EtherCIS3

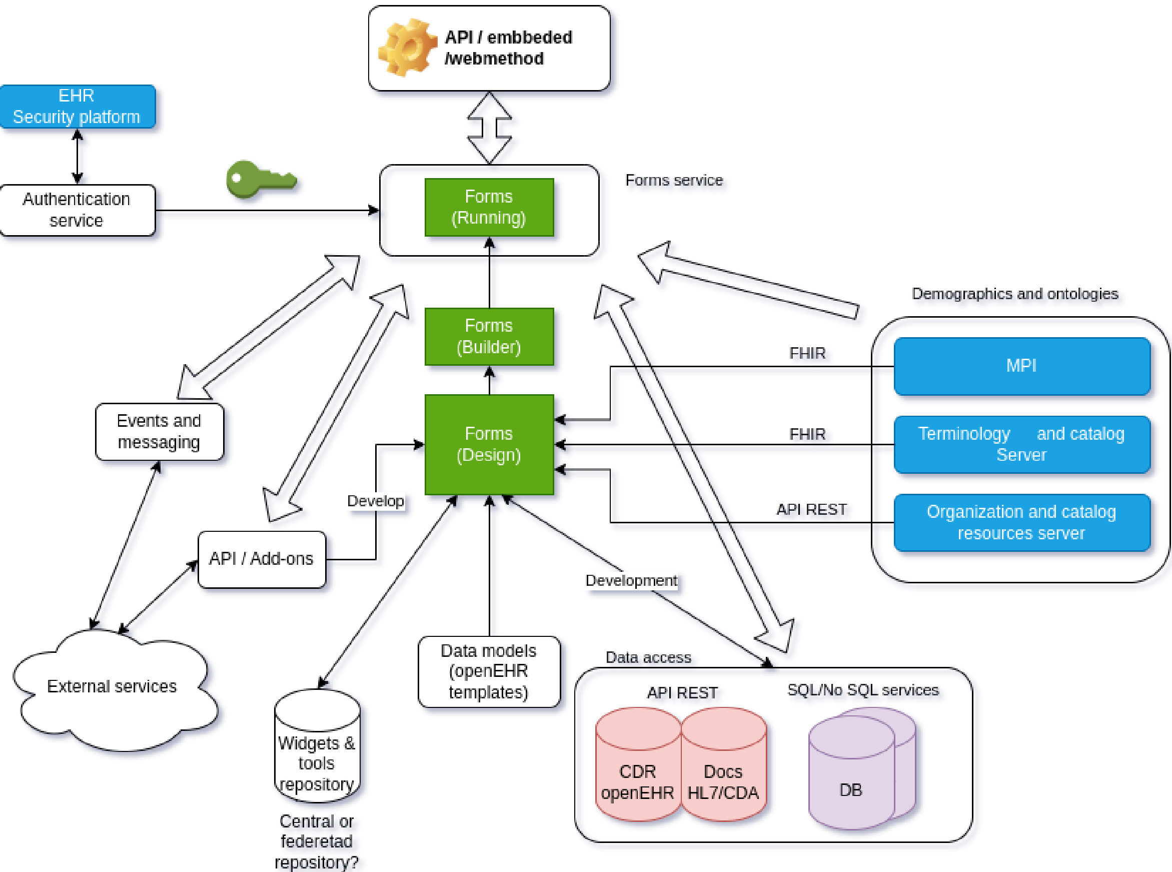
DRAFT 0.2 - C. Chevalley, November 2024

What Is It?

An open, extensible, scalable and secure clinical data platform enabling format agnostic data capture and semantic analytics.

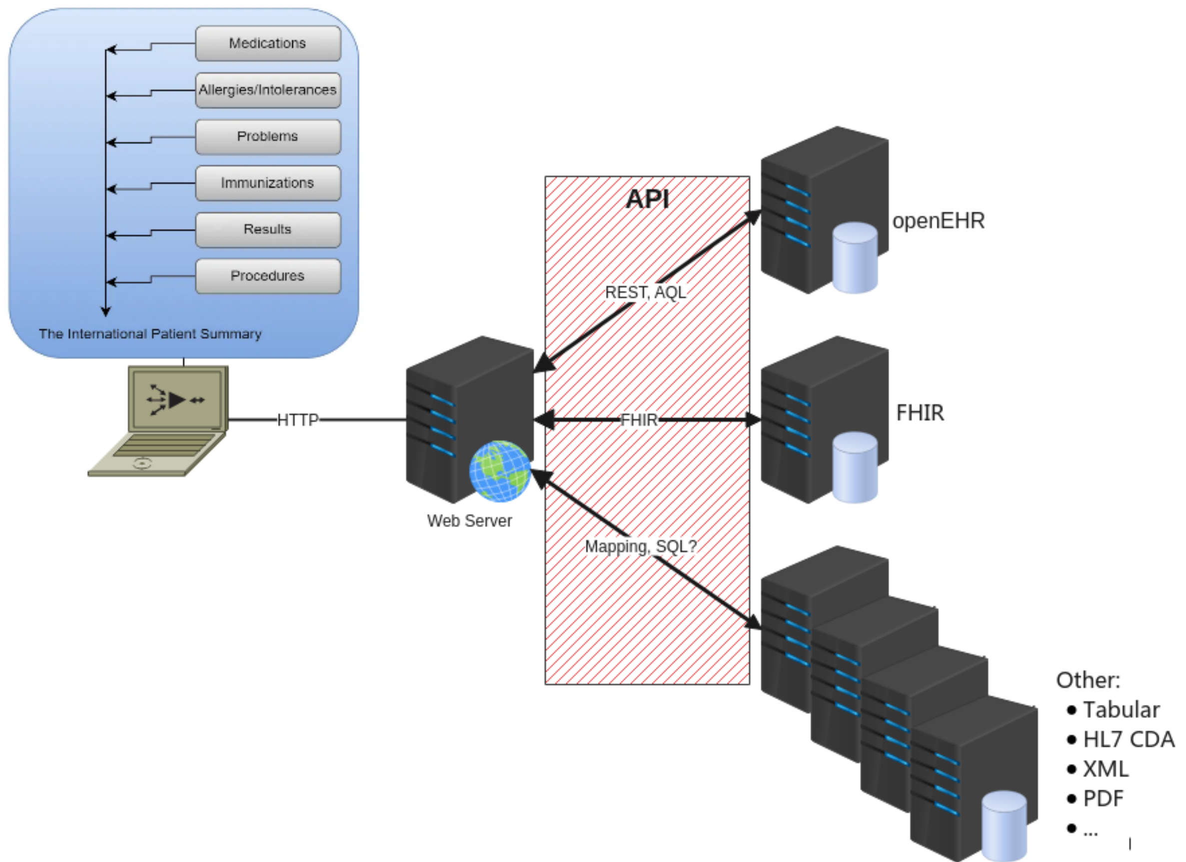
What Problem(s) Does It Solve?

The primary concern of this platform is to provide a framework to federate heterogeneous clinical data. For example, the following diagram is a common use case whenever attempting to mix various clinical data source (from https://discourse.openehr.org):

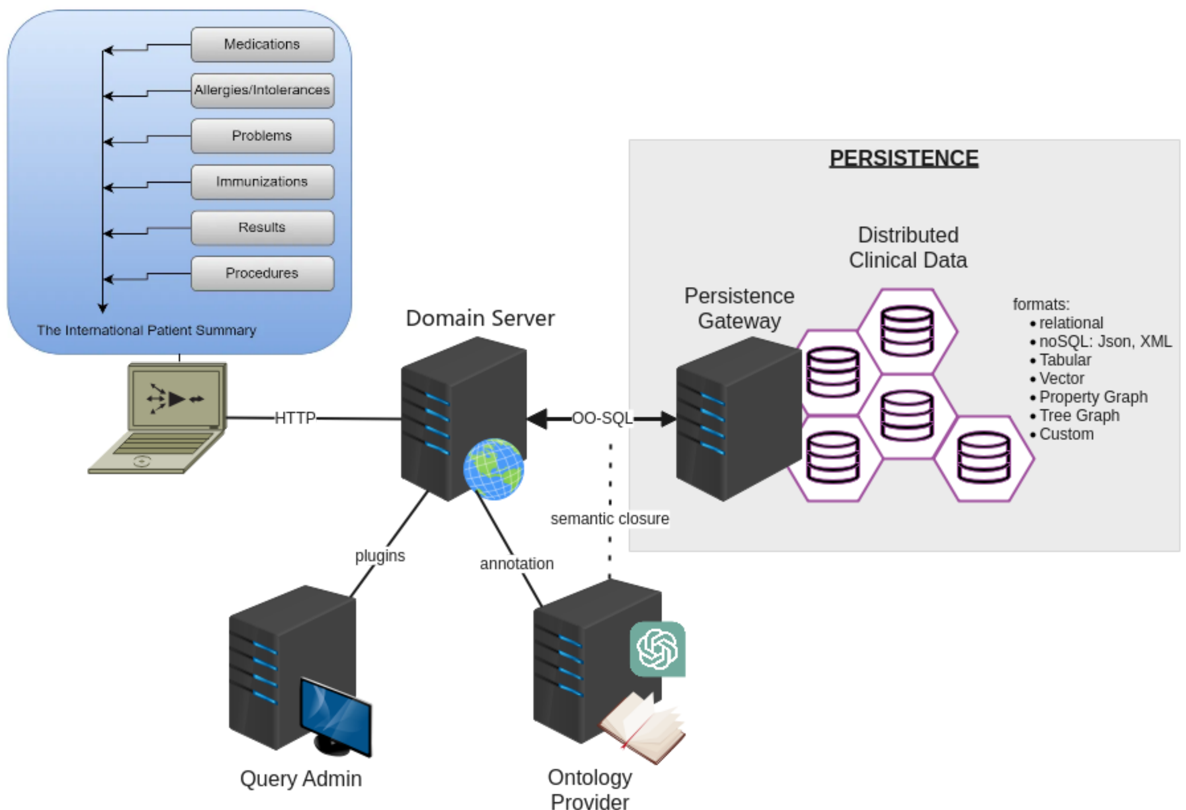


The issue is that datasources are accessed respectively to their specific client APIs. This is not efficient on large data sets, is difficult to deploy and maintain and is error prone, security/privacy is further a critical concern. Other common approaches to deal with heterogeneous data consists in "mapping" foreign formats to a well defined one, for example openEHR. There is an abundant literature about these attempts, but so far this is not convincing.

The API approach can be summarized as below (f.e. consolidating data to produce an "International Patient Summary":



We propose to replace the "API" layer by a common service able to manipulate data sources regardless of their format. Transactions are performed via a Domain Specific Language which is format agnostic.



The above approach easily supports cross platform querying, for example: "*We want to retrieve patient lists with clinical statuses by querying both FHIR (for demographic data) and OpenEHR systems*". Aside from the API/client issue, the security of such querying is problematic since it is subject to different patterns.

This model achieves:

- Single query pattern for all data sources (object oriented SQL)
- Data persistence is distributed and managed in a single consistent and secure data complex (see below for details)
- Semantics is supported by annotating legacy data with a corresponding terminology code (f.e. SNOMED-CT)
- Semantic annotations allow to associate transitive closure incrementally. Hence, queries are efficient on a semantic layer perspective
- Persistence modeling is fully managed and is used to generate and maintain the query context (DSL)
- Queries are built and maintained in a specific administrative process to ensure consistency and security

What Does This Approach Have That Is Superior to Other Initiatives?

Other initiatives are generally based on:

1. The client application consolidates results whenever accessing multiple data sources (or even the same data source due to API limits): this is a well known querying anti-pattern and results in lagging response time.

2. Foreign data sources are mapped to another format. Generally, this results in complex implementations (one or more mappings per format). This approach is known as "ETL", with the following issues (from *Data integration from traditional to big data: main features and comparisons of ETL approaches*):

    1. **Scalability issues** Traditional databases and processing systems may struggle to scale horizontally in Big Data environments, impacting efficiency in managing increasing data loads.
    2. **Performance constraints** Due to rising data volumes, traditional methods can lead to slower response times and reduced system performance.
    3. **Data variety challenges** Integrating and processing structured, semi-structured, and unstructured data forms is challenging with traditional methods.
    4. **Cost and complexity** Scaling traditional systems for Big Data can be expensive and complicated, involving hardware upgrades, software licensing, and specialized management skills.
    5. **Limited analytical capabilities** Traditional methods may lack advanced capabilities for extracting insights from Big Data, such as machine learning and predictive analytics.
3. A "limited" query framework. For example, openEHR's AQL is very limited in terms of its DML capabilities.

With this approach, whenever heterogeneous data are to be accessed, only one query language is to be learnt. The language is essentially an extension of SQL:2023 (**ISO/IEC 9075:2023**), wrapped into an Object Orientated DSL. Consequently, the query framework supports:

- a large set of information models: relational, NoSQL, property graph, vector (LLM), tree graph, tabular, GIS etc.
- a rich and extensible set of query operators and constructs (see SQL:2023 and postgresql documentation). NB the actual set of constructs is extended by jOOQ (multisets, temporal tables, implicit Joins etc.)
- an intuitive domain specific language (DSL) used to all interaction with the persistence layer
- a persistence model that is fully distributed and able to deal with large volume of data (distributed SQL). For large DB setting, a fully compatible platform is YugabyteDB. Citus can

also be used.

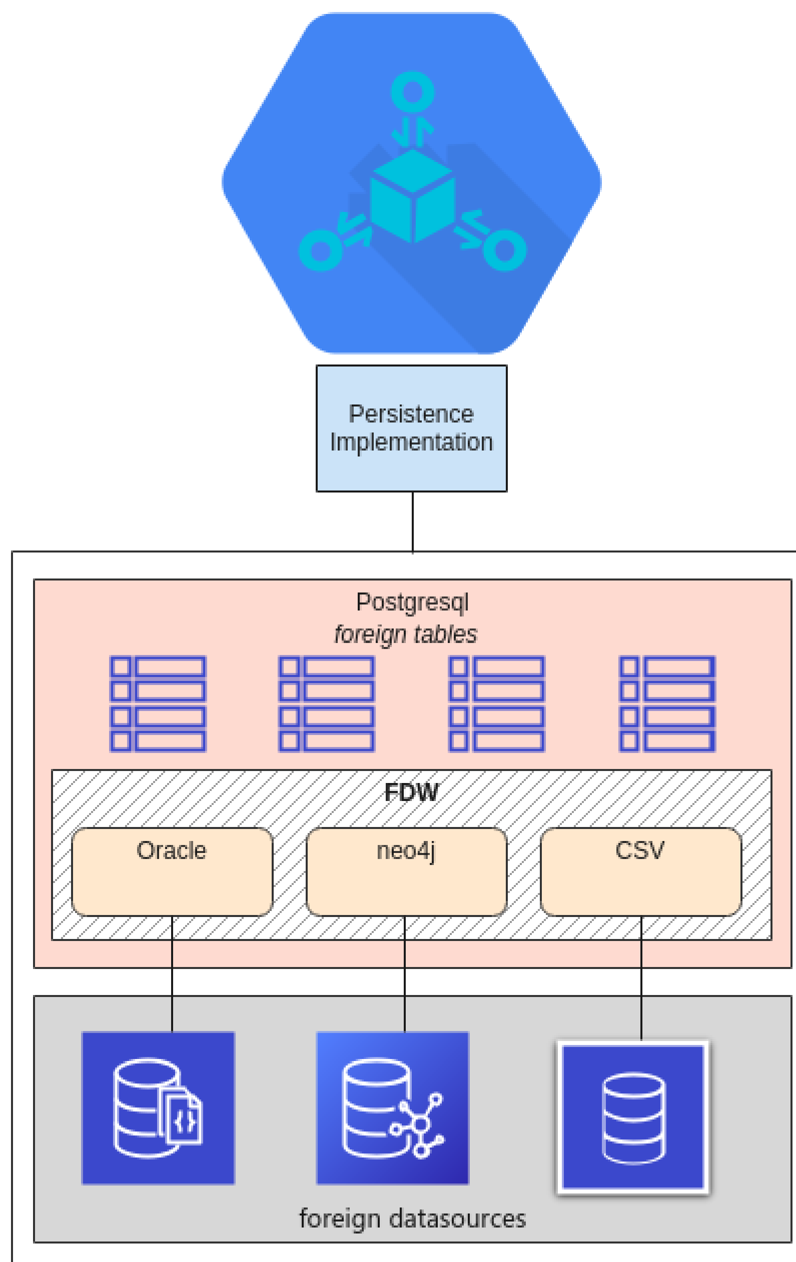
- a consistent security framework enforced at persistence level

On a platform standpoint, the system consists in "plugins" that are either standards or implemented by third parties as required. This allows to tailor a system according to actual requirements.

To what extent has any of this been proven to work in practice?

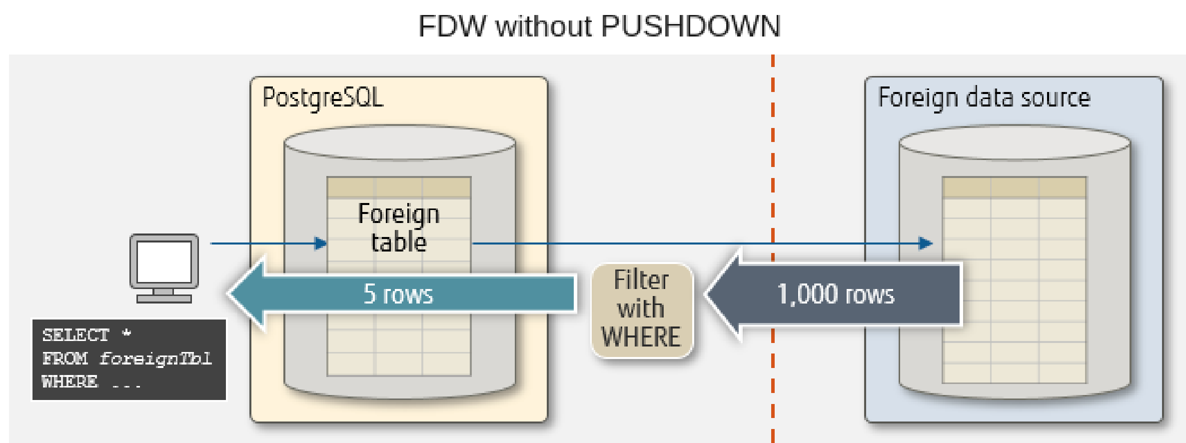
A seminal study illustrating this approach is Unified Access Layer with PostgreSQL FDW for Heterogeneous Databases. This approach is used in deployments such as Alibaba. The unified DB approach is also potentially used in many production sites and usually leverages on Postgresql FDW which we use extensively here.

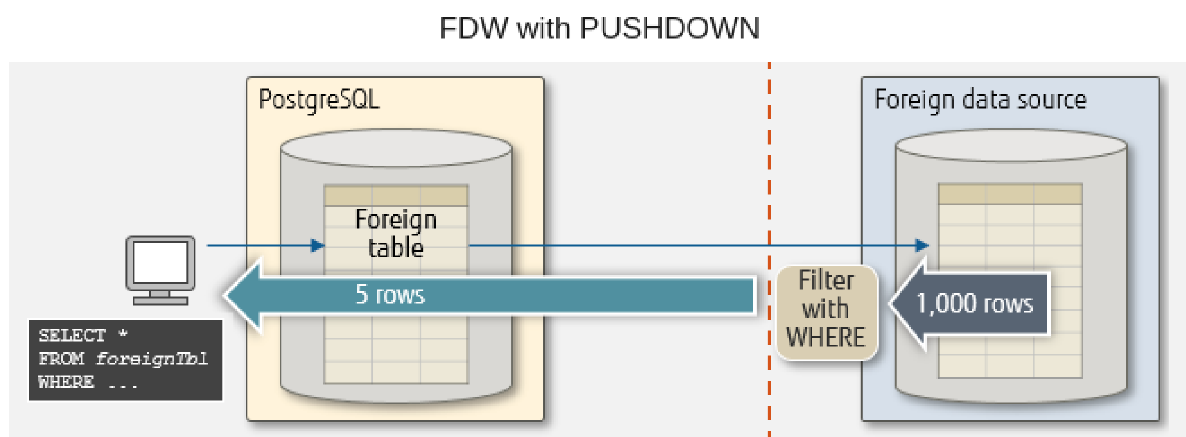
Then, the proposed architecture supports a persistence implementation depicted as below:



On a DSL standpoint, only the front-end data model with or without foreign table proxies is known. In other terms, the actual complexity of backend systems is hidden to the client applications. The foreign data sources can be actually complex, with multiple tables or structure. Queries shall be tested and optimized locally and result, for example, only in views (materialized or not) to the DB front-end.

Querying is performed as follows:





Since a FDW is a remote query processing agent, the postgresql front-end sends invocations and collects results. Results can be locally optimized to reduce network load as above. Queries can be locally monitored and optimized using their respective query planner and monitoring. The DB complex is a separate administrative function performs according to the deployment.

How Does this Differ from Current Approaches ?

Currently, the access to legacy data, is done via ad-hoc implementations including client code, sql queries etc.

For more advanced data format, a specialized API is used:

- AQL for openEHR
- FHIR search (although a SQL/FHIR interface exists)
- HL7 cql
- Cerner CCL (SQL?)
- ...

The main difference here is the encapsulation of foreign data into a common SQL based query framework. The drawback however is that Extraction and Loading is required (hence storage capacity planning is essential). In practical however, it is very rare that a legacy system is open for remote querying. In some instance it is not even feasible (f.e. data export from a GP system).

Architecture

Since the platform relies essentially on a database (more precisely on postgresql) that needs to perform at scale, support distributed system concepts, is extensible and reliable, several architectural patterns are used:

- Event Sourcing: the platform deals with event streams chained asynchronously.
- DDD or MDD (Domain/Model Driven Design): provides business logic abstraction in a bounded context (see Eric Evans 2003 - Domain Driven Design)
- Command Query Responsibility Segregation (CQRS) : essentially to avoid data storage bottlenecks

There are other architectural concerns such as multiple-coring and non-blocking interaction with the storage backend. This shall be dealt with at component level (e.g. outside the domain kernel since it is an infrastructural concern). Further other aspects including isolation (concurrency & distribution), failure (self-healing), state and deployment (k8s) shall be discussed in another document.

To apply the above patterns, the platform kernel is based on the Hexagonal architecture principles. The core (or domain kernel in this document) is responsible to instantiate plugins, parallelize routing request/response between them (as actors).

It is important to mention that the platform kernel inherently supports distribution, clustering, micro-services and containers (docker).

A DSL is used to query resources. This SQL dialect supports sophisticated querying, simplified path expressions to value points and parallel execution. The DSL is described in another document.

The platform is based on specialized plugins aiming at processing specialized functions according to the separation of concern pattern. A plugin implements specific use case (f.e. FHIR resource capture) and conforms to the shared nothing pattern thus limiting contentions between plugins and the core module. A plugin is loaded at startup or at runtime without downtime.

Queries are generated and invoked using a multi-levels modeling as described below (see "Querying Data" below).

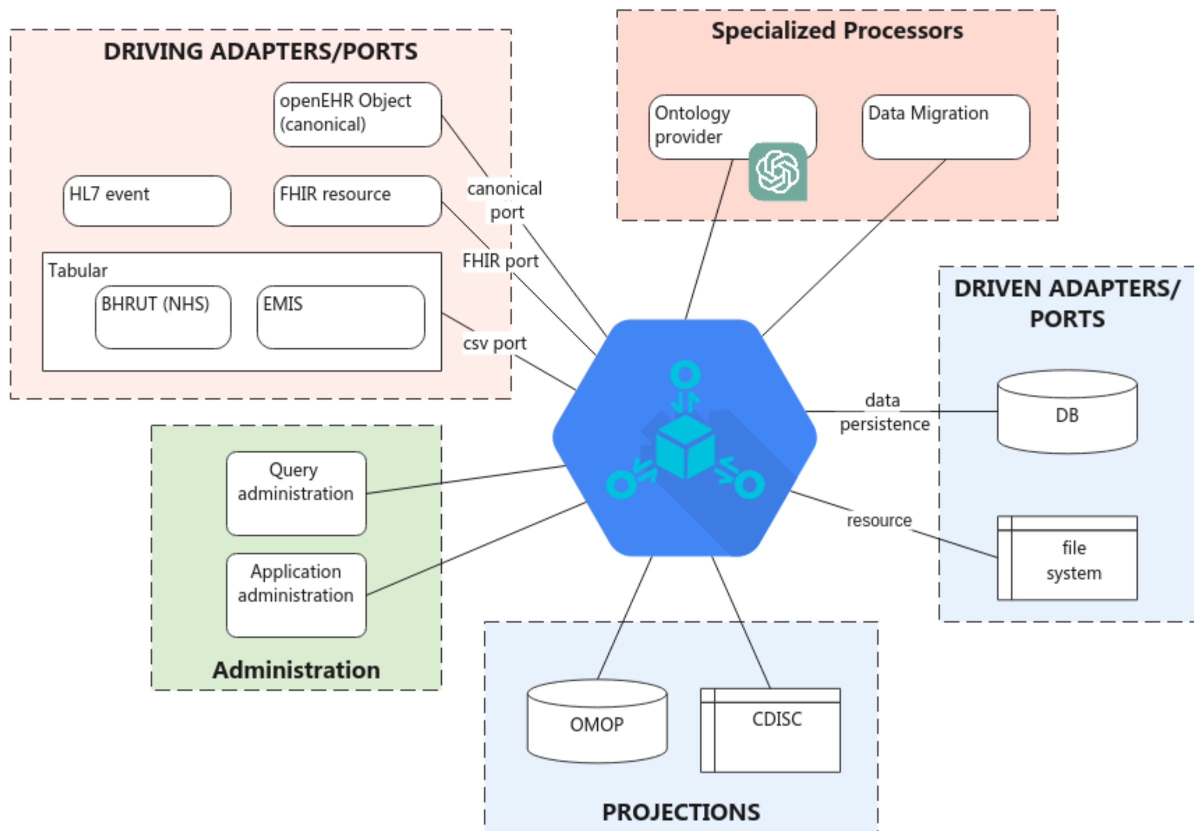
Dialogs with infrastructural resources (DB, filesystem, terminology server etc.) are likewise handled by specialized plugins. This enable application distribution in microservices.

DB management is always done by means of migrations that ensure the DSL and query expressions are always consistent with the data model (see "Data Storage" below)

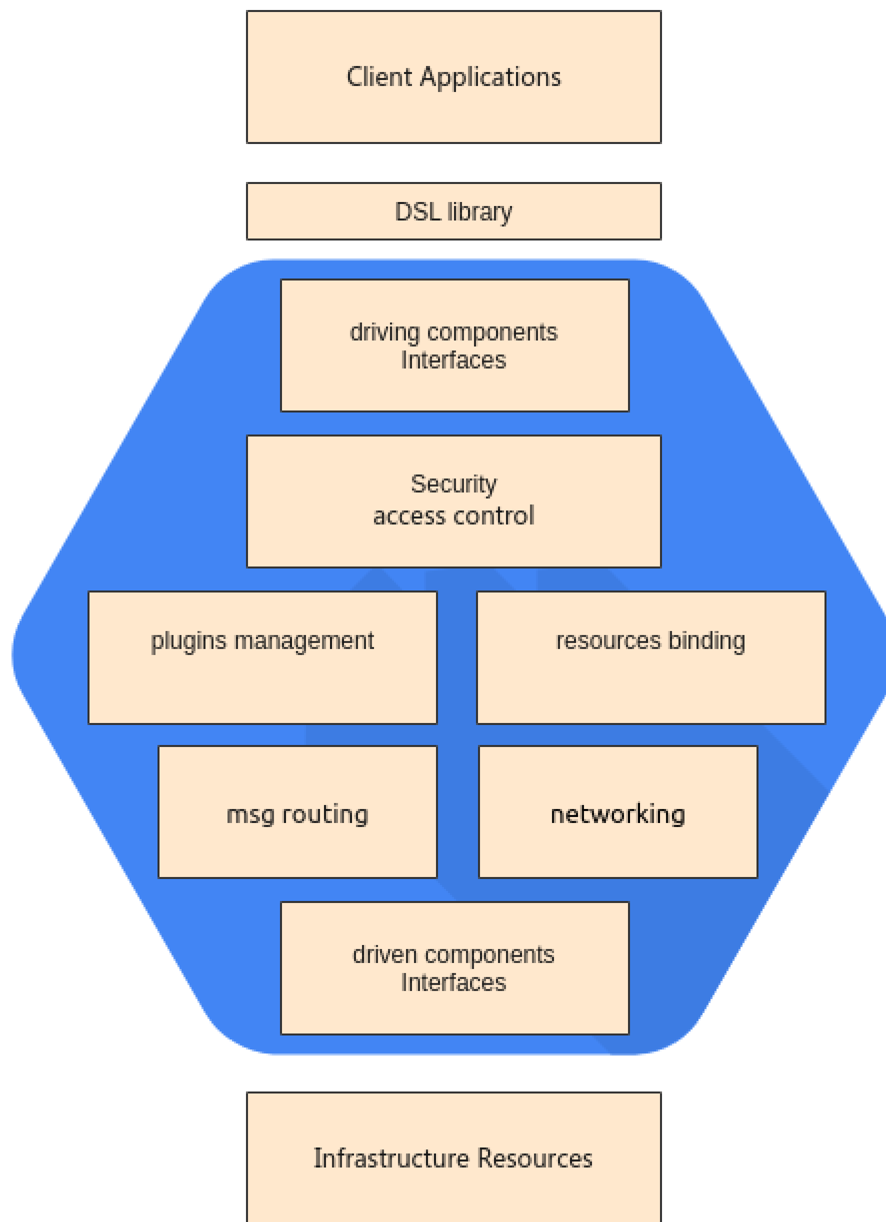
An modular administrative framework supports required utilities and front-ends (f.e. shell console, Web UI etc.).

A top level front-end aggregator combine system functions according to their respective implementation: HTTP, native UI, CLI etc. It is a facade to actual input plugins ("driving adapters" in hexa architecture jargon). It is based on the so-called "service aggregator pattern".

An hypothetical platform deployment can be schematized as below:



The domain kernel doesn't host any business logic. The business side is implemented by provided plugins. The domain kernel is responsible to manage the operation of plugins, route invocations between, enforce some security concerns and supports distribution using microservices.



See section "Domain Kernel" for details.

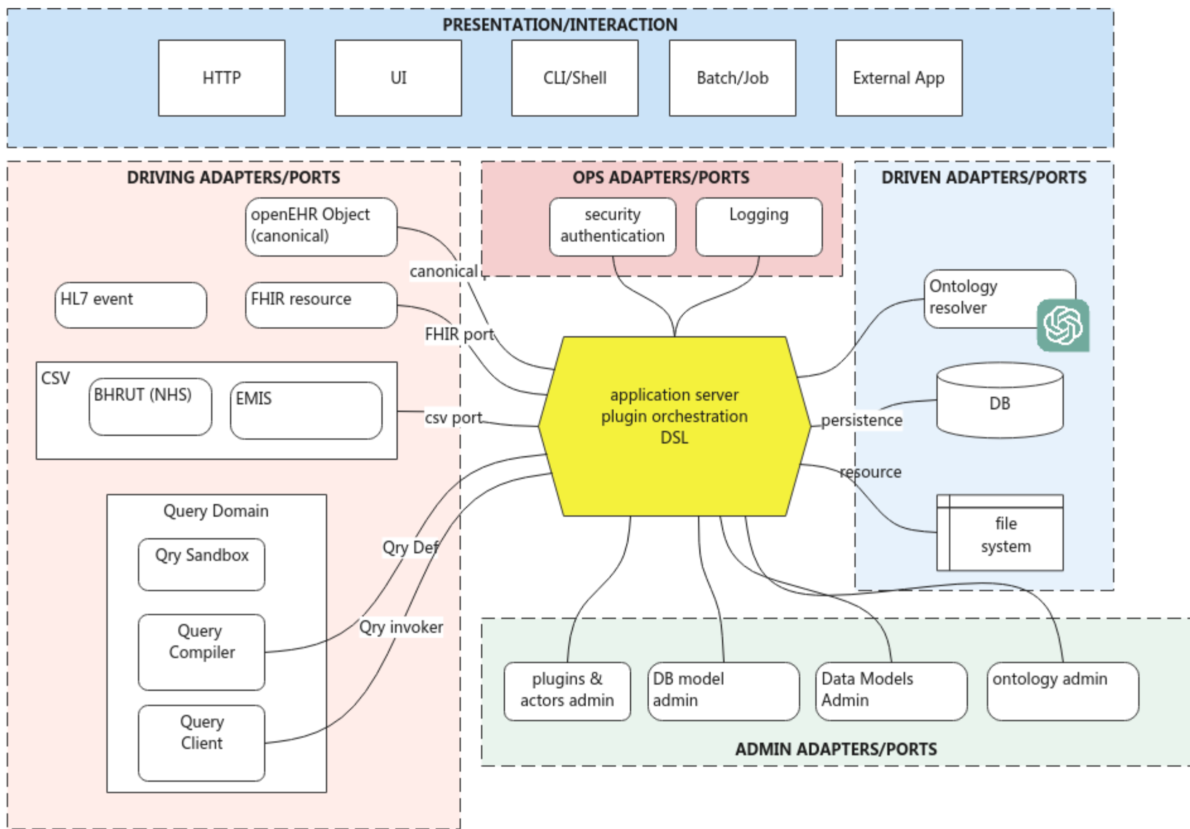
Synoptic

The core module component (yellow) is rather simple as its primary objectives consists of

1. running as an application process
2. maintaining DSL idiosyncrasies
3. Orchestrating plugins (instantiations)
4. Maintaining message invocations between them

Therefore, 3. and 4. implement the *wiring* of the application.

External infrastructural components used to access external features, interfaces with clients and technical components are deployed as required depending on system requirements.



Deployment

As mentioned, this architecture is not monolithic. It supports distributed deployments at multiple levels:

1. Distributed front-end services: HTTP gateway in particular
2. Microservices for plugins as services (f.e. openEHR data capure)
3. Distributed SQL. The DB does support YugabyteDB allowing large scale distributed deployments
4. Distributed common infrastructure for storage and network systems
5. DB architecture is essentially based on the notion of parallel querying (also known as parallel sharding)

Obviously, simple applications can be configured as stand alone monolith.

Stories

This section illustrates some of the above concepts.

Capturing Data

Assuming a simple data model combining openEHR as anchor and FHIR providing demographics:

1. The EHRs are initially created upon reception of FHIR demographic resources (patient in particular, this can be a batch job).
2. Created openEHR EHRs are stored in the backend and hold references to the original FHIR resource.
3. The matching FHIR resource is stored in the DB.
4. Afterward, openEHR compositions are either captured via a defined interface (REST, extraction from a legacy openEHR platform ...). The openEHR adapter performs validation and canonical json encoding as required.

5. The canonical json entry is then persisted by the respective storage adapter.

Querying Data

NB. direct SQL querying to a live DB is not allowed for security reasons. Access controls are enforced at DB engine level!

Designing Queries

- Queries are designed according to a defined use case (f.e. querying patient's vitals filtered on some criteria such as date range and other factors).
- Building the query is actually coding it using the provided core DSL.
- The coding is done within a mock-up sandbox which is an interactive environment (aka. IDE). It allows building and testing queries consistent with the data model and DSL. A sandbox is similar to a Jupyter notebook and used for coding only.
- A final test should be performed in a staging environment to detect regressions, conflicts and integration issues.
- To simplify complex queries design and maintenance, the design environment supports the concept of boilerplates. For example, templating standard UNION of queries to multiple heterogeneous resources, querying vitals to both FHIR and openEHR etc. The boilerplates should be identified, versioned, documented, tested and maintained in a repository for further reuse.

Inserting Queries in Production

1. The query packaging consists of the coded query as tested above and an envelope defining its attributes (name, version, arguments).
2. It is compiled as a plugin.
3. The query plugin is then released to production by the orchestrator which exposes it to the runtime environment.
4. The query plugin is now effectively callable by client modules.

Calling a Query

- On an client standpoint (UI, REST etc.) a query is invoked by passing its identifier and optional arguments.
- The orchestrator resolve the query identification, get the arguments and pass it to its respective implementation.
- A call can be either synchronous or asynchronous, with or without response.

Data Storage

- The design and architecture of the Persistence Module leverages the vast extensibility of postgresql (as well as its avatars to some extent such as yugabyteDB). The DB shall be deployed with support of distributed SQL for scalability and performance purpose. YugabyteDB, Citus support most of the requirements to fulfill these goals.

- Other natively supported DB include (please note that FDW might be required. DB modelization is a specialized task. DB capabilities may restrict the application domain, f.e. path expression is not supported by all of these engines!)
    - Aurora MySQL Edition
    - Aurora PostgreSQL Edition
    - Azure SQL Data Warehouse (Azure Synapse Analytics)
    - Azure SQL Database
    - BigQuery

    - CockroachDB
    - DB2 LUW
    - Derby
    - DuckDB
    - EXASOL
    - Firebird
    - H2
    - HANA
    - HSQLDB
    - Informix
    - MariaDB
    - Microsoft Access
    - MySQL
    - Oracle
    - PostgreSQL
    - Redshift
    - SQL Server
    - SQLite
    - SingleStore (MemSQL)
    - Snowflake
    - Sybase Adaptive Server Enterprise
    - Sybase SQL Anywhere
    - Teradata
    - Trino
    - Vertica
    - YugabyteDB
- Out of the box, the platform supports the openEHR Reference Model and FHIR resources.
- All database DDL (data definitions) are done from migrations. DB security policy shall prevent direct modification of the DB by a user.
- Migrations are designed and tested in a mock-up sandbox. Once complete, they are released to a staging system to test for regressions and integration issues. Whenever a major change occurs, the DSL will required to be upgraded accordingly (compiled). Dependent component(s) restarted (see below).

  NB. This procedure follows standard DB practice and shall be defined by a system logbook for security and operation compliance as required.

Unless a critical change occurs, altering the persistence model can be done without downtime as:

1. DB migrations can generally be done without DB interruption. The migration ensures that data integrity is preserved by design.
2. Whenever DSL needs upgrading, impacted plugin upgrades can be done incrementally thanks to the built-in redundancy mechanism in a multi-nodes (or microservices) context.

Actor Model

The processing (that is plugins invocation and orchestration) is based on a concurrent actor model. The main advantage of this approach is not only to support query execution parallelism but also to support load balancing, fallback strategies and operating concepts such as live upgrades.

Plugins are administered at runtime: monitored, started/stopped, deleted etc.

Security

Security requirements include client access controls (OAuth2), permissioning and most importantly DB security policies. One key technique used to enforce credentials on the data value access chain shall be *impersonation*; this shall be applied to both running plugins and DB accesses.

This shall be described in a separate document.

Domain Kernel

The domain kernel is the central component of the platform architecture. It consists of a Plugin Management Framework orchestrating the components used to deal with specific use case(s).

Introduction

Numerous enterprise grade platforms offer some kind of plugins or add-ons capability. A plugin framework is an essential feature to support true flexibility, extensibility, adaptability to unforeseen and future requirements. That is, while it is desirable to keep a core system minimalistic, users should be able to tailor the platform according to actual site requirement. Further, the framework should enable integrators to customize existing features and implement specific ones.

There are many platforms supporting plugins (we usually think of Eclipse and IntelliJ), and popular frameworks including OSGI (Open Service Gateway Initiative), spring-plugin, Pf4J/Kotlin. These will serve us as guidelines for the actual implementation, we will reuse existing materials as much as possible.

The objectives of the domain kernel are:

- Orchestrate all user traffic between "driving" and "driven" components in an hexagonal architecture context
- Manage client session wherever needed
- Manage permissions and access rights
- Handle plugin life cycles
- Monitor plugin activities
- Support audit trails and event capture
- Support clustering of plugins
- Allows defining multiple way of plugin method invocations

Plugin framework enables visibility and consistency of operating components while being able, to some extent, to perform runtime operation:

- update a plugin without halting the whole platform (live update),
- inserting, again to some extent, new feature without interruption,
- starting a plugin only on request (lazy loading).

Security is an integral part of the platform. It is the essential motivation of implementing a dedicated framework (as opposed to reusing an existing one such as Spring). The reason is to ensure that all request to the data and resources are done in a control manner and are auditable.

For example, a DK orchestrate the following plugins

- Client communication I/F: protobuf, RPC ...
- External data persistence and encoding: openEHR, FHIR, HL7v2 (ADT in particular), exotic format (HTML, PDF ...), SOAP
- Task Planning, CDS etc.
- Link to an adverse reaction server to detect possible allergies

- Terminology server
- AI used to perform (some) terminology resolution and closure

While openEHR specification also provides some guidance in the Platform Service Model. Some interesting aspects in particular regarding versioning will be retained in the rest of this document.

Functional Requirements

- Provide a plugin framework as core (domain) application server, to enable Third Party developers to extend the platform with their own features.
- A plugin shall be identified by the following attributes (see openEHR types for definitions)
    - id: String
    - name: DV_TEXT
    - version: SemVer
    - description: DV_TEXT
    - license: String
    - state: CREATED|DISABLED|STARTED|STOPPED
    - uid: HIER_OBJECT_ID (with versioning)
    - time_created: DV_DATE_TIME
    - time_updated: DV_DATE_TIME
    - dependencies: Array of String (IDs of required plugins prior starting this one)
    - load_mode: static or dynamic (on-request)
    - Compatible Platform Version, in particular, required platform components
- Plugins self-register to the platform (e.g. no XML or Yaml configuration). The domain app server maintains a plugin registry
- Plugin features are invoked via messages sent to extension points
- Plugins support various way of interactions including
    - Synchronous
    - Asynchronous
    - Publisher/Subscriber
- Plugins sequence loading is performed according to the defined dependencies using a self-organizing pattern (see Boto Bako, Andreas Borchert, Norbert Heidenbluth, Johannes Mayer "Plugin-Based Systems with Self-Organized Hierarchical Presentation" SERP 2006 about this concept)
- The plugin loader mechanism is responsible to ensure that plugin are legitimate and secure (see below)
- Plugins shall be managed at **runtime**, plugin administration allows to perform maintenance and/or other administrative operations without requiring a complete shutdown of the platform.
- A static plugin configuration shall be possible. That is the plugin configuration can be dealt with outside a running instance. This is required f.e. in case of operating critical failure.
- Whenever the platform is deployed as a cluster, a procedure shall be provided to ensure plugins consistency across all its nodes.
- If applicable, a plugin shall support specific configuration. F.e. a data layer I/F shall accept configuration such as DB URL, credentials to use etc. Ideally, plugin specific configuration asset shall be clearly identified (f.e. a specific YAML config file), a resource or a combination as required.
- Plugins shall be packaged into self-contained jar file with a relevant manifest.

- To be inserted into the platform, a plugin shall have a set of administrative attributes provided and verified including:
  - Contact information
  - credentials if applicable
  - domain if applicable
  - PGP signature
  - security access rules (ACLs)
- Operating Administration shall provide the following
  - list of known plugins
  - configure a plugin
  - install/update/delete/enable/disable according to each plugin dependencies
  - set a restore point (baselining the runtime environment to allow falling back to a previous stable version). See note about cluster deployment and consistency!
  - fallback to restore point (as above)
  - Platform + defined plugins (w/configuration) backup/restore. State preservation should be transparent to the user, but fully documented to help in restoring the platform to its latest state.

Non Functional Requirements

1. The Plugin Framework shall integrate seamlessly in the Kotlin/Ktor environment. That is, supports plugins written using Kotlin/Ktor conventions and use Kotlin/Ktor functionality such as dependency injection.
2. The deployment of plugins shall require minimal effort and, in particular, not needing specific configuration file (such as XML configuration as it was the case in EtherCIS and a major source of confusions).
3. The development of plugins shall be simple and fully documented in a step-by-step cookbook.

In Scope

- plugin framework as the central domain application module
- management API to perform plugin administration

Out Of Scope

- Plugin market place considering its associated risks do not outweigh possible benefits

Approach

Architecture

As discussed, the idea is to provide third parties with extension points in the platform.

Hence, the envisioned platform is a *pure* plugin framework, that is where all services are plugins (f.e. OSGI, Eclipse etc.). Further to enable various invocation means, most plugins are Actors. Actors are producers and consumers of messages orchestrated by the domain module. In other terms, this module is built on Reactive Streams concepts. The message brokerage between components is performed by the domain module (see below for details).

Plugin Design

To be validated for insertion in the platform, each plugin shall have a number of cross cutting concerns fulfilled:

- specific logging (slf4j)
- initialization function at startup
- shutdown function (optional)
- plugin resource configuration function that can also be called at runtime
- change active state (enabled/disabled)
- configure access control rules (if applicable)

Platform Core Plugins

A number of essential platform services allowing dependency resolutions:

- Authentication
- Core (DSL)

Other non essential, but expected services include:

- RM Persistence
- RM Validation
- REST controllers

These dependencies are loaded prior to any third-party plugins (that is, essential platform resources can't be overloaded).

Operation

At server instance startup, PMF resolves the order of plugins to be loaded as defined by each plugin dependencies and configuration (static vs dynamic).

Extension Points

Instead of paraphrasing existing materials, we suggest to follow the ideas (and potentially implementation) defined in pf4j extensions and sbp (for the class loader dealing with plugin dependencies)

The extension points include

- Web Interface
- Specialized services: ehr, composition, contribution, folder, terminology
- Persistence Layer (DAO)
- Authentication
- Auditing
- System (f.e. timer)

Servers Deployment in a Cluster

This is a summary of items to ensure for cluster consistency.

For **all nodes** in a server cluster:

- define the same plugin set
- all plugins are configured the same way
- all plugins have the same state
- plugin restore points are identical
- fall back to a restore point is performed on all nodes

Plugin Security

Plugin shall be provided by site organization for their own specific needs. Each plugin after thorough verification shall be sealed with a PGP signature. Market Place could be defined on this basis.

Whenever deployed in a cluster, the issue regarding secret sharing is critical when it comes to PGP signature verification.

Other Concerns

The issue about plugin consistency when deploying the platform as a cluster. We can envision that the platform shall be often deployed using Docker images in a Kubernetes environment. We have to figure out in more details how plugin maintenance is propagated in a cluster.

Custom class loader with plugin dependency management (as described in sbp). In particular, we have to ensure that a custom class loader doesn't negatively impact the platform class loading operation (f.e. Spring Boot classloader). Note that this issue is rather important as we have experienced some issue related to java class marshalling and XmlBeans.

Reactive Streams or Pipelines

The Message Oriented Middleware (MOM) is a distribution communication layer oriented toward exchanging messages between platform components or actors. In a first phase, MOM shall be using internal message passing, but in the future it shall be able to use message queue such as RabbitMQ or Kafka or other in order to implement distributed platforms deployed with microservices. For more details about Reactive Streams see https://reactivemanifesto.org/

Since the platform shall use mostly point-to-point (PtP) communication between actors, internal programmatic message passing shall be prioritized (NB. message queues are notorious for bottlenecks, deadlocks and significant resource consumption, hence they should be used only whenever unavoidable).

DSL: An Overview

The DSL is based on jOOQ with extensions to support path expressions and semantic resolutions to reference value points.

To ensure that DSL is consistent with the database schema, the following applies:

1. The DB is maintained with SQL scripts managed by FlyWay (called *migrations*)
2. DB schema is therefore baselined on the migrations.
3. At startup, the kernel checks that the DB schema matches the baseline
4. DSL is generated from the DB schema. To achieve this we use a DB test container which DB schema is configured from flyway migrations.
5. Any further DB alterations comply to the above rules. Whenever required, a migration also *migrate* DB content.

Populating an openEHR Composition

The following illustrates data value access paths.

Right now, using OpenEHR objects requires cryptic path expressions that are reported hard to use. As Knowledge Models (see openEHR CKM) are quite intelligible, we want the same expressiveness when building and querying the corresponding objects. For example, using the International Patient Summary (IPS) we can easily populate as follows:

```
with(composition){
set("language" to "th")
set("composer" to "PARTY_IDENTIFIED".set("name" to "Sylvia Blake"))

with(composition.at("context")) {
    set("start_time" to "DV_DATE_TIME".set("value" to "2018-02-01T09:00"))
    set("setting" to "DV_CODED_TEXT".set(
        "value" to "other care",
        "defining_code" to "CODE_PHRASE".set("phrase" to "openehr::238"))
       )
    set("health_care_facility" to "PARTY_IDENTIFIED".set("name" to "Hospital"))
}
//set some interesting value points
with(composition.items("@..'Allergy Intolerance'")){
    set("@..'Substance'" to "DV_TEXT".set("value" to "cat hair"))
    set("@..'Verification status'" to "DV_TEXT".set("value" to "Confirmed"))
}
with(composition.at("@..'Vital Signs'")){
    set("@E!..'Height/Length'" to
        "DV_QUANTITY".set("magnitude" to 170, "units" to "cm"))
    set("@..Weight" to
        "DV_QUANTITY".set("magnitude" to 100, "units" to "kg"))
    set("@..Respiration..Rate" to
        "DV_QUANTITY".set("magnitude" to 80, "units" to "/min"))
    set("@..'Pulse/Heart beat'..Rate" to
        "DV_QUANTITY".set("magnitude" to 120, "units" to "/min"))
    set("@..'Body temperature'..Temperature" to
        "DV_QUANTITY".set("magnitude" to 37, "units" to "Cel"))
    set("@..Systolic" to
        "DV_QUANTITY".set("magnitude" to 120, "units" to "mm[Hg]"))
    set("@..Diastolic" to
        "DV_QUANTITY".set("magnitude" to 90, "units" to "mm[Hg]"))
}
```

- Value points are referenced by their name using simplified path expressions (f.e. `@..Weight` to access a value point named 'Weight')
- Values are passed using either standard openEHR value types (DV_QUANTITY, DV_CODED_TEXT...) or plain values
- Quick note on syntax:
    - '$' is used for plain-old jsonpath.
    - '@' means that arguments should be interpreted as `name/value='...'`.
    - '@E!' indicates a named ELEMENT.

Querying

Similarly, DSL supports an extended and comprehensive object oriented SQL dialect to query DB objects (here openEHR)

```
val SUBJECT = COMPOSITION.ehr().STATUS
val CONTEXT = COMPOSITION.CONTENT.at("$.context") as Field<JSONB?>
val CONTENT = COMPOSITION.CONTENT

select(
    SUBJECT.at("@..'family group id'/id").`as`("patientID"),
    CONTEXT.at("$.start_time.value").`as`("dateCreated"),
```

```
    CONTENT.at("@..'Vital Signs'..Weight/magnitude").`as`("Weight"),
).
from(COMPOSITION).
where(
    CONTENT.at("@..'Vital Signs'..Weight").
        isGt("DV_QUANTITY".set("magnitude" to 100, "units" to "kg")))
    .and(
        SUBJECT.at("@..'family group id'/id").eq("55175056")
    ).
    orderBy(rmContext().at("$.start_time.value").desc()).
    limit(10).fetch().formatJson()
```

The result is something like:


```
{
  "fields": [
    {
      "name": "patientID",
      "type": "String"
    },
    {
      "name": "dateCreated",
      "type": "String"
    },
    {
      "name": "Weight",
      "type": "Double"
    }
  ],
  "records": [
    [
      "55175056",
      "2021-12-03T17:34:06.849379+01:00",
      110.10
    ]
  ]
}
```


As with composition creation, we use

1. simplified intuitive path expressions to reference value points
2. smart comparators using openEHR value types (f.e. comparing a DV_QUANTITY)
3. Implicit joins (`COMPOSITION.ehr()`) to simplify further SQL expressions

Indexing is based on the simplified path expression (which translate internally to a function)

Binding FHIR Resources

The platform consumes FHIR resources and store them into a Property Graph. One immediate benefit is to be able to link an openEHR EHR to its corresponding FHIR Patient resources holding Demographics. The FHIR resource can be further used to perform querying as needed.

For example, we want to create EHRs from FHIR Patient resources. We simply invoke:

```
//create an EHR corresponding to a FHIR patient resource
val ehrId = EhrFromFhir(dslContext, FhirFlavor.R5).createEHR(patientResource)!!

//link these two objects in a lookup table
RmFhir(dslContext).associate(nodeId, ehrId)
```

`nodeId` is the FHIR resource as stored into the property graph. We can now add some compositions to the openEHR EHR. We use openEHR EHRs as anchor points for querying to enable patient centricity.

A query could be:

```
val fhir = COMPOSITION.ehr().quadrmxref().node()

select(
    COMPOSITION.EHR_ID.`as`("ehrId"),
    fhir.PROPERTIES.at("$.resource.name.given").`as`("givenName"),
    fhir.PROPERTIES.at("$.resource.name.family").`as`("family"),
    COMPOSITION.CONTENT.at("@..Temperature/magnitude").
        cast(Double::class.java).`as`("Temperature"),
    COMPOSITION.CONTENT.at("@..Symptoms/value").`as`("Symptoms")
).from(
    fhir, COMPOSITION
).where(
    fhir.PROPERTIES.
        at("$..identifier..[?(@.system == \"https://terms.sil-th.org/id/th-
cid\")].value").
        eq("4733178998248")).
fetch().
formatJSON()
```

As above, jOOQ implicit joins are used, this allows to write simple correlated queries (see FROM clause)

We filter the selection based on a simulated patient identifier (here it is based on the ID card).

The result looks like the following:


```
{
  "fields": [
    {
      "name": "ehrId",
      "type": "UUID"
    },
    {
      "name": "givenName",
      "type": "JSONB"
    },
    {
      "name": "family",
      "type": "JSONB"
    },
    {
      "name": "Temperature",
      "type": "DOUBLE"
    },
```

```
    {
      "name": "Symptoms",
      "type": "TEXT"
    }
  ],
  "records": [
    [
      "cfd76ef3-1472-4949-9750-e3bdaefb4e76",
      [
        "toto"
      ],
      "manee",
      37.2,
      "Chills \/ rigor \/ shivering"
    ]
  ]
}
```

Querying Using Terminology

IMPORTANT: the following illustrates a rather naive approach to querying on terminology. However, since SNOMED-CT is both a terminology and an ontology (e.g. contexts), more work and experiments shall be conducted. One possibility would be f.e. to integrate SNOMED-CT ECL as a SQL dialect extension. This might be introduced as more sophisticated language capabilities are available (openCypher support in particular).

The platform embeds terminology transitive closures depending on the use case. Assuming SNOMED CT, a terminology based filter is specified by a predicate such as `isA`:

```
select(...)
from (...)
where
    COMPOSITION.CONTENT.at("@E!..isA{SCTID::15188001}")

    - OR -

    fhir.PROPERTIES.at("@E!..isA{SCTID::15188001}")
```

This can be translated as "an element which coded name matches an hearing loss", the following entries are possible:

- Impaired hearing
- HOH - Hard of hearing
- Hard of hearing
- Hearing impairment
- Difficulty hearing
- Hypoacusis
- HL - Hearing loss
- HI - Hearing impairment
- Hearing impaired
- Hearing loss (disorder)

This applies to any format where a value point can be identified by mean of a terminology code (openEHR, FHIR)

Notes

- a list of code can be supplied in the predicate expression.
- Other predicate constructs shall be provided at a later stage.
- in the above FHIR scenario, the filter checks on an Observation resource which code matches the argument similarly to an openEHR ELEMENT.

Combining Data Sources

Due to the rich query expression patterns, querying can be combined using SQL constructs such as:

- Common Table Expressions (CTE)
- Algebraic Operators (in particular UNION)
- Correlated Queries
- Sub queries
- etc.

For example, a single query can be a UNION of openEHR and FHIR queries returning the same fields