is what is retrieved to the FHIR client

### 6.5.3 ‘Read’ operation internal flow

To process a ‘read resource’ request, the FHIR bridge receives the get request on a ‘resource provider’ (like <https://github.com/ehrbase/fhir-bridge/blob/develop/src/main/java/org/ehrbase/fhirbridge/fhir/provider/ConditionResourceProvider.java>).

The method annotated with @Read is the one doing the processing. For instance in ConditionResourceProvider, the method will be:

```
@Read()
@SuppressWarnings("unused")
public Condition getConditionById(@IdParam IdType identifier)
```

The logic on this one is similar to the search but simpler, since there is only one resource to be retrieved, and the search params are just one: the resource identifier. So a similar AQL query like the one used for the search is used to get a COMPOSITION by identifier, we also check that complies with a specific OPT.

The query results are processed, mapping to a FHIR resource and returning that. HAPI FHIR serializes the resource to JSON and delivers that to the FHIR client.

If the query results are empty, the FHIR bridge returns a 404 Not Found.



---

Terminology Validation

---

This section provides information about external terminology validation using remote terminology services.

## 7.1 Introduction

Since the last version of the openEHR SDK, EHRbase now offers the validation of external terminologies in addition to local and openehr ones.

This feature is based on constraints defined in the openEHR templates and allows to validate every single coded elements in a composition.

The following example demonstrates how to define the constraint in order to validate a coded element based on the standard value set <http://hl7.org/fhir/ValueSet/surface> define in HL7 FHIR.

```
<attributes xsi:type="C_SINGLE_ATTRIBUTE">
  <rm_attribute_name>defining_code</rm_attribute_name>
  <existence>
    <lower_included>true</lower_included>
    <upper_included>true</upper_included>
    <lower_unbounded>false</lower_unbounded>
    <upper_unbounded>false</upper_unbounded>
    <lower>1</lower>
    <upper>1</upper>
  </existence>
  <children xsi:type="C_CODE_REFERENCE">
    <rm_type_name>CODE_PHRASE</rm_type_name>
    <occurrences>
      <lower_included>true</lower_included>
      <upper_included>true</upper_included>
      <lower_unbounded>false</lower_unbounded>
      <upper_unbounded>false</upper_unbounded>
      <lower>1</lower>
      <upper>1</upper>
    </occurrences>
  </children>
</attributes>
```

(continues on next page)

(continued from previous page)

```

<node_id/>
<referenceSetUri>terminology://fhir.hl7.org/ValueSet/$expand?url=http://hl7.
org/fhir/ValueSet/surface</referenceSetUri>
</children>
</attributes>

```

According to the constraint defined above, a valid composition should look like (using a valid code coming from <http://hl7.org/fhir/ValueSet/surface>):

```

"value": {
  "_type": "DV_CODED_TEXT",
  "value": "Buccal",
  "defining_code": {
    "_type": "CODE_PHRASE",
    "terminology_id": {
      "_type": "TERMINOLOGY_ID",
      "value": "http://hl7.org/fhir/ValueSet/surface"
    },
    "code_string": "B"
  }
}

```

Otherwise the submitted composition will be rejected by EHRbase indicating the error.

**Note:** The external terminology validation API available in the openEHR SDK module provides a generic mechanism that could be extended to support any remote terminology service.

However, please note that current implementation only supports FHIR terminology server.

## 7.2 Configuration

The following subsections provide information on the configuration of the external terminology validation in EHRbase.

### 7.2.1 Configuring EHRbase

EHRbase supports the following properties in order to properly configure the feature:

Key	Default Value	Description
validation.external-terminology.enabled	false	Whether to enable external terminology validation feature.
validation.external-terminology.fail-on-error	false	Indicates if validation should pass in case of connection error.
validation.external-terminology.provider.*		External terminology provider details.

The following `application.yml` illustrates how to configure the provider details:



```
# External Terminology Validation Properties
validation:
  external-terminology:
    enabled: true
    fail-on-error: true
    provider:
      fhir-server-1:
        type: fhir
        url: https://r4.ontoserver.csiro.au/fhir
#   fhir-server-2:
#     type: fhir
#     url: https://localhost:9876/fhir
```

- It is possible to register one or several providers.
- As mention above, `fhir` is the only type currently supported.
- The `url` property is the base URL of the terminology server.

## 7.2.2 Using provider with Two-Way SSL

If the remote terminology server requires to establish a communication channel using Two-Way SSL (Mutual Authentication), EHRBase can setup the SSL context used by HTTP client with the following configuration properties:

Key	Default Value	Description
<code>client.ssl.enabled</code>	<code>false</code>	Whether to enable SSL support.
<code>client.ssl.key-password</code>		Password used to access the key in the key store.
<code>client.ssl.key-store</code>		Path to the key store.
<code>client.ssl.key-store-password</code>		Password used to access the key store.
<code>client.ssl.key-store-type</code>		Type of the key store.
<code>client.ssl.trust-store</code>		Path to the trust store.
<code>client.ssl.trust-store-password</code>		Password used to access the trust store.
<code>client.ssl.trust-store-type</code>		Type of the trust store.

The following `application.yml` illustrates how to configure the SSL context:

```
# SSL Properties (used by Spring WebClient and Apache HTTP Client)
client:
  ssl:
    enabled: true
    key-password: MySecretPassword
    key-store: C:/ehrbase/config/keystore.p12
    key-store-password: Azerty123456
    key-store-type: pkcs12
    trust-store: C:/ehrbase/config/truststore.p12
    trust-store-password: Qwerty123456
    trust-store-type: pkcs12
```



## CHAPTER 8

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`