Mickaël Misbach Master Thesis

École Polytechnique Fédérale de Lausanne & The Hyve

Milestone 1 Report: Objective 1 System Design

Building a common and privacy-preserving front end for open-source clinical research platforms

The Hyve Supervisors:
Ward Weistra*
Dr. Bo Gao*

Master Student:
Mickaël Misbach*†

EPFL Supervisors:
Jean-Louis Raisaro†
Dr. Juan Troncoso-Pastoriza†

EPFL Professor:
Prof. Jean-Pierre Hubaux†

†: {*firstname.lastname*}*@epfl.ch*
∗: {*firstname*}*@thehyve.nl*

Contents

Chapter 1

Background & Related Work

1.1 Background

1.1.1 Clinical Research Systems

1.1.1.1 PIC-SURE API / IRCT

The IRCT implementation of the PIC-SURE API is the combination of four different components, that are open-source and available on [2]. They are implemented in Java and use standard technologies: web application archive (WAR) [6] for deployment, Hibernate [1] for data storage. First the Communication Layer (IRCT-CL) implements the RESTful service that is exposed and documented by [4]. The core component is the Application Programming Interface (IRCT-API), it handles the execution of queries and processing of results. An instance can be extended using the IRCT Extension (IRCT-EXT) that provides hooks and additional features without having to modify the core code. Finally the Resource Interface (IRCT-RI) connects to the different resources through connectors.

PIC-SURE API Overview PIC-SURE is resource based: each source of data (e.g. i2b2, tranSMART, etc.) is considered a resource. Each of these resources declare through their configuration the kind of clauses they support. The configuration is done through the database. Four types of clauses are supported: *select*, *where*, *process* and *join*, but here we are mainly making use of *select* and *where*.

The *select* clauses specify the data that the user of the API wants extracted from the database. The resources declare what kind of operations their *select* support, for example *AGGREGATE* can be defined to extract aggregated values. Then each of the operations (or the default operation when none is declared), declare the fields they support. For *AGGREGATE*, the resource could declare two fields:

- *FUNCTION*: what function to use among a list of permitted values, e.g. *COUNT*, *MIN*, *MAX*
- *DIMENSION*: the dimension along which to do to the aggregation, e.g. *PATIENT*

Each of the fields declare the type of value they take: either an enumerated value, or a data type such as *String* or *Integer* that has to specify the Java class implementing it. Example of two *select* clauses:

```
"select": [ {
    "operation": "AGGREGATE",
    "fields": {
        "FUNCTION": "COUNT",
        "DIMENSION": "PATIENT"
    }
}, {
    "field": {
        "pui": "/resource/study/Age/",
        "dataType": "INTEGER"
    }
} ]
```

The other main clause supported is *where* and is used to specified constraints on the data to be selected. Just like for *select* and the supported operations, the resource declares the predicates that *where* supports. The predicates can have fields, and fields have a type: this is just like *select*. Example:

```
"where": [ {
    "field": {
        "pui": "/resource/study/Age/",
        "dataType": "INTEGER"
    },
    "predicate": "CONSTRAIN_VALUE_NUMERIC",
    "fields": {
        "OPERATOR": ">=",
        "CONSTRAINT": "20"
    }
} ]
```

In order to construct queries made of the clauses previously described, the resources expose a tree of entities. Each of those entities, if they are queryable, declare a data type. Each of the predicates used for the *where* clauses also declare one or more supported data types: this is the mechanism used to know which entities support which predicates. In order to link the tree nodes together, each resource declares what kind of relationships between nodes exist. For example if the tree supports the relationship `CHILDREN`, a client can request all the children of a certain node.

1.1.2 OpenID Connect

OpenID Connect is a protocol enabling authentication and authorization using a third-party implementing the server-side of the protocol. It has several types of flow and many different options, but here we focus on what we are using in our solutions, i.e. the implicit flow.

The OIDC implicit flow as shown figure 1.1 follows the following steps:

1. The user requests the client identifier
2. The OIDC client sends its client identifier

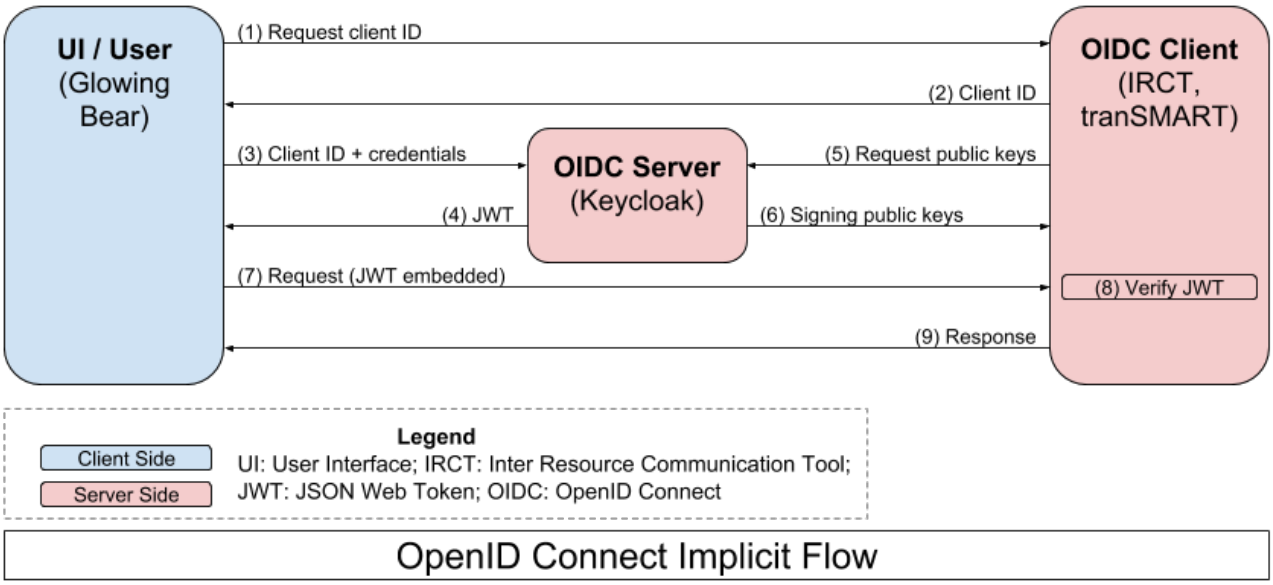

Figure 1.1: OpenID Connect Implicit Flow

3. The user performs the authentication with the OIDC server
4. Which gets him its token
5. In parallel to this the OIDC client requests the signing public keys used by the server
6. The OIDC server sends those keys to the OIDC client
7. The user makes use of the API exposed by the OIDC client, and embed in the HTTP headers the authenticating token
8. The client verify the validity of the token using the public signing keys of the server, and extract if necessary the JWT claims (authorizations)
9. The OIDC client can process the request and sends back the response to the user

1.1.2.1 JSON Web Token

A JWT is three distinct base64-encoded values separated by a dot: two JSON objects and a signature. The first is the header, which contains metadata about the JWT, such as the signing algorithm used and the identifier of the key used to perform the signing. Example:

```
{
  "alg": "RS256",
  "typ": "JWT",
  "kid": "eTFrdyrNxXLNHI7p0Ywybc7z1SBHTEcqWcMTybtdvQY"
}
```

The second is the payload of the JWT, and contains the identity of the authenticated user, its authorizations, and can contain any kind of custom fields set up by the administrator of the OIDC

server. Other standard fields include expiration time, client identifier, token issuer, etc. Those data are called the JWT claims. Example:


```
{
  "jti": "6fd4f480-05ce-471f-a4ef-f35cb6a8e0d0",
  "exp": 1523454086,
  "nbf": 0,
  "iat": 1523453186,
  "iss": "http://localhost:8081/auth/realms/master",
  "aud": "glowing-bear",
  "sub": "df110f80-fa32-4174-970d-e42a7b24ae9f",
  "typ": "Bearer",
  "azp": "glowing-bear",
  "nonce": "N0.285733398034069715234531986 56",
  "auth_time": 1523453186,
  "session_state": "74cd6393-9d24-4140-a87a-9e557f45499b",
  "acr": "1",
  "resource_access": {
    "account": {
      "roles": [
        "role1",
        "role2"
      ]
    }
  },
  "preferred_username": "test",
  "email": "test@test.com"
}
```


The last value is a signature generated by the OIDC server that serves two purposes: verify that the issuer was indeed the OIDC server, and protect the integrity of the token. Several algorithm can be used to generate this signature, here we use the algorithm *RS256*, i.e. the encryption of a *SHA256* hash using the asymmetric cryptosystem RSA.

Chapter 2

Interoperability Layer for Clinical Research Systems

In this chapter we are presenting the solution for the first objective of the project: setting up an interoperability layer for the main open-source clinical research systems, namely tranSMART, i2b2 and their respective derivations. The general design of the solution is first presented, then the steps that are required to implement this system are explained.

2.1 Design of the Interoperability Layer

This section describes in details the design of our solution. It aims to allow the cohort explorer UI Glowing Bear to support the following systems:

- i2b2
- SHRINE
- tranSMART 17.1 (REST API v2)

Because of the fundamental differences in the APIs, it is not possible to browse and query several resources at the same time, we thus restrict the scope to using one system at the time, while having compatibility with different ones. After being logged in, the user is able to choose from which resource do the queries. A resource is defined as an instance of one the supported systems.

Figure 2.1 shows the system diagram after completion of the objective 1. We can see that IRCT acts as a back end component for Glowing Bear, all its communications go through it using the PIC-SURE API. IRCT communicates with the native APIs of the back end systems.

2.1.1 General Workflow

The general workflow of the system starts in Glowing Bear. Then a specific IRCT resource is selected for the rest of the operations of this session. The user browses the tree of entities to construct query

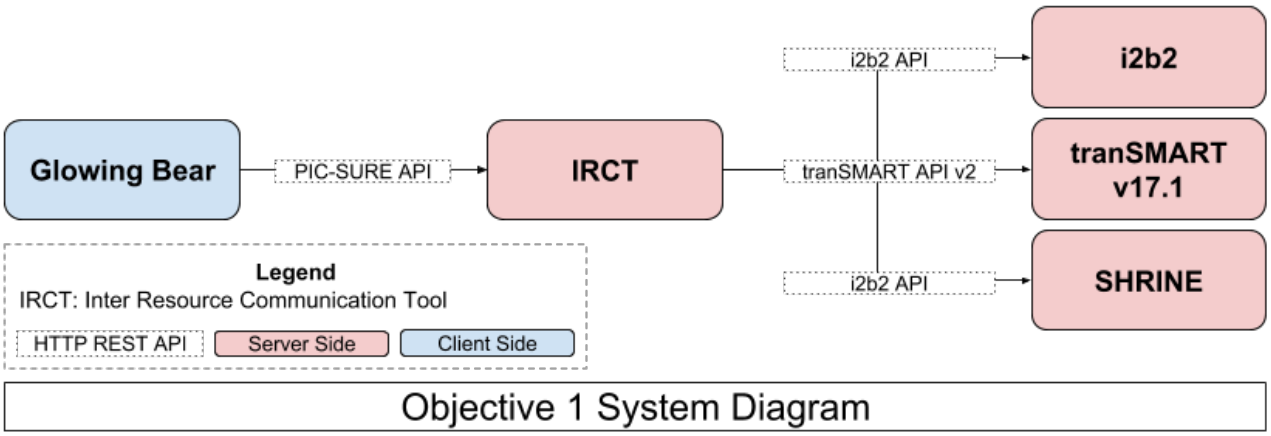

Figure 2.1: System diagram of objective 1: Interoperability layer for a common front end for clinical research platforms.

constraints using the PIC-SURE API. Each resource exposes this tree however it wants, but always within sticking to a common interface. When the constraints for selecting patients are established, the user chooses what data to export. During those last operations, the counts corresponding to the query are updated dynamically with background requests to IRCT, which itself does requests to the resources with their native API. Finally once the full query is constructed, the user executes the final query and is able to download the results, again using one common API.

2.1.2 Access Management

The authentication is delegated to a third-party using the OpenID Connect protocol [5], here the third-party we use is Keycloak but it could be swapped with any software implementing the same protocol.

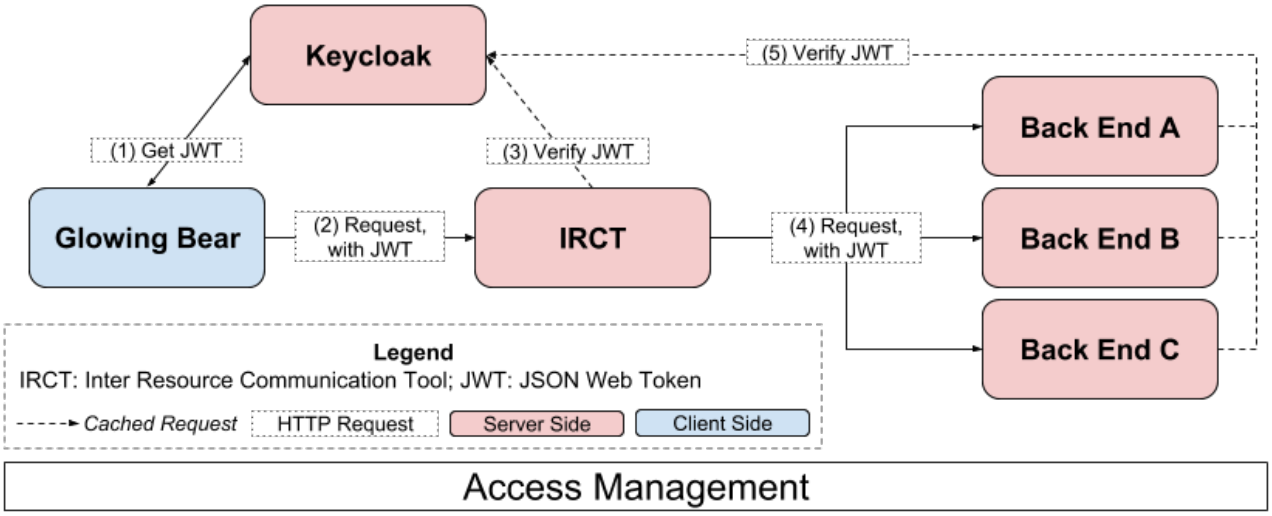

Figure 2.2: Diagram of the access management design.

Figure 2.2 shows the high-level picture of the access management in our solution, which revolves

around Keycloak. All components, i.e. IRCT and the back ends, verify the JSON Web Token embedded in the requests to authenticate the user and enforce its authorizations.

The steps are:

1. The client performs the authentication with Keycloak, with the client ID of the back end (which is set up to be the same as the IRCT resource name), and get a JSON Web Token containing the user authorizations
2. Glowing Bear makes HTTP requests to IRCT, with the JWT embedded in the headers.
3. IRCT verify the JWT using the public key of Keycloak. The JWT contains the identifier of the signing key, which IRCT uses to requests the corresponding public key to Keycloak (and is cached to minimize the requests) and then verify the signature.
4. The authenticated request is forwarded to the back end the user is using, the token remains the same.
5. The back end verify the JWT using the same method as IRCT, and is responsible to enforce its own authorizations, which are contained in the JWT.

Note that the JWT are verified twice in this design. The JWT contains the client identifier, which must be verified by the OIDC clients. IRCT do not have its own client identifier, but verify that the identifier matches the one of the back end. The back end has its own client identifier, and for the sake of consistency we enforce that the IRCT resource name is the same as the client identifier.

2.1.3 Glowing Bear Queries

One of the main implementation challenge is replacing in Glowing Bear the tranSMART API v2 by the PIC-SURE API. In Glowing Bear, all the operations the user undertakes have the objective of constructing a query to select data. In order to give an overview of the goal of the changes in Glowing Bear, see below an example of a query with the tranSMART API v2 and its equivalent using the PIC-SURE API. The query exports the gender for all patients aged from 20 to 25 years old in the *CATEGORICAL_VALUES* study.

tranSMART API v2 Example Query

```
POST /export/<id>/run
{
"constraint": {
"type": "and",
"args": [
  {
    "type": "and",
    "args": [ {
      "type": "subselection", "dimension": "patient",
      "constraint": {
        "type": "and",
        "args": [
          {
            "type": "and",
            "args": [
              { "type": "concept", "conceptCode": "CV:DEM:AGE" },
              { "type": "value", "valueType": "NUMERIC", "operator": ">=", "value": 20 },
              { "type": "value", "valueType": "NUMERIC", "operator": "<=", "value": 25 }
            ]
          },
          { "type": "study_name", "studyId": "CATEGORICAL_VALUES" }
        ]
    } } ]
  }, {
    "type": "or",
    "args": [
      {
        "type": "and",
        "args": [
          { "type": "concept", "conceptCode": "CV:DEM:SEX:F" },
          { "type": "study_name", "studyId": "CATEGORICAL_VALUES" }
        ]
      }, {
        "type": "and",
        "args": [
          { "type": "concept", "conceptCode": "CV:DEM:SEX:M" },
          { "type": "study_name", "studyId": "CATEGORICAL_VALUES" }
        ]
      } ]
  }
]
},
"elements":
  [{ "dataType": "clinical", "format": "TSV", "dataView": "default" }]
}
```

PIC-SURE API Equivalent Query

```
POST /queryService/runQuery
{
  "select": [ {
    "operation": "EXPORT",
    "fields": { "data": "clinical", "file": "TSV" }
  }, {
    "field": {
      "pui": "/<resource>/Public Studies/CATEGORICAL_VALUES/Demography/Gender/Male/",
      "dataType": "ENUM_VALUE"
    }
  }, {
    "field": {
      "pui": "/<resource>/Public Studies/CATEGORICAL_VALUES/Demography/Gender/Female/",
      "dataType": "ENUM_VALUE"
    }
  } ],
  "where": [
    {
      "field": {
        "pui": "/<resource>/Public Studies/CATEGORICAL_VALUES/Demography/Age/",
        "dataType": "INTEGER"
      },
      "predicate": "CONSTRAIN_VALUE_NUMERIC",
      "fields": { "OPERATOR": ">=", "CONSTRAINT": "20" }
    }, {
      "field": {
        "pui": "/<resource>/Public Studies/CATEGORICAL_VALUES/Demography/Age/",
        "dataType": "INTEGER"
      },
      "predicate": "CONSTRAIN_VALUE_NUMERIC",
      "fields": { "OPERATOR": "<=", "CONSTRAINT": "25" },
      "logicalOperator": "AND"
    }, {
      "field": {
        "pui": "/<resource>/Public Studies/CATEGORICAL_VALUES/",
        "dataType": "STUDY"
      },
      "predicate": "CONTAINS",
      "logicalOperator": "AND"
    }
  ],
  "alias": "My query"
}
```

We observe that the tranSMART API v2 uses a *subselection* of patients to specify the constraints, and then the additional constraints (linked together by an *OR*, and with the subselection by an *AND*) are used to select the data to be exported. To pursue the same objective, PIC-SURE uses *where*

clauses for specifying constraints and *select* clauses to select the data to be exported.

2.1.4 Back End Systems

IRCT communicates with all of the supported back end systems using their native APIs. The compatible resources expose their tree of concept of concepts, which contain the information on how to construct queries. They support the following basic query types, based on a specific set of constraints:

- Count queries: number of matching patients
- Patient set queries: identifiers on matching patients
- Data queries: any kind of medical data about the matching patients

The constraints used can be:

- Concept constraints: presence of a specific ontology item for a patient
- Value constraints: some numeric, text or date value satisfying some constraints

Some features specific to some back end systems are supported only when using this system. This is the case for systems implementing the tranSMART API v2:

- Constraints based on studies / clinical trials
- Constraints based on pedigree / relation type

2.2 Implementation of the Interoperability Layer

Here is described component-per-component the implementation of the solution presented in the last section. Each of the sections first explains what is the current status of the relevant components, and then what needs to be modified or implemented. First we describe the deployment of all the relevant components that need to be used in the system (refer to figure 2.1). Then we are going over all the modifications made to Glowing Bear, regarding the change of the API it uses. Finally we go over the modifications of the core of IRCT, and the implementation of all the resources it uses.

2.2.1 Deploying Original Components

First of all we are compiling, setting up and deploying all the building blocks of the solution: Glowing Bear, IRCT, tranSMART 17.1, i2b2, SHRINE. The exhaustive information about the deployments using Docker can be found in appendix B.1. These components require some servers that are deployed with Docker: WildFly (application server), PostgreSQL (database server), lighttpd (web server)

We are deploying Glowing Bear from the sources using the Angular command `ng serve`, which is practical for development. For production deployment it is deployed through the web server. The version used is branched off the development branch `table`, which is the most up to date branch at the time of the writing, and is regularly rebased on this same branch. It is configured to use the locally deployed instance of back end components.

Keycloak is deployed as an OpenID Connect server using its official Docker image that sets up a working instance. It uses the same PostgreSQL database that other deployments are using. The version used is 3.4.3. It is configured with a local user data source managed by itself, and one client for each back end that will use this instance.

IRCT is compiled from sources and deployed using Docker. The version used is TBD. It is configured to use one local instance of each type of supported back end systems: i2b2, tranSMART 17.1, SHRINE. The configuration is done through the local PostgreSQL database.

I2b2 is compiled from sources and deployed using Docker. The version used is TBD. The Spring configuration files and test data comes from the demo dataset that is provided with i2b2, and allows to have a demonstration running instance, done through the local PostgreSQL database.

tranSMART 17.1 is compiled from sources and deployed using Docker. The version used is TBD. The configuration is done through a grails configuration file, and some test data is loaded in the local PostgreSQL database used.

First step of the SHRINE deployment it to modify the existing i2b2 deployment. Its demo data (loaded in PostgreSQL) is duplicated three times, to replicate (with the appropriate configuration) three different instances of i2b2 (but served through the same web service). Then three instances of the SHRINE web services are deployed: this corresponds to a setting where the SHRINE network has three nodes. It is compiled from sources and deployed using Docker. The version used is TBD. It is configured through simple configuration files. The SHRINE instances use a deployment of the MySQL database server for their data. The SHRINE webclient is served by the web server.

2.2.2 Glowing Bear Modifications

Glowing Bear is pretty heavily modified, the idea being to make it a native PIC-SURE API client by replacing completely the tranSMART client API implementation. In general the API requests are modified through the calls in `ResourceService`, and the response through the models in `src/app/models/`. On a very high-level, the important steps of the queries as mapped as such:

- Step 1 (patients subselection with constraints): defining PIC-SURE *where* clauses, making count queries queries with the *select* clause
- Step 2 (selection of data to export): defining *select* clauses for data export, making count queries as well
- Step 3 (data export): executing the export query previously built

The organization of this section follows Glowing Bear typical workflow:

- Client initialization: loading and login
- Explore the tree of concepts
- Step 1: Construct a query or re-use a saved query
- Step 2: Select the data to export
- Save a constructed query
- Step 3: Export the data

Compatibility with tranSMART versions 17.1+ is later restored through the implementation of an IRCT resource interface (see 2.2.3.2.2). The reasoning behind this choice is that maintaining compatibility of two different but similar APIs in Glowing Bear would be possible, but complicated, which translates into additional efforts spent on the implementation, and later on the maintenance of the code: this would be sub-optimal as these efforts are better spent elsewhere. The potential downside of this choice is a time delay for the requests as we are introducing an additional middle-component, these are formally measured in a later chapter.

2.2.2.1 Authentication & Authorization

IRCT uses OpenID Connect to authenticate and authorize the users of its client applications, while Glowing Bear originally implements the client side of the OAuth2 protocol for authorization with tranSMART. The modification to Glowing Bear is the migration from the authorization protocol OAuth2 [**oauth2**] to the authentication and authorization protocol OpenID Connect [**openidconnect**]. Since OpenID Connect is a layer on top of OAuth2, the modifications to migrate the client code from OAuth2 to OpenID Connect are not significant, and are made by making use of the *angular-auth-oidc-client* library [**angular-auth-oidc-client**]. The flow remains the same: Glowing Bear obtains a JSON Web Token (JWT) from the OpenID Connect provider, which will then be embedded in the header of the HTTP requests to IRCT. These modifications are made by replacing the `EndPointService` by `AuthenticationService`, which implements the logic of the authentication. Additionally a small component `AutoLoginComponent` redirects to the login page of the OIDC server if the authentication token is not valid or not present.

2.2.2.2 IRCT Resources Support

2.2.2.2.1 Resources List After the user has successfully logged in, Glowing Bear requests the list of IRCT resources available and their definitions. Then the user chooses from the list which resource to use for the current session: it is possible to use only one resource at a time. The selected resource and its definition are kept in the new `IRCTResourceService` service that keeps and processes all the information about the resource returned by PIC-SURE. Among other things, this service is used to make the connection between tree data types and possible predicates that can be applied on them. The method `ResourceService.getResources()` is added to retrieve the PIC-SURE resources with the following API call:

```
GET /resourceService/resources
```

Response:

```
[
  {
    "id": 1,
    "name": "resource name (e.g. i2b2-local)",
    "implementation": "resource type (which implementation is used, e.g. i2b2XML)",
    "relationships": [ supported relationships between tree nodes, e.g. CHILD ],
    "logicaloperators": [ "AND", "OR", "NOT" ],
    "predicates": [ supported predicates and their properties, e.g.: {
        "predicateName": "predicate name (e.g. CONTAINS)",
        "displayName": "displayed name (e.g. Contains)",
        "description": "description",
        "default": true if this predicate should be selected by default,
        "fields": [ {
            "name": "field name (e.g. By Encounter)",
            "path": "field code (to be used in query, e.g. ENCOUNTER)",
            "description": "By Encounter",
            "required": true,
            "dataTypes": [ data type of the value field(s) of this predicate ],
            "permittedValues": [ permitted values, if it categorical ]
          } ],
        "dataTypes": [ data types to which this predicate applies (e.g. STRING) ],
        "paths": [ paths to which this predicate applies (empty for all) ]
      } ],
    "selectOperations": [ supported operations for select (e.g. AGGREGATE) ],
    "selectFields": [ supported fields for the operations, e.g. COUNT for AGGREGATE ],
    "joins": [ ],
    "sorts": [ ],
    "processes": [ ],
    "visualization": [ ],
    "dataTypes": [ data types and their properties, e.g.: {
        "name": "name of the data type",
        "pattern": "regex to validate the value",
        "description": "description of the data type"
    } ]
  },
  ...
]
```

2.2.2.2.2 Specific Features `IRCTResourceService` provides methods that determine if some specific features are supported, which allows some features of Glowing Bear to be enabled or disabled:

- `supportsCounts()`: returns *true* if queries of type `SELECT AGGREGATE COUNT` are supported, which enables the live counts display in the UI

- `supportedCountsDimensions()`: if `supportsCounts()` is *true*, returns the countable dimensions (most commonly *Patients* and *Observations*).
- `supportsNestedClauses()`: returns *true* if the *where* clause predicates `CLAUSE_NEST` and `CLAUSE_UNNEST` allowing nesting are supported
- `supportsMinMax()` returns *true* if queries of type `SELECT AGGREGATE MIN/MAX` are supported
- `supportsStudies()` / `supportsClinicalTrials()`: returns *true* if queries based on those specific dimensions are supported
- `supportsPedigrees()`: returns *true* if constraints based on pedigrees are supported
- `supportsGetTreeWithDepth()`: returns *true* if the tree can be retrieved with a specified depth (if the resource declare a tree relationship as `CHILDREN-DEPTH-X`)
- `supportsDataExport()`: returns *true* if data export queries are supported

tranSMART Studies The tranSMART REST API v2 has several calls that are specific to studies, while the PIC-SURE API does not have a direct equivalent to this. However since the studies are completely embedded within the concept tree, i.e. a concept belongs to one study or it is cross-study, they can simply be abstracted into the tree of entities exposed by PIC-SURE: we create a `STUDY` data type for the tree entities (see 2.2.2.3). `TreeNodeService.isTreeNodeAStudy()` implements the recognition of the `STUDY` data type.

If the resource supports it, `ConstraintService.loadStudies()` loads the list of studies by calling `ResourceService.getStudies()`, which is modified to use the following PIC-SURE call:

```
POST /queryService/runQuery
{
  "select": [ {
    "operation": "AGGREGATE",
    "fields": {
        "FUNCTION": "values",
        "DIMENSION": "study"
    }
  } ],
  "alias": "get_studies"
}
```

2.2.2.3 Concepts Tree

The API call in `ResourceService.getTreeNodes()` is modified according to the following:

- `depth` parameter is replaced by `relationship`: if the resource supports the retrieval of nodes with a depth more than one, then it will be achieved through a specific entities relationship API call `CHILDREN-DEPTH-X`;
- parameters `hasCounts` and `hasTags` are removed (these fields are always returned, although can be empty according to the resource implementation)

The API request becomes:

```
GET /resourceService/path/<resource>/<path>/?relationship=<relationship>
```

Response:

```
[
  {
    "pui": "/<resource>/<path>/",
    "name": "Concept internal name",
    "displayName": "Concept name",
    "description": "Concept description",
    "ontology": "Ontology Code",
    "ontologyId": "Concept ID in the ontology",
    "relationships": [ supported relationships ],
    "counts": {},
    "dataType": {
      "name": "Data type name",
      "pattern": "Validation Regex",
      "description": "Data type description"
    },
    "attributes": {
      "visualattributes": "Visual attributes",
      "customAttributeName": "customAttributeValue"
    }
  },
  ...
]
```

The `dataType` field allows to know which constraints defined in the resource can be applied, thus what options can be presented to the user when a query is constructed with the help of the `IRCTResourceService`. The `visualattributes` field allows to modify the appearance of the concept in the UI, for example if it's a folder containing concept, or a leaf node. Its presence is optional so if it is absent, the appearance will stay as it is by default, this behavior is defined in the `GbTreeNodesComponent` component. The other fields can easily be mapped to original fields in Glowing Bear.

In `TreeNodeService.loadTreeNodes()`, `loadTreeNext()` is called to iteratively load the nodes. `loadTreeNext()` needs to be modified to account for `ResourceService.getTreeNodes()` possibly not being able to load with a depth greater than one, meaning it must be called for every node of the tree. This is determined using `IRCTResourceService.supportsGetTreeWithDepth()`. The difference between a node and a leaf is made with the use of the `relationships` field: if the node supports the relationship `CHILD`, it is a node for which children can be requested.

`TreeNodeService.processTreeNodes()` and `processTreeNode()` process the received JSON to load it into the internal `treeNodes` array containing the tree in memory, they should be adapted to fit the PIC-SURE JSON format. This allows all the other parts of Glowing Bear that use the node to access the information needed.

2.2.2.4 Query Step 1: Constraints

The original Glowing Bear workflow for creating the constraints is the following:
`GbConstraintComponent.onDrop()` processes the node being dropped in the query construction panel in the UI by calling `ConstraintService.generateConstraintFromSelectedNode()` to generate the constraint based on the dropped node.
It uses `ConstraintService.generateConstraintFromConstraintObject()` to construct the individual constraint objects.

The constraints generated in the step 1 of the query corresponds to the *where* clauses of the PIC-SURE query: they define the criterion the resulting data must satisfy. The components based on `modules/gb-data-selection-module/constraint-component/GbConstraintComponent` and the models based on `models/constraint-models/Constraint` correspond to the different PIC-SURE predicates that Glowing Bear supports. Overall the Glowing Bear workflow stays the same at a high-level, but its implementation at the low-level undergoes significant changes to bring compatibility with PIC-SURE.

`ConstraintService` is initialized with `IRCTResourceService`: this allows the service to link the data types of the tree nodes with the constraints they support.
`ConstraintService.generateConstraintFromConstraintObject()` is the core method of the constraint generation, and is thus completely re-implemented and is renamed
`generateConstraintFromDataType()`. It is modified to take as input the data types coming from the PIC-SURE tree, and is using the `IRCTResourceService` to get the constraints corresponding to the data types, returning a `Constraint` object. `generateConstraintFromSelectedNode()` is modified to have two cases only: it is a queryable node, i.e. it has a data type, or not. If it has a data type it uses `generateConstraintFromDataType()` to get the constraint, if not it calls itself recursively with the children nodes.

`generateConstraintFromDataType()` supports the following PIC-SURE data types:

- Primitive
  - Numeric types: `INTEGER`, `LONG`, `FLOAT`, `DOUBLE`
  - Date types: `DATE`, `DATETIME`
  - String type: `STRING`
- Custom
  - Enumerated type: `ENUM_FIELD` and `ENUM_VALUE` (enumerated value exposed through the tree and not as a value)
  - Ontology concept type: `CONCEPT` (simple concept without value)
  - Study: `STUDY` (restrict to a specific study)
  - Pedigree: `PEDIGREE` (constraint based on relationship, e.g. parents of another selection of patients)

2.2.2.4.1 *where* Models The way the constraints are represented internally need modification, as the nature of the tranSMART constraints and the PIC-SURE *where* clauses are slightly different: they are more generic, but more importantly the supported predicate for each data type are known only at

the runtime. Constraints models in `src/app/models/constraint-models/` are now not based on the type of the constraint, but on the predicate used for the constraint. The interface `Constraint` remains, but the members `toQueryObjectWithSubselection()`, `toQueryObjectWithoutSubselection()` and `parent` are removed. The equivalent of the subselection in PIC-SURE would be the dimension, but (1) it is defined by the resources themselves, (2) it is integrated into the *select* clauses, which is handled in the second step as it is not considered a constraint; justifying the removal of those members. They also now contain data about themselves specified by the IRCT resource: predicates name, description, paths and data types it applies to, its fields (name, code, description, required flag, data type and permitted values).

`toQueryObject()` takes care of generating the JSON of the constraint: all the implementing methods now needs to generate the PIC-SURE *where* clauses. Example of two different *where* clauses that would be produced by the corresponding `toQueryObject()` methods:

```
{
    "field": {
        "pui": "/transmart/study1/Gender/Male/",
        "dataType": "ENUM_VALUE"
    },
    "predicate": "CONTAINS"
}

{
    "field": {
        "pui": "/transmart/study1/Age/",
        "dataType": "INTEGER"
    },
    "predicate": "CONSTRAIN_VALUE_NUMERIC",
    "fields": {
        "OPERATOR": "==",
        "CONSTRAINT": "5"
    }
}
```

2.2.2.4.2 Instantiating Constraints Because the predicates and the fields they have are known only at runtime, we create a service `ConstraintFactoryService` that handles the instantiating of properly initialized `Constraint` objects. A method `createConstraint()` taking as parameters the data type and and predicate returns the `Constraint` object. While some pre-defined predicates are supported by Glowing Bear (see list below), or even required for some features, not all can be supported. For this reason a generic `Constraint` is created, that allows Glowing Bear to handle unknown constraints that resources might declare.

The different constraint models with their associated predicate, that together support all the PIC-SURE data types listed before, are listed below:

- `ConceptConstraint`: predicate `CONSTRAINT_CONCEPT`, valid for `ENUM_FIELD`, `ENUM_VALUE`, `CONCEPT` and all primitive types
- `FieldConstraint`: predicate `CONSTRAINT_FIELD`, valid for arbitrary values of fields in dimensions

(e.g. data types `STUDY`, `TRIAL_VISIT`), and used to create constraint based on observation date

- `PedigreeConstraint`: predicate `CONSTRAINT_PEDIGREE`, valid for pedigree data types
- `PatientSetConstraint`: predicate `CONSTRAINT_PATIENT_SET`, valid for patient set (imported through the UI)

Note that `StudyConstraint` and `TrialVisitConstraint` are merged within the new `FieldConstraint`, similarly `GbStudyConstraintComponent` into the new `GbFieldConstraintComponent`. The trial-visit logic form `GbConceptConstraintComponent` is moved to `GbFieldConstraintComponent`. Note also that not all constraints are supported by all resources, the support is known by using `IRCTResourceService.supports*()` methods.

Concept Constraint This is a simple constraint based on the presence of a concept and possibly its associated value. A change from the original behavior is the way the values of the enumerated field are recuperated: this is done through the tree by looking at the data types, enumerated fields have a `ENUM_FIELD` types, and its possible values are children of the node with the type `ENUM_VALUE`. Another change concerns the aggregates values, see 2.2.2.5 for more information. The supported operators are `==`, `<=`, `>=`, `<`, `>`, `LIKE[exact]`, `LIKE[begin]`, `LIKE[end]` and `LIKE[contains]`.

Example of concept *where* clauses:

```
{
    "where": [ {
            "field": {
                "pui": "/transmart/study1/Age/",
                "dataType": "INTEGER"
            },
            "predicate": "CONSTRAIN_CONCEPT",
            "fields": { "OPERATOR": ">=", "VALUE":"20" }
        }, {
            "field": {
                "pui": "/transmart/study1/Age/",
                "dataType": "INTEGER"
            },
            "predicate": "CONSTRAIN_CONCEPT",
            "fields": { "OPERATOR": "<=", "VALUE":"25" },
            "logicalOperator": "AND"
        }, {
            "field": {
                "pui": "/transmart/study1/Gender/Male/",
                "dataType": "ENUM_VALUE"
            },
            "predicate": "CONSTRAIN_CONCEPT"
        } ]
}
```

Field Constraint This new constraint allows to restrict according to the value of field in some dimension of the data. It allows to query for a study, clinical-trial visit, observation date, and others. It is

implemented in the constraint model `FieldConstraint` and the component `GbFieldConstraintComponent`.

Example of field *where* clauses generated:

```
{
    "where": [ {
            "predicate": "CONSTRAIN_FIELD",
            "fields": {
                "DIMENSION": "study", "FIELD":"study_id",
                "OPERATOR": "==", "VALUE": "ORACLE_1000_PATIENT"
            }
        }, {
            "predicate": "CONSTRAIN_FIELD",
            "fields": {
                "DIMENSION": "trial_visit", "FIELD":"rel_time_num",
                "OPERATOR": "==", "VALUE": "2"
            },
            "logicalOperator": "AND"
        }, {
            "predicate": "CONSTRAIN_FIELD",
            "fields": {
                "DIMENSION": "observation", "FIELD":"start_date",
                "OPERATOR": "<=", "VALUE": "2010-02-22"
            },
            "logicalOperator": "AND"
        } ]
}
```

Pedigree Constraint Constraints based on pedigree are special in that they are based on other constraints (e.g. getting the parents of patients aged more than 50). We add for them a predicate `CONSTRAINT_PEDIGREE`, which has three fields:

- `TYPE`, required: list of permitted values which are either the type of pedigree (parents of, siblings of, etc.) or the indicator of the end of the pedigree constraints `CONSTRAINT_END` (see example after)
- `BIOLOGICAL`, optional, default to `BOTH`: either `YES`, `NO` or `BOTH`
- `SHARE_HOUSEHOLD`, optional, default to `BOTH`: either `YES`, `NO` or `BOTH`

Example of a pedigree *where* clause:

```
{
    "where": [
        {"predicate": "CONSTRAINT_PEDIGREE", "fields": {
          "TYPE": "PARENTS_OF", "BIOLOGICAL":"YES"
        } },
        <classic constraints>...,
```

```
        {"predicate": "CONSTRAINT_PEDIGREE", "fields": { "TYPE": "CONSTRAINT_END" } }
    ]
}
```

Similarly to other constraints, the logic construction is implemented into `GbPedigreeConstraintComponent` and the model representing such a constraint is `PedigreeConstraint`. It is only enabled if `IRCTResourceService.supportsPedigree()` determines so.

Patient Set Constraint Through the UI, the user of Glowing Bear can import a patient set and use it as a constraint, this handled by `GbPatientSetConstraintComponent` and stored into a model `PatientSetConstraint`, which are modified to adapt to the PIC-SURE format. It supports either an array of patient identifiers, or the identifier of a patient set.

Example of patient set *where* clauses:

```
{
    "where": [
        {"predicate": "CONSTRAINT_PATIENT_SET", "fields": { "PATIENT_SET_ID": "52" } },
        {"predicate": "CONSTRAINT_PATIENT_SET", "fields": {
          "PATIENT_IDS": "[2, 32, 96]"
        } }
    ]
}
```

2.2.2.4.3 Logical Operators Glowing Bear supports the definition of nested inclusion criteria, with the criteria belonging to the same group being linked by a logical operator *AND* or *OR*. PIC-SURE allows queries with several *where* clauses and lets each resource declares the logical operators it supports to link them. However the link between the clauses is flat, a clause defines its relationship with the previous clause: nested queries are not possible natively with PIC-SURE, but a workaround is presented below.

For resources that support nested queries, they declare the support for the `NESTING` predicate, having a field called `TYPE` with the permitted values `START` and `END`. By using these it is possible for resources to support nested queries, see the following example for the methodology: Consider the following nested query constructed in Glowing Bear:

```
((H AND B) OR (D AND S)) AND X AND (H OR F)
```

It would have the PIC-SURE query with the following *where* clauses:

```
{
    "where": [
        {"predicate": "NESTING", "fields": { "type": "START" } },
        {"predicate": "NESTING", "fields": { "type": "START" } },
        { H },
        { B, "logicalOperator": "AND" },
```

```
        {"predicate": "NESTING", "fields": { "type": "END" } },
        {"predicate": "NESTING", "fields": { "type": "START" }, "logicalOperator": "OR" },
        { D },
        { S, "logicalOperator": "AND" },
        {"predicate": "NESTING", "fields": { "type": "END" } },
        {"predicate": "NESTING", "fields": { "type": "END" } },
        { X, "logicalOperator": "AND" },
        {"predicate": "NESTING", "fields": { "type": "START" }, "logicalOperator": "AND" },
        { H },
        { F, "logicalOperator": "AND" },
        {"predicate": "NESTING", "fields": { "type": "END" } }
    ]
}
```

`CombinationConstraint`, `GbConstraintComponent` and `ConstraintService.generateConstraintFromConstraintObject()` are modified to...

- support this construction by storing an array of `Constraint`, appropriately setting their `logicalOperator` fields and adding the nesting *where* clauses;
- allowing or not the nesting of queries according to the resource capabilities.

Query nesting status is reflected in the UI by removing the `add criterion` boxes for the attributes with a level equal to or lower than 1 if it is not supported.

The method `ConstraintService.generateSelectionConstraint()` is what is used by other parts of the code to generate the constraints defined in the first step in a format that fits the API call. We rename it to `generateWhereAttribute()` to fit the PIC-SURE jargon, and modify the method accordingly. It returns a `CombinationConstraint` as described in the previous paragraph.

Negation The logical operator `NOT` is at the same level as the `AND` and `OR`. For this reason the resources declare all the following operators: (1) `AND`, (2) `OR`, (3) `NOT`, (4) `AND NOT`, (5) `OR NOT`. Which allows all kinds of queries.

2.2.2.4.4 User Interface When adding criterion at the step 1, the UI has to know what data type is the entity in order to know what input from the user is expected. This behavior is implemented in the `GbConstraintComponent` and its extending components. These components are modified or removed to fit all the modifications previously described, to arrive at a state where there is one component for each of the supported predicates described section 2.2.2.4.2. Additionally they use the information provided by `IRCTResourceService` to:

- offer to the user a list of the supported predicates to choose from,
- know what fields the predicate needs,
- enforce format of the fields input values with the regex or offer a dropdown list of permitted values,

- enforce required or optional fields,
- display aggregates about the concept (such as the min, max, etc.).

The parent `GbConstraintComponent` is modified to hold the data type of the dropped node, and allow for the switch between predicates (if multiple predicates are supported by the data type). Then the extending components, one for each predicate (with the addition of the one for unknown predicates), are used according to the chosen predicate.

2.2.2.5 Query Step 1: Aggregates

Glowing Bear does three kinds of aggregate queries: *counts*, *min / max* and *values*. *Counts* queries are made when the user presses *Update Counts* after modifying constraints (in step 1) or after modifying selected attributes for export (in step 2). *Min / max* and *values* queries are made when constructing constraints from the UI, to help the user the user by displaying some metadata. The API calls and calling methods made are modified to use PIC-SURE as described below. As support for aggregate *select* calls is not mandatory for the resources, if the resource does not support it the related features are disabled (more info in section 2.2.2.2.2).

`ResourceService.getCounts()`, `getStudies()` are merged into `getAggregate()`. This means that all the code calling those methods need to be adapted to fit the inputs and outputs of the new `getAggregate()`. This notably includes `GbConceptConstraintComponent.initializeConstraints()`, `QueryService.updateInclusionCounts()`, `updateExclusionCounts()`, `updateCounts_2()`, `updateConceptsAndStudiesForSubjectSet()`, `ConstraintService.loadStudies()`.

`getAggregate()` becomes more generic and accepts the following arguments:

- `aggregateType`: *count*, *min*, *max*, *values*
- `dimension`: among the dimensions declared by the resource
- `pui`: optionally a concept path if it is relevant for the query
- `dataType`: data type of the `pui`

Example of PIC-SURE aggregate *select* clauses:

```
POST /queryService/runQuery
{
  "select": [ {
    "field": { "pui": "/path/to/concept/", "dataType": "STRING" },
    "operation": "AGGREGATE",
    "fields": { "FUNCTION": "count",  "DIMENSION": "patient" }
  }, {
    "operation": "AGGREGATE",
    "fields": { "FUNCTION": "count", "DIMENSION": "observation" }
  }, {
    "field": { "pui": "/path/to/concept/", "dataType": "INTEGER" },
```

```
    "operation": "AGGREGATE",
    "fields": { "FUNCTION": "min" }
  }, {
    "field": { "pui": "/path/to/concept/", "dataType": "INTEGER" },
    "operation": "AGGREGATE",
    "fields": { "FUNCTION": "max" }
  }, {
    "operation": "AGGREGATE",
    "fields": { "FUNCTION": "values",  "DIMENSION": "study" }
  }  ],
  "where": [ <constraints> ],
  "alias": "<query_alias>"
}
```

2.2.2.6 Query Step 2: Data Selection

In the PIC-SURE paradigm, this step is about preparing the *select* statement of the query: the output for step 3 is the added *select* causes to the query. The changes in this part are minimal, as they are mainly about the modification of the resulting JSON containing the concepts the user wishes to get as a result, but they represent the same information.

Originally this information is stored in a `CombinationConstraint`, with individual constraints linked by a `OR`. Here for this purpose we are creating a new `PICSURESelectAttribute` model containing the different *select* clauses, which for each clause holds the path of the concept and the associated data type. For the sake of consistency the method `ConstraintService.generateProjectionConstraint()` is renamed to `generateSelectAttribute()` and returns a `PICSURESelectAttribute`. Some additional minor modifications are made in the methods using this, such as `QueryService.updateCounts_2()`, `updateExports()` and `GbExportComponent.runExportJob()`.

Example of *select* clauses generated during the second step:

```
"select": [ {
    "field": {
        "pui": "/<resource>/Public Studies/CATEGORICAL_VALUES/Demography/Age/",
        "dataType": "INTEGER"
    }
    }, {
    "field": {
        "pui": "/<resource>/Public Studies/CATEGORICAL_VALUES/Demography/Gender/Male/",
        "dataType": "ENUM_VALUE"
    }
    }, {
    "field": {
        "pui": "/<resource>/Public Studies/CATEGORICAL_VALUES/Demography/Gender/Female/",
        "dataType": "ENUM_VALUE"
    }
} ]
```

2.2.2.7 Saving a Query

On top of modifying the `Query` model to fit the PIC-SURE format, only the API calls to handle query saving need to be modified, the calling code remains the same. The IRCT implementation to save and re-use saved queries is not complete: it is originally possible to only save queries, but not list or load them. See section 2.2.3.1.1 for the additional implementation in IRCT to add these features.

In `ResourceService.saveQuery()` the API request becomes:

```
POST /queryService/queries
{
    "queryName": <query_name>,
    "query": {
        <query_body>
    }
}
```

Response:

```
{
    "queryId": <query_id>
}
```

In `ResourceService.getQueries()` the API request becomes:

```
GET /queryService/queries
```

Response:

```
{
    [
        "queryId": <query_id>,
        "queryName": <query_name>,
        "query": {
            <query_body>
        }
    ], [
        ...
    ],
    ...
}
```

In `ResourceService.updateQuery()` the API request becomes:

```
PUT /queryService/queries/<query_id>
{
```

```
    "queryName": <query_name>,
    "query": {
        <query_body>
    }
}
```

Response:

```
{
    "message": <status_message>
}
```

In `ResourceService.deleteQuery()` the API request becomes:

```
DELETE /queryService/queries/<query_id>
```

Response:

```
{
    "message": <status_message>
}
```

2.2.2.8 Query Step 3: Data Export

Using data export with only the native *select* and *where* clauses of the PIC-SURE API reveals a bit limiting as IRCT originally supports only tabular results, i.e. a classic row-columns result. Moreover there is no support for custom export parameters such as different types of data, only format of the result can be chosen between *TABULAR* or *JSON*. On the other end, backends such as tranSMART supports elaborated parameters for getting results, which is a required features for our system. For these reasons we are not using the native export mechanism of PIC-SURE in order to select the format of the data, but are defining a custom *process* clause named `EXPORT`, if the resource supports it. Modifications in Glowing Bear to use PIC-SURE can be summarized in three points:

- Get available data types and file formats for a specific query (export parameters);
- Submit export request;
- List exports and download them.

2.2.2.8.1 Available Export Parameters `ResourceService.getExportFileFormats()` requests to the back end the supported file formats for the exports. `getExportDataFormats()` requests to the back end the supported types of data to export, given the current query. These two requests are merged into `getExportFormats()` and the API calls are replaced by using a PIC-SURE query with specific *select* clauses using the operation `EXPORT`, example:

```
POST /queryService/runQuery
{
  "select": [ {
    "operation": "EXPORT",
    "fields": { "SUPPORTED_FORMATS": "[data, file]" }
  },
    <query_select_clauses>
  ],
  "where": [ <query_where_clauses> ],
  "alias": "<query_alias>"
}
```

`QueryService.updateExports()` is modified to take into account the merging of these two methods and the result that is now represented differently.

2.2.2.8.2 Export Request `ResourceService.createExportJob()` and `runExportJob()` that respectively create an run an export job are merged into `runExportJob()`. The call made is similar the previous, using *select* clauses to specify the parameters of the export job:

```
POST /queryService/runQuery
{
  "select": [ {
    "operation": "EXPORT",
    "fields": { "data": "clinical", "file": "TSV" }
  }, {
    "operation": "EXPORT",
    "fields": { "data": "mrna", "file": "JSON" }
  },
    <query_select_clauses>
  ],
  "where": [ <query_where_clauses> ],
  "alias": "<query_alias>"
}
```

Note that the name of the fields match the possible values of the `SUPPORTED_FORMATS` field.

`GbExportComponent.runExportJob()` is modified to take into account the merging of these two methods and the result that is now represented differently.

2.2.2.8.3 Manage Exports `GbExportComponent.updateExportJobs()` takes care of refreshing the list of exports previously made by using `ResourceService.getExportJobs()` to make the API request. This API request is modified to:

```
GET /resultService/available
```

`GbExportComponent.downloadExportJob()` take care handling the download of the file containing

the exported results, and to do so it uses the API request made in `ResourceService.downloadExportJob()`. This call is modified to:

```
GET /resultService/result/<result_id>/ZIP?download=yes
```

Note that before that a first call is made to get the available formats for the results, but in the case of multi-dimensional data it will always return `ZIP`:

```
GET /resultService/availableFormats/<result_id>
```

2.2.3 IRCT-Related Implementation

2.2.3.1 IRCT Core Modifications

2.2.3.1.1 Saving Queries: Additional API Calls The original IRCT does implement an API call to save a specific query: `POST /queryService/savequery`. However it does not provide a way to manage the existing queries. As this is a required feature, we are adding it in the IRCT core. We modify the class `edu.harvard.hms.dbmi.bd2k.irct.cl.rest.QueryService` to add the following methods:

- `getQueries()`, implementing API call `GET /queryService/queries`
- `saveQuery()`, implementing API call `POST /queryService/queries`
- `deleteQuery()`, implementing API call `DELETE /queryService/queries/<query_id>`
- `updateQuery()`, implementing API call `PUT /queryService/queries/<query_id>`

The mapping API calls to methods is done using JAX-RS [3].

The class `irct.controller.QueryController` that implements the logic of managing queries. It needs to be modified so that the saved queries are associated to an user: the users should be able to manage only their own queries.

2.2.3.1.2 Data Export: Support of Additional Data Type The original IRCT supports results of following types: `TABULAR`, `JSON`, `HTML`, `IMAGE`. Supports for types of results is represented through the enum `edu.harvard.hms.dbmi.bd2k.irct.model.result.ResultDataType` and the the code that manipulates it. The data exports in tranSMART and i2b2 gives multi-dimensional data that are not supported with this construction. To support such results, we add an IRCT result type `ZIP`. The resources using this type declare it through their `irct.model.resource.implementation.QueryResourceImplementationInterface.getQueryDataType()` method.

In the newly created package `irct.model.result.zip` we create a new class that implement the `ZIP` result type, to do do it implements the classes `irct.model.result.Data` and `irct.model.result.Persistable`. This is a pretty simple implementation that only stores the resulting files from resources on the file system.

2.2.3.1.3 Authentication Modifications

TBD: modifications to support the authentication design

2.2.3.2 IRCT Resources Implementation

The IRCT resources needs to define two things:

- declaration of its parameters and the kind of requests it supports, through SQL statements loaded in the database (we are using PostgreSQL);
- a Java class implementing some interfaces in a way that match the declaration of the resource capabilities.

Some of these things are already existent, but needs modifications, and some others need to be implemented from scratch. The details of this is explained in the following paragraphs.

The interfaces that are to be implemented by the resources are the following:

- `ResourceImplementationInterface`: generic resource, provides methods for setup and type of resource
- `PathResourceImplementationInterface`: methods for traversing tree exposed by the resource
- `QueryResourceImplementationInterface`: methods for running queries

2.2.3.2.1 i2b2

The i2b2 resource implementation exists in the original IRCT, along with a library that allows to communicate with i2b2. However this implementation is not exactly adapted for our goal and needs modifications.

We make the i2b2 resource declare the following *select* operations:

- `AGGREGATE`, fields:
  - `FUNCTION`, mandatory, possible values: `COUNT`
  - `DIMENSION`, optional, possible values: `observation`, `patient`
- `EXPORT`, fields:
  - `SUPPORTED_FORMATS`: possible values: `file`
  - `file`: value among the ones returns with `SUPPORTED_FORMAT`
- (without predicate): select data to export

And the following *where* predicates:

- `CONSTRAIN_CONCEPT`, constraint based on a concept, field: `OPERATOR`, optional, possible values:
  - numeric values: `==`, `<=`, `>=`, `<`, `>`

  - string values: `LIKE[exact]`, `LIKE[begin]`, `LIKE[end]`, `LIKE[contains]`

- `CONSTRAINT_PATIENT_SET`, constraint based on a patient set, fields:
  - `PATIENT_SET_ID`: identifier of a patient set stored in the back end
  - `PATIENT_IDS`: array of patient identifiers

Browsing the i2b2 tree of concepts if already implemented and does not need modification, except from the visual attributes parameters that is adapted to fit with a common standard.

The modifications are mainly targeted at fitting the exposed *where* predicates and *select* operations. However there is a larger modification, which is to adapt the parsing of the *AND / OR* queries as described section 2.2.2.4.3 as they need to be re-organized in the resource implementation to fit the native i2b2 way of querying, which has a non flexible syntax:

```
(A OR B OR ...) AND (X OR Y OR ...) AND ...
```

Note that the i2b2 resource does not support query nesting.

2.2.3.2.2 tranSMART 17.1 We make the tranSMART 17.1 resource declare the following *select* operations:

- `AGGREGATE`, fields:
  - `FUNCTION`, mandatory, possible values: `COUNT`, `MIN`, `MAX`, `VALUES`
  - `DIMENSION`, optional, possible values: `observation`, `patient`, `study`, `trial_visit`
- `EXPORT`, fields:
  - `SUPPORTED_FORMATS`: possible values: `file`, `data`
  - `file`: value among the ones returns with `SUPPORTED_FORMAT`
- (without predicate): select data to export

And the following *where* predicates:

- `CONSTRAIN_CONCEPT`, constraint based on a concept, field: `OPERATOR`, optional, possible values:
  - numeric values: `==`, `<=`, `>=`, `<`, `>`
  - string values: `LIKE[exact]`, `LIKE[begin]`, `LIKE[end]`, `LIKE[contains]`
- `CONSTRAIN_FIELD`, constraint based on a field in one of the dimension, fields:
  - `DIMENSION`
  - `FIELD`
  - `OPERATOR`
  - `VALUE`

- `CONSTRAIN_PEDIGREE`, constraint based on relationship between patients, fields:
  - `TYPE`, e.g. `PARENTS_OF`, etc. among permitted values
  - `BIOLOGICAL`, possible values: `YES`, `NO`, `BOTH`
  - `CONSTRAINT_END`
- `CONSTRAINT_PATIENT_SET`, constraint based on a patient set, fields:
  - `PATIENT_SET_ID`: identifier of a patient set stored in the back end
  - `PATIENT_IDS`: array of patient identifiers
- `NESTING`, fields: `TYPE` (values: `start` or `end`)

First in order to support all the calls to tranSMART 17.1 backends that are needed, a simple client library for tranSMART REST API v2 is developed. In terms of features, this library offers all calls that were previously available in Glowing Bear, and that will be used to answer the calls from the PIC-SURE API.

The tree exposed through the PIC-SURE API is the same as the tranSMART 17.1. Natively PIC-SURE does not support browsing the tree at more than one level at a time, so we add another relationship supported in browsing the tree: `CHILDREN-DEPTH-X`, that will return children with a depth up to `X`. The visual attributes parameters is added in order to display with the proper type in the UI the node, an example of this is the study: it is a `STUDY` node in the tree, and in order to be displayed as such in the UI the visual attributes parameters is set as such.

All the predicates and operations described before are translated into the native API in a straightforward way, as all the information for input is available. The is a special case: if the queried concept belongs to a study, a study constraint is added, while if the concept is cross-study it is left as is. The nesting of queries described section 2.2.2.4.3 is implemented in the resource and translated in the native API.

2.2.3.2.3 SHRINE The SHRINE API is basically a limited i2b2 API (saves for some very minor additional fields). It supports only counts, which means no data exports. The only significant difference is in the results, as one query brings several results. We take the i2b2 implementation and slightly modify to the purpose of being compatible with SHRINE nodes.

The resource exposes the following *select* operations:

- `AGGREGATE`, fields:
  - `FUNCTION`, mandatory, possible values: `COUNT`
  - `DIMENSION`, optional, possible values: `observation`, `patient`

And the following *where* predicates:

- `CONSTRAIN_CONCEPT`, constraint based on a concept, field: `OPERATOR`, optional, possible values:
  - numeric values: `==`, `<=`, `>=`, `<`, `>`
  - string values: `LIKE[exact]`, `LIKE[begin]`, `LIKE[end]`, `LIKE[contains]`

2.2.4 Back Ends Modifications

2.2.4.1 i2b2 & derivatives

TBD: modify PM to support OIDC (will support medco and shrine as well)

2.2.4.2 tranSMART 17.1

TBD: using spring security, get OIDC compatibility

Appendix A

Implementation Timeline

A.1 Overall Planning

Work Packages	Task	Timeline (week #)																								
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
WP 1 Exploration & Design	Get familiar with technologies used	X	X																								
	Setup development environment			X																							
	Choice of scenario & PIC-SURE			X																							
	Detailed system design objective 1			X	X	X	X	X																			
	Detailed system design objective 2																		X								
WP2 System Implementation	Obj. 1: Common interoperability components								X	X	X	X	X	X													
	Obj. 1: i2b2 & SHRINE connectors												X	X	X	X	X										
	Obj. 2: MedCo connectors																			X							
	Obj. 2: Glowing Bear crypto module																			X	X						
WP3 System Evaluation	Obj. 1: Deployment & tests																	X									
	Obj. 2: Deployment & tests																				X						
	Scalability tests																					X					
	Performance measurement																						X				
WP4 Wrapping up	Deploy final version & incorporate optimizations																						X	X			
	Project report writing					X	X	X									X	X				X	X	X			
	Master thesis writing																								X	X	X
Milestones								M1										M2		M3	M4			M5			M6

Figure A.1: Gantt Chart of Planning v2 (updated at M1)

A.2 Objective 1: Interoperability Layer

The timeline aims to have first a working system with Glowing Bear, IRCT and i2b2 by taking an horizontal approach, i.e. have each feature working end-to-end individually. Then the different resources are made to be working with the system. Total of the implementation of objective 1 is 10 weeks, that are detailed in the breakdown.

- System initialization (login, resource list, etc.): *2w*
  - Setup: i2b2 + IRCT + keycloak deployment
  - IRCT resource: i2b2 resource declaration

  - Glowing Bear: `IRCTResourceService`, configuration
  - IRCT: small modification for OIDC support
  - i2b2: OIDC support modification of PM cell
- Concepts Tree: *1w*
  - Glowing Bear: PIC-SURE modifications
  - IRCT resource: i2b2 tree browsing
- Query Step 1: Constraints and Aggregates: *2w*
  - Glowing Bear: *where* clauses construction
  - IRCT resource: i2b2 constraints
  - Glowing Bear: aggregate queries
  - IRCT resource: i2b2 aggregate queries
- Query Step 2: Variables Selection: *0.5w*
  - Glowing Bear: *select* clauses construction
  - IRCT resource: i2b2 variables selection
- Query Saving: *0.5w*
  - IRCT core modification: add missing API calls
  - IRCT core modification: query per user
  - Glowing Bear: PIC-SURE modifications
- Query Step 3: Data Export: *1w*
  - Glowing Bear: PIC-SURE modifications
  - IRCT core modification: add zip file type
  - IRCT resource: i2b2 result export
- IRCT resource: tranSMART 17.1: *2w*
  - Setup: tranSMART 17.1 deployment
  - Modification for OIDC support
  - Concepts tree browsing
  - Constraints and Aggregates
  - Variables selection
  - Data export
- IRCT resource: SHRINE: *1w*
  - Setup: SHRINE deployment
  - IRCT resource: extend i2b2, make SHRINE-specific modifications

Appendix B

Docker-Based Testing Infrastructure

This appendix describes the docker-based infrastructure put in place to test the systems.

B.1 Docker Images

This section describes the different Docker images of the infrastructure.

B.1.1 WildFly Application Server

- Ports exposed
  - 8080: deployments endpoint
  - 9990: WildFly management interface
- Volumes
  - `/opt/jboss/wildfly/standalone/deployments/`: deployment folder of WildFly
  - `/opt/jboss/wildfly/standalone/configuration/`: configuration folder of WildFly
  - `/opt/jboss/.grails/`: Grails configuration folder (for user `jboss`)

This sets up a working WildFly server and install several tools used to build from source the different WARs that are deployed. Upon initial creation of the container there is no deployment, they are built on demand with the help of the build scripts that are shipped in the image. With the container running, run the following command to build a deployment:

```
docker exec -it <container_name> build-war.sh <deployment_name>
```

When running with the default docker-compose configuration, the name of the container would be `deployments_wildfly-server_1`. Below are described the different deployments that can be built.

B.1.1.1 i2b2

Deployment name: `i2b2`, URL exposed: `/i2b2/`

The i2b2 WAR is actually a deployment of Axis2. Within this deployment, all the i2b2 cells are deployed as AAR files (Axis2 Archive):

- `CRC`: Clinical Research Chart (data repository)
- `ONT`: Ontology management
- `PM`: Project Management (authentication and authorization)
- `WORK`: Workflow management (query, result sharing)
- `FR`: File Repository
- `IM`: Identity Management

B.1.1.2 IRCT

Deployment name: `irct`, URL exposed: `/IRCT-CL/`

Note that before you can build the IRCT deployment, i2b2 should have been built before (in order to deploy the JDBC drivers), and the database should up and initialized correctly. This is due to the fact that IRCT uses Hibernate to handle its data storage in the database, and when ran it will validate and update if necessary the database schema. To resolve an incompatibility of IRCT using Hibernate with the use of PostgreSQL, Hibernate is configured to add a prefix to all of the tables.

B.1.1.3 tranSMART 17.1

Deployment name: `transmart-17.1`, URL exposed: `/transmart-17.1/`

B.1.2 PostgreSQL Database Server

- Ports exposed
  - 5432: PostgreSQL port
- Volumes
  - `/var/lib/postgresql/data/`: PostgreSQL database files

This sets up a working PostgreSQL server and install several tools needed by the loading scripts that are ran upon the first run of the container. These scripts are copied in the `/docker-entrypoint-initdb.d` folder.

Below is an overview of the databases created.

B.1.2.1 i2b2

Contains the i2b2 database schemas for all the cells and the default demo data.

B.1.2.2 irct

Contains a snapshot of the IRCT database structure (that is updated as needed by Hibernate), and the resources information used by IRCT to connect to the resources:

- `i2b2-local`: the local i2b2 instance

B.1.2.3 transmart_17_1

Contains the structure and some default test data for tranSMART 17.1.

B.1.3 Lighttpd Web Server

- Ports exposed
  - 80: HTTP port

This sets up a working Lighttpd server with PHP and install several services:

- `/phppgadmin/`: phpPgAdmin PostgreSQL management tool
- `/i2b2-client/`: the i2b2 webclient, using the local i2b2 instance
- `/i2b2-admin/`: the i2b2 admin tool, managing the local i2b2 instance

B.2 Docker-Compose Run Configuration

A default `docker-compose.yml` is provided and works out of the box to create and deploy the images described section B.1. It creates a network to allow all the containers to communicate, exposes on the host the same ports as exposed by the container, and maps the WildFly volumes to directories alongside the Dockerfile. It does not require additional argument to be built and upped with the default configuration.

References

[1] *Hibernate Framework*. URL: https://en.wikipedia.org/wiki/Hibernate_(framework).

[2] *IRCT Source Code*. URL: https://github.com/hms-dbmi/IRCT.

[3] *JAX-RS: Java API for RESTful Web Services*. URL: https://en.wikipedia.org/wiki/Java_API_for_RESTful_Web_Services.

[4] *PIC-SURE RESTful API Documentation*. URL: http://bd2k-picsure.hms.harvard.edu.

[5] *todo*. URL: todo.

[6] *Web Application Archive (WAR)*. URL: https://en.wikipedia.org/wiki/WAR_(file_format).