BD2K PIC-SURE

RESTFULL API PROTOCOL

Version 1.1

Document History

Date	Name	Notes
5/5/16	Jeremy R. Easton-Marks	Initial Draft
5/5/16	Paul Avillach	Updates
6/30/16	Jeremy R. Easton-Marks	Updates
11/30/16	Jeremy R. Easton-Marks	Updated for 1.1

Table of Contents

Introduction

Overview

One of the stated goals of the NIH Big Data to Knowledge (BD2K) initiative is to harvest the wealth of information contained in Big Data to advance our knowledge of human health and disease. As part of this initiative the Harvard University Medical School (HMS) Department of Biomedical Informatics Patient-Centered Informatics Common: Standard Unification of Research Elements (PIC-SURE) is developing an open-source infrastructure that will foster the incorporation of multiple heterogeneous patient level clinical, omic, and environmental datasets. This system embraces the idea of decentralized repositories (resources) of varying types, and protocols. It provides a single simple secure communication interface that can perform queries, joins, and computations across different resources. To implement this we created the BD2K PIC-SURE RESTfull API. Short acronym is IRCT (Inter Resource Communication Tool) which provides a resource agnostic service through which multiple resources of varying types, and protocols can be accessed. Users communicate with this service using a series of Representative State Transfer (RESTful) calls.

The easiest way to understand what the IRCT is by explaining what it isn't. First and foremost it is not an attempt at a universal API. It is not going to be the one API, or protocol that all biomedical application should implement. It does not require existing resource to change their protocol. It does not define an ontology, or prefer one type of ontology over another. And it does not restrict the where, or how data is stored.

The core idea behind IRCT is that no API, protocol, ontology, or application is going to be a perfect fit for all biomedical research. However being able to combine data across different resources will provide a powerful tool for researches. The IRCT accomplishes this by allows different resources to be defined – by initial configuration - what they are, what they have, and what they can do. It can support existing resources without requiring them to make any changes, and allows new resources to be quickly integrated. This is accomplished by creating a 'Resource-Driven API'.

All actions that can be performed by resources from the IRCT are abstracted into a set of definitions. These definitions provide basic information such as what predicates are supported, and can be executed on the resource. Since the IRCT doesn't restrict what actions, and commands can be performed individual resources can define any number of predicate operators, data types, parameters, and. This model is flexible enough to support several different resource types, but still rigid enough to allow users to quickly create and execute different actions with the IRCT.

After a user executes an action in the IRCT it needs to be passed to the resource(s) that will execute it. It accomplished this by having a set of resource interfaces that translate that action into the protocol of the resource. Each resource interface supports a specific protocol, and can connect to multiple resources of that type. Results from resources that are returned from those actions are also converted into an IRCT result, which can be returned in many different formats. The result set can also be used to feed into other actions, or returned to the end user in a variety of different formats.

Document Goals

This document provides an introduction and guidelines for communicating with the IRCT. It also serves as an introduction to the resource-driven API that allows for the communication with multiple systems with one protocol. This will allow an end-user to explore, query, process, and visualize the rich set of biomedical data that exists. It does not provide for any technical guidelines for implementing an IRCT server or resource. It only serves as a description of the protocol, and ideas behind it. This document will continuously be updated as future versions of the IRCT RESTfull API are deployed.

Technical Overview

System Architecture

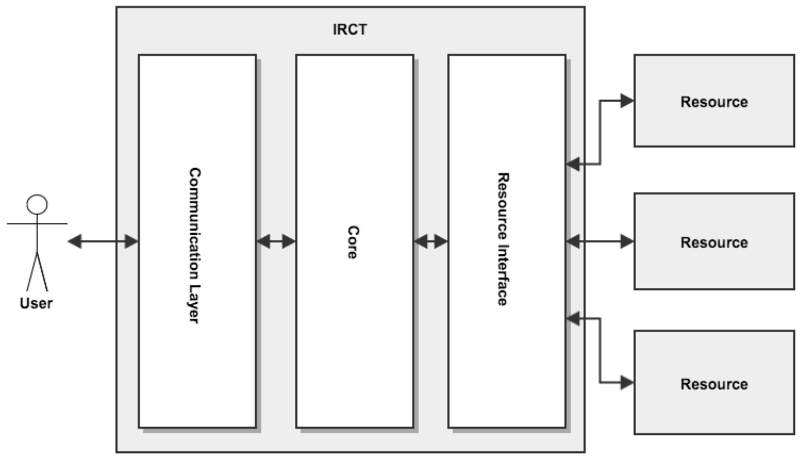


Figure 1: IRCT Overview

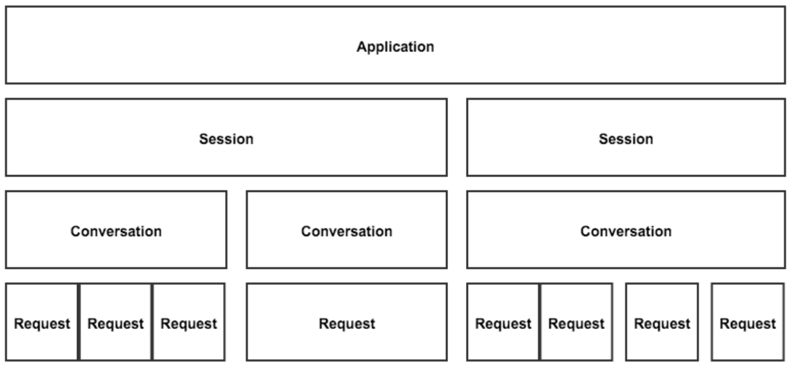
Our approach consisted of building four layers: a communication layer that implements the RESTful service for interacting with an end user; a process engine that manages the requests for queries, and process before returning it to the requester; a collection of extensions that can be used to add or remove additional functionality; and a set of resource interfaces that provides a means of communicating between the IRCT and different services such as i2b2/tranSMART, and ElasticSearch.

The communication layer implements a RESTful service that allows for a simple, and secure creation of actions and retrieving results. All communication with the server is handled through a secure connection, and only authorized users are allowed to use the system. RESTful requests from the user are used to create a series of commands that create queries, and processes that are then passed to the process engine. When a query is complete, a user can then request the results that are returned in the a desired format.

The core consists of two major components: an execution handler, and a results handler. The execution handler initiates, and manages processes that are requested by the communication layer. As part of this it ensure that all processes are run on the correct resources through the resource interface and return successful results. The results handler takes the results from the resource interfaces and combines them with other results, and processes before passing them back to the user through the communication layer to the requester.

The resource interface communicates with the different resources in their supported protocol. It accomplished this by converting the query, or process into the native calls for the given resource. When the given process or query is complete it converts the results from the native format to a different formats such as CSV, XML, JSON, and XSLT. Different resources interfaces can be easily created and added to the IRCT.

Scopes



Scopes are used to define the life cycle of information inside an application. They are useful in keeping data isolated to specific interactions. While more then the scopes listed here exist, the ones described below are the most important for interacting with the IRCT.

Application scope is defined as containing all the information that is for that application. In the case of IRCT this includes a list of all resources, joins, and configuration parameters. They are initiated when the application is started, and are lost when the application is stopped if the information has not been persisted.

All users interactions are accomplished inside a session scope. They are initiated when a user starts communicating with a server, and typically time out after a set period of time, which can range in time from minutes to hours. Either the user or the server can end the session. The IRCT saves information about which user is currently interacting with including basic security information.

Interactions that require multiple requests to accomplish task statefuly are considered to be in conversation scoped. A session may have multiple conversations occurring simultaneously, sequentially, or a combination of the two. Conversations are given a unique id that is tied to specific session. The IRCT uses conversation for creating actions such as queries, processes, and visualizations. This allows for actions to be broken up into many smaller requests.

Individual calls to a server that don't require previous or future calls are request scoped. They may or may not be occur within a conversation or may occur in the middle of a conversation without causing any interruptions to that conversation. They are typically used for information gathering purposes. The IRCT uses request scopes for individual calls such as retrieving results, traversing paths, or getting information about a resource.

Versioning

All services of the IRCT are versioned starting with release 1.0. The version is part of all RESTful URLs the system makes available. It is to be included after the protocol but before the service. All major releases do not require the 0 but all minor releases will require the sub-release number (e.g. `/rest/v1.1/systemService`). It is expected that old versions will be supported for at least 90 days after the new release is made available. The one exception to the versioning rule is to get the currently supported version. Any requests to no longer supported versions will include an error message stating that the version is no longer supported, and the correct version to use.

Errors

Two general categories of errors are expected to occur while communicating with the IRCT. The first and most common will be bad requests. This includes requests with invalid ids, or making requests to resources that do not support a specific action, or are incomplete. These errors will be returned using HTTP 4xx codes. These errors will include a JSON object that describes the type of error, and a message that describes the error itself. These messages should be as accurate as possible. The second type of error will be system errors and they will return a 5xx error code. The returned information in this error will be limited except to say that an internal error has occurred.

```
{
   “status” : “Invalid Process”,
   “message” : “The resource does not support this type of
process”
}
```

System

Overview

The IRCT server itself can be treated as another resource. It has its own supported data types, and joins. While it should not be considered as an initial source of data it can contain data that has been produced through querying, or computing. This data can then be further processed, or joined with other data sets.

System Information

To get the current system information a call is made to the `/rest/currentVersion` that will return the current REST version that is supported. The returned version number should then be used for all subsequent calls as described on page 10.

```
GET /rest/currentVersion
{
   "version" : ...VERSION...
}
```

Data Types

All fields and input have a set of data types that they support. These data types help to ensure that the data is sent in the correct format, as well as allowing data to be translated between different data types. The IRCT itself should provide a base set of data types that act as primitives. It is expected that all IRCT servers should support some of these. All resources, if needed, can extend these types. These primitive types include string, date, number, result set, column as well as others. A description of all the primitive data types can be found on page (46) and more information about the data type return format can be found on page (55).

To retrieve an array of data types the user makes a request to `/systemService/dataTypes`. It is expected that this will be called when a user initially begins to interact with the IRCT server through a session and not done repeatedly.

```
GET /rest/v1/systemService/dataTypes
[...DATA TYPES...]
```

Security

Overview

The IRCT provides a secure way of accessing patient data across multiple resources. It does not store any passwords, and itself does not do any authentication of which resources the user has access to. By not implementing any specific authentication mechanism it avoids the need for resources to conform to the same access restrictions. It also keeps the governance for individual access to be defined by the resources themselves. It does require that all resources that have access restrictions use the same identity service provider. This identity service provider may access other identity stores. The identity service provider must support OpenID Connect and all connections should be secured using HTTPS.

Creating a secure session with the IRCT consists of two steps. The first step involves authenticating against an Identity Provider. This requires a user to, using a browser, access a login page. This page will provide a form for login credentials that are submitted to the identity provider. Upon successful authentication the user is redirected back to the success page. This page can be used to retrieve a randomly generated 12-digit alphanumeric unique access key that the user will use to create a session for interacting with the IRCT. This identifier will allow a secure session to be generated within a set period of time (Currently 1 hour).

Upon successfully creating an access key, the user can then create a secure session. To start a secure session a user will pass the access key to the `/securityService/startSession` service. If the access key is valid then a secure connection is established, and all future requests within that session will be as the user who created that access key. All calls to resources through the IRCT will be as the identified user of that session. A session will remain active until a user either logs out using the `/securityService/endSession` command or through a timeout after inactivity in that session (Currently 120 minutes).

A user may access the IRCT and its services in two different ways. The first by being by passing the JWT token generated for another application to the `/securityService/startSession` service. The second is through logging into the IRCT itself without the use of another application. The process for accomplishing this is detailed more in the next section (IRCT Authentication).

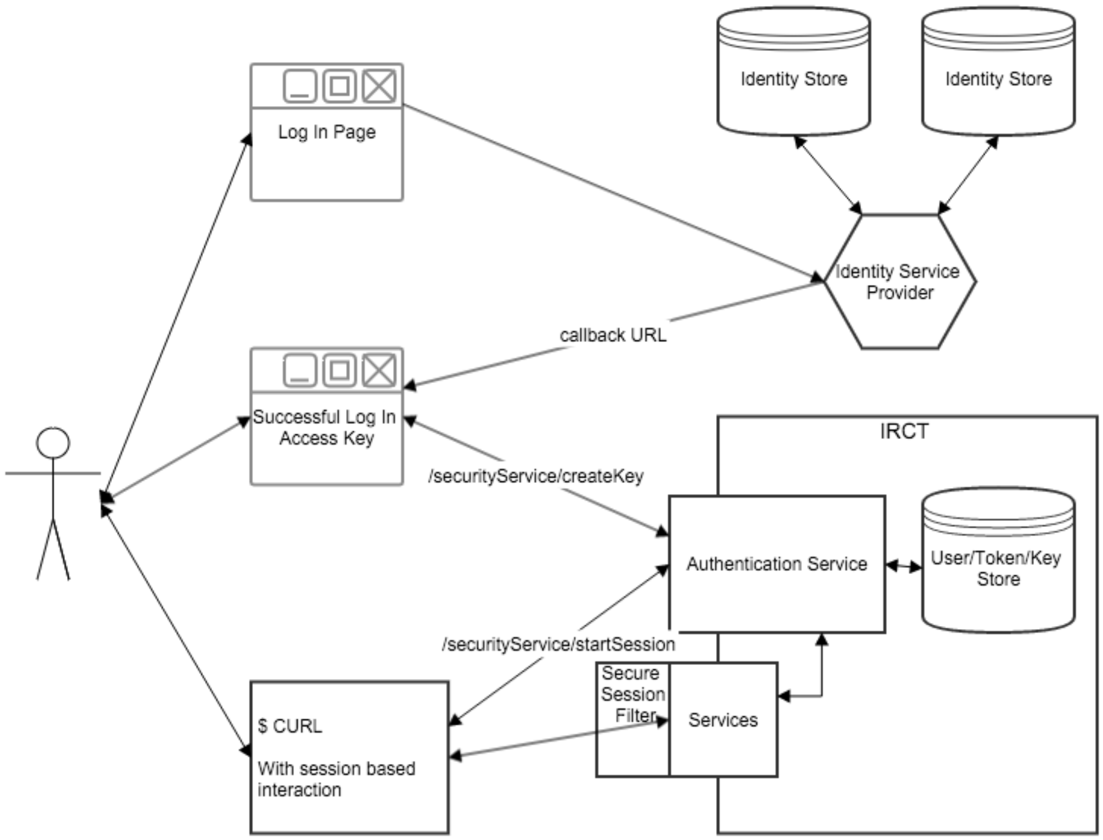

Identity Store
Identity Store
Log In Page
Identity Service Provider
callback URL
Successful Log In Access Key
IRCT
/securityService/createKey
Authentication Service
User/Token/Key Store
/securityService/startSession
Secure Session Filter
Services
$ CURL
With session based interaction

Figure 2: Security Overview

IRCT Authentication

The authenticating of user requires several steps as shown in Figure 2: Security Overview. After a user accesses the login page for the IRCT, the page should make a single call to the `/securityService/createState` service. This service will generate a random 32 character alphanumeric sequence. This sequence is passed to the outside authentication service and is returned by the authentication service. This is used to ensure that the user that initially made the authentication request is the same as the one who is authenticated.

```
GET /rest/v1/securityService/createState
{
   “state” : “...STATE...”
}
```

The random 32 character alphanumeric sequence is that is generated by the IRCT should be passed to the outside authentication service as the state variable. This call will also need to include the callback url /securityService/callback, response type as code, and response scope set as openId. Check with your authentication provider for any additional documentation.

```
GET
AUTHENTICATION_PROVIDER_URL/authorize/?response_type=code&client_
id=$CLIENT_ID&redirect_uri=$IRCT_URL/rest/v1/securityService/call
back&state=$STATE&scope=openid
```

Upon successfully logging in with the outside authentication provider the user will be logged redirected back to the IRCT. This callback will include a unique identification code generated by the authentication provider. It will also include the state parameter passed to it. The IRCT will check to make sure that the state parameter generated by the IRCT, and the one passed back by the outside authentication provider are equal.

GET /rest/v1/securityService/callback?code=*$CODE*&state=*$STATE*		
CODE	**REQUIRED**	Authorization code from outside authentication provider
STATE	**REQUIRED**	State code originally created by IRCT
Sends a redirect to the success URI if the user was logged in successfully. If log in was not successful it reports back the error.		

Secure Communications

All communications with the IRCT must be done in a secure session. This can be accomplished in two different ways. The first is by logging into the IRCT and then communicating with the services as you would normally. However, it is envisioned that most users would like to be able to access the IRCT from a variety of sources and don't want to pass their username and password. To facilitate this we allow a logged in user to create an IRCT access key by calling `/securityService/createKey`. The key generated is a random 12 digit alphanumeric string that can then be used to create a secure session as that user without the need to reenter their login credentials. This key will expire after a set period of time after it is created (Currently 12 hours).

```
GET /rest/v1/securityService/createKey
{
   "key" : "...KEY..."
}
```

To create a secure session with the IRCT access key a user needs to pass it to the `securityService/startSession` service. If the key is valid, and has not expired then a secure session will be started. This session will be associated with the user who generated the key. All interactions without outside resources will appear to be coming from that user. This session will stay active as long as the user interacts with the IRCT. However if the session times out and the IRCT access key is still valid then the key can be used again. This allows users to pause their work, and come back to it later without having to generate a new key.

GET /rest/v1/securityService/startSession?key=*$KEY*		
KEY	**REQUIRED**	IRCT Access Key

{
   “status” : “success”
}		

When a user is done using their secure session then they can call the `securityService/endSession` command. This will end the secure session and all subsequent calls will return with HTTP 403 error codes.

GET /rest/v1/securityService/endSession

{
   “status” : “success”
}

Example

```
GET /rest/v1/securityService/createKey
{
   “key” : “8Ryxfs3HuDR3”
}

GET /rest/v1/securityService/startSession?key=8Ryxfs3HuDR3
{
   “status” : “success”
}

GET /rest/v1/securityService/endSession
{
   “status” : “success”
}
```

Resources and Element Relationships

Overview

A resource can be considered anything that provides data, or processing methods to an end user. This broad definition allows the IRCT to communicate with many different types of services. As stated before it does not require the resource to conform to any specific protocol, or ontology. This means a resource can be as simple as a file, or as complicated as an i2b2 instance. The resources can exist anywhere from being hosted on the same server, to being hosted behind a firewall in another institution.

Resources define what they are, and what they can do through the IRCT. This includes information about what data types that they support as well as what actions such as queries or processes they may perform. These actions define how an end user uses the IRCT to interact with them. This allows for a single protocol and server to be used to communicate with any number of resources of any number of types. How the different actions are defined, and used are defined in their respective sections of this documentation.

Resource

A list of all available resources are available by calling the `/resourceService/resources` commands with no type filter. This will return a JSON array of available resources. If only resources that support a specific action are needed then the type filter can be added. Currently the type can be a `PROCESS`, or `QUERY`. The JSON schema returned that is used to define the resources can be found on page 49.

GET /rest/v1/resourceService/resources?type=*$TYPE*		
TYPE	**OPTIONAL**	Type of resource e.g. PROCESS, QUERY
[...RESOURCES...]		

Paths

Paths provide a basic way of navigating through the different resources and elements available to a user. The first level of a path is the resource name as defined by the IRCT. All subsequent sections are the path within the individual resource.

```
BASIC FORMAT:
   /RESOURCE/PATH/SUBPATH/...
```

All paths follow this format regardless of the ontology type. All paths levels have sub paths that are separated by the '/' character (`U+002F`). Different resources may choose to implement this in different methods. It is considered best practice for a path to represent a similar ontology as the resource itself implements. For example a resource may have different projects that are accessible. This would be displayed as the first level of the resource. Other resources may choose to use the first level of the path as the different subservices offered by the resource. Other resources may choose to implement the ontology tree as the first level forgoing any special initial path.

Entities

All paths within a resource correspond to entities. These entities contain information that can be explored. They can be part of an element in an ontology, a table in a database, or a field. Entities returned by the IRCT can contain a variety of information. The paths of the entities, as well as their types are used to restrict the type of queries that can be performed on them. More information about the schema returned by the IRCT to describe the different entities can be found on page 54.

Relationships, Search, and Relationship Traversal

To traverse the path relationships between entities a user makes a call to the resourceService. The default relationship type that is returned is CHILD. Different relationships can also be passed if the resource supports it. Information about what relationships a resource supports is contained in the resource information that is returned by the IRCT.

GET /rest/v1/resourceService/path/*$PATH*?relationship=*$RELATIONSHIP*&*search=$SEARCHTERM*		
PATH	**REQUIRED**	The path to the entity
RELATIONSHIP	**OPTIONAL**	The relationships for that entity. The default is CHILD.
SEARCHTERM	**OPTIONAL**	Search for all paths that contain this string
[...ENTITIES...]		

Example

```
GET /rest/v1/resourceService/resources
[
   {
      “id” : 1,
      “name” : “NHANES”,
      “ontologyType” : “TREE”,
      “implementation” : {
         “type” : “i2b2/tranSMART”,
         “returnEntity” : [...],
         “editableReturnEntity” : true
      },
      “supportedJoins” : [...],
      “supportedPredicates” : [...],
      “supportedProcesses : [],
   }, ...
]

GET /rest/v1/resourceService/path/NHANES
[
   {
        "pui":"/NHANES/Public Studies",
        "name":"Public Studies",
    }
]
```

Query Creation

Overview

As described above the IRCT is a resource driven API. For querying this means that each resource defines what components of a query that it supports and what they can do. Queries are defined inside the IRCT of containing clauses. These clauses can be of three types; Select, Where, and Join. Select clauses instruct the resource to return these data types, while Join clauses tell the resource how to join the data. Where clauses are more complicated as they tell the resource how to filter the data using different predicates. Predicates are a set of different filtering operations that can be performed by a query on a field. This may be something simple as filtering out all entries that a null value, or as complicated as returning only the genomic rarity of a SNP. Not all resources will support all three different types of clauses.

Resource Defined Predicates, and Queries

Resources define which predicates they can support. This information is unique to each resource and is defined by the IRCT (pg. 49). Predicate must include basic information such as its name, and the fields it uses. The fields contain information about what the field name, and what data it can use. The example below shows the JSON describing a predicate that allows a user to filter for fields that exist.

```
{
        "predicateName": "CONTAINS",
        "displayName": "Contains",
        "description": "Contains value",
        "default": true,
        "fields": [
          {
            "name": "By Encounter",
            "path": "ENOUNTER",
            "description": "By Encounter",
            "required": true,
            "dataTypes": [],
            "permittedValues": [
              "YES",
              "NO"
            ]
          }
        ],
        "dataTypes": [
          "STRING",
          "INTEGER",
          "FLOAT"
        ],
        "paths": []
      }
```

In this example the query can use this information to create a where clause on a path that will filter on values that only have Null or Not Null value.

Creating a Query

To create a query a user must start a conversation within the session. The conversation id (CID) is returned and is the unique query identifier. Multiple queries can occur simultaneously, with each having a unique query identifier. Queries will expire if too much time is spent between calls. Currently queries can only be run against one resource at a time.

```
GET /rest/v1/queryService/startQuery
{
   "cid" : ...CID...
}
```

Adding a where clause to query involves call the `/clause` command with the type being set to where. Each where clause contains a path to the resource and corresponding entity as well as predicate that defines what type of filtering should be done by that clause.

If the resource supports it several where clauses can be added to a query.

GET /rest/v1/*queryService/clause?type=where*		
cid	**REQUIRED**	Query Identifier
path	**REQUIRED**	Path the clause is running on
predicate	**REQUIRED**	Name of the predicate
logicalOperator	**OPTIONAL**	Type of logical operator to use
clauseID	**OPTIONAL**	Replaces a given clause
data-*	**OPTIONAL**	A series of parameters that are prepended with the field name and are equal to the user supplied field value.
{ “clauseId” : ...clauseID... }		

Adding a select clause

GET /rest/v1/*queryService/clause?type=select*		
cid	**REQUIRED**	
path	**REQUIRED**	
alias	**OPTIONAL**	
{ “clauseId” : ...clauseID... }		

Adding a join clause is similar to adding a where clause.

GET /rest/v1/*queryService/clause?type=join*		
cid	**REQUIRED**	Query Identifier
joinType	**REQUIRED**	Type of Join
data-*	**OPTIONAL**	A series of parameters that are prepended with the field name and are equal to the user supplied field value.
{ “clauseId” : ...clauseID... }		

Clauses can also be created by passing a JSON Object that conforms to one of the Clause JSON Schemas located on pages (57-59). Instead of a GET command call the user will use a POST command to the same URL with no parameters, and the JSON object as the body.

POST /rest/v1/*queryService/clause*		
BODY	**REQUIRED**	JSON Object
{ “clauseId” : ...clauseID... }		

Running a Query

After the creation of a query calling the runQuery service will run it. This will pass the query onto the execution process engine. They will

GET /rest/v1/*queryService/runQuery*		
cid	**REQUIRED**	Query Identifier
{ “resultId” : ...resultId... }		

Saving a Query

Before a query is run it can be saved by calling the saveQuery service. This will return a query id that then can be used to load the created query at a later date.

GET /rest/v1/*queryService/saveQuery*		
cid	**REQUIRED**	Query Identifier

```
{
   "queryId" : ...queryId...
}
```

Loading a Query

Calling the loadQuery service can retrieve any saved query. This will return a new conversation id. The query can then be edited or run again as described above.

GET /rest/v1/*queryService/loadQuery*		
queryId	**REQUIRED**	Query Identifier

```
{
   "cid" : ...cid...
}
```

Running a Query with JSON

Alternatively an entire query can be composed as a JSON object and run with command. The query object as defined by the schema on pg. 56 will be passed as the body.

POST /rest/v1/*queryService/runQuery*		
BODY	**REQUIRED**	JSON Object

```
{
   "resultId" : ...resultId...
}
```

Example

In this example we are going to run a query against a resource to find the systolic and diastolic blood pressure for all patients over the age of 65.

```
GET /rest/v1/queryService/startQuery
{
   "cid" : 1
}

GET
/rest/v1/queryService/clause?type=select&path=/resource/examinati
on/bloodpressure/systolic&cid=1
{
   "clause" : 0
}

GET
/rest/v1/queryService/clause?type=select&path=/resource/examinati
on/bloodpressure/diastolic&cid=1
{
   "clause" : 1
}

GET
/rest/v1/queryService/clause?type=where&path=/resource/demographi
c/age&predicate=FILTER&operator=GT&data-value=65&cid=1
{
   "clause" : 2
}

GET /rest/v1/queryService/runQuery?cid=1
{
   "resultId" : 5309
}
```

Process Creation

Overview

In a manner similar to query creation and execution, processes can be created to run a method on a remote resource if the resource already implements it . All processes take in a series of fields as one input, but unlike queries do not have a concept similar to clauses. Processes can be defined to take results of queries, or other processes as inputs. This allows for an entire several actions to be chained together to create a desired process.

Resource Defined Processes

In a manner similar to queries resources define which process they support. The example below shows the JSON describing a process to perform a t-test on a result set.

```
{
      "name" : "Simple t-Test",
      "description" : "Performs a t-test on the given result
set",
      "fields" : [
            {
               "name" : "Result Set",
               "path" : "rs",
               "description" : "Result set for t-test",
               "permittedValues"  : ['resultSet'],
               "required" : true
            },
            {
               "name" : "Numeric Vector",
               "path" : "x",
               "description" : "Column with numeric values",
               "permittedValues"  : ['column'],
               "required" : true
            },
            {
               "name" : "Optional Numeric Vector",
               "path" : "y",
               "description" : "Optional column with numeric
values",
               "permittedValues"  : ['column'],
               "required" : false
            },
      ]
}
```

In this simple example the t-test takes is supplied with the data for three fields from the user. The first being a result set that is required and must be of the type of result. The second field must also be passed, and should be the name of the column from the result set that has numeric values. The final field is an optional field that also must be the name of a column in the result set that contains numeric data. All this information is parsed by the IRCT and then passed to the resource that defined this process.

Creating a Process

To create a process a user must start a process conversation with the /startProcess command.

```
GET /rest/v1/processService/startProcess
{
   "cid" : ...CID...
}
```

The CID is returned is the process identifier. Like queries a user can create multiple processes that can occur simultaneously.

GET /rest/v1/processService/process		
cid	**REQUIRED**	Process Identifier
resource	**REQUIRED**	Resource Identifier
processName	**REQUIRED**	Name of the process
data-*	**OPTIONAL**	A series of parameters that are prepended with the field name and are equal to the user supplied field value.
{ “processId” : ...processId... }		

Running a Process

GET /rest/v1/processService/runProcess		
cid	**REQUIRED**	Process Identifier
{ “resultId” : ...resultId... }		

Running a Process with JSON

In a similar manner as Queries the IRCT can also be passed a JSON object to run a process. The JSON object schema as defined on pg 60 is passed as the request body.

POST /rest/v1/processService/runProcess		
BODY	**REQUIRED**	JSON Object
{ “resultId” : ...resultId... }		

Example

In this example we are going to run a T-Test against a resultSet.

```
GET /rest/v1/processService/startProcess
{
   “cid” : 1
}

GET
/rest/v1/processService/process?resource=openCPU&processName=Simp
le%20T-Test&data-rs=8713&data-x=systolic&data-y=sleep$cid=1
{
   “action” : 0
}

GET /rest/v1/processService/runProcess&cid=1
{
   “resultSet” : 65834
}
```

Results

Overview

Since the IRCT interacts with different resources it needs to be able to securely store, and make accessible the results of its different actions (Query, Process). Results can be of several different types (TABULAR, JSON, HTML, IMAGE). Depending on the result type they can be downloaded in different formats (e.g. XML, CSV, Excel). Results are only available to the person who ran the action that created them. However future releases will allow for different results to be more easily shared with a group or be made public.

Availability and Result Status

To check the availability of all results to a user a single command can return an array of different results and their current status. The status of a result can be in one of several different states. The first state is CREATED, this is the initial state where an action has been created but not yet run. The second state is RUNNING, which implies the actions related to this result are running but have not completed. Intermediate results, such as those that are not the final result but part of the intermediary step will be in the AVAILABLE state. When an action has been completed it will be in the COMPLETE state and may be downloaded. If the action encounters an error the result will reflect this by returning an ERROR state. Only results that are in the COMPLETE state may be downloaded.

```
GET /rest/v1/resultService/available
[
   {
      “resultId” : ...RESULT ID...,
      “status” : ...STATUS...
   }...
]
```

GET /rest/v1/resultService/resultStatus/RESULTID		
RESULTID	**REQUIRED**	Result Id

```
{
   “status” : ...STATUS...
}
```

Downloading a Result

When a result is in the COMPLETE state it can be downloaded. To check which formats are available for that result a user can call the `/availableFormats` with the result Id and an array of available formats that the result can be returned in.

GET /rest/v1/resultService/availableFormats/RESULTID		
RESULTID	**REQUIRED**	Result Id

```
[...FORMAT...]
```

To download a result in one of the supported formats a user calls the `/result` function. A format that is supported by the result type as well as the result idmust be included. If the user wishes to set the content-disposition to a filename they can set the URL parameter download as yes. This will cause most browsers to return the results as a downloadable file. The file will be in the format of IRCT-<RESULTID>.<FORMAT EXTENSION>.

GET /rest/v1/resultService/result/RESULTID/$FORMAT?download=$DOWNLOAD		
RESULTID	**REQUIRED**	Result Id
FORMAT	**REQUIRED**	Format type
DOWNLOAD	**OPTIONAL**	Yes, or No to set the content-disposition on the response header as an attachment or not. Downloaded files are named IRCT-<RESULTID>.<FORMAT>
Result		

Example

In this example we are going to check the status of a result, what formats it can be downloaded in, and retrieve it as a CSV file.

```
GET /rest/v1/resultService/resultStatus/5309
{
   “status” : “COMPLETE”
}

GET /rest/v1/resultService/availableFormats/5309
[
   “CSV”, “JSON”, “XML”, “XLSX”
]

GET /rest/v1/resultService/result/5309/CSV?download=YES
```

REST Reference

System Service

Get Version Information

```
GET /rest/currentVersion
{
   “version” : ...VERSION...
}
```

Get Data Types

```
GET /rest/v1/systemService/dataTypes
[...DATA TYPES...]
```

Security Service

Create State

```
GET /rest/v1/securityService/createState
{
   “state” : “...STATE...”
}
```

Create Key

```
GET /rest/v1/securityService/createKey
{
   “key” : “...KEY...”
}
```

Callback

`GET /rest/v1/securityService/callback`

CODE	**REQUIRED**	Authorization code from identity provider
STATE	**REQUIRED**	State originally created by the IRCT
ERROR	**OPTIONAL**	Any error occurred authenticating the user

```
{
   “status” : “success”
}
```

Start Session

`GET /rest/v1/securityService/startSession?key=`*`$KEY`*

KEY	**REQUIRED**	Authorization code from identity provider

```
{
   “status” : “success”
}
```

End Session

```
GET /rest/v1/securityService/endSession
{
   “status” : “success”
}
```

Resource Service

Resource List

GET /rest/v1/resourceService/resources?type=*$TYPE*		
TYPE	**OPTIONAL**	Type of resource e.g. Process, Query, Visualization
[...RESOURCES...]		

Path Traversal and Search

GET /rest/v1/resourceService/path/*$PATH*?relationship=*$RELATIONSHIP&search=$SEARCHTERM*		
PATH	**OPTIONAL**	The path to the entity. If no path is specified then all path resources are returned.
RELATIONSHIP	**OPTIONAL**	The relationships for that entity. The default is CHILD.
SEARCHTERM	**OPTIONAL**	Search for all paths that contain this string.
SEARCHONTOLOGYTYPE	**OPTIONAL**	Search for all paths that have an ontology type of this. Must be used with SEARCHONTOLOGYTERM.
SEARCHONTOLOGYTERM	**OPTIONAL**	Search for all paths that have an ontology term of this. Must be used with SEARCHONTOLOGYTYPE
[...ENTITIES...]		

Query Service

Start a Query

GET /rest/v1/*queryService/startQuery*

{
   “cid” : ***...CID...***
}

Save Query

GET /rest/v1/*queryService/saveQuery*		
cid	**REQUIRED**	Query Identifier

{
   “queryId” : ***...QUERYID...***
}		

Load Query

GET /rest/v1/*queryService/loadQuery*		
queryId	**REQUIRED**	Query Identifier

{
   “cid” : ***...CID...***
}		

Add/Update a Clause

POST /rest/v1/*queryService/clause*		
cid	**REQUIRED**	Query Identifier
PAYLOAD	JSON Object for a Select, Where, or Join Clause (See JSON Object Section)	

{
   “cid” : ***...CID...***
}		

Add/Update a Where Clause

GET /rest/v1/*queryService/clause?type=where*		
cid	**REQUIRED**	Query Identifier
path	**REQUIRED**	Path the clause is running on
predicate	**REQUIRED**	Type of predicate
logicalOperator	**OPTIONAL**	Type of logical operator to use
clauseId	**OPTIONAL**	Replaces a given clause
data-*	**OPTIONAL**	A series of fields for the where clause
{ "clauseId" : ...clauseID... }		

Add/Update a Select Clause

GET /rest/v1/*queryService/clause?type=select*		
cid	**REQUIRED**	Query Identifier
clauseId	**OPTIONAL**	Replaces a given clause
path	**REQUIRED**	Path the clause is to return
alias	**OPTIONAL**	Alias of the clause
{ "clauseId" : ...clauseID... }		

Add/Update a Join Clause

GET /rest/v1/*queryService/clause?type=join*		
cid	**REQUIRED**	Query Identifier
clauseId	**OPTIONAL**	Replaces a given clause
joinType	**REQUIRED**	Join to run
path	**REQUIRED**	Path the clause is running on
data-*	**OPTIONAL**	A series of fields for the join clause
{ "clauseId" : ...clauseID... }		

Add/Update a Sort Clause

GET /rest/v1/*queryService/clause?type=sort*		
cid	**REQUIRED**	Query Identifier
clauseId	**OPTIONAL**	Replaces a given clause
sortType	**REQUIRED**	Sort to run
path	**REQUIRED**	Path the clause is running on
data-*	**OPTIONAL**	A series of fields for the sort clause

```
{
   "clauseId" : ...clauseID...
}
```

Run a Query

GET /rest/v1/*queryService/runQuery*		
cid	**REQUIRED**	Query Identifier

```
{
   "resultId" : ...resultId...
}
```

Run a Query

POST /rest/v1/*queryService/runQuery*	
PAYLOAD	JSON Object for a Query (See JSON Object Section)

```
{
   "resultId" : ...resultId...
}
```

Process Service

Start a Process

```
GET /rest/v1/processService/startProcess
{
   "cid" : ...CID...
}
```

Update a Process

GET /rest/v1/processService/process		
cid	**REQUIRED**	Process Identifier
resource	**REQUIRED**	Resource Identifier
processName	**REQUIRED**	Name of the process
data-*	**OPTIONAL**	A series of parameters that are prepended with the field name and are equal to the user supplied field value.

```
{
   "clauseId" : ...clauseID...
}
```

Run a Process

GET /rest/v1/processService/runProcess		
cid	**REQUIRED**	Process Identifier

```
{
   "resultId" : ...resultId...
}
```

Results Service

Available Results

GET /rest/v1/resultService/available		
resultId	**REQUIRED**	Result Id

```
[
   {
      “resultId” : ...RESULT ID...,
      “status” : ...STATUS...
   }...
]
```

Result Status

GET /rest/v1/resultService/resultStatus/RESULTID		
RESULTID	**REQUIRED**	Result Id

```
{
   “status” : ...STATUS...
}
```

Formats Available

GET /rest/v1/resultService/availableFormats/RESULTID		
RESULTID	**REQUIRED**	Result Id

```
[...FORMAT...]
```

Download a result

GET /rest/v1/resultService/result/RESULTID/FORMAT?download=DOWNLOAD		
RESULTID	**REQUIRED**	Result Id
FORMAT	**REQUIRED**	Format type
DOWNLOAD	**OPTIONAL**	Yes, or No to set the content-disposition on the response header as an attachment or not. Downloaded files are named IRCT-<RESULTID>.<FORMAT>
Result		

Data Types

Boolean

```
{
   “name” : “boolean”,
   “pattern” : “^(true|false)$”,
   “description” : “A return can either be true or false”
}
```

Byte

```
{
   “name” : “byte”,
   “pattern” : “^[A-z0-9]{1,1}$”,
   “description” : “A single”
}
```

Date

```
{
   “name” : “date”,
   “pattern” : “^\d{4}\-(0?[1-9]|1[012])\-(0?[1-9]|[12][0-
9]|3[01])$”,
   “description” : “yyyy-mm-dd format for dates”
}
```

Date Time

```
{
   “name” : “dateTime”,
   “pattern” : “^(\d{4})-(\d{2})-(\d{2})
(\d{2}):(\d{2}):(\d{2})$”,
   “description” : “yyyy-mm-dd hh:mm:ss format for date and time.
With hours in the 24 hour format”
}
```

Time

```
{
   “name” : “time”,
   “pattern” : “^(\d{2}):(\d{2}):(\d{2})$”,
   “description” : “hh:mm:ss format for date and time. With hours
in the 24 hour format”
}
```

Double

```
{
   “name” : “double”,
   “pattern” : “^[0-9]{1,13}(\\.[0-9]*)$”,
   “description” : “A double value”
}
```

Float

```
{
   “name” : “float”,
   “pattern” : “^([+-]?\\d*\\.?\\d*)$”,
   “description” : “A float value”
}
```

Integer

```
{
   “name” : “integer”,
   “pattern” : “^\\d+$”,
   “description” : “An integer value”
}
```

Long

```
{
   “name” : “long”,
   “pattern” : “^-?\\d{1,19}$”,
   “description” : “A long value”
}
```

String

```
{
   “name” : “string”,
   “pattern” : “^.*$”,
   “description” : “A string value”
}
```

Result Set

```
{
   “name” : “resultSet”,
   “pattern” : “^-?\\d{1,19}$”,
   “description” : “A resultset identifier”
}
```

Column

```
{
   “name” : “column”,
   “pattern” : “^.*$”,
   “description” : “A column identifier”
}
```

SubQuery

```
{
   “name” : “subQuery”,
   “pattern” : “^.*$”,
   “description” : “An IRCT SubQuery”
}
```

Array

```
{
   “name” : “array”,
   “pattern” : “^.*$”,
   “description” : “An array”
}
```

JSON Schema

Resource

```
{
   "title" : "Resource",
   "type" : "object",
   "properties" : {
      "id" : {
         "type" : "long",
         "description" : "Unique identifier for the resource",
         "minimum" : 0
      },
      "name" : {
         "type" : "string",
         "description" : "Resource name"
      },
      "ontologyType" : {
         "enum" : [ "TREE", "FLAT", "GRAPH" ],
         "description" : "Resource ontology type"
      },
      "implementation" : {
         "$ref" : "#/definitions/implementation",
         "description" : "The implementing resource interface
type"
      },
      "dataTypes" : {
         "type" : "array",
         "items" : {
            "$ref" : "url://dataType.json#DataType"
         },
         "description" : "An array of dataTypes that are
supported by this resource"
      },
      "relationships" : {
         "type" : "array",
         "items" : {
            "type" : "string"
         },
         "description" : "An array of relationships supported by
the resource"
      },
      "logicalOperators" : {
         "type" : "array",
         "items" : {
             "enum": [ "AND", "OR", "NOT" ]
         },
         "description" : "The supported logical operators for
this resource"
      },
      "predicates" : {
         "type" : "array",
         "items" : {
             "$ref" : "#/definitions/predicate"
          },
         "description" : "Supported predicates for this resource"
      },
      "joins" : {
         "type" : "array",
         "items" : {
            "$ref" : "url://join.json#Join",
         }
      },
      "processes" : {
         "type" : "array",
         "items" : {
            "$ref" : "#/definitions/process"
         },
         "description" : "Supported processes for this resource"
      },
      "visualizations" : {
         "type" : "array",
         "items" : {
            "$ref" : "#/definitions/visualizations"
         },
         "description" : "Supported processes for this resource"
      }
   },
   "required" : [ "id", "name", "ontologyType" ],
   "definitions" : {
      "implementation" : {
         "properties" : {
           "type" : {
              "type" : "string",
              "description" : "Type of resource it is"
           },
           "returns" : {
              "type" : "array",
              "items" : {
                  "$ref" : "#definitions/field"
               },
               "description" : "An array of elements that are by
default returned"
           },
           "editableReturn" : {
              "type" : "boolean",
               "description" : "True if the returned entities can
be edited"
           }
         },
         "required" : [ "type" ],
         "additionalProperties" : false
      },
      "predicate" : {
         "properties" : {
            "name" : {
               "type" : "string",
               "description" : "Name of the predicate as
described by the resource"
            },
            "displayName" : {
               "type" : "string",
               "description" : "Name to display for the
predicate"
            },
            "description" : {
               "type" : "string",
               "description" : "The description of the predicate"
            },
            "default" : {
                "type" : "boolean",
                "description" : "Is this the default predicate"
            },
            "fields" : {
               "type" : "array",
              "items" : {
                 "$ref" : "#/definitions/field"
              },
               "description" : "An array of fields that are used
to form the predicate"
            },
            "dataTypes" : {
               "type" : "array",
               "items" : {
                  "$ref" : "url://dataType.json#DataType",
               },
               "description" : "A list of dataTypes that allow
the predicate to be run on those paths"
            },
            "paths" : {
               "type" : "array",
               "items" : {
                  "type" : "string"
               },
               "description" : "A list of REGEX patterns that
allow the predicate to be run on those paths"
            }
         },
         "required" : ["predicateName", "fields"],
         "additionalProperties" : false
      },
      "process" : {
         "properties" : {
            "name" : {
               "type" : "string",
               "description" : "Name of the process as described
by the resource"
```

```
            },
            "displayName" : {
               "type" : "string",
               "description" : "Name to display for the process"
            },
            "description" : {
               "type" : "string",
               "description" : "The description of the process"
            },
            "fields" : {
               "type" : "array",
               "items" : {
                  "$ref" : "#/definitions/field"
               },
               "description" : "An array of fields that are used
to form the process"
            },
            "returns" : {
               "type" : "array",
               "items" : {
                  "$ref" : "#definitions/field"
               }
            }
         },
         "required" : ["processName", "fields"],
         "additionalProperties" : false

      },
      "visualization" : {
         "properties" : {
            "name" : {
               "type" : "string",
               "description" : "Name of the visualization as
described by the resource"
            },
            "displayName" : {
               "type" : "string",
               "description" : "Name to display for the
visualization"
            },
            "description" : {
               "type" : "string",
               "description" : "The description of the
visualization"
            },
            "fields" : {
               "type" : "array",
               "items" : {
                  "$ref" : "#/definitions/field"
               },
               "description" : "An array of fields that are used
to form the process"
            },
            "returns" : {
               "enum" : [ "image", "html", "video" ]
            }
         }
      },
      "field" : {
         "properties" : {
            "name" : {
               "type" : "string",
               "description" : "The name of the field",
            },
            "path" : {
               "type" : "string",
               "description" : "The name of the field as
described by the resource",
            },
            "description" : {
               "type" : "string",
               "description" : "The description of the field"
            },
            "dataType" : {
               "type" : "url://dataType.json#DataType",
               "description" : "An array of dataTypes that are
supported by the field",
            },
            "permittedValues"  : {
               "type" : "array",
               "items" : {
                  "type" : "string"
               },
               "description" : "An array of values that are
permitted for this field"
            },
            "relationship" : {
               "type" : "string",
               "description" : "Used to return a list of
associated entities that can have that type of relationship to
the selected entity",
            },
            "required" : {
               "type" : "boolean",
               "description" : "Is this field requried"
            }
         },
         "required" : ["name", "path"],
         "additionalProperties" : false
      }

   }
}
```

Entity

```
{
   "title" : "Entity",
   "id" : "Entity",
   "type" : "object",
   "required" : [ "pui", "dataType"],
   "properties" : {
      "pui" : {
           "type" : "string",
           "description" : "The unique path to the entity."
      },
      "name" : {
           "type" : "string",
           "description" : "The name of the entity"
      },
      "displayName" : {
           "type" : "string",
           "description" : "The name of the entity to display"

      },
      "description" : {
           "type" : "string",
           "description" : "The description of the entity"
      },
      "ontology" : {
           "type" : "string",
           "description" : "Name of the ontology."
      },
      "ontologyId" : {
           "type" : "string",
           "description" : "Id of this entity in the ontology"
      },
      "dataType" : {
           "type" : "string",
           "description" : "Type of dataType as described by the
resource, or the IRCT of the entity."
      },
      "relationships" : {
           "type" : "array",
           "items" : {
           "type" : "string"
         },
           "description" : "An array of relationships that are
supported by this entity"
      },
      "count" : {
           "type" : "object",
           "patternProperties": {
           ".{1,}" : { "type": "string" }
           },
           "description" : "A map of counts available for that
entity"
```

```
      },
      "attributes" : {
           "type" : "object",
           "patternProperties": {
           ".{1,}" : { "type": "string" }
           },
           "description" : "A map of resource supplied
attributes"
      }
   }
}
```

DataType

```
{
   "title" : "DataType",
   "id" : "DataType",
   "type" : "object",
   "required" : [ "name" ],
   "properties" : {
      "name" : {
         "type" : "string",
         "description" : "The name of the dataType"
      },
      "pattern" : {
         "type" : "string",
         "description" : "The pattern of this dataType"
      },
      "description" : {
         "type" : "string",
         "description" : "The description of the dataType"
      },
      "typeOf" : {
         "type" : "string",
         "description" : "The parent type of this dataType"
      }
   }
}
```

Query

```
{
   "title" : "Query",
   "id" : "query",
   "type" : "object",
   "properties" : {
      "select" : {
         "type" : "array",
         "items" : {
            "$ref" : "#/definitions/selectClause"
         }
      },
      "where" : {
         "type" : "array",
         "items" : {
            "$ref" : "#/definitions/whereClause"
         }

      },
      "join" : {
         "type" : "array",
         "items" : {
            "$ref" : "#/definitions/joinClause"
         }
      },
      "sort" : {
         "type" : "array",
         "items" : {
            "$ref" : "#/definitions/sortClause"
         }
      },
      "subquery" : {
         "type" : "array",
         "items" : {
            "$ref" : "#/definitions/query"
         }
      }
   }
}
```

Where Clause

```
{
   "title" : "Where Clause",
   "id" : "whereClause",
   "type" : "object",
   "properties" : {
      “type” : {
        "enum" : [ "where" ],
        "description" : "Field must be set if being used outside
of a Query object"
      },
      "field" : {
         "$ref" : "#/definitions/field"
         "description" : "Field the predicate is to be run
against"
      },
      "predicate" : {
         "type" : "string",
         "description" : "The predicate name"
      },
      "fields" : {
         "type" : "array",
         "items" : {
            "$ref" : "#/definitions/objectMap"
         }
         "description" : "A Map of String, String that has the
field name as the key and the value as the value of that field
for that property"
      }
   }
}
```

Select Clause

```
{
   "title" : "Select Clause",
   "id" : "selectClause",
   "type" : "object",
   "properties" : {
      “type” : {
        "enum" : [ "select" ],
        "description" : "Field must be set if being used outside
of a Query object"
      },
      "field" : {
         "$ref" : "#/definitions/field"
         "description" : "Field the predicate is to be run
against"
      },
      "alias" : {
         "type" : "string",
         "description" : "Name of the data field to use in
results"
      },
      "operation" : {
         "type" : "string",
         "description" : "Name of the operation to use in the
query"
      },
      "fields" : {
         "type" : "array",
         "items" : {
            "$ref" : "#/definitions/objectMap"
         }
         "description" : "A Map of String, Object that has the
field name as the key and the value as the value of that field
for that property"
      }
   }
}
```

Join Clause

```
{
   "title" : "Join Clause",
   "id" : "joinClause",
   "type" : "object",
   "properties" : {
      “type” : {
        "enum" : [ "join" ],
        "description" : "Field must be set if being used outside
of a Query object"
      },
      "joinType" : {
         "type" : "string",
         "description" : "The join name"
      },
      "fields" : {
         "type" : "array",
         "items" : {
            "$ref" : "#/definitions/objectMap"
         }
         "description" : "A Map of String, Object that has the
field name as the key and the value as the value of that field
for that property"
      }
   }
}
```

Sort Clause

```
{
   "title" : "Sort Clause",
   "id" : "sortClause",
   "type" : "object",
   "properties" : {
      “type” : {
        "enum" : [ "sort" ],
        "description" : "Field must be set if being used outside
of a Query object"
      },
      "sortType" : {
         "type" : "string",
         "description" : "The sort name"
      },
      "fields" : {
         "type" : "array",
         "items" : {
            "$ref" : "#/definitions/objectMap"
         }
         "description" : "A Map of String, Object that has the
field name as the key and the value as the value of that field
for that property"
      }
   }
}
```

Process

```
{
      "title" : "Process",
      "id" : "process",
      "required" : [ "resource", "name" ],
      "properties" : {
         "resource" : {
            "type" : "string",
            "description" : "The name of the resource to run this 
process"
         },
         "name" : {
            "type" : "string",
            "description" : "The name of the process to be run on 
the resource"
         },
         "fields" : {
          "type" : "array",
          "items" : {
             "$ref" : "#/definitions/objectMap"
          }
          "description" : "A Map of String, String that has the 
field name as the key and the value as the value of that field 
for that property"
       }
      }
}
```

Map<String, String>

```
{
  "title": "Map<String,String>",
  "id" : "stringMap",
  "type": "object",
  "patternProperties": {
    ".{1,}": { "type": "string" }
  }
}
```

Map<String, Object>

```
{
  "title": "Map<String,Object>",
  "id" : "stringMap",
  "type": "object",
  "patternProperties": {
    ".{1,}": { "type": "object" }
  }
}
```